

microgpt を読む（高校生・大学初年次向け）

200行のプログラムで、AIが文章を作るしくみをのぞく

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

Ver. 0.1 October 4, 2026

※ 以下の内容はAIが生成したものであり、一応編者がチェックしていますが、間違いや思い込みが含まれている場合があることはご注意ください。

© 2026 Yoshio Okamoto. 教育目的での利用は自由です（出典を明記してください）。microgpt のプログラムは Andrej Karpathy の著作物です。

はじめに

ChatGPT や Claude のような AI（大規模言語モデル、Large Language Model, LLM）は、実はとても単純なことをくり返しています。それは「**これまでの文章の続きとして、次に来る言葉を予想する**」ことです。1つ予想したらそれを文章に付け足し、また次を予想する。これをくり返すと長い文章ができあがります。

microgpt は、Andrej Karpathy が 2026年2月に公開した、わずか約200行の Python プログラムです。LLM と同じしくみを、ごく小さな規模で動かします。扱うのは文章ではなく英語の人名で、「次の文字」を予想することを学び、最後に新しい名前を作り出します。NumPy や PyTorch のような特別なライブラリは使わず、Python に最初から入っている機能だけで書かれています。

この冊子では、このプログラムが何をしているのかを、できるだけ**文章と図**で説明します。数式は付録Aにまとめたので、本文は数式を読まなくても分かるように書いてあります。本文中の数値は、すべて実際にプログラムを動かして確かめたものです。LLM の説明でよく使われる用語には、最初に出てくるところで英語を括弧で添えました。

内容理解のためには、機械学習から LLM までのしくみを丁寧に解説している 3Blue1BrownJapan の動画シリーズ（YouTube, CG画像でわかりやすい）、ヨビノリたくみ氏の AI 解説動画シリーズ（YouTube, 数式理解に詳しい）を並行してご覧になることをおすすめします。

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

動かし方：Python 3 が入っていれば、ターミナル（Windows ではコマンドプロンプト）で `python microgpt.py` と入力すると動きます。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。筆者らの計算機では、学習と名前の生成を合わせて2〜3分ほどかかりました。

本文で「○行目」と書くのは、このプログラム（200行）の行番号です（入手先と行番号の対応は付録E）。プログラムを手元に開いて、並べて読むと分かりやすいでしょう。

1 全体の流れ

プログラムは、上から順に次の6つの部分でできています（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
自動微分	29-72	「どのつまみをどちらに回せばよいか」を自動で計算するしくみ
パラメータとモデル	74-144	4,192個のつまみ (パラメータ) と, 予想を行うモデル本体
学習	146-184	予想の外れ具合を測り, つまみを少しずつ調整する
生成	186-200	学習したモデルで新しい名前を作る

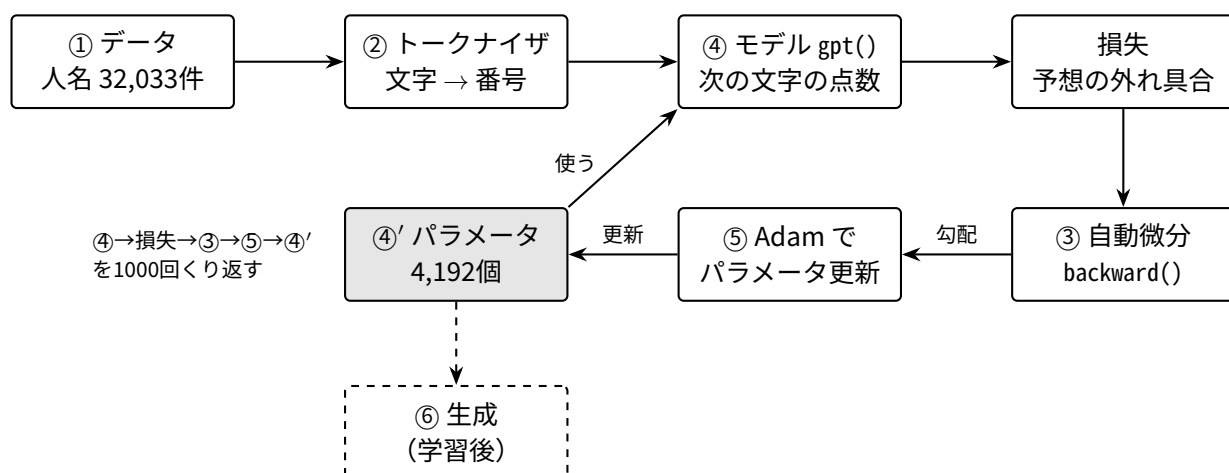


図1 microgpt の構成. 学習では ④ → 損失 → ③ → ⑤ の流れを1000回くり返し, パラメータを少しずつ変えていく. 学習が終わったら, 同じモデルとパラメータで ⑥ の生成を行う.

プログラムの冒頭には, 「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. 本物の LLM との違いは主に**規模と計算の速さ**で, **計算の中身**はこの200行と同じだ, という主張です (8節で比べます).

2 データと文字の番号

2.1 データ

学習に使うのは, アメリカの人名 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」(document)**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 文字に番号をつける

コンピュータは文字をそのまま計算に使えないので, 文字に番号をつけます (24-26行目). データに出てくる英小文字 a~z の26種類に 0~25 の番号をつけ, さらに**名前の始まりと終わりを表す特別な記号 BOS** (Beginning of Sequence, 26番)を加えます. 全部で27種類です. この27種類を, このモデルの**語彙** (vocabulary) と呼びます.

たとえば名前 emma は, 学習のときに次の番号の列に変換されます (157行目).

BOS, e, m, m, a, BOS → 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つの予想問題を解くことになります。

このように、文章を細かく区切った1つ1つを**トークン（token）**と呼び、文章をトークンに区切る規則を**トークナイザ（tokenizer）**と呼びます。microgpt では**1文字=1トークン**です¹。

3 モデルの中身：次の文字に点数をつける

3.1 1文字ずつ順に処理する

108-144行目の `gpt()` という関数がモデル（model）本体です。この関数は、**1回の呼び出しで1つの文字だけ**を受け取り、「次に来る文字」として27種類それぞれに点数をつけて返します。処理は図2の順に進みます。以下、1段ずつ見ていきます。

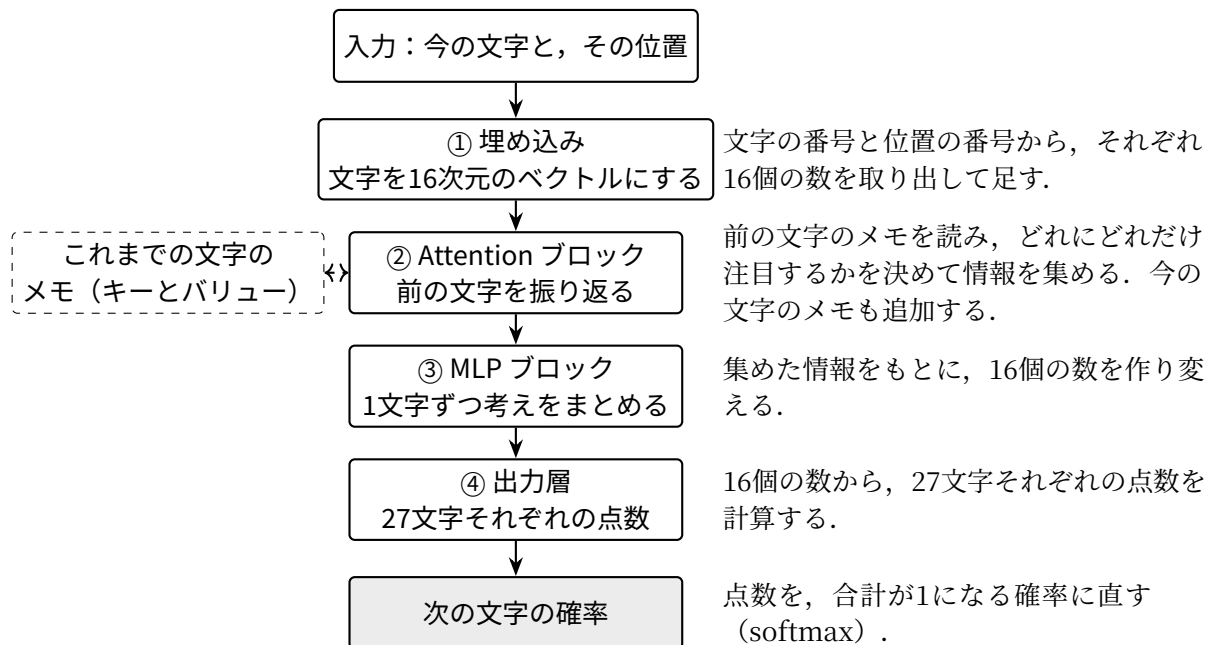


図2 `gpt()` が1つの文字を処理する流れ。Attention ブロックだけが、前の文字の情報（メモ）を使う。

3.2 文字を数の並びにする（埋め込み）

まず、文字を**16個の数の並び**（16次元のベクトル，vector）に置き換えます²。そのために、プログラムは2つの表を持っています。

¹本物の LLM では、よく出てくる文字の組み合わせ（たとえば“ing”や“the”）を1つのトークンにまとめる方法（BPE：バイト対符号化，Byte Pair Encoding など）が使われます。こうすると、同じ長さの文章をより少ないトークンで表せます。

²16という数は、表現力と計算量の兼ね合いで作者が決めた設計上の値です（このような値を、学習で決まるパラメータと区別してハイパーパラメータ，hyperparameter と呼びます）。16でなければならない理由はありませんが、後で説明するヘッドの数（4）で割り切れる必要があります。本物の LLM では、GPT-2 で768，GPT-3 で12,288 と、はるかに大きな値が使われています。

- ・ **文字の表**（プログラムでの名前は `wte`）：27行×16列の表で、27種類の文字それぞれに、1行ずつ16個の数が割り当てられています。たとえば文字 `e`（4番）には、表の5行目の16個の数が対応します。
- ・ **位置の表**（`wpe`）：16行×16列の表で、「名前の何文字目か」という位置ごとに16個の数が割り当てられています。

文字の番号の行と、位置の番号の行を取り出して足し合わせたものが、その文字の出発点になります（109-112行目）。この処理を**埋め込み**（`embedding`）と呼びます³。

表の数は、最初はでたらめな値です。学習が進むと、似た使われ方をする文字は似た数の並びになっていきます（5.5節で実際の例を見ます）。16個の数は、その文字の「特徴を書いた名札」のようなものだと考えるとよいでしょう。

位置の数を足すのは、次に説明する Attention のしくみが、それだけでは文字の順番を区別できないからです⁴。

3.3 前の文字を振り返る（Attention）

名前の次の文字を当てるには、今の文字だけでなく、それより前の文字も見する必要があります。たとえば `emma`, `ava`, `mia`, `anna` の4つの名前だけを考えてみましょう。今の文字が `a` のとき、次に来る文字は次のようになります。

- ・ 名前が `a` で始まったばかり（`ava` や `anna` の最初の `a`）なら、次は `v` か `n`
- ・ `m`, `i` と来た後の `a`（`mia` の最後の `a`）なら、次は終わり（`BOS`）

今の文字が同じ `a` でも、それより前に何があったかで答えが変わるのです。この「振り返り」を担うのが **Attention**（注意機構, `attention mechanism`）のしくみです（115-134行目）⁵。

Attention では、各文字の16個の数から、次の3種類の数の並び（ベクトル）を作ります（118-120行目）。

- ・ **クエリ**（`query`, 質問）：今の文字が「どんな情報を探しているか」
- ・ **キー**（`key`, 見出し）：その文字が「どんな情報を持っているか」の目印
- ・ **バリュー**（`value`, 中身）：その文字が実際に渡す情報

今の文字のクエリと、前の各文字のキーを比べ、よく合うものほど大きな重み（`attention weight`, 注目の度合い）をつけます。重みの合計は1です。そして、その重みで各文字のバリューを混ぜ合わせたものが、振り返りの結果になります。図書館で、自分の質問（クエリ）を本の見出し（キー）と照らし合わせ、合う本ほどよく読んで中身（バリュー）を集める、というイメージです。

メモが1つずつ増えていく 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)
```

文字は1つずつ順に処理され、そのたびに今の文字のキーとバリューがメモの末尾に追加されます（図3）。たとえば `emma` の2つ目の `m` を処理しているとき、メモにあるのは `BOS`, `e`, `m` と自分（`m`）の分だけで、次の `a` はまだ処理されていないのでメモにありません。今の文字

³`wte` は word token embedding（トークンの埋め込み）、`wpe` は word position embedding（位置の埋め込み）の略で、GPT-2 のプログラムの名前の付け方を受け継いでいます。

⁴Attention は、前の文字すべてを「袋に入れて」見渡すような計算なので、順番の情報を別に与える必要があります。microgpt では、位置の表の数も学習で決まるパラメータです。GPT-2 も同じ方式です。

⁵「振り返る」はこの冊子での言い方です。一般の LLM の説明では、このしくみを Attention（注意機構）と呼び、「どのトークンに注目するか」「注意を向ける（英語では `attend to`）」と表現します。また、前のトークンだけを見て未来を見ないことを「因果的（`causal`）」と言います。

が振り返れるのはメモにある文字だけなので、未来の文字は、見ようとしても見られないのです⁶。

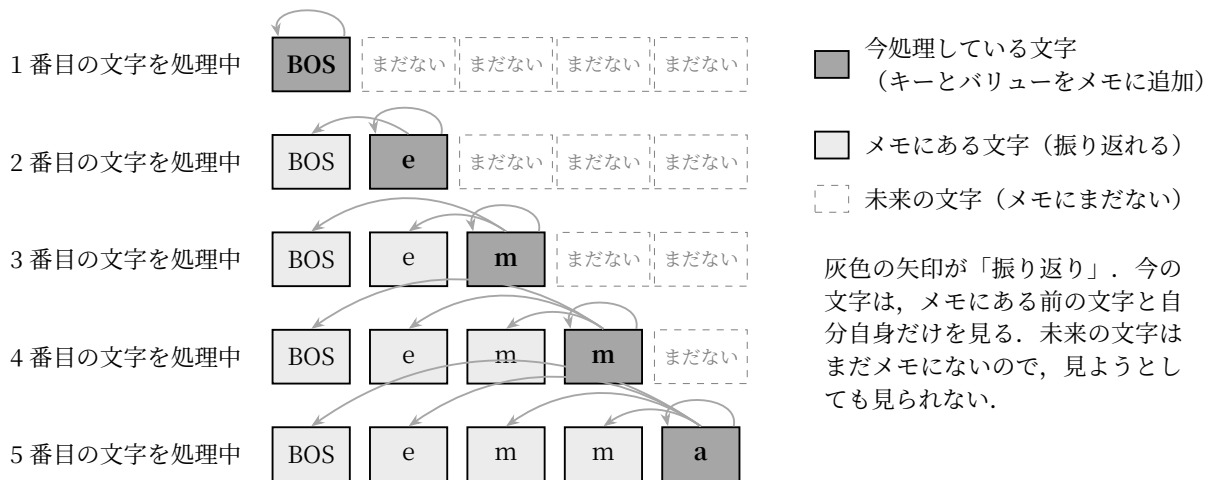


図3 emma を処理するとき、メモ（キーとバリュー）が1つずつ増えていく様子。

未来を見ないことは、とても大切です。学習のときには名前全体が分かっていますが、もし次の文字が見えてしまえば、モデルはそれを写すだけで正解できてしまい、何も学べません。名前を作るとき（7節）には未来の文字はまだ存在しないので、学習のときも同じ条件で練習させる必要があるのです。

4つのヘッドで別々に振り返る 1回の振り返りでは、「どの文字にどれだけ注目するか」という重みの付け方を1通りしか決められません。しかし、次の文字を当てるには、「直前の文字は何か」「名前の最初の文字は何か」など、いくつもの観点から前の文字を見たいことがあります。

そこで microgpt では、16個の数を4個ずつに分け、**4つのグループ（ヘッド、head）がそれぞれ別々に振り返ります**（図4、124–132行目）。同じメモを4人が別々の観点で読むようなものです。各ヘッドは自分の観点で重みを決めて、4個ずつの結果を出します。最後に4つの結果をつなぎ直して16個に戻し、もう一度変換して混ぜ合わせます（133行目）。このしくみを**マルチヘッド Attention（multi-head attention）**と呼びます。

実際に各ヘッドがどんな観点を受け持つかは、人間が決めるのではなく、学習で自然に決まります。7.3節の図15では、同じ emma に対して4つのヘッドがそれぞれ違う重みの付け方をしている様子が見られます。

3.4 1文字ずつ考えをまとめる（MLP）

MLP とは MLP は **Multi-Layer Perceptron（多層パーセプトロン）** の略です。パーセプトロン（perceptron）は、1958年にローゼンブラットが考えた、脳の神経細胞（ニューロン、neuron）をまねた計算のしくみです。いくつかの入力にそれぞれ重みを掛けて足し合わせ、その合計の大きさに応じて「反応する・しない」を決めます。これを何段か重ねたものが

⁶文のすべてのトークンをまとめて一度に計算する通常の Transformer では、未来のトークンを見ないように「マスク」（causal mask）という処理で隠す必要があります。microgpt は1文字ずつ処理するので、マスクが要りません。また、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しくみは、本物の LLM で生成を速くするために使われる「KV キャッシュ」（KV cache）と同じものです。

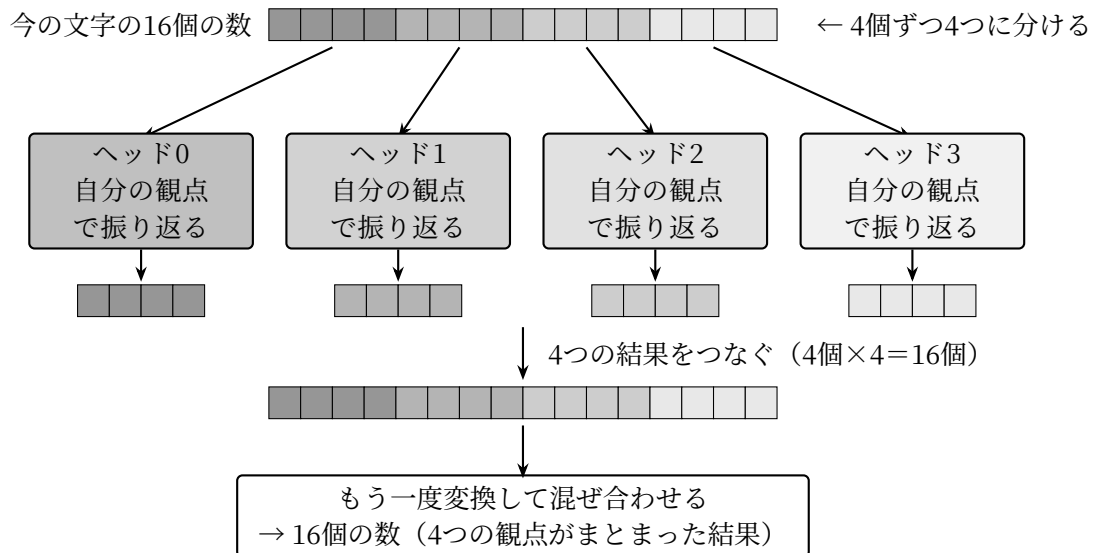


図4 マルチヘッド Attention. 16個の数を4つのヘッドに分け、それぞれが別々に振り返った結果をつなぎ直す。

「多層」パーセプトロンで、ニューラルネットワーク (neural network) の最も基本的な形です。Transformer の説明では、同じものを **Feed-Forward Network** (FFN, 順伝播ネットワーク) と呼ぶこともあります。

MLP がしていること MLP は、次の3段の計算です (137-140行目)。図5は、学習後のモデルに emma を入れ、最後の a を処理したときの実際の値です。

1. **1段目の変換 (16個→64個)** : 64個の「検出器」が、16個の数 (16次元のベクトル) を見て、それぞれ自分の反応の強さを計算します。検出器ごとに重みが違うので、それぞれが入力の別々のパターンに反応します。
2. **ReLU (Rectified Linear Unit)** : 反応が負の数なら0にします。つまり、パターンに合わない検出器は「反応しない」ことになります。図の例では、64個のうち反応したのは9個だけでした。300個の名前で調べると、1文字あたり平均で約8個です。どの検出器が反応するかは文字や文脈によって違い、64個すべてが、どこかの場面で反応していました。
3. **2段目の変換 (64個→16個)** : 反応した検出器が、それぞれ決まった情報を16個の数に書き足します。

最後に、この16個を MLP の入口の16個に足し戻します (141行目)。「元の情報を残しつつ、新しく分かったことを書き足す」ためのしくみで、**残差接続 (residual connection)** と呼ばれます。Attention ブロックの出口でも同じことをしています (134行目)。また、Attention と MLP の入口では、16個の数の大きさをそろえる処理 (**正規化, normalization**) をしています。どちらも学習を安定させる効果があります。

Attention との役割分担 Attention は、前の文字のメモを読んで情報を集める、**文字どうしの情報のやりとり**です。一方 MLP は、集めた情報をもとに**1つの文字の中で考えをまとめる**計算で、他の文字は見ません。どの文字にも同じ MLP が使われます。会議にたとえるなら、Attention は「他の人の意見を聞く」こと、MLP は「聞いた意見をもとに自分で考えをまとめる」ことにあたります。

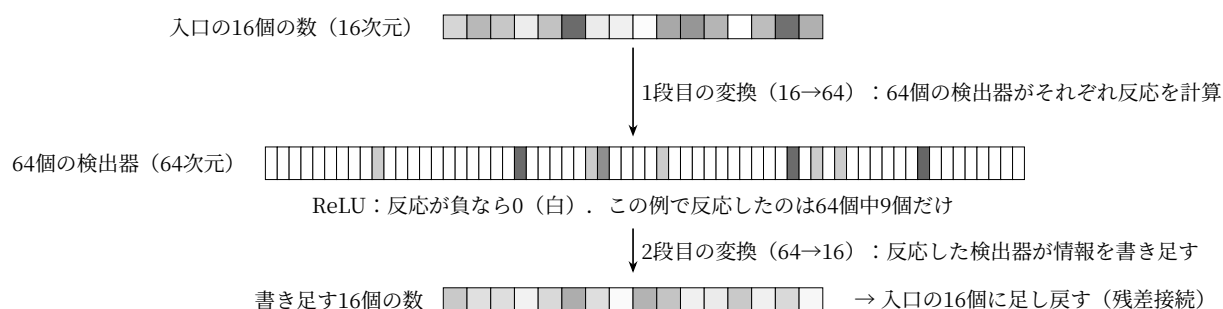


図5 MLP 中の計算（学習後のモデルに emma を入れ、最後の a を処理したときの実際の値）。色の濃さは数の大きさ（絶対値）を表す。

なぜ64個に広げるのか 途中で ReLU をはさむことが大切です。ReLU がなければ、2つの変換は1つの「16→16の変換」にまとめられてしまい、広げる意味がありません。ReLUがあるので、64個はそれぞれ「パターン検出器」として働き、検出器が多いほど、より多くのパターンを区別して扱えます。その代わり、パラメータと計算量は増えます。microgpt では、4,192個のパラメータのうち半分の2,048個が MLP です。4倍（16→64）は数学的に決まった値ではなく、元の Transformer や GPT-2 から受け継がれてきた経験則です（GPT-2 でも768→3072）。最後に16個に戻すのは、入口の16個に足し戻す（残差接続）ために、同じ個数である必要があるからです⁷。

3.5 点数から確率へ（softmax）

最後に、16個の数から27文字それぞれの点数（27次元のベクトル）を計算します（143行目）。この点数は**ロジット**（logit）とも呼ばれます。点数には負の数もあり、合計にも決まりがないので、そのままでは確率として使えません。そこで **softmax**（ソフトマックス）という計算で、「すべて0以上で、合計がちょうど1」になる確率に直します（図6）。点数の順位はそのまま保たれ、点数の差は確率ではより大きな差になります。こうして得られる27文字の確率が「次の文字の予想」（確率分布，probability distribution）です。softmax の計算の中身は付録A.4にあります。

4 学習のしくみ

4.1 「驚き」で外れ具合を測る（損失）

予想の良し悪しを測るために、microgpt は**正解の文字にどれだけの確率をつけていたか**を見ます。正解に高い確率をつけていれば良い予想、低い確率なら悪い予想です。

これを1つの数で表したものが**損失**（loss）です。損失は、正解につけた確率 p から図7のように決まります。 $p = 1$ （正解を確信していた）なら損失は0、 p が小さいほど損失は大きくなります。「正解を知ったときの驚きの大きさ」と考えると分かりやすいでしょう。学習の1ステップでは名前を1つ取り出し、その各文字での損失を平均します（156-169行目）。この損失は**交差エントロピー損失**（cross-entropy loss）と呼ばれます。

⁷大きな LLM では、MLP の部分に「パリはフランスの首都」のような事実の知識が蓄えられていると考える研究があります（Geva ら、2021）。3Blue1Brown の動画シリーズ（深層学習）の Chapter 7 も、この話題を扱っています。プログラムの変数名 `mlp_fc1`, `mlp_fc2` の `fc` は **fully connected**（全結合）の略で、入力すべての数が出力すべての数につながっている変換という意味です。

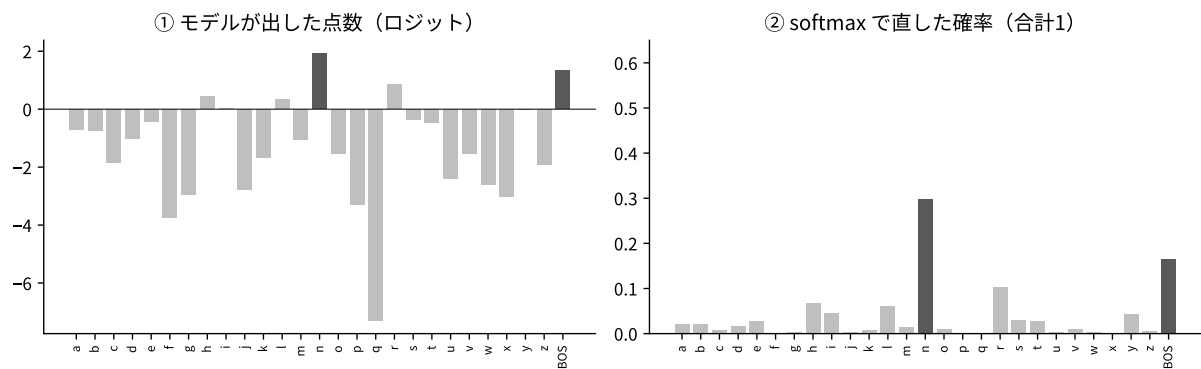


図6 softmaxのはたらき（学習後のモデルに emma を入れたときの，最後の a の次の文字）．① 点数には負の数もある．② softmax で直すと，すべて0以上，合計1の確率になる．濃い灰色は上位の n と BOS（終わり）．

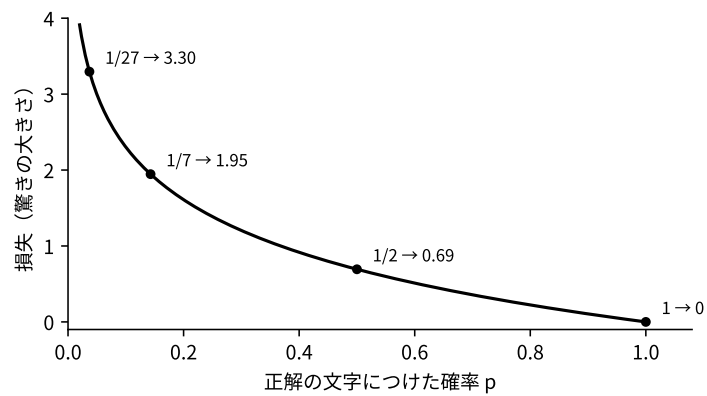


図7 正解の文字につけた確率と損失の関係．27文字に同じ確率（1/27）をつけるでたらめな予想では，損失は約3.30になる．

学習前のモデルは中身がでたらめなので、27文字にほぼ同じ確率をつけます。27文字に同じ確率をつけると、正解の確率は $1/27$ です。図7の曲線でこの点を読むと、損失は約3.30です（計算は付録A.5）。実際に1ステップ目の損失は3.37でした。学習（training）の目標は、この損失を下げることです。

4.2 4,192個のつまみ（パラメータ）

モデルの予想は、埋め込みの表や、Attention・MLP での変換に使う数で決まります。これらの数をまとめて**パラメータ**（parameter）と呼びます。パラメータは、縦横に数を並べた表、つまり**行列**（matrix）の成分です。microgpt には次の行列があり、成分を全部数えると 4,192 個になります。学習前は、すべて小さな乱数（だいたい $-0.16 \sim +0.16$ ）で始まります（80行目）。

名前	行列の大きさ	成分の数	役割
wte	27行×16列	432	文字ごとの16個の数（埋め込み）
wpe	16行×16列	256	位置ごとの16個の数（位置の埋め込み）
attn_wq, wk, wv, wo	16行×16列が4つ	1,024	Attention で使う変換（クエリ・キー・バリューを作る、結果をまとめる）
mlp_fc1	64行×16列	1,024	MLP の1段目（16→64）
mlp_fc2	16行×64列	1,024	MLP の2段目（64→16）
lm_head	27行×16列	432	出力層（16個の数→27文字の点数）
合計		4,192	

一方、計算の中を流れていく数（各文字の16個の数、クエリ・キー・バリュー、MLP の途中の64個、27個の点数）は、どれも1列に並んだ**ベクトル**です。行列とベクトルは「行列×ベクトル」という掛け算で出会います（94–95行目）。図8のように、行列の1行とベクトルの内積（成分ごとの積の和）が、出力の1つの数になります。行列の行の数だけ出力の数ができるので、16行×16列の行列なら16次元のベクトルが16次元に、64行×16列なら64次元に変わります⁸。

行列（パラメータ）	×	ベクトル	=	結果																									
<table border="1" style="border-collapse: collapse;"> <tr><td>1</td><td>2</td><td>0</td><td>1</td></tr> <tr><td>0</td><td>1</td><td>3</td><td>0</td></tr> <tr><td>2</td><td>0</td><td>1</td><td>1</td></tr> <tr><td>1</td><td>1</td><td>0</td><td>2</td></tr> </table>	1	2	0	1	0	1	3	0	2	0	1	1	1	1	0	2		<table border="1" style="border-collapse: collapse;"> <tr><td>1</td></tr> <tr><td>2</td></tr> <tr><td>0</td></tr> <tr><td>1</td></tr> </table>	1	2	0	1		<table border="1" style="border-collapse: collapse;"> <tr><td>6</td></tr> <tr><td>2</td></tr> <tr><td>3</td></tr> <tr><td>5</td></tr> </table>	6	2	3	5	2行目とベクトルの内積： $0 \times 1 + 1 \times 2 + 3 \times 0 + 0 \times 1 = 2$ → 結果の2番目の数
1	2	0	1																										
0	1	3	0																										
2	0	1	1																										
1	1	0	2																										
1																													
2																													
0																													
1																													
6																													
2																													
3																													
5																													

図8 行列×ベクトルの例（4行×4列）。行列の各行とベクトルの内積が、結果の各成分になる。microgpt では、行列の成分がパラメータ（つまみ）である。

⁸89行目では、4,192個のパラメータを1列に並べ直して `params` というリストにしています。これは Adam でつまみを1つずつ更新するための便宜で（176行目）、計算そのものでは行列として使われています。また、本物の LLM は文章のトークンをまとめて一度に計算するので、流れていく数も「トークンの数×次元」の行列になります。

パラメータは、ラジオの音を調整するつまみのようなものです（図9）。学習とは、4,192個のつまみを少しずつ回して、損失が小さくなる位置を探す作業です。

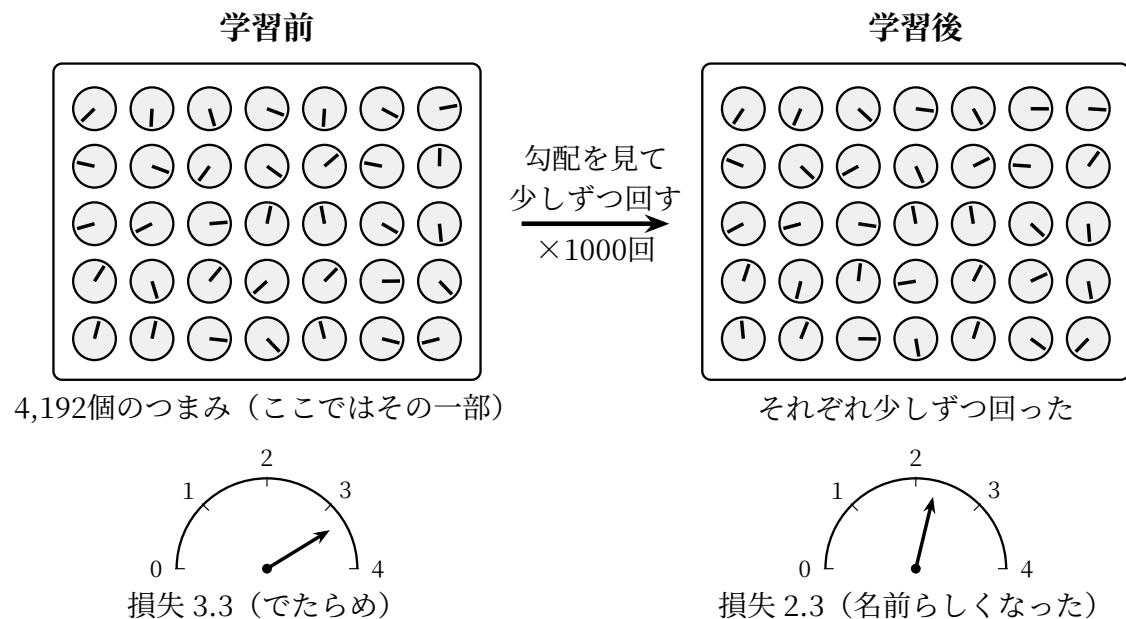


図9 パラメータを「つまみ」にたとえたイメージ図。1つのつまみが行列の1つの成分にあたる。学習でつまみを少しずつ回すと、損失のメーターが下がっていく（メーターの値は実際の学習の値）。

4.3 どのつまみを、どちらに回すか（勾配と自動微分）

4,192個のつまみを1つずつ試しに回して損失の変化を調べていては、時間がかかりすぎます。そこで、「このつまみを少しだけ増やすと、損失はどれだけ増えるか（減るか）」を全部のつまみについて一度に計算します。この値を**勾配**（gradient）と呼びます。勾配が正のつまみは減らし、負のつまみは増やせば、損失は下がります。

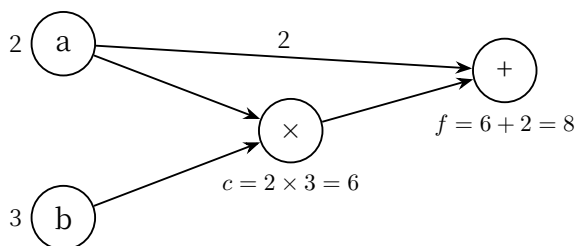
勾配を計算するのが**自動微分**（automatic differentiation, 略して autograd）です（29-72行目）。microgpt の計算は、すべて足し算・掛け算などの簡単な計算の組み合わせでできています。プログラムは計算をするたびに、「どの数とどの数から、どんな計算で作ったか」を記録していきます。こうして、計算のつながりを表す**計算グラフ**（computation graph）ができます。

小さな例で見てみましょう。 $a = 2$, $b = 3$ として $f = a \times b + a$ を計算します（図10）。

自動微分は、このように計算グラフを出力側から入力側へとさかのぼりながら、勾配を機械的に計算します⁹。microgpt の1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。172行目の `loss.backward()` の1回で、4,192個すべてのつまみの勾配が求まります。

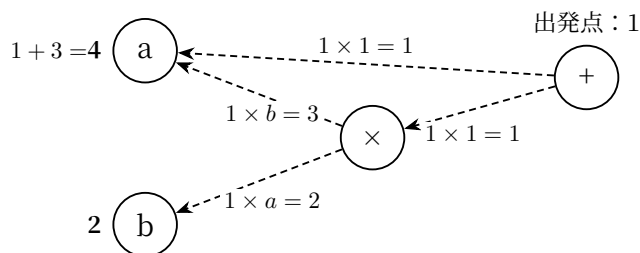
⁹この方法は**誤差逆伝播法**（バックプロパゲーション, backpropagation）と呼ばれ、ニューラルネットワークの学習の中心となる計算です。微分の**連鎖律**（chain rule）を、計算グラフに沿って順に当てはめたものです。数式による説明は付録A.5を見てください。

① 順方向：値を計算する（左→右）



$a=2$, $b=3$ から始めて、掛け算で $c=6$ 、足し算で $f=8$ 。計算するたびに「何から何を作ったか」を記録しておく。

② 逆方向：勾配を計算する（右→左）



f から出発して（ f 自身への影響は1）、記録をさかのぼる。足し算はそのまま1倍で伝え、掛け算は「相手の値」倍して伝える。 a には2つの道から1と3が届くので、合わせて4。

③ 確かめ：本当に a や b を少し増やしてみる

	a	b	f
元の値	2	3	8
a を 0.01 増やす	2.01	3	8.04 (0.04増=0.01の4倍)
b を 0.01 増やす	2	3.01	8.02 (0.02増=0.01の2倍)

②で求めた勾配（ a は4, b は2）は、「その数を少し増やすと、 f がその何倍増えるか」を表している。microgpt では f が損失で、 a や b が4,192個のつまみにあたる。

図 10 $f = a \times b + a$ の計算グラフ。① まず左から右へ値を計算する。② 次に右から左へ、「 f への影響の大きさ」（勾配）を計算する。足し算はそのまま、掛け算は相手の値を掛けて伝え、道が2つあれば足し合わせる。③ 実際に a や b を少し増やすと、 f は勾配の倍数だけ増える。

4.4 つまみの回し方の工夫 (Adam)

勾配が分かったら、つまみを回します（174–182行目）。基本は「勾配と反対の向きに、少しだけ回す」ことです（勾配降下法, gradient descent）。microgpt では、これを改良した Adam（アダム）という方法を使っています。Adam は Adaptive Moment Estimation（適応的モーメント推定）に由来する名前です。2014年に Kingma と Ba が発表しました¹⁰。Adam には次の2つの工夫があります。

- ・ **勾配の平均を使う**：1ステップでは名前を1つしか使わないので、勾配は名前ごとに大きくばらつきします。そこで、最近の勾配の平均（移動平均, moving average）を使い、ばらつきをならします。
- ・ **回す幅をそろえる**：勾配の大きさは、つまみによって何桁も違うことがあります。そこで、それぞれのつまみの勾配の大きさで割り算して、どのつまみもだいたい同じくらいの幅で動くようにします。

1回に回す幅の目安（学習率, learning rate）は 0.01 から始まり、学習が進むにつれて 0 まで少しずつ小さくしていきます（175行目）。式による説明は付録A.6にあります。

5 小さなデータで1ステップを追ってみる

32,033件のデータと 4,192個のパラメータでは、1ステップで何が起きているかを数値で追うのは大変です。そこで、プログラムの学習部分には手を加えずに、データとモデルの大きさだけを小さくして動かしてみました。

¹⁰モーメントは統計の用語で、1次のモーメントは平均、2次のモーメントは2乗の平均です。Adam は、勾配の平均と、勾配の2乗の平均を、学習しながら見積もり（推定し）、それに応じてつまみごとに回す幅を自動で調整（適応）します。頭文字を並べた略語ではないので、全部大文字の ADAM ではなく Adam と書きます。

5.1 設定

input.txt を、次の4つの名前だけにしました。

emma, ava, mia, anna

出てくる文字は a, e, i, m, n, v の6種類なので、BOS を加えた語彙は7種類です。

新しいプログラムを作ったわけではありません。元の microgpt.py の4行の数字を書き換えただけで、データの読み込み、モデル、自動微分、学習、生成の部分は1文字も変えていません。

行	元のプログラム	小さい例
76行目	n_embd = 16	n_embd = 4
77行目	block_size = 16	block_size = 5
78行目	n_head = 4	n_head = 1
152行目	num_steps = 1000	num_steps = 300

input.txt を4行にし、この4か所を書き換えれば、誰でも同じ結果を確かめられます。なお、以下の表に載せる勾配などの値は、数値をファイルに保存する数行を足して取り出しました。保存するだけなので計算には影響せず、生成される名前も4行を書き換えただけのプログラムと完全に一致します（付録D）。元の設定と小さい例を比べると、次のようになります。

	元の設定	小さい例
名前の数	32,033	4
語彙（文字の種類+BOS）	27	7
1文字あたりの数の個数（ベクトルの次元, n_embd）	16	4
ヘッドの数（n_head）	4	1
扱える最大の長さ（block_size）	16	5
学習ステップ数（num_steps）	1,000	300
パラメータの数	4,192	268
実行時間	2～3分	約2秒

5.2 1ステップ目：予想と損失

シャッフルの結果、1ステップ目に使われた名前は mia でした¹¹。モデルは次の4つの予想問題を解きます。正解につけた確率は次のとおりでした。

¹¹プログラムは名前の順番をシャッフル（ランダムに並べかえ）してから、先頭から順に1つずつ使います（20行目、156行目）。コンピュータの乱数は、「乱数の種」（random seed）と呼ばれる出発点の数から計算で作られ、同じ種からは毎回同じ数の列が出てきます。microgpt では12行目の random.seed(42) で種を42に決めているので、同じプログラムを動かせば、誰がやっても1ステップ目は mia になります。種を別の数に変えると、シャッフルの順番、パラメータの初期値、名前を作るときのかじ引きがすべて変わります。

入力	正解	正解につけた確率	1ステップ後
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 (終わり)	0.131	0.140
損失 (平均)		1.953	1.889

7種類に同じ確率をつけると、どれも $1/7 \approx 0.143$ です。学習前のモデルは、ほぼこの「でたらめ」な状態にあります。損失 1.953 も、でたらめに当てたときの値 ($1/7$ のときの損失、約 1.95) とほぼ同じです。

5.3 勾配とつまみの動き

`loss.backward()` で、268個すべてのつまみの勾配が求まります。いくつかの例を見てみましょう。

つまみ	学習前の値	勾配	1ステップ後の値
lm_head (文字 i の行の3番目)	0.1090	+0.461	0.0990
lm_head (文字 a の行の2番目)	0.0940	-0.438	0.1040
lm_head (文字 m の行の4番目)	-0.0270	-0.390	-0.0170
wte (文字 a の1番目)	0.0014	+0.282	-0.0086
wte (文字 e の1番目)	-0.1026	0	-0.1026

ここから、次のことが読み取れます。

- 勾配が正のつまみは減り、負のつまみは増えています。どれも損失を下げる向きです。
- 動いた幅は、どのつまみもちょうど 0.01 です。勾配が 0.46 でも 0.28 でも同じです。これは Adam の「回す幅をそろえる」工夫によるもので、最初のステップでは、すべてのつまみが学習率 (0.01) だけ、勾配と反対の向きに動きます。
- 勾配が 0 で、まったく動かないつまみもあります。268個のうち16個がそうでした。mia に入力として出てこない文字 e, n, v の埋め込み (3文字×4個=12個) と、使われなかった5番目の位置の埋め込み (4個) です。使われなかった部分は損失に影響しないので、勾配は0になります。

5.4 1ステップ後の変化

つまみを動かした後で、同じ mia をもう一度予想させると、前の表の右の列のように、4つの正解の確率がすべて少しずつ上がり、損失は 1.953 から 1.889 に下がりました。

ところが、ほかの名前の損失を調べると、次のようになっていました。

名前	学習前の損失	1ステップ後の損失
mia (学習に使った名前)	1.953	1.889 (下がった)
ava	1.971	1.938 (下がった)
emma	1.963	1.979 (上がった)
anna	2.001	2.002 (ほぼ同じ)

mia に合わせてつまみを動かすと、mia と同じく「a で終わる」ava は良くなりましたが、emma はかえって悪くなりました。1つの名前に合わせた調整が、ほかの名前にとって良いとは限らないのです。だからこそ、**名前を替えながら何度もくり返し**、全体として損失が下がるつまみの位置を探していきます。

5.5 300ステップ後

4つの名前を順番にくり返し使って300ステップ学習すると、損失は図11のように下がりました。

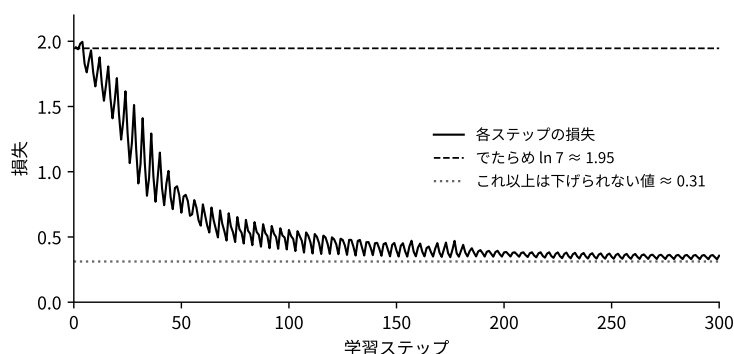


図 11 小さい例の損失の変化。点線は、これ以上は原理的に下げられない値（本文参照）。

学習前と学習後で、anna を1文字ずつ入れたときの「次の文字」の確率を比べたのが図12です。濃い灰色の棒が正解の文字です。

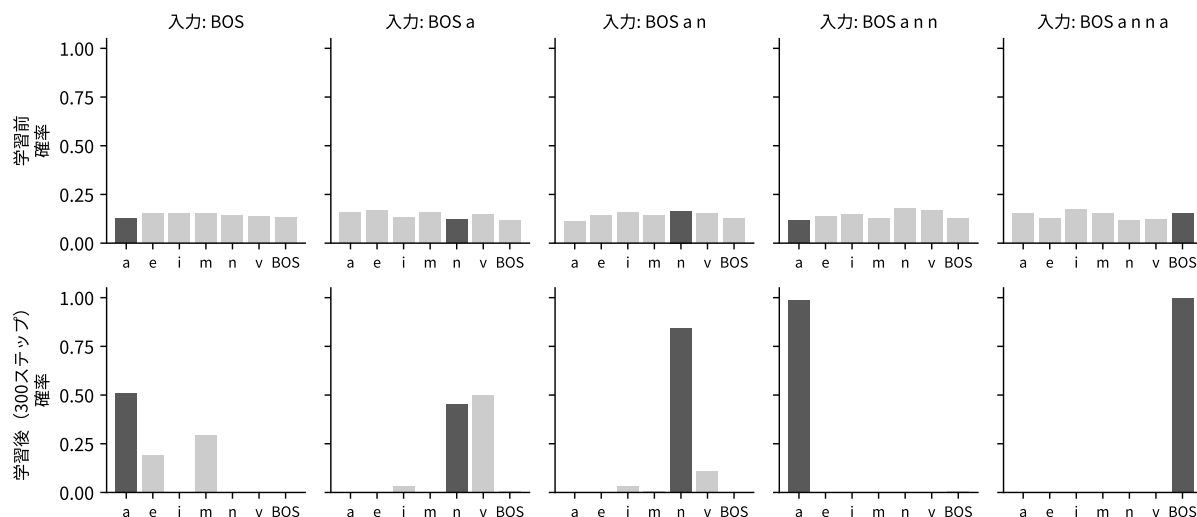


図 12 anna を入力したときの、次の文字の確率（上：学習前，下：300ステップ後）。濃い灰色が正解の文字。

学習前は、どの文字にもほぼ同じ確率をつけています。学習後は、次のようになっています。

- ・ **最初の文字**（入力 BOS）：a が 0.51，m が 0.30，e が 0.19。4つの名前の最初の文字は a が2つ（ava, anna），m と e が1つずつなので、その割合（2/4, 1/4, 1/4）に近い値です。

- ・ **a の次** (入力 BOS a) : n が 0.46, v が 0.50. a で始まる名前は ava と anna の2つなので, どちらが来るかは本当に半々です.
- ・ **ann の次や anna の次**: ほぼ確実に a, BOS (終わり) と予想しています. ここは答えが1つに決まるからです.

つまりモデルは, データから「どこで迷うべきで, どこで迷わなくてよいか」を学んでいます. 最初の文字や a の次は本当に迷う場面なので, どんなにうまく学習しても損失は0になりません. 4つの名前の場合, 損失を下げられる限界は約0.31で, 学習後の損失 (約0.35) はそこはかなり近づいています.

文字の名札も変わった 3.2節で, 文字の表 (wte) の16個の数は「特徴を書いた名札」のようなもので, 似た使われ方をする文字は似た数の並びになっていく, と説明しました. 小さい例 (1文字あたり4個の数) で, 実際の名札を比べてみます.

文字	学習前の名札 (4個の数)				学習後の名札 (300ステップ)			
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

学習後, **i と v の名札がほぼ同じ数の並び**になっています. 4つの名前の中で, i は mia に, v は ava に出てきて, どちらも**次に来る文字は必ず a**です. 「次に来る文字が同じ」という使われ方が似ているので, 名札も似たものに学習されたと考えられます. 2つの名札が「同じ向きを向いているか」を表す値 (コサイン類似度, 付録B) は, 学習前の0.64から, 学習後には1.00 (ほぼ完全に同じ向き) になりました. 名札をベクトル (矢印) として見ると何が起きているのかは, 付録Bで3次元の図を使って説明します.

作られた名前 学習後に名前を20個作らせると, ava が8個, anna が7個, mia が4個, emma が1個で, **4つの名前以外は1つも出てきませんでした**. データが少なすぎて, モデルは名前を「丸暗記」してしまったのです. 次の節で見るように, 32,033件で学習すると, データにない新しい名前が作られるようになります.

6 本番の学習 (32,033件)

6.1 「学習」とは何をすることか

AI の「学習」 (training, learning) は, 人間の学習とは意味が違います. AI にとって学習とは, **損失が小さくなるようにパラメータ (つまみ) を調整すること**です. それ以上でも以下でもありません.

これは, 理科の実験でおなじみの作業とよく似ています. たとえばばねにおもりをつるして伸びを測り, グラフに直線を引くとき, 私たちは「データの点と直線のずれ」がなるべく小さ

くなるように、直線の傾きと切片を決めます（最小二乗法，least squares method では、ずれの2乗の合計を最小にします）。このとき、

- ・傾きと切片が「パラメータ」（2個）
- ・ずれの2乗の合計が「損失」
- ・ずれが最小になる傾きと切片を求めることが「学習」

にあたります。microgpt がしていることも本質的には同じで、パラメータが4,192個に増え、損失が「驚きの平均」（交差エントロピー）に変わっただけです。ただし、パラメータが多すぎて式では解けないので、勾配を見て少しずつ動かす方法（勾配降下法）で、損失が小さくなる位置を探していきます。損失の値の読み方は付録A.5にあります。

6.2 損失の変化

元の設定に戻り、名前を替えながら1ステップを1000回くり返します。損失の変化を図13に示します。

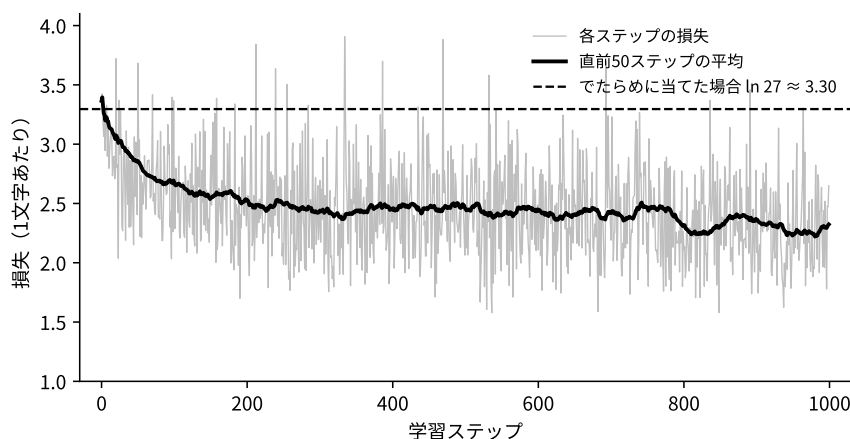


図 13 損失の変化。灰色は各ステップの損失（そのステップで使った1つの名前についての値），黒線は直前50ステップの平均。

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作らせると，次のように変わっていきました（7節と同じ方法）。

学習ステップ	作られた名前（最初の5個）
0（学習前）	jdodmicpalbdjxcj, pzol, bwhechtepsnqbdeq, liomhsugpozri, ck-ufnbqwpykuntmx
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」長さや響きになっています。

7 名前を作る

7.1 しくみ

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）。これを生成（generation）と呼びます。

1. BOS から始める。
2. `gpt()` で次の文字の確率を求める（195行目）。
3. その確率を当たりやすさにしたくじを引いて、文字を1つ選ぶ（196行目）。確率の高い文字ほど当たりやすい。
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。

たとえば `emma` の後（最後の `a` の次）では、温度0.5で調整した確率は `n` が0.62、BOS（終わり）が0.19で、ほかの文字が残りを残りを少しずつ分け合っています（図14）。これは、1000本のくじのうち「`n`」の当たりが約620本、「終わり」が約190本、残りの約190本がほかの文字、というくじを1本引くことにあたります。`n` が選ばれやすいのは確かですが、必ず選ばれるわけではなく、5回に1回くらいは「終わり」が出て `emma` で終わります。くじ引きなので、同じモデルからでも毎回ちがう名前が作られます¹²。

確率を求めるときには、**温度（temperature）** と呼ばれる値（0.5）で点数を割っています。温度が1より小さい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（図14）¹³。

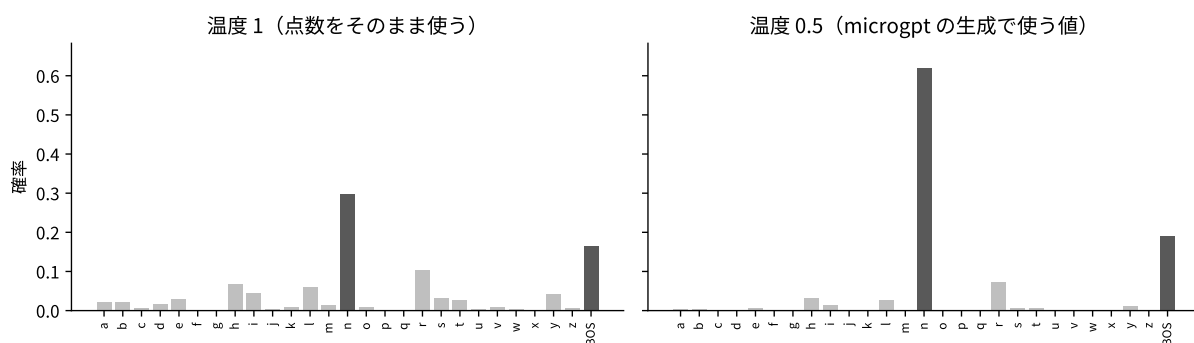


図 14 温度による確率の変化（図6と同じ場面）。温度0.5では、上位の `n` の確率が0.30から0.62に上がり、確率の低い文字はさらに小さくなる。

学習では名前のすべての位置の予想をまとめて扱いましたが、生成では**選んだ文字が次の入力になる**ので、1文字ずつ順にしか進めません。本物の LLM が文章を少しずつ表示するのも、同じ理由です。

¹²もし確率を考えずに完全にランダムに選ぶと、27文字が同じ確率になり、学習前のような、でたらめな文字列になります。逆に、くじ引きをやめて「いつも一番確率の高い文字」を選び続ける方法（貪欲法、**greedy decoding**）では、同じモデルからは毎回まったく同じ名前しか出てきません。確率を重みにしたくじ引きにすることで、名前らしさを保ちながら、いろいろな名前を作れるのです。プログラムでは、重み付きのくじを引く `random.choices` という関数を使っています。

¹³温度が1より小さいと無難な名前が多くなり、1より大きいと点数の差が縮まって、珍しい文字も選ばれやすくなります（変わった名前が多くなる）。温度は学習では使われず（学習のときは点数をそのまま使う）、名前を作るときに使う人が選ぶ設定値です。microgpt では187行目で作者が0.5に決めていて、学習し直さなくても、この値だけを変えて名前を作り直せます。本物の LLM でも温度はモデルの一部ではなく使うときの設定で、開発者向けの API では利用者が指定でき、チャットアプリでは提供する会社が決めた値が使われます。「温度」という名前は物理学から来ています（付録A.4）。

7.2 生成結果

1000ステップの学習の後に作られた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, データにはないが名前らしいものが多く含まれます. 5節の小さい例とは違い, 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¹⁴.

7.3 モデルの中をのぞく

学習後のモデルに emma を1文字ずつ入れたとき, 各位置で予想された「次の文字」の確率の上位を示します.

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 (em-man...)」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드의 Attention の重みを図15に示します.

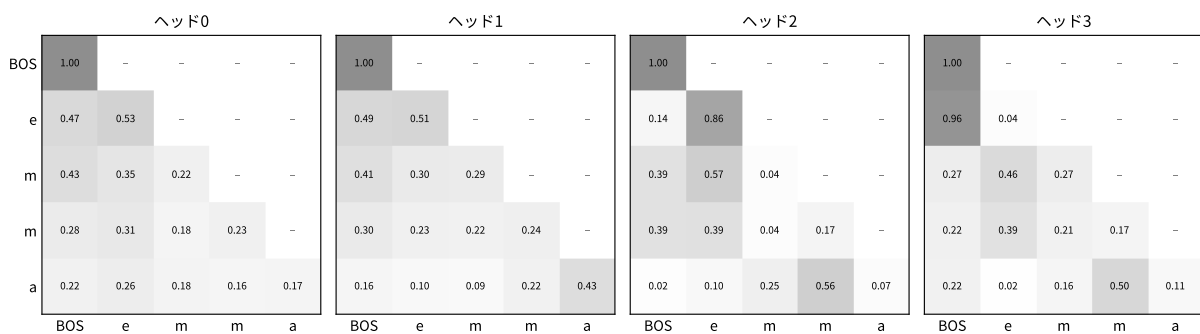


図 15 emma を入力したときの Attention の重み（学習後）. 行が今処理している文字, 列が振り返られる文字で, 各行の合計は1. 右上の「-」は, まだ存在しない未来の位置.

右上が空いているのは, 未来の文字のメモ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（たとえば最後の a の行では, ヘッド1は自分自身に, ヘッド2と

¹⁴データにないものを「それらしく」作り出すという点で, 本物の LLM が事実でないことをもっともらしく答えてしまう現象（ハルシネーション, hallucination）と同じしくみです. モデルは「正しいこと」ではなく「ありそうな続き」を出しているからです.

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¹⁵。

8 本物の GPT との比較

8.1 規模

	microgpt	GPT-2 (最小)	GPT-3
1文字 (1トークン) あたりの数の個数 (次元)	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、細かな違いが3点あるだけです（正規化の方法、定数項の有無、MLP で使う関数の種類）。それ以外の違いは、計算の中身ではなく**計算のしかた**にあります。

	microgpt	本物の LLM
トークン	1文字	よく出る文字の組み合わせ
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）
計算の単位	数を1つずつ	数の表（行列）をまとめて
計算機	CPU 1つで、1つずつ	多数の GPU で同時に
学習後の調整	なし	人間の好みに合わせた追加の学習 ¹⁶

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。本物の LLM は同じ種類の計算を GPU で大量に同時に行います。最近のモデルはさらにいくつかの工夫を加えていますが、「**文字を数の並び（ベクトル）にする → Attention と MLP をくり返す → 次のトークンの確率を出す**」という骨組みと、「**損失を勾配で下げる**」という学習の原理は同じです。

¹⁵学習で決まったパラメータの意味を人間が解釈するのは、一般にとっても難しいことです。モデルがなぜうまく予想できるのかを内部から説明する研究（解釈可能性の研究、interpretability）は、現在も盛んに行われています。

¹⁶ChatGPT や Claude のような対話型の AI は、大量の文章で「次のトークンの予想」を学んだ後、指示に従う例や人間の評価をもとにした追加の学習（指示調整 instruction tuning や RLHF Reinforcement Learning from Human Feedback など）を行って、質問に答えるのが上手になるよう調整されています。

9 まとめ

1. microgpt は、データ・トークナイザ・自動微分・モデル・学習・生成を、Python の標準機能だけで約200行に書いた GPT である。
2. モデルは「次の文字」に点数をつけ、softmax で確率に直す。Attention で前の文字を振り返って情報を集め、MLP で1文字ずつ考えをまとめる。
3. 学習では、正解につけた確率から「驚き」（損失）を測り、自動微分で全部のつまみ（行列の成分）の勾配を求め、Adam でつまみを少しずつ回す。
4. 4つの名前で試すと、1ステップでつまみがどう動くかを数値で追える。データが少ないとモデルは名前を丸暗記するが、32,033件で学習すると、名前らしい新しい名前を作れるようになる。
5. 本物の LLM との違いは主に規模と計算のしかたで、骨組みと学習の原理は同じである。

付録A 数式で見る microgpt

本文で言葉で説明した計算を、数式で書きます。ベクトル（数の並び）は太字で $\mathbf{x}$ のように書き、 x_i はその i 番目の数です。 W に添字をつけたもの (W_Q など) は行列です。 d はベクトルの次元 (microgpt では $d = 16$) , $\ln$ は自然対数です。

A.1 行列×ベクトル

m 行 n 列の行列 W (成分 W_{ij}) と n 次元のベクトル $\mathbf{x}$ の積 $\mathbf{y} = W\mathbf{x}$ は、 m 次元のベクトルで、その i 番目の成分は

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

です。つまり、行列の i 行目とベクトルの内積です (図8, 94–95行目)。

A.2 埋め込みと正規化

文字の番号に対応する w_{te} の行 $\mathbf{e}_{\text{文字}}$ と、位置に対応する w_{pe} の行 $\mathbf{p}_{\text{位置}}$ を足します。

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

正規化 (RMSNorm) では、各成分の2乗の平均の平方根で割って、大きさをそろえます (103–106行目)。 $\varepsilon = 10^{-5}$ は0で割るのを防ぐための小さな数です。

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

A.3 Attention

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)。

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

16個の数を4個ずつ4つのヘッドに分けます。ヘッド m のクエリを $\mathbf{q}^{(m)}$ 、位置 s のキーとバリューを $\mathbf{k}_s^{(m)}$ 、 $\mathbf{v}_s^{(m)}$ とし、今の位置を t とすると、重み a_s とヘッドの出力は

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

です。 $\mathbf{q} \cdot \mathbf{k}$ は内積で、クエリとキーの「合い具合」を表します。 $d_k = 4$ は1つのヘッドの次元です。和が $s = 0$ から t までしかないことが、「未来を見ない」ことに対応します。4つのヘッドの出力をつないで W_O を掛け、入口のベクトルを足します（残差接続、133-134行目）。

A.4 MLP, 出力と softmax

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

W_1 は 64×16 、 W_2 は 16×64 の行列です（137-141行目）。ReLU がなければ $W_2 W_1$ は1つの 16×16 の行列にまとめられてしまうので、途中を64次元に広げる意味がなくなります。最後に lm_head (27×16) を掛けて27個の点数 $z_1, \dots, z_{27}$ を得て、softmax で確率にします。

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

softmax は次の2段階の計算です。(1) それぞれの点数 z を e^z ($e = 2.718 \dots$ の z 乗) に置き換える。これで負の点数も含めてすべて正の数になり、点数が大きいほど値が急に大きくなる。(2) 全部の合計で割る。これで合計がちょうど1になる。たとえば3つの文字の点数が 2, 1, 0 なら、 $e^2 \approx 7.39$, $e^1 \approx 2.72$, $e^0 = 1$ で合計は11.11。それぞれを合計で割ると、確率は 0.67, 0.24, 0.09 になります。点数の差は1ずつでも、確率では差がずっと大きくなります。

生成のときは、点数を温度 T (0.5) で割ってから softmax を取ります。

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

この形は、統計力学 (statistical mechanics) のボルツマン分布 (Boltzmann distribution) と同じです。絶対温度 T の系で、エネルギー E_j の状態が実現する確率は

$$p_j = \frac{e^{-E_j/k_B T}}{\sum_i e^{-E_i/k_B T}} \quad (k_B \text{ はボルツマン定数, Boltzmann constant})$$

です。点数 z_j をエネルギーに負の符号をつけたもの ($z_j \leftrightarrow -E_j/k_B$) と見れば、softmax に温度を入れた式とまったく同じ形になります。 $T \rightarrow \infty$ ではすべての状態が等確率に近づき（一様分布, uniform distribution）、 $T \rightarrow 0$ では最もエネルギーの低い状態（最も点数の高い文字）だけが選ばれるようになります（貪欲法と同じ）。この対応から、softmax はボルツマン分布やギブス分布 (Gibbs distribution) と呼ばれ、温度 (temperature) という名前もここから来ています。

A.5 損失と勾配

名前の長さが n のとき、位置 t での正解の文字の確率を p_t として、損失は

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

です（交差エントロピー損失）。27文字に同じ確率をつけると、どの位置でも $p_t = 1/27$ なので、

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

となります。 $\ln 27$ は「 e を何乗すると27になるか」という数で、実際 $e^{3.30} \approx 27.1$ です。各位置の損失がすべて同じなので、平均も名前の長さに関係なく3.30です。小さい例の7種類なら $\ln 7 \approx 1.95$ です。

実際の1ステップ目が3.37と少し大きかったのは、学習前のパラメータが小さな乱数で、27文字への確率が「ほぼ」等しいだけで少しばらつくためです。 $-\ln p$ は確率が小さくなる側で急に大きくなる形をしているので（図7）、ばらつきがあると平均の損失は3.30より少し大きくなりやすいのです（ $-\ln p$ が下に凸な関数であることによる、イェンセンの不等式、Jensen's inequality）。

損失の値の読み方：パープレキシティ 損失 L に対して e^L を計算した値をパープレキシティ（perplexity）と呼びます。これは「モデルが平均して何個の候補で迷っているか」の目安になります。実際、 K 個の候補に等しい確率 $1/K$ をつけるモデルの損失は $\ln K$ なので、 $e^L = K$ となります。

- ・学習前： $L \approx 3.30 \rightarrow e^{3.30} \approx 27$. 27文字すべてで同じように迷っている（でたらめ）。
- ・学習後（最後の100ステップの平均）： $L \approx 2.28 \rightarrow e^{2.28} \approx 9.8$. だいたい10文字ほどに候補を絞り込めている。
- ・小さい例（5章、300ステップ後）： $L \approx 0.35 \rightarrow e^{0.35} \approx 1.4$. ほとんど迷っていない（丸暗記に近い）。

「損失が3.30から2.28に下がった」というより、「27択から約10択になった」と言うほうが、学習で何が起きたかを実感しやすいでしょう。

損失の曲線の読み方 図13の損失の曲線（学習曲線，learning curve）からは、次のことが読み取れます。

- ・最初に急に下がり、だんだん下がり方がゆるやかになる。学習率を徐々に小さくしていることも影響している。
- ・1ステップごとの値（灰色）は、そのステップで使った1つの名前によって大きくばらつく。傾向は移動平均（黒線）で見る。1ステップで使うデータを増やせば（ミニバッチ，mini-batch）、ばらつきは小さくなる。
- ・損失には、データそのものの不確かさから決まる下限がある。5章の小さい例では約0.31で、学習後の損失はそこに近づいた。
- ・ここで測っているのは学習に使った名前に対する損失（訓練損失，training loss）である。本来は、学習に使っていないデータでの損失（検証損失，validation loss）も測り、丸暗記（過学習，overfitting）していないかを確認する。microgpt は1000ステップでデータの約3%しか使わないので、過学習の心配はほとんどない。

勾配 $\partial L / \partial w$ は、パラメータ w を少し変えたときの L の変化の割合です。自動微分では、計算グラフを出力側からさかのぼり、各数 v について、その入力 u に

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

を足し込みます（連鎖律）。図10の例では、 $f = ab + a$ を直接微分した $\partial f / \partial a = b + 1 = 4$ 、 $\partial f / \partial b = a = 2$ と一致します。付録Cのプログラムで、数値微分と一致することを確認しています。

A.6 Adam

パラメータ w ごとに、勾配 g の移動平均 m と、 g^2 の移動平均 v を持ちます。 k ステップ目 ($k = 1, 2, \dots$) に

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

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 $\beta_1 = 0.85$, $\beta_2 = 0.99$, $\epsilon = 10^{-8}$, 学習率 η は0.01から0まで直線的に下げます。

1ステップ目 ($k = 1$) では $\hat{m} = g$, $\hat{v} = g^2$ となるので、 $\hat{m}/\sqrt{\hat{v}} = g/|g| = \pm 1$ です。つまり、勾配が0でないすべてのパラメータが、ちょうど $\eta = 0.01$ だけ、勾配と反対の向きに動きます。 5.3節で見た「どのつまみも0.01動く」のはこのためです。

A.7 内積とコサイン類似度

2つのベクトル $\mathbf{a}$, $\mathbf{b}$ のなす角を θ とすると、内積は

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

で、コサイン類似度はこれを長さで割った

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

です。内積は向きと長さの両方で決まりますが、コサイン類似度は向きだけで決まり、必ず -1 から 1 の間に入ります (付録B)。

付録B 名札ベクトルの向きを3次元で見る

5.5節で、学習後に文字 i と v の名札 (wte の行) がほぼ同じ数の並びになることを見ました。名札はベクトルなので、矢印として描けば「同じ向きを向いている」ことが目で見て分かるはずです。ただ、本文の小さい例は1文字あたり4個の数 (4次元) で、図に描けません。そこで、図に描けるよう**1文字あたり3個の数 (3次元) に変えて**、同じ4つの名前で学習し直しました。3次元でも学習はうまくいき、損失は約0.36まで下がって、生成される名前も4つの名前のどれかになりました。

図16の左 (学習前) では、 i と v の矢印は別々の方向を向いています。右 (学習後) では、 **i と v がほぼ同じ方向**を向いています。3個の数の並びは3次元のベクトルで、このように矢印として描けます。本文の16個の数の並び (16次元のベクトル) は図に描けませんが、考え方は同じで、似た使われ方をする文字の名札は、16次元の空間の中で似た向きを向くようになります。

矢印の長さをそろえた理由 モデルは Attention と MLP の入口で数の大きさをそろえる処理 (正規化) をするので、名札で主に意味を持つのは向きです。そこで図では、向きの違いだけが見えるように、すべての矢印の長さを1にそろえました。

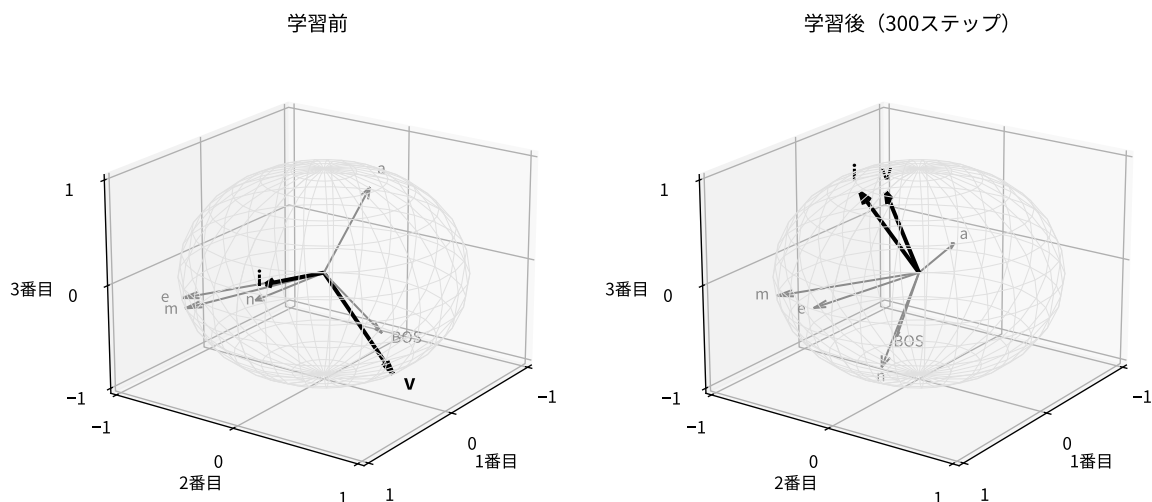


図 16 7種類の文字の名札（3次元のベクトル）を矢印で描いたもの（左：学習前，右：300ステップ後）。向きを比べやすいように，矢印の長さはすべて1にそろえてある。黒が i と v 。

向きの近さを1つの数で表す：コサイン類似度 2本の矢印の向きがどれだけ近いかは，**コサイン類似度** (cosine similarity) という1つの数で表せます。2本の矢印のなす角を θ としたときの $\cos \theta$ で，同じ向きなら1，直角なら0，逆向きなら -1 です。3次元の例で i と v のコサイン類似度を計算すると，学習前は0.04（ほぼ直角）でしたが，学習後は0.98になりました。

内積との違い コサイン類似度は，ベクトルの**内積**を2本の長さで割ったものです（式は付録A.7）。内積は向きと長さの両方で決まり，同じ向きでも長いベクトルほど値が大きくなります。3次元の例では， i と v の内積は $+0.0109$ ， i と a の内積は -0.0156 で，絶対値だけを見ると向きが大きく違う i と a の組のほうが大きくなります。これは a のベクトルが長い（ i の約3.6倍）ためです。向きが似ているかどうかを比べたいときは，長さの影響を除いたコサイン類似度のほうが適しています。

なお，モデル自身が Attention でクエリとキーの合い具合を測るときには，コサイン類似度ではなく，内積をそのまま使っています（129行目）。向きに加えて長さも結果に効くので，モデルは長さも使って「どれだけ強く注目するか」を調整できます。この付録のコサイン類似度は，学習後の名札を人間が比べるための道具です。

付録C Python コード：自動微分の最小例（標準ライブラリのみ）

図10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D 本文の数値と図の求め方

本文の数値と図は、元のプログラムに次の変更を加えて求めました。モデルと学習の計算そのものには手を加えていません。

- **損失の記録 (図13, 図11)** : 184行目の進行表示の代わりに、各ステップの損失をリストに保存した。
- **学習途中の生成 (6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、生成には別の乱数の種 (7) を使った。
- **内部の値 (図5, 図6, 図14, 7.3節の表, 図15, 図12)** : 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小数で書き直したプログラムで、各位置の確

率, Attention の重み, MLP 中の値を計算した. 3.4節の「1文字あたり平均約8個」は, データから無作為に選んだ300個の名前 (のべ2,156文字) について, 反応した検出器の数を平均した値 (7.7個) である.

- **小さい例 (5節)** : input.txt を4つの名前 (emma, ava, mia, anna) だけにし, `n_embd = 4`, `n_head = 1`, `block_size = 5`, `num_steps = 300` に変えた. 乱数の種 (42) はそのままである. 1ステップ目については, `loss.backward()` の直後に全パラメータの値と勾配を, 更新の直後に全パラメータの値を保存した. 5.5節の「下げられる限界」は, 4つの名前の最初の文字と a の次の文字の割合から計算した値 (約0.312) である.
- **3次元の例 (付録B)** : 小さい例の設定から `n_embd` だけを3に変えて学習し直した.

付録E microgpt のプログラムの入手先と行番号

本書は microgpt のプログラムを読みながら説明していますが, プログラムそのものは掲載していません. プログラムには利用条件 (ライセンス) が示されていないため, 作者の配布ページから入手してください.

- 入手先: Andrej Karpathy, microgpt (GitHub Gist)
<https://gist.github.com/karpathy/8627fe009c40f57531cb18360106ce95>
- 作者による解説: <https://karpathy.github.io/2026/02/12/microgpt/>
- 本書の行番号は, 2026年9月30日に取得した版 (全200行, 配布ページの最終更新は2026年9月19日) によります. 配布ページが更新されると, 行番号がずれる可能性があります.

本文で参照している行番号と内容の対応は, 次のとおりです.

行	内容	本書の節
1-7	プログラムの説明文 (「このファイルがアルゴリズムのすべて」)	1
9-12	ライブラリの読み込み. 12行目で乱数の種を42に設定	5.2
14-21	データ: input.txt を読み込み, 名前の順番をシャッフル	2.1
23-27	トークナイザ: 文字に番号をつけ, BOS を加える	2.2
29-72	自動微分 (Value クラス). 39-57行目が基本演算と局所的な影響の度合い, 59-72行目が逆方向の計算 <code>backward()</code>	4.3
74-90	パラメータ: 75-79行目が大きさの設定, 80行目が初期値 (乱数), 89行目で1列に並べ直す	3.2, 4.2
92-106	補助の関数: 93行目が GPT-2 との違いの注釈, 94-95行目が行列×ベクトル (<code>linear</code>), 97-101行目が <code>softmax</code> , 103-106行目が正規化 (<code>RMSNorm</code>)	3.5, 4.2, 8.2
108-144	モデル本体 <code>gpt()</code> : 109-112行目が埋め込み, 115-134行目が Attention (121-122行目がメモへの追加, 124-132行目が4つのヘッド), 135-141行目が MLP, 143行目が出力層	3
146-149	Adam の設定 (学習率, β_1 , β_2) と移動平均の入れ物	4.4
151-184	学習: 156-157行目で名前を1つ取り出し, 163-169行目で損失, 172行目で勾配, 174-182行目で Adam による更新	4, 5, 6
186-200	生成: 187行目が温度, 195行目が確率の計算, 196行目が重み付きのくじ引き	7