

深層学習（4）系列データと Transformer

（高校生・大学初年次向け）

RNN から Attention へ

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

Ver. 0.2 October 5, 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）。

はじめに

第3回では、画像のように空間的な並びに意味があるデータを扱いました。第4回では、文章や時系列のように時間的な順序に意味があるデータ（系列データ）を扱います。前半で古典的な再帰型ニューラルネットワーク（RNN）とその限界を見て、後半で現在の LLM の基盤である Transformer を扱います。

長い小説を読むとき、私たちは前のほうに出てきた人物や出来事を覚えておき、必要に応じて思い出しながら読み進めます。「過去の内容をどう覚えておき、どう思い出すか」——これが系列データを扱うときの中心の問題です。

回	題目
第1回	ニューラルネットワークの基礎
第2回	誤差逆伝播法と学習の実際
第3回	畳み込みニューラルネットワーク
第4回	系列データと Transformer

読み方 本文は、高校の数学（ベクトルの内積、指数関数と対数、三角関数、数列の和 $\sum$ 、条件付き確率）を使って読めるように書きました。Attention の計算は、行列を使わずに、トークンごとのベクトルの内積と重み付きの和で説明します。行列を使った一般的な書き方（RNN, LSTM, Attention, マルチヘッド注意）は、脚注と巻末の付録Aにまとめました。付録は読み飛ばしても本文の流れは追えます。例題の数値はすべて実際に計算した値です（付録Bのプログラム）。

記号の約束

- 系列の t 番目（時刻 t ）の入力を x_t 、隠れ状態を h_t と書きます（添字 t は時刻）。本来はどちらも複数の数を並べたベクトルですが、本文では考え方を説明するために1つの数として扱うことがあります。
- 太字の $\mathbf{q}$, $\mathbf{k}$, $\mathbf{v}$ はベクトル、 $\mathbf{q} \cdot \mathbf{k}$ は内積です。
- d はトークンのベクトルの次元（成分の数）、 d_k はクエリ・キーの次元、 n は系列の長さです。

1 系列データ

1.1 系列データの特徴

文章、音声、気温や株価の変動、地震波形などは、値が**順番に並んだデータ**です。これらには次の特徴があります。

1. **順序に意味がある**：「犬が人をかんだ」と「人が犬をかんだ」は、同じ単語からなるが意味が違う。
2. **長さが一定でない**：文の長さは文ごとに違う。
3. **離れた位置どうしが関係する**：「太郎は花子に本を渡した。彼は.....」の「彼」は、ずっと前の「太郎」を指す。

第1～3回のネットワークは入力の大きさが決まっていたので、これらの性質をそのまま扱うことができません。

1.2 言語モデルという課題

以下では、系列データの代表的な課題として、**それまでの単語から次の単語を予測する課題**（言語モデル, language model）を考えます。スマートフォンの予測変換が「よろしく」の次に「お願いします」を出すのと同じ課題です。つまり、

$$p(x_{t+1} \mid x_1, x_2, \dots, x_t)$$

を予測します。これは「 $x_1, \dots, x_t$ が与えられたときに、次が x_{t+1} である確率」（条件付き確率）で、数学Aの記号では $P_A(B)$ にあたります。語彙のすべての単語の中から次の単語を選ぶ多クラス分類で、クラスの数 は語彙の大きさです。

LLM は、まさにこの課題を大量の文章で学習したモデルです。

2 再帰型ニューラルネットワーク（RNN）

2.1 しくみ

RNN（Recurrent Neural Network）は、過去の情報を要約した「隠れ状態」 h_t をもち、入力を1つ読むたびにそれを更新していくネットワークです。

1枚の小さなメモ用紙を持って本を読むところを想像してください。1語読むたびに、メモの内容を「これまでのあらすじ」として書き直していきます。次の単語を予想するときは、このメモだけを頼りにします。このメモが隠れ状態 h_t です。

隠れ状態と入力それぞれ1つの数のときで書くと、

$$h_t = \tanh(w_h h_{t-1} + w_x x_t + b)$$

です。 h_{t-1} は1つ前の時刻の隠れ状態、 w_h , w_x は重み、 b はバイアスです。「1つ前の隠れ状態（さっきまでのメモ）」と「今の入力（いま読んだ単語）」の重み付きの和を活性化関数 $\tanh$ に通したものが、新しい隠れ状態（書き直したメモ）になります。各時刻の出力 $\hat{y}_t$ （次の単語の確率）は、隠れ状態 h_t から全結合層とソフトマックス関数で計算します¹。

¹実際には隠れ状態も入力もベクトルで、重みは行列になります。 $\mathbf{h}_t = \tanh(W_h \mathbf{h}_{t-1} + W_x \mathbf{x}_t + \mathbf{b})$, $\hat{\mathbf{y}}_t = \text{softmax}(W_y \mathbf{h}_t + \mathbf{c})$ です（付録A.1）。

自分の出力を次の時刻の入力に戻す（再帰する）ので、**再帰型**と呼ばれます。時間方向に展開すると図1のようになります。

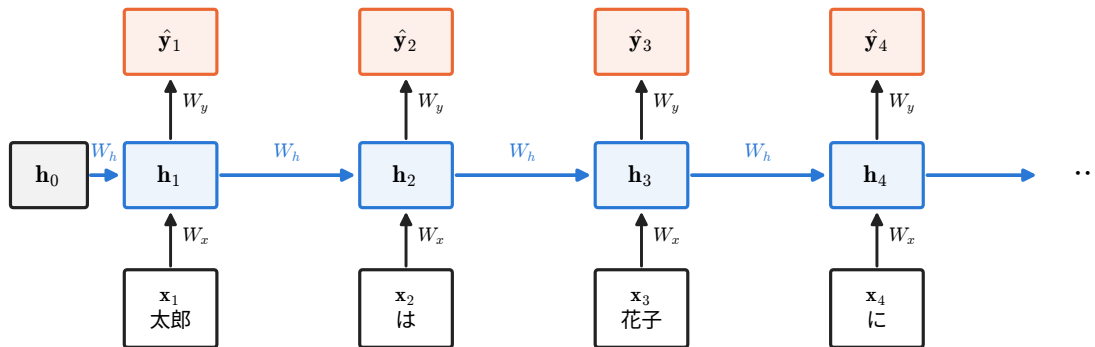


図1 時間方向に展開した RNN. どの時刻でも同じ重み W_x , W_h , W_y を使う（太字はベクトル, W は重みの行列）. 隠れ状態 h_t が, それまでに読んだ内容を要約して次の時刻に受け渡す.

重要なのは、**どの時刻でも同じ重みを使う**ことです. 第3回の畳み込みが空間方向に重みを共有したのと同じように, RNN は**時間方向に重みを共有**しています. メモの書き直し方のルールは, どのページを読んでいるときも同じなのです. そのため, どんな長さの系列にも同じネットワークを適用できます.

2.2 学習と勾配消失

RNN も誤差逆伝播法で学習します. 時間方向に展開した図1を1つの深いネットワークとみなし, 最後の時刻から最初の時刻へ誤差を逆向きに送ります（**通時的誤差逆伝播法**, Backpropagation Through Time, BPTT）.

誤差はどのように過去へ届くか 時刻 t の隠れ状態 h_t が, 後の時刻 T の隠れ状態 h_T に影響する道筋は,

$$h_t \longrightarrow h_{t+1} \longrightarrow h_{t+2} \longrightarrow \cdots \longrightarrow h_T$$

という1本道です. h_t を直接 h_T に結ぶ近道はなく, 必ず1ステップずつ受け渡されます.

そこで, 各ステップについて「 h_{s-1} を少し変えると h_s が何倍変わるか」を考えます. これが1ステップ分の微分 $\frac{\partial h_s}{\partial h_{s-1}}$ です. たとえば, h_t を少し変えると h_{t+1} が0.5倍変わり, h_{t+1} を少し変えると h_{t+2} が0.5倍変わるなら, h_t の変化は h_{t+2} には $0.5 \times 0.5 = 0.25$ 倍で伝わります. 第2回の連鎖律（合成関数の微分, 歯車のたとえ）と同じで, **1本道の倍率は, 各ステップの倍率の掛け算**です. したがって,

$$\frac{\partial h_T}{\partial h_t} = \frac{\partial h_{t+1}}{\partial h_t} \times \frac{\partial h_{t+2}}{\partial h_{t+1}} \times \cdots \times \frac{\partial h_T}{\partial h_{T-1}}$$

となります. 掛け合わせる数は, t から T までのステップの数, つまり $T - t$ 個です.

誤差逆伝播法では, 時刻 T に届いた誤差 ($\partial L / \partial h_T$) にこの倍率を掛けたものが, 時刻 t に届く誤差 ($\partial L / \partial h_t$) になります. 第2回で見た「誤差は後ろから掛け算で送られる」ことの,

時間方向の版です²。

簡単な場合で計算する 最も簡単な場合として、隠れ状態が1つの数で、活性化関数のない

$$h_t = a h_{t-1} + x_t$$

を考えます。 h_{t-1} を少し変えると h_t は a 倍変わるので、どのステップでも1ステップ分の微分は a です。 したがって

$$\frac{\partial h_T}{\partial h_t} = \underbrace{a \times a \times \cdots \times a}_{T-t \text{ 個}} = a^{T-t}$$

です (図2) 。 誤差もこの割合で逆向きに伝わります。

1ステップ分の微分を $T-t$ 回掛け合わせる ことになります。 $|a| < 1$ なら $T-t$ が大きくなるにつれて指数関数的に0に近づき ($0.9^{50} \approx 0.005$) , $|a| > 1$ なら爆発的に大きくなります ($1.1^{50} \approx 117$) 。 これは第2回の勾配消失・勾配爆発と同じ構造で、しかも同じ重みが何度も掛かります³。

第2回の伝言ゲームを思い出してください。 50人の列で、1人伝えるごとに声が0.9倍になれば、最後の人には0.5%の大きさしか届きません。

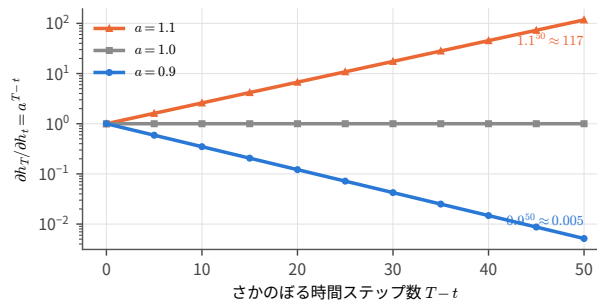


図2 1ステップの微分が a のとき、 $T-t$ ステップさかのぼった誤差の大きさ a^{T-t} (縦軸は対数目盛) 。

その結果、RNN は遠く離れた位置の関係を学習するのが苦手です。「彼」から50単語前の「太郎」に誤差を届けることが難しいのです。

2.3 LSTM

この問題を和らげるために考案されたのが **LSTM** (Long Short-Term Memory, Hochreiter と Schmidhuber, 1997年) です。 LSTM は、隠れ状態とは別に「記憶セル」 c_t をもち、それを

$$c_t = f_t \times c_{t-1} + i_t \times g_t$$

²大学の教科書では、掛け算の記号 $\prod$ (パイ, 和の記号 $\sum$ の掛け算版) を使って, $\frac{\partial h_T}{\partial h_t} = \prod_{s=t+1}^T \frac{\partial h_s}{\partial h_{s-1}}$ と書きます。「 $s=t+1$ から T までの $\frac{\partial h_s}{\partial h_{s-1}}$ をすべて掛け合わせる」という意味です (付録A.2) 。

³ベクトルの場合は、 W_h の固有値の絶対値が1より小さいか大きいかで同じことが起こります (付録A.2) 。 $\tanh$ の導関数は1以下なので、消失の側にさらに傾きます。

のように足し算で更新します⁴。 f_t (忘却ゲート) と i_t (入力ゲート) は0~1の値をとる量で、「前の記憶をどれだけ残すか」「新しい情報 g_t をどれだけ書き込むか」を入力に応じて調節します。

RNN が毎回メモを丸ごと書き直していたのに対して、LSTM は消しゴム (忘却ゲート：どれだけ消すか) とペン (入力ゲート：どれだけ書き足すか) を使い分けて、大事なことはそのまま残しておけるノートです。

$f_t \approx 1$ なら記憶がそのまま次の時刻に渡り、誤差も掛け算で小さくならず届きます。第2回の残差接続 (伝言の列の横に引いた直通の電話線) と同じ発想です。

LSTM は2010年代半ばまで、機械翻訳や音声認識で広く使われました。

2.4 RNN の根本的な限界

LSTM で改善されても、RNN には次の限界が残りました。

1. **情報がすべて h_t を通る**：どんなに長い文でも、過去の情報を固定の大きさのベクトル h_t に詰め込まなければならない。長編小説のあらすじを、ずっと同じ大きさのメモ用紙1枚に書き続けるようなものです。
2. **遠い位置への経路が長い**：位置1の情報が位置 t に届くには、 $t-1$ 回の更新を経る必要がある (図3(a))。
3. **並列計算ができない**： h_t を計算するには h_{t-1} が必要なので、時刻の順に1つずつしか計算できない。1列に並んだバケツリレーで、前の人が終わらないと次の人が動けないのと同じです。GPU の並列計算能力を生かせません。

3 Attention：必要な位置を直接参照する

3.1 考え方

RNN の限界を根本から解決したのが **Attention** (注意機構, attention mechanism) です。発想は単純で、過去の情報を1つのベクトルに詰め込むのではなく、必要なときに、必要な位置の情報を直接取りに行くというものです (図3(b))。

メモ用紙1枚に頼るのをやめて、本そのものを手元に置いておき、必要になったらそのページをめくって見返す——これが Attention です。「彼」が出てきたら、前のページをめくって「太郎」のところを直接読みに行けばよいのです。

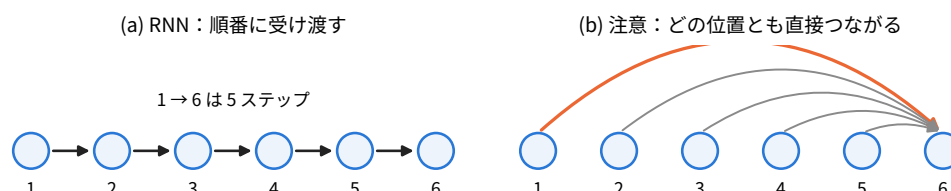


図3 情報の経路。(a) RNN では位置1の情報が位置6に届くまで5回受け渡される。(b) Attention では、どの位置からも1ステップで直接参照できる。

⁴実際には c_t , f_t , i_t , g_t はベクトルで、 $\times$ は成分ごとの積 ($\odot$ と書く) です (付録A.3)。

3.2 Attention の計算

Attention では、各トークン（単語）のベクトルから、次の3種類のベクトルを作ります⁵。

- ・ **クエリ $\mathbf{q}$** (query, 質問) : そのトークンが「どんな情報を探しているか」
- ・ **キー $\mathbf{k}$** (key, 見出し) : そのトークンが「どんな情報をもっているか」の目印
- ・ **バリュー $\mathbf{v}$** (value, 中身) : そのトークンが実際に渡す情報

図書館で、自分の質問（クエリ）を本の背表紙の見出し（キー）と照らし合わせ、よく合う本ほどしっかり読んで中身（バリュー）を集める、というイメージです⁶。1冊だけを選ぶのではなく、合い具合に応じて何冊かを「少しずつ」読むところがポイントです。

位置 i のトークンが、位置 $1, \dots, n$ のトークンを参照する計算は、次の3段階です。

1. **点数** : 自分のクエリ $\mathbf{q}_i$ と、各位置のキー $\mathbf{k}_j$ の内積を計算し、 $\sqrt{d_k}$ で割る。

$$s_j = \frac{\mathbf{q}_i \cdot \mathbf{k}_j}{\sqrt{d_k}} \quad (j = 1, \dots, n)$$

内積が大きいほど「よく合う」ことを表します⁷。

2. **注意の重み** : 点数をソフトマックス関数で確率に直す。

$$a_j = \frac{e^{s_j}}{e^{s_1} + e^{s_2} + \dots + e^{s_n}}$$

重みはすべて正で、合計は1です。これを**注意の重み**といいます。「どの本をどれくらいの割合で読むか」の配分です。

3. **出力** : 各位置のバリューを注意の重みで混ぜ合わせる。

$$\mathbf{O}_i = a_1 \mathbf{V}_1 + a_2 \mathbf{V}_2 + \dots + a_n \mathbf{V}_n$$

つまり各位置の出力は、**全位置のバリューを**、「**どれだけ参照したいか**」の重みで**混ぜ合わせた**ものです。1つの文の中でトークンどうしが参照し合うので、これを**自己注意** (self-attention) といいます。

この計算は、全位置の分をまとめて行列で1つの式 $\text{softmax}(\mathbf{Q}\mathbf{K}^\top/\sqrt{d_k})\mathbf{V}$ と書くことができます（付録A.4）。

3.3 例題

「太郎は花子に本を渡した。彼は……」から4つの単語を取り出し、クエリ・キーの次元を $d_k = 2$ として、次のベクトルを与えます（付録Bのプログラム）。

単語	クエリ $\mathbf{q}$	キー $\mathbf{k}$	バリュー $\mathbf{v}$
太郎	(0, 1)	(2, 1)	(1, 0, 0)
花子	(1, 0)	(1, -1)	(0, 1, 0)
本	(1, 1)	(-1, 0)	(0, 0, 1)
彼	(2, 2)	(0, 1)	(0, 0, 0)

⁵これらは、トークンのベクトルにそれぞれ別の重み行列 W_Q , W_K , W_V を掛けて作ります（付録A.4）。

⁶クエリ・キー・バリューの直観的な説明は、別の資料「microgpt を読む」3.3節や「LLMのしくみ入門」第3節にもあります。

⁷ $\sqrt{d_k}$ で割るのは、次元 d_k が大きいと内積の値が大きくなりすぎ、ソフトマックスの結果が極端（1つだけほぼ1、残りはほぼ0）になるのを防ぐためです。このため、この計算を Scaled Dot-Product Attention（縮尺付き内積注意）と呼びます。

「彼」のクエリ (2, 2) と各キーの内積は、

太郎: $2 \cdot 2 + 2 \cdot 1 = 6$, 花子: $2 \cdot 1 + 2 \cdot (-1) = 0$, 本: $2 \cdot (-1) + 2 \cdot 0 = -2$, 彼: $2 \cdot 0 + 2 \cdot 1 = 2$
です。 $\sqrt{2}$ で割ってソフトマックス関数をとると、注意の重みは

$$(0.929, 0.013, 0.003, 0.055)$$

となり、「彼」は「太郎」を強く参照します。バリューを混ぜ合わせた「彼」の出力は

$$0.929(1, 0, 0) + 0.013(0, 1, 0) + 0.003(0, 0, 1) + 0.055(0, 0, 0) = (0.929, 0.013, 0.003)$$

で、ほぼ「太郎」のバリューそのものです。「彼」というトークンが、「太郎」のページを開いて、その中身をほぼそのまま持ってきたわけです。全単語の注意の重みを図4(a) に示します。

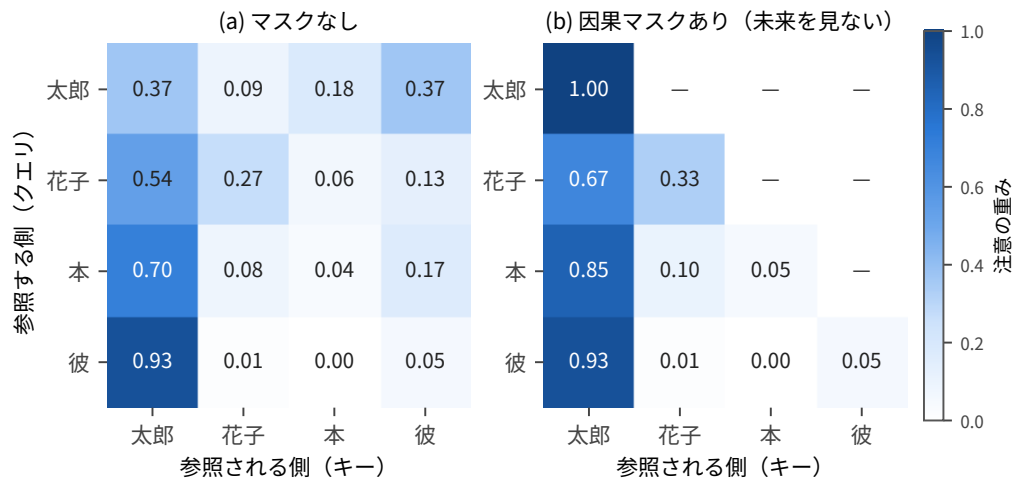


図4 例題の注意の重み。行が参照する側（クエリ）、列が参照される側（キー）で、各行の和は1. (b) は因果マスク（3.4節）をかけたもの。

実際のモデルでは、このクエリ・キー・バリューのベクトルは人が与えるのではなく、それらを作る重みを学習することで、文脈に応じて自動的に作られます。

3.4 因果マスク

言語モデルでは「次の単語を予測する」ので、位置 i の計算でそれより後ろ（未来）の単語を見てはいけません。推理小説を読みながら犯人を当てるのに、最後のページを先に見てはいけないのと同じです。そこで、未来の位置 $j > i$ の点数 s_j を $-\infty$ （マイナス無限大）に置き換えてからソフトマックス関数をとります⁸。

$e^{-\infty} = 0$ なので、未来の位置への注意の重みは0になります（図4(b)）。これを**因果マスク**（causal mask）といいます。未来のページに目隠しをするわけです。最後の単語「彼」はもともと全単語より後ろにあるので、マスクの有無で結果は変わりません。

⁸行列で書くと、点数の行列に「 $j \leq i$ なら 0, $j > i$ なら $-\infty$ 」という行列 M を加えることにあたります（付録 A.5）。

3.5 順序の情報がない：位置符号

自己注意には、次のような性質があります。

入力のトークンの順番を入れ替えると、出力も同じように入れ替わるだけで、各トークンの出力の値は変わらない。

内積と重み付きの和は、どれも順番によらない計算だからです（付録Bのプログラムで、入力を逆順にしても「彼」の出力が同じであることを確かめています）。単語カードを袋に入れて混ぜてしまうと、どの単語があったかはわかって、どんな順に並んでいたかはわからなくなるのと同じです。このままでは「犬が人をかんだ」と「人が犬をかんだ」を区別できません。

そこで、各トークンのベクトルに、位置を表すベクトル（位置符号, positional encoding）を足してから入力します。単語カードの1枚1枚に「1番目」「2番目」と番号を書き込んでおくようなものです。元の Transformer では、位置 pos のベクトルの成分番号 $2i$, $2i + 1$ に

$$PE_{\text{pos}, 2i} = \sin\left(\frac{\text{pos}}{10000^{2i/d}}\right), \quad PE_{\text{pos}, 2i+1} = \cos\left(\frac{\text{pos}}{10000^{2i/d}}\right)$$

が使われました（図5）。 $10000^{2i/d}$ は成分番号 i が大きいほど大きくなるので、成分ごとに周期の違う正弦波になり、小さな番号の成分は速く、大きな番号の成分はゆっくり変化します。

時計の秒針・分針・時計針の組み合わせで時刻を表すように、これらの組み合わせで位置を表します。

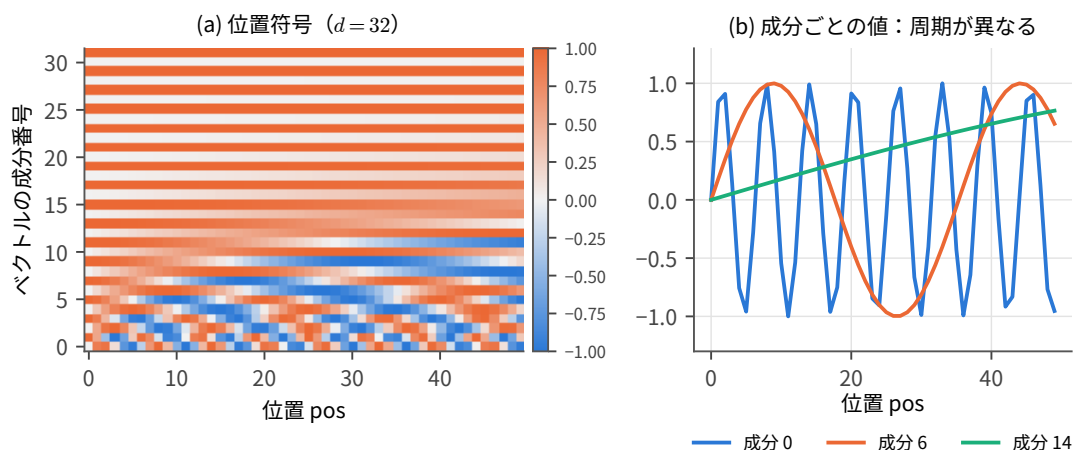


図5 位置符号. (a) 位置（横軸）ごとのベクトル（縦に32成分）。(b) いくつかの成分の値の変化。

最近のモデルでは、クエリとキーのベクトルを位置に応じて回転させる方法（RoPE）など、別の方式もよく使われます。いずれにせよ、**Attention** 自体は順序を知らないで、順序の情報を外から与える必要があるという点は共通です。

4 Transformer

4.1 マルチヘッド注意

実際には Attention を h 組並行して計算します（**マルチヘッド注意**, multi-head attention）。各組（ヘッド）は、それぞれ別の重みでクエリ・キー・バリューを作り、別々に Attention を計算します。最後に h 個のヘッドの出力をつなげて、もう一度変換して混ぜ合わせます（式は付録A.6）。

各ヘッドの次元は $d_k = d/h$ とするのが普通です。同じ文を複数の人が別々の観点で読むようなもので、ヘッドごとに異なる種類の関係（文法的な関係、指示語と指示対象の関係など）を捉えることが期待されています。国語の先生、歴史の先生、理科の先生が同じ文章を読めば、それぞれ違うところに注目する——その全員の読みを持ち寄るわけです。

4.2 Transformer ブロック

Transformer (Vaswani ら, 2017年) は、次の**ブロック**を N 層積み重ねたものです（図6）。

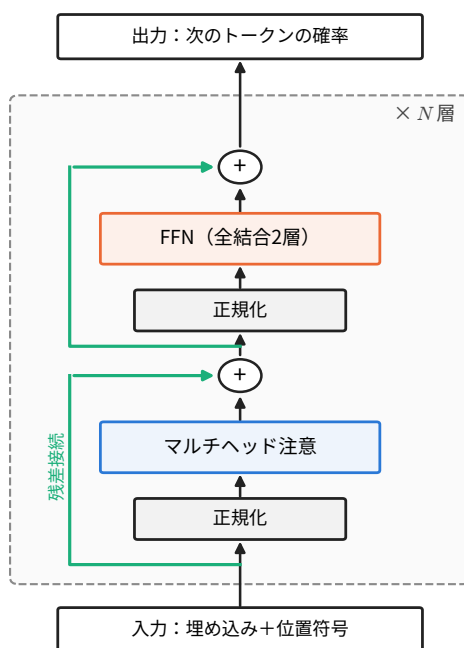


図6 Transformer ブロック（LLM で使われる、出力を生成する側の構成）。緑の線は残差接続。

1つのブロックでは、各トークンのベクトル列 X に対して次の2段階の計算を行います。

$$X \leftarrow X + \text{MultiHead}(\text{Norm}(X))$$

$$X \leftarrow X + \text{FFN}(\text{Norm}(X))$$

- **マルチヘッド注意** (MultiHead)：トークンどうしで情報をやり取りする部分。
- **FFN** (Feed-Forward Network)：各トークンのベクトルを個別に変換する2層の全結合ネットワーク（第1回の多層ネットワークそのもの）。隠れ層の次元は $4d$ 程度が普通です。
- **残差接続** ($X + \dots$) と **正規化** (Norm)：第2回で見た、深いネットワークを学習可能にするための工夫です。

つまり Transformer は、「トークン間の情報交換（Attention）」と「トークンごとの変換（FFN）」を交互に何十回も繰り返すネットワークです。最後の層の出力に全結合層とソフトマックス関数をかけて、次のトークンの確率を得ます。

授業でたとえば、**グループで話し合う時間**（Attention：互いの意見を聞いて取り入れる）と、**各自で考えをまとめる時間**（FFN：聞いたことを自分の中で整理する）を、交互に何十回も繰り返すようなものです。

4.3 パラメータの数

1つのブロックのパラメータは、おおよそ次のとおりです（バイアスと正規化の分は小さいので省略）。 d 行 d 列の重みは d^2 個の数からなります。

部分	重み	パラメータ数
注意	クエリ・キー・バリュー用と、最後の混ぜ合わせ用の4つ (各 $d \times d$)	$4d^2$
FFN	1層目 ($d \rightarrow 4d$) と2層目 ($4d \rightarrow d$) の2つ (各 $4d^2$)	$8d^2$
合計		$12d^2$

たとえば $d = 768$ 、12層のモデルでは、ブロックの部分が $12 \times 12 \times 768^2 \approx 8,500$ 万個、単語の埋め込み（語彙 50,257 語 $\times$ 768）が約3,900万個で、合計約1.24億個になります。これは GPT-2 の最小のモデルの規模です。Llama 3 8B は $d = 4096$ 、32層で、構成が少し異なるため単純にこの式どおりにはなりませんが、同じ考え方で約80億個になります。

4.4 学習と生成

学習では、第2回と同じく、次のトークンの予測の交差エントロピー

$$L = - \sum_t \log p(x_{t+1} | x_1, \dots, x_t)$$

を誤差逆伝播法で最小化します。各位置で「正解の次のトークンに与えた確率」の対数にマイナスを付けて、全位置分を足したものです。

正解（次のトークン）は文章そのものから得られるので、人手でラベルを付ける必要はありません（**自己教師あり学習**、「教師あり学習と教師なし学習」9.2節）。文章を1語ずつ隠した穴埋め問題は、どんな文章からでも自動的に作れるのです。

因果マスクのおかげで、**1つの文のすべての位置の予測を、1回の計算でまとめて行えます**。位置 t の予測は位置 t までしか見ないので、同時に計算しても「カンニング」になりません。穴埋め問題の全問を、答えを見ずに一斉に解けるわけです。RNN のように時刻の順に1つずつ計算する必要がなく、GPU で大量のデータを並列に学習できることが、Transformer の大規模化を可能にした大きな理由です。

生成では、1トークンずつ予測して末尾に加え、それをまた入力にする、という操作を繰り返します⁹。スマートフォンの予測変換で、出てきた候補をタップし続けて文を作るのと同じです。

⁹生成のしくみは、別の資料「microgpt を読む」7節でも、実際のプログラムを使って説明しています。

4.5 RNN との比較

	RNN	Transformer
過去の情報のもち方	固定の大きさの隠れ状態 h_t に要約 (メモ用紙1枚)	全位置のベクトルをそのまま保持 (本そのもの)
遠い位置への経路	距離に比例して長い	1ステップ (直接参照)
学習時の並列計算	できない (時刻の順)	できる (全位置を同時)
系列の長さ n に対する計算量	n に比例	n^2 に比例 (全ペアの内積)
順序の情報	構造に組み込まれている	位置符号で外から与える

Transformer の弱点は、全ペアの注意を計算するため計算量が系列の長さの2乗に比例することです。 n 人が全員と1回ずつ握手すると約 $n^2/2$ 回になるように、トークンが2倍になれば計算は約4倍になります。長い文書を扱うための工夫は、現在も活発に研究されています。

5 まとめ

1. 系列データは順序に意味があり、長さが一定でなく、離れた位置どうしが関係する。
2. RNN は隠れ状態を時間方向に受け渡し、重みを時間方向に共有する。しかし勾配が a^{T-t} のように指数関数的に消失・爆発し、遠い関係を学習しにくく、並列計算もできない。
3. Attention は、クエリとキーの内積で「どこを参照するか」を決め、必要な位置の情報を直接取り出す。全位置をまとめて計算できる。
4. 言語モデルでは因果マスクで未来を隠し、Attention が順序を知らないので位置符号で位置を与える。
5. Transformer は「Attention によるトークン間の情報交換」と「FFN によるトークンごとの変換」を、残差接続と正規化をはさんで何十層も重ねたものである。

シリーズ全体のまとめ

回	学んだこと	前の回とのつながり
第1回	ニューロンを重ね、非線形な活性化関数をはさむと、隠れ層が入力を扱いやすい表現に変換する	ロジスティック回帰＝ニューロン1個
第2回	誤差逆伝播法で全パラメータの勾配を効率よく求め、勾配降下法で学習する	最小二乗法・勾配降下法と同じ「損失の最小化」
第3回	空間方向の重み共有 (畳み込み) で、画像の局所的な特徴を階層的に学習する	第1回の「表現の変換」の画像版
第4回	時間方向の重み共有 (RNN) から、直接参照 (Attention) へ。Transformer が LLM の基盤となった	第2回の勾配消失と残差接続、第3回の重み共有

最小二乗法から LLM まで、一貫しているのは「**モデルを決め、損失関数を決め、勾配を使って損失を最小化する**」という枠組みです。変わってきたのは、モデルの構造 (何をどうつなぐか) と、その規模です。

付録A 行列とベクトルで書く

この付録では、偏微分と行列（大学初年次の線形代数）を使います。

記号 ベクトルは太字の小文字 ($\mathbf{x}_t, \mathbf{h}_t$)，行列は大文字 (W) で書きます。右肩の T は転置（行と列の入れ替え）で、べき乗ではありません。 $\mathbf{q}^\top \mathbf{k}$ は内積です。 $\odot$ は成分ごとの積です。 $\tanh$ は双曲線正接関数， softmax は行ごとに適用するソフトマックス関数です。 A.4節以降では、 n 個のトークンのベクトルを行として縦に並べた $n \times d$ 行列 X を使い、 $Q = XW_Q$ のように右から重み行列を掛けます。

A.1 RNN

$$\mathbf{h}_t = \tanh(W_h \mathbf{h}_{t-1} + W_x \mathbf{x}_t + \mathbf{b}), \quad \hat{\mathbf{y}}_t = \text{softmax}(W_y \mathbf{h}_t + \mathbf{c})$$

です。 W_h, W_x, W_y はすべての時刻で共通です。

A.2 通時的誤差逆伝播法と勾配消失

本文2.2節で説明したように、時刻 T の誤差が時刻 t に届くまでに、

$$\frac{\partial \mathbf{h}_T}{\partial \mathbf{h}_t} = \prod_{s=t+1}^T \frac{\partial \mathbf{h}_s}{\partial \mathbf{h}_{s-1}}$$

と、1ステップ分の微分を $T - t$ 回掛け合わせることになります。 $\prod$ は掛け算の記号です。 隠れ状態がベクトルのとき、 $\partial \mathbf{h}_s / \partial \mathbf{h}_{s-1}$ は「 $\mathbf{h}_{s-1}$ の各成分を少し変えると $\mathbf{h}_s$ の各成分が何倍変わるか」を並べた行列（ヤコビ行列）で、掛け算は行列の積になります。 活性化関数を見れば $\partial \mathbf{h}_s / \partial \mathbf{h}_{s-1} = W_h$ なので、同じ行列 W_h が $T - t$ 回掛かります。 W_h を固有ベクトルの方向に分けて考えると、各方向の成分は固有値 λ の $T - t$ 乗倍になるので、 $|\lambda| < 1$ の方向は消失し、 $|\lambda| > 1$ の方向は爆発します。 本文2.2節のスカラーの場合 (a^{T-t}) は、 W_h が1行1列の場合です。

A.3 LSTM

LSTM の記憶セルの更新は

$$\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \mathbf{g}_t$$

です。 忘却ゲート $\mathbf{f}_t$ と入力ゲート $\mathbf{i}_t$ は、前の隠れ状態と今の入力からシグモイド関数で計算され、各成分が0~1の値をとります。 $\partial \mathbf{c}_t / \partial \mathbf{c}_{t-1}$ は $\mathbf{f}_t$ を対角に並べた行列なので、 $\mathbf{f}_t \approx \mathbf{1}$ なら誤差はほとんど減衰せずに過去へ届きます。

A.4 Scaled Dot-Product Attention

n 個のトークンのベクトルを行に並べた X ($n \times d$) から、

$$Q = XW_Q, \quad K = XW_K, \quad V = XW_V$$

を作り (W_Q, W_K は $d \times d_k$, W_V は $d \times d_v$) ,

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right) V$$

を計算します。

- QK^T は $n \times n$ の行列で, (i, j) 成分は「位置 i のクエリ」と「位置 j のキー」の内積, つまり本文3.2節の点数 s_j (を $\sqrt{d_k}$ 倍したもの) です。
- softmax を行ごとにかけると, 各行が合計1の注意の重みになります。
- それを V に掛けると, 各行が「全位置のバリューを注意の重みで混ぜ合わせたもの」になります。

A.5 因果マスク

点数の行列に

$$M_{ij} = \begin{cases} 0 & (j \leq i) \\ -\infty & (j > i) \end{cases}$$

を加えてから softmax をとります。

$$\text{softmax}\left(\frac{QK^T}{\sqrt{d_k}} + M\right) V$$

A.6 マルチヘッド注意と FFN

ヘッド m ごとに別の重み $W_Q^{(m)}$, $W_K^{(m)}$, $W_V^{(m)}$ をもち,

$$\begin{aligned} \text{head}_m &= \text{Attention}(XW_Q^{(m)}, XW_K^{(m)}, XW_V^{(m)}), \\ \text{MultiHead}(X) &= [\text{head}_1, \dots, \text{head}_h] W_O \end{aligned}$$

です。[] は各ヘッドの出力を横につなげることを表します。FFN は, 各トークンのベクトル $\mathbf{x}$ に対して

$$\text{FFN}(\mathbf{x}) = W_2 f(W_1 \mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2$$

を計算します (W_1 は $4d \times d$, W_2 は $d \times 4d$, f は活性化関数)。

付録B Python コード (標準ライブラリのみ)

3.3~3.5節の例題を計算するプログラムです。外部ライブラリは使わず, Python 3 の標準機能だけで書いてあります。

```
# 自己注意 (Scaled Dot-Product Attention) の計算 (標準ライブラリのみ)
import math, unicodedata

def pad(t, w):
    """全角文字を幅2として右寄せする"""
    n = sum(2 if unicodedata.east_asian_width(c) in "FW" else 1 for c in t)
    return " " * (w - n) + t

tokens = ["太郎", "花子", "本", "彼"]
Q = [[0, 1], [1, 0], [1, 1], [2, 2]] # 各トークンのクエリ (d_k = 2)
K = [[2, 1], [1, -1], [-1, 0], [0, 1]] # キー
```

```

V = [[1, 0, 0], [0, 1, 0], [0, 0, 1], [0, 0, 0]] # バリュー

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

def attention(Q, K, V, causal=False):
    dk = len(Q[0])
    A, out = [], []
    for i, q in enumerate(Q):
        s = [sum(a * b for a, b in zip(q, k)) / math.sqrt(dk) for k in K]
        if causal:
            # 未来 (j > i) は見ない
            s = [v if j <= i else -math.inf for j, v in enumerate(s)]
        a = softmax(s)
        A.append(a)
        out.append([sum(a[j] * V[j][c] for j in range(len(V))) for c in range(len(V[0]))])
    return A, out

for causal in (False, True):
    A, out = attention(Q, K, V, causal)
    print("因果マスクあり" if causal else "因果マスクなし")
    print(" " + "".join(pad(t, 8) for t in tokens))
    for t, row in zip(tokens, A):
        print(pad(t, 4) + " " + "".join(f"{v:8.3f}" for v in row))
    print("「彼」の出力 =", [round(v, 3) for v in out[3]])

# 入力の順番を入れ替えると、出力も同じように入れ替わるだけ（順序の情報がない）
perm = [3, 2, 1, 0]
_, out1 = attention(Q, K, V)
_, out2 = attention([Q[p] for p in perm], [K[p] for p in perm], [V[p] for p in perm])
print("順番を逆にした入力での「彼」の出力 =", [round(v, 3) for v in out2[0]])

```

実行結果：

```

因果マスクなし
      太郎   花子     本     彼
太郎   0.365  0.089  0.180  0.365
花子   0.539  0.266  0.065  0.131
本     0.703  0.084  0.042  0.171
彼     0.929  0.013  0.003  0.055
「彼」の出力 = [0.929, 0.013, 0.003]
因果マスクあり
      太郎   花子     本     彼
太郎   1.000  0.000  0.000  0.000
花子   0.670  0.330  0.000  0.000
本     0.848  0.102  0.050  0.000
彼     0.929  0.013  0.003  0.055
「彼」の出力 = [0.929, 0.013, 0.003]
順番を逆にした入力での「彼」の出力 = [0.929, 0.013, 0.003]

```