

深層学習（2）誤差逆伝播法と学習の実際

（高校生・大学初年次向け）

重みをデータから求める方法

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1回では、ニューラルネットワークが「重み付きの和をとり、バイアスを足し、非線形な活性化関数に通す」ことを繰り返す関数であることを見ました。そこでは重みを人が決めましたが、実際のネットワークには数万～数十億個の重みがあり、人が決めることはできません。

第2回では、**重みをデータから学習する方法**を扱います。考え方はこれまでと同じく「損失関数を勾配降下法で最小化する」ことです。問題は、**膨大な数の重みについて、勾配をどうやって効率よく計算するか**です。その答えが**誤差逆伝播法**（バックプロパゲーション, backpropagation）です。

第1回のたとえで言えば、ネットワークには何万個もの「つまみ」がついています。出力の外れ具合（損失）を見ながら、どのつまみをどちらにどれだけ回せばよいかを、一度に全部知る方法——それが誤差逆伝播法です。

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（合成関数の微分、指数関数、数列の和 $\sum$ ）を使って読めるように書きました。ところどころ高校の範囲を少し越えるところ（偏微分）がありますが、その場でことばの説明を添えています。行列とベクトルを使った一般的な書き方は、**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

- $\frac{\partial L}{\partial w}$ は**偏微分**です。ほかのパラメータをすべて止めておいて、 w だけを少し動かしたときの L の変化の割合を表します。「 L の w 方向の傾き」と考えてください。すべてのパラメータについての偏微分をまとめて**勾配** (gradient) と呼びます。
- $W^{(1)}$, $\delta^{(2)}$ のように右肩に**かっこ付き**で書いた数字は**層の番号**で、べき乗ではありません。

- $\sigma(z) = 1/(1+e^{-z})$ はシグモイド関数で、その導関数は $\sigma'(z) = \sigma(z)(1-\sigma(z))$ です。 η (イータ) は学習率です。

1 なぜ効率のよい勾配計算が必要か

1.1 素朴な方法：数値微分

勾配を求める最も素朴な方法は、重みを1つずつ少しだけ動かして損失の変化を測る**数値微分**です。パラメータ w を前後に小さな量 ε (イプシロン) だけ動かして、

$$\frac{\partial L}{\partial w} \approx \frac{L(w + \varepsilon) - L(w - \varepsilon)}{2\varepsilon}$$

とします (ほかのパラメータは止めておきます)。微分の定義そのものです。つまみを1個だけ少し回して、損失のメーターがどれだけ動くかを見るわけです。

この方法では、パラメータ1個につき損失の計算 (順伝播) が2回必要です。パラメータが P 個あれば、1回の勾配計算に $2P$ 回の順伝播が必要になります。第1回の手書き数字の例 ($P \approx 8$ 万) では1回の更新に16万回の順伝播、80億パラメータの LLM では160億回です。8万個のつまみを1個ずつ回して試しては、日が暮れてしまいます。

1.2 誤差逆伝播法の効率

誤差逆伝播法を使うと、すべてのパラメータについての勾配が、順伝播1回とほぼ同じ手間の逆向きの計算1回で求まります。パラメータの数によらず、計算量はおおよそ順伝播の2~3倍で済むのです。

会社の売上が目標に届かなかったとき、社長が社員一人ひとりに順番に聞いて回るのは大変です。代わりに、社長は各部長に「あなたの部の責任はこれだけ」と伝え、部長は各課長に、課長は各係員に……と、上から下へ責任を分配していけば、一度の伝達で全員が自分の責任の大きさを知ることができます。誤差逆伝播法は、出力の「外れ」(誤差) を、出力側から入力側へこのように分配していく方法です。

以下、そのしくみを見ていきます。

2 連鎖律と計算グラフ

2.1 合成関数の微分 (連鎖律)

$y = f(u)$, $u = g(x)$ のとき、

$$\frac{dy}{dx} = \frac{dy}{du} \cdot \frac{du}{dx}$$

でした (数学Ⅲの合成関数の微分)。これを**連鎖律** (chain rule) といいます。「 x が少し変わると u が $\frac{du}{dx}$ 倍変わり、 u が少し変わると y が $\frac{dy}{du}$ 倍変わるので、全体では掛け算になる」という意味です。

かみ合った歯車を思い浮かべてください。歯車 A を1回転させると B が3回転し、B を1回転させると C が2回転するなら、A を1回転させたとき C は $3 \times 2 = 6$ 回転します。途中の

歯車の倍率を掛け合わせれば、全体の倍率がわかる——これが連鎖律です。

変数が多い場合も同じように考えられます。 L が途中の量 $u_1, \dots, u_m$ を通じて x に依存しているなら、

$$\frac{\partial L}{\partial x} = \sum_{k=1}^m \frac{\partial L}{\partial u_k} \cdot \frac{\partial u_k}{\partial x}$$

です。 x から L への「経路」が m 本あり、それぞれの経路で影響が伝わるので、経路ごとの影響を足し合わせるわけです。川の水が何本かの支流に分かれて流れ、下流で再び合流するとき、下流の水量は各支流の水量の合計になるのと同じです。

経路ごとに微分を掛け、すべての経路について足す、と覚えておけば十分です。

2.2 計算グラフ

簡単な例として、 $f = (x + y)z$ を考えます。途中の値を $q = x + y$ とおくと $f = qz$ です。この計算の流れを図にしたものを**計算グラフ** (computation graph) といいます (図1)。

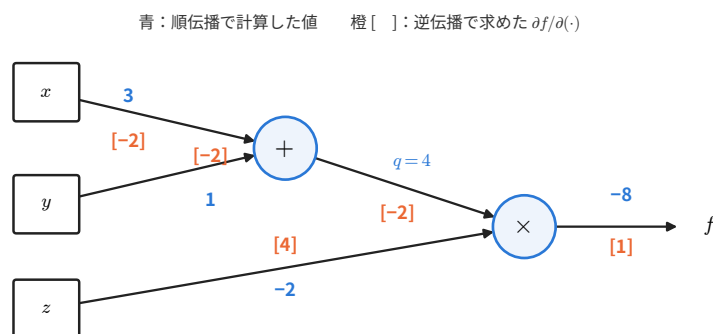


図1 $f = (x + y)z$ の計算グラフ。 $x = 3$, $y = 1$, $z = -2$ のとき、青の数字は左から右への計算（順伝播）で得られる値、橙の [] は右から左への計算（逆伝播）で得られる偏微分 $\partial f / \partial (\cdot)$ 。

順伝播（左から右） $q = 3 + 1 = 4$, $f = 4 \times (-2) = -8$ 。

逆伝播（右から左） 出力から入力に向かって、各点での $\partial f / \partial (\cdot)$ を求めます。

1. 出発点は $\partial f / \partial f = 1$ 。
2. 掛け算 $f = qz$ では、 q を少し変えると f は z 倍変わるので $\partial f / \partial q = z = -2$ 。同様に $\partial f / \partial z = q = 4$ 。
3. 足し算 $q = x + y$ では $\partial q / \partial x = \partial q / \partial y = 1$ なので、連鎖律により $\partial f / \partial x = \partial f / \partial q \times 1 = -2$, $\partial f / \partial y = -2$ 。

(確かめ: x を3から3.01に増やすと $f = (4.01)(-2) = -8.02$ で、0.01の -2 倍だけ変わります。)

ここで大事ななのは、各ノードは「自分の出力についての微分」を後ろから受け取り、自分の局所的な微分を掛けて前に渡すだけでよいことです。

- 足し算のノードは、受け取った値をそのまま両方に渡す。
- 掛け算のノードは、受け取った値に「相手の入力値」を掛けて渡す。

バケツリレーと同じで、各人は隣から受け取ったものを少し加工して次に渡すだけです。全体を見渡す必要はありません。

どんなに複雑な計算も単純なノードの組み合わせなので、この局所的な規則を後ろから順に適用すれば、すべての入力についての微分が1回の逆向きの計算で求まります。これが誤差逆伝播法の本質です。途中の値（ここでは $q = 4$ ）を順伝播のときに記録しておき、逆伝播で再利用する点も重要です。

3 ニューラルネットワークの逆伝播

3.1 2層ネットワーク

第1回の XOR と同じ形の、入力2個・隠れ層2個・出力1個のネットワークを考えます。活性化関数はすべてシグモイド関数とします。

- ・ 隠れ層 ($k = 1, 2$) : $z_k^{(1)} = W_{k1}^{(1)}x_1 + W_{k2}^{(1)}x_2 + b_k^{(1)}$, $h_k = \sigma(z_k^{(1)})$
- ・ 出力層 : $z^{(2)} = W_1^{(2)}h_1 + W_2^{(2)}h_2 + b^{(2)}$, $\hat{y} = \sigma(z^{(2)})$

$W_{kj}^{(1)}$ は「入力 j から隠れユニット k への重み」、 $W_k^{(2)}$ は「隠れユニット k から出力への重み」です。パラメータは重み6個とバイアス3個の計9個です。損失は、前の資料のロジスティック回帰と同じ交差エントロピー

$$L = -[y \log \hat{y} + (1 - y) \log(1 - \hat{y})]$$

です (y は正解で0か1)。

3.2 逆伝播の式

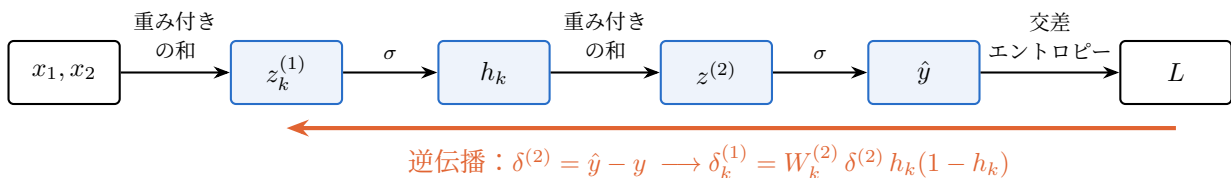


図2 2層ネットワークの順伝播（黒の矢印、左から右）と逆伝播（橙の矢印、右から左）。

各層の z についての偏微分を δ （デルタ）と書き、誤差と呼ぶことにします。先ほどのたとえば、各部署に割り当てられた「責任の大きさ」にあたります。出力側から順に求めます。

出力層 前の資料（ロジスティック回帰の勾配）と同じ計算で¹,

$$\delta^{(2)} = \frac{\partial L}{\partial z^{(2)}} = \hat{y} - y$$

です。「予測と正解のずれ」そのものです。 $z^{(2)} = W_1^{(2)}h_1 + W_2^{(2)}h_2 + b^{(2)}$ なので、 $W_k^{(2)}$ を少し変えると $z^{(2)}$ は h_k 倍変わります。連鎖律により、出力層の重みとバイアスの勾配は

$$\frac{\partial L}{\partial W_k^{(2)}} = \delta^{(2)} h_k, \quad \frac{\partial L}{\partial b^{(2)}} = \delta^{(2)}$$

と求まります。

¹前の資料「教師あり学習と教師なし学習」の4.5節と付録B.3. シグモイド関数と交差エントロピーを組み合わせると、連鎖律の途中で $\hat{y}(1 - \hat{y})$ が約分されて、この簡単な形になります。

隠れ層 誤差を1つ前の層に送ります。 h_k を少し変えると $z_k^{(2)}$ は $W_k^{(2)}$ 倍変わり、 $z_k^{(1)}$ を少し変えると h_k は $\sigma'(z_k^{(1)}) = h_k(1 - h_k)$ 倍変わります。連鎖律で掛け合わせると、

$$\delta_k^{(1)} = \frac{\partial L}{\partial z_k^{(1)}} = \frac{\partial L}{\partial z^{(2)}} \cdot \frac{\partial z^{(2)}}{\partial h_k} \cdot \frac{\partial h_k}{\partial z_k^{(1)}} = \delta^{(2)} W_k^{(2)} h_k(1 - h_k)$$

です。これから隠れ層の重みとバイアスの勾配が、出力層と同じ形で

$$\frac{\partial L}{\partial W_{kj}^{(1)}} = \delta_k^{(1)} x_j, \quad \frac{\partial L}{\partial b_k^{(1)}} = \delta_k^{(1)}$$

と求まります。

どの層でも、勾配は次の形をしています。

重みの勾配 = (その重みの行き先に後ろから届いた誤差 δ) $\times$ (その重みに入ってきた値)

そして誤差 δ は、後ろの層の誤差に、つながっている重みを掛け、活性化関数の導関数を掛けることで、出力側から入力側へ順に送られていきます。

責任のたとえで言えば、大きな重みでつながっている部下ほど大きな責任を分けられ、大きな値を送り込んでいた重みほど大きく修正されます。結果に大きく影響したところほど、大きく直すのです。

3.3 一般の多層ネットワーク

層がもっと多くても、隠れユニットがもっと多くても、まったく同じ形になります。隠れユニットが複数の出力先につながっているときは、連鎖律の「すべての経路について足す」にしたがって、各出力先から届く「誤差 $\times$ 重み」を足し合わせてから、活性化関数の導関数を掛けます。複数の上司から責任を分けられた人は、その合計を引き受ける、ということです。

順伝播では入力から出力へ値を送り、逆伝播では同じ重みを使って出力から入力へ誤差を送ります。各層で必要なのは、その層に入ってきた値と、その層の z の値だけで、これらは順伝播のときに記録しておきます。大学では、これを行列を使って簡潔に書きます（付録A）。

3.4 勾配チェック

逆伝播のプログラムは間違いやすいので、小さなネットワークで数値微分と比べて確かめるのが定石です（勾配チェック）。計算問題を別の解き方で解いて答えを照らし合わせる「検算」と同じです。付録B.2のプログラムで、XOR ネットワークの9個のパラメータすべてについて比べると、差は 10^{-11} 程度で、逆伝播の式が正しいことが確かめられます。

パラメータ	逆伝播	数値微分	差
$W_{11}^{(1)}$	-0.00844318	-0.00844318	1.4×10^{-12}
$W_1^{(2)}$	-0.19033744	-0.19033744	9.2×10^{-12}
$b^{(2)}$	-0.38068435	-0.38068435	1.1×10^{-11}

(9個のうち3個を抜き出したもの)

4 例題：XOR を学習する

4.1 設定

第1回で人が重みを決めた XOR を、今度はデータから学習させます（付録B.1のプログラム）。

- ・ ネットワーク：入力2・隠れ層2・出力1，活性化関数はすべてシグモイド関数
- ・ 初期値：重みは -1 から 1 の一様乱数，バイアスは 0
- ・ 損失：4つのデータの交差エントロピーの和
- ・ 更新：勾配降下法，学習率 $\eta = 1.0$ ，5000回

4.2 結果：うまくいく場合と、いかない場合

乱数の種（seed）を変えて，2つの初期値 A，B から学習させました（図3）。

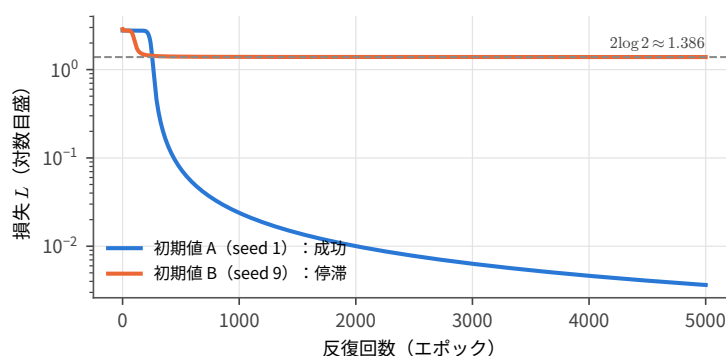


図3 損失の変化（縦軸は対数目盛）。初期値 A では損失が0に向かって減り続けるが，初期値 B では $2\log 2 \approx 1.386$ 付近で止まってしまう。

反復回数	0	100	500	1000	2000	5000
初期値 A の損失	2.774	2.772	0.075	0.024	0.010	0.004
初期値 B の損失	2.874	2.169	1.402	1.393	1.389	1.387

学習後の出力は次のとおりです。

入力	正解	初期値 A の出力	初期値 B の出力
(0, 0)	0	0.001	0.000
(0, 1)	1	0.999	1.000
(1, 0)	1	0.999	0.500
(1, 1)	0	0.001	0.500

図4は，入力平面の各点でのネットワークの出力を濃淡で表したものです。

初期値 A 最初の約200回は損失がほとんど減らない「停滞期」がありますが，その後急に減り，XOR を正しく学習します。第1回の図3と同じく，2本の直線で平面を3つの帯に分けています。

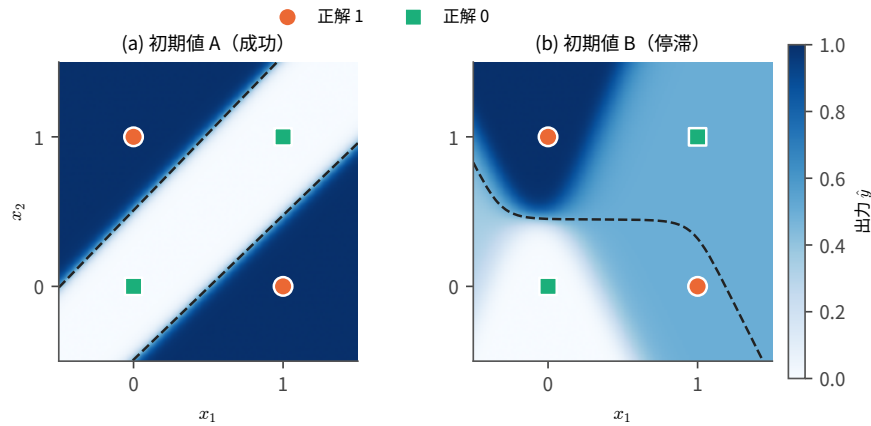


図4 学習後のネットワークの出力 $\hat{y}$ (濃いほど1に近い) . 破線は $\hat{y} = 0.5$ の境界. (a) では2本の境界線で XOR が正しく分けられている. (b) では右側の2点を分けられず、どちらにも0.5を出力している.

隠れ層の出力 (h_1, h_2) を見ると、入力 $(0, 0)$ と $(1, 1)$ はどちらも $(0.99, 0.99)$ に、入力 $(0, 1)$ は $(1.00, 0.01)$ に、入力 $(1, 0)$ は $(0.01, 1.00)$ に移されています.

第1回で人が設計した OR と NAND とは違う変換ですが、やはり**正解0の点を1か所に集め、直線で分けられる配置に変換**しています. 第1回のたとえば、人とは別の折り方で紙を折りたたんだのです. ネットワークは、人とは別の「解き方」を自分で見つけたわけです.

初期値 B 損失が $2 \log 2 \approx 1.386$ 付近で止まります. これは、2つの入力について0.5を出力し (損失 $\log 2$ ずつ) , 残りの2つを正しく当てている状態です². 勾配がほぼ0になる場所に捕まって、そこから抜け出せなくなっています.

4.3 損失関数の地形

勾配降下法は、**濃い霧の中で山を下りる**ようなものです. 遠くは見えず、わかるのは足元の傾きだけなので、とにかく下り坂の方向へ一歩ずつ進みます.

線形回帰やロジスティック回帰の損失は、おわんのような形 (下に凸な関数) で、谷底は1つでした. どこから下りても同じ谷底に着きます. しかし隠れ層のあるネットワークの損失は凸ではなく、**くぼ地 (谷底) が複数あったり、平らな場所 (峠や台地) があったり**します. どこに行き着くかは、出発点によって変わります.

鞍点 (あんてん) とは、馬の鞍 (くら) の形のように、ある方向に見ると谷底、別の方向に見ると山頂になっている点です. 地形でいえば**峠**にあたります. 峠は、尾根に沿って見れば一番低い所、谷に沿って見れば一番高い所です. そこでは傾きが0になるので、勾配降下法の動きが止まりやすくなります.

この例では、隠れユニットを2個から3個に増やすと、100通りの初期値のうち失敗は0回でした (2個のときは100回中14回失敗) . ユニットの増やすと、損失の地形がなだらかになり、よい解にたどり着きやすくなるのが経験的に知られています. 実際の深層学習では、ネ

²正解に確率0.5を与えると、そのデータの損失は $-\log 0.5 = \log 2 \approx 0.693$ です. 2つのデータでそうなるので $2 \log 2$ です.

ットワークを十分大きくし、後で述べる工夫を組み合わせることで、この問題はあまり深刻にならないことがわかっています。

5 確率的勾配降下法

5.1 ミニバッチ

これまでは、損失を全データの和（平均）として勾配を計算していました。しかしデータが数百万件あると、1回の更新のためにすべてのデータで順伝播・逆伝播を行うことになり、時間がかかります。

そこで、毎回データから少数（たとえば32件）を無作為に選び、そのミニバッチ（mini-batch）だけで勾配を計算して更新します。各パラメータ w について

$$w \leftarrow w - \eta \times (\text{ミニバッチの各データについての勾配 } \partial \ell_i / \partial w \text{ の平均})$$

とします（ ℓ_i はデータ i 1個の損失）³。これを**確率的勾配降下法**（Stochastic Gradient Descent, SGD）といいます。全データを一通り使い終わることを**1エポック**（epoch）と呼びます。

世論調査と同じ発想です。国民全員に聞かなくても、無作為に選んだ一部の人に聞けば、全体の傾向はおおよそわかります。ミニバッチの勾配は全データの勾配の「推定値」で、毎回少しずれますが、平均すれば正しい方向を向いています。

そのため毎回の更新方向はぶれますが、全体としては最小点に近づいていきます（図5）。

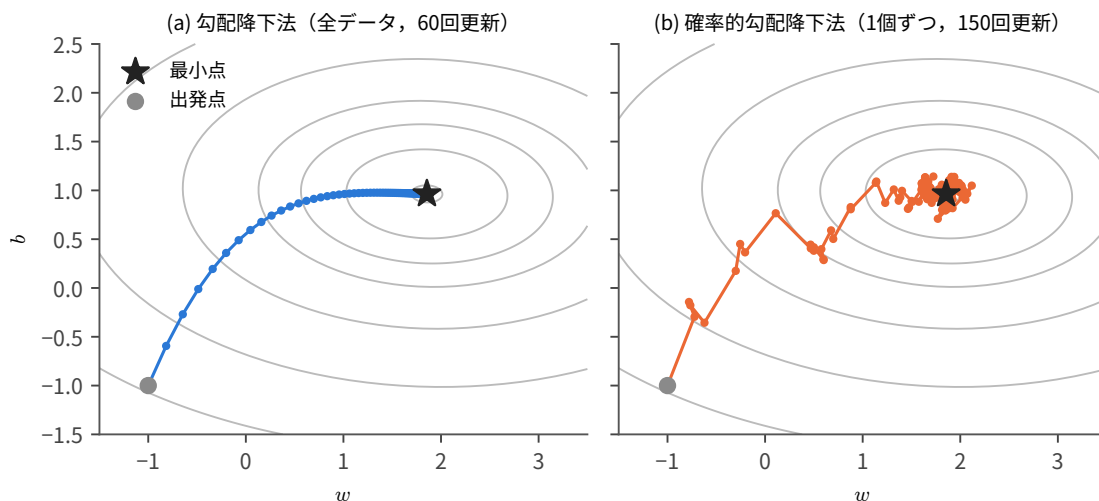


図5 直線の当てはめ（ $y = wx + b$ ，データ50個）での (a) 勾配降下法と (b) 確率的勾配降下法（データ1個ずつ，3エポック）のパラメータの動き。灰色の線は損失の等高線，星印が最小点。

図5(b)のように、確率的勾配降下法の経路はジグザグで、最小点の近くでもふらつき続けます。

³大学の教科書では、すべてのパラメータをまとめたベクトルを θ ，ミニバッチを B ，その個数を $|B|$ とし、 $\theta \leftarrow \theta - \eta \frac{1}{|B|} \sum_{i \in B} \nabla_{\theta} \ell_i$ と書きます。 $\nabla_{\theta} \ell_i$ は ℓ_i の勾配（各パラメータについての偏微分を並べたベクトル）です。

それでも、1回の更新の計算量が小さいので、同じ計算時間でずっと多くの更新ができます。また、このふらつきが浅いくぼ地や峠から抜け出す助けになるとも考えられています。

5.2 更新方法の改良

基本のSGDを改良した方法がいくつも使われています。代表的なものを挙げます。

- ・ **モメンタム** (momentum, 慣性) : 過去の勾配の移動平均 v を使って更新します。各パラメータ w について、

$$v \leftarrow \beta v + \frac{\partial L}{\partial w}, \quad w \leftarrow w - \eta v \quad (\beta \approx 0.9)$$

とします。ボールが坂を転がるときの慣性のように、ジグザグを打ち消して一定方向の動きを加速します。小さなくぼみなら、勢いで乗り越えることもできます。

- ・ **Adam** : 勾配の移動平均に加えて、勾配の2乗の移動平均も記録し、パラメータごとに学習率を自動調整します。現在最も広く使われている方法の一つです⁴。

6 勾配消失問題

6.1 深いネットワークで何が起きるか

各層にユニットが1個ずつしかない、細長いネットワークで考えてみます。

$$x \xrightarrow{w_1} h_1 \xrightarrow{w_2} h_2 \xrightarrow{w_3} \cdots \xrightarrow{w_L} h_L \longrightarrow L$$

各層は $h_l = f(z_l)$, $z_l = w_l h_{l-1} + b_l$ です。3.2節と同じように連鎖律で逆向きにたどると、入力に近い層の誤差は

$$\delta_1 = f'(z_1) \cdot w_2 f'(z_2) \cdot w_3 f'(z_3) \cdots w_L f'(z_L) \cdot \frac{\partial L}{\partial h_L}$$

と、各層の活性化関数の導関数 f' を何度も掛け合わせた形になります（ユニットが多い一般の場合は付録A.4）。歯車のたとえで言えば、倍率が1より小さい歯車を何段もつなぐと、最後の歯車はほとんど回らなくなります。

シグモイド関数の導関数 $\sigma'(z) = \sigma(z)(1 - \sigma(z))$ の最大値は $z = 0$ での 0.25 です（図6(a)）⁵。これを10回掛けると $0.25^{10} \approx 10^{-6}$ になり、入力に近い層にはほとんど誤差が届きません。その結果、入力に近い層の重みがほとんど更新されず、学習が進まなくなります。これを勾配消失問題 (vanishing gradient problem) といいます。

伝言ゲームで、1人伝えるごとに声が4分の1の大きさになるとしたら、10人目にはもう何も聞こえません。入力に近い層は、「どう直せばよいか」という知らせを受け取れないのです。

逆に、掛ける値が1より大きいと勾配が爆発的に大きくなることもあります（勾配爆発）。伝言ゲームで声がどんどん大きくなって、叫び声になってしまう状態です。

図6(b) は、10層のネットワークで実際に逆伝播を行い、各層の重みの勾配の大きさを比べたものです。シグモイド関数では、出力側（10層目）から入力側（1層目）に向かって勾配が1層ごとに数分の1ずつ小さくなり、1層目では10層目の約300万分の1です。

⁴Adam のしくみと式は、別の資料「microgptを読む」の4.4節と付録A.6で説明しています。

⁵ $s = \sigma(z)$ とおくと $0 < s < 1$ で、 $s(1-s) = \frac{1}{4} - (s - \frac{1}{2})^2 \leq \frac{1}{4}$ です。等号は $s = \frac{1}{2}$ 、つまり $z = 0$ のときです。

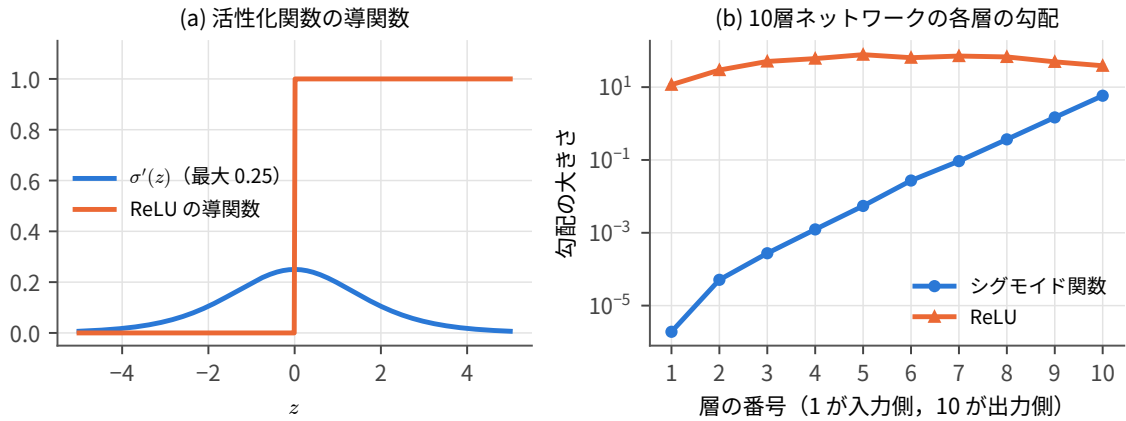


図6 (a) シグモイド関数と ReLU の導関数. (b) 10層（各層50ユニット）のネットワークで、ランダムな重みのもとで逆伝播したときの各層の勾配の大きさ（縦軸は対数目盛）。

6.2 対策

- **ReLU を使う**：ReLU の導関数は $z > 0$ でいつも1なので、掛け合わせても小さくなりません（声の大きさが変わらない伝言）。図6(b)でも、ReLU では各層の勾配がほぼ同じ大きさに保たれています。これが ReLU が広く使われるようになった大きな理由です。
- **初期値の工夫**：各層の出力のばらつきが層を通るたびに大きくも小さくもならないように、重みの初期値の分散を、そのユニットへの入力の数 n に合わせて決めます。ReLU では分散 $2/n$ （He の初期化）、シグモイド関数や $\tanh$ では分散 $1/n$ 程度（Xavier の初期化）がよく使われます⁶。伝言の最初に、各人の声の大きさをちょうどよく調整しておくようなものです。
- **残差接続（residual connection）**： $h^{(l)} = h^{(l-1)} + F(h^{(l-1)})$ のように、層の入力をそのまま出力に足す「近道」を設けます。逆伝播のとき、誤差がこの近道を通してそのまま入力側に届くので⁷、100層を超えるネットワークも学習できるようになりました（第4回の Transformer でも使われています）。伝言ゲームの列の横に、最初の人から最後の人まで直接つながる電話線を引いておくようなものです。
- **正規化層**：各層の出力を平均0・分散1程度にそろえる層（バッチ正規化、レイヤー正規化）をはさみ、値の大きさが層ごとにずれていくのを防ぎます。伝言の途中で、ときどき声の大きさを標準にそろえ直すわけです。

7 過学習への対策

ニューラルネットワークはパラメータが多いため、前の資料で見た**過学習**（訓練データにはよく合うが、未知のデータをうまく予測できない状態）が起きやすくなります。試験勉強で、過去問の答えを丸暗記した生徒を考えてください。過去問なら満点ですが、数字が少し違う新しい問題が出ると解けません。つまみが多すぎるネットワークは、訓練

⁶ 入力 n 個あると、重み付きの和は n 個の項の和なので、そのばらつき（分散）はおおよそ「重みの分散 $\times n \times$ 入力の大きさの2乗の平均」になります。重みの分散を $1/n$ 程度にしておけば、 n によらず出力のばらつきがそろいます。ReLU では負の側の半分が0になるので、その分を補って2倍にします。

⁷ $h + F(h)$ を h で微分すると $1 + F'(h)$ です。「1+」の部分のおかげで、 F' が小さくても掛け合わせた値が0に近づきません。

データの「答え」を丸暗記してしまいやすいのです。

代表的な対策は次のとおりです。

- **正則化（重み減衰）**：損失に $\lambda \times (\text{重みの2乗の和})$ を加えて、重みが大きくなりすぎるのを防ぎます。前の資料のリッジ回帰と同じ考え方です。
- **早期終了**：学習中に、訓練に使っていない検証データでの損失を記録しておき、それが増え始めたところで学習を打ち切ります。過去問の点は上がり続けているのに、模試（検証データ）の点が下がり始めたら、丸暗記が始まった合図です。
- **ドロップアウト**：学習時に、隠れ層のユニットを毎回ランダムに一定割合（たとえば50%）無効にします。スポーツのチームで、毎回の練習でランダムに何人か休ませると、特定のエースに頼らず誰もが動けるチームになるのと似ています。特定のユニットに頼りすぎない、頑健な表現を学ぶように促す効果があります。予測時にはすべてのユニットを使います。
- **データ拡張**：画像を少し回転・平行移動・反転させるなどして、訓練データを人工的に増やします。同じ問題の数字を変えた類題を作って練習するようなものです。
- **データを増やす**：結局のところ、最も確実な対策です。

8 まとめ

1. 勾配を数値微分で求めると、パラメータの数に比例した計算が必要になり、大きなネットワークでは現実的でない。
2. 誤差逆伝播法は、連鎖律を出力側から順に適用し、すべてのパラメータの勾配を順伝播と同程度の計算量で求める。
3. 重みの勾配は「後ろから届いた誤差 $\delta \times$ その重みに入ってきた値」であり、誤差 δ は「後ろの層の誤差 $\times$ 重み $\times$ 活性化関数の導関数」として出力側から入力側へ送られる。
4. ニューラルネットワークの損失は凸ではなく、初期値によっては学習が停滞する。ネットワークを大きくするとこの問題は起きにくくなる。
5. 実際の学習では、ミニバッチによる確率的勾配降下法、ReLU や初期値の工夫による勾配消失の回避、正則化などによる過学習の抑制を組み合わせる。

次回予告 ここまでのネットワークは、入力すべての成分をすべてのユニットにつなぐ「全結合」でした。第3回では、画像のように空間的な構造をもつデータに適した**畳み込みニューラルネットワーク**を扱います。

付録A 行列とベクトルで書く逆伝播

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（ $\mathbf{x}$, $\mathbf{h}$ ），行列は大文字（ W ）で書き、ベクトルは縦ベクトルとします。右肩の T は転置（行と列の入れ替え）で、ベキ乗ではありません。 $\mathbf{a} \odot \mathbf{b}$ は**成分ごとの積**（アダマール積）です： $(a_1, a_2)^T \odot (b_1, b_2)^T = (a_1 b_1, a_2 b_2)^T$ 。 $\frac{\partial L}{\partial W}$ は、 W の各成分で L を偏微分した値を W と同じ形に並べた行列です。関数 f をベクトルに作用させるときは、各成分に別々に作用させます。

A.1 2層ネットワーク

3.1節のネットワークは

$$\mathbf{z}^{(1)} = W^{(1)}\mathbf{x} + \mathbf{b}^{(1)}, \quad \mathbf{h} = \sigma(\mathbf{z}^{(1)}), \quad z^{(2)} = W^{(2)}\mathbf{h} + b^{(2)}, \quad \hat{y} = \sigma(z^{(2)})$$

と書けます。 $W^{(1)}$ は 2×2 , $W^{(2)}$ は 1×2 の行列です。 3.2節の結果は,

$$\delta^{(2)} = \hat{y} - y, \quad \frac{\partial L}{\partial W^{(2)}} = \delta^{(2)} \mathbf{h}^\top, \quad \frac{\partial L}{\partial b^{(2)}} = \delta^{(2)}$$

$$\boldsymbol{\delta}^{(1)} = \frac{\partial L}{\partial \mathbf{z}^{(1)}} = (W^{(2)\top} \delta^{(2)}) \odot \sigma'(\mathbf{z}^{(1)}), \quad \frac{\partial L}{\partial W^{(1)}} = \boldsymbol{\delta}^{(1)} \mathbf{x}^\top, \quad \frac{\partial L}{\partial \mathbf{b}^{(1)}} = \boldsymbol{\delta}^{(1)}$$

とまとめられます。

A.2 式の導出

出力層の誤差 $L = -[y \log \hat{y} + (1 - y) \log(1 - \hat{y})]$, $\hat{y} = \sigma(z^{(2)})$ とし, $\sigma'(z) = \sigma(z)(1 - \sigma(z))$ を使うと

$$\delta^{(2)} = \frac{dL}{d\hat{y}} \cdot \frac{d\hat{y}}{dz^{(2)}} = \left(-\frac{y}{\hat{y}} + \frac{1-y}{1-\hat{y}} \right) \hat{y}(1-\hat{y}) = -y(1-\hat{y}) + (1-y)\hat{y} = \hat{y} - y$$

です。

重みの勾配と隠れ層の誤差 $z^{(2)} = \sum_k W_k^{(2)} h_k + b^{(2)}$ なので $\partial z^{(2)} / \partial W_k^{(2)} = h_k$. 連鎖律より $\partial L / \partial W_k^{(2)} = \delta^{(2)} h_k$ です。これを横ベクトルに並べたものが $\delta^{(2)} \mathbf{h}^\top$ です。

次に, $z^{(2)}$ は h_k に $W_k^{(2)}$ の重みで依存するので $\partial L / \partial h_k = \delta^{(2)} W_k^{(2)}$. さらに $h_k = \sigma(z_k^{(1)})$ なので

$$\delta_k^{(1)} = \frac{\partial L}{\partial z_k^{(1)}} = \frac{\partial L}{\partial h_k} \cdot \frac{\partial h_k}{\partial z_k^{(1)}} = W_k^{(2)} \delta^{(2)} \sigma'(z_k^{(1)})$$

です。 k について並べると $(W^{(2)\top} \delta^{(2)}) \odot \sigma'(\mathbf{z}^{(1)})$ になります。最後に $z_k^{(1)} = \sum_j W_{kj}^{(1)} x_j + b_k^{(1)}$ なので $\partial L / \partial W_{kj}^{(1)} = \delta_k^{(1)} x_j$. これを k 行 j 列に並べた行列が $\boldsymbol{\delta}^{(1)} \mathbf{x}^\top$ (縦ベクトル $\times$ 横ベクトル) です。

A.3 一般の多層ネットワーク

L 層のネットワーク $\mathbf{z}^{(l)} = W^{(l)}\mathbf{h}^{(l-1)} + \mathbf{b}^{(l)}$, $\mathbf{h}^{(l)} = f(\mathbf{z}^{(l)})$ ($\mathbf{h}^{(0)} = \mathbf{x}$) でも, まったく同じ形になります。

$$\boldsymbol{\delta}^{(l)} = (W^{(l+1)\top} \boldsymbol{\delta}^{(l+1)}) \odot f'(\mathbf{z}^{(l)}), \quad \frac{\partial L}{\partial W^{(l)}} = \boldsymbol{\delta}^{(l)} \mathbf{h}^{(l-1)\top}, \quad \frac{\partial L}{\partial \mathbf{b}^{(l)}} = \boldsymbol{\delta}^{(l)}$$

順伝播では $W^{(l)}$ を掛けて入力から出力へ進み, 逆伝播では転置 $W^{(l)\top}$ を掛けて出力から入力へ誤差を送ります。 $W^{(l+1)\top} \boldsymbol{\delta}^{(l+1)}$ の第 k 成分は $\sum_m W_{mk}^{(l+1)} \delta_m^{(l+1)}$ で, 3.3節で述べた「各出力先から届く誤差 $\times$ 重みを足し合わせる」ことにあたります。

A.4 勾配消失を行列で書く

一般の式を繰り返し適用すると、入力に近い層の誤差は

$$\delta^{(1)} \sim W^{(2)\top} f'(\cdot) W^{(3)\top} f'(\cdot) \dots W^{(L)\top} f'(\cdot) \delta^{(L)}$$

のように、転置した重み行列と活性化関数の導関数を交互に掛け合わせた形になります（～は「およそこの形」の意味で、 $f'(\cdot)$ は成分ごとの積として掛かります）。6.1節の細長いネットワークは、すべての行列が1行1列の場合です。

付録B Python コード（標準ライブラリのみ）

本文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。

B.1 誤差逆伝播法による XOR の学習（第4節）

2-2-1 のネットワークで XOR を学習する（誤差逆伝播法）

```
import math, random
```

```
X = [(0, 0), (0, 1), (1, 0), (1, 1)]
```

```
Y = [0, 1, 1, 0]
```

```
def sigmoid(z):
```

```
    return 1.0 / (1.0 + math.exp(-z))
```

```
def train(seed, eta=1.0, epochs=5000, log=()):
```

```
    random.seed(seed)
```

```
    H = 2 # 隠れユニット数
```

```
    W1 = [[random.uniform(-1, 1) for _ in range(2)] for _ in range(H)]
```

```
    b1 = [0.0] * H
```

```
    W2 = [random.uniform(-1, 1) for _ in range(H)]
```

```
    b2 = 0.0
```

```
    history = []
```

```
    for ep in range(epochs + 1):
```

```
        # 勾配を 0 に初期化
```

```
        gw1 = [[0.0, 0.0] for _ in range(H)]; gb1 = [0.0] * H
```

```
        gw2 = [0.0] * H; gb2 = 0.0; L = 0.0
```

```
        for x, y in zip(X, Y):
```

```
            # --- 順伝播 ---
```

```
            z1 = [W1[k][0]*x[0] + W1[k][1]*x[1] + b1[k] for k in range(H)]
```

```
            h = [sigmoid(z) for z in z1]
```

```
            z2 = sum(W2[k]*h[k] for k in range(H)) + b2
```

```
            p = sigmoid(z2)
```

```
            L += -(y*math.log(p) + (1-y)*math.log(1-p))
```

```
            # --- 逆伝播 ---
```

```
            d2 = p - y #  $\delta(2) = \partial L / \partial z2$ 
```

```
            for k in range(H):
```

```
                gw2[k] += d2 * h[k] #  $\partial L / \partial W2\_k = \delta(2) h\_k$ 
```

```
                d1 = W2[k] * d2 * h[k] * (1 - h[k]) #  $\delta(1)\_k = W2\_k \delta(2) \sigma'(z1\_k)$ 
```

```
                gw1[k][0] += d1 * x[0] #  $\partial L / \partial W1\_kj = \delta(1)\_k x\_j$ 
```

```
                gw1[k][1] += d1 * x[1]
```

```
                gb1[k] += d1
```

```

        gb2 += d2
    history.append(L)
    if ep in log:
        print(f" epoch {ep:5d}: Loss = {L:.4f}")
    # --- 勾配降下 ---
    for k in range(H):
        W2[k] -= eta * gW2[k]
        b1[k] -= eta * gb1[k]
        for j in range(2):
            W1[k][j] -= eta * gW1[k][j]
    b2 -= eta * gb2
    return (W1, b1, W2, b2), history

def predict(params, x):
    W1, b1, W2, b2 = params
    h = [sigmoid(W1[k][0]*x[0] + W1[k][1]*x[1] + b1[k]) for k in range(len(b1))]
    return sigmoid(sum(W2[k]*h[k] for k in range(len(b1))) + b2), h

if __name__ == "__main__":
    for seed in (1, 9):
        print(f"seed = {seed}")
        params, hist = train(seed, log=(0, 100, 500, 1000, 2000, 5000))
        for x in X:
            p, h = predict(params, x)
            print(f" 入力 {x} → 出力 {p:.3f}   隠れ層 ({h[0]:.2f}, {h[1]:.2f})")

```

実行結果 (seed 1 が本文の初期値 A, seed 9 が初期値 B) :

```

seed = 1
epoch    0: Loss = 2.7741
epoch   100: Loss = 2.7721
epoch   500: Loss = 0.0746
epoch  1000: Loss = 0.0239
epoch  2000: Loss = 0.0100
epoch  5000: Loss = 0.0036
入力 (0, 0) → 出力 0.001   隠れ層 (0.99, 0.99)
入力 (0, 1) → 出力 0.999   隠れ層 (1.00, 0.01)
入力 (1, 0) → 出力 0.999   隠れ層 (0.01, 1.00)
入力 (1, 1) → 出力 0.001   隠れ層 (0.99, 0.99)
seed = 9
epoch    0: Loss = 2.8736
epoch   100: Loss = 2.1686
epoch   500: Loss = 1.4020
epoch  1000: Loss = 1.3928
epoch  2000: Loss = 1.3892
epoch  5000: Loss = 1.3874
入力 (0, 0) → 出力 0.000   隠れ層 (0.86, 0.05)
入力 (0, 1) → 出力 1.000   隠れ層 (0.02, 0.86)
入力 (1, 0) → 出力 0.500   隠れ層 (0.00, 0.00)
入力 (1, 1) → 出力 0.500   隠れ層 (0.00, 0.00)

```

B.2 勾配チェック (第3.4節)

逆伝播で求めた勾配を, 数値微分と比べて確かめる (勾配チェック)

```
import math, random
```

```

X = [(0, 0), (0, 1), (1, 0), (1, 1)]
Y = [0, 1, 1, 0]
sig = lambda z: 1.0 / (1.0 + math.exp(-z))

# パラメータを1本のリストにまとめる: W1(4個), b1(2個), W2(2個), b2(1個)
random.seed(1)
theta = [random.uniform(-1, 1) for _ in range(9)]

def unpack(t):
    return [[t[0], t[1]], [t[2], t[3]]], t[4:6], t[6:8], t[8]

def loss(t):
    W1, b1, W2, b2 = unpack(t)
    L = 0.0
    for x, y in zip(X, Y):
        h = [sig(W1[k][0]*x[0] + W1[k][1]*x[1] + b1[k]) for k in range(2)]
        p = sig(W2[0]*h[0] + W2[1]*h[1] + b2)
        L += -(y*math.log(p) + (1-y)*math.log(1-p))
    return L

def backprop(t):
    W1, b1, W2, b2 = unpack(t)
    g = [0.0] * 9
    for x, y in zip(X, Y):
        h = [sig(W1[k][0]*x[0] + W1[k][1]*x[1] + b1[k]) for k in range(2)]
        p = sig(W2[0]*h[0] + W2[1]*h[1] + b2)
        d2 = p - y
        for k in range(2):
            d1 = W2[k] * d2 * h[k] * (1 - h[k])
            g[2*k] += d1 * x[0]; g[2*k+1] += d1 * x[1]; g[4+k] += d1
            g[6+k] += d2 * h[k]
        g[8] += d2
    return g

def numerical(t, eps=1e-5):
    g = []
    for i in range(len(t)):
        tp = t[:]; tp[i] += eps
        tm = t[:]; tm[i] -= eps
        g.append((loss(tp) - loss(tm)) / (2 * eps)) # 中心差分
    return g

names = ["W1_11", "W1_12", "W1_21", "W1_22", "b1_1", "b1_2", "W2_1", "W2_2", "b2"]
for n, a, b in zip(names, backprop(theta), numerical(theta)):
    print(f"{n:6s} 逆伝播 {a:+.8f} 数値微分 {b:+.8f} 差 {abs(a-b):.1e}")

```

実行結果：

W1_11	逆伝播	-0.00844318	数値微分	-0.00844318	差	1.4e-12
W1_12	逆伝播	-0.00983020	数値微分	-0.00983020	差	2.1e-11
W1_21	逆伝播	-0.02303636	数値微分	-0.02303636	差	1.6e-11
W1_22	逆伝播	-0.02104372	数値微分	-0.02104372	差	9.9e-12
b1_1	逆伝播	-0.01830691	数値微分	-0.01830691	差	6.8e-12
b1_2	逆伝播	-0.04426312	数値微分	-0.04426312	差	1.9e-11
W2_1	逆伝播	-0.19033744	数値微分	-0.19033744	差	9.2e-12
W2_2	逆伝播	-0.18284084	数値微分	-0.18284084	差	1.4e-12
b2	逆伝播	-0.38068435	数値微分	-0.38068435	差	1.1e-11