AI 勉強会（第 2 回）

木更津工業高等専門学校
大枝真一

2017 年 10 月 16 日

1　はじめに

第 1 回 AI 勉強会では，ニューラルネットワークの基本である前向き計算と学習法である BP 法の説明を行った．第 2 回 AI 勉強会では，学習の効率化とコーディングについて説明する．

2　第 1 回の復習

2.1　ニューラルネットワークの前向き計算

ニューロンは複数の入力を受け取り，1 つの値を出力する．ある上位層 J のニューロン j は，下位層 I の I 個のニューロンから入力 o_i を受け取り，ニューロン j と i との結合荷重 w_{ij} との重み付き和を計算する．その重み付き和からニューロン j の閾値 θ_j を加えた値を活性化関数（応答関数）の引数としてその値を出力する．これらを式で表すと以下のようになる．

$$X_j = \sum_{i=1}^{I} w_{ij} o_i + \theta_j \tag{1}$$

$$o_j = f(X_j) \tag{2}$$

ただし，仮想ニューロンを導入することで，閾値 θ_j のパラメータを結合荷重として考慮することが可能となる．つまり，以下のように修正できる．

$$X_j = \sum_{i=1}^{I+1} w_{ij} o_i \tag{3}$$

$$o_j = f(X_j) \tag{4}$$

ここでは，活性化関数にはシグモイド関数を用いることにする．

$$f(x) = \frac{1}{1 + \exp(-\epsilon x)} \tag{5}$$

$$\tag{6}$$

この関数の微分は，式変形のテクニックを使うと次のようになる．

$$f'(x) = (1 - f(x))f(x) \tag{7}$$

$$\tag{8}$$

2.2 学習の目的

ニューラルネットワークは教師あり学習であり，ある入力に対して対応する値が出力されるように学習する．各出力ニューロン k に対応する教師信号 t_k が与えられたとき，各入力パターンに対する誤差 E_p と P 個の入力パターンの誤差の総和 E_{all} は以下の式で表すことができる．

$$E_p(w) = \frac{1}{2}\sum_{k=1}^{K}(o_k - t_k)^2 \tag{9}$$

$$E_{all}(w) = \sum_{p=1}^{P} E_p \tag{10}$$

ニューラルネットワークの学習は，この誤差総和 E_{all} を最小化し，入力に対して適切な出力が得られるように各結合重みを調整することである．そのために，$E_p(w)$ を最急降下法を用いて最小化する．つまり，次のようになる．

$$w^{(t+1)} = w^{(t)} - \eta\frac{\partial E_p(w^{(t)})}{\partial w^{(t)}} \tag{11}$$

2.3 BP 法

第 1 回 AI 勉強会では $\frac{\partial E_p(w_t)}{\partial w_t}$ の具体的な導出を行ったので，ここでは式だけを掲載する．活性化関数としてはシグモイド関数を用いる．

中間-出力層

$$w_{jk} = w_{jk} - \eta\delta_k o_j \tag{12}$$

$$\delta_k = (o_k - t_k)\epsilon(1 - o_k)o_k \tag{13}$$

中間層以下

$$w_{ij} = w_{ij} - \eta\delta_j o_i \tag{14}$$

$$\delta_j = \sum_{k=1}^{K} \delta_k w_{jk} \epsilon (1 - o_j) o_j \tag{15}$$

3　ニューラルネットワークのコーディング

導出した式をコーディングするにはコツが必要である．すぐできる人とそうでない人の差が生じるので，苦手な人のために，まずは「手抜き」コーディングを行う．

練習課題として XOR を学習するプログラムを作成する．XOR とは以下の表のような論理演算である．

表 1　XOR の論理演算

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0

つまり，ネットワーク構造は入力層が 2 個，出力層が 1 個のニューラルネットワークとなる．ここでは簡単のため中間層は 1 層で 2 個のニューロンを持つとする．つまり，3 層構造の 2-2-1 のネットワークとなる．

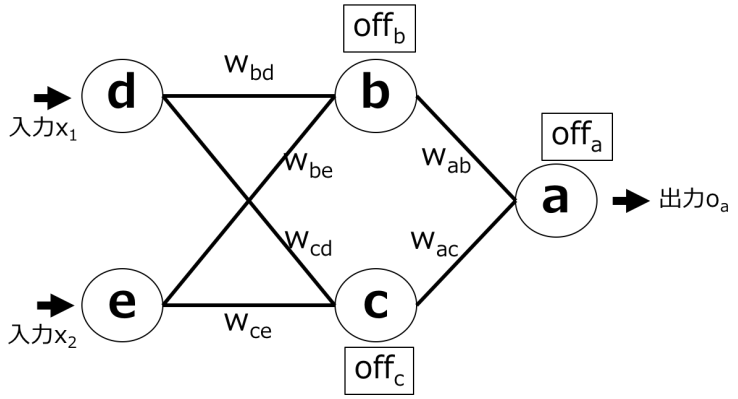

図 1　XOR を学習するネットワーク

ここで，各ニューロンに名前 A から E を与え，重みと閾値を図 1 のような変数名を与えること

とする．このときの具体的な計算を手動で行うことができれば，プログラムを作成できる．

図 2 のように重みと閾値にランダムに値を与える．また，シグモイド関数の傾き $\epsilon = 4.0$ とする．

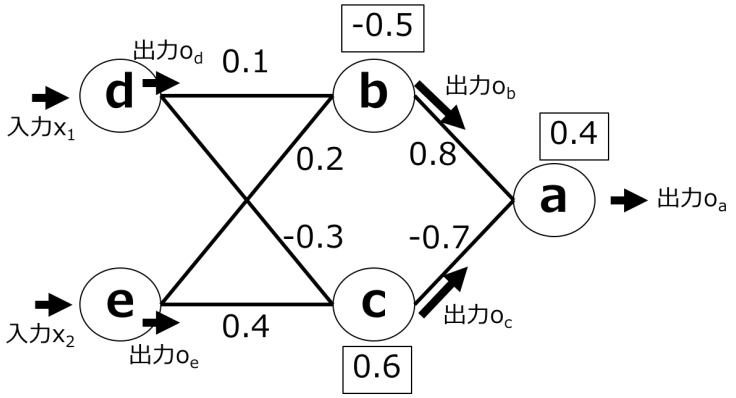

図 2　初期重みと閾値

3.1　前向き計算

前向き計算を行う．第 1 パターンの入力である $x_1 = 0.0, x_2 = 0.0$ を入力する．入力ニューロンは応答関数に通さないので，$o_d = x_1 = 0.0, o_e = x_2 = 0.0$ となる．次に，ニューロン b の出力 o_b を計算する．

$$X_j = \sum_{i=1}^{I} w_{ij} o_i + \theta_j \tag{16}$$

$$o_j = f(X_j) \tag{17}$$

より，

$$X_b = (0.1 \times 0.0) + (0.2 \times 0.0) + (-0.5) \tag{18}$$

$$= -0.5 \tag{19}$$

$$o_b = f(-0.5) \tag{20}$$

$$= 0.11920292202211755 \tag{21}$$

となる．

これを確かめるために，実際にプログラムを作成して計算してみよう．プログラムは以下の URL に置いてある．

サンプルプログラム

`https://github.com/crotsu/Study_AI`

この中の，easy_ff.py を用いる．

表 2　前向き計算（パターン 1 だけ）

x_1	x_2	y	o_e	o_d	o_c	o_b	o_a
0	0	0	0.0	0.0	0.12	0.92	0.36
0	1	1	0.0				
1	0	1	1.0				
1	1	0	1.0				

続いて，同様にニューロン c, a の出力 o_c, o_a を計算すると表 2 のようになる．

これを全パターンで前向計算を行うと表 3 のようになる．これは，feed_forward.py を用いると確認できる．また，学習済みの重みを与えると，前向き計算で XOR が正しく出力されていることが確認できる．

表 3　前向き計算（全パターン）

x_1	x_2	y	o_e	o_d	o_c	o_b	o_a
0	0	0	0.0	0.0	0.12	0.92	0.36
0	1	1	0.0	0.1	0.23	0.98	0.40
1	0	1	1.0	0.0	0.17	0.77	0.50
1	1	0	1.0	1.0	0.31	0.94	0.49

3.2　学習部分（BP 法による重みの修正）

重みを修正する部分のコーディングを行う．まず，各ニューロンの δ を計算し，次に重みを更新する．式をそのままコーディングするだけだが，慣れるまではややこしい．ひとつひとつ確認しながらコーディングする．

プログラムは bp.py になる．初期重みによっては収束しないことがある．実行結果として，ターミナル上に各学習時の計算結果，そして学習曲線（誤差曲線）が matplotlib.pyplot によって画面に表示される．横軸に学習回数（1epoch），縦軸に誤差（全 4 パターンの誤差の合計）が出力される．

4　学習の効率化

ニューラルネットワークの学習には非常に時間がかかる．そこで，学習効率化のために多くの手法が提案されている．ここでは，そのうちの手法をいくつか紹介する．

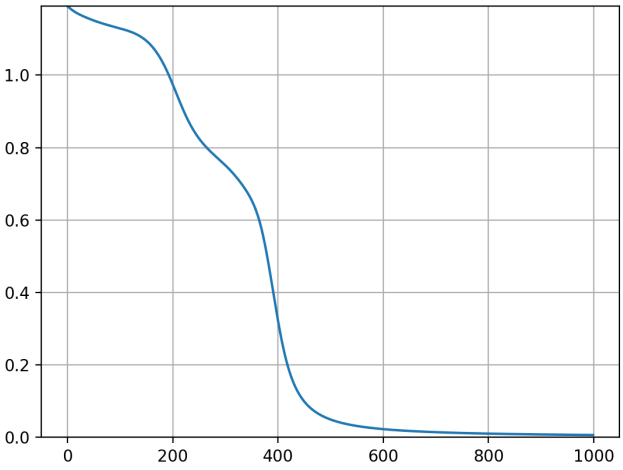

図 3　XOR を学習したときの誤差曲線（成功）

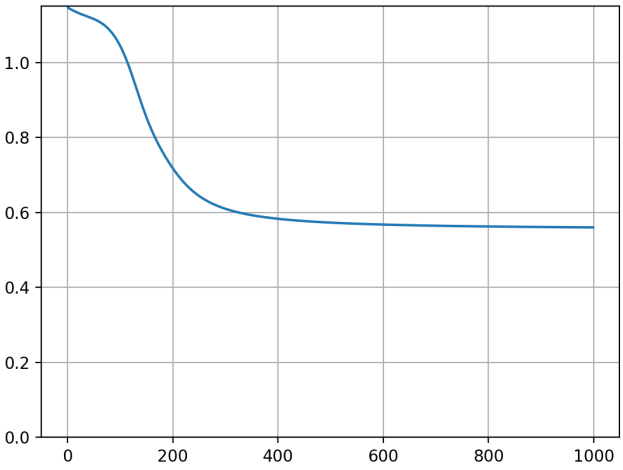

図 4　XOR を学習したときの誤差曲線（失敗）

4.1　逐次修正法

これまでに紹介した BP 法は，逐次修正法と呼ばれる手法である．トレーニングデータの内，パターンを一つ与えられてて，その都度修正する方法である．この手法は学習の効率化の手法ではないが，次に紹介する一括修正法との対比のため命名している．

また，逐次修正法は，そのパターンに対する学習しか行っておらず，全パターンの最小化になってはいない．そこで，全パターンを 1 つずつ順番に与えるのではなく，ランダムに与えることで解

決する．

4.2 一括修正法

一括修正法は，トレーニングデータの全パターンが与えられた後，一括して重みを修正する．逐次修正法のようにパターンを一つ与えられてて，各重みの修正量を計算するが修正は行わず，その修正量を蓄積しておく．そして，全パターンが与えられたら一括して修正するのである．一括修正法は E_{all} を極小化していることになるので学習速度が速くなる．しかし，トレーニングデータのパターン数が多い場合，1 回の修正量が大きくなり，収束しないことがある．

4.3 ミニバッチ法

ミニバッチ法は，逐次修正法と一括修正法の良いところを併せ持つ手法である．全パターンの内，ミニバッチとして少数データにわける．このミニバッチに対して一括修正を行う．つまり，ミニバッチサイズが 1 のとき，逐次修正法になり，全データのとき一括修正法になる．

ミニバッチサイズに依存する手法であるが，MNIST と呼ばれる文字認識の例では，全 5 万データに対してミニバッチサイズは 20 である．

また，学習回数をカウントするときに，パターンの提示回数なのか修正回数なのかによって，学習回数の比較ができなくなることがある．そこで，epoch と呼ばれる単位を導入して，学習回数とすることがある．epoch は定義によってことなるが一般的にはミニバッチ 1 回を 1epoch と呼ぶことが多い．

4.4 モーメント法

BP 法は重みを $\frac{\partial E(w)}{\partial w}$ に比例して修正する方法であるが，これを厳密に実行すると収束には無限回の修正が必要となる．学習を行えば行うほど，修正量が小さくなり修正されなくなるからである．実際にシステムを構築する場合，学習速度を高める必要があり，その方法としてモーメント法がある．モーメント法は重みの修正量に慣性項を付け加える方法である．

$$\delta w^{(t)} = \frac{\partial E_p(w^{(t)})}{\partial w^{(t)}} w^{(t+1)} = w^{(t)} - \eta \delta w^{(t)} + \alpha \delta w^{(t-1)} \tag{22}$$

慣性係数 α は 0 以上とする．経験上 0.8 くらいが良い．慣性係数は前回の重みの変化が今回の重みの変化に影響を度合いを定めている．この慣性項を付け加えることにより重みの変化に一種の慣性が生じ，誤差局面の細かな凹凸を無視するという効果を期待できる．

4.5 計算機実験による比較

プログラムは bp_withMomentum.py になる．4 パターンが提示されたときに 1epoch とカウントする．図 5 では逐次修正法，一括修正法，一括修正法とモーメント法を組み合わせた手法の 3 つ

を比較している．

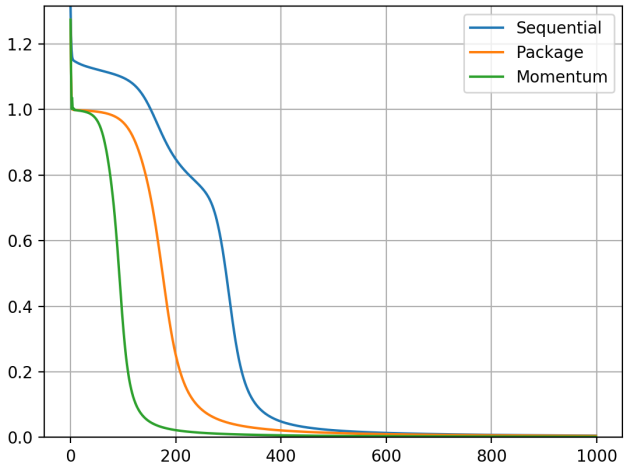

図 5　逐次修正法，一括修正法，モーメント法の比較

5　過学習と汎化性能

機械学習の教師あり学習では過学習に気をつける必要がある．学習対象となっているデータを入手して，学習を行うが，目標は将来の入手する未知データの分類である．手元のデータを精度良く分類するわけではない．つまり，トレーニングデータに対する正答率よりも，テストデータに対する正答率が高いニューラルネットワークの方が良いネットワークと言える．これを汎化性能が高いという．一般的に機械学習ではトレーニングデータを学習すればするほど，未知のテストデータの分類精度が落ちてしまう過学習と呼ばれる現象が生じる．そこで，トレーニングデータで学習をした直後に，テストデータを与えて汎化性能を測る．このとき，テストデータは決して学習に用いてはならない．これを繰り返して，トレーニングデータに対する誤差が十分に小さくなり，かつテストデータに対する誤差が上昇し始めたときに学習を打ち切る方法がある．これは early stopping と呼ばれ，よく用いられる手法である．

6　ベクトルと行列による表現

これまでわかりやすさを優先してスカラー表現で記述してきたが，実はベクトルと行列による表現を用いると，もっと簡単に数式で表すことができる．また，Python などベクトルや行列を扱うことのできるプログラミング言語を用いると，やはり簡単にコーディングできる．ただし，初学者にはわかりにくいため，慣れるまではスカラー表現の方が良いと思う．

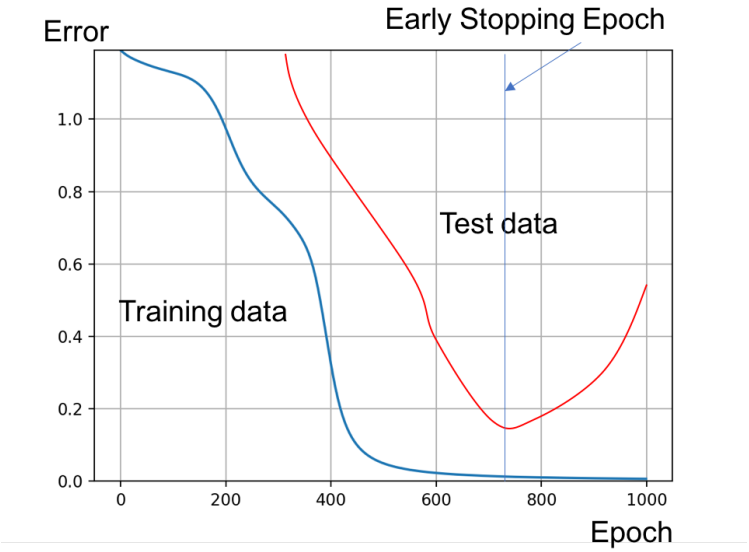

図 6　early stopping

前向き計算は，スカラー表現では次のように表していた．

$$X_j = \sum_{i=1}^{I} w_{ij} o_i \tag{23}$$

$$o_j = f(X_j) \tag{24}$$

これをベクトルと行列を用いると各層をひとまとめに表すことができる．

$$\boldsymbol{X} = \boldsymbol{W}\boldsymbol{o}_I \tag{25}$$

$$\boldsymbol{o}_J = \boldsymbol{f}(\boldsymbol{X}) \tag{26}$$

ただし，各ベクトルと行列は以下のように定義する．

$$\boldsymbol{o}_I = \begin{pmatrix} o_1 \\ o_2 \\ \vdots \\ o_I \end{pmatrix}$$

$$\boldsymbol{W} = \begin{pmatrix} w_{11} & w_{12} & \dots & w_{1I} \\ w_{21} & w_{22} & \dots & w_{2I} \\ \vdots & \vdots & \ddots & \vdots \\ w_{J1} & w_{J2} & \dots & w_{JI} \end{pmatrix}$$

付録 A　BP 法のプログラム

```
# ニューラルネットワークの法による学習BP
# 練習用のため拡張性がないアホアホプログラミング
#
# by oeda

import numpy as np
import matplotlib.pyplot as plt

# パラメータ
EPSILON = 4.0
ETA = 0.1
TIME = 1000

# シグモイド関数
def sigmoid(x):
    return 1/(1+np.exp(-1*EPSILON*x))


# トレーニングデータ
## 入力
inputs = [[0,0],
          [0,1],
          [1,0],
          [1,1]]
## 教師信号
teach = [0,1,1,0]


# 初期重みをランダムに与える
wab = (np.random.rand()-0.5)*2 * 0.3 # からの一様乱数-0.30.3
wac = (np.random.rand()-0.5)*2 * 0.3
wbd = (np.random.rand()-0.5)*2 * 0.3
wbe = (np.random.rand()-0.5)*2 * 0.3
wcd = (np.random.rand()-0.5)*2 * 0.3
wce = (np.random.rand()-0.5)*2 * 0.3
offa = (np.random.rand()-0.5)*2 * 0.3
offb = (np.random.rand()-0.5)*2 * 0.3
offc = (np.random.rand()-0.5)*2 * 0.3

x = []
y = []
# 学習
for t in range(TIME):

    errorAll = 0.0
    # 各パターンを提示
    for p in range(len(inputs)):

        ##########
```

```
        # 前向き計算
        ##########

        # 入力層
        outd = inputs[p][0]
        oute = inputs[p][1]

        # 中間層
        xb = wbd * outd + wbe * oute + offb
        outb = sigmoid(xb)

        xc = wcd * outd + wce * oute + offc
        outc = sigmoid(xc)

        # 出力層
        xa = wab * outb + wac * outc + offa
        outa = sigmoid(xa)

        error = (outa-teach[p])**2
        print(teach[p], outa, error)

        errorAll += error

        ##################
        # Back Propagation
        ##################

        deltaa = (outa-teach[p]) * EPSILON * (1.0-outa) * outa
        deltab = deltaa * wab * EPSILON * (1.0-outb) * outb
        deltac = deltaa * wac * EPSILON * (1.0-outc) * outc

        wab = wab - ETA * deltaa * outb
        wac = wac - ETA * deltaa * outc
        offa = offa - ETA * deltaa

        wbd = wbd - ETA * deltab * outd
        wbe = wbe - ETA * deltab * oute
        offb = offb - ETA * deltab

        wcd = wcd - ETA * deltac * outd
        wce = wce - ETA * deltac * oute
        offc = offc - ETA * deltac

    print(errorAll)
    print()

    # グラフ表示用の変数
    x.append(t)
    y.append(errorAll)

# グラフ表示
# 点どうしを直線でつなぐ
plt.plot(x, y)
```

```
# 適切な表示範囲を指定
ymin = 0.0
ymax = y[0]
plt.ylim(ymin, ymax)
# グリッド追加
plt.grid(True)
# 表示
plt.show()
```

付録 A　BP 法のプログラム (ベクトルと行列表現)

```
# ニューラルネットワークの法による学習BP
# ベクトルと行列を用いたプログラム
#
# by oeda

import numpy as np
import matplotlib.pyplot as plt

# パラメータ
EPSILON = 4.0
ETA = 0.1
TIME = 1000

# シグモイド関数
def sigmoid(x):
    return 1/(1+np.exp(-1*EPSILON*x))


# トレーニングデータ
## 入力
inputs = [[0,0],
          [0,1],
          [1,0],
          [1,1]]
## 教師信号
teach = [0,1,1,0]

# 初期重みをランダムに与える
np.random.seed(6)
weight1 = (np.random.rand(2, 2)-0.5)*2 * 0.3 # からの一様乱数-0.30.3
weight2 = (np.random.rand(1, 2)-0.5)*2 * 0.3
offset1 = (np.random.rand(2)-0.5)*2 * 0.3
offset2 = (np.random.rand(1)-0.5)*2 * 0.3

x = []
y = []
# 学習
for t in range(TIME):

    errorAll = 0.0
    out = []
    # 各パターンを提示
    for p in range(len(inputs)):
        # 前向き計算
        out1 = sigmoid(np.dot(weight1, inputs[p])+offset1)
        out2 = sigmoid(np.dot(weight2, out1)+offset2)
        out.append(out2)
        errorAll += (out2-teach[p])**2
```

```
        # BP
        delta2 = (out2-teach[p])*EPSILON*out2*(1.0-out2)
        weight2 -= ETA*delta2*out1
        offset2 -= ETA*delta2

        delta1 = EPSILON*out1*(1.0-out1)*delta2*weight2
        weight1 -= ETA*delta1*inputs[p]
        offset1 -= ETA*delta1[0]
    print(errorAll)

    # グラフ表示用の変数
    x.append(t)
    y.append(errorAll)

print('output')
print(out)

# グラフ表示
# 点どうしを直線でつなぐ
plt.plot(x, y)
# 適切な表示範囲を指定
ymin = 0.0
ymax = y[0]
plt.ylim(ymin, ymax)
# グリッド追加
plt.grid(True)
# 表示
plt.show()
```