Reading microgpt (for high school and first-year university students)

A look inside how AI writes text, through a 200-line program

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

Ver. 0.1 October 4, 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). The microgpt program is the work of Andrej Karpathy.

Introduction

> AI systems such as ChatGPT and Claude (large language models, or LLMs) actually repeat a very simple task over and over: **predicting the next word as a continuation of the text so far**. Once a word is predicted, it is added to the text, and the next one is predicted. Repeating this produces a long piece of writing.

microgpt is a Python program of only about 200 lines, released by Andrej Karpathy in February 2026. It runs the same mechanism as an LLM, on a very small scale. Instead of sentences it handles English personal names: it learns to predict the "next **letter**" and, at the end, invents new names. It uses no special libraries such as NumPy or PyTorch, only the features built into Python.

This booklet explains what the program does, using **words and figures** as much as possible. The formulas are collected in Appendix A, so the main text can be understood without reading any of them. All the numbers in the text were confirmed by actually running the program.

To understand the content, we recommend watching, alongside this booklet, the video series by 3Blue1BrownJapan (YouTube; easy to follow, with CG graphics), which carefully explains everything from machine learning to LLMs, and the AI lecture series by Takumi of Yobinori (YouTube; detailed on the mathematics).

- Program: https://gist.github.com/karpathy/8627fe009c40f57531cb18360106ce95
- Explanation by the author: https://karpathy.github.io/2026/02/12/microgpt/

How to run it: If Python 3 is installed, type `python microgpt.py` in a terminal (the Command Prompt on Windows). On the first run only, it downloads the training data of personal names (`input.txt`) from the Internet and saves it in the same folder. On our computers, training plus name generation took about 2–3 minutes.

In the text, "line ◯" refers to a line number in this program (200 lines); see Appendix E for where to get the program and what each range of lines does. It is easier to follow if you have the program open alongside.

1 Overview

The program consists of the following six parts, in order from the top (Figure 1).

Part	Lines	Contents
Data	14–21	Reads 32,033 names and shuffles their order
Tokenizer	23–27	Makes the rule for converting letters to numbers
Autograd	29–72	Automatically works out "which knob to turn, and which way"
Parameters and model	74–144	The 4,192 knobs (parameters) and the model that makes the predictions
Training	146–184	Measures how wrong the predictions are and adjusts the knobs little by little
Generation	186–200	Makes new names with the trained model

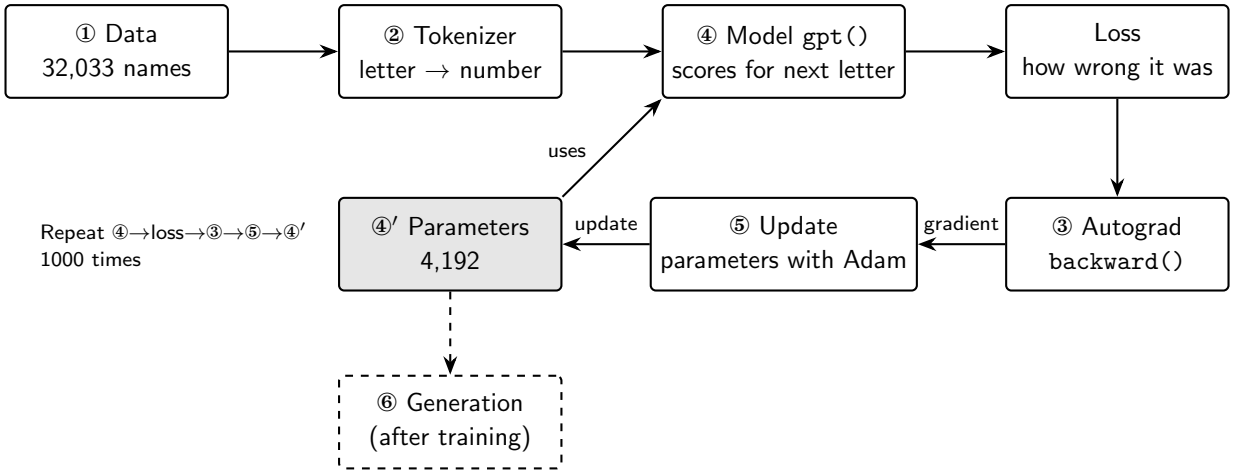


Figure 1 The structure of microgpt. In training, the flow ④ → loss → ③ → ⑤ is repeated 1000 times, changing the parameters little by little. After training, ⑥ generation is done with the same model and parameters.

The comment at the top of the program says, in effect, that "this file is the complete algorithm, and everything else is just efficiency." The claim is that the differences from real LLMs lie mainly in **scale and speed**, and that **the content of the computation** is the same as in these 200 lines (we compare them in Section 8).

2 Data and letter numbers

2.1 Data

The training data are 32,033 American personal names (`emma`, `olivia`, etc., one per line). **Each name is treated as one "document".** Just as an LLM learns the "next word" from a large amount of text, microgpt learns the "next letter" from a large number of names.

2.2 Giving letters numbers

A computer cannot calculate with letters directly, so each letter is given a number (lines 24–26). The 26 lowercase letters a–z that appear in the data get the numbers 0–25, and **a special symbol, BOS** (Beginning of Sequence), **that marks the beginning and end of a name** (number 26) is added. That makes 27 kinds in total. These 27 kinds are called the model's **vocabulary**.

For example, during training the name emma is converted into the following sequence of numbers (line 157).

$$\text{BOS},\ \texttt{e},\ \texttt{m},\ \texttt{m},\ \texttt{a},\ \text{BOS} \quad \longrightarrow \quad 26,\ 4,\ 12,\ 12,\ 0,\ 26$$

The first BOS is the signal "a name starts here," and the last BOS is the correct answer "the name ends here." The model solves five prediction problems: "after BOS comes e," "after BOS, e comes m," ..., "after BOS, e, m, m, a comes BOS (the end)."

Each small piece that a text is cut into in this way is called a **token**, and the rule for cutting text into tokens is called a **tokenizer**. In microgpt, **1 letter = 1 token**[1].

3 Inside the model: scoring the next letter

3.1 Processing one letter at a time

The function `gpt()` on lines 108–144 is the model itself. In **each call it receives only one letter** and returns a score for each of the 27 kinds as the "next letter." The processing proceeds in the order shown in Figure 2. We look at it one stage at a time below.

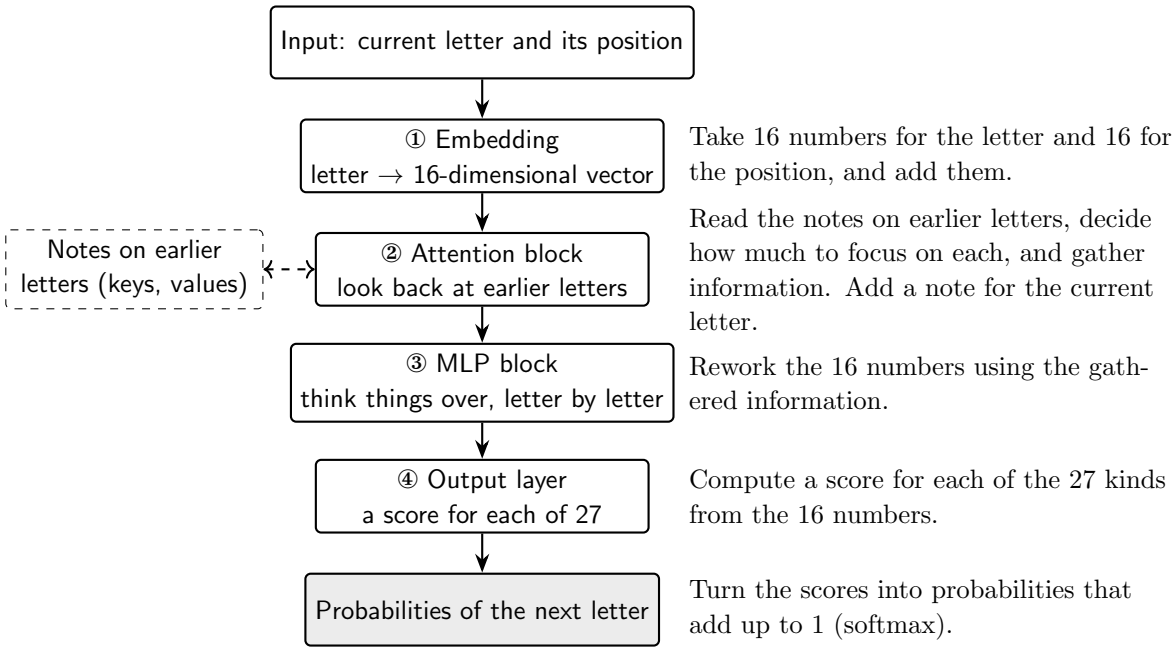


Figure 2 How `gpt()` processes one letter. Only the Attention block uses information about earlier letters (the notes).

3.2 Turning a letter into numbers (embedding)

First, each letter is replaced by a **list of 16 numbers** (a 16-dimensional **vector**)[2]. For this the program has two tables.

[1] Real LLMs use methods (such as BPE, byte pair encoding) that merge frequently occurring combinations of letters, such as "ing" or "the", into a single token. This lets the same text be represented with fewer tokens.

[2] The number 16 is a design value chosen by the author as a balance between expressive power and amount of computation (such values are called **hyperparameters**, to distinguish them from the parameters decided by training). There is no reason it must be 16, but it has to be divisible by the number of heads (4), explained later. Real LLMs use much larger values: 768 in GPT-2 and 12,288 in GPT-3.

- **The letter table** (named `wte` in the program): a table of 27 rows × 16 columns, in which each of the 27 kinds of letter is assigned one row of 16 numbers. For example, the letter e (number 4) corresponds to the 16 numbers in the 5th row.
- **The position table** (`wpe`): a table of 16 rows × 16 columns, in which each position, "which letter of the name this is," is assigned 16 numbers.

The row for the letter's number and the row for the position's number are taken out and added; the result is the starting point for that letter (lines 109–112). This process is called **embedding**[3].

The numbers in the tables start out as random values. As training proceeds, letters that are used in similar ways come to have similar lists of numbers (we will see a real example in Section 5.5). You can think of the 16 numbers as a "name tag describing the letter's features."

The position numbers are added because the Attention mechanism described next cannot, by itself, tell the order of the letters[4].

3.3 Looking back at earlier letters (Attention)

To guess the next letter of a name, the model needs to look not only at the current letter but also at the letters before it. Consider, for example, only the four names emma, ava, mia and anna. When the current letter is a, the next letter changes as follows.

- If the name has just started with a (the first a of ava or anna), the next letter is v or n.
- If the a comes after m, i (the last a of mia), the next is the end (BOS).

Even with the same current letter a, the answer changes depending on what came before. This "looking back" is the job of the **Attention** mechanism (lines 115–134)[5].

In Attention, three kinds of number lists (vectors) are made from the 16 numbers of each letter (lines 118–120).

- **Query** (question): what kind of information the current letter is looking for
- **Key** (heading): a label for what kind of information a letter holds
- **Value** (content): the information a letter actually passes on

The query of the current letter is compared with the key of each earlier letter, and the better they match, the larger the weight (**attention weight**, degree of attention) given. The weights add up to 1. The values of the letters, mixed together using these weights, are the result of looking back. Picture a library: you compare your question (query) with the headings of the books (keys), read the books that match more closely, and gather their contents (values).

The notes grow one at a time The important part here is the following two lines (lines 121–122).

```
keys[li].append(k)
values[li].append(v)
```

Letters are processed one at a time in order, and each time the key and value of the current letter are appended to the end of the notes (Figure 3). For example, when the second m of emma is being processed, the notes contain only BOS, e, m and the current m itself; the following a has

[3] `wte` stands for word token embedding and `wpe` for word position embedding; the names follow the conventions of the GPT-2 program.

[4] Attention is a calculation that surveys all the earlier letters as if they were "in a bag," so the order has to be supplied separately. In microgpt, the numbers in the position table are also parameters decided by training, as in GPT-2.

[5] "Looking back" is the wording used in this booklet. General explanations of LLMs call this mechanism attention, and describe it as deciding "which tokens to attend to (pay attention to)." Looking only at earlier tokens and never at future ones is called being **causal**.

not been processed yet, so it is not in the notes. The current letter can only look back at letters in the notes, so **future letters cannot be seen, even if the model tried**[6].

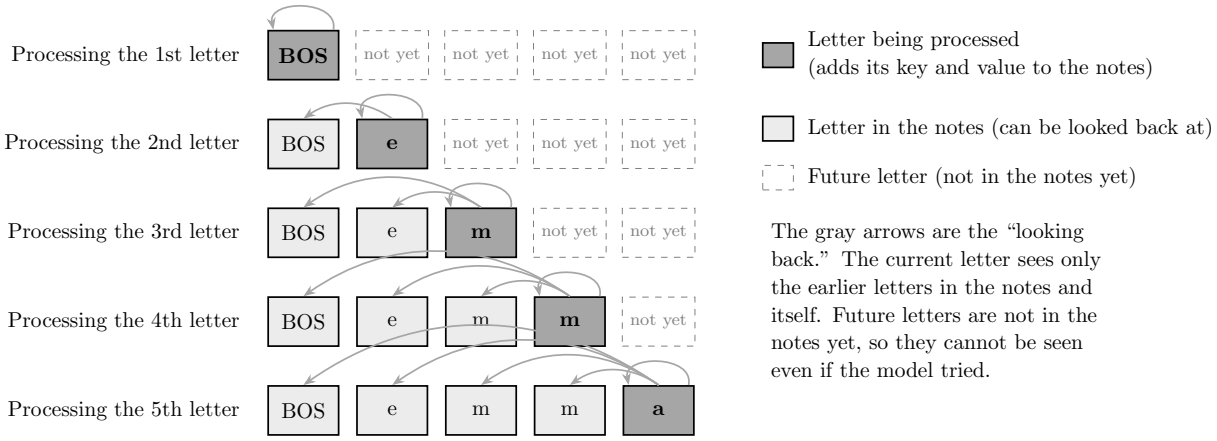


Figure 3 How the notes (keys and values) grow one at a time while emma is processed.

> Not looking at the future is very important. During training the whole name is known, but if the next letter were visible, the model could get the right answer just by copying it, and would learn nothing. When making names (Section 7), the future letters do not exist yet, so the model has to practice under the same conditions during training.

Four heads look back separately A single look back can decide only one way of weighting "how much to focus on which letter." But to guess the next letter, we may want to look at the earlier letters from several points of view, such as "what is the letter just before?" and "what is the first letter of the name?"

So microgpt splits the 16 numbers into four sets of 4, and **four groups (heads) each look back separately** (Figure 4, lines 124–132). It is like four people reading the same notes from different points of view. Each head decides its weights from its own point of view and produces 4 numbers. Finally the four results are joined back into 16 numbers and transformed once more to mix them (line 133). This mechanism is called **multi-head attention**.

Which point of view each head actually takes on is not decided by humans; it emerges naturally through training. In Figure 15 of Section 7.3, you can see the four heads weighting the same emma in different ways.

3.4 Thinking things over, letter by letter (MLP)

What MLP means MLP stands for **Multi-Layer Perceptron**. The perceptron is a calculating mechanism devised by Rosenblatt in 1958, modeled on the nerve cells (**neurons**) of the brain. It multiplies several inputs by their own weights, adds them up, and decides whether to "respond or not" according to the size of the total. Stacking several stages of this gives a "multi-layer" perceptron, the most basic form of a **neural network**. In explanations of the Transformer, the same thing is also called a **Feed-Forward Network** (FFN).

[6] An ordinary Transformer, which computes all the tokens of a sentence at once, has to hide future tokens with a process called a **causal mask** so they are not seen. microgpt processes one letter at a time, so no mask is needed. Also, the mechanism of keeping the past keys and values and adding only those of the new token is the same as the **KV cache** that real LLMs use to speed up text generation.

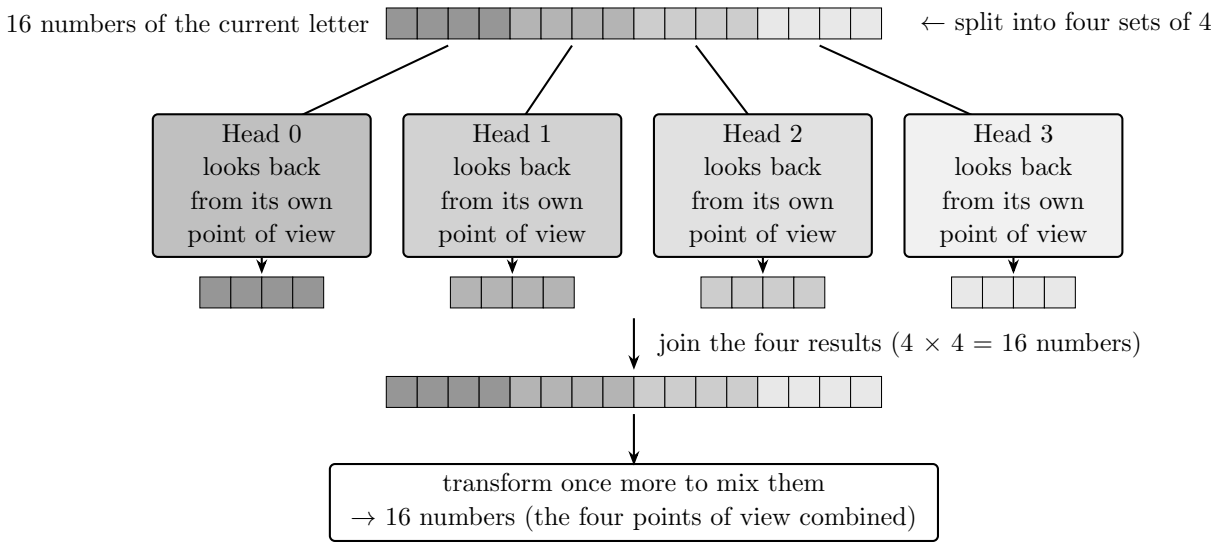


Figure 4 Multi-head attention. The 16 numbers are split among four heads, each looks back separately, and the results are joined again.

What the MLP does The MLP is the following three-stage calculation (lines 137–140). Figure 5 shows the actual values when emma is fed into the trained model and the final a is processed.

1. **First transformation (16 → 64)**: 64 "detectors" each look at the 16 numbers (the 16-dimensional vector) and compute how strongly they respond. Each detector has different weights, so each responds to a different pattern in the input.
2. **ReLU** (Rectified Linear Unit): a negative response is set to 0. In other words, detectors that do not match their pattern "do not respond." In the example in the figure, only 9 of the 64 responded. Checking 300 names, about 8 respond per letter on average. Which detectors respond depends on the letter and its context, and every one of the 64 responded somewhere.
3. **Second transformation (64 → 16)**: each detector that responded writes its own fixed piece of information into the 16 numbers.

Finally, these 16 numbers are added back to the 16 numbers at the entrance of the MLP (line 141). This mechanism, for "keeping the original information while writing in what was newly learned," is called a **residual connection**. The same is done at the exit of the Attention block (line 134). Also, at the entrance of Attention and of the MLP, the size of the 16 numbers is evened out (**normalization**). Both make training more stable.

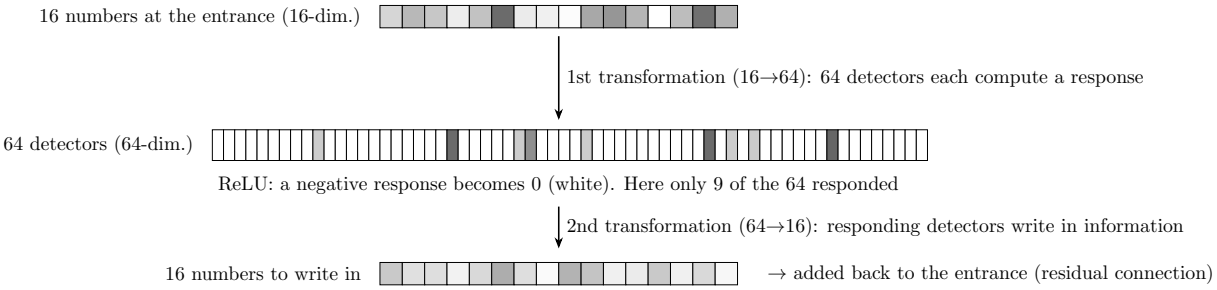


Figure 5 The calculation inside the MLP (actual values when emma is fed into the trained model and the final a is processed). The shade shows the size (absolute value) of each number.

Division of roles with Attention Attention reads the notes on earlier letters and gathers information: it is the **exchange of information between letters**. The MLP, on the other hand, is a calculation that **thinks things over within one letter** using the gathered information; it does not look at other letters. The same MLP is used for every letter. In terms of a meeting, Attention is "listening to other people's opinions," and the MLP is "organizing your own thoughts based on what you heard."

Why expand to 64? The ReLU in the middle is what matters. Without ReLU, the two transformations could be combined into a single "16 → 16 transformation," and expanding would be pointless. Because of ReLU, each of the 64 works as a "pattern detector," and the more detectors there are, the more patterns can be told apart. In exchange, the number of parameters and the amount of computation grow: in microgpt, half of the 4,192 parameters (2,048) are in the MLP. The factor of 4 (16 → 64) is not mathematically determined; it is a rule of thumb inherited from the original Transformer and GPT-2 (GPT-2 also uses 768 → 3072). The result is brought back to 16 at the end because it has to be the same size to be added back to the 16 numbers at the entrance (residual connection)[7].

3.5 From scores to probabilities (softmax)

Finally, a score for each of the 27 kinds (a 27-dimensional vector) is computed from the 16 numbers (line 143). These scores are also called **logits**. The scores can be negative and have no rule about their total, so they cannot be used as probabilities as they are. A calculation called **softmax** turns them into probabilities that are "all 0 or more and add up to exactly 1" (Figure 6). The order of the scores is kept, and differences in score become larger differences in probability. The resulting probabilities of the 27 kinds are the "prediction of the next letter" (a **probability distribution**). How softmax is calculated is explained in Appendix A.4.

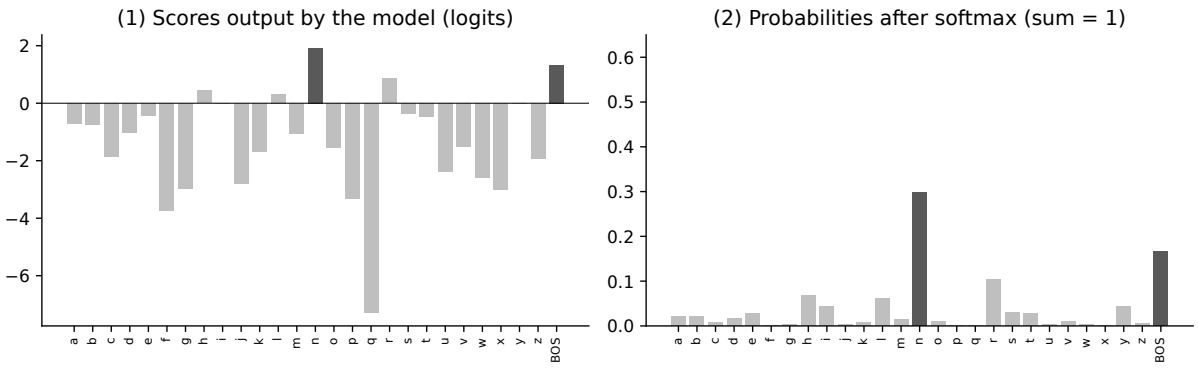


Figure 6 What softmax does (the letter after the final a, when emma is fed into the trained model). (1) The scores can be negative. (2) After softmax they are probabilities, all 0 or more and adding up to 1. Dark gray marks the top candidates, n and BOS (end).

[7]For large LLMs, there is research suggesting that factual knowledge such as "Paris is the capital of France" is stored in the MLP parts (Geva et al., 2021). Chapter 7 of 3Blue1Brown's video series on deep learning also deals with this topic. The fc in the program's variable names `mlp_fc1` and `mlp_fc2` stands for fully connected, meaning a transformation in which every input number is connected to every output number.

4 How training works

4.1 Measuring wrongness as "surprise" (loss)

To measure how good a prediction is, microgpt looks at **what probability it gave to the correct letter**. A high probability on the correct answer is a good prediction; a low one is a bad prediction.

The single number that expresses this is the **loss**. The loss is determined from the probability p given to the correct answer as in Figure 7. If $p = 1$ (the model was sure of the correct answer), the loss is 0; the smaller p, the larger the loss. It helps to think of it as "the size of the surprise on learning the correct answer." In one training step, one name is taken and the loss at each of its letters is averaged (lines 156–169). This loss is called the **cross-entropy loss**.

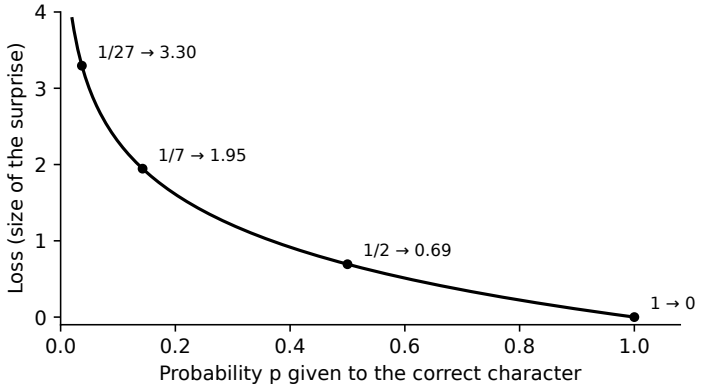


Figure 7 The relation between the probability given to the correct letter and the loss. A random guess that gives the same probability (1/27) to all 27 kinds has a loss of about 3.30.

Before training, the model's insides are random, so it gives almost the same probability to all 27 kinds. Then the probability of the correct letter is 1/27. Reading this point on the curve in Figure 7, the loss is about 3.30 (the calculation is in Appendix A.5). In fact, the loss at the first step was 3.37. The goal of **training** is to lower this loss.

4.2 4,192 knobs (parameters)

The model's predictions are decided by the numbers in the embedding tables and the numbers used for the transformations in Attention and the MLP. Together these numbers are called **parameters**. The parameters are the entries of tables of numbers arranged in rows and columns, that is, of **matrices**. microgpt has the following matrices, and counting all their entries gives 4,192. Before training, they all start as small random numbers (roughly -0.16 to $+0.16$) (line 80).

Name	Matrix size	Entries	Role
`wte`	27 rows × 16 columns	432	16 numbers for each letter (embedding)
`wpe`	16 rows × 16 columns	256	16 numbers for each position (position embedding)
`attn_wq, wk, wv, wo`	four of 16 × 16	1,024	transformations in Attention (making queries, keys, values; combining results)
`mlp_fc1`	64 rows × 16 columns	1,024	1st stage of the MLP (16→64)
`mlp_fc2`	16 rows × 64 columns	1,024	2nd stage of the MLP (64→16)
`lm_head`	27 rows × 16 columns	432	output layer (16 numbers → scores of 27 kinds)
Total		**4,192**	

On the other hand, the numbers that flow through the calculation (the 16 numbers of each letter, the queries, keys and values, the 64 numbers in the middle of the MLP, the 27 scores) are all **vectors**, lists of numbers in a single column. Matrices and vectors meet in the multiplication "matrix × vector" (lines 94–95). As in Figure 8, the inner product of one row of the matrix with the vector (the sum of the products of the components) becomes one number of the output. There are as many output numbers as the matrix has rows, so a 16 × 16 matrix turns a 16-dimensional vector into a 16-dimensional one, and a 64 × 16 matrix into a 64-dimensional one[8].

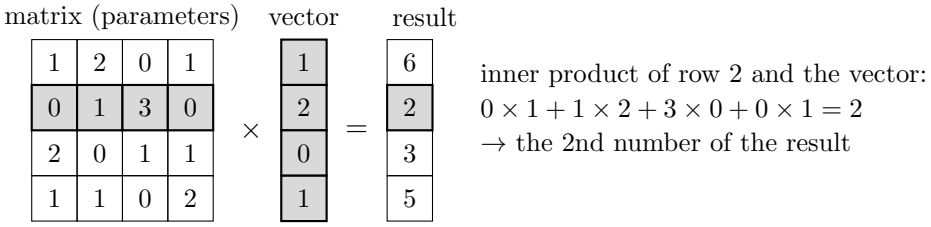


Figure 8 An example of matrix × vector (4 rows × 4 columns). The inner product of each row with the vector gives each component of the result. In microgpt, the entries of the matrices are the parameters (knobs).

> Parameters are like the **knobs** used to tune a radio (Figure 9). Training is the job of turning the 4,192 knobs little by little to find the positions where the loss becomes small.

4.3 Which knob to turn, and which way (gradients and autograd)

Turning each of the 4,192 knobs as a test and checking how the loss changes would take far too long. Instead, the program computes for all the knobs at once: "if this knob is increased slightly, how much does the loss go up (or down)?" This value is called the **gradient**. Decreasing the knobs with a positive gradient and increasing those with a negative gradient lowers the loss.

The gradient is computed by **automatic differentiation** (autograd, lines 29–72). Every calculation in microgpt is a combination of simple operations such as addition and multiplication.

[8] On line 89, the 4,192 parameters are rearranged into a single list called `params`. This is only for convenience, so that Adam can update the knobs one by one (line 176); in the calculation itself they are used as matrices. Also, real LLMs compute the tokens of a text all at once, so the numbers that flow through are also matrices of "number of tokens × dimension."

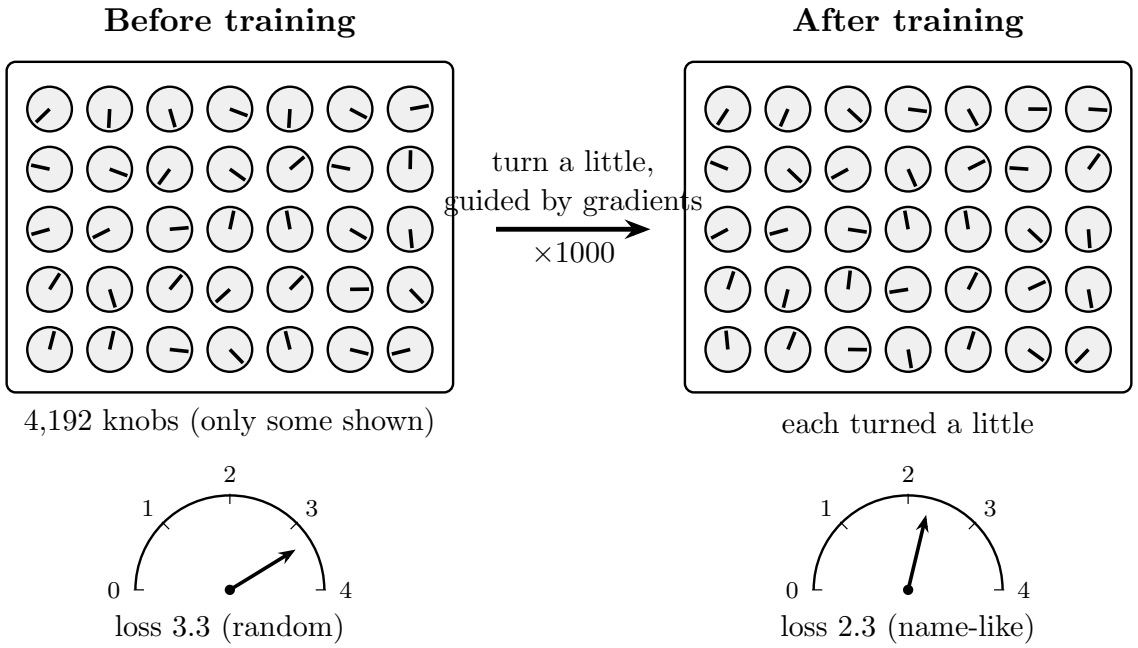


Figure 9 An illustration likening the parameters to "knobs." One knob corresponds to one entry of a matrix. Turning the knobs little by little in training brings the loss meter down (the meter values are those of the actual training).

Each time the program calculates something, it records "from which numbers, and by which operation, this was made." This builds up a **computation graph** that shows how the calculations are connected.

Let us look at a small example. With $a = 2$ and $b = 3$, we compute $f = a \times b + a$ (Figure 10).

In this way, autograd computes the gradients mechanically, tracing the computation graph back from the output side to the input side[9]. In one step of microgpt the computation graph becomes a large one made of tens of thousands of numbers, but what it does is the same as in this small example. A single call to `loss.backward()` on line 172 obtains the gradients of all 4,192 knobs.

4.4 A smarter way to turn the knobs (Adam)

Once the gradients are known, the knobs are turned (lines 174–182). The basic rule is "turn each knob a little in the direction opposite to its gradient" (**gradient descent**). microgpt uses an improved method called **Adam**. The name comes from **Adaptive Moment Estimation**; the method was published by Kingma and Ba in 2014[10]. Adam has two refinements.

- **Use an average of the gradients**: Since only one name is used per step, the gradients vary a lot from name to name. So an average of recent gradients (a **moving average**) is used to smooth out the variation.
- **Even out the turning amount**: The size of the gradient can differ by several orders of magnitude from knob to knob. So each knob's step is divided by the size of its gradient, so

[9] This method is called **backpropagation**, and it is the central calculation in training neural networks. It is the **chain rule** of differentiation applied step by step along the computation graph. For an explanation with formulas, see Appendix A.5.

[10] "Moment" is a statistical term: the first moment is the mean and the second moment is the mean of the squares. Adam estimates the mean of the gradients and the mean of the squared gradients as training proceeds, and adapts the step size of each knob automatically according to them. Since it is not an acronym of initial letters, it is written Adam rather than ADAM.

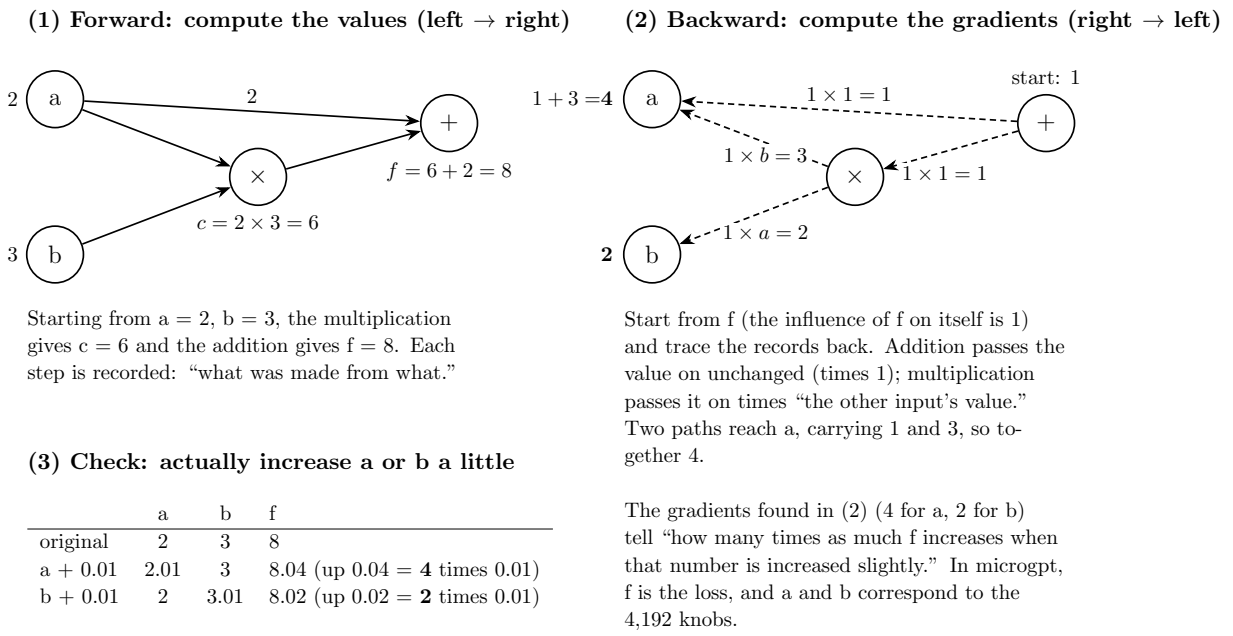


Figure 10 The computation graph of $f = a \times b + a$. (1) First compute the values from left to right. (2) Then compute, from right to left, "the size of the influence on f" (the gradient). Addition passes it on unchanged, multiplication multiplies by the other input's value, and if there are two paths they are added. (3) Actually increasing a or b a little increases f by the gradient times that amount.

that every knob moves by roughly the same amount.

The guide value for how far to turn in one step (the **learning rate**) starts at 0.01 and is made gradually smaller, down to 0, as training proceeds (line 175). An explanation with formulas is in Appendix A.6.

5 Following one step with a small data set

With 32,033 names and 4,192 parameters, it is hard to follow in numbers what happens in one step. So, without changing the training part of the program, we ran it with **only the data and the model made smaller**.

5.1 Settings

We replaced `input.txt` with just these four names:

`emma`, `ava`, `mia`, `anna`

The letters that appear are the six letters a, e, i, m, n, v, so with BOS the vocabulary has 7 kinds.

We did not write a new program. We only **changed the numbers on four lines** of the original microgpt.py; not a single character of the data loading, model, autograd, training or generation parts was changed.

Line	Original program	Small example
76	`n_embd = 16`	`n_embd = 4`
77	`block_size = 16`	`block_size = 5`
78	`n_head = 4`	`n_head = 1`
152	`num_steps = 1000`	`num_steps = 300`

With `input.txt` reduced to four lines and these four places changed, anyone can check the same results. The gradients and other values in the tables below were extracted by adding a few lines that save the numbers to a file. These lines only save, so they do not affect the calculation, and the generated names match those of the program with only the four lines changed (Appendix D). The original settings and the small example compare as follows.

	Original	Small example
Number of names	32,033	4
Vocabulary (kinds of letters + BOS)	27	7
Numbers per letter (vector dimension, `n_embd`)	16	4
Number of heads (`n_head`)	4	1
Maximum length (`block_size`)	16	5
Training steps (`num_steps`)	1,000	300
Number of parameters	4,192	268
Running time	2–3 minutes	about 2 seconds

5.2 Step 1: predictions and loss

After shuffling, the name used in step 1 was `mia`[11]. The model solves the following four prediction problems. The probabilities given to the correct answers were:

Input	Correct	Probability given	After 1 step
BOS	m	0.154	0.165
BOS m	i	0.116	0.123
BOS m i	a	0.171	0.184
BOS m i a	BOS (end)	0.131	0.140
Loss (average)		1.953	1.889

If the same probability is given to all 7 kinds, each gets $1/7 \approx 0.143$. The model before training is almost in this "random" state. The loss of 1.953 is also almost the same as the value for random guessing (the loss for 1/7, about 1.95).

5.3 Gradients and how the knobs move

`loss.backward()` gives the gradients of all 268 knobs. Let us look at a few examples.

Knob	Before	Gradient	After 1 step
`lm_head` (row of letter i, 3rd)	0.1090	+0.461	0.0990
`lm_head` (row of letter a, 2nd)	0.0940	−0.438	0.1040
`lm_head` (row of letter m, 4th)	−0.0270	−0.390	−0.0170
`wte` (letter a, 1st)	0.0014	+0.282	−0.0086
`wte` (letter e, 1st)	−0.1026	0	−0.1026

[11] The program shuffles (randomly reorders) the names and then uses them one at a time from the top (lines 20 and 156). Random numbers on a computer are calculated from a starting number called the **random seed**, and the same seed always gives the same sequence of numbers. In microgpt, line 12, `random.seed(42)`, sets the seed to 42, so anyone who runs the same program gets mia in step 1. Changing the seed to another number changes the shuffled order, the initial values of the parameters, and the lottery used when making names.

We can read the following from this.

- **Knobs with a positive gradient went down, and those with a negative gradient went up.** Each moved in the direction that lowers the loss.
- **Every knob moved by exactly 0.01.** It makes no difference whether the gradient is 0.46 or 0.28. This comes from Adam's "even out the turning amount" refinement: in the first step, every knob moves by exactly the learning rate (0.01), in the direction opposite to its gradient.
- **Some knobs have a gradient of 0 and do not move at all.** 16 of the 268 were like this: the embeddings of the letters e, n, v, which do not appear as inputs in `mia` (3 letters $\times$ 4 = 12), and the embedding of the unused 5th position (4). Parts that are not used do not affect the loss, so their gradient is 0.

5.4 The change after one step

After the knobs were moved, asking the model to predict the same `mia` again gave the right-hand column of the earlier table: all four correct probabilities rose a little, and the loss fell from 1.953 to 1.889.

However, checking the loss for the other names gave the following.

Name	Loss before	Loss after 1 step
`mia` (the name used for training)	1.953	1.889 (down)
`ava`	1.971	1.938 (down)
`emma`	1.963	1.979 (up)
`anna`	2.001	2.002 (about the same)

Moving the knobs to suit `mia` improved `ava`, which, like `mia`, "ends in a," but made `emma` worse. An adjustment made for one name is not necessarily good for the others. That is exactly why training **repeats many times with different names**, searching for knob positions that lower the loss overall.

5.5 After 300 steps

Training for 300 steps, using the four names in turn, lowered the loss as shown in Figure 11.

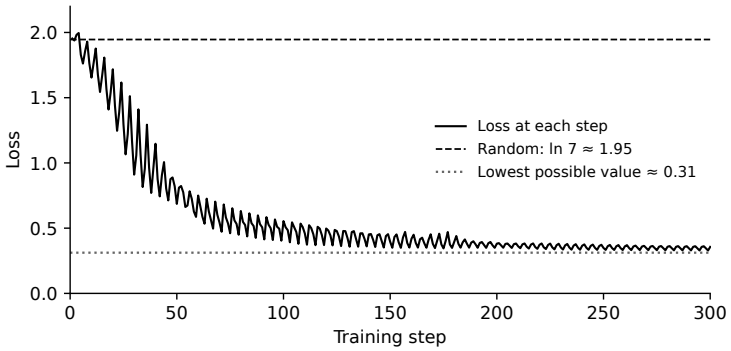


Figure 11 The change of the loss in the small example. The dotted line is the value below which the loss cannot go, in principle (see text).

Figure 12 compares, before and after training, the probabilities of the "next letter" when `anna` is fed in one letter at a time. The dark gray bars are the correct letters.

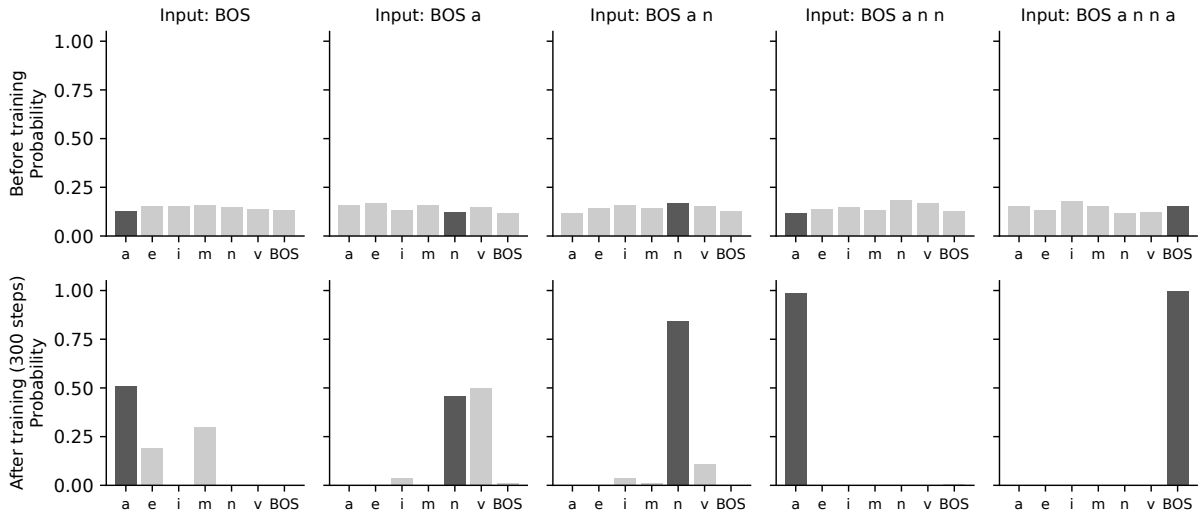


Figure 12 The probabilities of the next letter when `anna` is input (top: before training; bottom: after 300 steps). Dark gray marks the correct letter.

Before training, the model gives almost the same probability to every letter. After training:

- **The first letter** (input BOS): a is 0.51, m is 0.30, e is 0.19. Of the first letters of the four names, two are a (ava, anna) and one each is m and e, so these values are close to those proportions (2/4, 1/4, 1/4).
- **After a** (input BOS a): n is 0.46, v is 0.50. Two names start with a, ava and anna, so which comes next really is fifty-fifty.
- **After ann, and after anna**: the model predicts a and BOS (the end) almost certainly, because here only one answer is possible.

In other words, the model has learned from the data **"where it should be unsure, and where it need not be."** The first letter and the letter after a are genuinely uncertain, so however well the model is trained, the loss cannot reach 0. For the four names, the lowest possible loss is about 0.31, and the loss after training (about 0.35) has come quite close to it.

The letters' name tags changed too In Section 3.2 we explained that the 16 numbers in the letter table (`wte`) are like "name tags describing the letter's features," and that letters used in similar ways come to have similar lists of numbers. Let us compare the actual name tags in the small example (4 numbers per letter).

Letter	Name tag before training (4 numbers)	Name tag after training (300 steps)
a	+0.00 +0.04 +0.11 +0.08	+0.12 −0.21 +0.19 +0.18
e	−0.10 −0.03 +0.03 +0.01	−0.17 +0.07 −0.08 +0.28
i	+0.01 +0.11 −0.12 −0.05	**−0.13 +0.05 +0.05 −0.23**
m	−0.03 −0.08 −0.06 +0.02	−0.06 +0.03 −0.12 +0.00
n	+0.09 −0.10 +0.05 +0.07	−0.02 +0.01 −0.01 −0.01
v	−0.01 +0.05 +0.00 −0.04	**−0.13 +0.06 +0.05 −0.22**
BOS	−0.05 +0.05 −0.03 +0.05	−0.09 +0.23 −0.02 +0.01

After training, **the name tags of i and v have almost the same list of numbers**. Among the four names, i appears in mia and v in ava, and in both cases **the next letter is**

always a. Because they are used in a similar way ("the same letter comes next"), they seem to have learned similar name tags. The value expressing whether two name tags "point in the same direction" (the cosine similarity, Appendix B) went from 0.64 before training to 1.00 after training (pointing in almost exactly the same direction). What is happening when the name tags are viewed as vectors (arrows) is explained with a 3D figure in Appendix B.

The names it made When the trained model was asked to make 20 names, it produced `ava` 8 times, `anna` 7 times, `mia` 4 times and `emma` once: **not a single name other than the four**. With too little data, the model simply "memorized" the names. As the next section shows, training on 32,033 names lets the model make new names that are not in the data.

6 The real training run (32,033 names)

6.1 What "training" means

> "Training" or "learning" of an AI means something different from human learning. For an AI, training is **adjusting the parameters (knobs) so that the loss becomes small**. Nothing more, nothing less.

This is very similar to a familiar task in science experiments. For example, when we hang weights on a spring, measure how far it stretches, and draw a straight line on the graph, we choose the slope and intercept of the line so that "the gap between the data points and the line" is as small as possible (the **least squares method** minimizes the sum of the squared gaps). Here,

- the slope and intercept are the "parameters" (2 of them),
- the sum of the squared gaps is the "loss,"
- finding the slope and intercept that make the gap smallest is "training."

What microgpt does is essentially the same; only the parameters have increased to 4,192 and the loss has become the "average surprise" (cross-entropy). However, there are too many parameters to solve with a formula, so it searches for positions where the loss is small by moving little by little, guided by the gradients (gradient descent). How to read the value of the loss is explained in Appendix A.5.

6.2 How the loss changes

Back to the original settings, one step is repeated 1000 times with different names. Figure 13 shows how the loss changes.

The average over the first 50 steps was 2.85, and the average over the last 100 steps was 2.28. Because only one name is used per step, the loss at each step varies a lot from name to name (gray line). The 1000 steps use 1000 names, only about 3% of the whole data.

When the model partway through training was asked to make names, they changed as follows (same method as in Section 7).

Training step	Generated names (first 5)
0 (before training)	jdodmicpalbdjxcj, pzol, bwhechtepsnqbdeq, liomhsugpozri, ckufnbqwpykuntmx
100	airayne, areen, aln, en, aveeer
300	faran, ananan, male, kayay, bameh
1000	caran, ananan, nail, kaya, alan

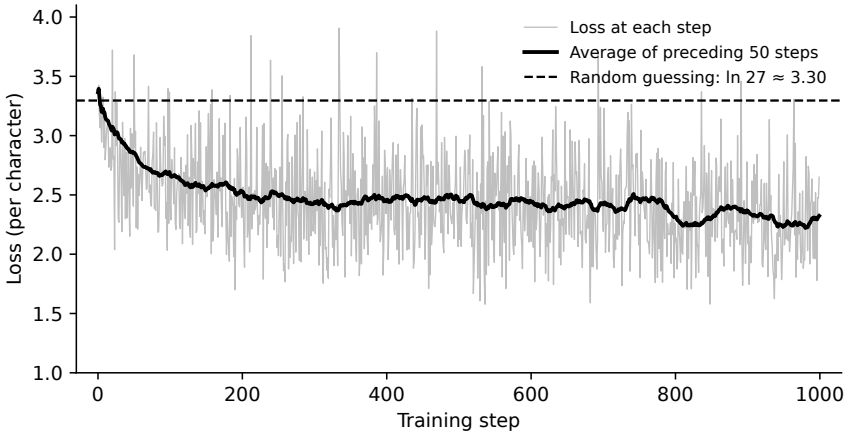


Figure 13 The change of the loss. Gray: the loss at each step (the value for the one name used in that step); black line: the average over the preceding 50 steps.

Before training, the letters are lined up at random and the final BOS rarely appears, so many names run on to the maximum of 16 letters. By 100 steps, vowels and consonants alternate, and from 300 steps on, the names have a "name-like" length and sound.

7 Making names

7.1 How it works

After training, new names are made with the same `gpt()` function (lines 186–200). This is called **generation**.

1. Start from BOS.
2. Get the probabilities of the next letter with `gpt()` (line 195).
3. **Draw a lot in which each letter's chance of winning is its probability**, and choose one letter (line 196). Letters with higher probability are more likely to win.
4. If the chosen letter is BOS, stop. Otherwise, use that letter as the next input and go back to 2.

For example, after emma (the letter after the final a), the probabilities adjusted with temperature 0.5 are 0.62 for n and 0.19 for BOS (end), with the other letters sharing the rest in small amounts (Figure 14). This is like drawing one ticket from a lottery of 1000 tickets in which about 620 are "n," about 190 are "end," and the remaining 190 or so are other letters. n is certainly likely to be chosen, but not certain: about once in five draws "end" comes up and the name ends as emma. Because it is a lottery, the same model makes a different name each time[12].

When the probabilities are computed, the scores are divided by a value called the **temperature** (0.5). A temperature below 1 widens the differences between scores, so letters with high probability are chosen even more often (Figure 14)[13].

[12] If letters were chosen completely at random, ignoring the probabilities, all 27 would be equally likely and the result would be a random string, as before training. Conversely, if the lottery were dropped and "the most probable letter" were always chosen (**greedy decoding**), the same model would always produce exactly the same name. A lottery weighted by the probabilities keeps the names name-like while still producing a variety of names. The program uses the function `random.choices`, which draws a weighted lot.

[13] A temperature below 1 gives more ordinary names; above 1 the differences shrink and rare letters are chosen more easily (more unusual names). The temperature is not used in training (training uses the scores as they are); it is a setting chosen by the user when making names. In microgpt the author set it to 0.5 on line 187, and the names can be remade with a different value without retraining. In real LLMs, too, the temperature is not part of

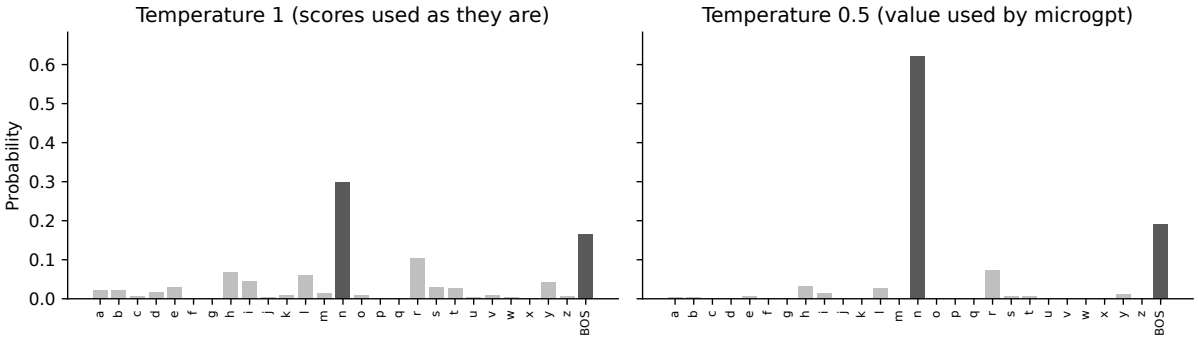


Figure 14 How the temperature changes the probabilities (same situation as Figure 6). At temperature 0.5, the probability of the top candidate n rises from 0.30 to 0.62, and letters with low probability become even less likely.

In training, the predictions at every position of a name could be handled together, but in generation **the chosen letter becomes the next input**, so the process can only move forward one letter at a time. This is also why real LLMs display their text a little at a time.

7.2 Results

The 20 names made after 1000 training steps are as follows (running the original program unchanged).

> kamon, ann, karai, jaire, vialan, karia, yeran, anna, areli, kaina, konna, keylen, liole, alerin, earan, lenne, kana, lara, alela, anton

> Some are real names, such as `ann`, `anna`, `lara` and `anton`, but many, like `kamon` and `vialan`, **are not in the data yet look like names**. Unlike the small example in Section 5, the model has not memorized names; it has learned the statistics of "how letters tend to be arranged in names."[14]

7.3 Looking inside the model

When `emma` is fed into the trained model one letter at a time, the top probabilities of the predicted "next letter" at each position are as follows.

Input	Probability of the next letter (top 5)
BOS	a 0.142, k 0.089, j 0.081, m 0.079, s 0.070
BOS e	l 0.284, r 0.124, n 0.115, m 0.063, a 0.059
BOS e m	i 0.221, a 0.218, e 0.184, y 0.090, o 0.066
BOS e m m	i 0.254, a 0.228, e 0.166, y 0.080, o 0.051
BOS e m m a	n 0.298, **BOS (end) 0.166**, r 0.103, h 0.068, l 0.061

the model but a setting at the time of use: in developer APIs the user can specify it, and in chat apps the value chosen by the provider is used. The name "temperature" comes from physics (Appendix A.4).

[14] In that it produces plausible things that are not in the data, this works by the same mechanism as the phenomenon in which real LLMs give convincing but untrue answers (**hallucination**). The model outputs not "what is true" but "a likely continuation."

The most common first letter of a name is a; after em comes a vowel; and after emma the top candidates are "n follows (emman…)" and "it ends here." The correct letters (m, m, a, end) are not always ranked first, but they are among the top candidates.

The Attention weights of the four heads at the same moment are shown in Figure 15.

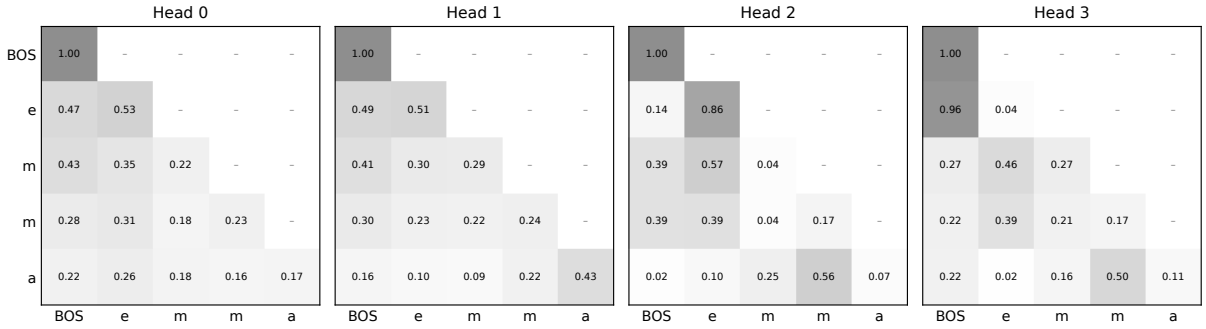


Figure 15 The Attention weights when `emma` is input (after training). Rows are the letter currently being processed, columns are the letter being looked back at; each row adds up to 1. The "–" marks in the upper right are future positions that do not exist yet.

The upper right is empty because there are no notes on future letters yet. The heads clearly distribute their weights differently (for example, in the row of the final a, head 1 puts a large weight on itself, while heads 2 and 3 put large weights on the second m). However, it is not easy to read from this figure what each head is capturing[15].

8 Comparison with real GPTs

8.1 Scale

	microgpt	GPT-2 (smallest)	GPT-3
Numbers per letter (per token; dimension)	16	768	12,288
Number of layers	1	12	96
Number of heads	4	12	96
Maximum length	16	1,024	2,048
Vocabulary size	27	50,257	50,257
Number of parameters	4,192	about 124 million	about 175 billion

GPT-3 has about 40 million times as many parameters as microgpt.

8.2 Differences in the content of the computation

As the comment on line 93 of the program says, microgpt's model has almost the same structure as GPT-2, with only three small differences (the normalization method, whether there are constant terms, and the kind of function used in the MLP). All other differences lie not in the content of the computation but in **the way it is computed**.

[15] It is generally very difficult for humans to interpret the meaning of parameters decided by training. Research that tries to explain from the inside why a model predicts well (**interpretability** research) is very active today.

	microgpt	Real LLMs
Token	1 letter	frequent combinations of letters
Data used in one step	1 name	millions of tokens (many documents together)
Unit of computation	one number at a time	tables of numbers (matrices) at once
Computer	one CPU, one at a time	many GPUs at the same time
Adjustment after training	none	further training to match human preferences[16]

In microgpt one training step takes 0.1 seconds or more, because Python computes the numbers one at a time. Real LLMs perform the same kinds of calculation in huge numbers at the same time on GPUs. Recent models add several further refinements, but the framework "**turn letters into lists of numbers (vectors) → repeat Attention and MLP → output the probability of the next token**" and the learning principle "**lower the loss using gradients**" are the same.

9 Summary

1. microgpt is a GPT that implements the data, tokenizer, autograd, model, training and generation in about 200 lines, using only Python's built-in features.
2. The model gives scores to the "next letter" and turns them into probabilities with softmax. Attention looks back at earlier letters to gather information, and the MLP thinks things over, letter by letter.
3. In training, the "surprise" (loss) is measured from the probability given to the correct answer, autograd finds the gradients of all the knobs (matrix entries), and Adam turns the knobs little by little.
4. With four names, we can follow in numbers how the knobs move in one step. With too little data the model memorizes the names, but trained on 32,033 names it can make new, name-like names.
5. The differences from real LLMs lie mainly in scale and the way the computation is done; the framework and the principle of learning are the same.

Appendix A microgpt in formulas

Here the calculations explained in words in the main text are written as formulas. Vectors (lists of numbers) are written in bold, as $\mathbf{x}$, and x_i is the i-th number. W with a subscript (W_Q, etc.) is a matrix. d is the dimension of the vectors ($d = 16$ in microgpt), and ln is the natural logarithm.

[16] Conversational AIs such as ChatGPT and Claude first learn "next-token prediction" from a huge amount of text, and are then adjusted with further training based on examples of following instructions and on human ratings (instruction tuning, RLHF: Reinforcement Learning from Human Feedback, etc.), so that they become good at answering questions.

A.1 Matrix times vector

The product $\mathbf{y} = W\mathbf{x}$ of an $m \times n$ matrix W (entries W_{ij}) and an n-dimensional vector $\mathbf{x}$ is an m-dimensional vector whose i-th component is

$$y_i = \sum_{j=1}^{n} W_{ij}\, x_j = W_{i1}x_1 + W_{i2}x_2 + \cdots + W_{in}x_n,$$

that is, the inner product of the i-th row of the matrix and the vector (Figure 8, lines 94–95).

A.2 Embedding and normalization

Add the row $\mathbf{e}_{\text{letter}}$ of `wte` for the letter's number and the row $\mathbf{p}_{\text{position}}$ of `wpe` for the position.

$$\mathbf{x} = \mathbf{e}_{\text{letter}} + \mathbf{p}_{\text{position}}$$

Normalization (RMSNorm) divides by the root mean square of the components to even out the size (lines 103–106). $\varepsilon = 10^{-5}$ is a small number that prevents division by 0.

$$\text{RMSNorm}(\mathbf{x}) = \frac{\mathbf{x}}{\sqrt{\dfrac{1}{d}\displaystyle\sum_{i=1}^{d} x_i^2 + \varepsilon}}$$

A.3 Attention

Make the query, key and value from the normalized vector (lines 118–120).

$$\mathbf{q} = W_Q\mathbf{x}, \qquad \mathbf{k} = W_K\mathbf{x}, \qquad \mathbf{v} = W_V\mathbf{x}$$

The 16 numbers are split into four heads of 4. With $\mathbf{q}^{(m)}$ the query of head m, $\mathbf{k}_s^{(m)}$ and $\mathbf{v}_s^{(m)}$ the key and value at position s, and t the current position, the weights a_s and the output of the head are

$$a_s = \text{softmax}_s\left(\frac{\mathbf{q}^{(m)} \cdot \mathbf{k}_s^{(m)}}{\sqrt{d_k}}\right), \qquad \text{head}_m = \sum_{s=0}^{t} a_s\, \mathbf{v}_s^{(m)}.$$

$\mathbf{q} \cdot \mathbf{k}$ is the inner product, which expresses how well the query and key "match." $d_k = 4$ is the dimension of one head. The fact that the sum only runs from $s = 0$ to t corresponds to "not looking at the future." The outputs of the four heads are joined, multiplied by W_O, and the vector from the entrance is added (residual connection, lines 133–134).

A.4 MLP, output and softmax

$$\mathbf{x} \leftarrow \mathbf{x} + W_2\,\text{ReLU}\big(W_1\,\text{RMSNorm}(\mathbf{x})\big), \qquad \text{ReLU}(z) = \max(0, z)$$

W_1 is a 64×16 matrix and W_2 is 16×64 (lines 137–141). Without ReLU, W_2W_1 would collapse into a single 16×16 matrix, and expanding the middle to 64 dimensions would be pointless. Finally, multiplying by `lm_head` (27×16) gives 27 scores $z_1, \ldots, z_{27}$, which softmax turns into probabilities.

$$p_j = \frac{e^{z_j}}{\sum_{i=1}^{27} e^{z_i}}$$

Softmax is a two-step calculation. (1) Replace each score z by e^z ($e = 2.718\ldots$ raised to the power z). This makes every number positive, including negative scores, and the larger the score, the more sharply the value grows. (2) Divide by the total. This makes the sum exactly 1. For

example, if three letters have scores 2, 1, 0, then $e^2 \approx 7.39$, $e^1 \approx 2.72$, $e^0 = 1$, with a total of 11.11. Dividing each by the total gives probabilities 0.67, 0.24, 0.09. Even though the scores differ by only 1 each, the probabilities differ much more.

In generation, the scores are divided by the temperature T (0.5) before taking the softmax.

$$p_j = \frac{e^{z_j/T}}{\sum_i e^{z_i/T}}$$

This has the same form as the **Boltzmann distribution** of **statistical mechanics**. In a system at absolute temperature T, the probability that a state with energy E_j is realized is

$$p_j = \frac{e^{-E_j/k_BT}}{\sum_i e^{-E_i/k_BT}} \qquad (k_B \text{ is the Boltzmann constant}).$$

If the score z_j is regarded as the energy with a minus sign ($z_j \leftrightarrow -E_j/k_B$), this is exactly the same form as softmax with a temperature. As $T \to \infty$ all states approach equal probability (the **uniform distribution**), and as $T \to 0$ only the lowest-energy state (the highest-scoring letter) is chosen (the same as greedy decoding). Because of this correspondence, softmax is also called the Boltzmann distribution or the **Gibbs distribution**, and this is where the name "temperature" comes from.

A.5 Loss and gradients

For a name of length n, with p_t the probability of the correct letter at position t, the loss is

$$L = -\frac{1}{n}\sum_{t=0}^{n-1} \ln p_t$$

(the **cross-entropy loss**). Giving the same probability to all 27 kinds gives $p_t = 1/27$ at every position, so

$$L = -\ln\frac{1}{27} = \ln 27 \approx 3.30.$$

$\ln 27$ is the number "to which e must be raised to give 27"; indeed $e^{3.30} \approx 27.1$. Since the loss at every position is the same, the average is also 3.30 regardless of the length of the name. For the 7 kinds of the small example it is $\ln 7 \approx 1.95$.

The actual first step gave a slightly larger value, 3.37, because the parameters before training are small random numbers, so the probabilities of the 27 kinds are only "almost" equal and vary a little. Since $-\ln p$ rises sharply as the probability decreases (Figure 7), such variation tends to make the average loss a little larger than 3.30 (this follows from the convexity of $-\ln p$, by **Jensen's inequality**).

Reading the value of the loss: perplexity The value e^L computed from the loss L is called the **perplexity**. It is a measure of "how many candidates, on average, the model is torn between." Indeed, a model that gives equal probability $1/K$ to K candidates has loss $\ln K$, so $e^L = K$.

- Before training: $L \approx 3.30 \to e^{3.30} \approx 27$. Equally torn between all 27 letters (random).
- After training (average of the last 100 steps): $L \approx 2.28 \to e^{2.28} \approx 9.8$. The candidates have been narrowed down to about 10 letters.
- Small example (Section 5, after 300 steps): $L \approx 0.35 \to e^{0.35} \approx 1.4$. Hardly torn at all (close to memorization).

Saying "it went from a choice among 27 to a choice among about 10" conveys what training achieved more vividly than "the loss went down from 3.30 to 2.28."

Reading the loss curve The following can be read from the loss curve (**learning curve**) in Figure 13.

- It drops sharply at first, then more and more gently. The gradual lowering of the learning rate also plays a part.
- The value at each step (gray) varies a lot depending on the one name used in that step; the trend is seen in the moving average (black line). Using more data per step (a **mini-batch**) would reduce the variation.
- The loss has a lower limit determined by the uncertainty in the data itself. In the small example of Section 5 it is about 0.31, and the loss after training approached it.
- What is measured here is the loss on the names used for training (the **training loss**). Normally one also measures the loss on data not used for training (the **validation loss**) to check that the model is not just memorizing (**overfitting**). microgpt uses only about 3% of the data in 1000 steps, so overfitting is hardly a concern.

The gradient $\partial L/\partial w$ is the rate of change of L when the parameter w is changed slightly. In autograd, the computation graph is traced back from the output side, and for each number v, the following is added to each of its inputs u:

$$\frac{\partial L}{\partial u} \mathrel{+}= \frac{\partial v}{\partial u} \cdot \frac{\partial L}{\partial v}$$

(the **chain rule**). In the example of Figure 10, this agrees with $\partial f/\partial a = b + 1 = 4$ and $\partial f/\partial b = a = 2$, obtained by differentiating $f = ab + a$ directly. The program in Appendix C confirms that it agrees with numerical differentiation.

A.6 Adam

For each parameter w, a moving average m of the gradient g and a moving average v of g^2 are kept. At the k-th step ($k = 1, 2, \dots$) they are updated as

$$m \leftarrow \beta_1 m + (1 - \beta_1)\, g, \qquad v \leftarrow \beta_2 v + (1 - \beta_2)\, g^2,$$

and after correcting for the bias of m, v toward 0 in the early steps with $\hat{m} = m/(1 - \beta_1^k)$, $\hat{v} = v/(1 - \beta_2^k)$, the parameter is set to

$$w \leftarrow w - \eta\, \frac{\hat{m}}{\sqrt{\hat{v}} + \epsilon}.$$

In microgpt, $\beta_1 = 0.85$, $\beta_2 = 0.99$, $\epsilon = 10^{-8}$, and the learning rate η is lowered linearly from 0.01 to 0.

In step 1 ($k = 1$), $\hat{m} = g$ and $\hat{v} = g^2$, so $\hat{m}/\sqrt{\hat{v}} = g/|g| = \pm 1$. That is, every parameter with a nonzero gradient moves by exactly $\eta = 0.01$ in the direction opposite to its gradient. This is why "every knob moved by 0.01" in Section 5.3.

A.7 Inner product and cosine similarity

If θ is the angle between two vectors $\mathbf{a}$ and $\mathbf{b}$, their inner product is

$$\mathbf{a} \cdot \mathbf{b} = \sum_i a_i b_i = |\mathbf{a}|\, |\mathbf{b}| \cos\theta,$$

and the cosine similarity is this divided by the lengths:

$$\cos\theta = \frac{\mathbf{a} \cdot \mathbf{b}}{|\mathbf{a}|\, |\mathbf{b}|}.$$

The inner product depends on both direction and length, while the cosine similarity depends only on direction and always lies between -1 and 1 (Appendix B).

Appendix B Viewing the directions of the name-tag vectors in 3D

In Section 5.5 we saw that after training the name tags (rows of `wte`) of the letters i and v become almost the same list of numbers. Since the name tags are vectors, drawing them as arrows should let us see that they "point in the same direction." However, the small example in the main text has 4 numbers per letter (4 dimensions), which cannot be drawn. So, to make a drawable figure, we **changed it to 3 numbers per letter (3 dimensions)** and retrained on the same four names. Training worked in 3D too: the loss fell to about 0.36, and every generated name was one of the four names.

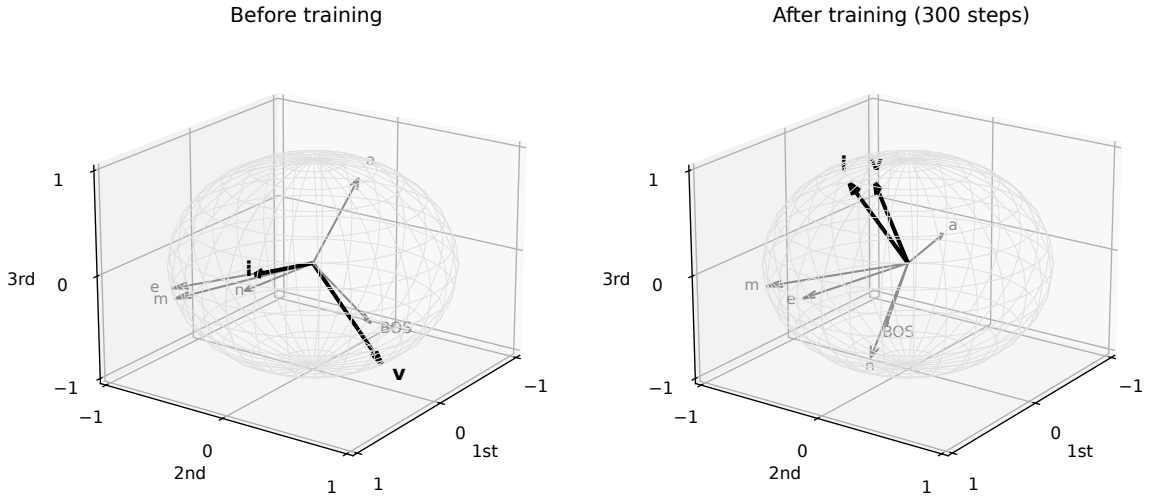


Figure 16 The name tags (3-dimensional vectors) of the 7 kinds of letter drawn as arrows (left: before training; right: after 300 steps). To make directions easy to compare, all arrows have been scaled to length 1. Black: i and v.

On the left of Figure 16 (before training), the arrows for i and v point in different directions. On the right (after training), **i and v point in almost the same direction**. A list of 3 numbers is a 3-dimensional vector and can be drawn as an arrow like this. The list of 16 numbers in the main text (a 16-dimensional vector) cannot be drawn, but the idea is the same: the name tags of letters used in similar ways come to point in similar directions in 16-dimensional space.

Why the arrows have equal length The model evens out the size of the numbers (normalization) at the entrance of Attention and the MLP, so what mainly matters in a name tag is its direction. In the figure, therefore, all arrows have been scaled to length 1 so that only differences in direction are visible.

Expressing closeness of direction in one number: cosine similarity How close the directions of two arrows are can be expressed by a single number, the **cosine similarity**. It is $\cos\theta$, where θ is the angle between the two arrows: 1 for the same direction, 0 for a right angle, and -1 for opposite directions. In the 3D example, the cosine similarity of i and v was 0.04 before training (almost a right angle) and 0.98 after training.

The difference from the inner product The cosine similarity is the **inner product** of the vectors divided by their two lengths (formula in Appendix A.7). The inner product depends on

both direction and length, and even for the same direction, longer vectors give larger values. In the 3D example, the inner product of i and v is $+0.0109$ and that of i and a is -0.0156; judging by absolute values alone, the pair i and a, whose directions differ greatly, gives the larger value. This is because the vector for a is long (about 3.6 times as long as i). To compare whether directions are similar, the cosine similarity, which removes the effect of length, is more suitable.

Note that when the model itself measures how well a query and a key match in Attention, it uses the inner product as it is, not the cosine similarity (line 129). Since length as well as direction affects the result, the model can use length to adjust "how strongly to attend." The cosine similarity in this appendix is a tool for humans to compare the trained name tags.

Appendix C Python code: a minimal example of autograd (standard library only)

This program computes the example of Figure 10 with a small autograd that has only addition and multiplication, and compares the result with numerical differentiation. The second half computes the number of parameters in Section 4.2. It is microgpt's `Value` class reduced to the minimum for explanation.

```
# Minimal autograd, checked by numerical differentiation (standard library only)
class V:
    def __init__(self, data, children=(), local=()):
        self.data, self.grad = data, 0.0
        self.children, self.local = children, local
    def __add__(self, o): return V(self.data + o.data, (self, o), (1.0, 1.0))
    def __mul__(self, o): return V(self.data * o.data, (self, o), (o.data, self.data))
    def backward(self):
        order, seen = [], set()
        def visit(v):                       # put children first (topological order)
            if v not in seen:
                seen.add(v)
                for c in v.children: visit(c)
                order.append(v)
        visit(self)
        self.grad = 1.0
        for v in reversed(order):           # chain rule, from the output side
            for c, g in zip(v.children, v.local):
                c.grad += g * v.grad        # a variable used twice: contributions add

a, b = V(2.0), V(3.0)
f = a * b + a
f.backward()
print("f =", f.data, " df/da =", a.grad, " df/db =", b.grad)

h = 1e-6                                    # check by numerical differentiation
fn = lambda x, y: x * y + x
print("numerical:", round((fn(2 + h, 3) - fn(2, 3)) / h, 4), round((fn(2, 3 + h) - fn(2, 3)) / h, 4))

# breakdown of the number of parameters in microgpt
vocab, d, ctx = 27, 16, 16
parts = {"token embedding wte": vocab * d, "position embedding wpe": ctx * d,
         "attention Wq,Wk,Wv,Wo": 4 * d * d, "MLP fc1,fc2": 2 * 4 * d * d,
         "output layer lm_head": vocab * d}
for k, v in parts.items(): print(f"{k}: {v}")
print("total", sum(parts.values()))
```

Output:

```
f = 8.0  df/da = 4.0  df/db = 2.0
numerical: 4.0 2.0
token embedding wte: 432
position embedding wpe: 256
attention Wq,Wk,Wv,Wo: 1024
MLP fc1,fc2: 2048
output layer lm_head: 432
total 4192
```

Appendix D How the numbers and figures in the text were obtained

The numbers and figures in the text were obtained by making the following changes to the original program. The calculations of the model and of training themselves were not modified.

- **Recording the loss (Figures 13 and 11)**: Instead of the progress display on line 184, the loss of each step was saved to a list.
- **Generation during training (table in Section 6)**: Inside the training loop, at the start of steps 0, 100 and 300 and after the end of training, 10 names each were generated by the same method as in Section 7 (temperature 0.5). To avoid affecting the random numbers of the training itself, the state of the random number generator was saved and restored before and after generation, and a separate seed (7) was used for generation.
- **Internal values (Figures 5, 6, 14, the table in Section 7.3, Figures 15 and 12)**: The parameter values after training were saved to a file, and with a program that rewrites the same calculation as `gpt()` using ordinary decimal numbers, the probabilities at each position, the Attention weights and the values inside the MLP were computed. The "about 8 per letter on average" in Section 3.4 is the average number of responding detectors (7.7) over 300 names chosen at random from the data (2,156 letters in total).
- **The small example (Section 5)**: `input.txt` was replaced with just four names (emma, ava, mia, anna), and the settings were changed to `n_embd = 4`, `n_head = 1`, `block_size = 5`, `num_steps = 300`. The random seed (42) was left unchanged. For step 1, the values and gradients of all parameters were saved right after `loss.backward()`, and the values of all parameters right after the update. The "lowest possible loss" in Section 5.5 is the value (about 0.312) calculated from the proportions of the first letters of the four names and of the letters after a.
- **The 3D example (Appendix B)**: Starting from the settings of the small example, only `n_embd` was changed to 3 and the model was retrained.

Appendix E Where to get the microgpt program, and its line numbers

This booklet explains microgpt while reading its program, but the program itself is not reproduced here. No license (terms of use) is given for the program, so please obtain it from the author's distribution page.

- Source: Andrej Karpathy, microgpt (GitHub Gist)
  https://gist.github.com/karpathy/8627fe009c40f57531cb18360106ce95
- Explanation by the author: https://karpathy.github.io/2026/02/12/microgpt/
- The line numbers in this booklet refer to the version obtained on September 30, 2026 (200 lines in total; the distribution page was last updated on September 19, 2026). If the page

is updated, the line numbers may shift.

The correspondence between the line numbers referred to in the text and their contents is as follows.

Lines	Contents	Section
1–7	Description of the program ("this file is the complete algorithm")	1
9–12	Loading libraries; line 12 sets the random seed to 42	5.2
14–21	Data: reads `input.txt` and shuffles the order of the names	2.1
23–27	Tokenizer: gives the letters numbers and adds BOS	2.2
29–72	Autograd (`Value` class): lines 39–57 are the elementary operations and their local influences, lines 59–72 the backward calculation `backward()`	4.3
74–90	Parameters: lines 75–79 set the sizes, line 80 sets the initial (random) values, line 89 rearranges them into one list	3.2, 4.2
92–106	Helper functions: line 93 is the comment on the differences from GPT-2, lines 94–95 matrix × vector (`linear`), lines 97–101 softmax, lines 103–106 normalization (RMSNorm)	3.5, 4.2, 8.2
108–144	The model `gpt()`: lines 109–112 embedding, lines 115–134 Attention (lines 121–122 appending to the notes, lines 124–132 the four heads), lines 135–141 MLP, line 143 output layer	3
146–149	Adam settings (learning rate, β_1, β_2) and storage for the moving averages	4.4
151–184	Training: lines 156–157 take one name, lines 163–169 the loss, line 172 the gradients, lines 174–182 the update by Adam	4, 5, 6
186–200	Generation: line 187 the temperature, line 195 computing the probabilities, line 196 the weighted lottery	7