Reading microgpt

Tracing GPT training and generation in 200 lines of Python — Study notes for science students

Edited by Yoshio Okamoto / Written by Claude (Anthropic) September 28, 2026

Introduction

microgpt is a Python program of about 200 lines, released by Andrej Karpathy in February 2026, that trains a GPT and generates text with it. It uses only the standard libraries `os`, `math` and `random`, and uses neither NumPy nor PyTorch. Data loading, the tokenizer, automatic differentiation, the Transformer itself, optimization and text generation are all contained in a single file.

This note is not a translation of Karpathy's own explanatory article; **it reads the program itself from top to bottom and re-explains it in the language of the notes so far (01–09)**. All the numbers in the text were confirmed by actually running the program. The original program is available at:

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

In the text, "line ◯" refers to the line number in this file (200 lines) as of September 2026. Only the few lines of code needed for the explanation are quoted; it is assumed that you have the whole program open alongside and read the two together.

How to run it: If Python 3 is installed, it runs with `python microgpt.py` (the same 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. From the second run on it uses that file, so no network connection is needed. On the computer used to prepare this note (Python 3.11), 1000 training steps plus the generation of 20 names took about 2 minutes 40 seconds.

Notation

- Vectors are column vectors and are written in bold, as $\mathbf{x}$. x_i is its i-th component.
- The superscript T denotes the **transpose**, not a power. $\mathbf{q}^\mathsf{T}\mathbf{k}$ is the inner product.
- d is the embedding dimension ($d = 16$ in microgpt), h is the number of heads ($h = 4$), and $d_k = d/h = 4$ is the dimension of one head.
- V (italic) is the vocabulary size ($V = 27$). It is different from the list of values, `values`, in the program.
- $p(\cdot)$ is a probability and ln is the natural logarithm. L denotes the loss.
- W with a subscript (W_Q, etc.) is a weight matrix. In microgpt a matrix is held as "a list of rows" and is multiplied from the left in the form $\mathbf{y} = W\mathbf{x}$ (the direction of multiplication is opposite to $Q = XW_Q$ in 06, but the computation is the same).

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 characters to numbers
Autograd	29–72	The `Value` class, which records the history of computations and obtains gradients by the chain rule
Parameters and model	74–144	The 4,192 parameters and the Transformer itself, `gpt()`
Training	146–184	Computes the loss, obtains the gradients and updates the parameters with Adam
Generation	186–200	Makes new names with the trained model

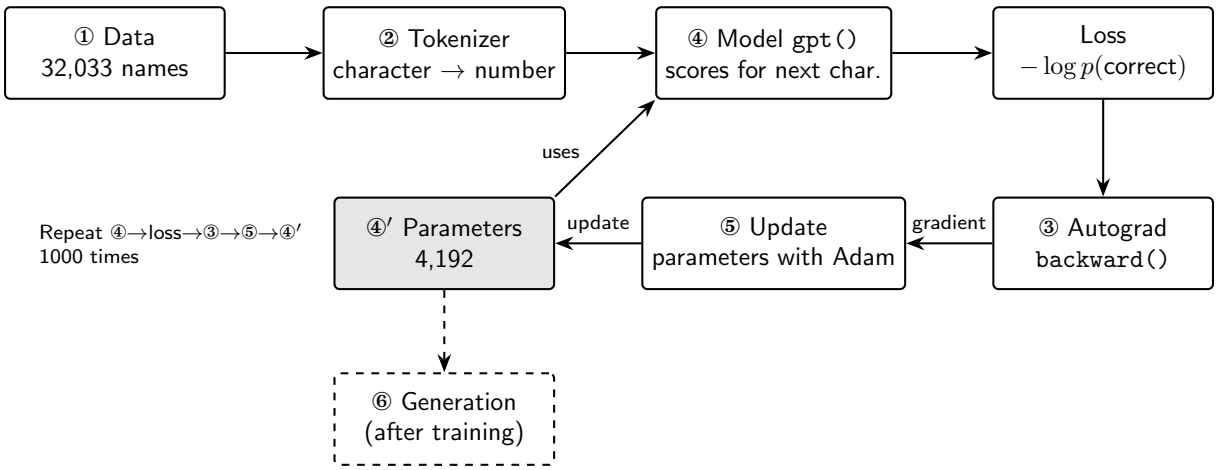


Figure 1 The structure of microgpt. In training, the flow ④ → loss → ③ → ⑤ is repeated 1000 times, changing the parameters little by little. After training, ⑥ generation is done with the same model and parameters.

The explanatory text at the top of the file 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 computation speed**, and that **the content of the computation** is the same as in these 200 lines (they are compared concretely in Section 8).

2 Data and tokenizer

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 character" from a large number of names.

2.2 Character-level tokenizer

Lines 24–26 assign the numbers 0–25 to the characters appearing in the data (the 26 lowercase letters a–z). Adding to these **a special symbol, BOS, that marks the beginning and end of a document** (number 26), the vocabulary size is $V = 27$.

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 at once: "after BOS comes e," "after BOS, e comes m," ..., "after BOS, e, m, m, a comes BOS (the end)."

It does not use tricks such as the BPE seen in 07 to merge several characters into one. It is the simplest possible tokenizer: **1 character = 1 token**.

3 Automatic differentiation: the `Value` class

3.1 What it does

Training requires $\partial L/\partial w$ (the gradient), the partial derivative of the loss L with respect to every parameter w. In 04, backpropagation was derived as layer-by-layer matrix formulas. microgpt does this **one number (scalar) at a time**.

One object of the `Value` class represents one number and remembers the following four things (lines 33–37).

Attribute	Meaning
`data`	the value of the number (computed in the forward pass)
`grad`	the partial derivative of the loss with respect to this number, $\partial L/\partial$(this number) (computed by backpropagation)
`_children`	the numbers used to make this number (the inputs of the computation)
`_local_grads`	the partial derivatives of this number with respect to each input (**local derivatives**)

For example, performing the multiplication $c = a \times b$ creates a new `Value`, recording a, b as its inputs and $\partial c/\partial a = b$, $\partial c/\partial b = a$ as its local derivatives (lines 43–45).

```
def __mul__(self, other):
    ...
    return Value(self.data * other.data, (self, other), (other.data, self.data))
```

Addition, powers, `exp`, `log` and `relu` likewise record the value and the local derivatives (lines 39–50). Subtraction and division are defined as combinations of addition, multiplication and powers (lines 51–57). In other words, **every computation in microgpt is a combination of 6 kinds of elementary operations whose local derivatives are known**.

As the computation proceeds, a **computation graph** recording which number was made from which numbers is built up automatically.

3.2 Backpropagation

`backward()` (lines 59–72) traverses the computation graph from the output side to the input side and obtains the gradients by the **chain rule**.

1. Arrange all the numbers in the graph in an order in which "inputs always come first" (topological order).
2. Set the `grad` of the output (the loss) itself to 1 ($\partial L/\partial L = 1$).
3. Go through the arranged order **in reverse**, and for each number v, **add** "local derivative $\times$ `grad` of v" to each of its inputs u.

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

The values are "added" because, when the same number is used in several computations, its partial derivative is the **sum** of the contributions from each path.

> **Math note: a small example**
> Compute $f = a \times b + a$ with $a = 2$, $b = 3$ (Figure 2). Writing the intermediate number as $c = a \times b = 6$, we have $f = c + a = 8$.

In backpropagation, first $\partial f/\partial f = 1$. Since the local derivative of addition is 1, the `grad` of c becomes 1 and 1 is added to the `grad` of a. Next, from $c = a \times b$, $b \times 1 = 3$ is added to the `grad` of a, and $a \times 1 = 2$ is added to the `grad` of b. The result is

$$\frac{\partial f}{\partial a} = 1 + 3 = 4, \qquad \frac{\partial f}{\partial b} = 2,$$

which agrees with $b + 1 = 4$, $a = 2$ obtained by differentiating $f = ab + a$ directly. Because a is used in two places, the contributions 1 and 3 from the two paths are added together. The program in Appendix A confirms that the result agrees with numerical differentiation.

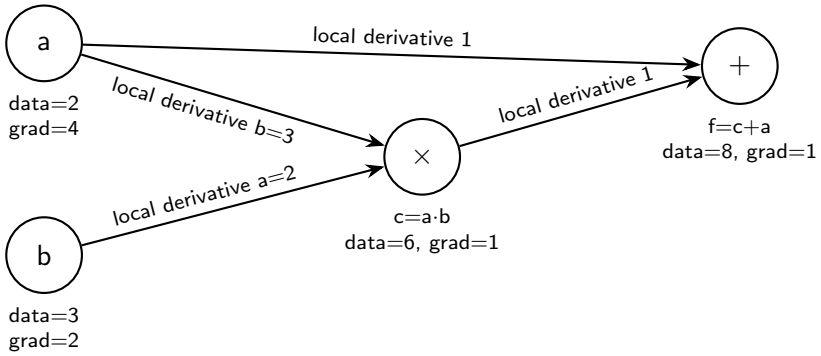


Figure 2 The computation graph of $f = a \times b + a$. The numbers beside the arrows are local derivatives; the `grad` under each circle is the partial derivative obtained by backpropagation.

In one training step of microgpt, this 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. Automatic differentiation in PyTorch and similar libraries follows the same principle; the difference is that it records operations in units of matrices (tensors) rather than scalars.

4 Parameters

4.1 Size settings

Lines 75–79 decide the size of the model.

Variable	Value	Meaning
`n_layer`	1	number of layers of Transformer blocks
`n_embd`	16	embedding dimension d
`block_size`	16	maximum length that can be handled (the longest name has 15 characters)
`n_head`	4	number of heads h
`head_dim`	4	dimension of one head $d_k = d/h$

4.2 List of parameters

All the parameters are initialized with normally distributed random numbers with mean 0 and standard deviation 0.08 (line 80). This is the "blank slate" state of the model before training.

Name	Size	Count	Role
`wte`	27×16	432	token (character) embedding
`wpe`	16×16	256	position embedding
`attn_wq, wk, wv, wo`	16×16 ($\times 4$)	1,024	attention W_Q, W_K, W_V, W_O
`mlp_fc1`	64×16	1,024	MLP layer 1 ($d \to 4d$)
`mlp_fc2`	16×64	1,024	MLP layer 2 ($4d \to d$)
`lm_head`	27×16	432	output layer (vector $\to$ scores of the 27 characters)
Total		**4,192**	

Applying "$12d^2$ per block" from Section 4.3 of 06 gives $12 \times 16^2 = 3{,}072$, which agrees with the total for attention and MLP in the table, $1{,}024 + 2{,}048$. The remaining 1,120 are the embeddings and the output layer.

5 The model itself: the `gpt()` function

5.1 Processing one token at a time

`gpt(token_id, pos_id, keys, values)` on lines 108–144 is the model itself. The Transformer seen in 06 gathered all the tokens of a sentence into a matrix X and processed them at once, but microgpt's `gpt()` processes **only one token per call** and returns 27 scores (logits) for that token's "next character."

The processing of one token proceeds in the following order.

1. Embedding (lines 109–112)
2. Attention block (lines 115–134)
3. MLP block (lines 135–141)
4. Output layer (line 143)

We look at them in order below.

5.2 Embedding

The row of `wte` corresponding to the character's number (a 16-dimensional vector) and the row of `wpe` corresponding to the position number are taken out and added.

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

As seen in Section 3.5 of 06, Attention by itself cannot distinguish order, so position information is added from outside in this way. Unlike the sinusoidal positional encoding introduced in 06, **the position vectors are also learned parameters**, as in GPT-2.

5.3 RMSNorm

At the entrance of each block, **normalization** is performed to even out the size of the vector (lines 103–106).

$$\text{RMSNorm}(\mathbf{x}) = \frac{\mathbf{x}}{\sqrt{\frac{1}{d}\sum_{i=1}^{d} x_i^2 + \varepsilon}} \qquad (\varepsilon = 10^{-5})$$

Dividing by the root mean square (RMS) of the components makes the RMS about 1. This corresponds to "Norm" in Figure 6 of 06, and is a simplified version of GPT-2's LayerNorm (subtract the mean, then divide by the standard deviation).

5.4 Attention block

First, a query, a key and a value are made 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 important part here is the next two lines (lines 121–122).

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

The key and value of the current token are **appended to the end of the lists of keys and values of the tokens so far**. Therefore, at the moment position t is being processed, the lists contain only the keys and values from position 0 to t.

The 16-dimensional vector is split into four pieces of 4 dimensions, and attention is computed for each head (lines 124–132). Writing the query of head m as $\mathbf{q}^{(m)}$ and the key and value at position s as $\mathbf{k}_s^{(m)}$, $\mathbf{v}_s^{(m)}$,

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

This is the Scaled Dot-Product Attention formula of 06 written out for one row (one query). The outputs of the four heads (4 dimensions each) are joined back into 16 dimensions and multiplied by W_O (line 133). Finally, the vector at the entrance of the block is added (line 134, **residual connection**).

> **Math note: why no causal mask is needed**
> In Section 3.4 of 06, a causal mask that adds $-\infty$ to the score matrix was used so that future tokens are not seen. microgpt has no masking at all.
> This is because the tokens are processed one at a time in order and keys and values are appended to the end, so when position t is processed, **the future keys and values do not exist yet**. The fact that the sum over s only runs from 0 to t itself plays the role of the causal mask.
> Also, this mechanism of "keeping the past keys and values and adding only those of the new token" is exactly the **KV cache** seen in Section 5 of 09. In real LLMs it is added later as a trick to speed up generation, but microgpt is written in this form from the start.

5.5 MLP block

The output of the attention block is normalized and passed through a two-layer fully connected network of $16 \to 64 \to 16$ dimensions (lines 137–140). The activation function is ReLU (0 if negative, unchanged if positive). Here too, the vector at the entrance is added (line 141, residual connection).

$$\mathbf{x} \leftarrow \mathbf{x} + W_2\,\mathrm{ReLU}\big(W_1\,\mathrm{RMSNorm}(\mathbf{x})\big)$$

This corresponds to the FFN in Section 4.2 of 06. The division of roles, in which the attention block handles the **exchange of information between tokens** and the MLP block handles the **transformation of each token**, is also the same as in 06.

5.6 Output layer

Finally, multiplying by `lm_head` (27×16) gives a score (logit) for each of the 27 characters (line 143). Passing these through the softmax function gives **the probability distribution of the next character**.

6 Training

6.1 Loss

In one training step, **only one name** is taken (line 156), and the model is made to predict the next character at each position (lines 163–168). With p_t the probability of the correct character at position t, the loss is the average of its negative logarithm (line 169).

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

This is the same as the cross-entropy loss seen in 07 and elsewhere. Since the parameters before training are random numbers, the 27 characters are predicted with almost equal probability, so we should get $L \approx \ln 27 \approx 3.30$. In fact, the loss at the first step was 3.37.

Since the whole computation of the loss is done with `Value` operations, at the end of the computation a computation graph from the 4,192 parameters to the loss has been built. A single call to `loss.backward()` on line 172 obtains the gradients of all the parameters.

6.2 Updating the parameters with Adam

Lines 174–182 update the parameters using the gradients. The method used is **Adam**, an improvement on the gradient descent $w \leftarrow w - \eta g$ seen in 04 (g is the gradient, η the learning rate).

Math note: Adam
For each parameter w, the **moving average** m of the gradient g and the moving average v of the squared gradient are kept (`m`, `v` in the program; different from V in the notation). At the k-th step ($k = 1, 2, \ldots$) 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, $\beta_1 = 0.85$, $\beta_2 = 0.99$, $\epsilon = 10^{-8}$.
Using the moving average m smooths out gradients that fluctuate from name to name. Also, dividing by $\sqrt{\hat{v}}$ makes parameters with large gradients and those with small gradients move by roughly the same amount.

The learning rate η starts at 0.01 and is lowered linearly to 0 as the steps proceed (line 175). Finally, the `grad` of each parameter is reset to 0 (line 182), and the next step begins.

6.3 Progress of training

This one step is repeated 1000 times with different names. Figure 3 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 fluctuates greatly depending on the name (gray line). The 1000 steps use 1000 names, only about 3% of the whole data.

When the model partway through training was made to generate names, they changed as follows (same method as in Section 7, temperature 0.5, 10 names each with the same random seed).

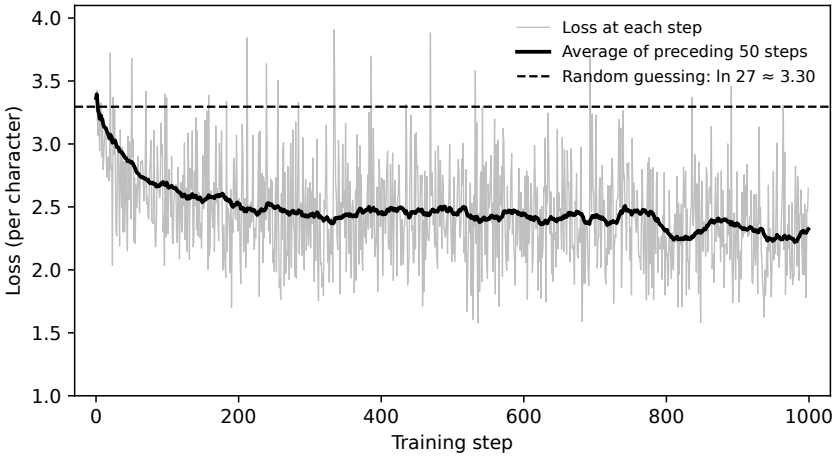


Figure 3 Change of the loss. Gray: the loss at each step (the value for the one name used in that step); black line: the average over the preceding 50 steps.

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

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

7 Generation

7.1 Mechanism

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

1. Start from BOS.
2. Obtain the scores of the next character with `gpt()`, divide them by the **temperature** T, and take the softmax to obtain a probability distribution (line 195).
3. **Draw one character by lot** according to those probabilities (line 196).
4. If the chosen character is BOS, stop. Otherwise, use that character as the next input and return to 2.

The temperature is set to $T = 0.5$. Dividing the scores by $T < 1$ widens the differences between them, so characters with high probability are chosen more often (see the note "LLM_ probability and temperature").

In computing the training loss (Section 6.1), the predictions at all positions were processed side by side, but in generation **the chosen character becomes the next input**, so the process can only advance one character at a time. This too is the same as the difference between training and generation seen in Section 4.4 of 06.

7.2 Generation results

The 20 names generated 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`, **do not appear in the data yet look name-like**. The model has not memorized names; it has learned the statistics of "character sequences likely to appear in names." In that it produces plausible things that are not in the data, it works by the same mechanism as the **hallucination** seen in Section 6 of 09.

7.3 Looking inside the model

When `emma` is fed into the trained model one character at a time, the top probabilities of the "next character" predicted at each position are as shown below (computed by the method of Appendix B).

Input	Probability of the next character (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 first character of a name is most often a; after em comes a vowel; and after emma, the top candidates are "n follows (emman…)" and "it ends here." The correct characters (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 4.

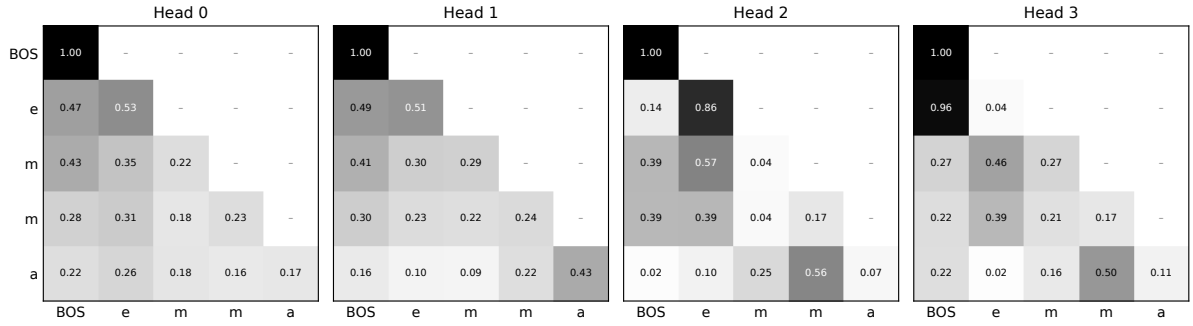


Figure 4 The attention weights when `emma` is input (after training). Rows are the token currently being processed (query), columns are the token being referred to (key); each row sums to 1. The "–" marks in the upper right are future positions that do not exist yet (math note in Section 5.4).

The upper right is empty because the future keys do not exist yet. It can be seen that the weights are distributed differently from head to head (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 (see the note "Why does it work even though we don't understand it?").

8 Comparison with real GPTs

8.1 Scale

	microgpt	GPT-2 (smallest)	GPT-3
Embedding dimension d	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; the differences are the following three points.

- It uses RMSNorm instead of LayerNorm (Section 5.3).
- The fully connected layers have no bias (constant term).
- The activation function of the MLP is ReLU, not GELU.

All other differences lie not in the content of the computation but in **the way it is computed**.

	microgpt	Real LLMs
Token	1 character	subwords made with BPE etc. (07)
Data used in one step	1 name	millions of tokens (many documents processed together)
Unit of computation	one number (scalar) at a time	grouped into matrices and tensors
Computer	one CPU, one number at a time in Python	many GPUs in parallel
Adjustment after training	none	instruction tuning, RLHF, etc. (09)

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 computation in massive parallel on GPUs. Recent models also add several further refinements, such as the handling of positions (RoPE) and shortcuts in the attention computation, but **the framework "embedding → repeated attention and MLP → probability of the next token" and the learning principle "lower the cross-entropy loss using gradients" are the same**.

9 Summary

1. microgpt is a GPT that implements the data, tokenizer, automatic differentiation, Transformer, Adam and generation in about 200 lines, using only the standard library.
2. Automatic differentiation breaks every computation down into elementary operations whose local derivatives are known, records them, and accumulates the gradients from the output side using the chain rule. It carries out the backpropagation of 04 mechanically, one scalar at a time.
3. The model processes one token at a time, appending keys and values to lists. This

makes a causal mask unnecessary and, at the same time, takes the form of a KV cache.

4. For each name, the cross-entropy loss is computed and the parameters are updated with Adam. Over 1000 steps the loss drops from $\ln 27 \approx 3.30$ to about 2.3, and random strings turn into name-like strings.
5. The differences from real LLMs lie mainly in scale and in the way the computation is done; the framework of the model and the principles of learning are the same.

Appendix A Python code: a minimal example of automatic differentiation (standard library only)

This program computes the example of the math note in Section 3.2 with a small automatic differentiation 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 the purpose of 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 B How the numbers and figures in the text were obtained

The numbers in Sections 6.3 and 7.3 and Figures 3 and 4 were obtained by making the following changes to the original program. The model and training parts were not modified.

- **Recording the loss (Figure 3)**: Instead of the progress display on line 184, the `loss.data` of each step was saved to a list.
- **Generation during training (table in Section 6.3)**: 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. Even so, the 20 names generated after training (Section 7.2) agreed with those from running the original program unchanged.
- **Internal values (table in Section 7.3 and Figure 4)**: The values of the parameters after training were saved to a file, and with a program that rewrites the same computation as `gpt()` using ordinary floating-point numbers, the probabilities at each position and the attention weights when `emma` is input were computed.