知能プログラミング演習 I 講義ノート

オリジナル作成者: 梅津 佑太
2025 年度更新: 稲津 佑 inatsu.yu@nitech.ac.jp

最終更新: 2025 年 6 月 2 日

概要

これは, 知能系 3 年生向けの演習用の講義ノート (メモ) です. python による実装や, 教科書ではわかりづらい数理的な部分について, 講義に関しての (ほぼ) 必要十分な題材だけをまとめました. この講義は, python を使ったアルゴリズムの実装をメインとした講義です. そのため, 微分や行列演算などの説明では, 数理的に厳密な記述は避け, これらを道具として自由に使える程度にまとめます.

(お断り) このノートはあくまでも上記の演習を受講した学生さんのためのものです. そのため, 内容にやや偏りがあるでしょうし, 品質も保証されたものではありません. ところどころ, ミスやタイポもあると思いますので, その点を十分に了承した上でお使いいただくようお願いします. ただし, 誤りが見つかった際には, 修正しますので, 不明瞭な部分やミスをお見かけした際には上記メールアドレスにご連絡いただけますと幸いです.

目次

1 python 入門 3
1.1 準備 . . . 3
1.2 numpy . . . 4
1.2.1 np.array による配列の生成 . . . 4
1.2.2 特殊な行列の生成 . . . 5
1.2.3 配列の操作 . . . 6
1.3 for 文, if 文 . . . 12
1.4 関数の定義 . . . 14
1.5 作図の基本 . . . 16
1.5.1 matplotlib . . . 17
1.5.2 seaborn . . . 17
1.6 おまけ . . . 19
1.6.1 パッケージのインストール . . . 19
1.6.2 Anaconda と spyder . . . 19

2 数学的な補足 20
2.1 線形代数 . . . 20
2.1.1 ベクトル空間 . . . 21
2.1.2 ベクトルの内積とノルム . . . 23
2.1.3 行列といろいろな演算 . . . 25
2.2 微分 . . . 28
2.2.1 一変数関数の微分 . . . 28
2.2.2 一変数関数の合成関数とその微分 . . . 30

2.2.3 多変数関数の微分 32

3 深層学習入門 (工事中) 37

3.1 パーセプトロン 37
3.1.1 単純パーセプトロン 38
3.1.2 パーセプトロン 40
3.2 深層ニューラルネットワーク 44
3.2.1 3 層ニューラルネットワーク 45
3.2.2 深層ニューラルネットワーク 51
3.3 オートエンコーダ 55
3.3.1 次元圧縮と主成分分析, オートエンコーダ 55
3.3.2 オートエンコーダの学習 58
3.3.3 オートエンコーダに関連するニューラルネットワークモデル 60
3.4 再帰型ニューラルネットワーク 62
3.4.1 再帰型ニューラルネットワークの学習 63
3.4.2 長 · 短期記憶 67
3.5 畳み込みニューラルネットワーク 70
3.5.1 畳み込みニューラルネットワークの順伝播 71
3.5.2 畳み込みニューラルネットワークの逆伝播 76
3.6 おまけ: いろいろな最適化アルゴリズム 86

1 python 入門

python はフリーの汎用的なプログラミング言語です．機械学習や統計解析を実装する際に, 頻繁に使われるようになりました．最近は企業や研究所でも使う人が多くなりましたし, python に慣れておけば R や matlab, octabe などの他言語も (比較的簡単に) 活用可能です．ということで, 一度は勉強しておいても損はしないはず.

ちなみに, python はオブジェクト言語としても活用できるが, ここではそこまで踏み込まずに, 最低限講義に必要なものを中心に述べることにする．このような話題については, ウェブ上にいろいろな解説もあるし, 素晴らしい書籍も世に出回っているのでそちらを参考にしてほしい.

1.1 準備

python では, 対話機能を強化した ipython がよく用いられるため, この資料でも ipython を利用する．多くの python の統合開発環境 (spyder など) も内部的に ipython が実行されている．ipython が実行されると以下のようなプロンプトが出力されて待機状態になる（CSE ならターミナルから `ipython` と打つ．spyder ならばデフォルトでは画面右下のコンソール領域に以下の表示がでる）.

```
In [1]:
```

これはシェルのような対話型のインターフェースであり, コマンドが入力できる状態になっている．ここに, 例えば以下のように `print("Hello, World")` と入力してリターンキーを押すと, お馴染みのフレーズが出力できる.

```
In [1]: print("Hello, World")
Hello, World
```

python では事前のデータ型指定なしにいきなり変数に値を代入することができる.

```
In [1]: x = 0              # x に 0 を代入

In [2]: x                  # その後，x とだけ打ってリターンすると値が表示される
Out[2]: 0
```

さらに以下のように `print` 関数で変数を表示させることもできる.

```
In [3]: print(x)           # 変数だけ表示
0
In [4]: print("x =", x)    # 文字列と合わせて表示
x = 0
```

ここまでのように対話型で処理を進めることもできるが, ある程度以上のプログラムを作成する場合は, 処理をファイルにまとめて記載しておきそれを一括で実行することが多い．例えば，以下のような内容を `exmaple01.py` として保存しておくとする.

```
print("--- exmaple01.py ---")
x = 0
x
print("x = ", x)
```

このファイルの内容を実行する方法はいくつか存在する.

1. spyder などの統合環境を使っている場合は, 実行用の GUI やショートカットキーを用いる (そもそもファイルの作成自体, 通常開発環境内で行う).
2. ターミナル環境であれば, 以下のように python コマンドの引数にファイルを渡す

   ```
   $ python example01.py
   --- exmaple01.py ---
   ```

```
x =  0
```

3. ipython 内で %run コマンドを使う

```
In [1]: %run example01.py
--- exmaple01.py ---
x =  0
```

exmaple01.py では 3 行目で変数 x のみが記載されている．対話型モードでは, 変数だけを入力してリターンすると変数の内容が表示されたが, このファイルを実行した例ではこの行に相当する表示が行われてないことに注意されたい．ファイルからの一括実行で変数を表示させたい場合は明示的に print などの表示用関数を使う．

1.2 numpy

numpy は python に実装されている, 汎用的な数値計算ライブラリである．ベクトルや行列などの配列を扱う場合には (ほぼ) 確実に必要となるだろう．**numpy を使う場合には，まず import コマンドを実行する．**

```
In [1]: import numpy as np
```

これが実行された後は

- **np.(numpy の関数名)(引数)**

として, numpy の関数を呼び出すことができる．

1.2.1 np.array による配列の生成

配列とはベクトルや行列の一般化だと思えば良い．例えば, ベクトルは 1 次元の配列だし, 行列は 2 次元の配列とも言える（機械学習だと 2 次元の配列をデータとして扱うことが多く, 3 次元以上の配列をテンソルと呼んで区別したりもする）．numpy で配列を生成する関数の書式は

- np.array(object, dtype)

であり, object で数値列を指定し, ここで, dtype とはデータの型であり, int(整数) または float(浮動小数) を指定することができる．dtype を指定しなければ, object と同じ型で配列が生成される．例えば,

```
In [2]: A = np.array([[1,2], [3,4]], dtype = float)
```

とすれば, float 型の 2 次元配列 (行列)

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

が作られる．作られた変数の名前 A を打ってリターンすると値が確認できる．

```
In [3]: A
Out[3]:
array([[1., 2.],
       [3., 4.]])
```

上の数式との対応を確認されたい．また,

```
In [4]: b = np.array([1,2,3], dtype = float)
```

とすれば, float 型の 1 次元配列 (ベクトル)

$$b = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

が作られる．ただし, データの型を指定しない場合, これらは int 型となるので注意が必要である[*1]．データの型を調べたければ,

- **(配列)**.dtype

[*1] dtype のデフォルトは関数によってバラバラなので, 使いながら慣れていってほしい．

とすれば良い. 例えば,

```
In [5]: A.dtype
Out[5]: dtype('float64')

In [6]: b.dtype
Out[6]: dtype('float64')
```

また, 配列のサイズを調べるときは

- **(配列)**.shape

とする. 例えば,

```
In [7]: A.shape
Out[7]: (2, 2)

In [8]: b.shape
Out[8]: (3,)
```

連続する整数でベクトルを作りたい場合, 以下のような書き方がよく使われる

```
In [9]: np.array(range(10))
Out[9]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

ここで, range は連番の整数値を作ることができ, 引数を一つとると上のように 0 から受け取った値の一つ前まで, 引数を二つ受け取ると以下のように

```
In [10]: np.array(range(5,10))
Out[10]: array([5, 6, 7, 8, 9])
```

一つ目の引数の値から二つ目の引数の値の一つ前までの整数値列を作成できる. range 自体は numpy とは独立して python に元からある関数で, 後に出てくる for 文の記述にもよく用いられる.

1.2.2 特殊な行列の生成

よく使う特殊な行列やベクトルの生成用関数として例えば以下のようなものがある:

- np.identity(n, dtype) # 単位行列
- np.zeros(shape, dtype) # 全ての要素がゼロの行列やベクトル
- np.ones(shape, dtype) # 全ての要素が 1 の行列やベクトル

で生成できる. ここで, n や shape は配列の大きさを指定する引数である.

```
In [1]: np.identity(4)
Out[1]:
array([[1., 0., 0., 0.],
       [0., 1., 0., 0.],
       [0., 0., 1., 0.],
       [0., 0., 0., 1.]])

In [2]: np.zeros(2)
Out[2]: array([0., 0.])

In [3]: np.ones((2,3))
Out[3]:
array([[1., 1., 1.],
       [1., 1., 1.]])
```

ちなみに, identity では 2 次元配列しか作れない[*2]が, zeros では多次元配列を作ることができる. 例えば,

[*2] $n \times n$ の単位行列の生成のみに使用出来る.

表 1　numpy による乱数生成

関数	機能
`np.random.seed(seed)`	乱数のシードの固定
`np.random.permutation(x)`	x をランダムに並べ替える
`np.random.normal(loc, scale, size)`	正規乱数の発生
`np.random.uniform(low, high, size)`	一様乱数の発生

```
In [9]: np.zeros((2, 3, 3), dtype = int)
Out[9]:
array([[[0, 0, 0],
        [0, 0, 0],
        [0, 0, 0]],

       [[0, 0, 0],
        [0, 0, 0],
        [0, 0, 0]]])
```

とすれば, 3×3 次元のゼロ行列が 2 個並んだような int 型の配列が得られる.

最後に, 乱数の生成について述べておこう. numpy ではいろいろな確率分布からの乱数を

- `np.random.(関数名)(引数)`

で生成できる. 例えば, 正規乱数を発生させたければ

- `np.random.normal(loc, scale, size)`

とすれば良い. ここで, `loc` と `scale` はそれぞれ正規分布の平均と標準偏差を表す引数であり, `size` は配列のサイズを指定する引数で, `zeros` と同様に多次元配列を定義することができる. 平均 0, 標準偏差 1 の 2×3 行列を作るには以下のようにする.

```
In [10]: np.random.normal(0, 1, (2,3))
Out[10]:
array([[-1.60311173,  0.81807137,  1.77353401],
       [ 0.26225532, -0.1919794 ,  0.60838859]])
```

なお, このようにして作られる配列は確率的に変動するものなので, 実行するたびに出力が変わってしまう. そのため, 乱数を発生させるようなアルゴリズムを設計する場合には,

- `np.random.seed(seed)`

で乱数を固定して出力結果が変わらないようにすることができる. 表 1 に `random` で生成できる乱数をまとめた. これらのうちのいくつかは深層学習におけるパラメータの初期値を設定する際や, データのインデックスを並べ替える際に利用できる.

1.2.3　配列の操作

配列を生成できれば, 次は配列間の和などの四則演算や, 配列の関数を実行できる. python では四則演算として, 足し算 “+”, 引き算 “−”, 掛け算 “∗”, 割り算 “/” ができ, これらの演算は配列であっても同様に実行可能である[*3]. いくつか例を挙げておく.

```
In  [1]:  2 + 3
Out [1]:  5    # int 型

In  [2]:  2.0 + 3.0
Out [2]:  5.0    # float 型
```

[*3] 当たり前だが, 四則演算を行うためには “適切なサイズの” 配列が要求される.

```
In  [3]:  5.0 - 2
Out [3]:  3.0    # float 型を含む演算では float 型が出力される

In  [4]:  2 * 3
Out [4]:  6

In  [5]:  4/2
Out [5]:  2.0    # 割り算の出力は float 型
```

また, 配列の演算は基本的に要素ごとに作用する.

```
In  [1]:  a = np.array([1, 2], dtype = float)

In  [2]:  b = np.array([3, 1], dtype = float)
```

として二つの配列を定義すると,

```
In  [3]:  a + b
Out [3]:  array([4., 3.])
```

が出力されるし, 定数倍なども成分ごとに計算される.

```
In  [4]:  2 * a
Out [4]:  array([2., 4.])
```

配列の要素へのアクセスは

```
In  [5]:  a[0]
Out [5]:  1.0

In  [6]:  a[0:1]
Out [6]:  array([1.])

In  [7]:  a[0:2]
Out [7]:  array([1., 2.])
```

などとすれば良い. ただし, python のインデックスは 0 から始まることに注意する. また, 0:1 は 0, 0:2 は 0, 1 などと, コロンの右側の数字よりもひとつ小さな整数までのインデックスを表している.

多次元の配列でも類似した方法でアクセスができる. 以下の例では, 単位行列を作成して, 1 行 1 列要素と 1 行 2 列要素を表示している（ここでも python のインデックスは 0 からであることに注意）.

```
In [8]: A = np.identity(3)

In [9]: A
Out[9]:
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])

In [10]: A[0,0]
Out[10]: 1.0

In [11]: A[0,1]
Out[11]: 0.0
```

また, 以下のようにすると例えば, 3 行目の 1 列目から 2 列目という指定もできる.

```
In [12]: A[2,0:2]
Out[12]: array([0., 0.])
```

“:” とすると, 全ての要素を指定でき, 以下の例では 2 列目の全ての要素が参照できる.

```
In [13]: A[:,1]
Out[13]: array([0., 1., 0.])
```

補足 1. 実は python には broadcast と呼ばれる機能があり, 通常計算できない演算を行うことができる. 例えば, 2×2 行列 A と 2 次元ベクトル b を次のように定義する.

```
In  [1]:  A = np.array([[1, 2], [3, 4]], dtype = float)

In  [2]:  b = np.array([1,2], dtype = float)
```

当然, 数学的には A と b の間に, (少なくとも和や差などの) 演算は定義できないわけであるが,

```
In  [3]:  A + b
Out [3]:  array([[2., 4.],
                 [4., 6.]])

In  [4]:  A * b
Out [4]:  array([[1., 4.],
                 [3., 8.]])
```

のように, 頑張って計算してくれる. とはいえ, やはり計算できない演算もあり,

```
In  [5]:  c = np.array([1, 2, 3], dtype = float)
```

を定義すると,

```
In  [6]:  A + c
ValueError:  operands could not be broadcast together with shapes (2,2) (3,)
```

のように, “配列の大きさが違うので演算できません”という意味のエラーが返ってくる. **よくわからない演算をやってしまうと, プログラムのバグにもつながるので, 慣れないうちは broadcast 機能は用いずに, ちゃんとした数学的に正しい演算を行うよう心がけてほしい**[a]**.**

[a] かっこいいプログラムを組もうとしてバグが発生するというのは世の常.

アルゴリズムを実装する際には, 上記の演算のほか, 内積 (**inner product**) や直積 (**outer product**)[*4] などの演算も簡単に行えると良い. 具体的には, 次のように内積などの計算を行える.

```
In  [1]:  a = np.array([1, 2], dtype = float)

In  [2]:  b = np.array([3, 1], dtype = float)

In  [3]:  np.dot(a, b)    # a^T b (= b^T a)
Out [3]:  5.0

In  [4]:  np.outer(a, b)    # ab^T 縦ベクトルと横ベクトルの積が結果, 行列となることに注意
Out [4]:  array([[3., 1.],
                 [6., 2.]])

In  [5]:  np.outer(b, a)    # ba^T
Out [5]:  array([[3., 6.],
                 [1., 2.]])
```

などとなる. また, 行列とベクトルや, 行列同士の内積も同様に

```
In  [1]:  A = np.array([[1, 2], [3, 4]], dtype = float)

In  [2]:  x = np.array([1,2], dtype = float)

In  [3]:  np.dot(A, x)    # Ax
Out [3]:  array([5., 11.])

In  [6]:  B = np.array([[3, 2, 1], [0, 1, 0]], dtype = float)

In  [7]:  np.dot(A, B)    # AB
Out [7]:  array([[3.,  4., 1.],
```

[*4] 英語で outer product なので “外積”と呼ぶ場合もあるが, 通常外積はベクトル積 (クロス積) $a \times b$ を意味するので要注意!

```
                [9., 10., 3.]])
```

となる. ただし, 上の例で積 BA を計算しようとするとエラーが返ってくる.

```
In  [8]:  np.dot(B, A)    # BA
ValueError:  shapes (2,3) and (2,2) not aligned:  3 (dim 1) != 2 (dim 0)
```

これは, A が 2×2 行列で B が 2×3 行列の場合, 積 BA が定義されないためである（このことの意味がわからない場合は線形代数のテキストで行列の積の定義を復習すること）. 関連して, 以下で `np.dot(Z,c)` はエラーにならないのに, `np.dot(Z,d)` がエラーになる理由を考えてほしい.

```
In [9]: Z = np.identity(3)                  # サイズ３の単位行列

In [10]: c = np.array([1,2,3])              # 1 次元配列

In [11]: d = np.array([[1,2,3]])            # 1x3 の 2 次元配列

In [12]: np.dot(Z,c)
Out[12]: array([1., 2., 3.])

In [13]: np.dot(Z,d)
ValueError: shapes (3,3) and (1,3) not aligned: 3 (dim 1) != 1 (dim 0)
```

補足 2. 一方, 通常の直積 (Kronecker 積) は二つの配列, 例えば $m \times n$ 行列 A と $p \times q$ 行列 B, が与えられたとき, $mp \times nq$ 行列

$$A \otimes B = (a_{ij}B)_{i,j}$$

を返す演算の事であるが, numpy ではやや仕様が異なる. 例えば,

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}, \qquad B = \begin{bmatrix} 3 & 2 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

としたとき, 通常の直積の定義によれば 4×6 行列

$$A \otimes B = \left[\begin{array}{ccc|ccc} 3 & 2 & 1 & 6 & 4 & 2 \\ 0 & 1 & 0 & 0 & 2 & 0 \\ \hline 9 & 6 & 3 & 12 & 8 & 4 \\ 0 & 3 & 0 & 0 & 4 & 0 \end{array}\right]$$

が出力されるはずだが, `np.outer(A, B)` を実行すると,

$$A, B \mapsto \left[\begin{array}{cccccc} 3 & 2 & 1 & 0 & 1 & 0 \\ \hline 6 & 4 & 2 & 0 & 2 & 0 \\ \hline 9 & 6 & 3 & 0 & 3 & 0 \\ \hline 12 & 8 & 4 & 0 & 4 & 0 \end{array}\right]$$

のように, $a_{ij}B$ をベクトル化したものを並べたものが出力される. **ところで, 通常の直積はふたつのベクトル間の演算を表すので, 2 次元以上の配列に対して `np.outer` を用いる際には注意が必要した方が良い**[a].

[a] というか, ちゃんと Kronecker 積を計算する関数 `np.kron` があるので, 必要ならそちらを使いましょう.

配列の和や平均, 最大値, 最小値などを求める関数も numpy には実装されている. 割と便利なのが, この手の関数は配列のどの軸を基準にして演算を行うかを指定できるところにある. 例えば, 上で定義した 2×2 の行列 A に対して,

```
In  [1]:  np.sum(A)
```

```
Out [1]:  10.0

In  [2]:  np.sum(A, axis = 0)
Out [2]:  array([4., 6.])

In  [3]:  np.sum(A, axis = 1)
Out [3]:  array([3., 7.])
```

などとなる. つまり,

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

を考えたときに, axis を指定しなければ, 全要素で和を計算する, つまり,

$$A \mapsto 1 + 2 + 3 + 4 = 10$$

という計算結果を出力する. また, axis = 0 や 1 と指定した場合にはそれぞれ,

$$\texttt{axis} = 0 \Rightarrow A \mapsto [1 + 3, 2 + 4] = [4, 6]$$
$$\texttt{axis} = 1 \Rightarrow A \mapsto [1 + 2, 3 + 4] = [3, 7]$$

のように, 列あるいは行ごとに関数を計算する.

numpy を利用することで, 配列の結合やサイズの変更もできる. 配列の結合は

- np.append(arr, values, axis)

で行う. 適切なサイズの配列になっていないとエラーが返ってくるわけだが, 2 次元以上の配列の場合は, axis で結合する方向を指定できる. 指定しない場合は, 1 次元配列を束ねたものとして出力される.

```
In  [1]:  np.append(1, 2)
Out [1]:  array([1, 2])

In  [2]:  A = np.array([[1, 2], [3, 4]], dtype = float)

In  [3]:  B = np.array([[3, 2, 1], [0, 1, 0]], dtype = float)

In  [4]:  np.append(A, A)
Out [4]:  array([1., 2., 3., 4., 1., 2., 3., 4.])

In  [5]:  np.append(A, A, axis = 0)
Out [5]:  array([[1., 2.],
                 [3., 4.],
                 [1., 2.],
                 [3., 4.]])

In  [6]:  np.append(A, A, axis = 1)
Out [6]:  array([[1., 2., 1., 2.],
                 [3., 4., 3., 4.]])

In  [7]:  np.append(A, B)
Out [7]:  array([1., 2., 3., 4., 3., 2., 1., 0., 1., 0.])

In  [8]:  np.append(A, B, axis = 0)
ValueError:  all the input array dimensions except for the concatenation axis must
    match exactly

In  [9]:  np.append(A, B, axis = 1)
Out [9]:  array([[1., 2., 3., 2., 1.],
                 [3., 4., 0., 1., 0.]])
```

サイズの変更は

- np.reshape(a, newshape)

表 2　代表的な numpy の関数

関数名	機能
`np.append(arr, values)`	二つの配列を束ねて一つの配列にする
`np.reshape(a, newshape)`	配列のサイズを変更する
`np.dot(a, b)`	二つの配列の内積, i.e., $\boldsymbol{a}^\top \boldsymbol{b}$
`np.outer(a, b)`	二つの配列の直積, i.e., $\boldsymbol{a}\boldsymbol{b}^\top$
`np.sum(a, axis)`	配列の和
`np.mean(a, axis)`	配列の平均
`np.max(a, axis)`	配列の最大値
`np.min(a, axis)`	配列の最小値
`np.exp(x)`	配列の要素ごとの指数関数
`np.log(x)`	配列の要素ごとの対数関数 (底は $e = \exp(1)$)
`np.sin(x)`	配列の要素ごとの sin 関数
`np.cos(x)`	配列の要素ごとの cos 関数

で実行できる. `newshape` で変換後の配列のサイズを指定する.

```
In  [1]: A = np.array(range(1, 17), dtype = float) # 1 から 16 の 1 次元配列をまず作成

In  [2]: A
Out [2]: array([ 1., 2., 3., 4., 5., 6., 7., 8., 9., 10., 11., 12., 13., 14., 15.,
    16.])

In  [3]: B = np.reshape(A, newshape = (4, 4)) # 4 × 4 に A のサイズを変更して B に保存

In  [4]: B
Out [4]: array([[ 1.,  2.,  3.,  4.],
                [ 5.,  6.,  7.,  8.],
                [ 9., 10., 11., 12.],
                [13., 14., 15., 16.]])

In  [5]: np.reshape(B, newshape = (2, 8))
Out [5]: array([[ 1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.],
                [ 9., 10., 11., 12., 13., 14., 15., 16.]])

In  [6]: np.reshape(B, newshape = (8, 2))
Out [6]: array([[ 1.,  2.],
                [ 3.,  4.],
                [ 5.,  6.],
                [ 7.,  8.],
                [ 9., 10.],
                [11., 12.],
                [13., 14.],
                [15., 16.]])
```

また, 特に`-1` とすれば, 行ごとに配列を束ねた 1 次元の配列が得られる (上の続きで `np.reshape(B, newshape = -1)` としてみるとよい).

(よく使うであろう) 代表的な numpy の関数を表 2 にまとめた. もちろん, これ以外の関数も numpy には実装されているので, 必要であれば自分で調べる力も身につけてほしい.

1.3 for 文, if 文

python では, for 文や if 文などは **tab を利用して**入れ子を作る[*5]. for 文は基本的に

- `for` (変数) `in` (変数の動く範囲):
  実行する処理　　# この行の先頭に **tab**.

とすれば良い. 変数の後の `in` と変数の動く範囲 (オブジェクト) のコロン : を忘れないよう注意されたい. また, spyder など開発環境の editor ではユーザーが打たなくても for の次の行では勝手に tab が入る場合もある. 例えば, `country = ['Japan', 'America', 'England', 'Mexico']` として文字列を定義したとしよう[*6]. これらを for 文で順番に出力させると

```
In [1]: country = ['Japan', 'America', 'England', 'Mexico']

In [2]: for i in country:
   ...:     print(i)
   ...:                          # 2回リターンすると実行される
Japan
America
England
Mexico
```

もう少し実用的な例として,

$$1 + 2 + \cdots + 10$$

を for 文で実行することを考えよう. もちろん, for 文を使わなくても,

```
In  [1]:  a = np.array(range(1, 11), dtype = float)

In  [2]:  np.sum(a)
Out [2]:  55
```

とすれば簡単に実行できるわけだが, `np.sum` を for 文で再現するための練習だと思ってガマンしてほしい. やること自体は簡単で, a の要素を順番に足せばいい.

```
In  [1]:  x = 0

In  [2]:  for n in range(1, 11):
    ...:      x = x + n
    ...:      print(x)
    ...:                              # 同じく2回リターンすると実行
```

ここで, x は和の結果を保存しておくための “入れ物” である. イメージとしては, 数列 $\{x_n\}$ を,

$$x_0 = 0, \qquad x_{n+1} = x_n + n \quad (n \geq 1)$$

で定義したようなものだ. 2 重 for 文など, for 文を重ねる際には, for 文の中で for 文を定義すれば良い. つまり,

- `for` (変数 1) `in` (変数 1 の動く範囲):
  `for` (変数 2) `in` (変数 2 の動く範囲):
  実行する処理

のように, 2 回 tab を使って実装する. 例えば,

```
In  [1]:  for i in range(1, 11):
    ...:      for j in range(1, 11):
```

[*5] while 文などもこれに準じる.

[*6] クォーテーション '' で囲むと, オブジェクトは文字列として扱われる. また, ここでは詳細に立ち入らないが, “[,]” で囲まれたオブジェクトは python でリストと呼ばれるデータ構造にななる.

表 3　2 重 for 文で出力される順番のイメージ．赤字がアルゴリズムが結果を出力する順番を表している．i を固定するごとに，j を動かしたときの結果が順に出力される．

i \ j	**1**	**2**	**3**	**4**	**5**
1	1	2	3	4	5
2	6	7	8	9	10
3	11	12	13	14	15
4	16	17	18	19	20
5	21	22	23	24	25

```
...:             print(i/j)          # 内側の for の処理は tab でさらにもう 1 段階下げる
...:                                 # spyder などの場合は自動で下げてくれる
```

とすれば，自然数 $i, j = 1, \ldots, 10$ に対して，その比 i/j が出力される．出力される順番は，まず $i = 1$ に対して，$j = 1, \ldots, 10$ としたときの結果が出力され，次に $i = 2$ としたときの結果が出力される．これを $i = 10$ になるまで繰り返す．気分としては表 3 の通り．

ところで，for 文をたくさん重ねると，実行時間が爆発的に増える．例えば，変数 $X_1, \ldots, X_k$ でそれぞれ n 回の反復を繰り返すとすると，全体で n^k の繰り返しが必要となり，for 文の重複数に対して指数的に探索回数が増えてしまう．そのため，ベクトルや配列の演算だけで計算できるのであれば，多重 for 文を使わずに実装できるようになっておくと良い．

if 文も for 文と同様に

- `if` (条件式):
  　条件式が真のときに実行する処理

のように，やはり tab を利用して入れ子を作る．条件式が複数ある場合には，

- `if` (条件式):
  　条件式が真のときに実行する処理

  `else`:
  　条件式が偽のときに実行する処理

や，

- `if` (条件式 1):
  　条件式 1 が真のときに実行する処理

  `elif` (条件式 2):
  　条件式 1 が偽かつ条件式 2 が真のときに実行する処理

  `else`:
  　条件式 1 が偽かつ条件式 2 が偽のときに実行する処理

とする．条件分岐の際の `elif` や `else` には for 文のときのように tab で入れ子を作らないことに注意する．例えば，変数 x に対して，x の符号を返す処理を行いたければ

```
In  [1]:  x = 10

In  [2]:  if x > 0:
...:          print(1)
...:      else:              # ここは tab を一段戻す必要があることに注意
...:          print(-1)
...:
1
```

とする．これは，引数 `x` が正なら 1，それ以外 (ゼロ以下) で -1 を帰す処理

$$x \mapsto \begin{cases} 1, & x > 0 \\ -1, & x \leq 0 \end{cases}$$

を行うことに相当する．ところが, if 文では `if` の部分の条件に配列を指定することはできない．つまり, 配列に対して同じ処理を実行すると,

```
In  [1]:  x = np.array([-1, 0, 1])

In  [2]:  if x > 0:
    ...:      print(1)
    ...:  else:
    ...:      print(-1)
    ...:
The truth value of an array with more than one element is ambiguous.  Use a.any() or
    a.all()
```

となって python からお叱りを受けてしまう．そこで, 少しアタマをひねって, for 文と組み合わせることで,

```
In  [3]:  for a in x:     # np.array の x を in の後に書くと x の各要素が a に一つずつ順番に入る
    ...:      if a > 0:
    ...:          print(1)
    ...:      else:
    ...:          print(-1)
    ...:
-1
-1
1
```

として, 配列にも適用できるよう修正する．

ところで, python では論理値 (`True` や `False`) に対する演算も実装されており, 実は

```
In  [1]:  2*(x > 0)-1
```

とすれば, 同じ処理を実行できる．イメージとしては,

```
In  [1]:  x > 0
Out [1]:  array([False, False, True])
```

によって返される論理値の配列が `False = 0`, `True = 1` であるものと思って配列に対する演算を行っていると思えば良い．for 文と同様に, if 文も条件分岐が複雑になると, 処理時間の問題が生じることもある[*7]．また, 可読性の低いコードにもなってしまうので, 可能であれば if 文もサボる工夫ができると良い．さらに, 可読性に関連して, アルゴリズム中に毎回 for 文や if 文を利用して処理を行うのもわずらわしいので, 頻繁に利用するものに関しては, 次節で述べるように関数として定義しておくと良い．

1.4 関数の定義

前節で述べた for 文や if 文などと同様に, 関数を定義する際にも tab を利用して入れ子を作る．最も簡単な関数の作り方は,

- `def` 関数名 (引数):
  関数の本体
  `return` 関数の出力

とすることである．例えば, 関数 $f(x) = x$ や $g(x) = e^x$ を定義したければ

```
In  [1]:  def f(x):
              return x

In  [2]:  def g(x):
              return np.exp(x)
```

などとすれば良い．

[*7] for 文の各繰り返し中で, 常に if 文で条件チェックする場合など．

for 文や if 文なども, 関数の定義中に用いることができる. 例えば, フィボナッチ数列[*8]

$$a_0 = 1, \quad a_1 = 1, \quad a_{n+2} = a_{n+1} + a_n \qquad (n \geq 0)$$

の第 n 項を出力する関数は

```
In  [1]:  def Fib(n):
    ...:      if n == 0 or n == 1:
    ...:          return(1)
    ...:      else:
    ...:          y = 0
    ...:          x = 1
    ...:          for i in range(n):
    ...:              x, y = x + y, x
    ...:          return(x)
    ...:
```

などとすれば良い[*9]. 初めの 5 項 ($n = 0, 1, 2, 3, 4$) を出力すると

```
In  [1]:  Fib(0)
Out [1]:  1

In  [2]:  Fib(1)
Out [2]:  1

In  [3]:  Fib(2)
Out [3]:  2

In  [4]:  Fib(3)
Out [4]:  3

In  [5]:  Fib(4)
Out [5]:  5
```

となる.

関数の返り値は複数にすることもできる. これは return の後に, 単純にカンマでつなぐだけで実現できる.

```
In [1]: def f(x):
   ...:     return x, x**2
   ...:
```

これ実行すると, 以下のように表示される.

```
In [2]: f(10)
Out[2]: (10, 100)
```

返り値を保存する場合は, 別々の変数に代入することも, 一つの変数にまとめることもできる.

```
In [3]: a, b = f(10)

In [4]: a            # a と b にそれぞれ 10 と 100 が保存される
Out[4]: 10

In [5]: b
Out[5]: 100
```

[*8] Leonardo Fibonacci (1170 年頃 - 1250 年頃). ちなみに, フィボナッチは “ボナッチの息子”という意味で, 本名は Leonardo Pisano. “ピサのレオナルド”という意味らしいがどうにも大仰な名前をつけられてしまったものである.

[*9] 動的計画法を用いることでもっと効率良くフィボナッチ数列を計算させることもできるが, ここでの主題ではないので省略する.

```
In [6]: c = f(10)    # c に 10 と 100 が両方保存され，‘[]’で添字を指定

In [7]: c
Out[7]: (10, 100)

In [8]: c[0]
Out[8]: 10

In [9]: c[1]
Out[9]: 100
```

それぞれの返り値は np.array でもよい.

```
In [10]: def g(x):
    ...:     return np.zeros(3), x*np.ones(3)
    ...:

In [11]: g(2)
Out[11]: (array([0., 0., 0.]), array([2., 2., 2.]))
```

関数の引数には様々な型を渡すことができるが, 少し特殊な方法として, 「関数」を関数の引数に渡すこともできる. 例えば, 以下のように三つの関数を定義します.

```
In [12]: def funcA(x):
    ...:     print("call funcA")
    ...:     print("x = ", x)
    ...:
    ...: def funcB(x):
    ...:     print("call funcB")
    ...:     print("exp(x) = ", np.exp(x))
    ...:
    ...: def funcC(func):
    ...:     func(10)
    ...:
```

最後の関数 funcC は, funcA もしくは funcB を引数として受け取ることを想定しています. ‘func(10)’ という処理では, 引数として渡された funcA か funcB のどちらかが引数 10 を伴って実行されます. funcC を実行するには, 以下のようにします.

```
In [13]: funcC(funcA)
call funcA
x =  10

In [14]: funcC(funcB)
call funcB
exp(x) =  22026.465794806718
```

このようにすることで, ユーザーが「関数」を指定して処理内容を操作するような機能が実現できます.

1.5 作図の基本

関数の増減や解析結果を視覚的に見るために, グラフとして表示することは重要なものであろう. ここでは, matplotlib と seaborn を用いた作図について述べる.

1.5.1 matplotlib

matplotlib は

- `flom matplotlib import pyplot as plt`

または

- `import matplotlib.pyplot as plt`

として, numpy と同様にエディタの初めに宣言する. pyplot では, 散布図や棒グラフを始め, ヒストグラムや関数のグラフなどいろいろな図を作成することができる. 作図の基本は

- `plt.plot(x 軸, y 軸)`
  `plt.show()`

とすることである. `plt.plot` と `plt.show` の間に `plt.legend` や `plt.xlabel`, `plt.ylabel` を挟むことで, 凡例や x 軸, y 軸が何を表しているかを表示することもできる. また, `plt.grid` を挟むと, グリッド線も表示できる. なお, 作成したグラフを pdf として保存するには, `plt.show` の前に,

- `plt.savefig("保存先のパス/図の名前.pdf")`

を指定しておく.

関数[*10] $f(x) = \exp(-x^2/2)/\sqrt{2\pi}$ を区間 $[-3, 3]$ でプロットしたければ,

```
def f(x):     # 関数の定義
    y = np.exp( - x**2 / 2 ) / np.sqrt( 2 * np.pi )
    return(y)
x = np.linspace(start = -3, stop = 3, num = 100)     # 区間 [−3, 3] を 100 分割
y = f(x)     # 関数の出力
plt.plot(x, y, label = 'Gaussian')     # 関数のプロット
plt.legend()     # 凡例 (Gaussian) の表示
plt.show()     # 図の出力
```

を実行すると良い. 曲線の太さや色, 凡例のフォントサイズを変更したい場合は, `lw`, `color`, `fontsize` を指定しておく. さらに, `plt.plot` を繰り返すことで, グラフを重ねることもできる. 具体的には,

```
x = np.linspace(start = 0, stop = 2*np.pi, num = 100)
y_sin = np.sin(x)
y_cos = 2*np.cos(x)
plt.plot(x, y_sin, label = 'sin', lw = 5)
plt.plot(x, y_cos, label = '2cos', lw = 5)
plt.grid()
plt.legend(fontsize = 12)
plt.show()
```

を実行すると, 正弦関数 $g(x) = \sin x$ と余弦関数 $h(x) = 2\cos x$ を一つの図として出力できる. 図 1 は上で定義した関数 $f(x)$ の出力結果 (a) と, $g(x) = \sin x$, $h(x) = 2\cos x$ の出力結果 (b) を表したものである.

1.5.2 seaborn

pyplot と同様に seaborn も python で実装されている作図デバイスの一つであり, やはり

- `import seaborn as sns`

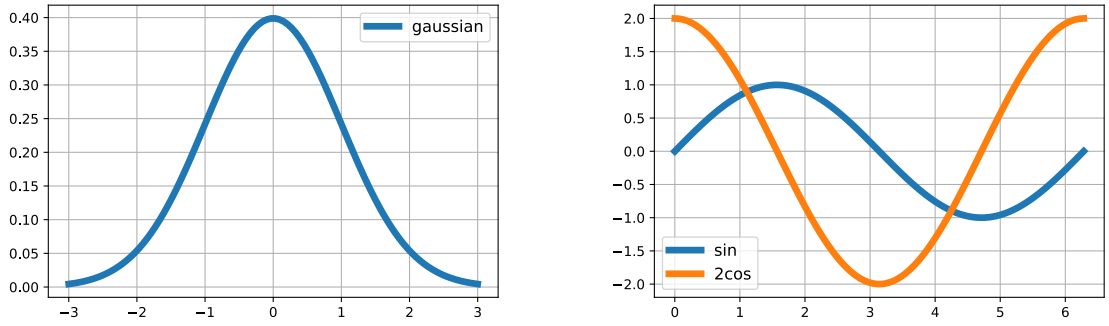
のようにエディタの初めに宣言する. matplotlib による関数を呼び出す前に, 関数 `set` を用いて

- `sns.set()`

としておくことで, matplotlib の出力がよりオシャレなものになる (こともある)[*11]. ちなみに, 図 1 を seaborn を使って出力したものが図 2 である.

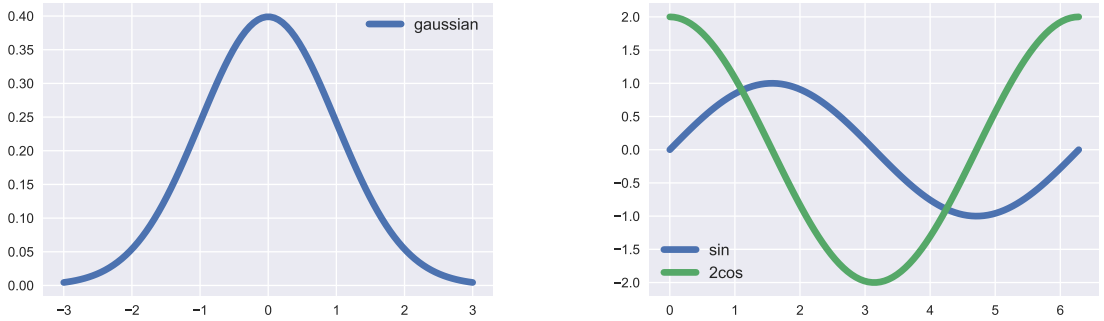
[*10] この関数は標準正規 (ガウス) 分布の確率密度関数として, 良く知られているものである.

[*11] オシャレかどうかは個人の判断にお任せする. 実際に使ってみてカッコいいと思えば使えばいいし, そうでなければ別に使う必要はない. 僕個人の感想としては, seaborn を使おうが使うまいが, 図そのものがわかりやすければそれで構わないと思う.



(a) $f(x) = \exp(-x^2/2)/\sqrt{2\pi}$ (b) $g(x) = \sin x, h(x) = 2\cos(x)$

図 1　出力した関数の図．左は $f(x)$, 右は $g(x) = \sin x$ と $h(x) = 2\cos x$ を区間 $[0, 2\pi]$ でプロットしたもの．



(a) $f(x) = \exp(-x^2/2)/\sqrt{2\pi}$ (b) $g(x) = \sin x, h(x) = 2\cos(x)$

図 2　seaborn を利用して出力した関数の図．左は $f(x)$, 右は $g(x) = \sin x$ と $h(x) = 2\cos x$ を区間 $[0, 2\pi]$ でプロットしたもの．

話を戻して，もう少し実用的な使い方について説明する．seaborn の便利な機能の一つとして，ヒートマップ[*12]に関するものがある．具体的に言うと，データ解析では confusion matrix[*13]を用いて，分類結果を可視化するということがよく行われる．confusion matrix については，3 章で改めて説明するので，ここではそういったものを作図するのに seaborn は便利くらいに思ってもらえば良い．seaborn を用いたヒートマップの作成は

- `sns.heatmap(data)`

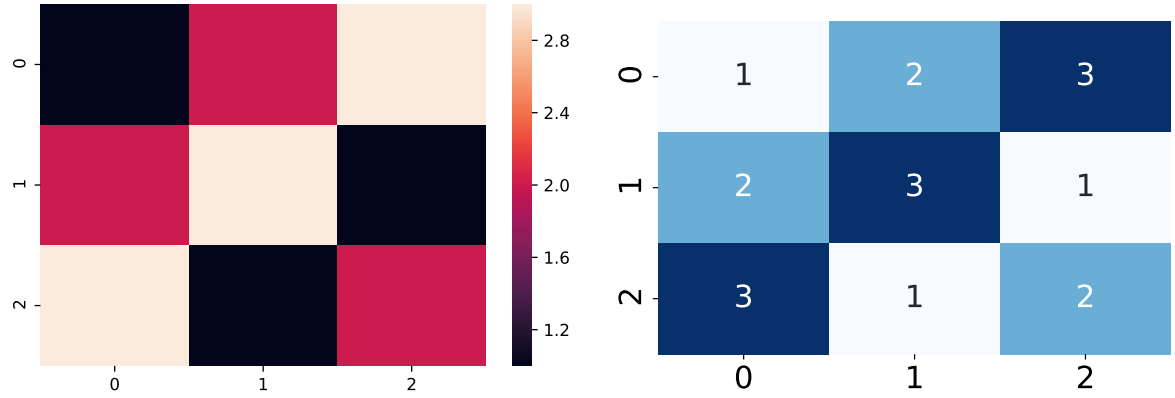
によって行うことができる．行列

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 3 & 1 \\ 3 & 1 & 2 \end{bmatrix}$$

に対して，ヒートマップを作成しよう．

```
In  [1]:  A = np.array([[1, 2, 3], [2, 3, 1], [3, 1, 2]], dtype = int)
```

[*12] 行列を色として可視化したもの．
[*13] 日本語では混同行列と呼ばれる．単語からだけだと何を言っているのかわかりにくい．



(a) デフォルトで出力されるヒートマップ　　(b) 変数を変えた場合の出力結果

図 3　seaborn によるヒートマップの作図

`In [2]: sns.heatmap(A)`
とすれば, 図 3(a) のような結果が表示される.
図の右側にあるカラーバーやヒートマップの色そのものを変えたければ, 引数に

- `cbar = False`
- `cmap ="好きな色"`

を指定する. また, ヒートマップで配列の値を表示させたければ,

- `annot = True`

とする. 同じ行列 A に対して, 以下のコードを実行すると, 図 3(b) のようになる.

```
In [3]: plt.rcParams['font.size'] = 20 # フォントサイズを変更するためのおまじない
In [4]: sns.heatmap(A, annot = True, cbar =False, cmap="Blues")
```

1.6　おまけ

1.6.1　パッケージのインストール

python に望んだパッケージがインストールされているか否かは

- `import` (パッケージ名)

を実行すれば確認できる. インストールされていなければエラーが返ってくるので, それで判断すれば良い. とはいえ, インストール済みのパッケージすべてを確認したいこともあるだろう. その場合は, 作業ディレクトリ上で

- `pip list`

を実行すれば, インストール済みのパッケージと, それらのバージョンを確認できる.
もし, 必要なパッケージがインストールされていなければ,

- `pip install` (パッケージ名)

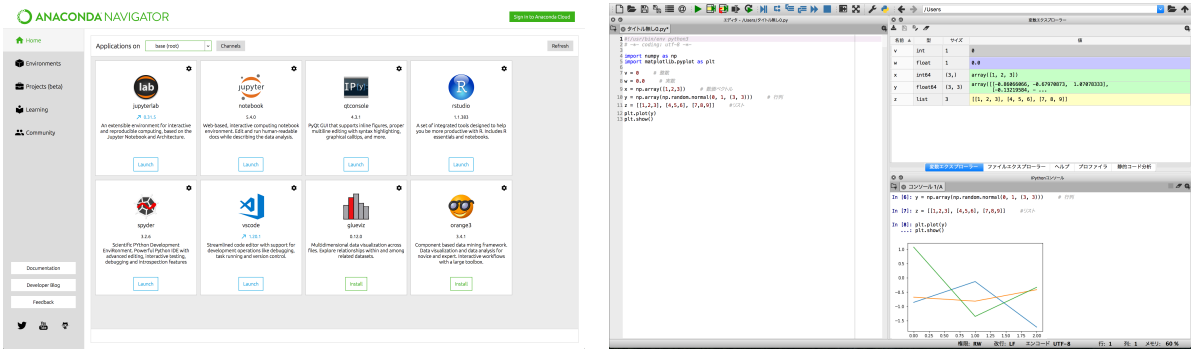
とすることで, インストールできる. インストールした後は, numpy などと同様に, エディタの初めに

- `import` (パッケージ名)

と宣言することで, それ以降そのパッケージに含まれる関数を利用できる.

1.6.2　Anaconda と spyder

自分の環境でも python を利用してみたいというときには, Anaconda を使って python をダウンロードすると良い. Anaconda を使う最大の利点は, python 本体のみならず, よく利用されるパッケージも同時



(a) Anaconda でインストール済みのアプリケーション．rstudio (b) spyder の実行画面．エディタやコンパイラなどを GUI 環境でという，R 言語の統合開発環境もインストールされている．python 利用できる．左がエディタ，右上と右下がそれぞれ変数エクスプローを使いたい場合は，spyder, jupyter notebook, jupyterlab を利用 ラーとコンパイラになっている．
できる．

図 4　Anaconda 起動時の画面 (a) と spyder の実行画面 (b)．

にインストールしてくれる点にある．Anaconda を使わない場合，必要なパッケージは python 本体をインストールした後で個別にインストールしなければならないので割と面倒くさい．ただし，Anaconda を利用することで，独自の python 環境を構築できなくなるので，自分好みの環境の構築したいという人にはオススメしない．とはいえ，初めて python を使う場合には，Anaconda をダウンロードするだけで，環境が確実に完成するというだけでもありがたい．

Anaconda を起動すると，(Mac では) 図 4 のように，インストール済みのアプリケーションが表示される[*14]．python 関連のアプリケーションとして，jupytelab, jupyter notebook, spyder がある．jupyterlab は jupyter notebook とほぼ同じ機能を持つもので，ブラウザ上で python を実行することができる．また，spyder は，いわゆる統合開発環境のことで，エディタとコンパイラなどを GUI として利用できるようにしたものである．spyder では，変数エクスプローラで保存済みの変数の確認や，ショートカットを利用した実行などができる．

2　数学的な補足

“アルゴリズムを実装するのに数学なんていらない!!”と考える人もいるだろうけど，そういう考えは改めておいたほうが良い．少なくとも，常に配列 (行列) のサイズを意識しながら実装することで，ある程度のバグを減らすことができる．また，行列演算を自由に行うことができれば，不要な for 文を減らすことにもつながる．微分に関しても同様である．勾配降下法のようなアルゴリズムは，基本的に目的関数の 1 階微分や 2 階微分を利用して更新が行われる．したがって，関数の微分を計算できなければ，アルゴリズムを実装することさえ困難になってしまう[*15]．

この章では，アルゴリズムの実装でムダにつまずかないために，基本的な線形代数と微分の結果を説明する．そのため，行列やベクトル演算を，道具として自由に使うための準備と捉えてほしい．

2.1　線形代数

線形代数とは書いたものの，ここでの目標は，“行列やベクトルの演算を自由に行うことができるようになること”である．そのため，線形写像としての行列や，そこから導かれる性質[*16]については述べない．

[*14] python だけでなく rstudio など，他の言語もインストールされる．
[*15] まあ，数値微分で頑張って勾配計算することもあるにはあるが，普通に微分できるに越したことはない．
[*16] 線形代数の延長線上に，Banach 空間や Hilbert 空間などがあり，関数解析で非常に重要だということはコメントしておく．

2.1.1 ベクトル空間

まず, ベクトルとはなんだったか思い出そう. n 次元実ベクトル $\boldsymbol{v}$ とは, n 個の実数 $v_1, v_2, \ldots, v_n$ を**縦に**並べたもの (**列ベクトル**) で,

$$\boldsymbol{v} = \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{pmatrix} \qquad \text{あるいは} \qquad \boldsymbol{v} = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix}$$

などと表し, n 次元ベクトルの空間を $\mathbb{R}^n$ で表す. もう少し簡略化された表現として, $\boldsymbol{v} = (v_i)_{i=1,\ldots,n}$ と書いたり, 添字 i の範囲が文脈から明らかである場合には $\boldsymbol{v} = (v_i)_i$ や $\boldsymbol{v} = (v_i)$ などと略記されることもある. なお, $n = 1$ の場合は実数を表すわけだが, この場合にはベクトルと区別してスカラーと呼ぶ (こともある). また, $\boldsymbol{v}$ が n 次元ベクトルであることを $\boldsymbol{v} \in \mathbb{R}^n$ と表す[*17]. 例えば,

$$\boldsymbol{a} = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \qquad \boldsymbol{b} = \begin{pmatrix} 2 \\ 3 \end{pmatrix}$$

はそれぞれ 3 次元ベクトルと 2 次元ベクトルであり, $\boldsymbol{a} \in \mathbb{R}^3, \boldsymbol{b} \in \mathbb{R}^2$ である. 縦に長いベクトルだと, 紙面を大きく圧迫するので,

$$\boldsymbol{a} = (1, 0, 0)^\top, \qquad \boldsymbol{b} = (2, 3)^\top$$

のように, 転置記号 $\top$ を用いて, 横ベクトルのように表現することもある.

ところで, 1 章では, python の出力表示に合わせて, 大カッコ[*18] $[\cdots]$ を用いたが, 丸カッコ $(\cdots)$ の方が広く用いられているように思われる.

さて, ベクトル空間の定義は次の通りであった.

[*17] ベクトル $\boldsymbol{v}$ が n 次元ベクトルのなす集合の要素であることを意味している.

[*18] 角カッコやブラケットともよぶ.

定義 1 (ベクトル空間)**.** 集合 V が $\mathbb{R}$ 上のベクトル空間であるとは, 以下のすべての性質が成り立つ場合をいう[a].

1. 和が定義されている: $^\forall \boldsymbol{x}, \boldsymbol{y} \in V \Rightarrow \boldsymbol{x}+\boldsymbol{y} \in V$. つまり, ふたつの n 次元ベクトルの和はやはり n 次元ベクトル.
2. スカラー倍が定義されている: $^\forall k \in \mathbb{R}, ^\forall \boldsymbol{x} \in V \Rightarrow k\boldsymbol{x} \in V$. つまり, n 次元ベクトルのスカラー倍はやはり n 次元ベクトル.
3. 和とスカラー倍に関して, 次の性質が成り立つ.
   - 加法の交換則: $\boldsymbol{x}+\boldsymbol{y} = \boldsymbol{y}+\boldsymbol{x}$, $^\forall \boldsymbol{x}, \boldsymbol{y} \in V$. つまり, 和の順番を変えても結果は変わらない.
   - 加法の結合則: $(\boldsymbol{x}+\boldsymbol{y})+\boldsymbol{z} = \boldsymbol{x}+(\boldsymbol{y}+\boldsymbol{z})$, $^\forall \boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z} \in V$. つまり, 3 つ以上の和の計算はどこから行っても結果は変わらない.
   - 加法の単位元 (ゼロ元)$\mathbf{0}$ の存在: $^\forall \boldsymbol{x} \in V, ^\exists \mathbf{0} \in V$ s.t. $\boldsymbol{x}+\mathbf{0} = \boldsymbol{x}$. つまり, 和の結果を変えない特別な元 $\mathbf{0}$ が存在する.
   - 加法の逆元の存在: $^\forall \boldsymbol{x} \in V, ^\exists \boldsymbol{x}' \in V$ s.t. $\boldsymbol{x}+\boldsymbol{x}' = \mathbf{0}$. つまり, $\boldsymbol{x}$ を相殺するような特別な元 $\boldsymbol{x}'$ が存在する[b].
   - 加法とスカラーの分配則 I: $k(\boldsymbol{x}+\boldsymbol{y}) = k\boldsymbol{x}+k\boldsymbol{y}$, $^\forall \boldsymbol{x} \in V, \boldsymbol{y} \in V, ^\forall k \in \mathbb{R}$.
   - 加法とスカラーの分配則 II: $(k+l)\boldsymbol{x} = k\boldsymbol{x}+l\boldsymbol{x}$, $^\forall \boldsymbol{x} \in V, ^\forall k, l \in \mathbb{R}$.
   - スカラー倍の結合則: $(kl)\boldsymbol{x} = k(l\boldsymbol{x})$, $^\forall \boldsymbol{x} \in V, ^\forall k, l \in \mathbb{R}$. つまり, 逐次的にスカラー倍を施しても良い.
   - スカラー倍の単位元の存在: $\Rightarrow 1\boldsymbol{x} = \boldsymbol{x}$, $^\forall \boldsymbol{x} \in V$. つまり, スカラー倍の結果を変えない特別な元 $1 \in \mathbb{R}$ が存在する.

[a] より一般に, $\mathbb{R}$ の代わりに体 K(加減乗除ができるもの) を考えて, K 上のベクトル空間 V を定義できる.
[b] つまり, $\boldsymbol{x}' = -\boldsymbol{x}$ のこと.

やや天下りであるが, 二つの n 次元ベクトル $\boldsymbol{x} = (x_i), \boldsymbol{y} = (y_i)$ と任意のスカラー $k \in \mathbb{R}$ に対して,

$$\boldsymbol{x}+\boldsymbol{y} = (x_i+y_i), \qquad k\boldsymbol{x} = (kx_i)$$

によって, ベクトルの和とスカラー倍を定義しよう. ここで, 例えば和の定義では, 左辺はベクトル同士の和であるのに対し, 右辺はスカラー同士の和となっており, "+"の意味が異なることに注意しておく. また, 加法の単位元 (ゼロ元) をゼロベクトルとして

$$\mathbf{0} = \underbrace{(0, \ldots, 0)}_{n \text{ 個}}^\top \in \mathbb{R}^n$$

によって, スカラー 0 が n 個並んだベクトルとして定義する. この代数構造によって $\mathbb{R}^n$ は $\mathbb{R}$ 上のベクトル空間となることが簡単に示せる[*19]. 例えば, 加法の交換則に関しては

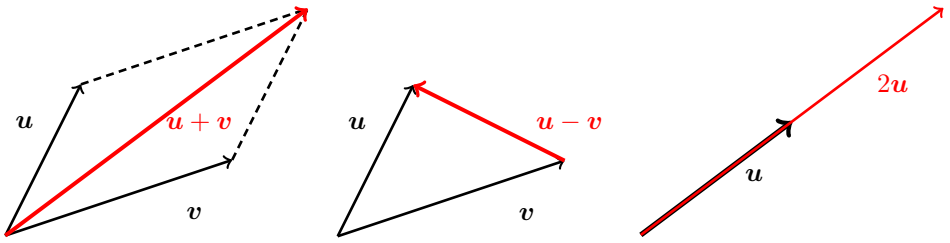
$$\boldsymbol{x}+\boldsymbol{y} \overset{\text{(i)}}{=} (x_i+y_i) \overset{\text{(ii)}}{=} (y_i+x_i) \overset{\text{(iii)}}{=} \boldsymbol{y}+\boldsymbol{x}$$

より, 成立することがわかる. ここで, (i) はベクトルの和の定義, (ii) は実数の和が交換可能なこと, (iii) は再びベクトルの和の定義をそれぞれ用いた.

定義 1 で重要なことは, 和とスカラー倍の定義に応じて, 具体的な計算規則 (3 の和とスカラー倍に関する性質) が定まるということである. また, 定義から, 異なる二つのベクトル空間同士の演算は (少なくとも上の定義からだけでは) 定義できないことにも注意する[*20]. 具体例を挙げておくと, 例えば $\boldsymbol{x} = (1,2,3)^\top, \boldsymbol{y} = (-1,0,3), k = 2$ のとき,

$$\boldsymbol{x}+\boldsymbol{y} = \begin{pmatrix} 0 \\ 2 \\ 6 \end{pmatrix}, \qquad k\boldsymbol{x} = \begin{pmatrix} 2 \\ 4 \\ 6 \end{pmatrix}$$

[*19] 定義 1 の和とスカラー倍に関する性質は, 実際には, 和とスカラー倍の定義に基づいて示すべき命題である.
[*20] 実装でもそうだけど, 例えば, 3 次元ベクトルと 2 次元ベクトルの和は計算できないですよね?



(a) 和　(b) 差　(c) スカラー倍

図 5　ベクトルの和, スカラー倍のイメージ

などとなる. ところで, 和やスカラー倍などの代数的な演算ができるものとしてベクトル空間を定義したが, 図 5 のように, (高校でも勉強した通り) ベクトル空間における演算は幾何的な構造も併せ持っている.

2.1.2　ベクトルの内積とノルム

ベクトルの演算で特によく使うものとして, まず**内積 (inner product)** を定義しよう. 内積とは, 次の条件を見たすものであった.

定義 2 (内積)**.** ベクトル空間 V の内積とは, 以下の条件を満たす写像 $\langle\cdot,\cdot\rangle : V \times V \to \mathbb{R}$ である[a].
双線型性: $\langle \boldsymbol{u}+\boldsymbol{v}, \boldsymbol{w}\rangle = \langle \boldsymbol{u}, \boldsymbol{w}\rangle + \langle \boldsymbol{v}, \boldsymbol{w}\rangle$ & $\langle k\boldsymbol{u}, \boldsymbol{v}\rangle = \langle \boldsymbol{u}, k\boldsymbol{v}\rangle = k\langle \boldsymbol{u}, \boldsymbol{v}\rangle$, $^\forall \boldsymbol{u}, \boldsymbol{v}, \boldsymbol{w} \in V, ^\forall k \in \mathbb{R}$.
対称性: $\langle \boldsymbol{u}, \boldsymbol{v}\rangle = \langle \boldsymbol{v}, \boldsymbol{u}\rangle$, $^\forall \boldsymbol{u}, \boldsymbol{v} \in V$.
非負性: $\langle \boldsymbol{u}, \boldsymbol{u}\rangle \geq 0$ & $\langle \boldsymbol{u}, \boldsymbol{u}\rangle = 0 \Leftrightarrow \boldsymbol{u} = \boldsymbol{0}$, $^\forall \boldsymbol{u} \in V$.

[a] ここでは複素空間の内積は考えない.

さて, 二つのベクトル $\boldsymbol{u} = (u_i), \boldsymbol{v} = (v_i) \in \mathbb{R}^n$ に対して, $\boldsymbol{u}$ と $\boldsymbol{v}$ の (標準) 内積を

$$\langle \boldsymbol{u}, \boldsymbol{v}\rangle = \boldsymbol{u}^\top \boldsymbol{v} = \sum_{i=1}^{n} u_i v_i \tag{1}$$

で定義しよう[*21]. これが内積の定義を満たすことは容易に確認できる. 例えば, 対称性は

$$\boldsymbol{u}^\top \boldsymbol{v} = \sum_{i=1}^{n} u_i v_i = \sum_{i=1}^{n} v_i u_i = \boldsymbol{v}^\top \boldsymbol{u}$$

となることからわかる. ただし, 二つ目の等号では, スカラーの積が交換可能であることを用いた. 具体的には, 例えば,

$$\boldsymbol{u} = \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix}, \ \boldsymbol{v} = \begin{pmatrix} -1 \\ 2 \\ 0 \end{pmatrix} \Rightarrow \boldsymbol{u}^\top \boldsymbol{v} = (1,0,1) \begin{pmatrix} -1 \\ 2 \\ 0 \end{pmatrix} = 1 \times (-1) + 0 \times 2 + 1 \times 0 = -1$$

$$\boldsymbol{x} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \ \boldsymbol{y} = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \Rightarrow \boldsymbol{x}^\top \boldsymbol{y} = (1,0) \begin{pmatrix} 0 \\ 1 \end{pmatrix} = 1 \times 0 + 0 \times 1 = 0$$

のように, スカラーの積と和の演算になる.

ベクトル空間に内積が定まると,

$$\|\boldsymbol{v}\| = \sqrt{\langle \boldsymbol{v}, \boldsymbol{v}\rangle} = \sqrt{\sum_{i=1}^{n} v_i^2}$$

[*21] $(\boldsymbol{u}, \boldsymbol{v})$ と表されることもある. また, 内積の定義を満たす写像が一意に定まるとは限らない. 例えば, 定数倍してもやはり内積の定義を満たすし, ややフライング気味だが, 非負定値対称行列 M に対して $\boldsymbol{u}^\top M \boldsymbol{v}$ もまた内積の定義を満たす.

によって自然に**ノルム**が定義され[*22], したがって, ふたつのベクトル $\boldsymbol{u}, \boldsymbol{v}$ 間に**距離** $\|\boldsymbol{u}-\boldsymbol{v}\|$ を定義することができる. なお, ノルムの定義は以下の通りであり, 上で定義したものがノルムの定義を満たすこともやはり簡単に確認できる[*23].

定義 3 (ノルム)**.** ベクトル空間 V のノルムとは, 以下の条件を満たす写像 $\|\cdot\| : V \to \mathbb{R}$ である.
非退化性: $\|\boldsymbol{v}\| = 0 \Leftrightarrow \boldsymbol{v} = \boldsymbol{0}$
斉次性: $\|k\boldsymbol{v}\| = |k|\|\boldsymbol{v}\|,\ ^\forall\boldsymbol{v} \in V, ^\forall k \in \mathbb{R}$.
劣加法性[a]: $\|\boldsymbol{u}+\boldsymbol{v}\| \le \|\boldsymbol{u}\| + \|\boldsymbol{v}\|,\ ^\forall\boldsymbol{u}, \boldsymbol{v} \in V$.

[a] 三角不等式のこと

注意 1. 線形代数の教科書や参考書を読むと, 定義 3 にくわえ,
非負性: $\|\boldsymbol{v}\| \ge 0,\ ^\forall\boldsymbol{v} \in V$
もノルムの定義として採用されているものが多い. しかし, ノルムの非負性は, 定義 3 の条件をすべて用いることで自動的に成り立つ. つまり, $\|\cdot\|$ がノルムならば, 任意の $\boldsymbol{v} \in \mathbb{R}^n$ に対して,

$$0 = \|\boldsymbol{0}\| = \|\boldsymbol{v} + (-\boldsymbol{v})\| \le \|\boldsymbol{v}\| + \|-\boldsymbol{v}\| = \|\boldsymbol{v}\| + \|\boldsymbol{v}\| = 2\|\boldsymbol{v}\| \Rightarrow \|\boldsymbol{v}\| \ge 0$$

が成り立つ (注意終).

上で定義したノルムは ℓ_2-ノルムと呼ばれ $\|\cdot\|_2$ と表されることもあるが, 特に明示しない限り $\|\cdot\|$ で ℓ_2-ノルムを表すものとする. なぜ特別に ℓ_2-ノルムと呼ぶかというと, 同じベクトル空間でもいろいろなノルムを定義することができるためである. 例えば, ベクトル $\boldsymbol{v} \in \mathbb{R}^n$ に対して,

$$\|\boldsymbol{u}\|_1 = \sum_{i=1}^{n} |v_i|, \qquad \|\boldsymbol{v}\|_p = \left(\sum_{i=1}^{p} v_i^p\right)^{1/p} \quad (p \ge 1), \qquad \|\boldsymbol{v}\|_\infty = \max_{i=1,\dots,n} |v_i|$$

などはそれぞれ ℓ_1-ノルム, ℓ_p-ノルムおよび ℓ_∞-ノルム[*24]と呼ばれ, やはりノルムの定義を満たす. ところで, ℓ_p ノルムにおいて $0 \le p < 1$ の場合はノルムの定義を満たさず (何が満たされず, 代わりにどういう性質があるか考えてみよう), 正確な意味でのノルムにはならないものの, データ解析ではしばしば用いられる. そのため, $0 \le p < 1$ の場合も含めて, ℓ_p-ノルムと呼ばれることがあるので注意してほしい. 特に, ℓ_1-ノルムや ℓ_0-ノルム

$$\|\boldsymbol{v}\|_0 = \lim_{p\to 0} \|\boldsymbol{v}\|_p = (v_i\text{のうち非ゼロであるものの個数})$$

に関する最適化は, モデル選択やスパース推定などと呼ばれる重要な問題と密接に関連している.

定義ばかり続くと退屈なので, ここで高校で勉強した (はずの) 内積とここで定義した内積の関係を考えてみよう. 高校では, ふたつのベクトル $\boldsymbol{u}$ と $\boldsymbol{v}$ の内積は

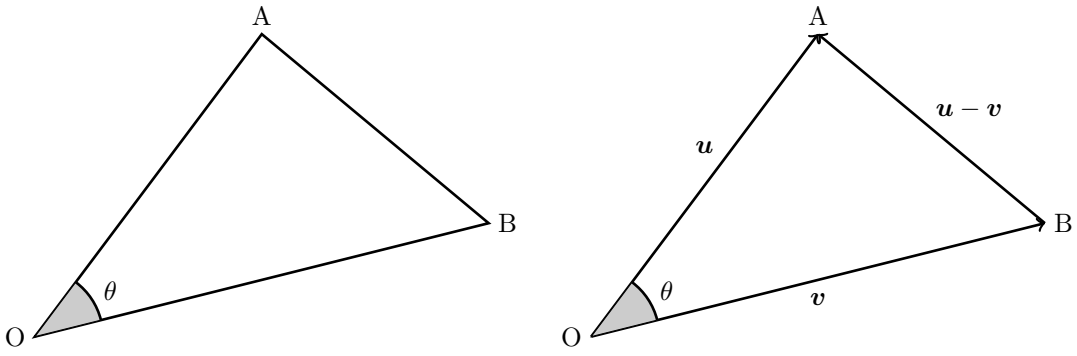
$$\langle \boldsymbol{u}, \boldsymbol{v} \rangle = \|\boldsymbol{u}\|\|\boldsymbol{v}\| \cos\theta \tag{2}$$

と教わったはずだ. ここで, θ はベクトル $\boldsymbol{u}$ と $\boldsymbol{v}$ のなす角である[*25]. 実は, 高校で勉強した内積 (2) は, 初めに定義した内積の定義 (1) と一致することを示すことができる. まず, 余弦定理を思い出そう. 余弦定理とは, 図 6(a) のような三角形が与えられたときに,

$$c^2 = a^2 + b^2 - 2ab\cos\theta$$

が成り立つというものであった. ただし, $a = |\overrightarrow{\mathrm{OA}}|, b = |\overrightarrow{\mathrm{OB}}|, c = |\overrightarrow{\mathrm{AB}}|$ である. いま, $\|\boldsymbol{v}\| = a, \|\boldsymbol{v}\| = b$

[*22] これを内積から誘導されたノルムと呼ぶこともある.
[*23] 三角不等式だけちょっと面倒かもしれない.
[*24] sup ノルムとも呼ばれる.
[*25] 3 次元以上の場合, どういう向きで角度を図るかという微妙な問題があるがここではあまり気にしないでおく.



(a) (b)

図 6　余弦定理とベクトルの関係

および $\|\boldsymbol{u}-\boldsymbol{v}\| = c$ を満たすふたつのベクトル $\boldsymbol{u}, \boldsymbol{v}$ を考えると, 図 6(b) のように, 三角形 ABC と合同な三角形が得られる. すると, ノルムの定義より

$$c^2 = \|\boldsymbol{u}-\boldsymbol{v}\|^2 = (\boldsymbol{u}-\boldsymbol{v})^\top(\boldsymbol{u}-\boldsymbol{v}) = \|\boldsymbol{u}\|^2 - 2\boldsymbol{u}^\top\boldsymbol{v} + \|\boldsymbol{v}\|^2 = a^2 + b^2 - 2\boldsymbol{u}^\top\boldsymbol{v}$$

であり, 両辺を見比べれば

$$\boldsymbol{u}^\top\boldsymbol{v} = ab\cos\theta = \|\boldsymbol{u}\|\|\boldsymbol{v}\|\cos\theta$$

が成り立つ. ベクトル $\boldsymbol{u}$ と $\boldsymbol{v}$ の内積が 0 ならば $\boldsymbol{u}$ と $\boldsymbol{v}$ は直交するというが[*26], これは $\cos\theta = 0$, つまり $\theta = 90^\circ$ ということと同値であり, 幾何学的な直感と一致している. つまり, ふたつのベクトル $\boldsymbol{u}$ と $\boldsymbol{v}$ が直行していれば, 三平方の定理 (**Pythagoras の定理**) が成り立つ. また, 劣加法性についても, 二辺の長さの和は残りの一辺の長さよりも大きいという, よく知られている結果と一致している. ところで, 任意の実数 θ に対して, $|\cos\theta| \leq 1$ であるから,

$$\left|\frac{\boldsymbol{u}^\top\boldsymbol{v}}{\|\boldsymbol{u}\|\|\boldsymbol{v}\|}\right| \leq 1 \Leftrightarrow |\boldsymbol{u}^\top\boldsymbol{v}| \leq \|\boldsymbol{u}\|\|\boldsymbol{v}\|$$

が成り立つ. 最後の不等式は, **Schwarz の不等式**として知られている.

2.1.3　行列といろいろな演算

行列とは, ベクトルと同様に実数を並べたものであり, ベクトルが実数を直線的 (1 方向) に並べたものであるのに対し, 行列は平面的 (2 方向) に並べたものである. より正確には, $m \times n$ 次元行列とは

$$A = \begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{pmatrix} \quad \text{または} \quad A = \begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{bmatrix}$$

のように実数を並べたものであり, 行列の各要素を成分と呼び, 行列 A が $m \times n$ 行列であることを, $A = (a_{ij})_{i=1,\ldots,m;j=1,\ldots,n} \in \mathbb{R}^{m\times n}$ と表す. 添字の動く範囲が明らかな場合には単に $A = (a_{ij})$ とも表す. $\boldsymbol{a}_j = (a_{ij})_{i=1,\ldots,m}$ を A の**列ベクトル**, $\mathbf{a}_i^\top = (a_{ij})_{j=1,\ldots,n}$ を A の行ベクトルと呼ぶ[*27]. 行ベクトルの定義に転置記号があるのは, 行ベクトルが横ベクトルであるためである. 列ベクトルや行ベクトルを用いれば,

[*26] $\boldsymbol{u}\perp\boldsymbol{v}$ と書くこともある.
[*27] 表記が紛らわしいので, 列ベクトルを太字のイタリック, 行ベクトルを太字のローマンで表すことにする.

行列 A は

$$A = (\boldsymbol{a}_1, \ldots, \boldsymbol{a}_n) = \begin{pmatrix} \mathbf{a}_1^\top \\ \vdots \\ \mathbf{a}_m^\top \end{pmatrix}$$

と書くことができる. 例えば,

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

とすると,

$$\begin{pmatrix} 1 \\ 4 \\ 7 \end{pmatrix}, \qquad \begin{pmatrix} 2 \\ 5 \\ 8 \end{pmatrix}, \qquad \begin{pmatrix} 3 \\ 6 \\ 9 \end{pmatrix}$$

はそれぞれ A の第 1, 第 2 および第 3 列ベクトルであり,

$$(1,2,3), \qquad (4,5,6), \qquad (7,8,9)$$

はそれぞれ A の第 1, 第 2 および第 3 行ベクトルである.

ベクトルと異なり, 成分の並びが平面的になったことで, 複雑で難しくなったように思えるが, ただ単に実数が並んだだけである. 別に特別難しい数学的な記号を作ったわけではないので, データベースやテキストファイルのようなものだと思ってほしい.

もう少し詳細になぜ行列を考える必要があるかについて述べよう. 定義 1 で説明した通り, ベクトル空間が与えられると, そこには和とスカラー倍が定義された. また, 加法の結合法則から, ベクトルを好きなだけスカラー倍して足しあげるという操作ができる. つまり, n 個の m 次元ベクトル $\boldsymbol{a}_1 = (a_{i1}), \ldots, \boldsymbol{a}_n = (a_{in}) \in \mathbb{R}^m$ と n 個のスカラー $v_1, \ldots, v_n \in \mathbb{R}$ に対して, これらの線形結合

$$v_1\boldsymbol{a}_1 + \cdots + v_n\boldsymbol{a}_n \tag{3}$$

を考えることができる. すると, 内積の定義と同じように考え, $\boldsymbol{v} = (v_i) \in \mathbb{R}^n$ とすることで, (3) は

$$\begin{aligned} v_1\boldsymbol{a}_1 + \cdots + v_n\boldsymbol{a}_n &= \boldsymbol{a}_1 v_1 + \cdots + \boldsymbol{a}_n v_n \\ &= (\boldsymbol{a}_1, \ldots, \boldsymbol{a}_n) \begin{pmatrix} v_1 \\ \vdots \\ v_n \end{pmatrix} = A\boldsymbol{v} \end{aligned} \tag{4}$$

と書き換えることができそうな気がするし, 実際にこの予想は適当に積を定義することができれば, 正しい. 最後の式 $A\boldsymbol{v}$ が具体的にどのような演算で定義されるかはすぐ後で説明する. ところで, 勘の鋭い方は, 二つ目の等号で

$$(v_1, \ldots, v_n) \begin{pmatrix} \boldsymbol{a}_1 \\ \vdots \\ \boldsymbol{a}_n \end{pmatrix}$$

としても良いように思われるかもしれないが, 残念ながらこれは間違いである. 内積の定義を思い返してみると, 内積とは同じ長さのベクトル間で定義されるものであった. ところが, $\boldsymbol{a}_i \in \mathbb{R}^m$ を縦に並べると, 結果として mn 次元の列ベクトルが出てきてしまい n 次元ベクトルである $v_1, \ldots, v_n$ と長さが異なってしまう. 一方, 正しい計算では, m 次元ベクトルを横に n 個並べるわけだから, 行方向で見ると n 次元ベクトル同士の内積になっているはずだ. これを確認するため, 一度 (3) の各成分がどういう状況にあるかを考えてみよう. ベクトル演算の定義より, (3) は

$$\begin{pmatrix} v_1 a_{11} + \cdots + v_n a_{1n} \\ \vdots \\ v_1 a_{m1} + \cdots + v_n a_{mn} \end{pmatrix}$$

であるような m 次元ベクトルであり, その第 i 成分は

$$v_1 a_{i1} + \cdots + v_n a_{in} = \sum_{j=1}^{n} v_j a_{ij}$$

で与えられる. 最後の式は, 内積の定義より $\mathbf{a}_i^\top \boldsymbol{v}$, つまり, A の行ベクトルと $\boldsymbol{v}$ の内積で書くことができる. したがって, (3) は $\mathbf{a}_i^\top \boldsymbol{v}$ を m 個並べたベクトルであり, これが (4) に等しいので,

$$A\boldsymbol{v} = \begin{pmatrix} \mathbf{a}_1^\top \boldsymbol{v} \\ \vdots \\ \mathbf{a}_m^\top \boldsymbol{v} \end{pmatrix} \tag{5}$$

として, 行列 $A \in \mathbb{R}^{m\times n}$ とベクトル $\boldsymbol{v} \in \mathbb{R}^n$ の積が定まる.

行列とベクトルの積が (5) で定義されることは分かったが, 行列同士の和やスカラー倍はどうなるだろうか. 理想的には, 行列の和が換法則や結合法則を持っており, しかも, 行列とベクトルの積の間に分配法則などの演算ができると嬉しい. つまり,

$$(A+B)\boldsymbol{v} = A\boldsymbol{v} + B\boldsymbol{v}, \qquad k(A\boldsymbol{v}) = (kA)\boldsymbol{v}$$

などが成り立つように行列演算を定義したい[*28]. 結果から述べると, ベクトルの和やスカラー倍と同様に, 成分ごとに和やスカラー倍を定義する. したがって, $A = (a_{ij}), B = (b_{ij}) \in \mathbb{R}^{m\times n}$ および $k \in \mathbb{R}$ が与えられたとき,

$$A + B = (a_{ij} + b_{ij}), \qquad kA = (ka_{ij})$$

とすれば良い. なお, この和とスカラー倍によって, $m \times n$ 行列のなす空間もまたベクトル空間 (線形空間) になる.

行列とベクトルの演算に関しては, 上の定義でほとんどの計算が可能となる. あと, 定義しておくと便利なのは行列同士の積である. 行列同士の積は, (5) のように考えれば良い. つまり, $A \in \mathbb{R}^{m\times n}$ と $\boldsymbol{b}_1, \ldots \boldsymbol{b}_q \in \mathbb{R}^n$ に対して,

$$(A\boldsymbol{b}_1, \ldots, A\boldsymbol{b}_p) = A(\boldsymbol{b}_1, \ldots, \boldsymbol{b}_p) = AB \in \mathbb{R}^{m\times p}$$

とする. 正確には, 二つの行列 $A \in \mathbb{R}^{m\times n}, B \in \mathbb{R}^{n\times p}$ に対して, 積 AB の各成分は

$$AB = (\mathbf{a}_i^\top \boldsymbol{b}_j)_{i=1,\ldots,m;j=1,\ldots,p},$$

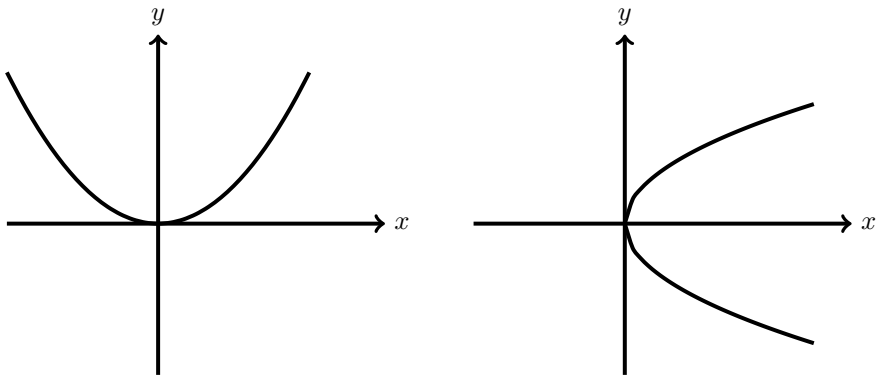
つまり, A の行ベクトルと B の列ベクトルの内積で与えられる. ここで, A の列数と B の行数が等しいことに注意する. これらが等しくない場合は, 行列の積が計算できないので, 計算機で実装する際や, て計算する場合などは行列のサイズを常に意識しておくと良い.

おまけとして, 行列の積の意味について説明しておく. 行列 $A \in \mathbb{R}^{m\times n}$ とベクトル $\boldsymbol{v} \in \mathbb{R}^n$ の積は m 次元ベクトル $A\boldsymbol{v}$ であったことを思い出そう. つまり, 行列 A によって, ベクトル $\boldsymbol{v}$ は n 次元から m 次元のベクトルへと変換される. これを写像の言葉を使うと, 行列 A とは, n 次元ベクトル空間 $\mathbb{R}^n$ を m 次元ベクトル空間 $\mathbb{R}^m$ に移す写像 $A : \mathbb{R}^n \to \mathbb{R}^m$ である, ということになる. 一方, 行列 $B \in \mathbb{R}^{p\times m}$ を考えると, B は写像 $B : \mathbb{R}^m \to \mathbb{R}^p$ を表す. A によって変換された $\boldsymbol{v} \in \mathbb{R}^n$ の像は $A\boldsymbol{v} \in \mathbb{R}^m$ であり, これをさらに B で移すと

$$B(A\boldsymbol{v}) = BA\boldsymbol{v} \in \mathbb{R}^p$$

となるので, 行列の積 BA は, 写像 $A : \mathbb{R}^n \to \mathbb{R}^m$ と $B : \mathbb{R}^m \to \mathbb{R}^p$ の合成写像 $B \circ A$ と解釈できる.

[*28] 二つ目の等式の左辺は, m 次元ベクトル $A\boldsymbol{v}$ の k 倍, 右辺はスカラー倍した $m \times n$ 行列 kA とベクトル $\boldsymbol{v}$ の積を表している.



(a) 関数の例: 一つの x に対して, ただ一つの y が定まる.

(b) 関数ではない例: 一つの x に対して, 二つの y が対応している.

図 7　関数と呼べるものとそうでないものの例

2.2　微分

微分や積分などの解析的な計算は, 線形代数と同様に基本的な現代のデータ解析ツールの一つになっている. この講義では少なくとも積分計算は出てこないので, 具体的な関数の微分についての定義や計算方法について説明する. 特に, 深層学習ではパラメータがベクトルや行列などの配列で与えられることが多いため, その辺りを中心に述べることにする. 以降では, 関数の出力はスカラーであるとする.

2.2.1　一変数関数の微分

そもそも関数[*29]とは, ある変数 x に対して, 別の変数 y を一つ対応させる操作 $x \mapsto y$ のことであった. 例えば, $y = x^2$ と書くと, 実数 $x \in \mathbb{R}$ に対してその 2 乗を出力するものである. より一般に, $y = f(x)$ と書いたときには, $x = a$ を代入したときに決まる関数の値として $f(a)$ を定める, と言う具合である.

適当な集合 V の元を実数 $\mathbb{R}$ に対応させる写像 $f : V \to \mathbb{R}$ が “関数” と呼ばれるためには, 一つの $x \in V$ に対して, 一つの $y \in \mathbb{R}$ が定まらなければならない. 例えば, 上で書いた $y = f(x) = x^2$ は関数であるが, x について解き直した $x = \pm\sqrt{y}$ は y に対応する値が一意に定まらないため y の関数とは呼ばない (図 7 参照). 一方で, $x \in \mathbb{R}$ に対して, $f(x) = (2x + 1, x^2)^\top$ は $x \in \mathbb{R}$ に対して, 2 次元ベクトル $(2x + 1, x^2)$ をただ一つ対応させるため関数である. 何が言いたいかというと, 一つの x に対して, y_1 または y_2 を対応させるものは関数と呼ばないのである. とはいえ, そんなことを言っていると, いわゆる**逆関数**を定義できないので, 通常は逆関数を計算する場合は適当な制限を置くことで逆関数を定義することが一般的である[*30].

さて, 多少話題が逸れたので, 本節のテーマである, 一変数関数の微分について述べよう. 一変数関数とは, 関数の引数が実数 $x \in \mathbb{R}$ であるような関数のこととする. 一変数関数 $f : \mathbb{R} \to \mathbb{R}$ が微分可能であるとは, 次のことを言うのであった. 以下の定義では, 関数の微分とは連続関数についての性質であることに注意する[*31].

[*29] 古い教科書には函数と書かれている. もともとは中国語から輸入されたものであるが, 当用漢字に含まれなかったことから “関数” と記載されることになったらしい.

[*30] ちなみに, 関数 $f(x)$ に対して, $g(f(x)) = x$ となるような g を f の逆関数と呼び $g = f^{-1}$ と表す.

[*31] 微分可能ならば連続だが, 連続だからといって微分可能であるとは限らない (e.g., $f(x) = |x|$).

定義 4 (一変数関数の微分)**.** 任意の $x \in \mathbb{R}$ に対して, 連続な一変数関数[a] $f : \mathbb{R} \to \mathbb{R}$ に関する極限

$$\lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

が存在するとき, f は微分可能であるといい, そのときの極限の値を $f'(x)$ と書くこのとき, $f'(x)$ を $f(x)$ の x での微分と呼ぶ. $f'(x)$ はまた,

$$\frac{\mathrm{d}f(x)}{\mathrm{d}x}$$

とも書かれる. $x = a$ での微分係数を

$$f'(a) \quad \text{や} \quad \left.\frac{\mathrm{d}f(x)}{\mathrm{d}x}\right|_{x=a}, \quad \nabla f(a)$$

などと書く.

[a] 実数値関数が連続であるとは, $^\forall a \in \mathbb{R}, \lim_{x \to a} f(x) = f(a)$ となることであった.

実際に微分の定義を用いて関数の微分を計算する機会はあまり多くないと思うがいくつか例を挙げておく.

例 1. 関数 $f(x) = x^2$ は微分可能で, その微分は $f'(x) = 2x$ である. なぜなら, 任意の実数 $x \in \mathbb{R}$ に対して,

$$\frac{(x+h)^2 - x^2}{h} = 2x + h \to 2x \quad \text{as } h \to 0$$

となるためである.

例 2. 関数 $f(x) = 1/x$ は微分可能で, その微分は $f'(x) = -1/x^2$ である. なぜなら, 任意の実数 $x \in \mathbb{R}$ に対して,

$$\frac{1/(x+h) - 1/x}{h} = \frac{-1}{x(x+h)} \to -\frac{1}{x^2} \quad \text{as } h \to 0$$

となるためである.

例 3. より一般に, 任意の実数 α に対して, $f(x) = x^\alpha$ は微分可能で, その微分は $f'(x) = \alpha x^{\alpha-1}$ である. 証明は省略するが, ざっくりと言うと, まず自然数 n に対して二項定理を用いて $f'(x) = nx^{n-1}$ となることを確認する. 次に, 自然数 n の代わりに $m = -n$ (整数) でも同じことが成り立つことを, 二項定理を利用して確認する. そして, $q = m/n$ (m, n は整数) に対して, x^{qn} を考えて (整数の場合に帰着させることで), やはり同じことが成り立つことを確認する. 最後に, 実数の連続性により, 実数 α をいくらでも近い有理数で近似する (有理数の極限 $q \to \alpha$ を取る) ことで任意の実数 α に対して $f(x) = x^\alpha$ が微分可能であることを確認できる[*32].

例 4. 関数 $f(x) = \sin x$ は微分可能で, その微分は $f'(x) = \cos x$ である. なぜなら, 任意の実数 x に対して, 三角関数の加法定理より

$$\frac{\sin(x+h) - \sin x}{h} = \frac{\sin x(\cos h - 1) + \cos x \sin h}{h} = \sin x \frac{\cos h - 1}{h} + \cos x \frac{\sin h}{h}$$

となるが,

$$\frac{\sin h}{h} \to 1 \quad \& \quad \frac{\cos h - 1}{h} = -\frac{\sin h}{h} \frac{\sin h}{\cos h + 1} \to -1 \times \frac{0}{1+1} = 0 \quad \text{as } h \to 0$$

となるためである.

[*32] step by step で関数の性質を調べる良い例である. Lebesgue 積分も似たような考え方で積分を定義する (単関数 → 非負関数 → 一般の関数).

表 4　代表的な初等関数とその微分

関数 $f(x)$	$f(x)$ の微分
$x^\alpha\ (\alpha \in \mathbb{R})$	$\alpha x^{\alpha-1}$
e^x	e^x
$\log x$	$1/x$
$\sin x$	$\cos x$
$\cos x$	$-\sin x$
$\tan x$	$1+\tan^2 x$
$\sinh x = (e^x - e^{-x})/2$	$\cosh x$
$\cosh x = (e^x + e^{-x})/2$	$\sinh x$
$\tanh x = \sinh x/\cosh x$	$1-\tanh^2 x$

自然対数の底 (ネイピア数) を

$$e = \lim_{h\to 0}(1+h)^{1/h}$$

で定義する. このとき, ものすごくざっくり言えば,

$$e \approx (1+h)^{1/h} \Leftrightarrow \frac{e^h-1}{h} \approx 1 \Rightarrow \lim_{h\to 0}\frac{e^h-1}{h} = 1$$

が成り立つ. $\approx$ の正しい解説はいろいろな解析の教科書や参考書に書いてあるはずなのでそちらを参照してほしい. このことを認めれば, 指数関数と対数関数の微分について以下のことが成り立つ.

例 5. 関数 $f(x) = e^x$ は微分可能で, その微分は $f'(x) = e^x$ である. なぜなら, 任意の実数 x に対して,

$$\frac{e^{x+h}-e^x}{h} = e^x \times \frac{e^h-1}{h} \to e^x \quad \text{as } h \to 0$$

となるためである.

例 6. 関数 $f(x) = \log x$ は $x > 0$ で微分可能で, その微分は $f'(x) = 1/x$ となる. なぜなら,

$$\frac{\log(x+h)-\log x}{h} = \log\left\{\left(1+\frac{h}{x}\right)^{x/h}\right\}^{1/x} \to \log e^{1/x} = \frac{1}{x} \quad \text{as } h \to 0$$

となるためである.

初等的な関数[*33]の微分を表 4 にまとめたので, 必要であれば参照してほしい.

2.2.2　一変数関数の合成関数とその微分

前節で簡単な関数の微分について説明した. ところが, 実際にデータ解析を行う場合には, もう少し複雑な関数の微分が必要な場合が多々ある. そこで, 本節では合成関数に対して成立する微分の規則について述べる[*34]. 和や差の微分に関しては, 関数ごとに微分すれば良いのでここでは割愛する.

まず, 関数の積と商に関する微分について復習する. 関数の積に関しては “ビブンそのまま + そのままビブン”, 商に関しては “ビブンそのまま − そのままビブン” という感じで勉強したと思う. まずはこの辺りを正しく理解しよう. まずは積の微分だ. 関数 h が, ある関数 $f(x)$ と $g(x)$ を用いて $h(x) = f(x)g(x)$ と変

[*33] 初等関数とは, 大雑把に言えば, 高校までで習う関数とその合成で得られる関数のこと. ガンマ関数やベッセル関数などは初等関数ではない.

[*34] とはいえ, この辺りはほとんど高校で教わる数学の復習である.

えているとする．例えば $f(x) = e^x$, $g(x) = \cos x$ ならば $h(x) = e^x \cos x$ という具合である．まず, 微分の定義 4 に沿って計算すると,

$$\begin{aligned}\frac{h(x+h)-h(x)}{h} &= \frac{f(x+h)g(x+h)-f(x)g(x)}{h} \\ &= f(x+h)\frac{g(x+h)-g(x)}{h} + \frac{f(x+h)-f(x)}{h}g(x)\end{aligned}$$

となるが, $f(x)$ と $g(x)$ が微分可能ならば $h \to 0$ の極限を取ることで

$$\to f(x)g'(x) + f(x)g'(x) = h'(x)$$

となることがわかる．つまり, 高校で教わったとおりである．

次に, 商の微分を考えよう．微分可能な関数 $f(x)$ と $g(x)$ を用いて, $h(x) = f(x)/g(x)$ と書けているとする．ただし, $g(x)$ は x の各点で 0 ではないとする[*35]．このとき,

$$\begin{aligned}\frac{h(x+h)-h(x)}{h} &= \frac{1}{h}\left(\frac{f(x+h)}{g(x+h)} - \frac{f(x)}{g(x)}\right) \\ &= \frac{1}{h} \times \frac{f(x+h)g(x) - f(x)g(x+h)}{g(x+h)g(x)} \\ &= \frac{1}{h} \times \frac{(f(x+h)-f(x))g(x) - f(x)(g(x+h)-g(x))}{g(x+h)g(x)} \\ &= \frac{1}{g(x+h)g(x)}\left(\frac{f(x+h)-f(x)}{h}g(x) - f(x)\frac{g(x+h)-g(x)}{h}\right) \\ &\to \frac{1}{g(x)^2}(f'(x)g(x) - f(x)g'(x)) \quad \text{as } h \to 0\end{aligned}$$

より, 商の微分規則が導出できる．特に, $f(x) = 1$ ならば, $f'(x) = 0$ なので

$$h'(x) = -\frac{g'(x)}{g(x)^2}$$

となる．

最後に合成関数の微分について説明しよう．合成関数の微分は**連鎖律**としても知られており, ニューラルネットワークの推定アルゴリズムを実装する上で重要な役割を果たす．合成関数について簡単に復習しよう．二つの関数 $f : U \to V, g : V \to W$ が与えられたとき, f と g の合成関数 $g \circ f : U \to W$ は U を W に移す関数である．具体的には, $x \in U$ は f によって $f(x) \in V$ に移されるが, これをさらに g で $g(f(x))$ に移すのである．例えば, $f(x) = \sin x, g(x) = e^x$ なら,

$$g \circ f(x) = g(f(x)) = e^{f(x)} = e^{\sin x}$$

となる．微分の定義より, 合成関数の微分は

$$\frac{g \circ f(x+h) - g \circ f(x)}{h} = \frac{g(f(x+h)) - g(f(x))}{f(x+h) - f(x)} \times \frac{f(x+h)-f(x)}{h}$$

であるが, f は連続関数なので, $h \to 0$ で $f(x+h) \to f(x)$ であるから, 第 1 項で $h \to 0$ の代わりに $f(x+h) \to f(x)$ の極限を考えることで,

$$\begin{aligned}\frac{g(f(x+h)) - g(f(x))}{f(x+h) - f(x)} &\to g'(f(x)), \\ \frac{f(x+h)-f(x)}{h} &\to f'(x) \qquad \text{(微分の定義より)}\end{aligned}$$

[*35] 0 になる場合は, 適当に 0 になるところでの微分を定義すれば良い.

となる．まとめると, 合成関数 $g \circ f$ の微分は

$$(g \circ f)'(x) = \lim_{h \to 0} \frac{g \circ f(x+h) - g \circ f(x)}{h} = g'(f(x))f'(x)$$

となる．ただし, この議論にはやや穴がある ($h = 0$ の近傍で g の値が一定となる場合) ので, 興味があれば適宜参考書などを確認して各自納得してほしい．連鎖律という言葉の意味を納得するために, 4 つの関数 $f_1, f_2, f_3, f_4 : \mathbb{R} \to \mathbb{R}$ の合成関数

$$(f_4 \circ f_3 \circ f_2 \circ f_1)(x) = f_4(f_3(f_2(f_1(x))))$$

を考えよう．$f_3 \circ f_2 \circ f_1 = g_3$ を一つの関数と見ることで, 合成関数の微分から

$$(f_4 \circ g_3)'(x) = f_4'(g_3(x))g_3'(x)$$

となる．$f_2 \circ f_1 = g_2$ とすれば, $g_3 = f_3 \circ g_2$ であるから, 再び合成関数の微分を用いて

$$g_3'(x) = (f_3 \circ g_2)'(x) = f_3'(g_2(x))g_2'(x)$$

が得られる．最後に, やはり合成関数の微分から

$$g_2'(x) = (f_2 \circ f_1)'(x) = f_2'(f_1(x))f_1'(x)$$

がわかる．つまり, 微分が始めの合成関数から g_3 や g_2 を通して関数の内側に向けて “連鎖” しているように思えるわけである．ところで, 合成関数は合成する順番を変えると, 対応する出力が変わるので注意すること．

例をあげてこの節を終えよう．

例 7. $f(x) = \sin x, g(x) = e^{-x}$ とする．もちろん, $f'(x) = \cos x, g'(x) = -e^{-x}$ である．このとき,

$$\begin{aligned}
h_1(x) &= f(x)g(x) = e^{-x}\sin x \\
h_2(x) &= \frac{f(x)}{g(x)} = e^{x}\sin x \\
h_3(x) &= (f \circ g)(x) = f(e^{-x}) = \sin e^{-x} \\
h_4(x) &= (g \circ f)(x) = g(\sin x) = e^{-\sin x}
\end{aligned}$$

とすれば, それぞれの関数の微分は

$$\begin{aligned}
h_1'(x) &= f'(x)g(x) + f(x)g'(x) = e^{-x}\cos x - e^{-x}\sin x = e^{-x}(\cos x - \sin x) \\
h_2'(x) &= \frac{f'(x)g(x) - f(x)g'(x)}{g(x)^2} = \frac{e^{-x}\cos x + e^{-x}\sin x}{(e^{-x})^2} = e^{x}(\cos x + \sin x) \\
h_3'(x) &= f'(g(x))g'(x) = f'(e^{-x})(-e^{-x}) = -e^{-x}\cos e^{-x} \\
h_4'(x) &= g'(f(x))f'(x) = g'(\sin x)\cos x = -e^{-\sin x}\cos x
\end{aligned}$$

となる[*36].

2.2.3 多変数関数の微分

引数が 1 次元でなく多次元の場合, f を特に多変数関数と呼ぶ．この節と次の説では, 多変数関数のうち, 引数がベクトルや行列で与える場合についての微分 (特に偏微分) について説明する[*37].

n 次元ベクトルを引数に持つ多変数関数 $f : \mathbb{R}^n \to \mathbb{R}$ を考える．例えば, 変数 x, y, z に対して,

$$f(x, y, z) = x + y + z, \qquad g(x, y, z) = x^2 + 2xy + z^2 - 1$$

[*36] h_1 は $\sin x$ と e^x の商, h_2 は $\sin x$ と e^x の積だと思うことも, もちろんできる．

[*37] 多変数関数には, 全微分や方向微分と呼ばれる概念もあり, 非常に重要なものではあるが, ここでは割愛する．

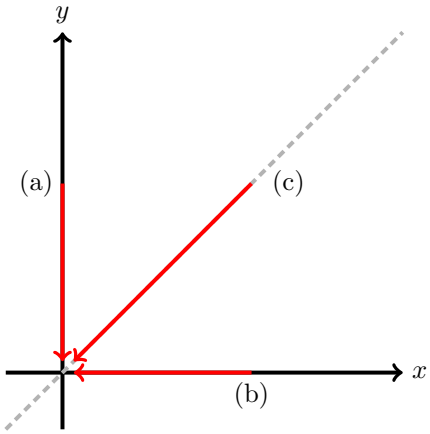


図 8　極限の取り方. (a) case1, (b) case2, (c) case3

は多変数 (3 変数) 関数である. 1 変数の場合とは異なり, 多変数関数の微分は**偏微分 (partial derivative)** と呼ばれる[*38]. 偏 (partial) の意味は, 興味のある変数以外を固定された数だと思って微分するためである.

まず, 多変数関数が連続であるとはどういうことか述べておく.

定義 5 (多変数関数の連続性)**.** $\boldsymbol{x} = (x_1, \ldots, x_n) \in \mathbb{R}^n$ に関する n 変数関数 $f : \mathbb{R}^n \to \mathbb{R}$ が点 $\boldsymbol{a} \in \mathbb{R}^n$ で連続であるとは,

$$\lim_{\boldsymbol{x} \to \boldsymbol{a}} f(\boldsymbol{x}) = f(\boldsymbol{a})$$

が成り立つことである.

定義 5 に現れる極限の $\boldsymbol{x} \to \boldsymbol{a}$ は, $\boldsymbol{x}$ をどの方向から $\boldsymbol{a}$ に近づけても “同じ値” $f(\boldsymbol{a})$ に収束するという意味である. これは 1 変数の場合にはないものであり, 注意が必要だ.

例 8. 2 変数関数

$$f(x, y) = \frac{x^2 - y^2}{x^2 + y^2}$$

を定義し, 原点 $(0, 0)$ での連続性を考えよう. 以下, 図 8 の 3 つの状況を考えてみる.

case1:　$x = 0$ を固定し, $y \to 0$ とする.

case2:　$y = 0$ を固定し, $x \to 0$ とする.

case3:　$x = y$ とし, $x\ (= y) \to 0$ とする.

まず, case1 と case2 の場合はそれぞれ,

$$\lim_{(0,y)\to(0,0)} \frac{x^2 - y^2}{x^2 + y^2} = \lim_{y \to 0} \frac{0^2 - y^2}{0^2 + y^2} = -1$$

および

$$\lim_{(x,0)\to(0,0)} \frac{x^2 - y^2}{x^2 + y^2} = \lim_{x \to 0} \frac{x^2 - 0^2}{x^2 + 0^2} = 1$$

となる. さらに, case3 では

$$\lim_{(x,x)\to(0,0)} \frac{x^2 - y^2}{x^2 + y^2} = \lim_{x \to 0} \frac{x^2 - x^2}{x^2 + x^2} = 0$$

となり, いずれの場合でも異なる極限が現れてしまう. したがって, 関数 f は原点で連続ではない.

[*38] 1 変数関数の微分は**常微分**と呼ばれる.

例 9. 2 変数関数

$$g(x,y)=\frac{xy}{\sqrt{x^2+y^2}}$$

は, $g(0,0)$ ならば原点で連続である. $(x,y)\to(0,0)$ の極限を考える代わりに, (x,y) を $x=r\cos\theta, y=r\sin\theta$ と極座標変換し, 任意 θ に対して, $r\to 0$ の極限を考える. このとき,

$$\lim_{(x,y)\to(0,0)}\frac{xy}{\sqrt{x^2+y^2}}=\lim_{r\to 0}\frac{r^2\sin\theta\cos\theta}{\sqrt{r^2(\cos^2\theta+\sin^2\theta)}}=\lim_{r\to 0}r\sin\theta\cos\theta=0$$

が任意の θ について成り立つ. したがって, x,y に関して, どの方向から極限を取っても 0 に収束することがわかり, 結果として $g(x,y)$ は原点で連続となる.

先に述べたように, 連続な多変数関数に対して, 興味のある変数以外を定数だと思って微分したものを偏微分と呼ぶ. 例えば, はじめに定義した

$$g(x,y,z)=x^2+2xy+z^2-1$$

の x に関する偏微分は

$$\begin{aligned}&\lim_{h\to 0}\frac{g(x+h,y,z)-g(x,y,z)}{h}\\&=\lim_{h\to 0}\frac{\{(x+h)^2+2(x+h)y+z^2-1\}-(x^2+2xy+z^2-1)}{h}=2x+2y\end{aligned}$$

となる. 同様に, $g(x,y,z)$ の y および z での微分はそれぞれ

$$\lim_{h\to 0}\frac{g(x,y+h,z)-g(x,y,z)}{h}=2x,\qquad\lim_{h\to 0}\frac{g(x,y,z+h)-g(x,y,z)}{h}=2z$$

となる. 1 変数関数の微分とほぼ同様であるが, 偏微分の定義は次の通りである.

定義 6 (偏微分)**.** 連続な多変数関数 $f:\mathbb{R}^n\to\mathbb{R},\ f(\boldsymbol{x})=f(x_1,\dots,x_n)$ に対して, 以下の極限

$$\lim_{h\to 0}\frac{f(x_1,\dots,x_{i-1},x_i+h,x_{i+1},\dots,x_n)-f(x_1,\dots,x_{i-1},x_i,x_{i+1},\dots,x_n)}{h}$$

が存在するとき, $f(\boldsymbol{x})$ は x_i で偏微分可能であるといい,

$$\frac{\partial f(\boldsymbol{x})}{\partial x_i},\quad \partial_{x_i}f(\boldsymbol{x}),\quad f_{x_i}(\boldsymbol{x})$$

などと書く[a].

[a] 例えば, 2 変数関数 $f(x,y)$ の x,y に関する偏微分をそれぞれ $\partial_x f(x,y),\partial_y f(x,y)$ とかく. なお, ∂ はパーシャルやラウンドなどと呼ぶ.

偏微分の定義は, 関数の引数が行列 $W=(w_{ij})\in\mathbb{R}^{m\times n}$ になっても同様に定義する. つまり, 興味のある x_{ij} のみで微分し, それを x_{ij} での偏微分と呼ぶことにするのである. そして, 偏微分可能な多変量関数 $f(\boldsymbol{x})$ $(\boldsymbol{x}\in\mathbb{R}^n)$ や $f(W)$ $(W\in\mathbb{R}^{m\times n})$ の微分は, ベクトルや行列の成分ごとの偏微分

$$\frac{\partial f(\boldsymbol{x})}{\partial\boldsymbol{x}}=\left(\frac{\partial f(\boldsymbol{x})}{\partial x_i}\right)\in\mathbb{R}^n,\qquad\frac{\partial f(W)}{\partial W}=\left(\frac{\partial f(W)}{\partial w_{ij}}\right)\in\mathbb{R}^{m\times n}$$

で定義し, **ベクトル微分**や**行列微分**とよぶ. 点 $\boldsymbol{a}$ におけるベクトル微分や A における行列微分も, 一変数関数の微分と同様に

$$\nabla f(\boldsymbol{a})=\left.\frac{\partial f(\boldsymbol{x})}{\partial\boldsymbol{x}}\right|_{\boldsymbol{x}=\boldsymbol{a}},\qquad\nabla f(A)=\left.\frac{\partial f(W)}{\partial W}\right|_{W=A}$$

などと書く. したがって, 一般に, 実数値関数をベクトル (や行列) で微分すると, そのサイズのベクトル (や行列) と同じサイズのベクトル (や行列) が得られる.

例 10. 冒頭の関数 $g(x,y,z) = x^2 + 2xy + z^2 - 1$ について,

$$\frac{\partial g(x,y,z)}{\partial x} = 2x + 2y, \quad \frac{\partial g(x,y,z)}{\partial y} = 2y, \quad \frac{\partial g(x,y,z)}{\partial z} = 2z$$

であったことから,

$$\frac{\partial g(x,y,z)}{\partial (x,y,z)} = \begin{pmatrix} 2x+2y \\ 2y \\ 2z \end{pmatrix} = 2\begin{pmatrix} x+y \\ y \\ z \end{pmatrix}$$

である.

例 11. 各 $\boldsymbol{x} \in \mathbb{R}^n$ に対して, $\mathbb{R}^n$ の内積

$$f(\boldsymbol{w}) = \boldsymbol{w}^\top \boldsymbol{x} = w_1 x_1 + \cdots + w_n x_n$$

は, $\boldsymbol{w} \in \mathbb{R}^n$ に関する多変数関数である. 各 w_i に対して,

$$\frac{\partial f(\boldsymbol{w})}{\partial w_i} = x_i$$

であるから,

$$\frac{\partial f(\boldsymbol{w})}{\partial \boldsymbol{w}} = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} = \boldsymbol{x}$$

となる.

例 12. 各 $\boldsymbol{x} \in \mathbb{R}^m$ および $\boldsymbol{y} \in \mathbb{R}^n$ に対して,

$$f(W) = \boldsymbol{x}^\top W \boldsymbol{y} = \sum_{i=1}^{m}\sum_{j=1}^{n} w_{ij} x_i y_j$$

を $W \in \mathbb{R}^{m\times n}$ の関数だとする. このとき, 各 w_{ij} に対して,

$$\frac{\partial f(W)}{\partial w_{ij}} = x_i y_j$$

であるから,

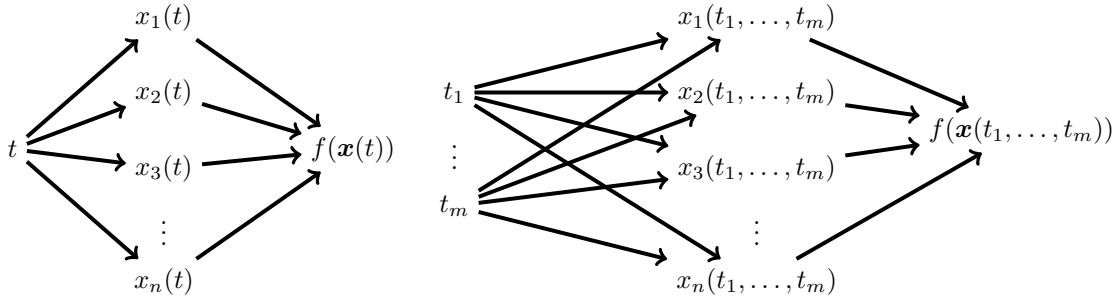
$$\frac{\partial f(W)}{\partial W} = \begin{pmatrix} x_1y_1 & \cdots & x_1y_n \\ \vdots & \ddots & \vdots \\ x_my_1 & \cdots & x_my_n \end{pmatrix} = \boldsymbol{x}\boldsymbol{y}^\top$$

となる.

2 変数関数 $f(x,y)$ について, さらに x, y が $t \in \mathbb{R}$ の関数, つまり $x = x(t), y = y(t)$ とかけているとしよう. つまり, f は $\mathbb{R} \ni t \mapsto x(t), y(t) \in \mathbb{R}$ と $\mathbb{R}^2 \ni (x,y) \mapsto f(x,y) \in \mathbb{R}$ の合成関数である. このとき, 関数 $f(x(t), y(t))$ の t に関する微分を考える. ただし, f は x および y に関して偏微分可能であるとし, $x(t), y(t)$ は t に関して微分可能[*39]であるとする. まず, 以下の等式が成り立つ.

$$\begin{aligned}
&\frac{f(x(t+h), y(t+h)) - f(x(t), y(t))}{h} \\
&= \frac{f(x(t+h), y(t+h)) - f(x(t), y(t+h))}{h} + \frac{f(x(t), y(t+h)) - f(x(t), y(t))}{h} \\
&= \frac{f(x(t+h), y(t+h)) - f(x(t), y(t+h))}{x(t+h) - x(t)} \frac{x(t+h) - x(t)}{h}
\end{aligned}$$

[*39] したがって, $x(t), y(t)$ は t に関して連続.



(a) 引数が一つだけの場合　　(b) 引数が複数ある場合

図 9　合成関数の微分のイメージ.

$$+\frac{f(x(t),y(t+h))-f(x(t),y(t))}{y(t+h)-y(t)}\frac{y(t+h)-y(t)}{h}$$

ここで, $x(t)$ は連続であるから, 最後の式の第 1 項は f の $x(t)$ での偏微分と, $x(t)$ の t での微分の積である. 同様に, 第 2 項は f の $y(t)$ での偏微分と, $y(t)$ の t での微分の積となる. したがって, $f(x(t),y(t))$ の t での微分は

$$\frac{\mathrm{d}f(x(t),y(t))}{\mathrm{d}t}=\frac{\partial f(x(t),y(t))}{\partial x(t)}\frac{\mathrm{d}x(t)}{\mathrm{d}t}+\frac{\partial f(x(t),y(t))}{\partial y(t)}\frac{\mathrm{d}y(t)}{\mathrm{d}t}$$

となる. $x(t)$ や $y(t)$ が t に依存することが明らかな場合や, 混乱の恐れがない場合には単に

$$\frac{\mathrm{d}f(x,y)}{\mathrm{d}t}=\frac{\partial f(x,y)}{\partial x}\frac{\mathrm{d}x}{\mathrm{d}t}+\frac{\partial f(x,y)}{\partial y}\frac{\mathrm{d}y}{\mathrm{d}t}$$

あるいは

$$\frac{\mathrm{d}f}{\mathrm{d}t}=\frac{\partial f}{\partial x}\frac{\mathrm{d}x}{\mathrm{d}t}+\frac{\partial f}{\partial y}\frac{\mathrm{d}y}{\mathrm{d}t}$$

などともかく. より一般に, $\mathbb{R}^n$ 上の実数値関数 $f(\boldsymbol{x})$ において $\boldsymbol{x}=\boldsymbol{x}(t)=(x_1(t),\ldots,x_n(t))$ のとき, f の t による微分は

$$\frac{\mathrm{d}f(\boldsymbol{x})}{\mathrm{d}t}=\sum_{i=1}^{n}\frac{\partial f(\boldsymbol{x})}{\partial x_i}\frac{\mathrm{d}x_i}{\mathrm{d}t}=\left(\frac{\mathrm{d}\boldsymbol{x}}{\mathrm{d}t}\right)^{\top}\frac{\partial f(\boldsymbol{x})}{\partial \boldsymbol{x}}$$

となる. ただし,

$$\frac{\mathrm{d}\boldsymbol{x}}{\mathrm{d}t}=\left(\frac{\mathrm{d}x_i}{\mathrm{d}t}\right)\in\mathbb{R}^n$$

とした. 同様に, 各 x_i が写像 $\mathbb{R}^m\ni(t_1,\ldots,t_m)\mapsto x_i(t_1,\ldots,t_m)\in\mathbb{R}$ で与えられる場合, 関数 $f(\boldsymbol{x}(t_1,\ldots,t_m))$ の変数 t_j での偏微分は

$$\frac{\partial f(\boldsymbol{x}(t_1,\ldots,t_m))}{\partial t_j}=\sum_{i=1}^{n}\frac{\partial f(\boldsymbol{x})}{\partial x_i}\frac{\partial x_i}{\partial t_j}=\left(\frac{\partial \boldsymbol{x}}{\partial t_j}\right)^{\top}\frac{\partial f(\boldsymbol{x})}{\partial \boldsymbol{x}}\tag{6}$$

となる. イメージとしては図 9 の通りである. つまり, 図 9(a) では, t から $f(\boldsymbol{x}(t))$ に伸びる矢線に関する微分を順にかける. このとき, 矢印の先にある関数は, 根元にある変数で微分する. そして, 変数 x_i の数だけ和を取る. このイメージは図 9(b) でも同様である[*40].

[*40] このイメージは深層学習のパラメータ推定でも重要なので, 図の意味を納得しておくと良い.

ところで, (6) はベクトル微分なので, $\boldsymbol{t} = (t_1, \ldots, t_m)^\top$ に関する偏微分をまとめることで,

$$\frac{\partial f(\boldsymbol{x}(\boldsymbol{t}))}{\partial \boldsymbol{t}} = \left(\left(\frac{\partial \boldsymbol{x}}{\partial t_j} \right)^\top \frac{\partial f(\boldsymbol{x})}{\partial \boldsymbol{x}} \right) = \begin{pmatrix} \left(\frac{\partial \boldsymbol{x}}{\partial t_1} \right)^\top \frac{\partial f(\boldsymbol{x})}{\partial \boldsymbol{x}} \\ \vdots \\ \left(\frac{\partial \boldsymbol{x}}{\partial t_m} \right)^\top \frac{\partial f(\boldsymbol{x})}{\partial \boldsymbol{x}} \end{pmatrix} = \left(\frac{\partial \boldsymbol{x}}{\partial t_1}, \ldots, \frac{\partial \boldsymbol{x}}{\partial t_m} \right)^\top \frac{\partial f(\boldsymbol{x})}{\partial \boldsymbol{x}} \in \mathbb{R}^m$$

と書き換えることができる. 最後の式に現れる行列

$$\frac{\partial \boldsymbol{x}}{\partial \boldsymbol{t}} = \left(\frac{\partial \boldsymbol{x}}{\partial t_1}, \ldots, \frac{\partial \boldsymbol{x}}{\partial t_m} \right) = \begin{pmatrix} \frac{\partial x_1(\boldsymbol{t})}{\partial t_1} & \cdots & \frac{\partial x_1(\boldsymbol{t})}{\partial t_m} \\ \vdots & \ddots & \vdots \\ \frac{\partial x_n(\boldsymbol{t})}{\partial t_1} & \cdots & \frac{\partial x_n(\boldsymbol{t})}{\partial t_m} \end{pmatrix} \in \mathbb{R}^{n \times m}$$

は**ヤコビ行列 (Jacobian matrix)** と呼ばれ, $\boldsymbol{x} : \mathbb{R}^m \to \mathbb{R}^n$ を多変数ベクトル値関数としてみたときに, 偏微分を並べた行列である. ヤコビ行列は, 変数変換した際の面積や体積の微小変化を符号付きで表すもので, 重積分の変数変換などに現れる重要な行列である.

3 深層学習入門 (工事中)

ここから, いよいよ深層学習のいくつかのトピックに触れる. はじめに, ニューラルネットワークの構成要素であるパーセプトロンについて説明する. その後, 近年では深層学習の基本的なツールになりつつある, 深層ニューラルネットワーク, オートエンコーダーや再帰型ニューラルネットワーク, 畳み込みニューラルネットワークのフレームワークについて説明する.

3.1 パーセプトロン

パーセプトロン (perceptron) とは, Rosenblatt によって 1958 年に提案された, 視覚と脳の機能をモデル化した数理モデルである. 具体的には, d 次元の入力 $\boldsymbol{x} \in \mathbb{R}^d$ と, 出力 $y \in \mathbb{R}$ が与えられたとき, パラメータ $w_0 \in \mathbb{R}$ および $\boldsymbol{w} = (w_1, \ldots, w_d)^\top \in \mathbb{R}^d$ を用いて

$$y = f(w_0 + \boldsymbol{w}^\top \boldsymbol{x}) = f(w_0 + w_1 x_1 + \cdots w_d x_d)$$

によって, 入力と出力の関係を記述する. ここで, w_0 は $\boldsymbol{x}$ に依存しないため**バイアス項** (あるいは**切片項**) と呼ばれる. また, f は既知の関数で**活性化関数**と呼ばれる既知の関数である. 入出力に関して n 組のデータ $(y_i, \boldsymbol{x}_i), i = 1, \ldots, n$ が得られたとき, 入出力間の誤差をパラメータ $w_0, \boldsymbol{w}$ の関数として,

$$E(w_0, \boldsymbol{w}) = \sum_{i=1}^{n} L(y_i, f(w_0 + \boldsymbol{w}^\top \boldsymbol{x}_i)) = \sum_{i=1}^{n} E_i(w_0, \boldsymbol{w})$$

を定義する. ここで, 標本ごとの誤差関数 $E_i(w_0, \boldsymbol{w}) = L(y_i, f(w_0 + \boldsymbol{w}^\top \boldsymbol{x}_i))$ は, モデルの出力 $f(w_0 + \boldsymbol{w}^\top \boldsymbol{x}_i)$ がどれほど出力 y_i に対して当てはまっているかを測る関数である. 例えば, $y \in \mathbb{R}$ ならば, 二乗損失

$$E_i(w_0, \boldsymbol{w}) = (y_i - f(w_0 + \boldsymbol{w}^\top \boldsymbol{x}_i))^2$$

がよく用いられる. また, $y \in \{0, 1\}$, つまり 0 か 1 のどちらかの値をとる場合には

$$E_i(w_0, , \boldsymbol{w}) = -y_i \log f(w_0 + \boldsymbol{w}^\top \boldsymbol{x}_i) - (1 - y_i) \log(1 - f(w_0 + \boldsymbol{w}^\top \boldsymbol{x}_i))$$

がしばしば用いられる. この損失は, 独立なベルヌーイ分布の負の対数尤度関数であり, その解釈は後で述べることにする. そして, 目標は, 標本全体を通して得られる誤差関数 $E(w_0, \boldsymbol{w})$ を最小にするようなパラメータ $w_0, \boldsymbol{w}$ を推定することである. $\tilde{\boldsymbol{w}} = (w, \boldsymbol{w}^\top)^\top, \tilde{\boldsymbol{x}}_i = (1, \boldsymbol{x}_i^\top)^\top \in \mathbb{R}^d$ とすれば,

$$w_0 + \boldsymbol{w}^\top \boldsymbol{x}_i = \tilde{\boldsymbol{w}}^\top \tilde{\boldsymbol{x}}_i$$

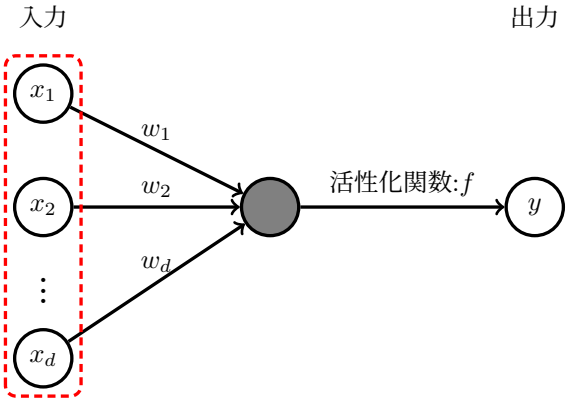


図 10　パーセプトロンのグラフ表現

書き換えることができるため, 以降では特に明記しない限り, $\boldsymbol{w}^\top \boldsymbol{x}_i$ と書いたら $w_0 + w_1 x_{i,1} + \cdots + w_d x_{i,d}$ を表すものとする.

ところで, パーセプトロンをグラフのように表すと, 図のようになる.

3.1.1　単純パーセプトロン

Rosenblatt による初期のパーセプトロンでは, 定義関数 $f(x) = \mathbf{1}\{x > 0\}$ が用いられた. なお, 定義関数 $\mathbf{1}\{A\}$ とは, 命題 A が真なら 1, 偽なら 0 を返す関数である. 特に, 出力が二値, つまり $y \in \{0, 1\}$ であって, 活性化関数を定義関数としたモデルは**単純パーセプトロン**と呼ばれている. ここで, 出力の意味は, 例えば $y = 1$ なら正解 (あるいは良品), $y = 0$ なら不正解 (あるいは不良品), のようにラベルを表している. 入出力に関して n 組のデータ $(y_i, \boldsymbol{x}_i), i = 1, \ldots, n$ が得られたとき, Rosenblatt は入出力間の誤差をパラメータ $\boldsymbol{w}$ の関数として,

$$E(\boldsymbol{w}) = \sum_{i=1}^{n} (y_i - \mathbf{1}\{\boldsymbol{w}^\top \boldsymbol{x}_i > 0\})^2$$

とし, この誤差を最小にするような $\boldsymbol{w}$ を推定する問題を考えた. 当然ながら, 誤差の小さなモデルを良いモデルと解釈する[*41]. そして, 今でいう**確率的勾配降下法**のプロトタイプによってパラメータが推定できることと, 適当な条件のもと, 有限回の反復で必ずアルゴリズムが収束することを示した. 標本ごとの誤差関数

$$E_i(\boldsymbol{w}) = (y_i - \mathbf{1}\{\boldsymbol{w}^\top \boldsymbol{x}_i > 0\})^2$$

の意味するところを考えよう. まず y_i と $\mathbf{1}\{\boldsymbol{w}^\top \boldsymbol{x}_i > 0\}$ はいずれも 0 または 1 の値しかとらない. したがって, 正解ラベルが $y_i = 1$ で $\boldsymbol{w}^\top \boldsymbol{x}_i > 0$ ならば $\mathbf{1}\{\boldsymbol{w}^\top \boldsymbol{x}_i > 0\} = 1$ なので, 標本 $\boldsymbol{x}_i$ を与えたときの損失は $E_i(\boldsymbol{w}) = 0$ となる. 一方で, $\boldsymbol{w}^\top \boldsymbol{x}_i \leq 0$ ならば $\mathbf{1}\{\boldsymbol{w}^\top \boldsymbol{x}_i > 0\} = 0$ であることから, $\boldsymbol{x}_i$ に関する損失は $E_i(\boldsymbol{w}) = 1$ となる.

誤差関数を最小にするために, 次のようなアルゴリズムを考える. つまり, パラメータの初期値を $\boldsymbol{w}^{(1)}$ とし,

$$\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} - \eta(\mathbf{1}\{\boldsymbol{w}^{(t)\top} \boldsymbol{x}_t > 0\} - y_t)\boldsymbol{x}_t \tag{7}$$

によって, パラメータを更新する. ここで, $(y_t, \boldsymbol{x}_t)$ は t ステップ目に参照する入力と正解ラベルであり, η はあらかじめ定められた定数である. ただし, $\boldsymbol{x}_t$ はバイアス項を含む $d + 1$ 次元のベクトルであることに注意する. (7) の意味するところは, t ステップ目のパラメータを用いて $\boldsymbol{x}_t$ の出力を予測したとき, 予測結果が正解ラベルと一致したらパラメータの値は変えないということである. 一方で, 予測結果が正解ラベルと

[*41] パラメータ $\boldsymbol{w}$ を推定するためのデータを訓練データとよぶ.

表 5　2 次元の入力を持つデータの例.

標本番号	入力 x_1	入力 x_2	正解ラベル y
1	0	0	0
2	1	0	0
3	0	1	0
4	1	1	1

異なる, つまり正解ラベルが 0 (または 1) のときに, 予測結果が 1(または 0) となる場合には, その結果に応じてパラメータを更新する. なお, パラメータの更新は一般に誤差が小さくなるまで繰り返す必要があるが, 単純パーセプトロンでは, 適当な条件のもとでは, 必ずアルゴリズムは有限回の反復で終了することが知られている[*42].

具体例を挙げて説明しよう. 例えば, 表 5 のように 2 次元の入力 $\boldsymbol{x} = (x_1, x_2)^\top$ と正解ラベル $y \in \{0, 1\}$ からなる 4 つのデータが与えられたとする. パラメータの初期値を $w_0 = 0, w_1 = 1, w_2 = -1$ とし, 学習率を $\eta = 0.5$ としておく. まず, 標本番号 1 のデータでは,

$$w_0 + w_1 x_1 + w_2 x_2 = 0 \to \mathbf{1}\{w_0 + w_1 x_1 + w_2 x_2 > 0\} = 0$$

となり, 正解ラベル $y = 0$ と一致するのでパラメータの更新は行わない. 次に, 標本番号 2 のデータでは,

$$w_0 + w_1 x_1 + w_2 x_2 = 1 \to \mathbf{1}\{w_0 + w_1 x_1 + w_2 x_2 > 0\} = 1$$

となり, 正解ラベル $y = 1$ と一致しないので, (7) を用いてパラメータの更新を行う. 更新後のパラメータは

$$\begin{pmatrix} w_0 \\ w_1 \\ w_2 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ -1 \end{pmatrix} - 0.5 \times (1 - 0) \times \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} -0.5 \\ 0.5 \\ -1 \end{pmatrix}$$

となる. 更新後のパラメータを用いて標本番号 3 のデータを評価すると

$$w_0 + w_1 x_1 + w_2 x_2 = -1.5 \to \mathbf{1}\{w_0 + w_1 x_1 + w_2 x_2 > 0\} = 0$$

となり, 正解ラベル $y = 0$ と一致するから, パラメータの更新は行わない. 標本番号 4 のデータでは,

$$w_0 + w_1 x_1 + w_2 x_2 = -1 \to \mathbf{1}\{w_0 + w_1 x_1 + w_2 x_2 > 0\} = 0$$

なので, 正解ラベル $y = 1$ と一致しないので, パラメータの更新を行う. 更新後のパラメータは

$$\begin{pmatrix} w_0 \\ w_1 \\ w_2 \end{pmatrix} = \begin{pmatrix} -0.5 \\ 0.5 \\ -1 \end{pmatrix} - 0.5 \times (0 - 1) \times \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ -0.5 \end{pmatrix}$$

となる. データを一通り見終わった後, データ全体を通した誤差を評価する. 現在のパラメータ $\boldsymbol{w} = (0, 1, -0.5)^\top$ を用いて誤差関数 $E(\boldsymbol{w})$ を計算すると,

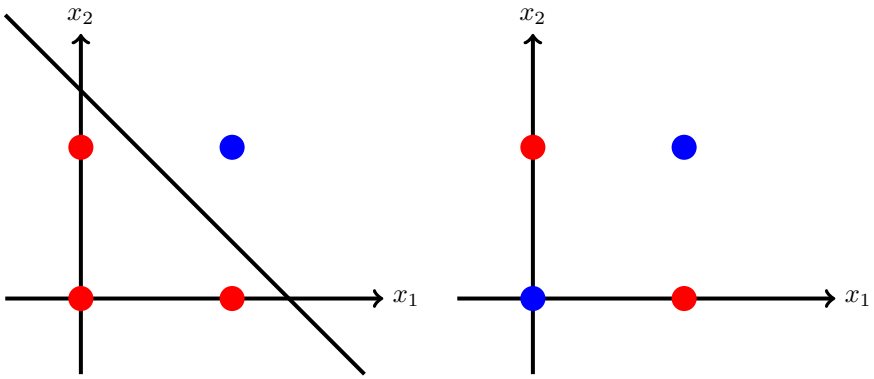
$$E(\boldsymbol{w}) = \sum_{i=1}^{n} (y_i - \mathbf{1}\{\boldsymbol{w}^\top \boldsymbol{x}_i > 0\})^2 = (0-0)^2 + (0-0)^2 + (0-1)^2 + (1-1)^2 = 1$$

となる. 一つの訓練データを一回りすることを**エポック**と呼び, **エポック数**とは何巡データを回して学習するかを表す数である. 今の場合, エポック数 1 では, 訓練誤差は 0 にならないが, 表 6 のように, 一つのデータを何巡かすると, 適当な条件のもとで単純パーセプトロンの訓練誤差は 0 になる. つまり, パラメータが $\boldsymbol{w} = (-0.5, 0.5, 0.5)^\top$ と更新されると, それ以降パラメータは更新されない.

[*42] つまり, 有限回の反復で訓練誤差が 0 になる.

表 6　表 5 のデータを用いた場合の, エポックごとのパラメータと訓練誤差の推移. 4 エポック終了時に訓練誤差が 0 となり, それ以降パラメータの更新は行われない.

エポック	標本番号とパラメータ $\boldsymbol{w} = (w_0, w_1, w_2)$				エポックごとの訓練誤差
	1	2	3	4	
1	$(0, 1, -1)$	$(-0.5, 0.5, -1)$	$(-0.5, 0.5, -1)$	$(0, 1, -0.5)$	1
2	$(0, 1, -0.5)$	$(-0.5, 0.5, -0.5)$	$(-0.5, 0.5, -0.5)$	$(0, 1, 0)$	1
3	$(0, 1, 0)$	$(-0.5, 0.5, 0)$	$(-0.5, 0.5, 0)$	$(0, 1, 0.5)$	2
4	$(0, 1, 0.5)$	$(-0.5, 0.5, 0.5)$	$(-0.5, 0.5, 0.5)$	$(-0.5, 0.5, 0.5)$	0
5	$(-0.5, 0.5, 0.5)$	$(-0.5, 0.5, 0.5)$	$(-0.5, 0.5, 0.5)$	$(-0.5, 0.5, 0.5)$	0



(a) 線形分離可能な例　　(b) 線形分離可能でない例

図 11　線形分離可能なデータとそうでないデータ. 図の赤い円と青い円は, 入力 $\boldsymbol{x} = (x_1, x_2)^\top$ の所属するクラスを表しており, 図 (a) の直線は線形分離可能な分離平面の例を表している. 図 (b) のデータは, いかなる分離平面を用いても平面で二つのクラスを完全に分離することはできない.

有限回の更新で単純パーセプトロンのアルゴリズムが停止するための十分条件を与える. 二つのクラス $\mathcal{C}_1, \mathcal{C}_2$ からなる訓練データ $\{(y_i, \boldsymbol{x}_i) \mid i = 1, \ldots, n\}$ が**線形分離可能**であるとは, あるパラメータ $\hat{\boldsymbol{w}}$ が存在して,

$$\boldsymbol{x}_i \in \mathcal{C}_1 \Rightarrow \hat{\boldsymbol{w}}^\top \boldsymbol{x}_i > 0$$
$$\boldsymbol{x}_i \in \mathcal{C}_2 \Rightarrow \hat{\boldsymbol{w}}^\top \boldsymbol{x}_i \leq 0$$

が成り立つことをいう. また, この $\hat{\boldsymbol{w}}$ でもって, 訓練データ $\{(y_i, \boldsymbol{x}_i) \mid i = 1, \ldots, n\}$ は完全分離可能であるという. 例えば, 図 11(a) は線形分離可能であり, 図 11(b) はいかなる平面をとっても二つのクラスを完全に分離することができず, 線形分離可能ではない.

線形分離可能なデータに対して, 有限回の反復で単純パーセプトロンのアルゴリズムが停止するということが知られている. 証明は web やニューラルネットワーク関連の書籍で調べることができるので, そちらを参照してほしい.

定理 1. 訓練データ $\{(y_i, \boldsymbol{x}_i) \in \{0, 1\} \times \mathbb{R}^d \mid i = 1, \ldots, n\}$ が線形分離可能ならば, 単純パーセプトロンは有限回の反復で訓練データを完全分離する.

3.1.2　パーセプトロン

前節では, Rosenblatt による, 活性化関数として定義関数を用いた単純パーセプトロンについて説明した. では一般の活性化関数では, どのようにパラメータを学習すれば良いだろうか. 以下, 活性化関数 f

および誤差関数 E は, パラメータ $\boldsymbol{w}$ に関して微分可能であるとする. 特に, 活性化関数として恒等写像 $f(x) = x$ を用いた場合,

$$y = f(\boldsymbol{w}^\top \boldsymbol{x}) = \boldsymbol{w}^\top \boldsymbol{x} = w_0 + w_1 x_1 + \cdots + w_d x_d$$

となるが, これは**線形回帰モデル**とも呼ばれる. また, 活性化関数としてシグモイド関数 $f(x) = 1/(1+e^{-x})$ を用いた場合

$$y = f(\boldsymbol{w}^\top \boldsymbol{x}) = \frac{1}{1 + e^{-\boldsymbol{w}^\top \boldsymbol{x}}}$$

は**ロジスティック回帰モデル**とも呼ばれる. 出力が $\{0, 1\}$ の二値である場合, ロジスティック回帰モデルは, ラベルの生起確率を表現するモデルであると解釈できる. つまり, 入力 $\boldsymbol{x}$ を与えたときのラベル $Y = 1$ が生起する確率

$$y = \mathrm{P}(Y = 1 \mid \boldsymbol{x}) = \frac{1}{1 + e^{-\boldsymbol{w}^\top \boldsymbol{x}}}$$

を表しており, $\hat{\boldsymbol{w}}^\top \boldsymbol{x} > 0$ ならば $\mathrm{P}(Y = 1 \mid \boldsymbol{x}) \geq 1/2$ なので, $Y = 1$ が生起しやすく, $\hat{\boldsymbol{w}}^\top \boldsymbol{x} > 0$ ならば $\mathrm{P}(Y = 1 \mid \boldsymbol{x}) \leq 1/2$ なので, $Y = 0$ が生起しやすいモデルとなっている. 表記を区別するために, 入力 $\boldsymbol{x}_i$ で評価したときのモデルの出力を

$$\pi_i = \frac{1}{1 + e^{-\boldsymbol{w}^\top \boldsymbol{x}_i}}$$

と表すことにしよう. このとき, 正解ラベル y_i が**ベルヌーイ分布** $\mathrm{Ber}(\pi_i)$ からの独立な標本[*43]だとすれば, $y_1, \ldots y_n$ の同時分布は

$$p(y_1, \ldots, y_n) = \prod_{i=1}^{n} \pi_i^{y_i} (1 - \pi_i)^{1 - y_i}$$

となり, したがって, その負の対数尤度関数は

$$\begin{aligned} -\log p(y_1, \ldots, y_n) &= -\sum_{i=1}^{n} \{y_i \log \pi_i + (1 - y_i) \log(1 - \pi_i)\} \\ &= -\sum_{i=1}^{n} \Big\{ y_i \log f(\boldsymbol{w}^\top \boldsymbol{x}_i) + (1 - y_i) \log(1 - f(\boldsymbol{w}^\top \boldsymbol{x}_i)) \Big\} \end{aligned}$$

となることがわかる. この値を $\boldsymbol{w}$ に関して最小化することで, $\boldsymbol{w}$ の最尤推定量が得られる. ところで, この誤差関数は初めに述べた, 二値分類問題に対する誤差関数と全く同じものであることに注意しておく[*44].

■確率的勾配降下法 パラメータ $\boldsymbol{w}$ を推定するためのアルゴリズムを考えよう. 解くべき問題は

$$\min_{\boldsymbol{w} \in \mathbb{R}^{d+1}} E(\boldsymbol{w}) = \sum_{i=1}^{n} E_i(\boldsymbol{w}) \tag{8}$$

であった. ただし, $E_i(\boldsymbol{w})$ は標本ごとの誤差関数で, 出力 y とモデルの出力 $f(\boldsymbol{w}^\top \boldsymbol{x})$ の乖離度を測るものである. 単純パーセプトロンのアルゴリズムと同様に, 標本ごとにパラメータを更新することを考える. ただし, 出力ラベルの偏りを緩和するため[*45], 各ステップで用いる標本はデータ $\{(y_i, \boldsymbol{x}_i) \mid i = 1, \ldots, n\}$ をランダムにシャッフルしておく[*46]. シャッフルしたデータの添字を (やや厳密性には欠けるが) 同じ添字 i で表せば, (8) を直接解く代わりに, 標本ごとに

$$\min_{\boldsymbol{w} \in \mathbb{R}^{d+1}} E_i(\boldsymbol{w}) \tag{9}$$

[*43] 確率変数 Y がベルヌーイ分布 $\mathrm{Ber}(\pi)$ に従うとは, $\mathrm{P}(Y = 1) = \pi$ & $\mathrm{P}(Y = 0) = 1 - \mathrm{P}(Y = 1) = 1 - \pi$ であることをいう.

[*44] 実際, ニューラルネットワークで用いられる誤差関数は, 適当な確率分布の対数尤度関数として表現できることが多い.

[*45] 例えば, 出力データが $(0, \ldots, 0, 1, \ldots, 1, 2, \ldots, 2)$ のように並んでいるときに, データを順番に読み込むと各ステップでの推定に偏りが生じてしまう.

[*46] 正確には, データの添字をランダムにシャッフルする. 例えば, $(1, 2, 3) \mapsto (2, 1, 3)$ とシャッフルした場合には, 各ステップで $(y_2, \boldsymbol{x}_2), (y_1, \boldsymbol{x}_1), (y_3, \boldsymbol{x}_3)$ を順番に用いる. また, このシャッフルはエポックごとに毎回行なう.

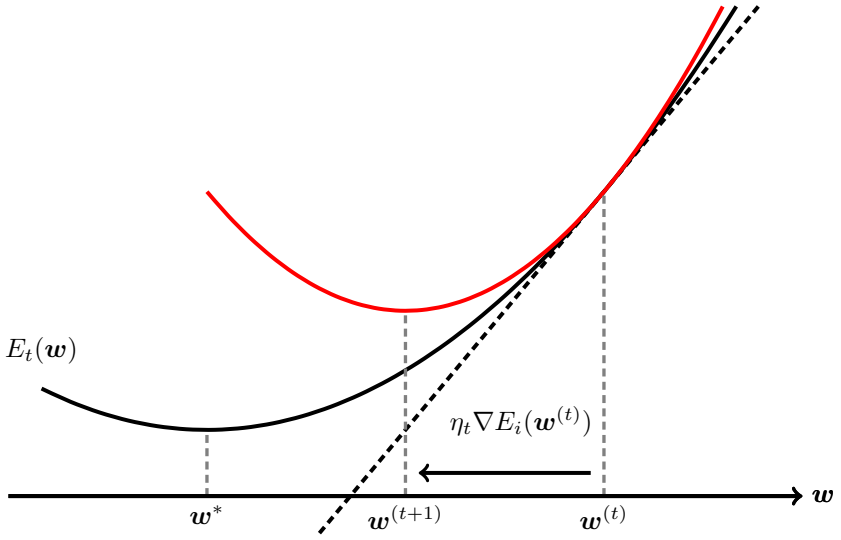


図 12　確率的勾配降下法のイメージ．黒の実線は, t ステップ目で訓練データ $\{(y_i, \boldsymbol{x}_i) \mid i = 1, \ldots, n\}$ からランダムにサンプリングしたデータ $(y_t, \boldsymbol{x}_t)$ に関する誤差関数 $E_t(\boldsymbol{w})$ であり, $\boldsymbol{w}^*$ はその最小解を表している．破線は点 $\boldsymbol{w}^{(t)}$ における誤差関数 $E_t(\boldsymbol{w})$ の接平面, 赤線は Lagrange 関数である．確率的勾配降下法では, $\boldsymbol{w}^{(t)}$ を用いて定義される Lagrange 関数の最小化点 $\boldsymbol{w}^{(t+1)}$ を更新されたパラメータの値とする．

を解くことによって, エポックごとに誤差関数を最適化する．一般に, (9) を解析に解くことは困難なので, ある $\tilde{\boldsymbol{w}}$ で線形近似した

$$\min_{\boldsymbol{w} \in \mathbb{R}^{d+1}} \nabla E_i(\tilde{\boldsymbol{w}})^\top (\boldsymbol{w} - \tilde{\boldsymbol{w}}) \tag{10}$$

の最小化を考えよう．ここで, $\nabla E_i(\tilde{\boldsymbol{w}})$ は, 誤差関数 $E_i(\boldsymbol{w})$ の $\tilde{\boldsymbol{w}}$ での微分係数であり, この目的関数は点 $\tilde{\boldsymbol{w}}$ における誤差関数 $E_i(\boldsymbol{w})$ の接平面を表している[*47]．ところで, (10) は $\boldsymbol{w}$ に関して線形であるから, ($\nabla E_i(\tilde{\boldsymbol{w}})$ の符号次第で) いくらでも小さくできる．そこで, 最適化する範囲に制限を設け, $\{\boldsymbol{w} \in \mathbb{R}^{d+1} \mid \|\boldsymbol{w} - \tilde{\boldsymbol{w}}\| \leq t\}$ での最小化問題

$$\min_{\boldsymbol{w} \in \mathbb{R}^{d+1}} \nabla E_i(\tilde{\boldsymbol{w}})^\top (\boldsymbol{w} - \tilde{\boldsymbol{w}}) \quad \text{s.t.} \quad \|\boldsymbol{w} - \tilde{\boldsymbol{w}}\|^2 \leq t^2 \tag{11}$$

を解くことにする[*48]．最小化問題 (11) の意味するところは,“平面 $\nabla E_i(\tilde{\boldsymbol{w}})^\top (\boldsymbol{w} - \tilde{\boldsymbol{w}})$ の最小値を中心 $\tilde{\boldsymbol{w}}$, 半径 t の球内で求める” ということだ．$\eta > 0$ に対して, **Lagrange 乗数**を $1/(2\eta)$ とする最適化問題 (11) の **Lagrange 関数**を

$$L(\boldsymbol{w}) = \nabla E_i(\tilde{\boldsymbol{w}})^\top (\boldsymbol{w} - \tilde{\boldsymbol{w}}) + \frac{1}{2\eta} \|\boldsymbol{w} - \tilde{\boldsymbol{w}}\|^2$$

とする[*49]．ここで, η は後に学習率と呼ばれる定数である．いま, $\tilde{\boldsymbol{w}}$ は固定した点なので, $L(\boldsymbol{w})$ は $\boldsymbol{w}$ に関する 2 次関数となる．したがって, 微分してゼロ点を求めることで $\boldsymbol{w}$ の最適解 $\hat{\boldsymbol{w}}$ を得ることができる．つまり,

$$\frac{\partial L(\boldsymbol{w})}{\partial \boldsymbol{w}} = \nabla E_i(\tilde{\boldsymbol{w}}) + \frac{1}{\eta}(\boldsymbol{w} - \tilde{\boldsymbol{w}})$$

[*47] つまり, 法線ベクトルが $\nabla E_i(\tilde{\boldsymbol{w}})$ であり, 点 $\tilde{\boldsymbol{w}}$ を通る平面のこと．
[*48] s.t. とは **subject to** の略である．**such that** の意味で用いる場合もあるので, その場合は文脈によって判断してほしい．
[*49] 第 2 項の $1/2$ は計算を簡単にするおまじない．

より,

$$\nabla E_i(\tilde{\boldsymbol{w}}) + \frac{1}{\eta}(\hat{\boldsymbol{w}} - \tilde{\boldsymbol{w}}) = \boldsymbol{0} \Rightarrow \hat{\boldsymbol{w}} = \tilde{\boldsymbol{w}} - \eta \nabla E_i(\tilde{\boldsymbol{w}})$$

が得られる. したがって, 逐次的に最適化を行う場合には, t ステップ目のパラメータの推定値と学習率をそれぞれ, $\boldsymbol{w}^{(t)}$ および η_t としたとき, $t+1$ ステップ目のパラメータを

$$\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} - \eta_t \nabla E_t(\boldsymbol{w}^{(t)}) \tag{12}$$

として更新すれば良い. データをランダムに並べ替えて勾配降下法を行うことから, このアルゴリズムは**確率的勾配降下法**と呼ばれる. 図 27 は確率的勾配降下法でパラメータが更新される様子を表したものである. $\boldsymbol{w}^{(t)}$ を $\nabla E_t(\boldsymbol{w}^{(t)})$ の方向に η_t 倍だけ移動したものが $\boldsymbol{w}^{(t+1)}$ となる.

注意 2. 活性化関数を恒等写像とした線形回帰モデル

$$y = \boldsymbol{w}^\top \boldsymbol{x}$$

で, 二乗損失を最小にしたい場合, わざわざ確率的勾配降下法によってパラメータを推定する必要はなく, 解析的にパラメータ $\boldsymbol{w}$ の推定値を得ることができる. つまり, 誤差関数

$$E(\boldsymbol{w}) = \sum_{i=1}^{n} (y_i - \boldsymbol{w}^\top \boldsymbol{x}_i)^2 = \|\boldsymbol{y} - X\boldsymbol{w}\|^2$$

をパラメータ $\boldsymbol{w}$ で微分して $\boldsymbol{0}$ とすれば,

$$-2X^\top(\boldsymbol{y} - X\boldsymbol{w}) = \boldsymbol{0} \Rightarrow X^\top X\boldsymbol{w} = X^\top \boldsymbol{y}$$

となる[a]. このとき, $X^\top X$ の一般化逆行列 $(X^\top X)^\dagger$ を用いれば, パラメータ $\boldsymbol{w}$ の推定値は

$$\hat{\boldsymbol{w}} = (X^\top X)^\dagger X^\top \boldsymbol{y}$$

で与えられる. また, ロジスティック回帰モデルの推定に関しても, 通常は確率的勾配降下法ではなく, **ニュートン法**や**スコア法**などの, 誤差関数の 2 階微分を用いた逐次推定のアルゴリズムがよく用いられる.

[a] この方程式は**正規方程式 (normal equation)** とも呼ばれる.

いくつか, アルゴリズムを実装する際の注意点について説明する. まず, 適切な学習率 η_t を用いることで, 確率的勾配降下法の学習はほとんど確実に局所解 (誤差関数が凸関数なら大域解) へ収束することが保証される[*50]. ここで述べたアルゴリズムを用いる場合には, 最適解への収束のための十分条件は

$$\sum_{t=1}^{\infty} \eta_t = \infty \quad \& \quad \sum_{t=1}^{\infty} \eta_t^2 < \infty$$

であることが知られている. そのため, $\eta_t = \eta_0/t$ のように, ステップ数に反比例するような学習率を用いると良い[*51]. ただし, 確率的勾配降下法を修正, 拡張した手法も数多く提案されており, ここで説明したアルゴリズム以外のものを用いる場合には, 適宜学習率を変更しなければならない[*52]. また, データが大量にある場合など, 1 つのエポック内で η_t が非常に小さくなることがある. すると, (12) による確率的勾配降下法の更新はほとんど行われなくなってしまうため, 通常はエポックごとに学習率を更新することが多い. 最

[*50] "**ほとんど確実に**" とは確率論で用いられる用語で, "**確率 1 で**" と同義.
[*51] $\sum_{t=1}^{\infty} t^{-1} = \infty, \sum_{t=1}^{\infty} t^{-2} = \pi^2/6 < \infty$ なので, 収束の十分条件を満たす.
[*52] 最近は色々な勾配降下法ベースのアルゴリズムが提案されていて, adam や RMSprop, ネステロフの加速法などもパーセプトロンや後で述べるニューラルネットワークの学習アルゴリズムとして用いられている.

Algorithm 1 確率的勾配降下法

入力: データ $\{(y_i, \boldsymbol{x}_i) \mid i = 1, \ldots, n\}$, 学習率 η_0, 活性化関数 f, 誤差関数 E, エポック数 T

パラメータ $\boldsymbol{w}$ の初期化
for $e = 1$ to T **do**
  $\mathcal{I} \leftarrow$ { ランダムシャッフル後のデータのインデックス }
  $\eta_t \leftarrow \eta_0 / e$ ▷ 学習率の更新
  for t in $\mathcal{I}$ **do**
    $\boldsymbol{w} \leftarrow \boldsymbol{w} - \eta_e \nabla E_t(\boldsymbol{w})$ ▷ パラメータの更新
  end for
end for

出力: パラメータ $\boldsymbol{w} \in \mathbb{R}^{d+1}$ の推定値

後に, 単純パーセプトロンでは, エポック終了時に改めてエポックごとの誤差を評価することを説明したが, やはりデータが大量にある場合には, 一つのエポックでパラメータを訓練データの数だけ更新した後に, 改めて訓練データをすべて用いて誤差評価を行うのは二度手間となる. そこで, 訓練誤差の評価を行う場合には, 毎回標本ごとの誤差を計算しておき, 最後にその誤差を平均することで, エポックごとの平均とする. 以上のことを踏まえて, Algorithm 1 のようにして確率的勾配法を実装する.

なお, Algorithm 1 では, エポック数を指定して, その数だけ t に関する for 文を繰り返すプログラムであるが, エポックごとの誤差が許容誤差を下回るまで繰り返すとしても良い. その場合,

- **for** $e = 1$ `to` **do**

に当たる部分を, (多少 Algorithm 1 を修正して)

- **while** `average`$(\mathcal{E}_e) >$ `tol` **do**

とする. ただし, tol は許容誤差を表しており, “エポックごとの誤差の平均が許容誤差よりも大きければ, while 以下の文を繰り返す” という意味である[*53].

3.2 深層ニューラルネットワーク

パーセプトロンとその学習アルゴリズムを理解することができれば, (言い過ぎかもしれないが) 深層学習の実装に関するほとんどすべてのことを学んだことになる. というのも, 畳み込みニューラルネットワーク以外のモデルは基本的に, パーセプトロンを組み合わせることで, 大体のことは表現できてしまうためだ. では, どこに違いがあるかというと, パーセプトロンのようにデータが入出力のペアで与えられている (**教師あり学習**) とか, 画像のみが与えられていて, そこから特徴を抽出する (**教師なし学習**), あるいは, テキストのように前後の文脈を考慮したデータ解析を行いたいなどの, データの質や解析目的によって, 異なるネットワークが構成されるのである. あるいは, ネットワークにおいて重要なパラメータ以外は重みを 0 に潰したい, つまり, スパースにパラメータを推定したい, というモチベーションがある場合は, 誤差関数そのものに手を加えることもある. とはいえ, どのようなモデルを作ったとしても, 基本的には確率勾配降下法によってパラメータを推定することができる.

この節では, まず簡単な **3 層ニューラルネットワーク**のモデルとパラメータの推定方法について説明する. 次に, ようやく深層学習のモデルとなる**深層ニューラルネットワーク**について述べる. その際, 層を深くする利点などについても触れる.

[*53] while 文は無限ループを生む可能性があるため, while 以下の文を繰り返す上限 (エポック数) も併用しておくと良い.

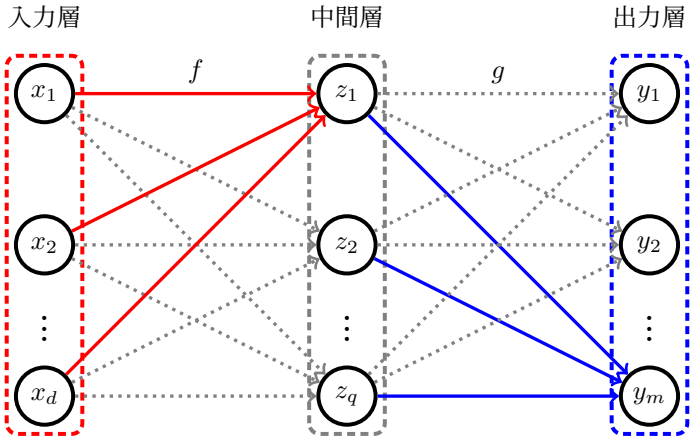


図 13　d 次元の入力から, q 次元の中間層を通して, m 次元の出力を返す 3 層ニューラルネットワークのグラフ表現. 各矢線には重みパラメータが割り当てられている. 赤線や青線など, 共通のユニットに伸びる矢線で表されるのットワークが一つのパーセプトロンを表現している. なお, f は入力層から出力層への活性化関数, g は中間層から出力層への活性化関数である.

3.2.1　3 層ニューラルネットワーク

3 層ニューラルネットワークは, ニューラルネットワークの基本的なモデルであり, 冒頭で述べたようにパーセプトロンをいくつも組み合わせたモデルである. 図 13 は 3 層ニューラルネットワークをグラフとして表現したものである. データは d-次元の入力 $\boldsymbol{x} = (x_1, \ldots, x_d)^\top$ と, m 次元の出力 $\boldsymbol{y} = (y_1, \ldots, y_m)^\top$ のペアであるとする.

まず, いくつか言葉の定義をしておこう. 複数のネットワークを介してモデルを表現することから, **層 (layer)** という言葉を用いてニューラルネットワークを説明できる. 図 13 では, 各層を破線で囲んでおり, この輪が一つの層を表している. **入力層**や**出力層**とは, 読んで字のごとく, 入力 $\boldsymbol{x}$ や出力に関する層であり, これまで入力, 出力と呼んでいたものに相当する. **中間層**とは入力層と出力層の間にある層のことであり, 実際には観測できない未知の変数からなる層である. また, 各変数は丸で囲んであるが, これをニューラルネットワークの言葉では**ユニット**とよぶ. したがって, 例えば, 図 13 のネットワークでは, 入力層と出力層はそれぞれ, d 個のユニットと m 個のユニットから構成されている. この d と m はデータが観測されると自動的に決定される. 中間層は q 個のユニットから構成されているが, ユニット数 q は, 通常, 解析者があらかじめ定める自然数である[*54]. ネットワークの各矢線には重み (パラメータ) が付加されており, (バイアス項を含む) 線形結合を**活性化関数**で写像することで, 次の層へとネットワークをつなげる. 図 13 では, 入力層から中間層への活性化関数は f であり, 中間層から出力層への活性化関数は g である. 特に混乱のない限り, “入力層から中間層”などと言わず, 単に活性化関数と呼んだり, 活性化関数 f などと呼ぶことにする.

グラフ表現を眺めていてもパラメータを推定できないので, 具体的に数式としてモデルを記述しよう. 入力層から中間層へのネットワークにおいて, z_j への矢線上のパラメータを $w_{j0}, w_{j1}, \ldots, w_{jd}$ とすれば, 活性化関数 f を用いて

$$z_j = f(w_{j0} + w_{j1}x_1 + \cdots + w_{jd}x_d) = f(\boldsymbol{w}_j^\top \boldsymbol{x}), \qquad j = 1, 2, \ldots, q$$

となる. ただし, パーセプトロンで述べたように, ここでも $\boldsymbol{w}_j = (w_{j0}, \ldots, w_{jd})^\top, \boldsymbol{x} = (1, x_1, \ldots, x_d)^\top \in \mathbb{R}^{d+1}$ のように, 切片項を含めて $\boldsymbol{w}_j$ や $\boldsymbol{x}$ を定義していることに注意する. また, 中間層についても, 特に断らない限り, $\boldsymbol{z} = (1, z_1, \ldots, z_q)^\top$ であるものとする. 同様に, 中間層から出力層へのネットワークにおい

[*54] 当然, 中間層のユニット数を客観的に決定する研究も存在する.

て, y_k への矢線上のパラメータを $v_{j0}, v_{j1}, \ldots, v_{jd}$ とすれば, 活性化関数 g を用いて

$$y_k = g(v_{k0} + v_{k1}z_1 + \cdots + v_{kq}z_q) = g(\boldsymbol{v}_j^\top \boldsymbol{z}), \qquad k = 1, 2, \ldots, m$$

となる. ここで, 表記を簡潔にするため, やや表記の乱用ではあるが,

$$f(W\boldsymbol{x}) = \begin{pmatrix} f(\boldsymbol{w}_1^\top \boldsymbol{x}) \\ \vdots \\ f(\boldsymbol{w}_q^\top \boldsymbol{x}) \end{pmatrix}$$

と表すことにする. ただし,

$$W = \begin{pmatrix} \boldsymbol{w}_1^\top \\ \vdots \\ \boldsymbol{w}_q^\top \end{pmatrix} = \begin{pmatrix} w_{10} & w_{11} & \cdots & w_{1d} \\ \vdots & & \ddots & \vdots \\ w_{q0} & w_{q1} & \cdots & w_{qd} \end{pmatrix} \in \mathbb{R}^{q\times(d+1)}$$

はパラメータを並べた行列であり, 1 列目はモデルの切片項に対応するパラメータである. こうすることで, 入力層から中間層へのネットワークが簡単に

$$\boldsymbol{z} = \begin{pmatrix} 1 \\ z_1 \\ \vdots \\ z_q \end{pmatrix} = \begin{pmatrix} 1 \\ f(W\boldsymbol{x}) \end{pmatrix}$$

とかける. 同様に, 中間層から出力層に関しても

$$\boldsymbol{y} = \begin{pmatrix} y_1 \\ \vdots \\ y_m \end{pmatrix} = g(V\boldsymbol{z})$$

となる. ただし,

$$V = \begin{pmatrix} \boldsymbol{v}_1^\top \\ \vdots \\ \boldsymbol{v}_m^\top \end{pmatrix} = \begin{pmatrix} v_{10} & v_{11} & \cdots & v_{1q} \\ \vdots & & \ddots & \vdots \\ v_{m0} & v_{m1} & \cdots & v_{mq} \end{pmatrix} \in \mathbb{R}^{m\times(q+1)}$$

は中間層から出力層へのパラメータを並べた行列である. パーセプトロンとは異なり, ニューラルネットワークの出力は多次元になりうる. そのため, 中間層から出力層の活性化関数 g は, パーセプトロンの場合とは異なるものが用いられる. 特に, 多クラス分類問題を考える場合, 活性化関数 g : として, **ソフトマックス関数**

$$g(V\boldsymbol{z}) = \frac{1}{\sum_{i=1}^m e^{\boldsymbol{v}_i^\top \boldsymbol{z}}} \begin{pmatrix} e^{\boldsymbol{v}_1^\top \boldsymbol{z}} \\ \vdots \\ e^{\boldsymbol{v}_m^\top \boldsymbol{z}} \end{pmatrix} \in \mathbb{R}^m$$

が用いられることが多い. シグモイド関数と同様に, ソフトマックス関数の成分の和も 1 になるため, その第 j 成分は入力が j 番目のクラスに属する確率 (所属確率) を表すものと解釈できる. 同様に, 多クラス分類問題の場合, 誤差関数として**クロスエントロピー**

$$-\sum_{i=1}^m y_i \log g(\boldsymbol{v}_i^\top \boldsymbol{z})$$

が用いられる. ただし, $\boldsymbol{y} = (y_1, \ldots, y_m)^\top$ は i 番目の標本がクラス j にぞくすれば $y_j = 1$, それ以外で 0 であるようなベクトルである[*55]. 例えば, 4 クラス分類問題で, クラスが 0, 1, 2, 3 のとき, クラス 2 に対応

[*55] 1-of-k 表現や one hot 表現と呼ばれる.

する $\boldsymbol{y}$ は $\boldsymbol{y} = (0,0,1,0)^\top$ となる．$\boldsymbol{y}$ に関するこの表記は分類問題特有のものであり, 回帰問題の場合は単純に m 個の出力を持つベクトルとして扱うが, $m = 2$, つまり, 2 値分類問題におけるパーセプトロンの誤差を含む一般化となっていることは簡単に確認することができる．出力が多次元であるようなデータを扱う問題は, **マルチタスク学習**としても知られている．

なお, このニューラルネットワークモデルを中間層のパラメータ $\boldsymbol{z}$ を使わずに表現すると,

$$\boldsymbol{y} = g(V(1, f(W\boldsymbol{x})^\top)^\top) = (g \circ V \circ \tilde{f} \circ W)(\boldsymbol{x})$$

のように, W や V を対応する線形写像と思うことで, 合成関数として書くこともできる．ここで, 関数 $\tilde{f}$ とは, q 次元のベクトルを $q+1$ 次元のベクトルに移す関数 $\tilde{f} : \mathbb{R}^q \to \mathbb{R}^{q+1}$ であり,

$$\tilde{f}(\boldsymbol{x}) = \begin{pmatrix} 1 \\ f(\boldsymbol{x}) \end{pmatrix}$$

とする．

3 層ニューラルネットワークの具体的なモデルを一つ例示しておく．

例 13. 活性化関数はいずれも恒等写像 $f(x) = x, g(x) = x$ であるとし, 切片項はない, つまり, $w_{j0} = v_{k0} = 0$ とする．したがって, 通常のベクトルの作法に従って, $\boldsymbol{x} = (x_1, \ldots, x_d)^\top, \boldsymbol{z} = (z_1, \ldots, z_q)^\top$ を考える[*56]．このとき,

$$\boldsymbol{z} = f(W\boldsymbol{x}) = \begin{pmatrix} f(\boldsymbol{w}_1^\top \boldsymbol{x}) \\ \vdots \\ f(\boldsymbol{w}_q^\top \boldsymbol{x}) \end{pmatrix} = \begin{pmatrix} \boldsymbol{w}_1^\top \boldsymbol{x} \\ \vdots \\ \boldsymbol{w}_q^\top \boldsymbol{x} \end{pmatrix} = W\boldsymbol{x}$$

から,

$$\boldsymbol{y} = g(V\boldsymbol{z}) = \begin{pmatrix} g(\boldsymbol{v}_1^\top \boldsymbol{z}) \\ \vdots \\ g(\boldsymbol{v}_m^\top \boldsymbol{z}) \end{pmatrix} = \begin{pmatrix} \boldsymbol{v}_1^\top \boldsymbol{z} \\ \vdots \\ \boldsymbol{v}_q^\top \boldsymbol{z} \end{pmatrix} = V\boldsymbol{z} = VW\boldsymbol{x}$$

となる．$q < \min\{d, m\}$ であるとき, このモデルは**縮小ランク回帰モデル**と呼ばれる．縮小ランク回帰モデルでは, d 次元の入力から m 次元の出力を返す線形回帰モデルを

$$\boldsymbol{y} = B\boldsymbol{x}$$

としたとき, 未知の回帰係数行列 $B \in \mathbb{R}^{m \times d}$ が低ランクであると考え, B を “縦長”の行列 $V \in \mathbb{R}^{m \times q}$ と “横長”の行列 $W \in \mathbb{R}^{q \times d}$ を用いて $B \approx VW$ と近似する．これによって, (B のランク q を選択する問題は残るが) 実質的に最適化すべきパラメータ数を減らすことができる．

■誤差逆伝播法　ニューラルネットワークにおけるパラメータ推定のアルゴリズムは**誤差逆伝播法**と呼ばれる．ここでは, 誤差が逆伝播するとはどういうことかも含めて, パラメータ推定のためのアルゴリズムを説明する．なお, 誤差逆伝播のイメージをつかむためには, 次節の深層ニューラルネットワークのアルゴリズムを理解できれば自然に納得すると思われる．

アルゴリズムの基本は, パーセプトロンの節で述べた確率的勾配降下法である．3 層ニューラルネットワークのモデルにおいて, 推定すべきパラメータは $W \in \mathbb{R}^{q \times (d+1)}$ と $V \in \mathbb{R}^{m \times (q+1)}$ であった．したがって, 誤差関数はパラメータ W と V の関数 $E(W, V)$ である．誤差関数としては, 回帰問題であれば二乗損失

$$E(W, V) = \frac{1}{2}\|\boldsymbol{y} - g(V\boldsymbol{z})\|^2 = \frac{1}{2}\sum_{i=1}^{m}(y_i - g(\boldsymbol{v}_i^\top \boldsymbol{z}))^2$$

が用いられる．ただし, この場合, 活性化関数 g を恒等写像

$$g(\boldsymbol{v}_i^\top \boldsymbol{z}) = \boldsymbol{v}_i^\top \boldsymbol{z} = v_{i0} + v_{i1}z_1 + \cdots + v_{iq}z_q$$

[*56] そのため, $W \in \mathbb{R}^{q \times d}, V \in \mathbb{R}^{m \times q}$ である．

とする．また, 先に述べたように, 分類問題であれば, 誤差関数はクロスエントロピー

$$E(W,V) = -\sum_{i=1}^{m} y_i \log g(\boldsymbol{v}_i^\top \boldsymbol{z})$$

とし, 活性化関数 g はソフトマックス関数である．このように, 問題に応じて誤差関数と中間層から出力層への活性化関数を指定するメリットとして, 誤差関数を中間層から出力層へのパラメータ V で微分した際に, その微分が簡潔に書けることによる．もちろん, 活性化関数 g をこのように定めないといけないというルールはないので, 問題に応じて適宜選んで良いが, その場合, 誤差関数の微分が多少煩雑になる．なお, 入力層から中間層への活性化関数 f はパーセプトロンの場合と同様に, シグモイド関数などが用いられる他,

- Rectified Linear Unit (ReLU): $f(x) = \max\{0, x\}$
- ハイパボリックタンジェント: $f(x) = \tanh x = (e^x - e^{-x})/(e^x + e^{-x})$

などがよく用いられる．特に, ReLU は最近のニューラルネットワークに欠かせないものとなっている．

誤差関数をパラメータ V で微分しよう．まず, 回帰問題を考え, 活性化関数 g は恒等写像であるとする．$g(\boldsymbol{v}_i^\top \boldsymbol{z})$ を $\boldsymbol{v}_j$ で偏微分すると,

$$\frac{\partial g(\boldsymbol{v}_i^\top \boldsymbol{z})}{\partial \boldsymbol{v}_j} = \begin{cases} \boldsymbol{0}, & i \neq j \\ \boldsymbol{z}, & i = j \end{cases}$$

であるから, 誤差関数を $\boldsymbol{v}_j$ で偏微分すると,

$$\begin{aligned} \frac{\partial E(W,V)}{\partial \boldsymbol{v}_j} &= \frac{1}{2}\sum_{i=1}^{m} \frac{\partial}{\partial \boldsymbol{v}_j}(y_i - g(\boldsymbol{v}_i^\top \boldsymbol{z}))^2 = \frac{1}{2}\frac{\partial}{\partial \boldsymbol{v}_j}(y_j - g(\boldsymbol{v}_j^\top \boldsymbol{z}))^2 \\ &= -(y_j - g(\boldsymbol{v}_j^\top \boldsymbol{z}))\frac{\partial g(\boldsymbol{v}_j^\top \boldsymbol{z})}{\partial \boldsymbol{v}_j} = (g(\boldsymbol{v}_j^\top \boldsymbol{z}) - y_j)\boldsymbol{z} \end{aligned}$$

となることがわかる．したがって, $V = (\boldsymbol{v}_1, \ldots, \boldsymbol{v}_m)^\top$ だったことを思い出すと, 誤差関数の V での微分は

$$\frac{\partial E(W,V)}{\partial V} = \left(\frac{\partial E(W,V)}{\partial \boldsymbol{v}_1}, \ldots, \frac{\partial E(W,V)}{\partial \boldsymbol{v}_m}\right)^\top = \begin{pmatrix} (g(\boldsymbol{v}_1^\top \boldsymbol{z}) - y_1)\boldsymbol{z}^\top \\ \vdots \\ (g(\boldsymbol{v}_m^\top \boldsymbol{z}) - y_m)\boldsymbol{z}^\top \end{pmatrix} = (g(V\boldsymbol{z}) - \boldsymbol{y})\boldsymbol{z}^\top$$

となる．一方, 分類問題の場合には,

$$g(\boldsymbol{v}_i^\top \boldsymbol{z}) = \frac{1}{\sum_{k=1}^{m} e^{\boldsymbol{v}_k^\top \boldsymbol{z}}} e^{\boldsymbol{v}_i^\top \boldsymbol{z}}$$

であるから, 合成関数の微分を用いて

$$\frac{\partial e^{\boldsymbol{v}_i^\top \boldsymbol{z}}}{\partial \boldsymbol{v}_j} = \frac{\partial e^{\boldsymbol{v}_i^\top \boldsymbol{z}}}{\partial \boldsymbol{v}_j^\top \boldsymbol{z}} \frac{\partial \boldsymbol{v}_i^\top \boldsymbol{z}}{\partial \boldsymbol{v}_j} = \begin{cases} \boldsymbol{0}, & i \neq j \\ e^{\boldsymbol{v}_j^\top \boldsymbol{z}}\boldsymbol{z}, & i = j \end{cases}$$

に注意して, $i = j$ ならば

$$\begin{aligned} \frac{\partial g(\boldsymbol{v}_j^\top \boldsymbol{z})}{\partial \boldsymbol{v}_j} &= \frac{1}{(\sum_{k=1}^{m} e^{\boldsymbol{v}_k^\top \boldsymbol{z}})^2}\left(\frac{\partial e^{\boldsymbol{v}_j^\top \boldsymbol{z}}}{\partial \boldsymbol{v}_j}\sum_{k=1}^{m} e^{\boldsymbol{v}_k^\top \boldsymbol{z}} - e^{\boldsymbol{v}_j^\top \boldsymbol{z}}\sum_{k=1}^{m}\frac{\partial e^{\boldsymbol{v}_k^\top \boldsymbol{z}}}{\partial \boldsymbol{v}_j}\right) \\ &= \frac{1}{(\sum_{k=1}^{m} e^{\boldsymbol{v}_k^\top \boldsymbol{z}})^2}\left(e^{\boldsymbol{v}_j^\top \boldsymbol{z}}\boldsymbol{z}\sum_{k=1}^{m} e^{\boldsymbol{v}_k^\top \boldsymbol{z}} - e^{\boldsymbol{v}_j^\top \boldsymbol{z}}e^{\boldsymbol{v}_j^\top \boldsymbol{z}}\boldsymbol{z}\right) \\ &= \left(1 - \frac{e^{\boldsymbol{v}_j^\top \boldsymbol{z}}}{\sum_{k=1}^{m} e^{\boldsymbol{v}_k^\top \boldsymbol{z}}}\right)\frac{e^{\boldsymbol{v}_j^\top \boldsymbol{z}}}{\sum_{k=1}^{m} e^{\boldsymbol{v}_k^\top \boldsymbol{z}}}\boldsymbol{z} = (1 - g(\boldsymbol{v}_j^\top \boldsymbol{z}))g(\boldsymbol{v}_j^\top \boldsymbol{z})\boldsymbol{z} \end{aligned}$$

となる．同様に, $i \neq j$ ならば

$$\frac{\partial g(\boldsymbol{v}_i^\top \boldsymbol{z})}{\partial \boldsymbol{v}_j} = \frac{1}{(\sum_{k=1}^{m} e^{\boldsymbol{v}_k^\top \boldsymbol{z}})^2}\left(\frac{\partial e^{\boldsymbol{v}_i^\top \boldsymbol{z}}}{\partial \boldsymbol{v}_j}\sum_{k=1}^{m} e^{\boldsymbol{v}_k^\top \boldsymbol{z}} - e^{\boldsymbol{v}_i^\top \boldsymbol{z}}\sum_{k=1}^{m}\frac{\partial e^{\boldsymbol{v}_k^\top \boldsymbol{z}}}{\partial \boldsymbol{v}_j}\right)$$

$$= \frac{1}{(\sum_{k=1}^{m} e^{\boldsymbol{v}_k^\top \boldsymbol{z}})^2} \left(-e^{\boldsymbol{v}_i^\top \boldsymbol{z}} e^{\boldsymbol{v}_j^\top \boldsymbol{z}} \boldsymbol{z} \right) = -g(\boldsymbol{v}_i^\top \boldsymbol{z}) g(\boldsymbol{v}_j^\top \boldsymbol{z})$$

となる．したがって, 誤差関数を $\boldsymbol{v}_j$ で偏微分すれば,

$$\begin{aligned}
\frac{\partial E(W,V)}{\partial \boldsymbol{v}_j} &= -\sum_{i=1}^{m} y_i \frac{\partial}{\partial \boldsymbol{v}_j} \log g(\boldsymbol{v}_i^\top \boldsymbol{z}) = -\sum_{i=1}^{m} y_i \frac{1}{g(\boldsymbol{v}_i^\top \boldsymbol{z})} \frac{\partial g(\boldsymbol{v}_i^\top \boldsymbol{z})}{\partial \boldsymbol{v}_j} \\
&= -\sum_{i \neq j} y_i \frac{1}{g(\boldsymbol{v}_i^\top \boldsymbol{z})} (-g(\boldsymbol{v}_i^\top \boldsymbol{z}) g(\boldsymbol{v}_j^\top \boldsymbol{z}) \boldsymbol{z}) - y_j \frac{1}{g(\boldsymbol{v}_j^\top \boldsymbol{z})} (1 - g(\boldsymbol{v}_j^\top \boldsymbol{z})) g(\boldsymbol{v}_j^\top \boldsymbol{z}) \boldsymbol{z} \\
&= \left\{ g(\boldsymbol{v}_j^\top \boldsymbol{z}) \sum_{i \neq j} y_i - y_j (1 - g(\boldsymbol{v}_j^\top \boldsymbol{z})) \right\} \boldsymbol{z} = (g(\boldsymbol{v}_j^\top \boldsymbol{z}) - y_j) \boldsymbol{z}
\end{aligned}$$

が得られる．ただし, 最後の等号で $\sum_{i=1}^{m} y_i = 1$ となることを用いた．以上より, 誤差関数を V で微分したものは回帰問題の場合と同じものになる．

注意 3. 念のため, 一般の活性化関数 g についても計算しておこう．以下, 簡単のため,

$$\frac{\partial g(\boldsymbol{v}_i^\top \boldsymbol{z})}{\partial \boldsymbol{v}_j} = \frac{\partial g(\boldsymbol{v}_i^\top \boldsymbol{z})}{\partial \boldsymbol{v}_j^\top \boldsymbol{z}} \frac{\partial \boldsymbol{v}_j^\top \boldsymbol{z}}{\partial \boldsymbol{v}_i} = \partial_j g(\boldsymbol{v}_i^\top \boldsymbol{z}) \boldsymbol{z} \in \mathbb{R}^q$$

と書くことにする[*57]．なお, $\partial_j g(\boldsymbol{v}_i^\top \boldsymbol{z})$ はスカラーである．また,

$$\partial_j g(V\boldsymbol{z}) = \begin{pmatrix} \partial_j g(\boldsymbol{v}_1^\top \boldsymbol{z}) \\ \vdots \\ \partial_j g(\boldsymbol{v}_m^\top \boldsymbol{z}) \end{pmatrix} \in \mathbb{R}^m, \quad \boldsymbol{\partial} g(V\boldsymbol{z}) = (\partial_1 g(V\boldsymbol{z}), \ldots, \partial_m g(V\boldsymbol{z})) \in \mathbb{R}^{m \times m}$$

としておく．このとき, 二乗損失を用いた場合は

$$\begin{aligned}
\frac{E(W,V)}{\partial \boldsymbol{v}_j} &= \frac{1}{2} \sum_{i=1}^{m} \frac{\partial}{\partial \boldsymbol{v}_j} (y_i - g(\boldsymbol{v}_i^\top \boldsymbol{z}))^2 = -\sum_{i=1}^{m} (y_i - g(\boldsymbol{v}_i^\top \boldsymbol{z})) \frac{\partial g(\boldsymbol{v}_i^\top \boldsymbol{z})}{\partial \boldsymbol{v}_j} \\
&= \sum_{i=1}^{m} (g(\boldsymbol{v}_i^\top \boldsymbol{z}) - y_i) \partial_j g(\boldsymbol{v}_i^\top \boldsymbol{z}) \boldsymbol{z} = (g(V\boldsymbol{z}) - \boldsymbol{y})^\top \partial_j g(V\boldsymbol{z}) \boldsymbol{z}
\end{aligned}$$

より,

$$\begin{aligned}
\frac{\partial E(W,V)}{\partial V} &= \left(\frac{\partial E(W,V)}{\partial \boldsymbol{v}_1}, \ldots, \frac{\partial E(W,V)}{\partial \boldsymbol{v}_m} \right)^\top \\
&= \begin{pmatrix} \partial_1 g(V\boldsymbol{z})^\top (g(V\boldsymbol{z}) - \boldsymbol{y}) \boldsymbol{z}^\top \\ \vdots \\ \partial_m g(V\boldsymbol{z})^\top (g(V\boldsymbol{z}) - \boldsymbol{y}) \boldsymbol{z}^\top \end{pmatrix} = \boldsymbol{\partial} g(V\boldsymbol{z})^\top (g(V\boldsymbol{z}) - \boldsymbol{y}) \boldsymbol{z}^\top
\end{aligned}$$

となる．なお, g が恒等写像の場合には, $\boldsymbol{\partial} g(V\boldsymbol{z}) = I_m$ となって, 先の結果と一致する[*58]．次に,

$$\frac{\partial}{\partial \boldsymbol{v}_j} \log g(\boldsymbol{v}_i^\top \boldsymbol{z}) = \partial_j \log g(\boldsymbol{v}_i^\top) \boldsymbol{z}$$

および

$$\partial_j \log g(V\boldsymbol{z}) = \begin{pmatrix} \partial_j \log g(\boldsymbol{v}_1^\top \boldsymbol{z}) \\ \vdots \\ \partial_j \log g(\boldsymbol{v}_m^\top \boldsymbol{z}) \end{pmatrix} \in \mathbb{R}^m,$$

[*57] 文脈によるが, “簡単のため” とはテクニカルタームで, “本質は失わないけど話を簡単にするために” という意味．今の場合, 表記を簡略化するために, という意味で記号を再定義しただけ．多項式に関するものだと, “簡単のため, 最高次の係数を 1 とする” という表現があるが, これは, 多項式の全ての係数を最大次数の係数 (もちろん非ゼロ) で割って, それを改めて係数として定義し直せば問題の本質は損なわれない, という意味である．例えば, $2x^2 - 1 = 0$ を x について解こうが, $x^2 - 1/2 = 0$ を x について解こうが, 答えに違いはない．

[*58] I_m は m 次元単位行列．

$$\boldsymbol{\partial} \log g(V\boldsymbol{z}) = (\partial_1 \log g(V\boldsymbol{z}), \ldots, \partial_m \log g(V\boldsymbol{z})) \in \mathbb{R}^{m\times m}$$

を定義する. このとき, 誤差関数としてクロスエントロピーを用いると,

$$\frac{\partial E(W,V)}{\partial \boldsymbol{v}_j} = -\sum_{i=1}^{m} y_i \partial_j \log g(\boldsymbol{v}_i^\top \boldsymbol{z})\boldsymbol{z} = -\boldsymbol{y}^\top \partial_j \log g(V\boldsymbol{z})\boldsymbol{z}$$

となり, 二乗損失の場合と同じ計算をすれば,

$$\frac{\partial E(W,V)}{\partial V} = \boldsymbol{\partial} \log g(V\boldsymbol{z})^\top \boldsymbol{y}\boldsymbol{z}^\top$$

となり, いずれにしろ, 誤差関数の微分はやや煩雑なものとなる. (注意終)

話を戻して, 以下では, 誤差関数が二乗損失なら g は恒等写像, クロスエントロピーなら g はソフトマックス関数であるとしよう. つまり,

$$\frac{\partial E(W,V)}{\partial V} = (g(V\boldsymbol{z}) - \boldsymbol{y})\boldsymbol{z}^\top$$

であり, 特に,

$$\frac{\partial E(W,V)}{\partial \boldsymbol{v}_i^\top \boldsymbol{z}} = g(\boldsymbol{v}_i^\top \boldsymbol{z}) - y_i$$

が成立している. 誤差関数の $W = (\boldsymbol{w}_1, \ldots, \boldsymbol{w}_q)^\top$ での微分を計算しよう. まず, $\boldsymbol{z} = (1, f(W\boldsymbol{x})^\top)^\top$ であったことを思い出す. 合成関数の微分を用いて, 誤差関数を $\boldsymbol{w}_j$ で偏微分すれば,

$$\frac{\partial E(W,V)}{\partial \boldsymbol{w}_j} = \sum_{i=1}^{m} \frac{\partial E(W,V)}{\partial \boldsymbol{v}_i^\top \boldsymbol{z}} \frac{\partial \boldsymbol{v}_i^\top \boldsymbol{z}}{\partial \boldsymbol{w}_j} = \sum_{i=1}^{m} (g(\boldsymbol{v}_i^\top \boldsymbol{z}) - y_i) \frac{\partial \boldsymbol{v}_i^\top \boldsymbol{z}}{\partial \boldsymbol{w}_j}$$

となる.

$$\boldsymbol{v}_i^\top \boldsymbol{z} = v_{i0} + v_{i1} z_1 + \cdots + v_{iq} z_q = v_{i0} + v_{i1} f(\boldsymbol{w}_1^\top \boldsymbol{x}) + \cdots + v_{iq} f(\boldsymbol{w}_q^\top \boldsymbol{x})$$

であるから,

$$\frac{\partial \boldsymbol{v}_i^\top \boldsymbol{z}}{\partial \boldsymbol{w}_j} = v_{ij} \frac{\partial f(\boldsymbol{w}_j^\top \boldsymbol{x})}{\partial \boldsymbol{w}_j} = v_{ij} \nabla f(\boldsymbol{w}_j^\top \boldsymbol{x})\boldsymbol{x}$$

に注意すれば結局[*59],

$$\begin{aligned}
\frac{\partial E(W,V)}{\partial \boldsymbol{w}_j} &= \sum_{i=1}^{m} (g(\boldsymbol{v}_i^\top \boldsymbol{z}) - y_i) v_{ij} \nabla f(\boldsymbol{w}_j^\top \boldsymbol{x})\boldsymbol{x} \\
&= (g(\boldsymbol{v}_1^\top \boldsymbol{z}) - y_1), \ldots, g(\boldsymbol{v}_m^\top \boldsymbol{z}) - y_m)) \begin{pmatrix} v_{1j} \\ \vdots \\ v_{mj} \end{pmatrix} \nabla f(\boldsymbol{w}_j^\top \boldsymbol{x})\boldsymbol{x} \\
&= (g(V\boldsymbol{z}) - \boldsymbol{y})^\top \tilde{\boldsymbol{v}}_j \nabla f(\boldsymbol{w}_j^\top \boldsymbol{x})\boldsymbol{x}
\end{aligned}$$

となる. ここで, $\tilde{\boldsymbol{v}}_j$ は V の列ベクトルである $(j = 1, \ldots, q)$. したがって, $\odot$ でベクトルの成分ごとの積[*60]を表すことにすれば,

$$\frac{\partial E(W,V)}{\partial W} = \left(\frac{\partial E(W,V)}{\partial \boldsymbol{w}_1}, \ldots, \frac{\partial E(W,V)}{\partial \boldsymbol{w}_q} \right)^\top$$

[*59] $\nabla f(\boldsymbol{w}_j^\top \boldsymbol{x})$ とは, $\left.\frac{\mathrm{d}f(x)}{\mathrm{d}x}\right|_{x=\boldsymbol{w}_j^\top \boldsymbol{x}} \in \mathbb{R}$ のことであった.

[*60] つまり, $\boldsymbol{a} \odot \boldsymbol{b} = (a_i b_i)_i \in \mathbb{R}^n$, $^\forall \boldsymbol{a}, \boldsymbol{b} \in \mathbb{R}^d$ のこと. **Hadamard 積**とも呼ばれ, 行列の Hadamard 積も成分ごとの積として定められる.

$$= \begin{pmatrix} \tilde{\boldsymbol{v}}_1^\top (g(V\boldsymbol{z}) - \boldsymbol{y}) \nabla f(\boldsymbol{w}_1^\top \boldsymbol{x}) \boldsymbol{x}^\top \\ \vdots \\ \tilde{\boldsymbol{v}}_q^\top (g(V\boldsymbol{z}) - \boldsymbol{y}) \nabla f(\boldsymbol{w}_q^\top \boldsymbol{x}) \boldsymbol{x}^\top \end{pmatrix} = \begin{pmatrix} \tilde{\boldsymbol{v}}_1^\top \\ \vdots \\ \tilde{\boldsymbol{v}}_q^\top \end{pmatrix} (g(V\boldsymbol{z}) - \boldsymbol{y}) \odot \begin{pmatrix} \nabla f(\boldsymbol{w}_1^\top \boldsymbol{x}) \\ \vdots \\ \nabla f(\boldsymbol{w}_q^\top \boldsymbol{x}) \end{pmatrix} \boldsymbol{x}^\top$$

が得られる. ここで, $\nabla f(W\boldsymbol{x}) = (\nabla f(\boldsymbol{w}_i^\top \boldsymbol{x}))_i \in \mathbb{R}^q$,

$$\tilde{V} = (\tilde{\boldsymbol{v}}_1, \ldots, \tilde{\boldsymbol{v}}_q) = \begin{pmatrix} v_{11} & \cdots & v_{1q} \\ \vdots & \ddots & \vdots \\ v_{m1} & \cdots & v_{mq} \end{pmatrix} \in \mathbb{R}^{m \times q}$$

とすれば, $\tilde{V}$ は V の 1 列目を取り除いた行列であり, したがって,

$$\frac{\partial E(W,V)}{\partial W} = \tilde{V}^\top (g(V\boldsymbol{z}) - \boldsymbol{y}) \odot \nabla f(W\boldsymbol{x}) \boldsymbol{x}^\top$$

となることがわかる. 以上より, t ステップ目での入力 $\boldsymbol{x}_t$ とパラメータ $W^{(t)}$ に対して, 中間層の出力は

$$\boldsymbol{z}^{(t)} = \begin{pmatrix} 1 \\ f(W^{(t)} \boldsymbol{x}_t) \end{pmatrix}$$

となるから, η_t を t ステップ目の学習率として,

$$V^{(t+1)} = V^{(t)} - \eta_t (g(V^{(t)} \boldsymbol{z}^{(t)}) - \boldsymbol{y}) \boldsymbol{z}^{(t)\top}$$
$$W^{(t+1)} = W^{(t)} - \eta_t \tilde{V}^{(t)\top} (g(V^{(t)} \boldsymbol{z}^{(t)}) - \boldsymbol{y}) \odot \nabla f(W^{(t)} \boldsymbol{x}) \boldsymbol{x}^\top$$

によって確率的勾配降下法を実装することができる.

ところで, パラメータ W を更新する際に, 誤差関数の $V\boldsymbol{z}$ での微分

$$\boldsymbol{\delta}_V = \left. \frac{\partial E(W,V)}{\partial V\boldsymbol{z}} \right|_{V\boldsymbol{z} = V^{(t)} \boldsymbol{z}^{(t)}}$$
$$= \left(\left. \frac{\partial E(W,V)}{\partial \boldsymbol{v}_1^\top \boldsymbol{z}} \right|_{\boldsymbol{v}_1^\top \boldsymbol{z} = \boldsymbol{v}_1^{(t)\top} \boldsymbol{z}^{(t)}}, \ldots, \left. \frac{\partial E(W,V)}{\partial \boldsymbol{v}_m^\top \boldsymbol{z}} \right|_{\boldsymbol{v}_m^\top \boldsymbol{z} = \boldsymbol{v}_m^{(t)\top} \boldsymbol{z}^{(t)}} \right)^\top = g(V^{(t)} \boldsymbol{z}^{(t)}) - \boldsymbol{y}$$

を用いて, 誤差関数の $W\boldsymbol{x}$ での微分

$$\boldsymbol{\delta}_W = \left. \frac{\partial E(W,V)}{\partial W\boldsymbol{x}} \right|_{W\boldsymbol{x} = W^{(t)} \boldsymbol{x}}$$
$$= \left(\left. \frac{\partial E(W,V)}{\partial \boldsymbol{w}_1^\top \boldsymbol{x}} \right|_{\boldsymbol{w}_1^\top \boldsymbol{x} = \boldsymbol{w}_1^{(t)\top} \boldsymbol{x}}, \ldots, \left. \frac{\partial E(W,V)}{\partial \boldsymbol{w}_q^\top \boldsymbol{x}} \right|_{\boldsymbol{w}_q^\top \boldsymbol{x} = \boldsymbol{w}_q^{(t)\top} \boldsymbol{x}} \right)^\top = \tilde{V}^{(t)\top} \boldsymbol{\delta}_V \odot \nabla f(W^{(t)} \boldsymbol{x})$$

を計算することができる. つまり, 出力層から中間層への誤差関数の微分 $\boldsymbol{\delta}_V$ を利用して中間層から入力層への誤差関数の微分 $\boldsymbol{\delta}_W$ を計算することができる. なお, $\boldsymbol{x} \to \boldsymbol{z} \to \boldsymbol{y}$ のような, 入力層から出力層へのネットワークを**順伝播**とよぶのに対し, 誤差がネットワークに逆らうように更新されることから, $\boldsymbol{\delta}_V$ から $\boldsymbol{\delta}_W$ を計算することを**逆伝播**とよぶ. 逆伝播を利用してパラメータを更新することを**誤差逆伝播法**と呼ぶ. 3 層ニューラルネットワークにおける誤差逆伝播法のアルゴリズムは Algorithm 2 の通りである.

3.2.2 深層ニューラルネットワーク

深層ニューラルネットワークとは, 大雑把に言えば, 3 層ニューラルネットワークの中間層を増やしたもので, これにより, より複雑なモデルを表現できるようになる. 図 14 は, 深層ニューラルネットワークのグラフ表現であり, 全部で r 個の中間層を持つネットワークを表している. なお, 第 j 層から第 $j+1$ 層へのネットワークは

$$\boldsymbol{z}_{j+1} = f_j(W_j \boldsymbol{z}_j)$$

Algorithm 2 誤差逆伝播法

入力: データ $\{(y_i, \boldsymbol{x}_i) \mid i = 1, \ldots, n\}$, 学習率 η_0, 中間層のユニット数 q, 活性化関数 f, g, 誤差関数 E, エポック数 T

パラメータ W, V の初期化
for $e = 1$ to T **do**
  $\mathcal{I} =$ { ランダムシャッフル後のデータのインデックス }
  $\eta_t \leftarrow \eta_0 / e$ ▷ 学習率の更新
  for i in $\mathcal{I}$ **do**
    $\boldsymbol{z} \leftarrow (1, f(W\boldsymbol{x})^\top)^\top$ ▷ 順伝播
    $\boldsymbol{\delta}_V \leftarrow g(V\boldsymbol{z}) - \boldsymbol{y}$; $\boldsymbol{\delta}_W \leftarrow \tilde{V}^\top \boldsymbol{\delta}_V \odot \nabla f(W\boldsymbol{x})$ ▷ 逆伝播
    $V \leftarrow V - \eta_t \boldsymbol{\delta}_V \boldsymbol{z}^\top$; $W \leftarrow W - \eta_t \boldsymbol{\delta}_W \boldsymbol{x}^\top$ ▷ パラメータの更新
  end for
end for

出力: パラメータ W, V の推定値

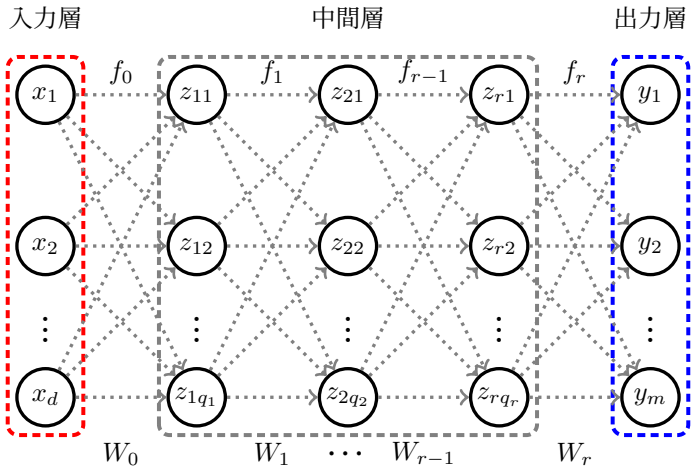


図 14　深層ニューラルネットワークのグラフ表現. r 個の中間層があり, 各層のユニット数はそれぞれ q_r である. また, それぞれの層における活性化関数は f_j, ネットワークの重みパラメータは W_j $(j = 0, \ldots, r)$ である.

と書くことができる. ただし,

$$W_j = \begin{pmatrix} \boldsymbol{w}_{j1}^\top \\ \vdots \\ \boldsymbol{w}_{jq_{j+1}}^\top \end{pmatrix} = \begin{pmatrix} w_{j10} & w_{j11} & \cdots & w_{j1,q_j} \\ \vdots & \vdots & \ddots & \vdots \\ x_{jq_{j+1}0} & w_{jq_{j+1}1} & \cdots & w_{jq_{j+1},q_j} \end{pmatrix}$$

は第 j 層から第 $j+1$ 層へのネットワークにおけるパラメータであり, 各 $\boldsymbol{w}_{jk}$ $(k = 1, \ldots, q_{j+1})$ は $q_j + 1$ 次元のベクトルである. なお, ネットワークを通して得られるモデルの出力は $f_r(W_r \boldsymbol{z}_r)$ である.

誤差関数は未知パラメータ $W_j \in \mathbb{R}^{q_{j+1} \times (q_j+1)}$ $(j = 0, \ldots, r)$ に依存する関数で, これまでと同様に, 回帰問題であれば

$$E(W_0, \ldots, W_r) = \|\boldsymbol{y} - f_r(W_r \boldsymbol{z}_r)\|^2 = \sum_{i=1}^{m} (y_i - f_r(\boldsymbol{w}_{ri}^\top \boldsymbol{z}_r))^2$$

が用いられ, この場合, f_r として恒等写像を用いるものとする．また, 分類問題の場合は, 誤差関数としてクロスエントロピー

$$E(W_0,\ldots,W_r) = -\sum_{i=1}^{m} y_i \log f_r(\boldsymbol{w}_{ri}^\top \boldsymbol{z}_r)$$

を用いる．ただし, f_r はソフトマックス関数である．このように誤差関数を設定することで, やはり 3 層ニューラルネットワークと同じように, 誤差関数の (最後の中間層から出力層への) パラメータ W_r での微分が

$$\frac{\partial E(W_0,\ldots,W_r)}{\partial W_r} = (f_r(W_r\boldsymbol{z}_r) - \boldsymbol{y})\boldsymbol{z}_r^\top$$

と, 簡単になる．以下, 深層ニューラルネットワークのパラメータ推定のための誤差逆伝播法について説明する．

■誤差逆伝播法　すでに述べたように, 誤差関数を最後の中間層から出力層へのパラメータで微分すると,

$$\frac{\partial E(W_0,\ldots,W_r)}{\partial W_r} = \frac{\partial E(W_0,\ldots,W_r)}{\partial W_r\boldsymbol{z}_r}\frac{\partial W_r\boldsymbol{z}_r}{\partial W_r} = (f_r(W_r\boldsymbol{z}_r) - \boldsymbol{y})\boldsymbol{z}_r^\top$$

と書くことができる．3 層ニューラルネットワークの場合と同じように考えて,

$$\boldsymbol{\delta}_r = \frac{\partial E(W_0,\ldots,W_r)}{\partial W_r\boldsymbol{z}_r} = f_r(W_r\boldsymbol{z}_r) - \boldsymbol{y}$$

と書くことにしよう．同様に,

$$\boldsymbol{\delta}_j = \frac{\partial E(W_0,\ldots,W_r)}{\partial W_j\boldsymbol{z}_j} = \left(\frac{\partial E(W_0,\ldots,W_r)}{\partial \boldsymbol{w}_{jk}^\top\boldsymbol{z}_j}\right)_{k=1,\ldots,q_{j+1}}$$

と書くことにする．$\boldsymbol{\delta}_j$ と $\boldsymbol{\delta}_{j+1}$ の関係を調べることで, 誤差逆伝播法における誤差の更新を考える．表記を簡単にするため, $u_{jk} = \boldsymbol{w}_{jk}^\top\boldsymbol{z}_j$ $(k = 1,\ldots,q_{j+1})$, つまり, $\boldsymbol{u}_j = W_j\boldsymbol{z}_j$ とする．以下, 添字が 3 つ並ぶのでやや煩わしく感じるかもしれないが, 3 層ニューラルネットワークにおけるパラメータ W での微分と対応していることを意識しながら, 確認してもらいたい．

まず, 合成関数の微分を用いることで

$$\delta_{jk} = \frac{\partial E(W_0,\ldots,W_r)}{\partial u_{jk}} = \sum_{l=1}^{q_{j+2}} \frac{\partial E(W_0,\ldots,W_r)}{\partial u_{j+1,l}}\frac{\partial u_{j+1,l}}{\partial u_{jk}}$$

と書くことができる．和の中の第 1 項目は, $\boldsymbol{\delta}_{j+1}$ の定義から $\delta_{j+1,l}$ である．第 2 項目は,

$$u_{j+1,l} = \boldsymbol{w}_{j+1,l}^\top\boldsymbol{z}_{j+1} = w_{j+1,l0} + w_{j+1,l1}f_j(u_{j1}) + \cdots + w_{j+1,lq_{j+1}}^\top f_j(u_{jq_{j+1}})$$

となることから,

$$\frac{\partial u_{j+1,l}}{\partial u_{jk}} = w_{j+1,lk}\nabla f_j(u_{jk})$$

となる．したがって,

$$\begin{aligned}\delta_{jk} &= \sum_{l=1}^{q_{j+2}} \delta_{j+1,l}w_{j+1,lk}\nabla f_j(u_{jk}) \\ &= (\delta_{j+1,1},\ldots,\delta_{j+1,q_{j+2}})\begin{pmatrix} w_{j+1,1k} \\ \vdots \\ w_{j+1,q_{j+2}k}\end{pmatrix}\nabla f_j(u_{jk}) = \boldsymbol{\delta}_{j+1}^\top\tilde{\boldsymbol{w}}_{j+1,k}\nabla f_j(u_{jk})\end{aligned}$$

Algorithm 3 深層ニューラルネットワークにおける誤差逆伝播法

入力: データ $\{(y_i, \boldsymbol{x}_i) \mid i = 1, \ldots, n\}$, 学習率 η_0, 中間層の数 r, 中間層のユニット数 $q_1, \ldots, q_r$, 活性化関数 $f_0, \ldots, f_r$, 誤差関数 E, エポック数 T

パラメータ $W_0, \ldots, W_r$ の初期化
for $e = 1$ to T **do**
  $\mathcal{I} =$ { ランダムシャッフル後のデータのインデックス }
  $\eta_t \leftarrow \eta_0/e$ ▷ 学習率の更新
  for t in $\mathcal{I}$ **do**
    for $j = 0$ to r **do**
      $\boldsymbol{z}_{j+1} \leftarrow f_j(W_j \boldsymbol{z}_j)$ ▷ 順伝播
    end for
    for $j = r - 1$ to 1 **do**
      $\boldsymbol{\delta}_j \leftarrow \tilde{W}_{j+1}^\top \boldsymbol{\delta}_{j+1} \odot \nabla f_j(W_j \boldsymbol{z}_j)$ ▷ 逆伝播: $\boldsymbol{\delta}_r = f_r(W_r \boldsymbol{z}_r) - \boldsymbol{y}$
    end for
    for $j = 0$ to r **do**
      $W_j \leftarrow W_j - \eta_e \boldsymbol{\delta}_{j+1} \boldsymbol{z}_j^\top$ ▷ パラメータの更新
    end for
  end for
end for

出力: パラメータ W, V の推定値

となることがわかる．ここで, $\tilde{\boldsymbol{w}}_{j+1,k}$ $(k = 1, \ldots q_{j+1})$ は W_{j+1} の列ベクトルである．したがって,

$$\begin{aligned}\boldsymbol{\delta}_j &= \begin{pmatrix} \boldsymbol{\delta}_{j+1}^\top \tilde{\boldsymbol{w}}_{j+1,1} \nabla f_j(u_{j1}) \\ \vdots \\ \boldsymbol{\delta}_{j+1}^\top \tilde{\boldsymbol{w}}_{j+1,q_{j+1}} \nabla f_j(u_{jq_{j+1}}) \end{pmatrix} = \begin{pmatrix} \tilde{\boldsymbol{w}}_{j+1,1}^\top \\ \vdots \\ \tilde{\boldsymbol{w}}_{j+1,q_{j+1}}^\top \end{pmatrix} \boldsymbol{\delta}_{j+1} \odot \begin{pmatrix} \nabla f_j(u_{j1}) \\ \vdots \\ \nabla f_j(u_{jq_{j+1}}) \end{pmatrix} \\ &= \tilde{W}_{j+1}^\top \boldsymbol{\delta}_{j+1} \odot \nabla f_j(\boldsymbol{u}_j) = \tilde{W}_{j+1}^\top \boldsymbol{\delta}_{j+1} \odot \nabla f_j(W_j \boldsymbol{z}_j)\end{aligned}$$

となる．ただし, $\tilde{W}_{j+1}$ は W_{j+1} の第 1 列目を取り除いた $q_{j+2} \times q_{j+1}$ 次元の行列である．また, 最後の等式で $\boldsymbol{u}_j = W_j \boldsymbol{z}_j$ であることを用いた.

以上より, 誤差の逆伝播は, $\boldsymbol{\delta}_r = f_r(W_r \boldsymbol{z}_r) - \boldsymbol{y}$ として, $j = r - 1, \ldots, 0$ に対して,

$$\boldsymbol{\delta}_j = \tilde{W}_{j+1}^\top \boldsymbol{\delta}_{j+1} \odot \nabla f_j(W_j \boldsymbol{z}_j)$$

とすることで逐次的に計算することができる．ただし, $\boldsymbol{z}_0 = (1, \boldsymbol{x}^\top)^\top$ とする．最後に,

$$\frac{\partial E(W_0, \ldots W_r)}{\partial W_j} = \frac{\partial E(W_0, \ldots W_r)}{\partial W_j \boldsymbol{z}_j} \frac{\partial W_j \boldsymbol{z}_j}{\partial W_j} = \boldsymbol{\delta}_j \boldsymbol{z}_j^\top$$

に注意すれば, 確率的勾配降下法によるパラメータの更新は

$$W_j \leftarrow W_j - \eta \boldsymbol{\delta}_j \boldsymbol{z}_j^\top$$

とすれば良い．ただし, η は学習率である．深層ニューラルネットワークにおける誤差逆伝播法のアルゴリズムは Algorithm 3 の通りである．ただし, Algorithm 3 において, $\boldsymbol{z}_0 = (1, \boldsymbol{x}^\top)^\top$ であり, $\boldsymbol{z}_{r+1} = f_r(W_r \boldsymbol{z}_r)$ はモデルの出力を表している．

3.3 オートエンコーダ

データ解析を行う場合, データの特徴を捉えるために, データの次元を削減して, 例えば 2 次元平面上で可視化することがある. これは, データの特徴をうまく捉えつつ, より少数の変数でデータを説明したいというモチベーションに基づく解析手法であり, **次元圧縮**などとも呼ばれる. 次元圧縮は, データ解析のためだけでなく, 例えば, 画像の圧縮と復元などにも応用される. ここでは, 次元圧縮を行うためにニューラルネットワークを用いる手法について説明する. これまでとは異なり, 次元圧縮, 正解ラベルや関数の出力値を用いない, いわゆる**教師なし学習**と呼ばれる種類のデータ解析手法である.

以下ではまず, 代表的な教師なし学習である主成分分析について説明し, オートエンコーダとの関係を述べる. その後, オートエンコーダのいくつかの拡張について説明する. なお, 近年注目を集めている **generative adversarial network (GAN)** や **variational auto encoder (VAE)** なども, ここで説明するオートエンコーダに深く関わっているが, この講義で扱うモデルのレベルをはるかに超えるため, これらについては述べない. 興味があれば, 原論文や web などで各自調べてもらいたい.

3.3.1 次元圧縮と主成分分析, オートエンコーダ

主成分分析とは, データを要約する変数を構成することを目的とした多変量解析の手法であり, 次元圧縮のための手法として古くから研究されてきた. 主成分分析の目的は, d 次元のベクトル $\boldsymbol{x} = (x_1, \ldots, x_d)^\top$ の線形結合

$$z = \boldsymbol{w}^\top \boldsymbol{x} = w_1 x_1 + \cdots + w_d x_d$$

によって, データを要約する特徴 $w_1, \ldots, w_d$ を見つけることである. どのような線形結合を考えれば, データを要約したと言えるだろうか. これを定式化するために, 観測データは d 次元空間からの標本であるが, 背後にはデータをよく説明できる低次元の構造があるとしよう. 言い換えれば, 図 15 のように, n 個の観測値 $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n \in \mathbb{R}^d$ を

$$z_i = \boldsymbol{w}^\top \boldsymbol{x}_i$$

と線形変換したとき, 上手に $\boldsymbol{w}$ を推定すれば, この $\boldsymbol{w}$ に沿ってデータの線形変換 $z_1, \ldots, z_n$ は最もばらついているとする. なお, $\boldsymbol{w}$ の推定値を $\boldsymbol{w}_1$ としたとき, $\boldsymbol{w}_1$ は**第 1 主成分軸**と呼ばれる. 次に, $\boldsymbol{w}_1$ に直行する $\boldsymbol{w}_2$ 方向に $z_1, \ldots, z_n$ が最もばらついているとき, $\boldsymbol{w}_2$ を**第 2 主成分軸**と呼び, 以下同様である[*61].

データのばらつきを測る尺度の一つに分散があることを思い出そう. $\bar{\boldsymbol{x}} = \sum_{i=1}^n \boldsymbol{x}_i / n$ を $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$ の標本平均 (ベクトル) とする. このとき,

$$\bar{z} = \frac{1}{n}\sum_{i=1}^n z_i = \frac{1}{n}\sum_{i=1}^n \boldsymbol{w}^\top \boldsymbol{x}_i = \boldsymbol{w}\bar{\boldsymbol{x}}$$

を $z_1, \ldots, z_n$ の標本平均とすれば, その標本分散は

$$\frac{1}{n}\sum_{i=1}^n (z_i - \bar{z})^2 = \frac{1}{n}\sum_{i=1}^n \{\boldsymbol{w}^\top(\boldsymbol{x}_i - \bar{\boldsymbol{x}})\}^2 = \boldsymbol{w}^\top \left\{ \frac{1}{n}\sum_{i=1}^n (\boldsymbol{x}_i - \bar{\boldsymbol{x}})(\boldsymbol{x}_i - \bar{\boldsymbol{x}})^\top \right\} \boldsymbol{w} = \boldsymbol{w}^\top V_X \boldsymbol{w}$$

と書き換えることができる. ただし, V_X は $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$ の標本分散である[*62]. したがって, $\boldsymbol{w}$ として,

$$\hat{\boldsymbol{w}} = \arg\max_{\boldsymbol{w} \in \mathbb{R}^d} \boldsymbol{w}^\top V_X \boldsymbol{w}$$

とすれば良いように思われる. ところが, V_X は非負定値行列であるから, $\boldsymbol{w}$ の各成分を十分大きくすれば, この目的関数も好きなだけ大きくすることができてしまう. そこで, 代わりに

$$\max_{\boldsymbol{w} \in \mathbb{R}^d} \frac{\boldsymbol{w}^\top V_X \boldsymbol{w}}{\|\boldsymbol{w}\|^2}$$

[*61] つまり, 各主成分軸は $\mathbb{R}^d$ の**正規直交系**をなす.

[*62] V_X は標本分散共分散行列とも呼ばれる.

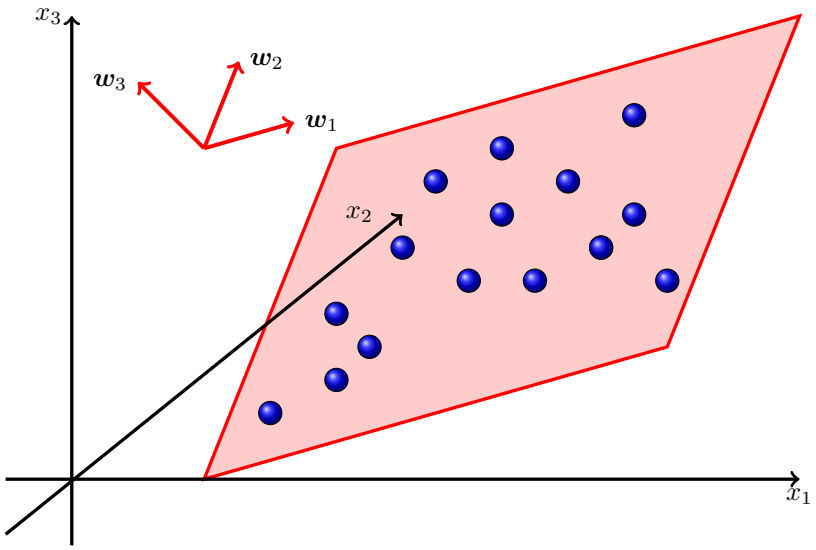


図 15　主成分分析のイメージ．$\boldsymbol{w}_1, \boldsymbol{w}_2, \boldsymbol{w}_3$ は主成分軸を表している．

を解くことによって $\boldsymbol{w}$ の推定値としよう．目的関数をじっと眺めると，$\boldsymbol{w}$ を定数倍しても分母と分子で打ち消しあうので，この問題は

$$\max_{\boldsymbol{w}\in\mathbb{R}^d} \boldsymbol{w}^\top V_X \boldsymbol{w} \quad \text{subject to} \quad \|\boldsymbol{w}\|^2 = 1$$

と等価である．したがって，Lagrange 乗数を λ として，Lagrange 関数

$$L(\boldsymbol{w}) = \boldsymbol{w}^\top V_X \boldsymbol{w} + \lambda(1 - \|\boldsymbol{w}\|^2)$$

を最大化しよう．$L(\boldsymbol{w})$ をパラメータ $\boldsymbol{w}$ で微分すれば，

$$\frac{\partial L(\boldsymbol{w})}{\partial \boldsymbol{w}} = 2V_X \boldsymbol{w} - 2\lambda \boldsymbol{w}$$

となり，これをゼロとすれば，最適解の満たす方程式

$$V_X \boldsymbol{w} = \lambda \boldsymbol{w}$$

が得られる．したがって，$\boldsymbol{w}$ の最適解は V_X の固有値問題の解である[*63]．$\|\boldsymbol{w}\|^2 = 1$ であることに注意すれば，この方程式の両辺で $\boldsymbol{w}$ との内積を取れば，

$$\lambda = \boldsymbol{w}^\top V_X \boldsymbol{w}$$

つまり，求める $\boldsymbol{w}$ は V_X の最大固有値に対応する固有ベクトルである．これを $\boldsymbol{w}_1$ とし，対応する固有値を λ_1 とすれば，(先に述べたが)$\boldsymbol{w}_1$ は第 1 主成分軸と呼ばれ，また，λ_1 は**第 1 主成分**と呼ばれる．第 2 主成分軸は，最適化問題

$$\max_{\boldsymbol{w}\in\mathbb{R}^d} \boldsymbol{w}^\top V_X \boldsymbol{w} \quad \text{subject to} \quad \|\boldsymbol{w}\|^2 = 1,\ \boldsymbol{w}^\top \boldsymbol{w}_1 = 0$$

を解くことで得られる．念のため，第 2 主成分軸が V_X の 2 番目に大きな固有値に対応する固有ベクトルとして得られることを確認しておこう．先と同様に，Lagrange 関数

$$L(\boldsymbol{w}) = \boldsymbol{w}^\top V_X \boldsymbol{w} + \lambda(1 - \|\boldsymbol{w}\|^2) + \mu \boldsymbol{w}^\top \boldsymbol{w}_1$$

[*63] つまり，λ は V_X の固有値であり，$\boldsymbol{w}$ は対応する固有ベクトルである．

を用意する．これを $\boldsymbol{w}$ で微分すると

$$\frac{\partial L(\boldsymbol{w})}{\partial \boldsymbol{w}} = 2V_X\boldsymbol{w} - 2\lambda\boldsymbol{w} + \mu\boldsymbol{w}_1$$

となるので, この微分をゼロとしたとき

$$V_X\boldsymbol{w} = \lambda\boldsymbol{w} - \frac{\mu}{2}\boldsymbol{w}_1$$

が得られる．$\|\boldsymbol{w}\|^2 = 1, \boldsymbol{w}^\top\boldsymbol{w}_1 = 0$ に注意すれば,

$$\lambda = \boldsymbol{w}^\top V_X\boldsymbol{w}$$

となる．$\boldsymbol{w} \neq \boldsymbol{w}_1$ なので, 右辺を最大化する $\boldsymbol{w}$ は V_X の 2 番目に大きな固有値 λ_2 に対応する固有ベクトル $\boldsymbol{w}_2$ である．なお, 元の変数の次元 d に対して, より小さな q 次元空間でどの程度データを説明できるかを測る尺度として, **寄与率**

$$\frac{\lambda_1 + \cdots + \lambda_q}{\lambda_1 + \cdots + \lambda_d}$$

が用いられる．例えば, 第 q 主成分までの寄与率が 80% というと, 元の d 次元の構造のうち 80% を q 個の主成分軸で表現できたと解釈する．

オートエンコーダとの関係を明確にするために, 主成分分析で得られた q 個の主成分軸 $\boldsymbol{w}_1, \ldots, \boldsymbol{w}_q \in \mathbb{R}^d$ と最適化問題についてあらためて考えてみよう．主成分分析の目的は, 低次元空間でデータを表現することであり, 結果として, d 次元空間上の点は q 次元空間に射影される．つまり, 任意の点 $\boldsymbol{x} \in \mathbb{R}^d$ は

$$\boldsymbol{x} \approx \sum_{j=1}^{q} \langle \boldsymbol{w}_j, \boldsymbol{x} \rangle \boldsymbol{w}_j$$

で近似される[*64]．これを, n 個の観測点 $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$ とその近似 $\sum_{j=1}^{q}\langle \boldsymbol{w}_j, \boldsymbol{x}_1\rangle\boldsymbol{w}_j, \ldots, \sum_{j=1}^{q}\langle \boldsymbol{w}_j, \boldsymbol{x}_n\rangle\boldsymbol{w}_j$ の誤差を小さくするために $\boldsymbol{w}_1, \ldots, \boldsymbol{w}_q$ を求める問題だと考えれば,

$$\min_{\boldsymbol{w}_1,\ldots,\boldsymbol{w}_q} \sum_{i=1}^{n} \left\| \boldsymbol{x}_i - \sum_{j=1}^{q} \langle \boldsymbol{w}_j, \boldsymbol{x}_i \rangle \boldsymbol{w}_j \right\|^2$$

を解くことで, $\mathbb{R}^d$ の低次元表現 $\mathbb{R}^q$ が得られるであろう．ただし, $\boldsymbol{w}_1, \ldots, \boldsymbol{w}_q$ は正規直交系, つまり, $\boldsymbol{w}_i^\top\boldsymbol{w}_j = 1$ $(i = j)$ および 0 $(i \neq j)$ である．なお, $\boldsymbol{x}_i$ の代わりに, 中心化した $\boldsymbol{x}_i - \bar{\boldsymbol{x}}$ を考えることで, この最適化問題と主成分分析によって q 個の主成分軸を求める問題は等価となる．ところで, $W = (\boldsymbol{w}_1, \ldots, \boldsymbol{w}_q)^\top \in \mathbb{R}^{q\times d}$ とすれば,

$$\sum_{j=1}^{q} \langle \boldsymbol{w}_j, \boldsymbol{x}_i \rangle \boldsymbol{w}_j = \sum_{j=1}^{q} \boldsymbol{w}_j\boldsymbol{w}_j^\top\boldsymbol{x}_i = W^\top W\boldsymbol{x}_j$$

と書き換えることができる．したがって, 最適化問題は

$$\min_{W\in\mathbb{R}^{q\times d}} \sum_{i=1}^{n} \|\boldsymbol{x}_i - W^\top W\boldsymbol{x}_i\|^2 \quad \text{subject to} \quad WW^\top = I_q$$

となる．ただし, 制約は W が正規直交系であることを考慮したものであり, I_q は q 次元の単位行列である．$\tilde{\boldsymbol{x}}_i = W^\top W\boldsymbol{x}_i$ とすれば, $\tilde{\boldsymbol{x}}_i$ は $\boldsymbol{x}_i$ の q 次元への圧縮 $W\boldsymbol{x}_i$ を元の次元 d へ復元したもの $W^\top W\boldsymbol{x}_i$ と見る

[*64] このノートでは特に気にする必要はないけれど, これは $\boldsymbol{x}$ の **Fourier 級数展開**を次数 q で打ち切ったものである．この近似は, **Karhunen-Loéve 展開**の特別な場合としても知られている．同様に, $\|\boldsymbol{x}\|^2 \approx \sum_{j=1}^{q} |\langle \boldsymbol{w}_j, \boldsymbol{x}\rangle|^2$ (**Riesz-Fischer の定理**の有限近似) や, 任意の $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^d$ に対して, $\langle \boldsymbol{x}, \boldsymbol{y}\rangle \approx \sum_{j=1}^{q}\langle \boldsymbol{w}_j, \boldsymbol{x}\rangle\langle \boldsymbol{w}_j, \boldsymbol{y}\rangle$ (**Parseval の定理**の有限近似) を考えることもできる．詳しくは関数解析の書籍を参照してもらいたい．

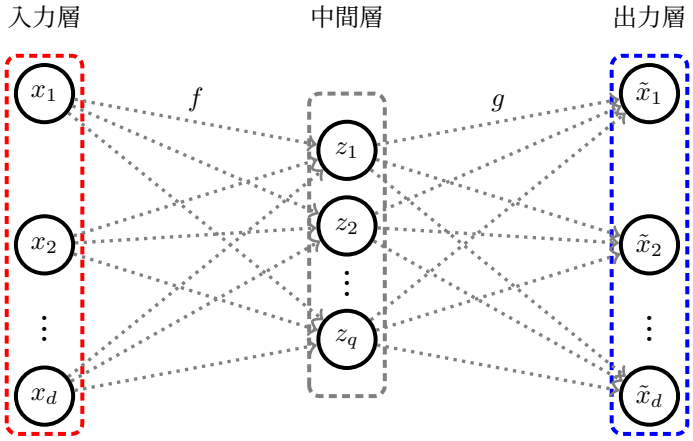


図 16　オートエンコーダのグラフ表現．モデルとしては 3 層ニューラルネットワークと同じであるが，出力層が正解ラベルではなく，復元後のデータである点が異なる．中間層のユニット数は一般に $q \leq d$ とする．中間層のユニット数が入出力層に比べて少ないため，砂時計型ニューラルネットワークとも呼ばれる．

ことができる．つまり，f, g を恒等写像としたとき，$\tilde{\boldsymbol{x}}_i = g(W^\top f(W\boldsymbol{x}))$ のように，復元後のデータは恒等写像の合成関数である．以上をまとめると，主成分分析とは

$$\min_{W \in \mathbb{R}^{q \times d}} \sum_{i=1}^{n} \|\boldsymbol{x}_i - g(W^\top f(W\boldsymbol{x}_i))\|^2 \quad \text{subject to} \quad WW^\top = I_q$$

によって，パラメータ W を推定する問題に他ならない．そのため，

- 教師なし学習であること
- 入力層と中間層のバイアス項がないこと
- 入力層から中間層へのパラメータと，中間層から出力層へのパラメータが同じもの[*65]
- 正規直交系である制約が含まれていること

であることを除けば，3.2.1 節で述べた 3 層ニューラルネットワークと (ほぼ) 同じものであると理解できる．このモデルは，低次元へ圧縮し，元の次元へ自分自身を復元することから，**オートエンコーダ**と呼ばれ，主成分分析の非線形化と考えることができる．図 16 はオートエンコーダをネットワークとしてグラフ表現したものである．一般に，中間層のユニット数が入出力層よりも小さいことから，**砂時計型ニューラルネットワーク**としても知られている．なお，圧縮 · 復元という文脈との関係性から，入力層から中間層への変換は**符号化 (encoder)**，中間層から出力層への変換は**復号化 (decoder)** と呼ばれる．

3.3.2　オートエンコーダの学習

オートエンコーダで用いる誤差関数や活性化関数も，3.2 節で述べたニューラルネットワークと同様である．以降，入力層と中間層はバイアス項を含むものとし，$\boldsymbol{x}^\dagger$ や $\boldsymbol{z}^\dagger$ で，バイアス項を含むベクトル $(1, \boldsymbol{x}^\top)^\top$ や $(1, \boldsymbol{z}^\top)^\top$ を表すものとする[*66]．中間層のユニット数が q であるときに，バイアス項を含むパラメータ $W_0 \in \mathbb{R}^{q \times (d+1)}, W_1 \in \mathbb{R}^{d \times (q+1)}$ を用いて，入力 $\boldsymbol{x} \in \mathbb{R}^d$ に対するオートエンコーダの出力 $\tilde{\boldsymbol{x}}$ は

$$\tilde{\boldsymbol{x}} = g(W_1 \boldsymbol{z}^\dagger) = g(W_1(1, f(W_0 \boldsymbol{x}^\dagger)^\top)^\top)$$

で与えられる．ただし，f, g は活性化関数である．3 層ニューラルネットワークの場合と同様に，$\boldsymbol{x}$ が連続値であれば，復号化した $\tilde{\boldsymbol{x}}$ との二乗誤差

$$E(W_0, W_1) = \frac{1}{2}\|\boldsymbol{x} - \tilde{\boldsymbol{x}}\|^2 = \frac{1}{2}\|\boldsymbol{x} - g(W_1 \boldsymbol{z}^\dagger)\|^2$$

[*65] この制約を課すことを**重み共有**と呼ぶ．

[*66] 入力 $\boldsymbol{x}$ と出力 $\tilde{\boldsymbol{x}}$ での誤差を評価するため，区別しておかないと混乱を招いてしまう．

を小さくするように, パラメータ W_0, W_1 を推定する. ただし, g として恒等写像を用いる. この場合, 主成分分析を非線形にしたような圧縮表現 $\boldsymbol{z}$ が得られることとなる. また, $\boldsymbol{x}$ が 0 または 1 であるような 2 値のベクトルであれば, クロスエントロピー

$$E(W_0, W_1) = -\sum_{j=1}^{d} x_j \log \tilde{x}_j = -\sum_{j=1}^{d} x_j \log g(\boldsymbol{w}_{1j}^\top \boldsymbol{z}^\dagger)$$

を誤差関数として用いる. ただし, この場合は g としてシグモイド関数が用いられる[*67].

以下で, 誤差逆伝播法によるパラメータ推定について説明する.

■誤差逆伝播法 誤差関数として二乗誤差, 中間層から出力層への活性化関数 g として恒等写像を用いる場合, これまでと同様に誤差関数を $W_1 = (\boldsymbol{w}_{11}, \ldots, \boldsymbol{w}_{1d})^\top \in \mathbb{R}^{d\times(q+1)}$ で微分すると

$$\frac{\partial E(W_0, W_1)}{\partial W_1} = (\tilde{\boldsymbol{x}} - \boldsymbol{x})\boldsymbol{z}^{\dagger\top}$$

と簡単に書くことができる. では, 誤差関数がクロスエントロピーで, g がシグモイド関数の場合はどうだろうか. シグモイド関数

$$g(x) = \frac{1}{1 + e^{-x}}$$

の微分が

$$\frac{\mathrm{d}g(x)}{\mathrm{d}x} = g(x)(1 - g(x))$$

で与えられることを思い出そう. したがって, 合成関数の微分を用いれば,

$$\begin{aligned}\frac{\partial \log g(\boldsymbol{w}_{1j}^\top \boldsymbol{z}^\dagger)}{\partial \boldsymbol{w}_{1j}} &= \frac{1}{g(\boldsymbol{w}_{1j}^\top \boldsymbol{z}^\dagger)} \frac{\partial g(\boldsymbol{w}_{1j}^\top \boldsymbol{z}^\dagger)}{\partial \boldsymbol{w}_j} \\ &= \frac{1}{g(\boldsymbol{w}_{1j}^\top \boldsymbol{z}^\dagger)} \times g(\boldsymbol{w}_{1j}^\top \boldsymbol{z}^\dagger)(1 - g(\boldsymbol{w}_{1j}^\top \boldsymbol{z}^\dagger))\boldsymbol{z}^\dagger = (1 - g(\boldsymbol{w}_{1j}^\top \boldsymbol{z}^\dagger))\boldsymbol{z}^\dagger\end{aligned}$$

であり, $k \neq j$ ならば $\frac{\partial \log g(\boldsymbol{w}_{1j}^\top \boldsymbol{z}^\dagger)}{\partial \boldsymbol{w}_{1k}} = \boldsymbol{0}$ となることがわかる. よって,

$$\frac{\partial E(W_0, W_1)}{\partial \boldsymbol{w}_{1j}} = -x_j \frac{\partial \log g(\boldsymbol{w}_{1j}^\top \boldsymbol{z}\dagger)}{\partial \boldsymbol{w}_{1j}} = x_j(g(\boldsymbol{w}_{1j}^\top \boldsymbol{z}^\dagger) - 1)\boldsymbol{z}^\dagger$$

であるから, $\boldsymbol{1}_d = (1, \ldots, 1)^\top \in \mathbb{R}^d$ を 1 が d 個並んだベクトルすれば,

$$\begin{aligned}\frac{\partial \log g(\boldsymbol{w}_{1j}^\top \boldsymbol{z}^\dagger)}{\partial W_1} &= \left(\frac{\partial \log g(\boldsymbol{w}_{1j}^\top \boldsymbol{z}^\dagger)}{\partial \boldsymbol{w}_{11}}, \ldots, \frac{\partial \log g(\boldsymbol{w}_{1j}^\top \boldsymbol{z}^\dagger)}{\partial \boldsymbol{w}_{1d}}\right)^\top = \begin{pmatrix} x_1(g(\boldsymbol{w}_{11}^\top \boldsymbol{z}^\dagger) - 1) \\ \vdots \\ x_d(g(\boldsymbol{w}_{1d}^\top \boldsymbol{z}^\dagger) - 1) \end{pmatrix} \boldsymbol{z}^\top \\ &= \{\boldsymbol{x} \odot (g(W_1 \boldsymbol{z}^\dagger) - \boldsymbol{1}_d)\}\boldsymbol{z}^{\dagger\top} = \{\boldsymbol{x} \odot (\tilde{\boldsymbol{x}} - \boldsymbol{1}_d)\}\boldsymbol{z}^{\dagger\top}\end{aligned}$$

以下では, 誤差関数が二乗誤差で, g が恒等写像の場合には

$$\boldsymbol{\delta}_1 = \frac{\partial E(W_0, W_1)}{\partial W_1 \boldsymbol{z}^\dagger} = \tilde{\boldsymbol{x}} - \boldsymbol{x}$$

誤差関数がクロスエントロピーで, g がシグモイド関数の場合には

$$\boldsymbol{\delta}_1 = \frac{\partial E(W_0, W_1)}{\partial W_1 \boldsymbol{z}^\dagger} = \boldsymbol{x} \odot (\tilde{\boldsymbol{x}} - \boldsymbol{1}_d)$$

と書くことにする.

[*67] これまでと異なり, 出力の和が 1 となる必要がないため, ソフトマックス関数を用いなくても良い.

Algorithm 4 オートエンコーダの誤差逆伝播法

入力: データ $\{\boldsymbol{x}_i \in \mathbb{R}^d \mid i = 1, \ldots, n\}$, 学習率 η_0, 中間層のユニット数 q, 活性化関数 f, g, 誤差関数 E, エポック数 T

パラメータ W_0, W_1 の初期化
for $e = 1$ to T **do**
  $\mathcal{I} =$ { ランダムシャッフル後のデータのインデックス }
  $\eta_e \leftarrow \eta_0/e$ ▷ 学習率の更新
  for t in $\mathcal{I}$ **do**
    $\boldsymbol{x}^\dagger \leftarrow (1, \boldsymbol{x}^\top)^\top;\ \boldsymbol{z}^\dagger \leftarrow (1, f(W_0\boldsymbol{x}^\dagger)^\top)^\top;\ \tilde{\boldsymbol{x}} \leftarrow g(W_1\boldsymbol{z}^\dagger)$ ▷ 順伝播
    $\boldsymbol{\delta}_1 \leftarrow \tilde{\boldsymbol{x}} - \boldsymbol{x}$ または $\boldsymbol{\delta}_1 \leftarrow \boldsymbol{x} \odot (\tilde{\boldsymbol{x}} - \mathbf{1}_d);\ \boldsymbol{\delta}_0 \leftarrow \tilde{W}_1^\top \boldsymbol{\delta}_1 \odot \nabla f(W_0\boldsymbol{x}^\dagger)$ ▷ 逆伝播
    $W_1 \leftarrow W_1 - \eta_e \boldsymbol{\delta}_1 \boldsymbol{z}^\dagger;\ W_0 \leftarrow W_0 - \eta_e \boldsymbol{\delta}_0 \boldsymbol{x}^\dagger$ ▷ パラメータの更新
  end for
end for

出力: パラメータ W_0, W_1 の推定値および, エポックごとの誤差関数 $\mathcal{E} \in \mathbb{R}^T$

誤差関数の W_0 での微分を計算しよう. とはいえ, やるべきことはこれまでと同様であり, まず $\boldsymbol{u} = W_0\boldsymbol{x}^\dagger$ とし,

$$\boldsymbol{\delta}_0 = \frac{\partial E(W_0, W_1)}{\partial \boldsymbol{u}}$$

を考える. 3.2.1 節で説明したことと同様の計算を行えば,

$$\boldsymbol{\delta}_0 = \tilde{W}_1^\top \boldsymbol{\delta}_1 \odot \nabla f(W_0\boldsymbol{x}^\dagger)$$

となる. したがって, パラメータの更新は

$$W_1 \leftarrow W_1 - \eta \boldsymbol{\delta}_1 \boldsymbol{z}^{\dagger\top}, \qquad W_0 \leftarrow W_0 - \eta \boldsymbol{\delta}_0 \boldsymbol{x}^{\dagger\top}$$

とすれば良い. ここで, η は学習率である.

3.3.3 オートエンコーダに関連するニューラルネットワークモデル

オートエンコーダは, ニューラルネットワークの中で比較的初期の段階 (2006 年) に提案されたもので, 以降様々な拡張がなされてきた. その中には,

- 単純にオートエンコーダの中間層を深くする手法
- ロバストにパラメータを推定するため, 入力にあえてノイズを加える手法
- 有効な特徴を抽出するために, 前処理としてオートエンコーダを利用する手法
- 中間層のユニット数を自動的に決定するために誤差関数に (中間層の値 z_j を正確に 0 に縮小するという意味で) スパース性を促す罰則を加える手法

をはじめ, 様々なものが提案されている. 以下では, デノイジングオートエンコーダを紹介し, 生成モデルとの関係性, 特に最適輸送との関連について述べる.

■デノイジングオートエンコーダ 以下, オートエンコーダの活性化関数 g は恒等写像であるものとし, 誤差関数は二乗損失, つまり,

$$E(W_0, W_1) = \frac{1}{2}\|\boldsymbol{x} - \tilde{\boldsymbol{x}}\|^2$$

であるものとする. ただし, $\tilde{\boldsymbol{x}} = W_1(1, f(W_0\boldsymbol{x}^\dagger)^\top)^\top$ であり, $\boldsymbol{x}^\dagger$ はバイアス項を含む入力 $(1, \boldsymbol{x}^\top)^\top$ である. 主成分分析の部分で説明したように, オートエンコーダは, 入力を要約する低次元表現を獲得すること

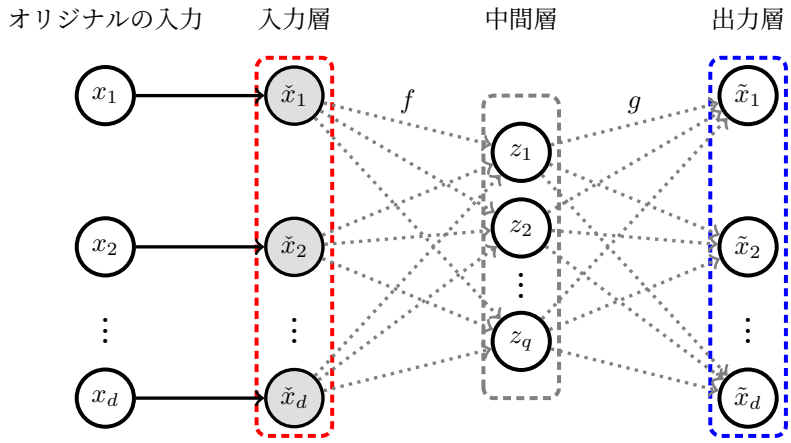


図 17　デノイジングオートエンコーダのグラフ表現．オリジナルの入力にノイズを加えたものが入力層として与えられることを除けば，ほぼ通常のオートエンコーダと同じものである．

を目的としている．ところが，実際には，適当な条件のもとで，中間層の数 q が入力の次元 d より大きい，つまり，$q \geq d$ である場合には，誤差を最小にする活性化関数 f は恒等写像であることが知られている．直感的には，入力の次元よりも大きなものでパラメータを学習しようとすると，入力の値をそのまま出力層へ流してしまえば誤差をほとんどゼロにできてしまうということである[*68]．したがって，オートエンコーダで非線形な圧縮表現を得ようとすると，結果的に冗長なモデルを考えていることと同じになってしまう．このことは，単に $q \geq d$ である場合に限らず，$q \approx d$ の場合にもデータが単純であれば起こりうる問題である．この問題に対処するため，先に述べたように，誤差関数に中間層の数を少数にするような罰則を課したり，通常のオートエンコーダのように，$q \ll d$ であるようなモデルが用いられてきた．

これに対し，オートエンコーダの拡張 (あるいは亜種として)，2008 年に**デノイジングオートエンコーダ**が提案された (図 17)．オートエンコーダは，入力 $\boldsymbol{x}$ に意図的にノイズを加えることで，恒等関数が最適でない状況を作り出す手法と考えることができる．具体的には，まず，入力 $\boldsymbol{x}$ に適当なノイズ $\boldsymbol{\varepsilon}$ を加え，

$$\check{\boldsymbol{x}} = \boldsymbol{x} + \boldsymbol{\varepsilon}$$

をニューラルネットワークの入力層の値とする．すると，出力層では，ノイズが付加されたデータに基づく出力 $\tilde{\boldsymbol{x}}^\ddagger = W_1(1, f(W_0\check{\boldsymbol{x}}^\dagger)^\top)^\top$ が得られる．デノイジングオートエンコーダでは，このようにして得られた出力に基づき，

$$E(W_0, W_1) = \frac{1}{2}\|\boldsymbol{x} - \tilde{\boldsymbol{x}}^\ddagger\|^2$$

を小さくするようにパラメータ W_0, W_1 を学習する．パラメータ推定は，Algorithm 4 の 7 行目で

$$\boldsymbol{x}^\dagger \leftarrow (1, \boldsymbol{x}^\top)^\top$$

の代わりに，

$$\boldsymbol{x}^\dagger \leftarrow (1, (\boldsymbol{x} + \boldsymbol{\varepsilon})^\top)^\top$$

とすれば良いので推定アルゴリズムの詳細については，省略することにする．

ところで，どのようなノイズを加えるかについては，様々な方法が知られており，例えば以下のようなものがよく用いられる:

- 入力の一部，例えば 30% をランダムに選び，0 で上書きする (**salt and pepper noise** とも呼ばれる)．

[*68] 言い換えれば，入力データに対してオーバーフィットしてしまうということ．

- σ^2 を既知として, $\varepsilon \sim \mathrm{N}(0, \sigma^2)$ をオリジナルの入力に加える (e.g., $\check{\boldsymbol{x}} = \boldsymbol{x} + \varepsilon \boldsymbol{x}$ とする).
- σ^2 を既知として, $\boldsymbol{\varepsilon} \sim \mathrm{N}(0, \sigma^2 I_d)$ をオリジナルの入力に加える (e.g., $\check{\boldsymbol{x}} = \boldsymbol{x} + \boldsymbol{\varepsilon}$ とする).

以下では, デノイジングオートエンコーダの**輸送写像**[*69]としての解釈を述べるため, ノイズは $\boldsymbol{\varepsilon} \sim \nu = \mathrm{N}(0, \sigma^2 I_d)$ から得られるものとする. つまり, ノイズの確率密度関数は

$$\nu(\boldsymbol{\varepsilon}) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{\|\boldsymbol{\varepsilon}\|^2}{2\sigma^2}}$$

で与えられるものとする. また, 入力はある確率分布からのサンプルであるとし, $\boldsymbol{x} \sim \pi$ としておく. このとき, デノイジングオートエンコーダは, 次の変分問題

$$\min_g \mathbb{E}_\pi \mathbb{E}_\nu \|g(\boldsymbol{x} + \boldsymbol{\varepsilon}) - \boldsymbol{x}\|^2$$

を解くことと等価であることが知られている. ただし, g は十分に広いクラスの関数を表現でき, 停留点を達成できるものとする[*70]. この最適化問題の意味するところは, “ノイズが加えられた入力と, そこから得られるニューラルネットワークの出力が与えられたときに, 平均的にオリジナルの入力を近似する関数を推定せよ”ということである. このとき, 目的関数

$$\mathbb{E}_\pi \mathbb{E}_\nu \|g(\boldsymbol{x} + \boldsymbol{\varepsilon}) - \boldsymbol{x}\|^2 = \iint \|g(\boldsymbol{x} + \boldsymbol{\varepsilon}) - \boldsymbol{x}\|^2 \nu(\boldsymbol{\varepsilon}) \pi(\boldsymbol{x}) \mathrm{d}\boldsymbol{\varepsilon} \mathrm{d}\boldsymbol{x}$$

を g で変分することで, この問題の解 g^* は

$$g^*(\boldsymbol{x}) = \boldsymbol{x} - \frac{1}{\nu * \pi(\boldsymbol{x})} \int \boldsymbol{\varepsilon} \nu(\boldsymbol{\varepsilon}) \pi(\boldsymbol{x} - \boldsymbol{\varepsilon}) \mathrm{d}\boldsymbol{\varepsilon}$$

で与えられる. ただし,

$$\nu * \pi(\boldsymbol{x}) = \int \nu(\boldsymbol{\varepsilon}) \pi(\boldsymbol{x} - \boldsymbol{\varepsilon}) \mathrm{d}\boldsymbol{\varepsilon}$$

は二つの分布 ν および π の**畳み込み積分**である. 最適解の写像としての解釈を考えよう. まず, 第 1 項は恒等写像 $\boldsymbol{x} \mapsto \boldsymbol{x}$ である. また, 第 2 項は意図的に加えられたノイズを除去したことに伴う補正項 (おつり) であると考えられる. したがって, デノイジングオートエンコーダは, 入力 $\boldsymbol{x}$ を補正項の方向に輸送する輸送写像とみなすことができる.

ところで, ノイズの分布が平均 0 の正規分布であることを用いると, **Stein の等式**

$$\boldsymbol{\varepsilon} \nu(\boldsymbol{\varepsilon}) = -\sigma^2 \nabla \nu(\boldsymbol{\varepsilon})$$

を用いれば, 最適解はさらに

$$g^*(\boldsymbol{x}) = \boldsymbol{x} + \sigma^2 \nabla \log(\nu * \pi)(\boldsymbol{x})$$

と書き換えることができることが知られている. これは, 最適解 g^* が, 入力 $\boldsymbol{x}$ を畳み込むことで得られる対数尤度 $(\nu * \pi)(\boldsymbol{x})$ の勾配方向へ $\boldsymbol{x}$ を輸送するものとして解釈できる. つまり, $\boldsymbol{x}$ を $\nu * \pi$ から得られるスコア $-\nabla \log(\nu * \pi)(\boldsymbol{x})$ に関する勾配降下法とみなすことができる.

3.4 再帰型ニューラルネットワーク

計時的に測定されたデータなど, 観測された値が過去の観測履歴に影響を受けるデータは多い. 例えば, 文章などは, 文法の構造を持つため, それぞれのテキストを個別に解析するよりは, その前後関係を考慮したモデルの方がより良い結果が得られることが期待される. また, 画像データも, あるピクセルにおける輝

[*69] やや数学的な毛色が強いため, 数学が苦手な方は飛ばしてもらって結構である. とはいえ, この手の理論はニューラルネットワークの解釈や, 様々な謎を解くために重要な道具であることを理解しておくと後々役に立つかもしれない.

[*70] よくわからなければ, デノイジングオートエンコーダの出力と入力の二乗損失だと思えば良い.

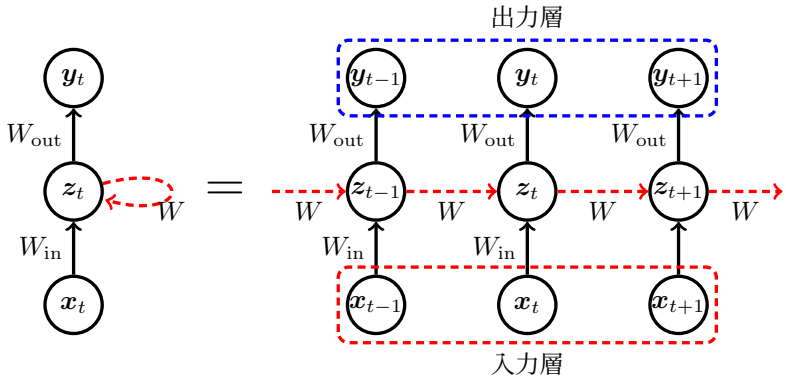


図 18　再帰型ニューラルネットワークのグラフ表現．深層ニューラルネットワークとは異なり, 中間層が時点間で連鎖していることが見て取れる．各時点におけるネットワークは単純な 3 層ニューラルネットワークである．また, 図の左側は再帰型ニューラルネットワークを簡略化した表現であり, 実態は右図のようになっている．

度は, そのピクセルの近傍と近い値をとることは直感的にもわかりやすいと思う．これまで説明したニューラルネットワークは, このような時間情報 (あるいは近傍の情報) を考慮しないモデルであった．ここでは, このような依存関係を考慮したニューラルネットワークについて説明する．モデルとしてはこれまでのものよりも多少複雑にはなるが, やるべきことはこれまで通り, 順伝播, 逆伝播, パラメータの更新を順に行うということである．したがって, あまり難しく考えすぎず, この点を意識しながら読んでもらいたい．

3.4.1　再帰型ニューラルネットワークの学習

再帰型ニューラルネットワークのグラフ表現は図 18 のように表現することができる．図 18 の左辺は再帰型ニューラルネットワークを簡略化した表現であり, 実際には, 各時点ごとに入出力が定まるネットワークである．各時点におけるネットワーク (黒線) は単純な 3 層ニューラルネットワークであるが, 各時点での中間層が, 赤の破線のように連鎖している点が異なる．再帰型ニューラルネットワークでは, 中間層での伝播を考えることで, 時点間の依存関係を表現したモデルと考えることができる．ネットワークにおけるパラメータは, 各時点で共通に用いられる重み $W_{\text{in}}, W, W_{\text{out}}$ である．

再帰型ニューラルネットワークを数式で表現しよう．T 時点からなる入出力の組を $\{(\boldsymbol{t}_t, \boldsymbol{x}_t) \in \mathbb{R}^m \times \mathbb{R}^d \mid t = 1, \dots, T\}$ とする．実際の観測値は $\boldsymbol{y}_t$ ではなく, $\boldsymbol{t}_t$ としていることに注意しよう．中間層は q 個のユニットからなる, つまり $\boldsymbol{z}_t \in \mathbb{R}^q$ とし, 中間層への活性化関数を f, 出力層 $\boldsymbol{y}_t$ への活性化関数を g とする．オートエンコーダの場合と同様に, $\boldsymbol{x}_t$ や $\boldsymbol{z}_t$ がバイアス項を含む場合には, $\boldsymbol{x}_t^\dagger = (1, \boldsymbol{x}_t^\top)^\top, \boldsymbol{z}_t^\dagger = (1, \boldsymbol{z}_t^\top)^\top$ と表すことにする．このとき, 各時刻において, 順伝播は

$$\begin{aligned} \boldsymbol{z}_t &= f(W_{\text{in}} \boldsymbol{x}_t^\dagger + W \boldsymbol{z}_{t-1}) \\ \boldsymbol{y}_t &= g(W_{\text{out}} \boldsymbol{z}_t^\dagger), \qquad t = 1, \dots, T \end{aligned}$$

となる．ここで, $W_{\text{in}} \in \mathbb{R}^{q \times (d+1)}$ および $W_{\text{out}} \in \mathbb{R}^{m \times (q+1)}$ はバイアス項を含み, $W \in \mathbb{R}^{q \times q}$ はバイアス項を含まないものとする．また, $\boldsymbol{z}_T = \boldsymbol{0} \in \mathbb{R}^q$ である．これまでと同様に, 回帰問題を考える場合, 誤差関数 E は二乗誤差

$$E(W_{\text{in}}, W, W_{\text{out}}) = \frac{1}{2} \sum_{t=1}^{T} \|\boldsymbol{t}_t - \boldsymbol{y}_t\|^2 = \frac{1}{2} \sum_{t=1}^{T} \|\boldsymbol{t}_t - g(W_{\text{out}} \boldsymbol{z}_t^\dagger)\|^2$$

を用いる．ただし, この場合 g は恒等写像である．また, 分類問題の場合, g としてソフトマックス関数を用い, 誤差関数はクロスエントロピー

$$E(W_{\text{in}}, W, W_{\text{out}}) = -\sum_{t=1}^{T} \sum_{j=1}^{m} t_{tj} \log y_{tj} = -\sum_{t=1}^{T} \sum_{j=1}^{m} t_{tj} \log g(\boldsymbol{w}_{\text{out},j}^\top \boldsymbol{z}_t^\dagger)$$

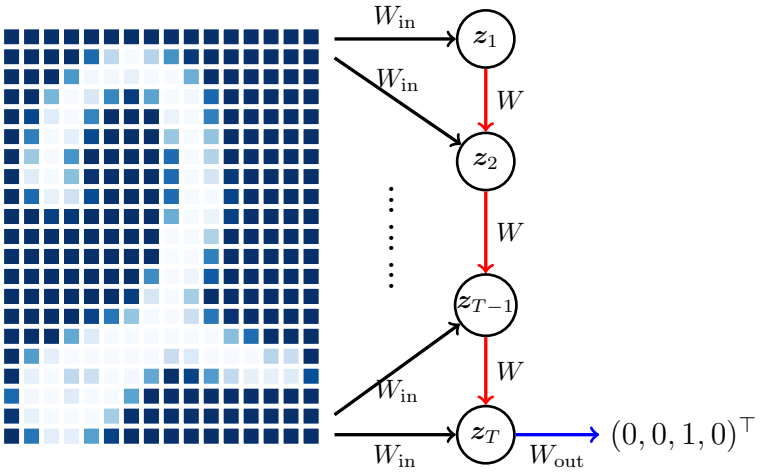


図 19　画像の分類における再帰型ニューラルネットワークのイメージ．many to many 型のネットワークとは異なり，many to one 型のネットワークでは全ての画像をスキャンした後に，モデルの出力が得られることに注意する．

とする．

中間層間の重み W がバイアス項を含まない理由について簡単に説明しておく．中間層の出力は線形結合 $W_{\text{in}}\boldsymbol{x}_t^\dagger + W\boldsymbol{z}_{t-1}$ に活性化関数を通したものである．ここで，もし W がバイアス項を含むとすると，活性化関数 f の引数の第 j 成分は

$$
\begin{aligned}
&\boldsymbol{w}_{\text{in},j}^\top \boldsymbol{x}^\dagger + \boldsymbol{w}_j^\top \boldsymbol{z}_{t-1}^\dagger \\
&= (w_{\text{in},j0} + w_{\text{in},j1}x_1 + \cdots + w_{\text{in},jd}x_d) \\
&= w_{\text{in},j0} + w_{j0} + (w_{\text{in},j1}x_1 + \cdots + w_{\text{in},jd}x_d) + (w_{j1}w_1 + \cdots + w_{j,t-1}z_{q,t-1})
\end{aligned}
$$

となり，$\boldsymbol{w}_{\text{in},j}$ と $\boldsymbol{w}_j$ の切片項が加法的に $w_{\text{in},j0} + w_{j0}$ 現れてしまう．したがって，$w_{\text{in},j0}$ および w_{j0} がパラメータ推定の際に区別できなくなってしまう[*71]．言い換えれば，線形結合 $\boldsymbol{w}_{\text{in},j}^\top \boldsymbol{x}^\dagger + \boldsymbol{w}_j^\top \boldsymbol{z}_{t-1}^\dagger$ におけるバイアス項を $\tilde{w} = w_{\text{in},j0} + w_{j0}$ としてしまえば，バイアス項は一つで良いということになってしまうのである．そのため，通常は W_{in} の方にバイアス項を含ませることが多い．

図 18 のように，時点ごとに出力の予測値が得られるネットワークを **many to many** 型のネットワークと呼ばれる．例えば，株価の予測などのように，1 日ごとに終わり値を将来にわたって予測する場合には，1 日ごとの終わり値の予測値が日ごとに得られるので，これは many to many 型のネットワークとして表現できる．また，テキスト翻訳などでは，日本語の入力に対して，英語の翻訳結果を予測する場合，日本語の単語それぞれに対して，英単語を出力するので，やはりこれも many to many 型のネットワークである．

画像の分類における再帰型ニューラルネットワークでは，画像の各行が $\boldsymbol{x}_t \in \mathbb{R}^d$ に対応する．ところで，このネットワークでは，各時点ごとに出力を計算するわけではなく，すべての行を順伝播した後に初めてラベルの予測値が得られる．そのため，d 個の入力から一つの出力が得られるので，**many to one** 型のニューラルネットワークとも呼ばれる (図 19)．したがって，many to many 型のネットワークとは異なり．many to one 型のネットワークにおける順伝播は

$$
\begin{aligned}
\boldsymbol{z}_t &= f(W_{\text{in}}\boldsymbol{x}_t^\dagger + W\boldsymbol{z}_{t-1}), \qquad t = 1, \ldots, T \\
\boldsymbol{y}_T &= g(W_{\text{out}}\boldsymbol{z}_T^\dagger)
\end{aligned}
$$

[*71] つまり，バイアス項の推定値がどちらのパラメータのものか区別できない．これを**識別性**がないということもある．

となる．また, 誤差関数は many to many 型のネットワークと同様に, 回帰問題ならば

$$E(W_{\text{in}}, W, W_{\text{out}}) = \frac{1}{2}\|\boldsymbol{t}_T - \boldsymbol{y}_T\|^2 = \frac{1}{2}\|\boldsymbol{t}_T - g(W_{\text{out}}\boldsymbol{z}_T^\dagger)\|^2,$$

分類問題ならば

$$E(W_{\text{in}}, W, W_{\text{out}}) = -\sum_{j=1}^{m} t_{tj} \log y_{tj} = -\sum_{j=1}^{m} t_{tj} \log g(\boldsymbol{w}_{\text{out},j}^\top \boldsymbol{z}_t^\dagger)$$

を用いるものとする．ところで, many to many 型のネットワークにおいて, $\boldsymbol{t}_t = \boldsymbol{y}_t = \boldsymbol{0}$ $(t = 1, \ldots, T-1)$ としてみよう．このとき, 誤差関数は, many to one 型のネットワークの誤差関数と一致する．実際, 回帰問題の場合は明らかであり, 分類問題の場合は $0 \times \log 0 = 0$ と約束すれば, 二つの誤差関数は一致することがわかる．したがって, many to one 型のネットワークは many to many 型のネットワークに含まれる．

■誤差逆伝播法　パラメータ $W_{\text{in}} \in \mathbb{R}^{q\times(d+1)}, W \in \mathbb{R}^{q\times q}, W_{\text{out}} \in \mathbb{R}^{m\times(q+1)}$ の更新式を導出するための誤差逆伝播法について説明する．すでに述べたように, many to one 型のネットワークは many to many 型のネットワークに含まれるため, より一般的な many to many 型のネットワークにおける誤差を計算する．many to one 型のネットワークの誤差逆伝播を計算したければ, 以下に述べる結果において $\boldsymbol{t}_t = \boldsymbol{y}_t = \boldsymbol{0}$ $(t = 1, \ldots, T-1)$ とすれば良い．

さて, 誤差関数は回帰問題であれば

$$E(W_{\text{in}}, W, W_{\text{out}}) = \frac{1}{2}\sum_{t=1}^{T} \|\boldsymbol{t}_t - g(W_{\text{out}}\boldsymbol{z}_t^\dagger)\|^2,$$

分類問題であれば

$$E(W_{\text{in}}, W, W_{\text{out}}) = -\sum_{t=1}^{T}\sum_{j=1}^{m} t_{tj} \log g(\boldsymbol{w}_{\text{out},j}^\top \boldsymbol{z}_t^\dagger)$$

だった．ただし, 回帰問題では g は恒等写像であり, 分類問題の場合 g はソフトマックス関数である．

各 t に対して, $\boldsymbol{u}_t = W_{\text{in}}\boldsymbol{x}_t^\dagger + W\boldsymbol{z}_{t-1}, \boldsymbol{v}_t = W_{\text{out}}\boldsymbol{z}_t^\dagger$ としよう．ただし, $\boldsymbol{z}_0 = \boldsymbol{0} \in \mathbb{R}^q$ である．このとき, 順伝播は簡単に

$$\boldsymbol{z}_t = f(\boldsymbol{u}_t), \qquad \boldsymbol{y}_t = g(\boldsymbol{v}_t), \quad t = 1, \ldots, T$$

となる．まず, 合成関数の微分を利用すれば, 回帰問題, 分類問題のいずれの場合でも

$$\frac{\partial E(W_{\text{in}}, W, W_{\text{out}})}{\partial W_{\text{out}}} = \sum_{t=1}^{T} \frac{\partial E(W_{\text{in}}, W, W_{\text{out}})}{\partial \boldsymbol{v}_t} \frac{\partial \boldsymbol{v}_t}{\partial W_{\text{out}}} = \sum_{t=1}^{T} (g(\boldsymbol{v}_t) - \boldsymbol{t}_t)\boldsymbol{z}_t^{\dagger\top}$$

となることがわかる．$\boldsymbol{\delta}_{\text{out},t} = g(\boldsymbol{v}_t) - \boldsymbol{t}_t$ とし, $D_{\text{out}} = (\boldsymbol{\delta}_{\text{out},1}, \ldots, \boldsymbol{\delta}_{\text{out},T}) \in \mathbb{R}^{m\times T}$ とする．さらに,

$$Z^\dagger = \begin{pmatrix} 1 & \boldsymbol{z}_1^\top \\ \vdots & \vdots \\ 1 & \boldsymbol{z}_T^\top \end{pmatrix} = (\boldsymbol{1}, Z) \in \mathbb{R}^{T\times(q+1)}$$

とする．Z は中間層の出力であり, したがって, $Z^\dagger$ は中間層のすべての出力にバイアス項を含めた行列である[*72]．すると, 誤差関数をパラメータ W_{out} で微分したものは

$$\frac{\partial E(W_{\text{in}}, W, W_{\text{out}})}{\partial W_{\text{out}}} = \sum_{t=1}^{T} (g(\boldsymbol{v}_t) - \boldsymbol{t}_t)\boldsymbol{z}_t^{\dagger\top} = D_{\text{out}} Z^\dagger$$

[*72] $\boldsymbol{1} \in \mathbb{R}^T$ は 1 を T 個並べたベクトル.

となることがわかる．なお, many to one 型ネットワークの場合は $\boldsymbol{t}_t = \boldsymbol{y}_t = \boldsymbol{0}$ $(t = 1, \ldots, T-1)$ とすれば良いから,

$$\frac{\partial E(W_{\mathrm{in}}, W, W_{\mathrm{out}})}{\partial W_{\mathrm{out}}} = (g(\boldsymbol{v}_T) - \boldsymbol{t}_T)\boldsymbol{z}_T^{\dagger\top}$$

である．したがって, この場合 $\boldsymbol{\delta}_t = \boldsymbol{0}$ $(t = 1, \ldots, T-1)$ と思って差し支えない．

次に, 誤差関数の W での微分を考えよう．まず,

$$\frac{\partial E(W_{\mathrm{in}}, W, W_{\mathrm{out}})}{\partial W} = \sum_{t=1}^{T} \frac{\partial E(W_{\mathrm{in}}, W, W_{\mathrm{out}})}{\partial \boldsymbol{u}_t} \frac{\partial \boldsymbol{u}_t}{\partial W}$$

であり, $\boldsymbol{u}_t$ の定義から, 各 $t = 1, \ldots, T$ に対して

$$\frac{\partial \boldsymbol{u}_t}{\partial W} = \boldsymbol{z}_{t-1} \in \mathbb{R}^q$$

である．

$$\boldsymbol{\delta}_t = \frac{\partial E(W_{\mathrm{in}}, W, W_{\mathrm{out}})}{\partial \boldsymbol{u}_t}$$

として, $\boldsymbol{\delta}_t$ を計算する．$\boldsymbol{z}_t$ からの順伝播が, t 時点目の出力層と $t+1$ 時点目の中間層 $\boldsymbol{z}_{t+1}$ に流れることに注意すれば, 再び合成関数の微分を用いて,

$$\frac{\partial E(W_{\mathrm{in}}, W, W_{\mathrm{out}})}{\partial \boldsymbol{u}_t} = \left(\frac{\partial \boldsymbol{u}_{t+1}}{\partial \boldsymbol{u}_t}\right)^{\top} \frac{\partial E(W_{\mathrm{in}}, W, W_{\mathrm{out}})}{\partial \boldsymbol{u}_{t+1}} + \left(\frac{\partial \boldsymbol{v}_t}{\partial \boldsymbol{u}_t}\right)^{\top} \frac{\partial E(W_{\mathrm{in}}, W, W_{\mathrm{out}})}{\partial \boldsymbol{v}_t}$$

が得られる[*73]．なお,

$$\frac{\partial E(W_{\mathrm{in}}, W, W_{\mathrm{out}})}{\partial \boldsymbol{u}_{t+1}} = \boldsymbol{\delta}_{t+1}, \qquad \frac{\partial E(W_{\mathrm{in}}, W, W_{\mathrm{out}})}{\partial \boldsymbol{v}_t} = \boldsymbol{\delta}_{\mathrm{out},t}$$

であることに注意する．$\boldsymbol{u}_{t+1} = W_{\mathrm{in}}\boldsymbol{x}_{t+1}^{\dagger} + W\boldsymbol{z}_t = W_{\mathrm{in}}\boldsymbol{x}_{t+1}^{\dagger} + Wf(\boldsymbol{u}_t)$ なので,

$$\frac{\partial \boldsymbol{u}_{t+1}}{\partial \boldsymbol{u}_t} = \frac{\partial}{\partial \boldsymbol{u}_t}(W_{\mathrm{in}}\boldsymbol{x}_{t+1}^{\dagger} + Wf(\boldsymbol{u}_t)) = W\frac{\partial f(\boldsymbol{u}_t)}{\partial \boldsymbol{u}_t} = W\mathrm{diag}(\nabla f(\boldsymbol{u}_t))$$

となる．ただし, ベクトル $\boldsymbol{v} \in \mathbb{R}^q$ に対して, $\mathrm{diag}(\boldsymbol{v})$ は $\boldsymbol{v}$ の各成分を対角要素とする $q \times q$ 次元の対角行列である．また, $W_{\mathrm{out}} = (\boldsymbol{w}_{\mathrm{out},0}, \boldsymbol{w}_{\mathrm{out},1}, \ldots, \boldsymbol{w}_{\mathrm{out},q}) = (\boldsymbol{w}_{\mathrm{out},0}, \tilde{W}_{\mathrm{out}})$ とすれば,

$$\begin{aligned}\frac{\partial \boldsymbol{v}_t}{\partial \boldsymbol{u}_t} &= \frac{\partial W_{\mathrm{out}}\boldsymbol{z}_t^{\dagger}}{\partial \boldsymbol{u}_t} = \frac{\partial}{\partial \boldsymbol{u}_t}\left(\boldsymbol{w}_{\mathrm{out},0} + \tilde{W}_{\mathrm{out}}\boldsymbol{z}_t\right) \\ &= \frac{\partial}{\partial \boldsymbol{u}_t}\left(\boldsymbol{w}_{\mathrm{out},0} + \tilde{W}_{\mathrm{out}}f(\boldsymbol{u}_t))\right) = \tilde{W}_{\mathrm{out}}\mathrm{diag}(\nabla f(\boldsymbol{u}_t))\end{aligned}$$

が得られる．以上より,

$$\begin{aligned}\frac{\partial E(W_{\mathrm{in}}, W, W_{\mathrm{out}})}{\partial \boldsymbol{u}_t} &= \{W\mathrm{diag}(\nabla f(\boldsymbol{u}_t))\}^{\top}\boldsymbol{\delta}_{t+1} + \{\tilde{W}_{\mathrm{out}}\mathrm{diag}(\nabla f(\boldsymbol{u}_t))\}^{\top}\boldsymbol{\delta}_{\mathrm{out},t} \\ &= \mathrm{diag}(\nabla f(\boldsymbol{u}_t))(W^{\top}\boldsymbol{\delta}_{t+1} + \tilde{W}_{\mathrm{out}}^{\top}\boldsymbol{\delta}_{\mathrm{out},t})\end{aligned}$$

となるが, 適当なサイズの対角行列 $\mathrm{diag}(\boldsymbol{v})$ とベクトル $\boldsymbol{u}$ に対して $\mathrm{diag}(\boldsymbol{v})\boldsymbol{u} = \boldsymbol{v} \odot \boldsymbol{u}$ $(= \boldsymbol{u} \odot \boldsymbol{v})$ が成り立つので, 結局,

$$\boldsymbol{\delta}_t = (W^{\top}\boldsymbol{\delta}_{t+1} + \tilde{W}_{\mathrm{out}}^{\top}\boldsymbol{\delta}_{\mathrm{out},t}) \odot \nabla f(\boldsymbol{u}_t)$$

[*73] 興味のある人は, なぜこのような格好で合成関数の微分が計算できるか考えてみてほしい．ヒントはやはり合成関数の微分であり, まずは定義通り成分ごとに計算することである．

となることがわかる．ただし, $\boldsymbol{\delta}_{T+1} = \mathbf{0} \in \mathbb{R}^q$ とする．したがって,

$$\frac{\partial E(W_{\text{in}}, W, W_{\text{out}})}{\partial W} = \sum_{t=1}^{T} \boldsymbol{\delta}_t \boldsymbol{z}_{t-1}$$

となる．W_{out} で誤差関数を微分したときと同様に, $D = (\boldsymbol{\delta}_1, \ldots, \boldsymbol{\delta}_T) \in \mathbb{R}^{q \times T}$, $\tilde{Z} = (\boldsymbol{z}_0, \ldots, \boldsymbol{z}_{T-1})^\top \in \mathbb{R}^{T \times q}$ とする．ここで, $\boldsymbol{z}_0 = \mathbf{0} \in \mathbb{R}^q$ であり, $\tilde{Z}$ は最後の中間層 $\boldsymbol{z}_T$ を含まないことに注意する．すると, 誤差関数をパラメータ W_{out} で微分したものは

$$\frac{\partial E(W_{\text{in}}, W, W_{\text{out}})}{\partial W_{\text{out}}} = D\tilde{Z}$$

となる．

最後に, 誤差関数の W_{in} での微分を求めよう．この計算は簡単で, $\boldsymbol{u}_{\text{in},t} = W_{\text{in}} \boldsymbol{x}_t^\dagger$ としたとき,

$$\frac{\partial E(W_{\text{in}}, W, W_{\text{out}})}{\partial \boldsymbol{u}_{\text{in},t}} = \frac{\partial E(W_{\text{in}}, W, W_{\text{out}})}{\partial \boldsymbol{u}_t} \frac{\partial \boldsymbol{u}_t}{\partial \boldsymbol{u}_{\text{in},t}} = \frac{\partial E(W_{\text{in}}, W, W_{\text{out}})}{\partial \boldsymbol{u}_t} = \boldsymbol{\delta}_t$$

に注意すれば良い．つまり, 入力層の誤差はすでに計算した $\boldsymbol{\delta}_t$ を使い回すことができるということである．したがって,

$$X^\dagger = \begin{pmatrix} 1 & \boldsymbol{x}_1^\top \\ \vdots & \vdots \\ 1 & \boldsymbol{x}_T^\top \end{pmatrix} = (\mathbf{1}, X) \in \mathbb{R}^{T \times (d+1)}$$

とすれば,

$$\frac{\partial E(W_{\text{in}}, W, W_{\text{out}})}{\partial W_{\text{in}}} = \sum_{t=1}^{T} \frac{\partial E(W_{\text{in}}, W, W_{\text{out}})}{\partial \boldsymbol{u}_{\text{in},t}} \frac{\partial \boldsymbol{u}_{\text{in},t}}{\partial W_{\text{in}}} = \sum_{t=1}^{T} \boldsymbol{\delta}_t \boldsymbol{x}_t^\dagger = DX^\dagger$$

となる．以上をまとめると, 再帰型ニューラルネットワークのアルゴリズムは Algorithm 5 のようになる．初めに述べたように, モデルとしてはやや複雑に見えるが, やることはこれまでとほぼ同じである．なお, 実装の際に逆伝播やパラメータの更新を行う場合, 中間層の過去の値や誤差の微分をすべて用いるので, D や Z の各行にこれらを保存するようにしておくと良い．また, 深層ニューラルネットワークとは異なり, ユーザーが中間層の数を事前に決めることはできないので, これまでよりも for 文の使用回数が増える．

3.4.2 長・短期記憶

再帰型ニューラルネットワークは, その構造上中間層の数が増えてしまい, 勾配消失や**勾配爆発**といった問題が起こりやすい．これを見るため, 現在の誤差 $\boldsymbol{\delta}_t$ が q 時点過去の誤差 $\boldsymbol{\delta}_{t-q}$ にどの程度影響するか見てみよう．$\boldsymbol{\delta}_t$ の定義より, $\boldsymbol{u}_t = W_{\text{in}} \boldsymbol{x}_t^\dagger + W\boldsymbol{z}_{t-1}$ とすれば,

$$\begin{aligned} \boldsymbol{\delta}_{t-1} &= (W^\top \boldsymbol{\delta}_t + \tilde{W}_{\text{out}}^\top \boldsymbol{\delta}_{\text{out},t-1}) \odot \nabla f(\boldsymbol{u}_{t-1}) \\ &= W^\top \boldsymbol{\delta}_t \odot \nabla f(\boldsymbol{u}_{t-1}) + \tilde{W}_{\text{out}}^\top \boldsymbol{\delta}_{\text{out},t-1} \odot \nabla f(\boldsymbol{u}_{t-1}) \end{aligned}$$

である．次に, $q = 2$ では,

$$\begin{aligned} \boldsymbol{\delta}_{t-2} &= W^\top \boldsymbol{\delta}_{t-1} \odot \nabla f(\boldsymbol{u}_{t-2}) + \tilde{W}_{\text{out}}^\top \boldsymbol{\delta}_{\text{out},t-2} \odot \nabla f(\boldsymbol{u}_{t-2}) \\ &= (W^\top)^2 \boldsymbol{\delta}_t \bigodot_{j=1}^{2} \nabla f(\boldsymbol{u}_{t-j}) + \sum_{k=1}^{2} (W^\top)^{2-k} \tilde{W}_{\text{out}}^\top \boldsymbol{\delta}_{\text{out},t-k} \bigodot_{j=k}^{2} \nabla f(\boldsymbol{u}_{t-j}) \end{aligned}$$

となる．ただし, 適当な大きさのベクトル $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_n$ に対して, $\bigodot_{j=1}^{n} \boldsymbol{v}_j = \boldsymbol{v}_1 \odot \cdots \odot \boldsymbol{v}_n$ である．この操作を繰り返すことで, $t - q$ 時点目での誤差 $\boldsymbol{\delta}_{t-q}$ は

$$\boldsymbol{\delta}_{t-q} = (W^\top)^q \boldsymbol{\delta}_t \bigodot_{j=1}^{q} \nabla f(\boldsymbol{u}_{t-j}) + \sum_{k=1}^{q} (W^\top)^{q-k} \tilde{W}_{\text{out}}^\top \boldsymbol{\delta}_{\text{out},t-k} \bigodot_{j=k}^{q} \nabla f(\boldsymbol{u}_{t-j})$$

Algorithm 5 再帰型ニューラルネットワークの誤差逆伝播法

入力: データ $\{(T_i, X_i) \in \mathbb{R}^{m\times T} \times \mathbb{R}^{d\times T} \mid i = 1, \ldots, n\}$, 学習率 η_0, 中間層のユニット数 q, 活性化関数 f, g, 誤差関数 E, エポック数 N ▷ $T_i = (\boldsymbol{t}_1, \ldots, \boldsymbol{t}_T)$

パラメータ $W_{\text{in}}, W, W_{\text{out}}$ の初期化
for $e = 1$ to N **do**
  $\mathcal{I} = \{$ ランダムシャッフル後のデータのインデックス $\}$
  $\eta_e \leftarrow \eta_0/e$ ▷ 学習率の更新
  for i in $\mathcal{I}$ **do**
    for $t = 1$ to T **do**
      $\boldsymbol{z}_t \leftarrow f(W_{\text{in}}\boldsymbol{x}_t^{\dagger} + W\boldsymbol{z}_{t-1});\ \boldsymbol{y}_t \leftarrow g(W_{\text{out}}\boldsymbol{z}_t^{\dagger})$
    end for ▷ 順伝播: $\boldsymbol{z}_0 = \boldsymbol{0}$
    for $t = T$ to 1 **do**
      $\boldsymbol{\delta}_{\text{out},t} \leftarrow g(W_{\text{out}}\boldsymbol{z}_t^{\dagger}) - \boldsymbol{t}_t;\ \boldsymbol{\delta}_t \leftarrow (W^{\top}\boldsymbol{\delta}_{t+1} + \tilde{W}_{\text{out}}^{\top}\boldsymbol{\delta}_{\text{out},t}) \odot \nabla f(\boldsymbol{u}_t)$
    end for ▷ 逆伝播: $\boldsymbol{\delta}_{T+1} = \boldsymbol{0}$
    $W_{\text{in}} \leftarrow W_{\text{in}} - \eta_e D X^{\dagger}$
    $W \leftarrow W - \eta_e D \tilde{Z}$
    $W_{\text{out}} \leftarrow W_{\text{out}} - \eta_e D_{\text{out}} Z^{\dagger}$ ▷ パラメータの更新
  end for
end for

出力: パラメータ $W_{\text{in}}, W, W_{\text{out}}$ の推定値

となることがわかる. したがって, すべての j で誤差 $\nabla f(\boldsymbol{u}_{t-j})$ の各成分の絶対値が 1 よりも大きい場合, $\bigodot_{j=1}^{q} \nabla f(\boldsymbol{u}_{t-j})$ の各成分は層を重ねるごとにだんだんと大きくなり, 結果として, $t-q$ ステップ目の誤差 $\boldsymbol{\delta}_{t-q}$ もそれに応じて大きくなってしまう. これが, 勾配発散の原因である. 一方, 誤差 $\nabla f(\boldsymbol{u}_{t-j})$ の各成分の絶対値が 1 よりも小さい場合には, $\bigodot_{j=1}^{q} \nabla f(\boldsymbol{u}_{t-j}) \approx \boldsymbol{0}$ となってしまい, $\boldsymbol{\delta}_t$ の影響が $\boldsymbol{\delta}_{t-q}$ に伝わらなくなってしまい, 勾配消失の問題が生じることになる. 例えば, f としてシグモイド関数を用いた場合, その微分は $\nabla f = f(1-f)$ となるが, $f \in [0, 1]$ であることに注意すると, $\nabla f \leq 1/4$ となる. したがって, シグモイド関数を用いたニューラルネットワークでは, 本質的に勾配消失の問題を抱えていることになる.

では, どのように勾配消失 (発散) の問題を解消すれば良いだろうか. 簡単のため, 逆伝播は適当な行列 $\check{W}$ と $\check{V}$ を用いて

$$\boldsymbol{\delta}_t = \check{W}^{\top}\boldsymbol{\delta}_{t+1} \odot \nabla f(\boldsymbol{u}_t)$$

と表されるものとしよう. ただし, $\boldsymbol{u}_t = \check{V}\boldsymbol{x}_t$ である. したがって, $\boldsymbol{\delta}_t$ の第 j 成分は

$$\delta_{t,j} = \nabla f(u_{t,j}) \sum_i \check{w}_{ij}\delta_{t+1,i} = \nabla f(u_{t,j})\check{w}_{jj}\delta_{t+1,j} + \nabla f(u_{t,j}) \sum_{i\neq j} \check{w}_{ij}\delta_{t+1,i}$$

で与えられる. 誤差 $\delta_{t+1,j}$ の情報が正しく $\delta_{t,j}$ に伝わるためには,

$$\nabla f(u_{t,j})\check{w}_{jj} = 1$$

であれば良い. そうでなければ, 勾配消失 (発散) が起こってしまうためである. このような活性化関数の性質を調べるため, 両辺を $u_{t,j}$ で積分すると,

$$f(u_{t,j}) = \frac{u_{t,j}}{\check{w}_{jj}} + C$$

となることがわかる[*74]. したがって, f は線形写像 (あるいは Affine 写像) であれば良いということになる. このとき,

$$f(u_{t+1,j}) = \check{\boldsymbol{v}}_j^\top \boldsymbol{x}_t$$

となり, 順伝播で, 定数項 $\check{\boldsymbol{v}}_j^\top \boldsymbol{x}_t$ を考慮するべきであろう. 実際には f として線形写像を用いることはないため, 通常の順伝播とは別に, 内部状態として

$$\boldsymbol{s}_t = \boldsymbol{s}_{t-1} + f(\boldsymbol{u}_t) \odot g(\boldsymbol{v}_t)$$

のような構造を考えることで, 逆伝播の際の勾配消失 (発散) を防ぐことが期待できる. ただし, $\boldsymbol{v}_t$ は中間層とパラメータの線形結合を表している. この考え方をニューラルネットワークに適用したものが**長・短期記憶 (Long-Short Term Memory; LSTM)** である.

LSTM 以降, その拡張がいくつか提案されているが, 最近でもよく利用されるモデルとして, **忘却ゲート付き LSTM** と呼ばれるものがある. 忘却ゲート付き LSTM では, 内部状態を更新する際に, 過去の内部状態をそのまま利用せず,

$$\boldsymbol{s}_t = \boldsymbol{l}_t \odot \boldsymbol{s}_{t-1} + f(\boldsymbol{u}_t) \odot g(\boldsymbol{v}_t)$$

とすることで, $\boldsymbol{s}_{t-1}$ の影響を薄れさせる (忘却させる) ものである. ここで, $\boldsymbol{l}_t \in [0,1]^d$ は忘却率を表すパラメータであり, 特に, $\boldsymbol{l}_{t,j} = 1$ なら, $s_{t-1,j}$ をすべて伝播させ, $\boldsymbol{l}_{t,j} = 0$ なら, $s_{t-1,j}$ を次の時点で考慮しないというわけである. 忘却のためのユニット $\boldsymbol{s}_t$ を考慮してネットワークを順伝播するために, まず通常の再帰型ニューラルネットワークの順伝播が

$$\begin{aligned}\boldsymbol{z}_t &= f(W_{\mathrm{in}}\boldsymbol{x}_t^\dagger + W\boldsymbol{z}_{t-1})\\ \boldsymbol{y}_t &= g(W_{\mathrm{out}}\boldsymbol{z}_t^\dagger), \qquad t = 1, \dots, T\end{aligned}$$

で与えられたことを思い出そう. 以下, LSTM で出力されるものが, 中間層 $\boldsymbol{z}_t$ および忘却のためのユニット $\boldsymbol{s}_t$ であることに注意し, LSTM の構造上 $\boldsymbol{z}_t$ と $\boldsymbol{s}_t$ のユニット数は等しく q であるとする. また, LSTM の内部で利用する活性化関数を f, h としておく. LSTM では, その構成要素を定義するための関数を**ゲート**とよび, 入力ゲートの出力は

$$\boldsymbol{i}_t = f(W_{\mathrm{i}}\boldsymbol{x}_t^\dagger + V_{\mathrm{i}}\boldsymbol{z}_{t-1} + U_{\mathrm{i}}\boldsymbol{s}_{t-1})$$

として伝播される. ただし, $\boldsymbol{z}_0 = \boldsymbol{s}_0 = \boldsymbol{0} \in \mathbb{R}^q$ であり, $W_{\mathrm{i}} \in \mathbb{R}^{q\times(d+1)}, V_{\mathrm{i}} \in \mathbb{R}^{q\times q}, U_{\mathrm{i}} \in \mathbb{R}^{q\times q}$ である. また, 忘却のためのユニットを計算するための忘却ゲートも入力ゲートと同様に

$$\boldsymbol{l}_t = f(W_{\mathrm{l}}\boldsymbol{x}_t^\dagger + V_{\mathrm{l}}\boldsymbol{z}_{t-1} + U_{\mathrm{l}}\boldsymbol{s}_{t-1})$$

とする. ただし, $W_{\mathrm{l}} \in \mathbb{R}^{q\times(d+1)}, V_{\mathrm{l}} \in \mathbb{R}^{q\times q}, U_{\mathrm{l}} \in \mathbb{R}^{q\times q}$ である. 通常の再帰型ニューラルネットワークにおける中間層への伝播が

$$W\boldsymbol{x}_t^\dagger + V\boldsymbol{z}_{t-1}$$

に活性化関数を通したもので与えられることに注意して, LSTM の内部状態 $\boldsymbol{s}_t$ を

$$\boldsymbol{s}_t = \boldsymbol{l}_t \odot \boldsymbol{s}_{t-1} + \boldsymbol{i}_t \odot g(W\boldsymbol{x}_t^\dagger + V\boldsymbol{z}_{t-1})$$

により更新する. なお, $\boldsymbol{s}_t$ はメモリセルとも呼ばれ, LSTM の由来となっている. 最後に, パラメータ $W_{\mathrm{o}} \in \mathbb{R}^{q\times(d+1)}, V_{\mathrm{o}} \in \mathbb{R}^{q\times q}, U_{\mathrm{o}} \in \mathbb{R}^{q\times q}$ を用いて, 出力ゲートへの伝播

$$\boldsymbol{o}_t = f(W_{\mathrm{o}}\boldsymbol{x}_t^\dagger + V_{\mathrm{o}}\boldsymbol{z}_{t-1} + U_{\mathrm{o}}\boldsymbol{s}_{t-1})$$

とし, LSTM の出力を

$$\boldsymbol{z}_t = \boldsymbol{o}_t \odot g(\boldsymbol{s}_t)$$

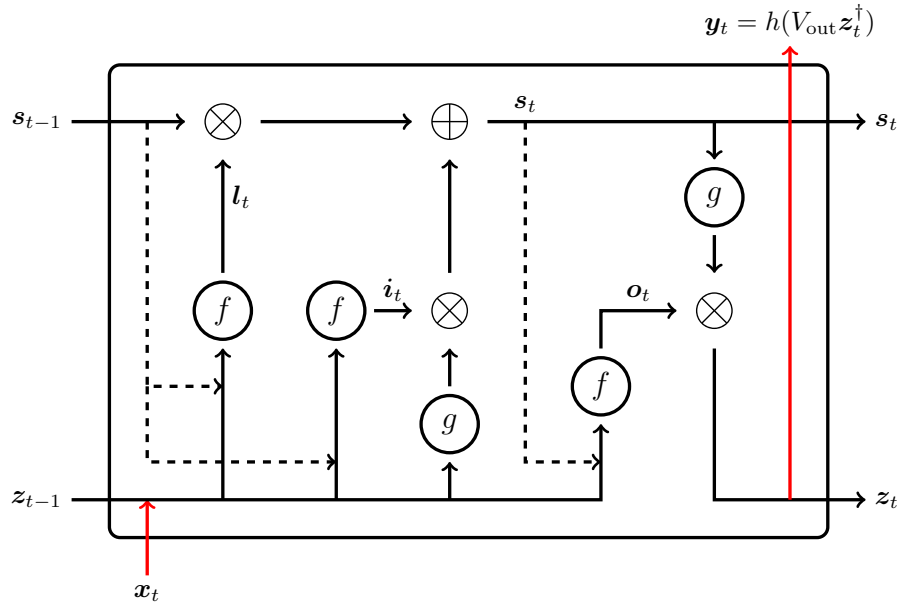


図 20　LSTM による順伝播の模式図. $\otimes$ はネットワーク内の出力に対して Hadamard 積をとることを表し, $\oplus$ は成分ごとの和を表している.

とする.

LSTM の順伝播を模式的に表すと図 20 のようになる. many to many 型のネットワークなど, 必要であれば各時点における出力層は活性化関数 g を用いて $h(V_{\mathrm{out}}\boldsymbol{z}_t^\dagger)$ で順伝播する. LSTM におけるパラメータは $W_{\mathrm{i}}, W_{\mathrm{l}}, W_{\mathrm{o}}, W \in \mathbb{R}^{q\times(d+1)}, V_{\mathrm{i}}, V_{\mathrm{l}}, V_{\mathrm{o}}, V, U_{\mathrm{i}}, U_{\mathrm{l}}, U_{\mathrm{o}} \in \mathbb{R}^{q\times q}$ および $V_{\mathrm{out}} \in \mathbb{R}^{m\times(q+1)}$ であり, 通常の再帰型ニューラルネットワークよりも最適化すべきパラメータが多いため複雑に見えるが, やはり逆伝播を計算することでこれまで通り実装することができる. なお, 通常は f としてシグモイド関数, g としてハイパボリックタンジェント関数が用いられるようである.

3.5　畳み込みニューラルネットワーク

最近の深層学習の発展で特に有名なものの一つは**畳み込みニューラルネットワーク**であろう. 畳み込みニューラルネットワークは, これまで画像や物体認識の分野で大きな成功を収めており, 脳科学の分野では現在の畳み込みニューラルネットワークの初期型とも呼べる**ネオコグニトロン**が 1980 年代に福島邦彦によって提案された.

畳み込みニューラルネットワークは主に**畳み込み層**, **プーリング層**および**全結合層**と呼ばれる層から構成される. 順伝播では, 畳み込みとプーリングを繰り返すことで, 物体の局所的な情報を伝播し, その後, 全結合層を通して最終的な出力層が得られる. 例えば, 1999 年に提案された LeNet では, $d \times d$ の入力画像を m 個のクラスに分類する問題に対して,

$$\begin{aligned}\text{入力層} &\to \text{畳み込み層} \to \text{プーリング層}\\ &\to \text{畳み込み層} \to \text{プーリング層}\\ &\to \text{全結合層} \to \text{全結合層} \to \text{出力層}\end{aligned}$$

というネットワークを用いた (図 3.5). 畳み込み層やプーリング層の組み合わせを考えることで[*75], 様々なネットワークを構成することが可能であり, 最近では 150 層を超えるものも提案されている. なお, 畳み込

[*74] C は積分定数.

[*75] 例えば, “畳み込み → 畳み込み → プーリング” を繰り返せば LeNet とは別のネットワークが得られる.

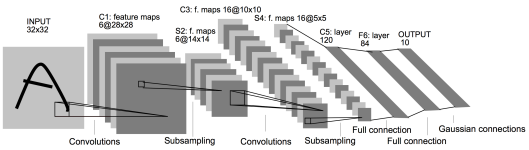


図 21　LeNet のネットワーク表現．図で Convolutions, Subsampling はそれぞれ畳み込み, プーリングを表しており, Full connection は全結合を示している．

み層が何回か繰り返されることはよくあるが, プーリング層が続くようなネットワークはあまり考えられていない．つまり, プーリング層の後には, ほとんどの場合畳み込み層か全結合層が置かれる．

以下では, 畳み込みニューラルネットワークの構成要素である畳み込み層, プーリング層および全結合層の順伝播についてまず説明する．その後, 実際に誤差逆伝播を計算し, パラメータを推定するためのアルゴリズムを導出する．

3.5.1　畳み込みニューラルネットワークの順伝播

畳み込みニューラルネットワークは, 畳み込み層, プーリング層および全結合層からなることはすでに述べた．ところで, RGB のカラー画像など, 通常画像は複数の**チャネル**により表現される．このことを考慮し, 以下では, 入力 X は K 個のチャネルからなる $d \times d$ の画像, つまり $X \in \mathbb{R}^{d \times d \times K}$ であるとする．したがって, 入力 X は K 個の $d \times d$ 行列 $X_1, \ldots, X_K \in \mathbb{R}^{d \times d}$ であり, まとめて $X = (X_1, \ldots, X_K)$ と表すことにする[*76]．また, 画像のフィルタリングを行う場合, 一つの**フィルタ**だけでなく, 複数のフィルタを用いて画像の局所的な特徴を抽出する．どのようなフィルタを用いるかは解析手法に依存するが, データから適合的にフィルタを決定することを考えると, フィルタは推定されるべきパラメータである．フィルタのサイズを $H \times H$ とすると, K チャネルの入力 $X_1, \ldots, X_K$ のそれぞれに対してフィルタを用意することになる．実際には, フィルタリングの出力が 1 枚の行列であることは少なく, 例えば M 個の異なるフィルタリングを行うことになる．したがって, 全部で M 個の $H \times H \times K$ 次元配列 $W_1, \ldots, W_M \in \mathbb{R}^{H \times H \times K}$ がフィルタのパラメータである．これをまとめて, $W = (W_1, \ldots, W_M) \in \mathbb{R}^{H \times H \times K \times M}$ と書くことにする．なお, W_{km} と書けば, さらに, これまでの深層学習のモデルと同様に, バイアス項も考慮することにしよう．バイアス項はフィルタリングごとに作用するものとし, $\boldsymbol{b} = (b_1, \ldots, b_M)^\top \in \mathbb{R}^M$ をバイアスと考える[*77]．

以上の準備のもと, それぞれの層でどのように順伝播を行うか説明する．なお, 以降では色々な添字が現れるので, どの添字がどの次元に対応するか注意しながら読み進めてもらいたい．表記の都合上, バイアス項は別にして説明するが[*78], これまでのアルゴリズムと本質的な差はないことには注意してほしい．

■畳み込み層　畳み込み層の入力は 3 次元配列 $Z = (Z_1, \ldots, Z_K) \in \mathbb{R}^{d \times d \times K}$ であるとする．なお, これ以降, d や K は単に入力 X の次元を表す引数でなく, あくまでも各層での入力の次元を表すものとする．ところで, 畳み込み層で行われる処理は, 画像処理でいうところのフィルタリングであるから, すでに述べたようにパラメータを用いて畳み込み層の出力が得られる．畳み込み層での出力を $Z' = (Z'_1, \ldots, Z'_M) \in \mathbb{R}^{H \times H \times M}$ とすれば, パラメータは 4 次元配列 $W = (W_1, \ldots, W_M) \in \mathbb{R}^{H \times H \times K \times M}$ および, バイアス項 $\boldsymbol{b} = (b_1, \ldots, b_M) \in \mathbb{R}^M$ である．

[*76] やや表記の乱用ではあるが, 行列がスカラーを縦横の 2 次元方向に並べるのに対して, $X = (X_1, \ldots, X_K)$ という表記によって, 行列を奥行きの方向に並べたものだと理解してもらいたい．

[*77] もちろん, チャネル内で共通のバイアス項を考える必要はない．つまり, 各チャネルごとに $B_m = (b_{ijm})_{i,j} \in \mathbb{R}^{H \times H}$ $(m = 1, \ldots, M)$ をバイアス項と考えることもある．この場合, バイアス項は 3 次元配列 $B = (B_1, \ldots, B_M) \in \mathbb{R}^{H \times H \times M}$ となる．

[*78] テンソル積を駆使すれば書けないことはないが表記が煩雑になりすぎて理解を妨げる可能性があるので, ご容赦願いたい．

畳み込み層で最も重要な処理は, 部分画像の線形変換である. はじめに, 畳み込みのイメージを説明しておこう. まず, m 番目チャネルでは,

$$u_{ijm} = \sum_{k=1}^{K} \sum_{p,q=1}^{H} w_{pqkm} z_{i+p-1,j+q-1,k} + b_m$$

という線形変換を行う. ただし, $i, j = 1, \ldots, d - H + 1$ は畳み込み後の行列のサイズである. そして, 活性化関数 f を用いて, $Z'_m = (f(u_{ijm}))_{i,j} \in \mathbb{R}^{H \times H}$ を m 番目の出力 (チャネル) とする. はじめの線形変換の意味について考えよう. まず, $W_{km} = (w_{pqkm}) \in \mathbb{R}^{H \times H}$ を m 番目に出力する (入力の)k チャネル目でのパラメータとし, $Z_{ijk} = (z_{i+p-1,j+q-1,k})_{p,q} \in \mathbb{R}^{H \times H}$ を i, j および k で添字付けられた Z の部分行列であるとしよう. したがって, 各 k に対して, Z_{ijk} は z_{ijk} を $(1,1)$ 成分にもつ $H \times H$ の行列

$$Z_{ijk} = \begin{pmatrix} z_{ijk} & \cdots & z_{i,j+H-1,k} \\ \vdots & \ddots & \vdots \\ z_{i+H-1,j,k} & \cdots & z_{i+H-1,j+H-1,k} \end{pmatrix}$$

である. すると, u_{ijm} は

$$u_{ijm} = \sum_{k=1}^{K} \langle W_{km}, Z_{ijk} \rangle + b_m = \langle W_m, Z_{ij} \rangle + b_m$$

とかける. ただし, $Z_{ij} = (Z_{ij1}, \ldots, Z_{ijK})$ は Z_{ijk} を並べた $H \times H \times K$ 次元の配列である. なお, $\langle \cdot, \cdot \rangle$ は配列の内積を表しており,

$$\langle A, B \rangle = \sum_{i,j} a_{ij} b_{ij} = \mathrm{tr}(A^\top B), \quad A, B \in \mathbb{R}^{H \times H}$$
$$\langle A, B \rangle = \sum_{k} \langle A_k, B_k \rangle = \sum_{k} \sum_{i,j} a_{ijk} b_{ijk}, \quad A, B \in \mathbb{R}^{H \times H \times K}$$

などで定義される[*79]. つまり, 畳み込み層で行われる処理は, これまでと同じように各入力 Z_{ij} とパラメータとの線形結合に活性化関数を作用させたものである. 以下, 内積 $\langle \cdot, \cdot \rangle$ の引数が, 何次元の配列なのかを意識しながら確認してもらいたい. 図 22 は畳み込み層での演算のイメージを表したものである.

ところで, 畳み込み層における和の計算では, 中央部分に対して端の部分があまり考慮されないという問題がある. これは, 例えば Z_k の $(1,1)$ 成分 (つまり, Z の $(1,1,1)$ 成分) が H 回しかスキャンされないのに対し, 中央部分が高々 H^2 回スキャンされることからも納得できる. また, もう少し荒いフィルタリングを行いたい場合には, 上記のようにフィルタ W_{km} を 1 ピクセルずつ移動させずに, 例えば 2 ピクセルずつ移動させた方が推定精度が良いかもしれない. そこで, 畳み込み層では, **パディング**および**ストライディング**という処理も同時に行われることが多く, これらについて述べておく.

パディング: パディングは, Z_k の端の部分が中央部分に比べてスキャンされる回数が少ないという問題を解消するために用いられるものである. 最も簡単なものは, **ゼロパディング**と呼ばれるものであり, Z_k の周りに 0 を並べるものである. つまり, パディング数を α とすれば, ゼロパディングでは $Z_k \in \mathbb{R}^{d \times d}$ の周りに 0 を並べた $(d + 2\alpha) \times (d + 2\alpha)$ の画像を処理することになる. 例えば, 次のような Z_k をパディング数 1 でゼロパディングすると,

$$Z_k = \begin{pmatrix} 1 & \cdots & 1 \\ \vdots & \ddots & \vdots \\ 1 & \cdots & 1 \end{pmatrix} \mapsto \begin{pmatrix} 0 & 0 & \cdots & 0 & 0 \\ 0 & 1 & \cdots & 1 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 1 & \cdots & 1 & 0 \\ 0 & 0 & \cdots & 0 & 0 \end{pmatrix}$$

[*79] $\langle A, B \rangle$ は python だと `np.sum(A*B)` で実行できる. この演算は A, B の Hadamard 積 $A \odot B$ の各成分の和を計算することと同じである.

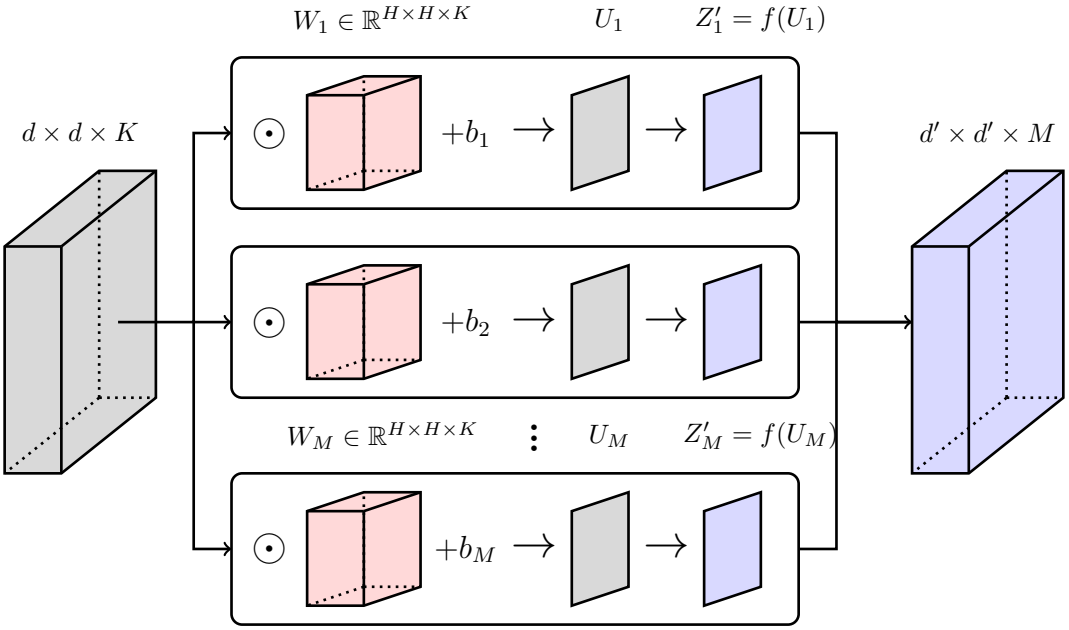


図 22　畳み込み層で行われる処理のイメージ．ただし, d' は畳み込み後の行列のサイズを表している．

となる．当然ではあるが, Z_k の周りを 0 で埋めることの正当性はなく, Z_k の周期性を考慮したものなどを始め, いろいろなパディング方法が提案されている．ただし, このノートでは, 説明を簡単にするために, Z_k の要素を利用したパディングは考えない．つまり, 元の Z_k に対して, パディングで拡大される部分は定数, 特に 0 であるとし, ゼロパディングのみを考えることにする．

ストライド:　ストライドでは, Z_k を 1 ピクセルずつスキャンせず, 複数個のピクセルごとにスキャンする, いわば複数段飛ばしの畳み込み方法である．パディングは畳み込み処理における u_{ijm} を計算する前に行っておく処理であるのに対し, ストライドは u_{ijm} の計算に直接関わるものである．つまり, ストライド数を β とすると, 畳み込み層における u_{ijm} の計算は次のようになる．

$$u_{ijm} = \sum_{k=1}^{K} \sum_{p,q=1}^{H} w_{pqkm} z_{\beta(i-1)+p,\beta(j-1)+q,k} + b_m$$

ただし, ストライドによって, 畳み込み後の $Z_k \in \mathbb{R}^{d\times d}$ は $(\frac{d-H}{\beta}+1) \times (\frac{d-H}{\beta}+1)$ であり, したがって, ストライド数 β は $d-H$ の約数としなければならないことに注意する[*80]．なお,

$$\tilde{Z}_{ijk} = \begin{pmatrix} z_{\beta(i-1)+1,\beta(j-1)+1,k} & \cdots & z_{i,\beta(j-1)+H,k} \\ \vdots & \ddots & \vdots \\ z_{\beta(i-1)+H,j,k} & \cdots & z_{\beta(i-1)+H,\beta(j-1)+H,k} \end{pmatrix}$$

とすれば, u_{ijm} は

$$u_{ijm} = \langle W_m, \tilde{Z}_{ij} \rangle + b_m$$

となり, u_{ijm} からなる行列に活性化関数 f を作用させたものが Z' となる．

パディングとストライドを考慮した畳み込み層の出力は Algorithm 6 の通りである．

[*80] “しなければならない”というのは実は言い過ぎで, β が $d-H$ の約数でなくても畳み込み層を計算することはできる．

Algorithm 6 畳み込み層

入力: 入力 $Z \in \mathbb{R}^{d\times d\times K}$, パラメータ $W \in \mathbb{R}^{H\times H\times K\times M}$, $\boldsymbol{b} \in \mathbb{R}^M$, パディング数 α, ストライド数 β, 活性化関数 f

$d' = (d + 2\alpha - H)/\beta + 1$ ▷ β は $d + 2\alpha - H$ の約数

$Z' = \mathbf{0} \in \mathbb{R}^{d'\times d'\times M}$ の初期化 ▷ 畳み込み後のサイズ

for $m = 1$ to M **do**

  for $i = 1$ to d' **do**

    for $j = 1$ to d' **do**

      $u_{ijm} = \langle W_m, \tilde{Z}_{ij}\rangle + b_m$

    end for

  end for

end for

$Z' = f(U)$

出力: 畳み込み層の出力 $Z' \in \mathbb{R}^{d'\times d'\times M}$

■プーリング層 プーリング層では, 画像処理における縮小処理を行う. ストライディングと同様に, プーリングにおいても “これ”といった決定的な手法はなく, いくつかの方法によるプーリングが用いられる.

プーリング層の入力を $Z \in \mathbb{R}^{d\times d\times K}$ としよう. 畳み込み層と同様に, プーリング層でも $d \times d$ の入力 Z_k をより小さな $s \times s$ のフィルタを用いて圧縮される. 畳み込み層との違いは, プーリング層ではパラメータを用いず, (通常は) パディングを行わないことが多いという点である.

ストライド β でプーリングする場合, 畳み込み層と同様に, Z_k の部分行列

$$Z_{ijk} = (z_{\beta(i-1)+p,\beta(j-1)+q,k})_{p,q} = \begin{pmatrix} z_{\beta(i-1)+1,\beta(j-1)+1,k} & \cdots & z_{i,\beta(j-1)+s,k} \\ \vdots & \ddots & \vdots \\ z_{\beta(i-1)+s,j,k} & \cdots & z_{\beta(i-1)+s,\beta(j-1)+s,k} \end{pmatrix} \in \mathbb{R}^{s\times s}$$

を考える. ただし, $i, j = 1, \dots (d - s)/\beta + 1$ である. そして, プーリングのための写像 g を用いて

$$z'_{ijk} = g(Z_{ijk})$$

をプーリング層の出力の (i, j, k) 成分とする. したがって, プーリングの処理はチャネルごとに作用し, 結果として出力のチャネル数は入力層のチャネル数と同じとなる.

写像 g としては例えば, Z_{ijk} の最大値や平均を考えることが多い. つまり, $P_{ij} = \{(\beta(i-1)+p, \beta(j-1)+q) \mid p, q = 1, \dots, s\}$ としたとき,

$$\text{(max プーリング)} \qquad z'_{ijk} = \max_{(r,s)\in P_{ij}} z_{rsk}$$

$$\text{(average プーリング)} \qquad z'_{ijk} = \frac{1}{s^2} \sum_{(r,s)\in P_{ij}} z_{rsk}$$

などがよく用いられる. また, max プーリングと average プーリングの中間的なものとして,

$$(L_p\text{プーリング}) \qquad z'_{ijk} = \left(\frac{1}{s^2} \sum_{(r,s)\in P_{ij}} z_{rsk}^p \right)^{1/p}$$

が用いられることもある. すぐにわかるように, L_p プーリングにおいて $p = 1$ とすれば average プーリングであり, (数学的に正確な表現ではないが) $p \to \infty$ で max プーリングとなる. 各 i, j に対して, チャネル

Algorithm 7 プーリング層

入力: 入力 $Z \in \mathbb{R}^{d\times d\times K}$, ストライド数 β, プーリングのための関数 f ▷ β は $d-s$ の約数

$d' = (d-s)/\beta + 1$ ▷ プーリング後のサイズ

$Z' = (\boldsymbol{z}'_{ij})_{i,j} = \mathbf{0} \in \mathbb{R}^{d'\times d'\times K}$ の初期化

for $i = 1$ to d' **do**

  for $j = 1$ to d' **do**

    $\boldsymbol{z}'_{ij} = g(Z_{ij})$

  end for

end for

出力: プーリング層の出力 $Z' = (\boldsymbol{z}'_{ij})_{i,j} \in \mathbb{R}^{d'\times d'\times K}$

ごとの出力をまとめて

$$\boldsymbol{z}'_{ij} = (z'_{ijk}) = g(Z_{ij})$$

と表すことにする. つまり, $Z_{ij} \in \mathbb{R}^{s\times s\times K}$ はもとの Z の部分配列であり, したがって $\boldsymbol{z}'_{ij}$ は, プーリングの出力 $Z' \in \mathbb{R}^{(\frac{d-s}{\beta}+1)\times(\frac{d-s}{\beta}+1)\times K}$ の 3 次元目の軸方向へ z'_{ijk} を並べたものである[*81]. 気分としては,

$$Z' = (\boldsymbol{z}'_{ij})_{ij} = \begin{pmatrix} \boldsymbol{z}'_{11} & \cdots & \boldsymbol{z}'_{1,(d-s)/\beta+1} \\ \vdots & \ddots & \vdots \\ \boldsymbol{z}'_{(d-s)/\beta+1,1} & \cdots & \boldsymbol{z}'_{(d-s)/\beta+1,(d-s)/\beta+1} \end{pmatrix}$$

が出力であり, 各 $\boldsymbol{z}'_{ij}$ は配列の奥行き (3 次元目の軸) 方向に並んでいると思ってもらいたい.

なお, max プーリングを行う場合, 極端に大きな値を持つピクセルは複数の i, j にわたって出力されることになる. そのため, この冗長性を省くために, max プーリングを用いる際にはストライド数をフィルタのサイズと同じ, つまり, $\beta = s$ とすることが多い. このようにストライド数を設定することで, P_{ij} の共通部分は空集合 ($P_{ij} \cap P_{i'j'} = \emptyset$, $^\forall(i', j') \neq (i, j)$) となり, したがって, Z_k の各成分は 1 度しかプーリングの際に考慮されなくなる. プーリング層における順伝播のアルゴリズムは Algorithm 7 の通りである.

■全結合層 畳み込みとプーリングを繰り返して得られた特徴 $Z \in \mathbb{R}^{d\times d\times K}$ に対して, すべての z_{ijk} に対して通常の (深層) ニューラルネットワークを構築する. つまり, d^2K 個のユニットからなる層から, d' 個のユニットからなる層へのネットワークを考える. i 番目の出力ユニット z'_i へ伸びるネットワークの重み $W_i \in \mathbb{R}^{d\times d\times K}$ とバイアス項 b_i を用いて,

$$u_i = \sum_{k=1}^{K}\sum_{p,q=1}^{d} w_{pqki} z_{pqk} + b_i = \langle W_i, Z\rangle + b_i$$

とする. そして, 活性化関数 h を用いて, $z'_i = h(u_i)$ を並べたものを全結合層の出力 $\boldsymbol{z}' = (h(u_i))_i \in \mathbb{R}^{d'}$ とする.

また, LeNet のように, 必要であれば d' 個のユニットからなる層からさらに次の層へ全結合を考えることがあるが, $\boldsymbol{z}'$ を出力層とすることもある. その場合, m クラスの分類問題であれば $d' = m$, h としてソフトマックス関数, 誤差関数としてクロスエントロピーを用いるのはこれまでと同じである.

念のため, 全結合層における順伝播のアルゴリズムをまとめておくと, Algorithm 8 の通りである.

[*81] python で max プーリングや average プーリングを行う場合には, 3 層配列 $A \in \mathbb{R}^{s\times s\times K}$ に対して `np.max(A, axis=(0,1))` や `np.mean(A, axis=(0,1))` のように, `axis` を固定すれば良い. このように `axis` を固定することで, 各 $A_i \in \mathbb{R}^{s\times s}$ ごとの最大値や平均を出力できる.

Algorithm 8 全結合層

入力: 入力 $Z \in \mathbb{R}^{d\times d\times K}$, パラメータ $W \in \mathbb{R}^{d\times d\times K\times d'}$, 活性化関数 $f \triangleright d'$ は出力される層のユニット数

1: $U = \mathbf{0} \in \mathbb{R}^{d'}$ の初期化
2: **for** $i = 1$ to d' **do**
3: $\quad u_i = \langle W_i, Z\rangle + b_i$
4: **end for**

出力: プーリング層の出力 $Z' = f(U) \in \mathbb{R}^{d'}$

3.5.2 畳み込みニューラルネットワークの逆伝播

畳み込みニューラルネットワークの順伝播が

入力層 → 畳み込み層とプーリング層の繰り返し → 全結合層の繰り返し → 出力層

で構成されることはすでに述べた. ここでは, 勾配降下法のアルゴリズムを設計するための逆伝播について説明する. 以下, 各層における入力は Z_r, 出力は Z_{r+1} であるとする. ただし, Z_r や Z_{r+1} の配列としての大きさは, どの層で計算したかに依存するの, その都度述べることにする.

■順伝播の統一的な表現 準備として, 畳み込み層, プーリング層および全結合層の順伝播が, 適当な配列やスカラーを用いて

$$u = \langle W, Z\rangle + b$$

に活性化関数 f を作用させたもの (を適当な順番に並べたもの) として記述できることを確認しておこう. すでに述べた通り, 畳み込み層と全結合層では, 入力とパラメータの線形結合に対して活性化関数を作用させて, それぞれの層における出力が得られることは明らかである. 残るは, プーリング層における演算が, どのように書けるかということである.

フィルタサイズを H, ストライド数を β とすれば, プーリング層では入力 $Z = (Z_1, \dots, Z_K) \in \mathbb{R}^{d\times d\times K}$ を $Z' = (Z'_1, \dots, Z'_K) \in \mathbb{R}^{d'\times d'\times K}$ に変換する操作であった. ただし, $d' = (d-H)/\beta + 1$ である. いま, 各 $i, j = 1, \dots, d'$ に対して, $P_{ij} = \{(\beta(i-1)+p, \beta(j-1)+q) \mid p, q = 1, \dots, H\}$ として, $W_{ijk} = (w_{ij,pqk})_{p,q} \in \mathbb{R}^{d\times d}$ を以下のように定義しよう. つまり, $Z_k = (z_{stk})_{s,t}$ として, max プーリングを用いる場合には, 各 k に対して

$$w_{ij,pqk} = \begin{cases} 1 & (p,q) = \arg\max_{(s,t)\in P_{ij}} z_{stk} \\ 0 & \text{その他} \end{cases}$$

とし, average プーリングを用いる場合には

$$w_{ij,pqk} = 1/s^2$$

とすれば良い. すると, このように定義した W_{ijk} を用いて, プーリング層での順伝播は

$$z'_{ijk} = \langle W_{ijk}, Z_k\rangle$$

とかける. ただし, z'_{ijk} はプーリング層の k チャネル目での出力 Z'_k の第 (i,j) 成分である. したがって, プーリングの写像 g は $\langle W_{ij,k}, Z_k\rangle$ に関する恒等写像と思って良い. なお, z'_{ijk} を k の軸方向に並べて

$$\boldsymbol{z}'_{ij} = (z'_{ijk})_k = \langle W_{ij}, Z\rangle$$

表 7　畳み込みニューラルネットワークにおける順伝播．入力と活性化関数をそれぞれ $Z \in \mathbb{R}^{d\times d\times K}$ および f とし，パラメータと出力を $W, \boldsymbol{b}$ および Z' とする．ただし，畳み込み層において $\tilde{Z}_{ij} = (\tilde{z}_{st})_{(s,t)\in P_{ij}}$，$P_{ij} = \{(\beta(i-1)+p, \beta(j-1)+q) \mid p,q = 1,\dots,H\}$ はパディング後の入力 $\tilde{Z}$ の P_{ij} に関する部分配列を表す．全結合層のみ，得られるユニットはベクトルであり，畳み込み層とプーリング層は 3 次元配列となる．また，各層においてフィルタのサイズや，ストライド数は異なることに注意する．

	畳み込み層	プーリング層	全結合層
パディング	α	–	–
ストライド	β	β	–
フィルタのサイズ	$H \times H$	$H \times H$	–
W のサイズ	$H \times H \times K \times M$	$d \times d \times d' \times d' \times K$	$d \times d \times K \times M$
$\boldsymbol{b}$ のサイズ	M	–	M
Z' のサイズ	$d' \times d' \times M$	$d' \times d' \times K$	M
Z' の要素	$d' = (d + 2\alpha - H)/\beta + 1$ $z'_{ijm} = f(u_{ijm})$ $u_{ijm} = \langle W_m, \tilde{Z}_{ij}\rangle + b_m$	$d' = (d - H)/\beta + 1$ $z'_{ijm} = u_{ijm}$ $u_{ijm} = \langle W_{ijm}, Z_m\rangle$	– $z'_i = f(u_i)$ $u_i = \langle W_i, Z\rangle + b_i$

とかくこともできる．また，W_{ijk} は添字 i,j,k にわたってとるので，重みは 5 層配列 $W = (W_{ijk})_{i,j,k} \in \mathbb{R}^{d\times d\times d'\times d'\times K}$ のように表すこともできるが，逆伝播の計算においては本質的な違いはない．なお，L_p プーリングについては，Z_{rk} の非線形変換を用いて同じようにかけるが，ここでは省略する[*82]．

畳み込みニューラルネットワークの，それぞれの層での順伝播をまとめると表 7 のようになる．

注意 4. max プーリングでは**タイ**，つまり，最大値を達成する (p,q) が複数あるかもしれない．どういうことかというと，$P_{ij} = \{(\beta(i-1)+p, \beta(j-1)+q) \mid p,q = 1,\dots,s\}$ に対して，

$$(p,q) \in \arg\max_{(s,t)\in P_{ij}} z_{stk} = \mathcal{P} = \{(s_j,t_j) \in P_{ij} \mid j = 1,\dots,\tau\}$$

のように右辺が集合となっている場合である．ここで，τ はタイの数を表すものとする．この場合，すでに定義したような重み W_{ijk} を作れなくなってしまうため，例えば，以下のような対処法が用いられる．

1. ランダムに j_* をサンプリングし，$(p,q) = (p_{j_*}, q_{j_*})$ なら $w_{pq,ijk} = 1$，それ以外で 0 とする．
2. すべての最大化点に均等に重みを割り振る．つまり，$(p,q) \in \mathcal{P}$ ならば $w_{pq,ijk} = 1/\tau$，それ以外で 0 とする．

1, 2 のいずれの方法を用いても，プーリング層の出力はやはり $z'_{ijk} = \langle W_{ijk}, Z_k\rangle$ とかけることに注意しよう．なお，1 を用いる場合は，j_* にサンプリングのランダムネスが生じる．したがって，j_* の選び方によって，畳み込みニューラルネットワークの学習結果が変わるこにも注意する．

■逆伝播の計算　畳み込みニューラルネットワークの順伝播と逆伝播のイメージは図 23 のようになる．ただし，図 23 の赤線で表されている逆伝播の実際の計算では，W_{r+1} や W_{r+2} を適当な部分配列に対して計算することに注意する．とはいえ，これまでの逆伝播と同様に，畳み込みニューラルネットワークの逆伝播でも本質的には同じことを行っていることに注意してほしい．以下ではまず各層における誤差 $\boldsymbol{\delta}_r$ の計算方法について述べ，その後，パラメータの更新規則について述べる．なお，Z_r から Z_{r+1} へのネットワークがプーリング層の場合，対応する活性化関数は恒等写像とし，$\nabla f_r(u_{ijk}) = 1$ であると考えれば良い．

[*82] 基本的には，同じ議論によって，L_p プーリングを用いた場合の逆伝播を計算できる．

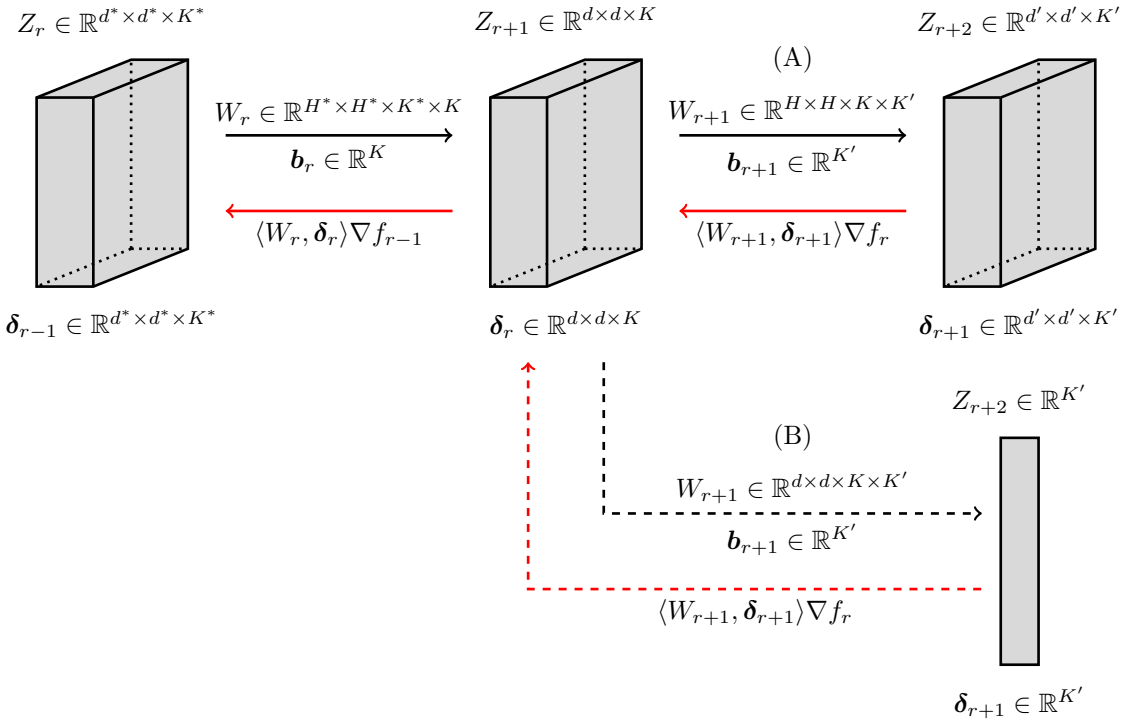


図 23　畳み込みニューラルネットワークの逆伝播のイメージ．黒線は準伝播および, 準伝播で用いるパラメータ, 赤線は逆伝播の更新式を表している．Z_{r+1} から Z_{r+2} へのネットワークのうち, (A) は畳み込み層やプーリング層のネットワーク, (B) は全結合層のネットワークを表したものである．ただし, 逆伝播における内積 $\langle \cdot, \cdot \rangle$ はあくまでも直感的な理解を助けるためのものであり, 実際の計算規則ではないことに注意する．同様に, ∇f_r は活性化関数の微分を表すものであるが, 実際には適当な引数で計算したものなので, この点も注意しよう．

■全結合層における逆伝播:　図 23 で, (B) の順伝播は全結合層における伝播であるから, 先に述べたように, $Z_{r+1} \in \mathbb{R}^{d \times d \times K}$ として,

$$u_{r+1,k} = \langle W_{r+1,k}, z_{r+1} \rangle + b_{r+1,k} = \sum_{l=1}^{K} \sum_{p,q=1}^{d} w_{r+1,pqlk} Z_{r+1,pql} + b_{r+1,k}, \quad k = 1, \ldots, K' \tag{13}$$

に対して, 活性化関数 f_{r+1} を用いて $Z_{r+2} = f_{r+1}(\boldsymbol{u}_{r+1})$ となる．ただし, $W_{r+2} = (W_{r+2,1}, \ldots, W_{r+2,K'}) \in \mathbb{R}^{d \times d \times K \times K'}$, $\boldsymbol{b}_{r+2} = (b_{r+2,1}, \ldots b_{r+2,K'})^\top \in \mathbb{R}^{K'}$ および $\boldsymbol{u}_{r+1} = (u_{r+1,1}, \ldots, u_{r+1,K'})^\top \in \mathbb{R}^{K'}$ である．したがって, $W_{r+1,k} \in \mathbb{R}^{d \times d \times K}$ は, 全結合層の出力の k 番目のユニットへのネットワークの重みである．

Z_{r+2} が出力層である場合, これまでと同じように, 適当な誤差関数 E に対して,

$$\boldsymbol{\delta}_{r+1} = \frac{\partial E}{\partial \boldsymbol{u}_{r+1}} = f_{r+1}(\boldsymbol{u}_{r+1}) - \boldsymbol{y}$$

となる．また, $r+2$ 層目 (つまり, Z_{r+2}) が, 深層ニューラルネットワークの中間層なら, $r+3$ 層目 (つまり, Z_{r+3}) の逆伝播を $\boldsymbol{\delta}_{r+2}$ として,

$$\boldsymbol{\delta}_{r+1} = \frac{\partial E}{\partial \boldsymbol{u}_{r+1}} = (W_{r+2}^\top \boldsymbol{\delta}_{r+2}) \odot \nabla f_{r+1}(\boldsymbol{u}_{r+1}) \in \mathbb{R}^{K'}$$

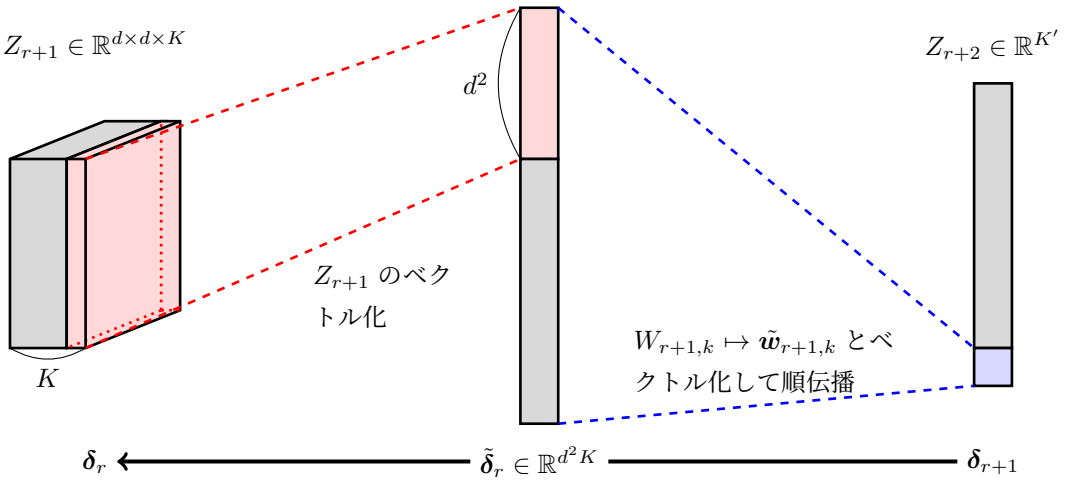


図 24　全結合層における順伝播と逆伝播の様子．逆伝播では，$W_{r+1,k}$ をベクトル化した $\tilde{\boldsymbol{w}}_{r+1,k} \in \mathbb{R}^{d^2K}$ を用いて作られる $K' \times d^2K$ の行列 $\tilde{W}_{r+1}$ を経由して行う．$\tilde{\boldsymbol{\delta}}_r \in \mathbb{R}^{d\times d\times K}$ は $Z_{r+1}, W_{w+1,k},\ (k = 1,\dots,K')$ および $\nabla f_r(U_r)$ をのベクトル化に基づき計算された逆伝播であり，これを元の配列と同じサイズに並べ替えたものが $\boldsymbol{\delta}_r \in \mathbb{R}^{d\times d\times K}$ である．ただし，f_r と U_r は Z_{r+1} へのネットワークの活性化関数および，パラメータと Z_r の線形結合である．

となる．ただし，W_{r+2} は Z_{r+2} から Z_{r+3} への順伝播の重みであり，これまで通り適当な大きさの行列を用いて表される[*83]．ところで，W_{r+2} の第 k 列を $\tilde{\boldsymbol{w}}_{r+2,k}$ とすれば，$\boldsymbol{\delta}_{r+1}$ の第 k 成分は

$$\delta_{r+1,k} = \tilde{\boldsymbol{w}}_{r+2,k}^\top \boldsymbol{\delta}_{r+2} \nabla f_{r+1}(u_{r+1,k}) = \langle \tilde{\boldsymbol{w}}_{r+2,k}, \boldsymbol{\delta}_{r+2} \rangle \nabla f_{r+1}(u_{r+1,k})$$

となって，k 番目のユニットへの重み $\tilde{\boldsymbol{w}}_{r+2,k}$ と誤差 $\boldsymbol{\delta}_{r+2}$ の内積に，活性化関数の勾配 $\nabla f_{r+1}(u_{r+1,k})$ をかけたものだということがわかる．

$\boldsymbol{\delta}_{r+1}$ から $\boldsymbol{\delta}_r$ の逆伝播を求める方法を説明する．まず，全結合層では図 24 のように，一度 Z_{r+1} をベクトル化し，その後，通常のニューラルネットワークの順伝播が行われると思うことができる．なお，$Z_{r+1} = (Z_{r+1,1}, \dots, Z_{r+1,K}) \in \mathbb{R}^{d\times d\times K}$ のベクトル化とは，各 k に対して，

$$Z_{r+1,k} = (\boldsymbol{z}_{r+1,1k}, \dots, \boldsymbol{z}_{r+1,dk}) \mapsto \tilde{\boldsymbol{z}}_{r+1,k} = \begin{pmatrix} \boldsymbol{z}_{r+1,1k} \\ \vdots \\ \boldsymbol{z}_{r+1,dk} \end{pmatrix} \in \mathbb{R}^{d^2}$$

として[*84]，各チャネルごとにベクトル化し，$\tilde{\boldsymbol{z}}_{r+1} = (\boldsymbol{z}_{r+1,1}^\top, \dots, \boldsymbol{z}_{r+1,K'}^\top)^\top \in \mathbb{R}^{d^2K}$ と並べたものである．同じように，パラメータ $W_{r+1,k} = (w_{r+1,ijmk})_{i,j,m} \in \mathbb{R}^{d\times d\times K}$ をベクトル化すれば，(13) の線形結合は

$$u_{r+1,k} = \tilde{\boldsymbol{w}}_{r+1,k}^\top \tilde{\boldsymbol{z}}_{r+1} + b_{r+1,k}$$

と書き換えることができる．さらに，$\tilde{W}_{r+1} = (\tilde{\boldsymbol{w}}_{r+1,1}, \dots, \tilde{\boldsymbol{w}}_{r+1,K'})^\top \in \mathbb{R}^{K'\times d^2K}$ として，

$$\boldsymbol{u}_{r+1} = \tilde{W}_{r+1} \tilde{\boldsymbol{z}}_{r+1} + \boldsymbol{b}_{r+1}$$

が得られる．つまり，これまで説明してきた，通常のニューラルネットワークと同じ構造に帰着できる．したがって，$Z_r \in \mathbb{R}^{d^*\times d^*\times K^*}$ から Z_{r+1} への順伝播で得られる $U_r = (u_{ijm})_{i,j,m} \in \mathbb{R}^{d\times d\times K}$ を Z_{r+1} と同じようにベクトル化したものを $\tilde{\boldsymbol{u}}_r$ とすれば，

$$\tilde{\boldsymbol{\delta}}_r = \langle \tilde{W}_{r+1}, \boldsymbol{\delta}_{r+1} \rangle \odot \nabla f_r(\tilde{\boldsymbol{u}}_r) = \tilde{W}_{r+1}^\top \boldsymbol{\delta}_{r+1} \odot \nabla f_r(\tilde{\boldsymbol{u}}_r) \in \mathbb{R}^{d^2K}$$

[*83] より正確には，Z_{r+3} のユニット数を K'' とすれば，$K'' \times K'$ の行列である．
[*84] つまり，$Z_{r+1,k}$ を列ごとに縦に並べたものである．

Algorithm 9 全結合層の逆伝播

入力: $W_{r+1} \in \mathbb{R}^{d\times d\times K\times K'}$, 誤差 $\boldsymbol{\delta}_{r+1} \in \mathbb{R}^{K'}$, 活性化関数の勾配 $\nabla f(U_r) \in \mathbb{R}^{d\times d\times K}$

1: $\boldsymbol{\delta}_r \leftarrow (\sum_{k=1}^{K'} W_{r+1,k}\delta_{r+1,k}) \odot \nabla f_r(U_r)$

出力: 誤差 $\boldsymbol{\delta}_r \in \mathbb{R}^{d\times d\times K}$

を元の Z_r と同じように, 再度 $d \times d \times K$ の配列に並べ替えれば $\boldsymbol{\delta}_r \in \mathbb{R}^{d\times d\times K}$ が得られる.

具体的に, $\tilde{\boldsymbol{\delta}}_r$ をどのように並べ替えれば $\boldsymbol{\delta}_r$ が得られるかを確認しておく. $\nabla f_r(\tilde{\boldsymbol{u}}_r) \in \mathbb{R}^{d^2K}$ を並べ替えれば $\nabla f_r(U_r) \in \mathbb{R}^{d\times d\times K}$ となるのは明らかであろう. 一方, $\tilde{W}_{r+1} = (\tilde{\boldsymbol{w}}_{r+1,1}, \ldots, \tilde{\boldsymbol{w}}_{r+1,K'})$ であるから,

$$\tilde{W}_{r+1}^\top \boldsymbol{\delta}_{r+1} = \sum_{k=1}^{K'} \tilde{\boldsymbol{w}}_{r+1,k}\delta_{r+1,k}$$

である. さらに, $\tilde{\boldsymbol{w}}_{r+1,k} \in \mathbb{R}^{d^2K}$ は $W_{r+1,k} = (w_{r+1,ijmk})_{i,j,m} \in \mathbb{R}^{d\times d\times K}$ のベクトル化であるから, 改めて $d \times d \times K$ の配列に戻そうと思えば,

$$\tilde{W}_{r+1}^\top \boldsymbol{\delta}_{r+1} \mapsto \sum_{k=1}^{K'} W_{r+1,k}\delta_{r+1,k}$$

とすれば良い. したがって,

$$\boldsymbol{\delta}_r = \left(\sum_{k=1}^{K'} W_{r+1,k}\delta_{r+1,k}\right) \odot \nabla f_r(U_r) \in \mathbb{R}^{d\times d\times K}$$

であり, その第 (i, j, m) 成分は

$$\delta_{r,ijm} = \sum_{k=1}^{K'} w_{r+1,ijmk}\delta_{r+1,k}\nabla f_r(u_{r,ijm})$$

となる[*85]. 全結合層の逆伝播をまとめると Algorithm 9 のようになる.

ところで, $\tilde{\boldsymbol{w}}_{r+1,ijm} = (w_{r+1,ijm1}, \ldots, w_{r+1,ijmK'})^\top \in \mathbb{R}^{K'}$ とすれば, これは

$$\delta_{r,ijm} = \tilde{\boldsymbol{w}}_{r+1,ijm}^\top \boldsymbol{\delta}_{r+1}\nabla f_r(u_{r,ijm}) = \langle \tilde{\boldsymbol{w}}_{r+1,ijm}, \boldsymbol{\delta}_{r+1}\rangle \nabla f_r(u_{r,ijm})$$

となり, やはりパラメータと誤差の内積に活性化関数の勾配をかけたものであることがわかる.

■畳み込み層における逆伝播 全結合層とは異なり, 畳み込み層の順伝播は, 入力の配列に対して着目する部分配列に対して計算されるので, やや複雑な処理を行っているように見えるが, 逆伝播の本質はやはり同じである. 畳み込み層は, 図 23 のネットワークにおいて, (A) に対応しており, パティング数とストライド数をそれぞれ α および β とすれば, 入力 $Z_{r+1} = (Z_{r+1,1}, \ldots, Z_{r+1,K}) \in \mathbb{R}^{d\times d\times K}$ は $Z_{r+2} = (Z_{r+2,1}, \ldots, Z_{r+2,K'}) \in \mathbb{R}^{d'\times d'\times K'}$ として出力される. ただし, $d' = (d+2\alpha-H)/\beta+1$ であり, H はフィルタのサイズである. 以下, $Z_{r+1,k}$ をパディング数 α でパディングしたものを $\tilde{Z}_{r+1,k} \in \mathbb{R}^{(d+2\alpha)\times(d+2\alpha)}$ とし, 畳み込みの際に着目する領域を $P_{ij} = \{(\beta(i-1)+p, \beta(j-1)+q) \mid p, q = 1, \ldots, H\}$ $(i, j = 1, \ldots, d')$ とする. なお, 先に述べたように, パディングはゼロパディングのみを考える.

パラメータを $W_{r1}, \ldots, W_{rM} \in \mathbb{R}^{H\times H\times K}$ および $b_{r1}, \ldots, b_{rM} \in \mathbb{R}$ とすると, 畳み込み層では

$$u_{r+1,ijm} = \langle W_{r+1,m}, \tilde{Z}_{r+1,ij}\rangle + b_{r+1,m} \tag{14}$$

[*85] python でこの計算を行う場合, for 文を利用するか, $\boldsymbol{\delta}_{r+1}$ のサイズを強制的に変更して `np.sum` を用いて実装できる. 例えば, `W` を `(M, K, d, d)` の numpy 配列, `v` が `(M, )` となっている場合, `v_reshape = v.reshape(M, 1, 1, 1)` として `v` のサイズを `(M, 1, 1, 1)` に変換できるので, `np.sum(W * v_reshape, axis=0)` とすれば良い.

に対して, 活性化関数 f_r を用いて $z_{r+2,ijm} = f_{r+1}(u_{r+1,ijm})$ という処理が行われる. ただし,

$$\tilde{Z}_{r+1,ij,k} = \begin{pmatrix} \tilde{z}_{r+1,\beta(i-1)+1,\beta(j-1)+1,k} & \cdots & \tilde{z}_{r+1,i,\beta(j-1)+H,k} \\ \vdots & \ddots & \vdots \\ \tilde{z}_{r+1,\beta(i-1)+H,j,k} & \cdots & \tilde{z}_{r+1,\beta(i-1)+H,\beta(j-1)+H,k} \end{pmatrix} \in \mathbb{R}^{H\times H}$$

に対して, $\tilde{Z}_{r+1,ij} = (\tilde{Z}_{r+1,ij,1}, \ldots, \tilde{Z}_{r+1,ij,K}) \in \mathbb{R}^{H\times H\times K}$ は $\tilde{Z}_{r,ij,k}$ を k の軸方向に並べた配列である.

逆伝播を計算するために, $\boldsymbol{\delta}_{r+1}$ は $d'\times d'\times K'$ の配列だとする. また, Z_r から Z_{r+1} の順伝播は $Z_{r+1} = f_r(U_r)$ とかけているとし, パディング数 α で拡大したものを $\tilde{Z}_{r+1} = f_r(\tilde{U}_r) \in \mathbb{R}^{(d+2\alpha)\times(d+2\alpha)\times K}$ と書くことにする. したがって, 各 $k = 1, \ldots, K$ に対して,

$$f_r(\tilde{u}_{r,pqk}) = \begin{cases} f_r(u_{r,p-\alpha,q-\alpha,k}) & \alpha+1 \le p,q \le \alpha+d \\ 0 & \text{その他} \end{cases}$$

である[*86]. 以上の準備のもと, (14) を書き換えると,

$$\begin{aligned} u_{r+1,ijm} &= \sum_{k=1}^{K}\sum_{p,q=1}^{H} w_{r+1,pqk,m}\tilde{z}_{r+1,\beta(i-1)+p,\beta(j-1)+q,k} + b_{r+1,m} \\ &= \sum_{k=1}^{K}\sum_{p,q=1}^{H} w_{r+1,pqk,m} f_r(\tilde{u}_{r,\beta(i-1)+p,\beta(j-1)+q,k}) + b_{r+1,m} \end{aligned}$$

となり, したがって, 合成関数の微分を用いて, $\boldsymbol{\delta}_r \in \mathbb{R}^{d\times d\times K}$ の (i,j,m) 成分は

$$\delta_{r,ijm} = \frac{\partial E}{\partial u_{r,ijm}} = \sum_{l=1}^{K'}\sum_{s,t=1}^{d'} \frac{\partial E}{\partial u_{r+1,stl}}\frac{\partial u_{r+1,stl}}{\partial u_{r,ijm}}$$

となることがわかる. なお, 定義より, $\delta_{r+1,stl} = \frac{\partial E}{\partial u_{r+1,stl}}$ である. また,

$$\begin{aligned} \frac{\partial u_{r+1,stl}}{\partial u_{r,ijm}} &= \frac{\partial}{\partial u_{r,ijm}}\left(\sum_{k=1}^{K}\sum_{p,q=1}^{H} w_{r+1,pqk,l} f_r(\tilde{u}_{r,\beta(s-1)+p,\beta(t-1)+q,k}) + b_{r+1,l}\right) \\ &= \sum_{k=1}^{K}\sum_{p,q=1}^{H} w_{r+1,pqk,l}\frac{\partial f_r(\tilde{u}_{r,\beta(s-1)+p,\beta(t-1)+q,k})}{\partial u_{r,ijm}} \end{aligned}$$

であるが, f_r の偏微分は

$$\frac{\partial f_r(\tilde{u}_{r,\beta(s-1)+p,\beta(t-1)+q,k})}{\partial u_{r,ijm}} = \begin{cases} \nabla f(\tilde{u}_{r,ijm}) & \beta(s-1)+p = i, \beta(t-1)+q = j, k = m \\ 0 & \text{その他} \end{cases}$$

であるが, これをパディング前の U_r を用いて表すと,

$$\frac{\partial f_r(\tilde{u}_{r,\beta(s-1)+p,\beta(t-1)+q,k})}{\partial u_{r,ijm}} = \begin{cases} \nabla f(u_{r,ijm}) & \beta(s-1)+\alpha+p = i, \beta(t-1)+\alpha+q = j, k = m \\ 0 & \text{その他} \end{cases}$$

となる. したがって,

$$\delta_{r,ijm} = \sum_{l=1}^{K'}\sum_{s,t=1}^{d'} \delta_{r+1,stl} w_{r+1,i-\alpha-\beta(s-1),j-\alpha-\beta(t-1),m,l} \nabla f_r(u_{r,ijm})$$

が得られる. ただし, p と q がそれぞれ $\beta(s-1)+\alpha+p = i$ および $\beta(t-1)+\alpha+q = j$ を満たすことを用いた[*87].

[*86] 不等式 $a \le i,j \le b$ は "$a \le i \le b$ かつ $a \le j \le b$" を意味する.

[*87] つまり, $p = i-\alpha-\beta(s-1), q = j-\alpha-\beta(j-1)$ を用いて, パラメータの添字を書き換えた.

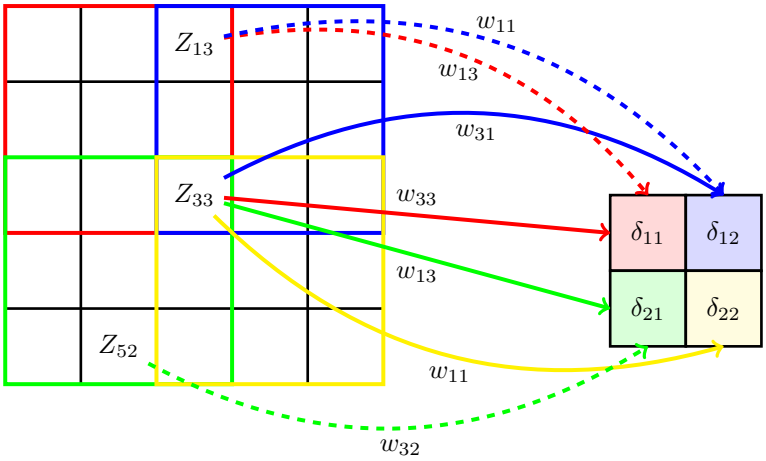


図 25　畳み込み層の逆伝播. 3×3 のフィルタを用いて畳み込む場合, Z_{33} は出力を計算する際に全て計算対象となる. 一方, Z_{13} は出力の $(1,1)$ 成分および $(1,2)$ 成分を計算する際にのみ計算対象となり, Z_{52} は $(2,1)$ 成分を計算する際にのみ計算対象となる.

s,t が 1 から d' まで動くとき, パラメータ $w_{r+1,i-\alpha-\beta(s-1),j-\alpha-\beta(t-1),m,l}$ の添字が負になる可能性があるので, 実際に動く範囲を計算しておこう. f_r の微分に, s,t に関する条件が入るので, 念のため, どこで和をとるのかを明確にしておこう. $W_{r+1,l}$ が $H \times H \times K$ の配列であることを思い出せば, $i-\alpha-\beta(s-1)$ も $1,\ldots,H$ のいずれかの値しかとれないはずである. したがって,

$$1 \leq i-\alpha-\beta(s-1) \leq H \Rightarrow \frac{i-\alpha-H}{\beta}+1 \leq s \leq \frac{i-\alpha-1}{\beta}+1$$

となる. これと, もともと $1 \leq s \leq d'$ であったことから,

$$\max\left\{1, \frac{i-\alpha-H}{\beta}+1\right\} \leq s \leq \min\left\{d', \frac{i-\alpha-1}{\beta}+1\right\}$$

であるような自然数 s で和を取れば良いことがわかる. 同様に, t は

$$\max\left\{1, \frac{j-\alpha-H}{\beta}+1\right\} \leq t \leq \min\left\{d', \frac{j-\alpha-1}{\beta}+1\right\}$$

を満たす自然数である[*88].

$\delta_{r+1,stl}w_{r+1,i-\beta(s-1),j-\beta(t-1),m,l}$ の意味を説明しておく. 簡単のため, 層の添字 $r+1$ は省略することにして, 図 25 のような畳み込み層のネットワークを考える. つまり, (パディング数 0 は適当にパディング済みの) 5×5 の行列を Z として, 3×3 のフィルタを用いて, ストライド数 2 で畳み込んだ結果を出力する. したがって, 畳み込んだ結果は 2×2 の行列である. 出力には同じサイズの誤差が付加されているため, それを $\boldsymbol{\delta}$ とし, 同じ大きさの畳み込み層の出力は省略する. いま, Z_{13} が畳み込まれるのは, 出力の $(1,1)$ 成分と $(1,2)$ 成分を計算する場合である. また, 出力の $(1,1)$ 成分を計算する際には, Z_{13} には重み w_{13} との積をとり, $(1,2)$ 成分を計算する際には w_{11} との積をとる. したがって, $\boldsymbol{\delta}$ から Z_{13} のセルに対応する逆伝播の $\delta_{r+1,stl}w_{r+1,i-\beta(s-1),j-\beta(t-1),m,l}$ に関する和は

$$\delta_{11}w_{13}+\delta_{12}w_{11}$$

となる. 同様に, Z_{52} のセルに対応する部分は

$$\delta_{21}w_{32}$$

[*88] 実数 x に対して, 小数点を切り捨てたものを $\lfloor x \rfloor$, 切り上げたものを $\lceil x \rceil$ とすれば, $s = \lceil \max\{1,(i-\alpha-H)/\beta+1\} \rceil, \ldots, \lfloor \min\{d',(i-\alpha-1)/\beta+1\} \rfloor$ ともかける.

Algorithm 10 畳み込み層の逆伝播

入力: $W_{r+1} \in \mathbb{R}^{H\times H\times K\times K'}$, 誤差 $\boldsymbol{\delta}_{r+1} \in \mathbb{R}^{d'\times d'\times K'}$, 活性化関数の勾配 $\nabla f_r(U_r) \in \mathbb{R}^{d\times d\times K}$, パディング数 α, ストライド数 β

$\tilde{\boldsymbol{\delta}}_r = (\tilde{\delta}_{ijk})_{i,j,k} = \mathbf{0} \in \mathbb{R}^{d\times d\times K}$ ▷ 誤差のもと
for $k = 1$ to K **do**
  for $i = 1$ to d **do**
    for $j = 1$ to d **do**
      $W = (w_{pqm})_{p,q,m} = \mathbf{0} \in \mathbb{R}^{d'\times d'\times K'}$ ▷ W_{r+1} の部分配列保存用の配列
      for $m = 1$ to K' **do**
        for $p = 1$ to d' **do**
          for $q = 1$ to d' **do**
            $p' \leftarrow i - \alpha - \beta(p-1); q' \leftarrow j - \alpha - \beta(j-1)$
            if $1 \le p', q' \le H$ **then**
              $w_{pqm} \leftarrow w_{r+1,p',q',k,m}$
            else
              $w_{pqm} \leftarrow 0$
            end if
          end for
        end for
      end for
      $\tilde{\delta}_{r,ijk} \leftarrow \langle \boldsymbol{\delta}_{r+1}, W \rangle$
    end for
  end for
end for

出力: 誤差 $\boldsymbol{\delta}_r = \tilde{\boldsymbol{\delta}}_r \odot \nabla f_r(U_r) \in \mathbb{R}^{d\times d\times K}$

である. さらに, Z_{33} は, 出力の $(1,1)$ 成分, $(1,2)$ 成分, $(2,1)$ 成分および $(2,2)$ 成分の全てを計算する場合なので, Z_{33} のセルに対応する逆伝播の $\delta_{r+1,stl} w_{r+1,i-\beta(s-1),j-\beta(t-1),m,l}$ に関する和は

$$\delta_{11}w_{33} + \delta_{12}w_{31} + \delta_{21}w_{13} + \delta_{22}w_{11}$$

のように計算される. つまり, 畳み込み層における逆伝播では, (パディング済みの入力) Z_{r+1} から Z_{r+2} への順伝播を考えたときに, 出力へつながるネットワークの重みと, 対応する $\boldsymbol{\delta}_{r+1}$ の成分の積和を計算するということである.

配列を用いてもう少し簡略化しておこう.

$$\boldsymbol{\delta}_{r+1,l} = (\delta_{r+1,stl})_{s,t} \in \mathbb{R}^{d'\times d'}$$

$$W_{r+1,ij,ml} = \begin{pmatrix} w_{r+1,i-\alpha,j-\alpha,m,l} & \cdots & w_{r+1,i,j-\alpha-\beta(d'-1),m,l} \\ \vdots & \ddots & \vdots \\ w_{r+1,i-\alpha-\beta(d'-1),j,m,l} & \cdots & w_{r+1,i-\alpha-\beta(d'-1),j-\alpha-\beta(d'-1),m,l} \end{pmatrix} \in \mathbb{R}^{d'\times d'}$$

とする. ただし, 例えば, 図 25 の Z_{13} や Z_{52} のように, 出力層へのネットワークで繋がっていない部分が

ある場合には, $W_{r+1,ij,ml}$ の対応する部分は 0 としておく. このとき,

$$\delta_{r,ijm} = \sum_{l=1}^{K'} \langle \boldsymbol{\delta}_{r+1,l}, W_{r+1,ij,ml} \rangle \nabla f_r(u_{r,ijm}) = \langle \boldsymbol{\delta}_{r+1}, W_{r+1,ijm} \rangle \nabla f_r(u_{r,ijm})$$

とかける. したがって, $\delta_{r,ijm}$ の計算はこれまで通り $\boldsymbol{\delta}_{r+1}$ と $W_{r+1,ijm}$ の内積に, 順伝播の誤差 $\nabla f_r(u_{r,ijm})$ をかけたものであり, これを i,j および m 成分の次元方向に並べたものが $\boldsymbol{\delta}_r$ となる. ただし, $\boldsymbol{\delta}_{r+1} = (\boldsymbol{\delta}_{r+1,1} \ldots, \boldsymbol{\delta}_{r+1,K'}), W_{r+1,ijm} = (W_{r+1,ij,m1}, \ldots, W_{r+1,ij,mK'}) \in \mathbb{R}^{d' \times d' \times K'}$ である. 畳み込み層の逆伝播は Algorithm 10 の通りである.

■プーリング層における逆伝播 プーリング層にはパラメータが含まれないが, ネットワークとして繋がっているので, ここでもやはり誤差の伝播が必要である. ストライド数を β とすれば, プーリング層では入力 $Z_{r+1} = (Z_{r+1,1}, \ldots, Z_{r+1,K}) \in \mathbb{R}^{d \times d \times K}$ を $Z_{r+2} = (Z_{r+2,1}, \ldots, Z_{r+2,K}) \in \mathbb{R}^{d' \times d' \times K}$ に変換する操作であった. ただし, H をプーリング層のフィルタサイズとしたとき, $d' = (d - H)/\beta + 1$ である.

Z_r から Z_{r+1} への順伝播において, パラメータと Z_r の線形結合を U_r としよう. このとき, すでに述べたように, プーリング層の順伝播も, 適当な配列 (行列) $W_{r+1,ijk} \in \mathbb{R}^{d \times d}$ を用いて

$$\begin{aligned} z_{r+2,ijk} = u_{r+1,ijk} &= \langle W_{r+1,ijk}, Z_{r+1,k} \rangle \\ &= \sum_{p,q=1}^{d} w_{r+1,ij,pqk} z_{r+1,pqk} = \sum_{p,q=1}^{d} w_{r+1,ij,pqk} f_r(u_{r,pqk}) \end{aligned}$$

とかける. このとき, 畳み込み層での計算と同じように, 合成関数の微分を用いて

$$\delta_{r,ijk} = \frac{\partial E}{\partial u_{r,ijk}} = \sum_{s,t=1}^{d'} \frac{\partial E}{\partial u_{r+1,stk}} \frac{\partial u_{r+1,stk}}{\partial u_{r,ijk}}$$

となる.

$$\frac{\partial u_{r+1,stk}}{\partial u_{r,ijk}} = \frac{\partial}{\partial u_{r,ijk}} \left(\sum_{p,q=1}^{d} w_{r+1,st,pqk} f_r(u_{r,pqk}) \right) = \sum_{p,q=1}^{d} w_{r+1,st,pqk} \frac{\partial f_r(u_{r,pqk})}{\partial u_{r,ijk}}$$

であるが,

$$\frac{\partial f_r(u_{r,pqk})}{\partial u_{r,ijk}} = \begin{cases} \nabla f_r(u_{r,ijk}) & p = i, q = j \\ 0 & \text{その他} \end{cases}$$

より, $\delta_{r+1,stk} = \frac{\partial E}{\partial u_{r+1,stk}}$ に注意すれば,

$$\delta_{r,ijk} = \sum_{s,t=1}^{d'} \delta_{r+1,stk} w_{r+1,st,ijk} \nabla f_r(u_{r,ijk})$$

である. これまでと同様に, $\boldsymbol{\delta}_{r+1,k} = (\delta_{r+1,stk})_{s,t}, W_{r+1,ijk} = (w_{r+1,st,ijk})_{s,t}$ とすれば,

$$\delta_{r,ijk} = \langle \boldsymbol{\delta}_{r+1,k}, W_{r+1,ijk} \rangle \nabla f_r(u_{r,ijk})$$

となり, やはり畳み込み層やこれまでの逆伝播と同じように書けることが見て取れる. プーリング層の逆伝播は Algorithm 11 のように行えば良い.

ところで, max プーリングを行う場合, フィルタのサイズとストライド数が同じであることから, $P_{i,j}$ は各 i,j について共通部分はない, つまり, $P_{i,j} \cap P_{i',j'}$ $(i',j') \neq (i,j)$ となる. このことを利用すると, max プーリングでの W_{r+1} は $d \times d \times K$ の 3 次元配列として実装することができる. つまり, max プーリングで得られる $W_{r+1,ijk}$ をすべての i,j,k にわたって持っておくことはせず,

$$W_{r+1,k} = \sum_{i,j=1}^{d'} W_{r+1,ijk} \in \mathbb{R}^{d \times d}$$

Algorithm 11 プーリング層の逆伝播

入力: $W_{r+1} \in \mathbb{R}^{d\times d\times d'\times d'\times K}$, 誤差 $\boldsymbol{\delta}_{r+1} = (\boldsymbol{\delta}_{r+1,1}, \ldots, \boldsymbol{\delta}_{r+1,K}) \in \mathbb{R}^{d'\times d'\times K}$, 活性化関数の勾配 $\nabla f_r(U_r) \in \mathbb{R}^{d\times d\times K}$

$\tilde{\boldsymbol{\delta}}_r = (\tilde{\delta}_{ijk})_{i,j,k} = \mathbf{0} \in \mathbb{R}^{d\times d\times K}$ ▷ 誤差のもと
for $k = 1$ to K **do**
  for $i = 1$ to d **do**
    for $j = 1$ to d **do**
      $\tilde{\delta}_{r,ijk} \leftarrow \langle \boldsymbol{\delta}_{r+1,k}, W_{r+1,ijk} \rangle$ ▷ $W_{r+1,ij} = (w_{r+1,st,ijk})_{s,t} \in \mathbb{R}^{d'\times d'}$
    end for
  end for
end for

出力: 誤差 $\boldsymbol{\delta}_r = \tilde{\boldsymbol{\delta}}_r \odot \nabla f_r(U_r) \in \mathbb{R}^{d\times d\times K}$

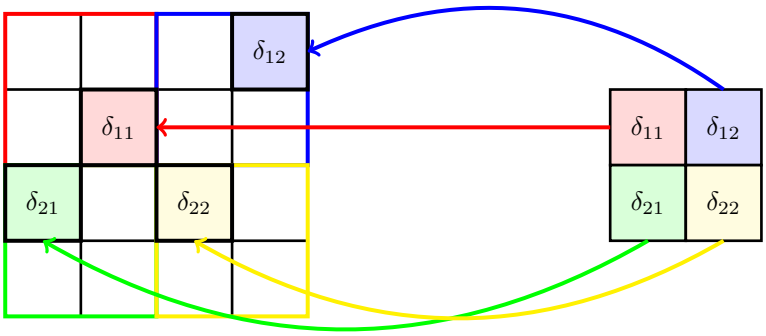


図 26　max プーリングを持ちた場合の逆伝播のイメージ．順伝播で最大値をとったセルに, 誤差が逆伝播される．したがって, 順伝播の際に各フィルタで最大となるセルを記憶しておき, それを W としておけば良い．逆伝播の際にはそのセルに対応する誤差を代入する．

を並べて $W_{r+1} = (W_{r+1,1}, \ldots, W_{r+1,K}) \in \mathbb{R}^{d\times d\times K}$ を用いれば良い．この重み W_{r+1} を用いた際の逆伝播のイメージは図 26 のようになる．ただし, 図 26 で左側にある行列は, この W_{r+1} に $\boldsymbol{\delta}_{r+1}$ を対応させたものである．一般のプーリングでは, やはり $\delta_{r,ijk}$ を計算する際に 5 次元配列が必要となるので, その点は注意すること．

■パラメータの更新　以上で, 畳み込みニューラルネットワークの逆伝播 $\boldsymbol{\delta}_r$ が計算できたことになる．最後に, パラメータの更新規則について説明する．図 23 を元に, $\boldsymbol{\delta}_r \in \mathbb{R}^{d\times d\times K}$ として, パラメータ W_{r+1} を更新することを考えよう．なお, プーリング層における重みは, 逆伝播を構築するために一時的に用意したものであるから, 実際には更新するパラメータではないことに注意する．

まず, 図 23 において, (B) のネットワークを構成した場合に, $W_{r+1} \in \mathbb{R}^{d\times d\times K\times K'}$ および $\boldsymbol{b}_{r+1} \in \mathbb{R}^{K'}$ の更新規則を考えよう．$Z_{r+1} \in \mathbb{R}^{d\times d\times K}$ であるから, $W_{r+1} = (W_{r+1,1,1}, \ldots, W_{r+1,K'}) \in \mathbb{R}^{d\times d\times K\times K'}$ として,

$$u_{r+1,k} = \langle W_{r+1,k}, Z_{r+1} \rangle + b_{r+1,k} = \sum_{l=1}^{K} \sum_{p,q=1}^{d} w_{r+1,pqlk} z_{r+1,pql} + b_{r+1,k}$$

とかける．よって，

$$\frac{\partial E}{\partial w_{r+1,pqlk}} = \frac{\partial E}{\partial u_{r+1,k}} \frac{\partial u_{r+1,k}}{\partial w_{r+1,pqlk}} = \delta_{r+1,k} z_{r+1,pql}$$

であり，これを p, q, l の方向に並べかえることで．

$$\frac{\partial E}{\partial W_{r+1,k}} = \delta_{r+1,k} Z_{r+1} \in \mathbb{R}^{d \times d \times K}$$

となることがわかる．また，同様の計算により，$\boldsymbol{b}_{r+1} = (b_{r+1,1}, \ldots, b_{r+1,K'})^\top \in \mathbb{R}^{K'}$ に対して

$$\frac{\partial E}{\partial b_{r+1,k}} = \frac{\partial E}{\partial u_{r+1,k}} \frac{\partial u_{r+1,k}}{\partial b_{r+1,k}} = \delta_{r+1,k}$$

次に，図 23 の (A) のネットワークで表されるような，畳み込み層のパラメータの更新について述べる．順伝播におけるパラメータ $W_{r+1} = (W_{r+1,1}, \ldots, W_{r+1,K}) \in \mathbb{R}^{H \times H \times K \times K'}$ と Z_r の線形結合は

$$u_{r+1,ijm} = \langle W_{r+1,m}, \tilde{Z}_{r+1,ij} \rangle + b_{r+1,m} = \sum_{k=1}^{K} \sum_{p,q=1}^{H} w_{r+1,pqk,m} \tilde{z}_{r+1,\beta(i-1)+p,\beta(j-1)+q,k} + b_{r+1,m}$$

であることはすでに述べた．ただし，$\tilde{Z}_{r+1,ij}$ は適当なパディング数でゼロパディングした後の入力 Z_{r+1} の $H \times H \times K$ 部分配列である．したがって，誤差関数をパラメータ $w_{r+1,pqk,m}$ で偏微分すると

$$\frac{\partial E}{\partial w_{r+1,pqk,m}} = \sum_{i,j=1}^{d'} \frac{\partial E}{\partial u_{r+1,ijm}} \frac{\partial u_{r+1,ijm}}{\partial w_{r+1,pqk,m}} = \sum_{i,j=1}^{d'} \delta_{r+1,ijm} \tilde{z}_{r+1,\beta(i-1)+p,\beta(j-1)+q,k}$$

であるから，p, q, k の方向に並べて

$$\frac{\partial E}{\partial W_{r+1,m}} = \sum_{i,j=1}^{d'} \delta_{r+1,ijm} \tilde{Z}_{r+1,ij} \in \mathbb{R}^{H \times H \times K}$$

となる．また，

$$\frac{\partial E}{\partial b_{r+1,m}} = \sum_{i,j=1}^{d'} \frac{\partial E}{\partial u_{r+1,ijm}} \frac{\partial u_{r+1,ijm}}{\partial b_{r+1,m}} = \sum_{i,j=1}^{d'} \delta_{r+1,ijm}$$

が得られる．

以上より，これらの勾配を用いて，確率的勾配降下法の更新を行えば，パラメータの推定を実行できる．

3.6 おまけ: いろいろな最適化アルゴリズム

確率的勾配降下法におけるパラメータの更新は，“現在のパラメータの値に対して，勾配方向に学習率の大きさだけ移動する”という単純なものであり，実装も非常に簡単なものだった．ところが，実際には，確率的勾配降下法の収束に時間がかかったり，学習率の選び方などの問題が残っている[*89]．というわけで，本節では，確率的勾配降下法の収束の速さを改善したり，学習率を不要にするためのパラメータ更新法について述べる．なお，パラメータを更新するためのアルゴリズムは**オプティマイザ (optimizer)** とも呼ばれる．

以下で述べるパラメータの更新法は，割と理論的に導出されたものが多く，細かい説明を述べるためにはやや準備が必要なものもあるため，2 章で述べたこと以上の数学が要求される場合には，直感的に理解できる範囲で説明することにする．また，どの手法にも一長一短があり，常にある手法が最良である，ということを述べるものではないことに注意する．つまり，実際にデータ解析を行う場合には，状況に応じて適切なオプティマイザの選択やパラメータチューニングの努力は怠ってはならない．したがって，これから述べることは，このような状況に臨機応変に対応するための選択肢を少しでも増やすことを目標とするものである．

[*89] とはいえ，ニューラルネットワーク以外の普通のモデルだと，収束までの時間はそれほど気にならなかったりする．

Algorithm 12 モーメンタム法

入力: 誤差関数の勾配 ∇E_t, パラメータ $\boldsymbol{w}^{(t-1)}, \boldsymbol{w}^{(t)}$, パラメータ $\alpha \in [0,1)$, 学習率 η (>0)

1: $\Delta^{(t)} = \boldsymbol{w}^{(t)} - \boldsymbol{w}^{(t-1)}$

2: $\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} + \alpha\Delta^{(t)} - \eta\nabla E_t(\boldsymbol{w}^{(t)})$

出力: パラメータ $\boldsymbol{w}$ の更新値 $\boldsymbol{w}^{(t+1)}$

■モーメンタム法 確率的勾配降下法は, 目的関数 $E(\boldsymbol{w})$ をパラメータの現在の値 $\boldsymbol{w}^{(t)}$ のまわりで評価して

$$\begin{aligned}\boldsymbol{w}^{(t+1)} &= \arg\min_{\boldsymbol{w}} \nabla E_t(\boldsymbol{w}^{(t)})^\top(\boldsymbol{w}-\boldsymbol{w}^{(t)}) + \frac{1}{2\eta}\|\boldsymbol{w}-\boldsymbol{w}^{(t)}\|^2 \\ &= \boldsymbol{w}^{(t)} - \eta\nabla E_t(\boldsymbol{w}^{(t)})\end{aligned}$$

とするものであった. これに対し, **モーメンタム法**では, 直前の勾配情報 $\nabla E_{t-1}(\boldsymbol{w}^{(t-1)})$ を慣性[*90]と考え,

$$\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} - \tilde{\alpha}\nabla E_{t-1}(\boldsymbol{w}^{(t-1)}) - \eta\nabla E_t(\boldsymbol{w}^{(t)})$$

によって, パラメータを更新する. 気分としては, “現在のパラメータを更新する前に少し立ち止まって, 前回の勾配方向にもう少し進むことも考慮して更新しましょう”という感じであり, $\boldsymbol{w}^{(t-1)}$ から $\boldsymbol{w}^{(t)}$ への更新の影響が $\tilde{\alpha}$ だけ残っているのである. 実装の際には,

$$\nabla E_{t-1}(\boldsymbol{w}^{(t-1)}) = -\frac{1}{\eta}(\boldsymbol{w}^{(t)} - \boldsymbol{w}^{(t-1)})$$

であることを考慮して,

$$\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} + \frac{\tilde{\alpha}}{\eta}(\boldsymbol{w}^{(t)} - \boldsymbol{w}^{(t-1)}) - \eta\nabla E_t(\boldsymbol{w}^{(t)})$$

とする. 第二項で $\alpha = \tilde{\alpha}/\eta$ として, $\alpha \in [0,1)$ を学習率とは別のパラメータとする (Algorithm 12). なお, $\alpha = 0$ の場合には, これまで通りの, 更新が行われることに注意する. また, 初期ステップ $(t=0)$ では $\Delta^{(t)} = 0$ とする.

■Nesterov の加速法 モーメンタム法では, 直前の勾配情報を慣性としてパラメータの更新で考慮することによって, アルゴリズムの修正を行った. ところで, せっかく慣性を考慮するのだから, 勾配も慣性によって移動した先で評価しようと考えるのは, それほど不自然ではないと思われる. そこで, モーメンタム法で評価される勾配 $\nabla E(\boldsymbol{w}^{(t)})$ の代わりに, 慣性 $\Delta^{(t)} = \boldsymbol{w}^{(t)} - \boldsymbol{w}^{(t-1)}$ を考慮した点 $\boldsymbol{v}^{(t)} = \boldsymbol{w}^{(t)} + \alpha\Delta^{(t)}$ で評価した勾配 $\nabla E_t(\boldsymbol{w}^{(t)} + \alpha\Delta^{(t)})$ を利用して

$$\boldsymbol{w}^{(t+1)} = \boldsymbol{v}^{(t)} - \eta\nabla E_t(\boldsymbol{v}^{(t)}) = \boldsymbol{w}^{(t)} + \alpha\Delta^{(t)} - \eta\nabla E_t(\boldsymbol{w}^{(t)} + \alpha\Delta^{(t)})$$

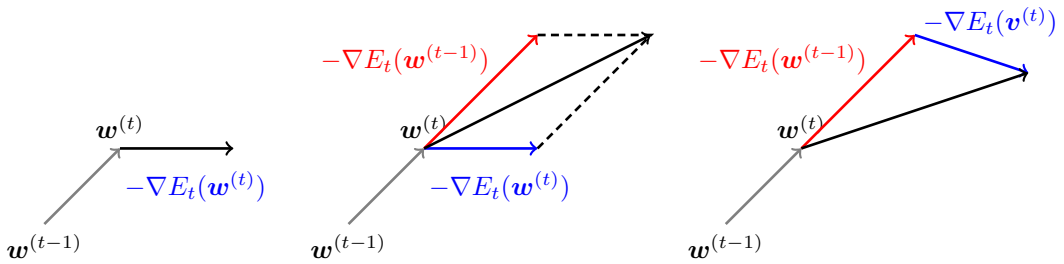
によってパラメータを更新しようというのが **Nesterov の加速 (勾配降下) 法**のアイデアである. 図 27 は, 通常の勾配降下法, モーメンタム法, および Nesterov の加速法によるパラメータ更新の違いを図示したものである. Nesterov の加速法とモーメンタム法の違いは, 勾配を評価する位置のみであるが, 更新後のパラメータの値はそれぞれの手法で異なる.

勾配はステップごとに異なるので, 慣性の強さを表すパラメータ α もステップごとに変化して良い, つまり, $\alpha = \alpha_t$ であるとしよう. このとき, 適当な条件のもとで[*91],

$$\alpha_t = \frac{s_t - 1}{s_{t+1}}, \qquad s_{t+1} = \frac{1 + \sqrt{1 + 4s_t^2}}{2}$$

[*90] 勾配降下法を “坂道からボールを転がすこと”と思えば, ボールは谷底でピタッと止まることはなく, したがって, 残った運動エネルギーの分だけ少し坂を登るイメージを慣性と呼んでいる.

[*91] 誤差関数が γ**-平滑**と呼ばれる条件を満たせば良い.



(a) 通常の勾配降下法　(b) モーメンタム法　(c) Nesterov の加速法

図 27　勾配降下法におけるパラメータ更新のイメージ．それぞれの図で, 灰色の実線は直前の更新, 黒の実線はパラメータの更新方向を表している．図 (b), (c) の赤の実線は直前のパラメータで評価した際の勾配 (慣性の向き) を表しており, 青の実線は, 各手法で考慮す勾配方向を表している．また, 図 (c) において $\boldsymbol{v}^{(t)} = \boldsymbol{w}^{(t)} + \alpha(\boldsymbol{w}^{(t)} - \boldsymbol{w}^{(t-1)})$ である．

Algorithm 13 Nesterov の加速法

入力: 誤差関数の勾配 ∇E_t, パラメータ $\boldsymbol{w}^{(t-1)}, \boldsymbol{w}^{(t)}$, 学習率 η (> 0)

1: $s_{t+1} = (1 + \sqrt{1 + 4s_t^2})/2$
2: $\boldsymbol{v}^{(t)} = \boldsymbol{w}^{(t)} + (s_t - 1)/s_{t+1} \cdot (\boldsymbol{w}^{(t)} - \boldsymbol{w}^{(t-1)})$
3: $\boldsymbol{w}^{(t+1)} = \boldsymbol{v}^{(t)} - \eta \nabla E_t(\boldsymbol{v}^{(t)})$

出力: パラメータ $\boldsymbol{w}$ の更新値 $\boldsymbol{w}^{(t+1)}$

と α_t を更新することで, 通常の勾配降下法に比べアルゴリズムの収束が早くなることが知られている[*92]．初期ステップ $(t = 0)$ では $s_0 = 1$, したがって $\alpha_0 = 0$ としておくことで, はじめのステップは通常の勾配法による更新となる．なお, (厳密には数学的帰納法を用いて示す必要はあるが) グラフを描けばすぐにわかるように,

$$\frac{t-1}{t+2} \leq \frac{s_t - 1}{s_{t+1}} \leq \frac{t-1}{t+1}$$

となることを用いて, s_t の更新は行わず, より簡単に $\alpha_t = (t-1)/(t+2)$ としても良い[*93]．

■AdaGrad　モーメンタム法や Nesterov の加速法は, 誤差関数の勾配のみを利用して更新を行うものだが, **(準) Newton 法**や **Fisher のスコア法**などに代表されるように 2 階微分を利用することでも, 収束を改善する可能性がある．また, 確率的勾配降下法を用いる場合, 学習率のスケジューリングによっても推定値の安定性や収束の速度は変わるため, 学習率をデータから効率的に更新することは有効であろう．**AdaGrad (Adaptive Gradient)** はその中でも初期[*94]に提案された手法であり, (ほぼ) 2 階微分の情報を利用しつつ, 学習率のチューニングを不要にしたパラメータの更新方法である．

以降, この後に述べる RMSProp, AdaDelta, Adam を含め, パラメータは成分ごとに更新されるものとし, $\odot$ で Hadamard 積を表すこととする．また, 許容誤差 ε に対して, $\boldsymbol{v} + \varepsilon = (v_i + \varepsilon)_i$ などと表すことにする[*95]．したがって, ε と書けば, ε が適当な数だけ並んだ行列やベクトルなどの配列だと思うことと同じである．

[*92] 正確には, ステップ T での通常の勾配降下法の収束レートが $\mathrm{O}(1/T)$ であるのに対して, 加速法を用いることで $\mathrm{O}(1/T^2)$ となる．
[*93] このように α_t をとるのは, $(t-1)/(t+2)$ が $(t-1)/(t+1)$ よりも良く $(s_t - 1)/s_{t+1}$ を近似しているためである．
[*94] とは言っても, AdaGrad の初出は 2011 年ではある．
[*95] $\boldsymbol{v}$ はベクトルだと思って書いているが, 当然行列についても同様に定義する．

Algorithm 14 AdaGrad

入力: 誤差関数の勾配 ∇E_t, パラメータ $\boldsymbol{w}^{(t)}$, 初期の学習率 η (> 0)

1: $\boldsymbol{v}_t = \boldsymbol{v}_{t-1} + \nabla E_t(\boldsymbol{w}^{(t)}) \odot \nabla E_t(\boldsymbol{w}^{(t)})$

2: $\boldsymbol{\eta}_t = \eta / \sqrt{\boldsymbol{v}_t}$

3: $\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} - \boldsymbol{\eta}_t \odot \nabla E_t(\boldsymbol{w}^{(t)})$

出力: パラメータ $\boldsymbol{w}$ の更新値 $\boldsymbol{w}^{(t+1)}$, 勾配の 2 次情報 $\boldsymbol{v}_t$

AdaGrad における学習率は以下のように, パラメータを更新する座標ごとに自動的に更新される:

$$\boldsymbol{v}_t = \boldsymbol{v}_{t-1} + \nabla E_t(\boldsymbol{w}^{(t)}) \odot \nabla E_t(\boldsymbol{w}^{(t)}), \qquad \boldsymbol{\eta}_t = \frac{\eta}{\sqrt{\boldsymbol{v}_t}}$$

ただし, η はあらかじめ設定された (初期の) 学習率であり, (すでに述べたように) $\boldsymbol{\eta}_t$ の更新は $\boldsymbol{v}_t$ の成分ごとに行われる. $\boldsymbol{v}_t$ は勾配の 2 次の情報を考慮するものであり, 初期値はいわゆるゼロ割りを避けるために, 十分小さな定数 (10^{-8} 程度で良い) を用いて $\boldsymbol{v}_0 = \varepsilon$ とする. そして, 通常の勾配法と同様に, パラメータを

$$\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} - \boldsymbol{\eta}_t \odot \nabla E_t(\boldsymbol{w}^{(t)})$$

と更新される (Algorithm 14). 通常の確率的勾配降下法における更新とは異なり, 学習率が成分ごとに異なることに注意する. $\boldsymbol{v}_t$ の意味を考えてみよう. 定義から, $\{\boldsymbol{v}_t\}$ は $t = T$ ステップ目において

$$\boldsymbol{v}_t = \varepsilon + \sum_{t=1}^{T} \nabla E_t(\boldsymbol{w}^{(t)}) \odot \nabla E_t(\boldsymbol{w}^{(t)})$$

と表すことができる. したがって, パラメータの更新が進むにつれ, AdaGrad では過去の勾配の 2 次の情報をすべて累積したものが得られることになる.

勾配の 2 乗 $\nabla E_t(\boldsymbol{w}^{(t)}) \odot \nabla E_t(\boldsymbol{w}^{(t)})$ が勾配の 2 階微分の情報を持っていることを簡単に述べておこう. ただし, 以下に述べることは, やや正確性に欠けるものであるため, AdaGrad で実際に行われることを直感的に理解するための “方便”だと思ってもらいたい. まず, 確率的勾配降下法では, 各ステップで誤差を更新するためのデータをランダムにサンプリングすることから, $\nabla E_t(\boldsymbol{w}^{(t)})$ は実際には確率変数である. したがって, 誤差関数の 2 階微分もやはり確率変数であり, $\mathbb{E}[-\nabla^2 E_t(\boldsymbol{w}^{(t)})]$ は勾配の 2 階微分の平均的な振る舞いを表すものと解釈できる[*96]. ただし,

$$\nabla^2 E_t(\boldsymbol{w}^{(t)}) = \left. \frac{\partial^2 E(\boldsymbol{w})}{\partial \boldsymbol{w} \partial \boldsymbol{w}^\top} \right|_{\boldsymbol{w}=\boldsymbol{w}^{(t)}}$$

である. さて, 適当な条件のもとで, 勾配の 2 階微分はの期待値は

$$\mathbb{E}[-\nabla^2 E_t(\boldsymbol{w}^{(t)})] = \mathbb{E}[\nabla E_t(\boldsymbol{w}^{(t)}) \nabla E_t(\boldsymbol{w}^{(t)})^\top]$$

と, 1 階微分のみを用いて書き換えることができる[*97]. ところで, 右辺の期待値の第 i 対角成分だけ抜き出してみてみると

$$(\mathbb{E}[\nabla E_t(\boldsymbol{w}^{(t)}) \nabla E_t(\boldsymbol{w}^{(t)})^\top])_{ii} = (\nabla E_t(\boldsymbol{w}^{(t)}))_i^2$$

とかける. これを縦に並べれば, $\nabla E_t(\boldsymbol{w}^{(t)}) \odot \nabla E_t(\boldsymbol{w}^{(t)})$ となることがわかる. つまり, AdaGrad は, 勾配の 2 階微分の対角成分のみを利用して, 確率的勾配降下法のオプティマイザとして利用できるようにしたものだという解釈ができる.

[*96] $\mathbb{E}$ は**期待値**を表す記号であり, 詳細については確率論の本などを参照してほしい. 大雑把に述べれば, 期待値は確率変数の平均的な振る舞いを代表する値である.

[*97] 右辺を **Fisher 情報量 (行列)** とよぶ.

Algorithm 15 RMSProp

入力: 誤差関数の勾配 ∇E_t, パラメータ $\boldsymbol{w}^{(t)}$, 初期の学習率 η (> 0), 減衰率 β, 学習率の許容誤差 ε

1: $\boldsymbol{v}_t = \beta \boldsymbol{v}_{t-1} + (1-\beta)\nabla E_t(\boldsymbol{w}^{(t)}) \odot \nabla E_t(\boldsymbol{w}^{(t)})$

2: $\boldsymbol{\eta}_t = \eta / \sqrt{\boldsymbol{v}_t + \varepsilon}$

3: $\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} - \boldsymbol{\eta}_t \odot \nabla E_t(\boldsymbol{w}^{(t)})$

出力: パラメータ $\boldsymbol{w}$ の更新値 $\boldsymbol{w}^{(t+1)}$, 勾配の 2 次情報 $\boldsymbol{v}_t$

■RMSProp　AdaGrad における勾配の 2 乗は, 過去の履歴をすべて同じ重みで更新するものであった. 一方で, 過去の履歴は現在までで徐々に薄れるだろうということを考慮すれば, 重み付きで更新すれば, より現在の情報を反映した勾配の更新が行えることが期待出来る[*98]. このようなアイデアの元, AdaGrad における誤差の 2 乗の更新を

$$\boldsymbol{v}_t = \beta \boldsymbol{v}_{t-1} + (1-\beta)\nabla E_t(\boldsymbol{w}^{(t)}) \odot \nabla E_t(\boldsymbol{w}^{(t)})$$

と修正したものが **RMSProp** である[*99]. $\boldsymbol{v}_t$ の初期値は $\boldsymbol{v}_0 = \boldsymbol{0}$ が用いられる, β は例えば 0.9 などと設定される. なお, Algorithm 15 に示したように, 学習率の更新もやや AdaGrad とは異なるが, いずれにしろゼロ割りを避けるための対処である.

RMSProp では, 勾配の 2 乗は**指数移動平均**をとることによって更新される. その結果, 過去の勾配の 2 乗の情報は, 減衰率 β に対して指数的に減少することがわかる. 簡単な例を通して, 勾配の情報が減少する様子を確認しておこう. 表記の簡略化のため, $\Psi_t = \nabla E_t(\boldsymbol{w}^{(t)}) \odot \nabla E_t(\boldsymbol{w}^{(t)})$ と表すことにする. このとき,

$$\begin{aligned}
\boldsymbol{v}_1 &= \beta \boldsymbol{v}_0 + (1-\beta)\Psi_1 = (1-\beta)\Psi_1 \\
\boldsymbol{v}_2 &= \beta \boldsymbol{v}_1 + (1-\beta)\Psi_2 = (1-\beta)(\beta\Psi_1 + \Psi_2) \\
\boldsymbol{v}_3 &= \beta \boldsymbol{v}_2 + (1-\beta)\Psi_3 = (1-\beta)(\beta^2\Psi_1 + \beta\Psi_2 + \Psi_3) \\
\boldsymbol{v}_4 &= \beta \boldsymbol{v}_3 + (1-\beta)\Psi_4 = (1-\beta)(\beta^3\Psi_1 + \beta^2\Psi_2 + \beta\Psi_3 + \Psi_4)
\end{aligned}$$

であり, したがって, T ステップ目では

$$\boldsymbol{v}_t = (1-\beta)\sum_{t=1}^{T} \beta^{T-t}\Psi_t$$

となり, 例えば, Ψ_1 の持つ情報は t ステップ目で β^{T-1} だけ減少することがわかる.

■AdaDelta　**AdaDelta** は RMSProp と同時期に提案されたパラメータ更新法であり, RMSProp と同様に勾配の 2 次の情報を取り入れたものである. RMSProp と異なる点は Algorithm 16 にある通り, 学習率の初期値 η がなくなった点である. 代わりに, $\boldsymbol{s}_t$ を用いることで, より直前の勾配情報を利用した学習率の更新を行っていることがわかる. なお, 原著では $\beta = 0.95, \varepsilon = 10^{-6}$ が推奨されている.

■Adam　AdaGrad や RMSProp, AdaDelta では, 指数移動平均をとることで, 本来更新すべき $\nabla E_t(\boldsymbol{w}^{(t)}) \odot \nabla E_t(\boldsymbol{w}^{(t)})$ に対してバイアスが生じてしまう. どの程度バイアスが生じるかを確認するために, 期待値 $\mathbb{E}[\nabla E_t(\boldsymbol{w}^{(t)}) \odot \nabla E_t(\boldsymbol{w}^{(t)})]$ を考えてみよう. 添字のチョイスはランダムなので, この期待値はステップ数 t に依存しない定数であると考えられる. 先に述べたように, RMSProp や AdaGrad における勾配の 2 乗 $\boldsymbol{v}_t$ の更新では, T ステップ目で

$$\boldsymbol{v}_T = (1-\beta)\sum_{t=1}^{T} \beta^{T-t}\nabla E_t(\boldsymbol{w}^{(t)}) \odot \nabla E_t(\boldsymbol{w}^{(t)})$$

[*98] 背後にある問題として, AdaGrad は急速に学習率が低下するという問題が知られている.

[*99] RMS は “root mean squared”, Prop は “propagation” の略であると思われるが, はっきりと書かれた文献が見当たらない. ご存知の方がいれば教えて欲しい.

Algorithm 16 AdaDelta

入力: 誤差関数の勾配 ∇E_t, パラメータ $\boldsymbol{w}^{(t)}$, 減衰率 β, 学習率の許容誤差 ε

1: $\boldsymbol{v}_t = \beta \boldsymbol{v}_{t-1} + (1-\beta)\nabla E_t(\boldsymbol{w}^{(t)}) \odot \nabla E_t(\boldsymbol{w}^{(t)})$

2: $\boldsymbol{u}_t = \sqrt{\boldsymbol{s}_t + \varepsilon}/\sqrt{\boldsymbol{v}_t + \varepsilon} \odot \nabla E_t(\boldsymbol{w}^{(t)})$

3: $\boldsymbol{s}_{t+1} = \beta \boldsymbol{s}_t + (1-\beta)\boldsymbol{u}_t \odot \boldsymbol{u}_t$

4: $\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} - \boldsymbol{u}_t$

出力: パラメータ $\boldsymbol{w}$ の更新値 $\boldsymbol{w}^{(t+1)}$, 勾配の 2 次情報 $\boldsymbol{v}_t, \boldsymbol{s}_{t+1}$

Algorithm 17 Adam

入力: 誤差関数の勾配 ∇E_t, パラメータ $\boldsymbol{w}^{(t)}$, チューニングパラメータ $\alpha, \beta_1, \beta_2, \varepsilon$

1: $\boldsymbol{m}_t = \beta_1 \boldsymbol{m}_{t-1} + (1-\beta_1)\nabla E_t(\boldsymbol{w}^{(t)})$

2: $\boldsymbol{v}_t = \beta_2 \boldsymbol{v}_{t-1} + (1-\beta_2)\nabla E_t(\boldsymbol{w}^{(t)}) \odot \nabla E_t(\boldsymbol{w}^{(t)})$

3: $\hat{\boldsymbol{m}}_t = \hat{\boldsymbol{m}}_t/(1-\beta_1^t)$

4: $\hat{\boldsymbol{v}}_t = \hat{\boldsymbol{v}}_t/(1-\beta_2^t)$

5: $\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} - \alpha \hat{\boldsymbol{m}}_t/(\sqrt{\hat{\boldsymbol{v}}_t} + \varepsilon)$

出力: パラメータ $\boldsymbol{w}$ の更新値 $\boldsymbol{w}^{(t+1)}$, 勾配の情報 $\boldsymbol{m}_t, \boldsymbol{v}_t$

となる. β はあらかじめ指定される, 非確率的な定数であるが, $\boldsymbol{v}_T$ は過去の $\boldsymbol{w}_t$ に依存するのでランダムであることに注意する. 両辺の期待値を取れば,

$$\mathbb{E}[\boldsymbol{v}_T] = (1-\beta)\sum_{t=1}^{T} \beta^{T-t}\mathbb{E}[\nabla E_t(\boldsymbol{w}^{(t)}) \odot \nabla E_t(\boldsymbol{w}^{(t)})] = (1-\beta^T)\mathbb{E}[\nabla E_t(\boldsymbol{w}^{(t)}) \odot \nabla E_t(\boldsymbol{w}^{(t)})]$$

となる. 最後の等式で, 等比級数の和 $\sum_{t=1}^{T} \beta^{T-t} = (1-\beta^T)/(1-\beta)$ を用いた. したがって, $\boldsymbol{v}_T$ を直接用いる代わりに, $\boldsymbol{v}_T/(1-\beta^T)$ を用いることで, 勾配の 2 乗の情報を正しく利用できることが期待される.

Adam はこのアイデアを用いてパラメータ更新を行うオプティマイザであり, 勾配の情報も指数移動平均によって更新するものである. したがって, 勾配の 2 乗の更新

$$\boldsymbol{v}_t = \beta_2 \boldsymbol{v}_{t-1} + (1-\beta_2)\nabla E_t(\boldsymbol{w}^{(t)}) \odot \nabla E_t(\boldsymbol{w}^{(t)})$$

に加え, 勾配の更新

$$\boldsymbol{m}_t = \beta_1 \boldsymbol{m}_{t-1} + (1-\beta_1)\nabla E_t(\boldsymbol{w}^{(t)})$$

も行う. ただし, β_1, β_2 は過去の履歴をどの程度考慮するかを表す定数である. そして, $\boldsymbol{m}_t$ と $\boldsymbol{v}_t$ の更新に伴うバイアスを

$$\hat{\boldsymbol{m}}_t = \frac{\boldsymbol{m}_t}{1-\beta_1^t}, \qquad \hat{\boldsymbol{v}}_t = \frac{\boldsymbol{v}_t}{1-\beta_2^t},$$

によって修正する. 最後に, 勾配更新の大きさ α とゼロ割りを防ぐための許容誤差 ε を用いて

$$\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} - \alpha \frac{\hat{\boldsymbol{m}}_t}{\sqrt{\hat{\boldsymbol{v}}_t} + \varepsilon}$$

によってパラメータを更新する (Algorithm 17). なお, チューニングパラメータとして $\alpha = 0.001, \beta_1 = 0.9, \beta_2 = 0.999, \varepsilon = 10^{-8}$ を用いることが推奨されている.