# microgpt を読む

200 行の Python で GPT の学習と生成をたどる　理系学生のための整理ノート

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

## はじめに

**microgpt** は，Andrej Karpathy が 2026 年 2 月に公開した，GPT の学習と生成を行う約 200 行の Python プログラムです。使っているのは標準ライブラリの `os`・`math`・`random` だけで，NumPy も PyTorch も使いません。データの読み込み，トークナイザ，自動微分，Transformer 本体，最適化，文章の生成までが 1 つのファイルに入っています。

このノートは，Karpathy 自身の解説記事の翻訳ではなく，**プログラムそのものを上から順に読み，これまでの整理ノート（01〜09）の言葉で説明し直したもの**です。本文中の数値は，すべて実際にプログラムを実行して確かめたものです。元のプログラムは次の場所にあります。

- プログラム本体：https://gist.github.com/karpathy/8627fe009c40f57531cb18360106ce95
- 作者による解説：https://karpathy.github.io/2026/02/12/microgpt/

本文で「 行目」と書くのは，2026 年 9 月時点のこのファイル（200 行）の行番号です。コードは説明に必要な数行だけを引用します。プログラム全体を手元に開いて，並べて読むことを想定しています。

**動かし方**：Python 3 が入っていれば，`python microgpt.py` で動きます（Windows でも同じ）。初回だけ，学習用の人名データ（`input.txt`）をインターネットから取得して同じフォルダに保存します。2 回目以降はそのファイルを使うので，ネットワークは不要です。このノートの作成に使った計算機（Python 3.11）では，学習 1000 ステップと生成 20 件で約 2 分 40 秒かかりました。

**記号の約束**

- ベクトルは縦ベクトルで，太字 $\mathbf{x}$ で表します。$x_i$ はその第 $i$ 成分です。
- 右肩の $\mathsf{T}$ は**転置**で，べき乗ではありません。$\mathbf{q}^{\mathsf{T}}\mathbf{k}$ は内積です。
- $d$ は埋め込みの次元（microgpt では $d = 16$），$h$ はヘッドの数（$h = 4$），$d_k = d/h = 4$ は 1 つのヘッドの次元です。
- $V$（イタリック）は語彙の大きさ（$V = 27$）です。プログラム中のバリューのリスト `values` とは別物です。
- $p(\cdot)$ は確率，ln は自然対数です。$L$ は損失を表します。
- $W$ に添字を付けたもの（$W_Q$ など）は重み行列です。microgpt では行列を「行のリスト」として持ち，$\mathbf{y} = W\mathbf{x}$ の形で左から掛けます（06 の $Q = XW_Q$ とは掛ける向きが逆ですが，中身は同じ計算です）。

## 1　全体の見取り図

プログラムは，上から順に次の 6 つの部分からなります（図 1）。

| 部分 | 行 | 内容 |
|---|---|---|
| データ | 14–21 | 人名 32,033 件を読み込み，順番をシャッフルする |
| トークナイザ | 23–27 | 文字を番号に変換する規則を作る |

| 部分 | 行 | 内容 |
|---|---|---|
| 自動微分 | 29–72 | 計算の履歴を記録し，連鎖律で勾配を求める `Value` クラス |
| パラメータとモデル | 74–144 | 4,192 個のパラメータと，Transformer 本体 `gpt()` |
| 学習 | 146–184 | 損失を計算し，勾配を求め，Adam でパラメータを更新する |
| 生成 | 186–200 | 学習したモデルで新しい名前を作る |

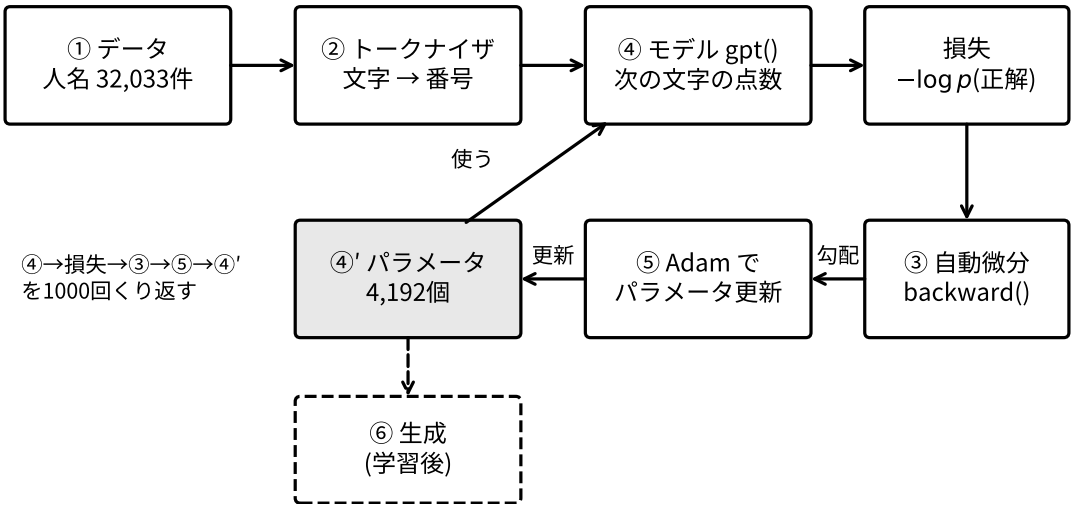



図 1　microgpt の構成。学習では　→ 損失 → →　の流れを 1000 回くり返し，パラメータを少しずつ変えていく。学習が終わったら，同じモデルとパラメータで　の生成を行う。

ファイル冒頭の説明文には，「このファイルがアルゴリズムのすべてであり，それ以外はすべて効率化にすぎない」という趣旨のことが書かれています。実際の LLM との違いは主に**規模と計算速度**であって，**計算の中身**はこの 200 行と同じだ，という主張です（8 節で具体的に比べます）。

## 2　データとトークナイザ

### 2.1　データ

学習に使うのは，米国の人名 32,033 件です（`emma`，`olivia` など，1 行に 1 つ）。**1 つの名前を 1 つの「文書」**とみなします。LLM が大量の文章から「次の単語」を学ぶのと同じように，microgpt は大量の名前から「次の文字」を学びます。

### 2.2　文字単位のトークナイザ

24–26 行目で，データに現れる文字（英小文字 a〜z の 26 種類）に 0〜25 の番号を付けます。これに，**文書の始まりと終わりを表す特別な記号 BOS**（26 番）を加え，語彙の大きさは $V = 27$ です。

名前 `emma` は，学習のときに次の番号の列に変換されます（157 行目）。

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

先頭の BOS は「ここから名前が始まる」という合図，末尾の BOS は「ここで名前が終わる」という正解です。モデルは「BOS の次は e」「BOS, e の次は m」……「BOS, e, m, m, a の次は BOS（終わり）」という 5 つの予測問題を同時に解くことになります。

07 で見た BPE のように複数の文字をまとめる工夫はしていません。**1 文字= 1 トークン**という最も単純なトークナイザです。

# 3 自動微分：Value クラス

## 3.1 何をするのか

学習には，損失 $L$ をすべてのパラメータ $w$ で偏微分した $\partial L/\partial w$（勾配）が必要です。04 では，誤差逆伝播法を層ごとの行列の式として導きました。microgpt はこれを **1 つ 1 つの数（スカラー）の単位**で行います。

`Value` クラスの 1 つのオブジェクトは，1 つの数を表し，次の 4 つを覚えています（33–37 行目）。

| 属性 | 意味 |
|---|---|
| `data` | その数の値（順伝播で計算される） |
| `grad` | 損失をその数で偏微分した値 $\partial L/\partial$(この数)（逆伝播で計算される） |
| `_children` | その数を作るのに使った数（計算の入力） |
| `_local_grads` | その数を，それぞれの入力で偏微分した値（**局所微分**） |

例えば掛け算 $c = a \times b$ を行うと，新しい `Value` が作られ，入力として $a$，$b$ を，局所微分として $\partial c/\partial a = b$，$\partial c/\partial b = a$ を記録します（43–45 行目）。

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

足し算，べき乗，`exp`，`log`，`relu` も同じように，値と局所微分を記録します（39–50 行目）。引き算や割り算は，足し算・掛け算・べき乗の組み合わせとして定義されています（51–57 行目）。つまり，**microgpt のすべての計算は，局所微分の分かっている 6 種類の基本演算の組み合わせ**です。

計算を進めると，どの数がどの数から作られたかという**計算グラフ**が自動的にでき上がります。

## 3.2 逆伝播

`backward()`（59–72 行目）は，計算グラフを出力側から入力側へたどり，**連鎖律**で勾配を求めます。

1. グラフのすべての数を，「入力が必ず先に来る」順番（トポロジカル順）に並べる。
2. 出力（損失）自身の `grad` を 1 とする（$\partial L/\partial L = 1$）。
3. 並べた順番を**逆に**たどり，各数 $v$ について，その入力 $u$ に「局所微分 $\times$ $v$ の `grad`」を**足し込む**。

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

「足し込む」のは，同じ数が複数の計算に使われていると，それぞれの経路からの寄与の**和**が偏微分になるからです。

> **数式補足：小さな例**
> $f = a \times b + a$ を，$a = 2$，$b = 3$ で計算します（図 2）。途中の数を $c = a \times b = 6$ とすると $f = c + a = 8$ です。
> 逆伝播では，まず $\partial f/\partial f = 1$。足し算の局所微分は 1 なので，$c$ の `grad` は 1，$a$ の `grad` に 1 が足されます。次に $c = a \times b$ から，$a$ の `grad` に $b \times 1 = 3$，$b$ の `grad` に $a \times 1 = 2$ が足されます。結果は
>
> $$\frac{\partial f}{\partial a} = 1 + 3 = 4, \qquad \frac{\partial f}{\partial b} = 2$$
>
> で，$f = ab + a$ を直接偏微分した $b + 1 = 4$，$a = 2$ と一致します。$a$ は 2 か所で使われているので，2

つの経路からの寄与 1 と 3 が足し合わされています。付録 A のプログラムで，数値微分と一致することを確かめています。

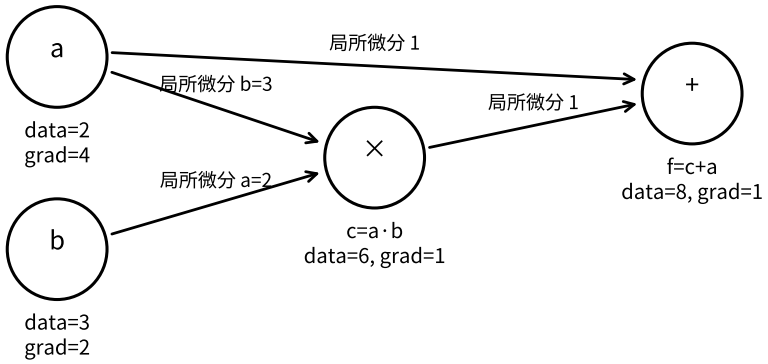



図 2　$f = a \times b + a$ の計算グラフ。矢印の脇の数が局所微分，丸の下の grad が逆伝播で求めた偏微分。

microgpt の 1 回の学習ステップでは，この計算グラフが**数万個の数**からなる大きなものになりますが，やっていることはこの小さな例と同じです。PyTorch などの自動微分も，原理は同じで，スカラーではなく行列（テンソル）の単位で記録するところが違います。

# 4　パラメータ

## 4.1　大きさの設定

75–79 行目で，モデルの大きさを決めます。

| 変数 | 値 | 意味 |
|---|---|---|
| `n_layer` | 1 | Transformer ブロックの層の数 |
| `n_embd` | 16 | 埋め込みの次元 $d$ |
| `block_size` | 16 | 扱える最大の長さ（最も長い名前は 15 文字） |
| `n_head` | 4 | ヘッドの数 $h$ |
| `head_dim` | 4 | 1 つのヘッドの次元 $d_k = d/h$ |

## 4.2　パラメータの一覧

パラメータはすべて，平均 0，標準偏差 0.08 の正規乱数で初期化されます（80 行目）。これが学習前のモデルの「白紙」の状態です。

| 名前 | 大きさ | 数 | 役割 |
|---|---|---|---|
| `wte` | $27 \times 16$ | 432 | トークン（文字）の埋め込み |
| `wpe` | $16 \times 16$ | 256 | 位置の埋め込み |
| `attn_wq, wk, wv, wo` | $16 \times 16$ を 4 つ | 1,024 | 注意の $W_Q$, $W_K$, $W_V$, $W_O$ |
| `mlp_fc1` | $64 \times 16$ | 1,024 | MLP の 1 層目（$d \to 4d$） |
| `mlp_fc2` | $16 \times 64$ | 1,024 | MLP の 2 層目（$4d \to d$） |
| `lm_head` | $27 \times 16$ | 432 | 出力層（ベクトル $\to$ 27 文字の点数） |
| 合計 | | **4,192** | |

06 の 4.3 節の「1 ブロックあたり $12d^2$」に当てはめると $12 \times 16^2 = 3{,}072$ で，表の注意と MLP の合計 $1{,}024 + 2{,}048$ と一致します。残りの 1,120 個が埋め込みと出力層です。

# 5　モデル本体：gpt() 関数

## 5.1　1 トークンずつ処理する

108–144 行目の `gpt(token_id, pos_id, keys, values)` がモデル本体です。06 で見た Transformer は，文のすべてのトークンを行列 $X$ にまとめて一度に処理しましたが，microgpt の `gpt()` は **1 回の呼び出しで 1 つのトークンだけ**を処理し，そのトークンの「次の文字」の点数（ロジット）27 個を返します。

1 つのトークンの処理は，次の順に進みます。

1. 埋め込み（109–112 行目）
2. 注意ブロック（115–134 行目）
3. MLP ブロック（135–141 行目）
4. 出力層（143 行目）

以下，順に見ます。

## 5.2　埋め込み

文字の番号に対応する `wte` の行（16 次元のベクトル）と，位置の番号に対応する `wpe` の行を取り出して足します。

$$\mathbf{x} = \mathbf{e}_{\text{文字}} + \mathbf{p}_{\text{位置}}$$

06 の 3.5 節で見たように，Attention 自体は順番を区別できないので，位置の情報をこうして外から加えます。06 で紹介した正弦波の位置符号とは違い，GPT-2 と同じく，**位置のベクトルも学習するパラメータ**です。

## 5.3　RMSNorm

各ブロックの入口で，ベクトルの大きさをそろえる**正規化**を行います（103–106 行目）。

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

成分の 2 乗平均の平方根（RMS）で割って，RMS を約 1 にします。06 の図 6 の「Norm」にあたる部分で，GPT-2 の LayerNorm（平均を引いてから標準偏差で割る）を簡単にしたものです。

## 5.4　注意ブロック

まず，正規化したベクトルからクエリ・キー・バリューを作ります（118–120 行目）。

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

ここで大事なのが次の 2 行です（121–122 行目）。

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

今のトークンのキーとバリューを，**それまでのトークンのキー・バリューのリストの末尾に追加**しています。したがって，位置 $t$ の処理をしている時点で，リストには位置 0 から $t$ までのキーとバリューしかありません。

16 次元のベクトルを 4 次元ずつ 4 つに分け，ヘッドごとに注意を計算します（124–132 行目）。ヘッド $m$ のクエリを $\mathbf{q}^{(m)}$，位置 $s$ のキー・バリューを $\mathbf{k}_s^{(m)}$，$\mathbf{v}_s^{(m)}$ とすると，

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

です。06 の Scaled Dot-Product Attention の式を，1 行（1 つのクエリ）分だけ書いたものです。4 つのヘッドの出力（各 4 次元）をつないで 16 次元に戻し，$W_O$ を掛けます（133 行目）。最後に，ブロックの入口のベクトルを足します（134 行目，**残差接続**）。

> **数式補足：因果マスクがいらない理由**
> 06 の 3.4 節では，未来のトークンを見ないように，点数の行列に $-\infty$ を加える因果マスクを使いました。microgpt にはマスクの処理がありません。
> トークンを 1 つずつ順に処理し，キーとバリューを末尾に追加していくので，位置 $t$ の処理のときには，**未来のキーとバリューがまだ存在しない**からです。$s$ の和が 0 から $t$ までしかないことが，そのまま因果マスクの役割を果たしています。
> また，この「過去のキーとバリューを保存しておき，新しいトークンの分だけ追加する」仕組みは，09 の 5 節で見た **KV キャッシュ**そのものです。実際の LLM では生成を速くするための工夫として後から加えるものですが，microgpt では最初からこの形で書かれています。

### 5.5 MLP ブロック

注意ブロックの出力を正規化し，16 次元 → 64 次元 → 16 次元の 2 層の全結合ネットワークに通します（137–140 行目）。活性化関数は ReLU（負なら 0，正ならそのまま）です。ここでも入口のベクトルを足します（141 行目，残差接続）。

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

06 の 4.2 節の FFN にあたる部分です。注意ブロックが**トークン間の情報交換**，MLP ブロックが**トークンごとの変換**を担うという役割分担も，06 と同じです。

### 5.6 出力層

最後に `lm_head`（$27 \times 16$）を掛けて，27 文字それぞれの点数（ロジット）を得ます（143 行目）。これを softmax 関数に通したものが，**次の文字の確率分布**です。

## 6 学習

### 6.1 損失

学習の 1 ステップでは，**名前を 1 つだけ**取り出し（156 行目），その各位置で次の文字を予測させます（163–168 行目）。位置 $t$ での正解の文字の確率を $p_t$ として，損失はその負の対数の平均です（169 行目）。

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

07 などで見た交差エントロピー損失と同じものです。学習前のパラメータは乱数なので，27 文字をほぼ等確率で予測し，$L \approx \ln 27 \approx 3.30$ になるはずです。実際に，1 ステップ目の損失は 3.37 でした。

この損失の計算はすべて `Value` の演算で行われるので，計算の終わりには，4,192 個のパラメータから損失までの計算グラフができ上がっています。172 行目の `loss.backward()` 1 回で，すべてのパラメータの勾配が求まります。

## 6.2 Adam によるパラメータの更新

174–182 行目で，勾配を使ってパラメータを更新します。04 で見た勾配降下法 $w \leftarrow w - \eta\, g$（$g$ は勾配，$\eta$ は学習率）を改良した **Adam** という方法です。

> **数式補足：Adam**
> パラメータ $w$ ごとに，勾配 $g$ の**移動平均** $m$ と，勾配の 2 乗の移動平均 $v$ を持ちます（プログラムの `m`, `v`。記号の約束の $V$ とは別物です）。$k$ ステップ目（$k = 1, 2, \ldots$）に
>
> $$m \leftarrow \beta_1 m + (1-\beta_1)\, g, \qquad v \leftarrow \beta_2 v + (1-\beta_2)\, g^2$$
>
> と更新し，最初のうち $m$, $v$ が 0 に偏っている分を $\hat{m} = m/(1-\beta_1^k)$, $\hat{v} = v/(1-\beta_2^k)$ で補正してから，
>
> $$w \leftarrow w - \eta\, \frac{\hat{m}}{\sqrt{\hat{v}} + \epsilon}$$
>
> とします。microgpt では $\beta_1 = 0.85$, $\beta_2 = 0.99$, $\epsilon = 10^{-8}$ です。
> 移動平均 $m$ を使うことで，1 つの名前ごとにばらつく勾配がならされます。また $\sqrt{\hat{v}}$ で割ることで，勾配が大きいパラメータも小さいパラメータも，だいたい同じくらいの幅で動くようになります。

学習率 $\eta$ は 0.01 から始めて，ステップが進むにつれて直線的に 0 まで下げていきます（175 行目）。最後に各パラメータの `grad` を 0 に戻して（182 行目），次のステップに進みます。

## 6.3 学習の経過

この 1 ステップを，名前を替えながら 1000 回くり返します。損失の変化を図 3 に示します。

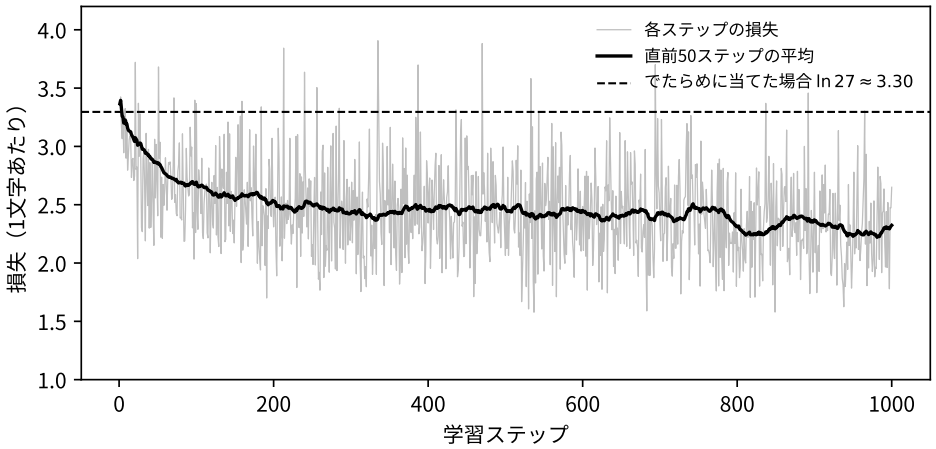



図 3　損失の変化。灰色は各ステップの損失（そのステップで使った 1 つの名前についての値），黒線は直前 50 ステップの平均。

最初の 50 ステップの平均は 2.85，最後の 100 ステップの平均は 2.28 でした。1 ステップに名前を 1 つしか使わないので，各ステップの損失は名前によって大きくばらつきます（灰色の線）。1000 ステップで使う名前は 1000 個で，データ全体の約 3% にすぎません。

学習の途中のモデルに名前を生成させると，次のように変わっていきました（7 節と同じ方法，温度 0.5，同じ乱数の種で 10 個ずつ）。

| 学習ステップ | 生成された名前（最初の 5 個） |
|---|---|
| 0（学習前） | jdodmicpalbdjxcj, pzol, bwhechtepsnqbdeq, liomhsugpozri, ckufnbqwpykuntmx |

| 学習ステップ | 生成された名前（最初の 5 個） |
|---|---|
| 100 | airayne, areen, aln, en, aveeer |
| 300 | faran, ananan, male, kayay, bameh |
| 1000 | caran, ananan, nail, kaya, alan |

学習前は文字がでたらめに並び，終わりの BOS もなかなか出ないので，最大の 16 文字まで続くものが多くあります。100 ステップで母音と子音が交互に並ぶようになり，300 ステップ以降は「名前らしい」長さと響きになっています。

# 7　生成

## 7.1　仕組み

学習が終わったら，同じ `gpt()` 関数で新しい名前を作ります（186–200 行目）。

1. BOS から始める。
2. `gpt()` で次の文字の点数を求め，**温度** $T$ で割ってから softmax をとり，確率分布を得る（195 行目）。
3. その確率に従って文字を 1 つ**くじ引きで選ぶ**（196 行目）。
4. 選んだ文字が BOS なら終わり。そうでなければ，その文字を次の入力にして 2 に戻る。

温度は $T = 0.5$ に設定されています。$T < 1$ で点数を割ると点数の差が広がり，確率の高い文字がより選ばれやすくなります（「LLM_ 確率と温度」の資料を参照）。

学習の損失の計算（6.1 節）では各位置の予測を並べて処理しましたが，生成では**選んだ文字を次の入力にする**ので，1 文字ずつ順にしか進めません。これも 06 の 4.4 節で見た学習と生成の違いと同じです。

## 7.2　生成結果

学習 1000 ステップの後に生成された 20 個の名前は次のとおりです（元のプログラムのまま実行）。

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

`ann`，`anna`，`lara`，`anton` のように実在する名前もありますが，`kamon` や `vialan` のように，**データにない，しかし名前らしいもの**が多く含まれます。モデルは名前を丸暗記したのではなく，「名前に現れやすい文字の並び方」の統計を学んでいます。データにないものを，それらしく作り出すという点で，09 の 6 節で見た**ハルシネーション**と同じ仕組みです。

## 7.3　モデルの内部をのぞく

学習後のモデルに `emma` を 1 文字ずつ入れたとき，各位置で予測された「次の文字」の確率の上位を示します（付録 B の方法で計算）。

| 入力 | 次の文字の確率（上位 5 つ） |
|---|---|
| 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（終わり）0.166**，r 0.103，h 0.068，l 0.061 |

名前の最初の文字は a が最も多く，`em` の次は母音，`emma` の次は「n が続く（emman…）」か「ここで終わる」が上位です。正解の文字（m, m, a, 終わり）が必ずしも 1 位ではありませんが，上位に入っています。

同じときの，4 つのヘッドの注意の重みを図 4 に示します。

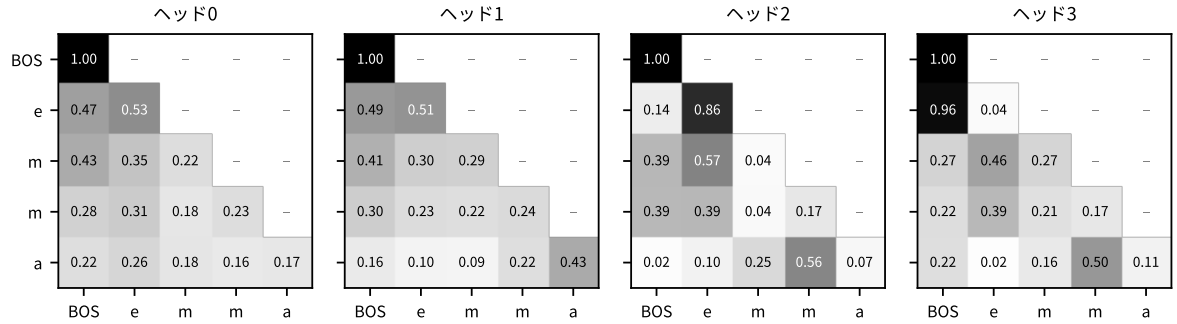



図 4　`emma` を入力したときの注意の重み（学習後）。行が今処理しているトークン（クエリ），列が参照されるトークン（キー），各行の和は 1。右上の「–」はまだ存在しない未来の位置（5.4 節の数式補足）。

右上が空いているのは，未来のキーがまだないためです。ヘッドによって重みの付け方が異なることが分かります（例えば最後の `a` の行では，ヘッド 1 は自分自身に，ヘッド 2 とヘッド 3 は 2 つ目の `m` に大きな重みを置いています）。ただし，それぞれのヘッドが何を捉えているのかを，この図から読み取るのは簡単ではありません（「なぜ分からないのにうまく行くのか」のノートを参照）。

# 8　本物の GPT との比較

## 8.1　規模

| | microgpt | GPT-2（最小） | GPT-3 |
|---|---|---|---|
| 埋め込みの次元 $d$ | 16 | 768 | 12,288 |
| 層の数 | 1 | 12 | 96 |
| ヘッドの数 | 4 | 12 | 96 |
| 扱える長さ | 16 | 1,024 | 2,048 |
| 語彙の大きさ | 27 | 50,257 | 50,257 |
| パラメータ数 | 4,192 | 約 1.24 億 | 約 1,750 億 |

GPT-3 のパラメータ数は microgpt の約 4,000 万倍です。

## 8.2　計算の中身の違い

プログラムの 93 行目の注釈にあるとおり，microgpt のモデルは GPT-2 とほぼ同じ構造で，違いは次の 3 点です。

- LayerNorm の代わりに RMSNorm を使う（5.3 節）。
- 全結合層にバイアス（定数項）がない。
- MLP の活性化関数が GELU ではなく ReLU。

これ以外の違いは，計算の中身ではなく**計算のしかた**にあります。

| | microgpt | 実際の LLM |
|---|---|---|
| トークン | 1 文字 | BPE などで作った部分語（07） |
| 1 ステップで使うデータ | 名前 1 つ | 数百万トークン（多数の文書をまとめて処理） |
| 計算の単位 | 1 つの数（スカラー）ごと | 行列・テンソル単位でまとめて |
| 計算機 | CPU 1 つ，Python で 1 つずつ | 多数の GPU で並列に |

| | microgpt | 実際の LLM |
|---|---|---|
| 学習後の調整 | なし | 指示調整・RLHF など（09） |

microgpt では 1 ステップの学習に 0.1 秒以上かかりますが，これは Python で数を 1 つずつ計算しているためです。実際の LLM は同じ種類の計算を GPU で大量に並列に行います。また，最近のモデルは，位置の扱い（RoPE）や注意の計算の省力化など，さらにいくつかの工夫を加えていますが，**「埋め込み → 注意と MLP の繰り返し → 次のトークンの確率」という骨組みと，「交差エントロピー損失を勾配で下げる」という学習の原理は同じ**です。

## 9　まとめ

1. microgpt は，データ・トークナイザ・自動微分・Transformer・Adam・生成を，標準ライブラリだけの約 200 行で書いた GPT である。
2. 自動微分は，すべての計算を局所微分の分かった基本演算に分解して記録し，出力側から連鎖律で勾配を足し込んでいく。04 の誤差逆伝播法を，スカラー単位で機械的に行うものである。
3. モデルは 1 トークンずつ処理し，キーとバリューをリストに追加していく。これにより因果マスクが不要になり，同時に KV キャッシュの形にもなっている。
4. 名前 1 つごとに交差エントロピー損失を求め，Adam でパラメータを更新する。1000 ステップで損失は $\ln 27 \approx 3.30$ から約 2.3 まで下がり，でたらめな文字列が名前らしい文字列に変わる。
5. 実際の LLM との違いは主に規模と計算のしかたで，モデルの骨組みと学習の原理は同じである。

## 付録 A　Python コード：自動微分の最小例（標準ライブラリのみ）

3.2 節の数式補足の例を，足し算と掛け算だけの小さな自動微分で計算し，数値微分と比べるプログラムです。後半では，4.2 節のパラメータ数を計算します。microgpt の `Value` クラスを，説明のために最小限にしたものです。

```
# 自動微分の最小例と数値微分による検算（標準ライブラリのみ）
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):                       # 子を先に並べる（トポロジカル順）
            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):           # 出力側から順に連鎖律
            for c, g in zip(v.children, v.local):
                c.grad += g * v.grad        # 同じ変数が 2 回使われると足し合わされる

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                                                    # 数値微分で検算
fn = lambda x, y: x * y + x
print(" 数値微分:", round((fn(2 + h, 3) - fn(2, 3)) / h, 4), round((fn(2, 3 + h) - fn(2,
↪  3)) / h, 4))

# microgpt のパラメータ数の内訳
vocab, d, ctx = 27, 16, 16
parts = {" トークン埋め込み wte": vocab * d, " 位置埋め込み wpe": ctx * d,
         " 注意 Wq,Wk,Wv,Wo": 4 * d * d, "MLP fc1,fc2": 2 * 4 * d * d,
         " 出力層 lm_head": vocab * d}
for k, v in parts.items(): print(f"{k}: {v}")
print(" 合計", sum(parts.values()))
```

実行結果：

```
f = 8.0  df/da = 4.0  df/db = 2.0
数値微分: 4.0 2.0
トークン埋め込み wte: 432
位置埋め込み wpe: 256
注意 Wq,Wk,Wv,Wo: 1024
MLP fc1,fc2: 2048
出力層 lm_head: 432
合計 4192
```

## 付録 B　本文の数値と図の求め方

本文の 6.3 節・7.3 節の数値と図 3・図 4 は，元のプログラムに次の変更を加えて求めました。モデルと学習の部分には手を加えていません。

- **損失の記録（図 3）**：184 行目の進行表示の代わりに，各ステップの `loss.data` をリストに保存した。
- **学習途中の生成（6.3 節の表）**：学習ループの中で，ステップ 0，100，300 の開始時と学習終了後に，7 節と同じ方法（温度 0.5）で 10 個ずつ名前を生成した。学習本体の乱数に影響しないよう，生成の前後で乱数の状態を保存・復元し，生成には別の種（7）を使った。こうしても，学習後の 20 個の生成結果（7.2 節）は元のプログラムのままの実行と一致した。
- **内部の値（7.3 節の表と図 4）**：学習後のパラメータの値をファイルに保存し，`gpt()` と同じ計算を普通の浮動小数点数で書き直したプログラムで，`emma` を入力したときの各位置の確率と注意の重みを計算した。