第 4 章 ニューラルネットワークの基礎

[はじめに]
昨今、人工知能あるいは AI という言葉が様々な場面で出てくるようになっている．人工知能や AI という言葉は頻繁に使われるにも関わらず、その定義はあいまいである上に、これまでこの分野ではブームと幻滅が繰り返されてきたと言われている．ここではそれを云々するのは本意ではない．ただ、認識や生成において優れた結果をもたらす技術が生まれ、ヒトをはじめとする生き物の「知能」の定義がなんであるのかを改めて問いなおさなければならないほどに、その結果はヒトのそれに迫り、あるいは超えるものとなっている．そして実用化を軸にさらに発展しつつあって、今後の社会に大きな影響を及ぼすであろうことは確かである．
これを念頭に置きつつ、この核となっている技術を学んでいくことにする．そしてもはやあまり顧みられることのないニューロン層が 1 層だけの単純なパーセプトロンを、数式との対応をとりながら、作っていくことにする．その過程で具体的なデータを用いて回帰問題や分類問題そして画像認識に取り組んでいく．そうして最後にそこから、多層化とともに非線形要素を入れて、ニューラルネットワークとして発展につないでいく．とはいえ昨今のブームを巻き起こしディープラーニングと言われるような大規模なニューラルネットワークを構成するには、そのうえさらに、さまざまな技術要素が必要であるから、それについてはここでは範疇外とする．しかし大規模なニューラルネットワークの構成要素やディープラーニングにまつわる技術を理解するうえでの、しっかりとした土台を築くことを期して、あえて単純なパーセプトロンについて多くの紙面を費やした．と同時に作るプログラムが、勉強のための一回限りの使い捨てではなく、やがてフレームワークとなっていくように構成を考えたつもりだから、多少の回り道は御容赦願えれば幸いだ．とまれ本書がニューラルネットワークそしてディープラーニングへの良き導きとなることを願うものである．

4.1 ニューロン

4.1.1 ニューロンとは
ニューラルネットワークは、ヒトをはじめとする生物の脳の神経細胞のネットワーク構造を模した数学モデルであって、その最小の単位はニューロンである．そのニューロンは、入力に反応して出力するが、その際、入力はそれぞれの重みがかかって足し合わされ、さらに、バイアスが加わった値が、ニューロンの興奮を決める．重みは文字通り、どの入力からの信号を重視するのかを決定づけ、バイアスは入力とその重みによって決まる値を、かさ上げしたり減じたりして、ニューロンの興奮度合いを調節する．そしてそこで入力とバイアスから決まる値、

に、出力がどう対応するかの関係を決める関数を活性化関数という．これを図 4.1.1 に示す．

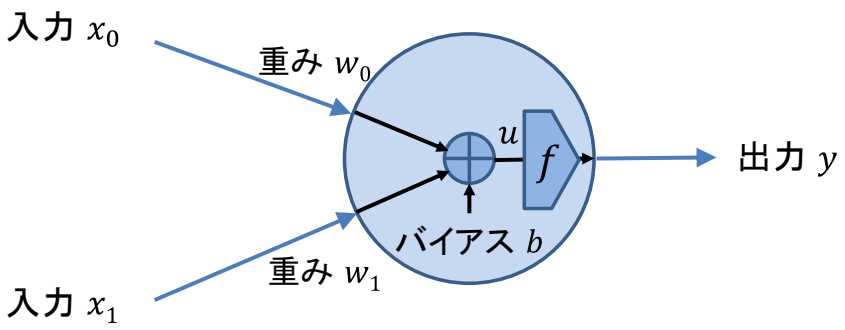

図 4.1.1 ニューロン

図 4.1.1 に示すように、ニューロンの入力を x_0 ，x_1 とし、それぞれに対応する重みを w_0 ，w_1　バイアスを b とし、活性化関数を f であらわせば、その出力 y は、中間値を u として次式で表される．

$$u = x_0 \times w_0 + x_1 \times w_1 + b$$
$$y = f(u) \qquad (4.1.1)$$

このニューロンが並列に複数個あって、個々のニューロンに同じ入力がつながっている場合を考える．

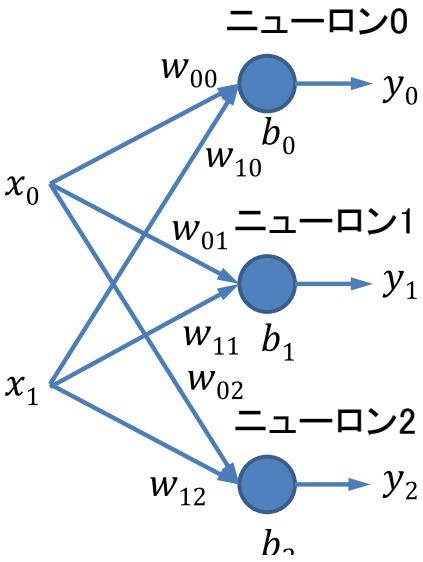

図 4.1.2 複数ニューロン

図 4.1.2 には、入力 x_0 ， x_1 が 3 つのニューロンにつながっている場合を示す．2 つの入力に対応する 2 つの重みとバイアスが、それぞれニューロン毎にある．すなわちニューロンに番号を付けてニューロン 0、ニューロン 1、ニューロン 2 と呼ぶと、ニューロン 0 の入力 x_0 に対する重みが w_{00} 、入力 x_1 に対する重みが w_{10} 、そしてバイアスが b_0 である．同様にそれぞれ入力 x_0 、x_1 に対する重みとバイアスは、ニューロン 1 で w_{01} 、w_{11} と b_1 、ニューロン 2 で、w_{02} 、w_{12} と b_2 である．各ニューロンの中間値と出力を u_0 、 u_1 、 u_2 および、 y_0 、y_1 、y_2 とする．式 4.1.1 の関係が、3 つのニューロンのそれぞれで成り立つとすれば、これらの変数の関係は次式であらわされる.

$$\begin{aligned} u_0 &= x_0 \times w_{00} + x_1 \times w_{10} + b_0 \\ u_1 &= x_0 \times w_{01} + x_1 \times w_{11} + b_1 \\ u_2 &= x_0 \times w_{02} + x_1 \times w_{12} + b_2 \\ y_0 &= f(u_0) \\ y_1 &= f(u_1) \\ y_2 &= f(u_2) \end{aligned} \tag{4.1.2}$$

各変数の添字の並びに注意しておきたい．とまれ、重みとバイアスの違いによって 3 つのニューロンはそれぞれ異なる出力を生成する.
式 4.1.2 はベクトルと行列を使って、次のように表すことができる.

$$\begin{aligned} (u_0 \quad u_1 \quad u_2) &= (x_0 \quad x_1)\begin{pmatrix} w_{00} & w_{01} & w_{02} \\ w_{10} & w_{11} & w_{12} \end{pmatrix} + (b_0 \quad b_1 \quad b_2) \\ (y_0 \quad y_1 \quad y_2) &= f(u_0 \quad u_1 \quad u_2) \end{aligned} \tag{4.1.3}$$

図 4.1.2 では縦に並んだ 2 つの入力が式 4.1.3 では横に並んだベクトル、また図 4.1.2 では縦に 3 つ並んだニューロンが式 4.1.3 では重みもバイアスも出力も横に並んでいるから、少々わかりにくいかもしれないが、図と式の関係を把握してほしい.
さて、式 4.1.3 は入力が 2 つ、ニューロンが 3 つの場合だが、一般的に入力が m 個、ニューロンが n 個の場合について同様に、

$$(u_0 \quad u_1 \quad \cdots \quad u_n) = (x_0 \quad x_1 \quad \cdots \quad x_m)\begin{pmatrix} w_{00} & w_{01} & \cdots & w_{0n} \\ w_{10} & w_{11} & \cdots & w_{1n} \\ \vdots & \vdots & & \vdots \\ w_{m0} & w_{m1} & \cdots & w_{mn} \end{pmatrix} + (b_0 \quad b_1 \quad \cdots \quad b_n)$$

$$(y_0 \quad y_1 \quad \cdots \quad y_n) = f(u_0 \quad u_1 \quad \cdots \quad u_n) \tag{4.1.4}$$

式 4.1.3、式 4.1.4 で、個々のニューロンにおいて、入力の重みづけ和にバイアスを足した中間値の u は、いわば入力の線形結合となっている．そしてその線形結合の結果に活性化関数による変換が行われていることになる．いっぽう線形回帰ではふつう、入力を基底関数によって変換し、しかも例えば多項式近似では x^0、x^1、x^2、・・・といったように、複数の基底関数によって多様な変換を入力に対して行ってから、重みづけ和を出力として、入力と出力の多様な関係に対応する．これに対して、ここで見てきたニューロンでは基本的に入力そのままの重みづけ和をとって、そのあとから活性化関数によって所望の応答を得ようとするという違いがある．

さてここでは入力も出力も複数だが、入力としてどこからくる信号かの違い、出力もどこから出てくる信号かの違い、で複数となっている．そして、いずれもベクトルとして一括して扱うわけだが、そのベクトルが場合によって異なることを考えることができる．例えばある時点では入力のベクトルが、これこれしかじかの値となっているが、別の場合には、入力のベクトルがそれとは異なる値となっているというように．そして出力のベクトルも入力のベクトルの違いに応じて別の値になるわけである．この場合によって違う入出力のベクトルの数はデータ数とかサンプル数ということにしよう．そして、そういうことを扱おうとするならば、入力も出力もベクトルを並べて行列の形であらわすのが都合が良い．入力や出力が k 通りの場合、すなわちデータ数を k として、

$$\begin{pmatrix} u_{00} & u_{01} & \cdots & u_{0n} \\ u_{10} & u_{11} & \cdots & u_{1n} \\ \vdots & \vdots & & \vdots \\ u_{k0} & u_{k1} & \cdots & u_{kn} \end{pmatrix} = \begin{pmatrix} x_{00} & x_{01} & \cdots & x_{0m} \\ x_{10} & x_{11} & \cdots & x_{1m} \\ \vdots & \vdots & & \vdots \\ x_{k0} & x_{k1} & \cdots & x_{km} \end{pmatrix}\begin{pmatrix} w_{00} & w_{01} & \cdots & w_{0n} \\ w_{10} & w_{11} & \cdots & w_{1n} \\ \vdots & \vdots & & \vdots \\ w_{m0} & w_{m1} & \cdots & w_{mn} \end{pmatrix} + \begin{pmatrix} b_0 & b_1 & \cdots & b_n \\ b_0 & b_1 & \cdots & b_n \\ \vdots & \vdots & & \vdots \\ b_0 & b_1 & \cdots & b_n \end{pmatrix}$$

$$\begin{pmatrix} y_{00} & y_{01} & \cdots & y_{0n} \\ y_{10} & y_{11} & \cdots & y_{1n} \\ \vdots & \vdots & & \vdots \\ y_{k0} & y_{k1} & \cdots & y_{kn} \end{pmatrix} = f\begin{pmatrix} u_{00} & u_{01} & \cdots & u_{0n} \\ u_{10} & u_{11} & \cdots & u_{1n} \\ \vdots & \vdots & & \vdots \\ u_{k0} & u_{k1} & \cdots & u_{kn} \end{pmatrix} \tag{4.1.5}$$

式 4.1.5 において、入力、中間値、出力が、行ごとーすなわちベクトルごとに式 4.1.4 と同じ関係があらわされている．このとき重みの行列は式 4.1.4 と変わらない．いっぽうバイアスについては、同じベクトルが加算されるのだが、行列の加算は同じ形状どうしなければできない

から、要素数 n のベクトルを行方向に拡張して k 行 n 列の行列にしている．このいくつかのデータをまとめて扱うのをバッチ処理という．言い換えれば入力や出力の行方向はバッチ処理の各データをあらわし、列方向が別々の入力または出力をあらわすということである．重みの行列 W では各行が各入力に対応し、各列が各ニューロンに対応することと併せて、各行列やベクトルの形状について、しっかり把握しておこう．

さて、式 4.1.4 ないしは式 4.1.5 は、ベクトルや行列にそれぞれ大文字の太字の 1 つの記号を与えて、次のようにあらわすことができる．

$$\boldsymbol{U} = \boldsymbol{X} * \boldsymbol{W} + \boldsymbol{B}$$
$$\boldsymbol{Y} = \boldsymbol{f}(\boldsymbol{U}) \tag{4.1.6}$$

なお行列積は乗法記号は特に用いずに併置であらわすことが多いが、ここでは便宜上、記号 $*$ でベクトルと行列の積や、行列同士の積をあらわすこととする．

4.1.2 ニューロンを python のプログラムで表す

ではここまでのところを python のプログラムとしておこう．ここまでの範囲ならば簡単に記述することもできる．しかしここでは、先々のニューラルネットワークの発展を考えて、敢えて全体構成に必要な形から作っていくこととする．

ここまでの説明にはないが、ニューロンにはいくつかのバリエーションがあるから、それらに共通な部分と、ここで説明したニューロンの機能を、層としてそれぞれ基底クラスの BaseLayer とその派生クラスの NeuronLayer に分けて実装していく．また、活性化関数は別のファイルにモジュールとして用意することとする．

まずは、BaseLayer の実装をリスト 4.1.1 に示す．これはニューロンの機能を作っていくので、ファイル Neuron.py とすることにしよう．そしてこの Neuron.py は、これからニューラルネットワークを作っていくうえで、いわば「カーネル」として、もっとも根本的な機能を担うことになる．

リスト 4.1.1：BaseLayer のコントラクタ

```
import Activators
import common_function as cf

class BaseLayer:
    def __init__(self, **kwargs):
        print('Initialize', self.__class__.__name__)
        self.width     = kwargs.pop('width',          None)
        activator_name = kwargs.pop('activate', 'Identity')

        self.w = None; self.b = None
        self.activator = cf.eval_in_module(activator_name, Activators)
```

後から必要なメソッドを追加していくこととして、まずは BaseLayer のコントラクタである．もちろんこれだけでは何もしないし、common_function と Activators にしかるべき内容を作りこまなければエラーする．とまれここで記述した内容を見ていこう．
common_function と Activators は別に作って import する．BaseLayer はクラスとして定義し、コントラクタである__init__(self, **kwargs)で引数を渡して初期化する．引数はあとで追加するから一括して**kwargs の辞書型にしている．はじめに自身を初期化することを、
print('Initialize', self.__class__.__name__) で宣言する．その際に、
self.__class__.__name__ で自身のクラス名を得ている．'__'で始まる特殊メソッドを連ねていてややこしく見えるが、こうすればこの場合の自クラスであるselfに限らず、クラス名を得られることを覚えておくと便利だ．
リスト 4.1.1 の範囲では、重み w とバイアス b の初期化の乱数の広がり幅と、活性化関数の種類を、それぞれ'width'と'activate'をキーとして kwargs から取り出している．重み w とバイアス b は初期化のためのメソッドを別に用意することとして、ここではいずれも None としている．最後に活性化関数を設定するが、これは common_function に関数 eval_in_module()を用意して行う．

Activators には恒等関数を定義しておこう．これをリスト 4.1.2 に示す．

リスト 4.1.2：Activators に恒等関数を定義する

```
class Identity:
    def forward(self, x):
        return x
```

Identity というクラスを forward というメソッド 1 つだけで定義する．恒等関数という名の通り引数 x をそのまま返り値とするだけである．なお、forward に対しては backward があるが、それは後から追加する．

さて、問題は common_function に定義する関数 eval_in_module()だ．ここでは、ごく簡単に作っておこう．

リスト 4.1.3：common_function.py に関数 eval_in_module()を定義する

```
def eval_in_module(class_name, module):
    return eval('module.' + class_name + '()')
```

リスト 4.1.3 に示す関数 eval_in_module()は、文字列をコードとして実行することのできる組み込み関数である eval()をそのまま使っているだけだ．表現は平易なのだが意味はちょっと難しい．この関数を eval_in_module('Identity', Activators）で呼び出したとしよう．このとき eval()に渡す引数は 'module.' + class_name + '()' に従い、文字列 'module.Identity()' となる．これが eval によって有効な文 module.Identity() となって解釈され、変数 module には Activators が指定されているから、変数 module の中身は Activators なので、Avtivators.Identity() が返り値となる．つまり module.Identity() の module が首尾よく Activators にすげ変わるのだ．また、実はこの関数 eval()は、便利である反面、文字列を実行してしまうので、問題が指摘されるものであるが、その対処は後で行うこととする．

さてこれで漸く、リスト 4.1.1 が実行できるようになった．ではリスト 4.1.1 の内容は Neuron.py として、これと同じディレクトリに、リスト 4.1.2 の Activators.py とリスト 4.1.2 の common_function.py を置いて実行してみよう．実行の画面上は何も起きない．しかしここで、

```
>>> layer = BaseLayer()
```

と入力してみよう．これで BaseLayer がインスタンス化されているはずだ．()を忘れるとインスタンス化されずにクラスに layer と別名をつけるだけになるから注意が必要だ．ともかくうまく動けば、リスト 4.1.1 のコントラクタ__init__()で記述した print 分に従って、

```
Initialize BaseLayer
```

と出力される．またここで、dir(layer)と打つと、'__'ではじまる特殊メソッドに続いて、ここで定義した'activator', 'b', 'name', 'w', 'width'がリスト形式で返される．さらに、

```
>>> layer.activator
```

と入力すれば、
<Activators.Identity object at 0x0000012B0EA34788>
という風に(末尾の数値はその都度違う)表示され、layer の活性化関数が Activators.py に定義した恒等関数に設定されていることが確認できる.

さてそれでは次に、ここで説明してきた単純なニューロンの場合について、その本体の部分を作っていこう. これは BaseLayer と同じ Neuron.py に置くことにする.

リスト 4.1.4：NeuronLayer のコントラクタ

```
class NeuronLayer(BaseLayer):
    def __init__(self, *configuration, **kwargs):
        if len(configuration) == 2:
            m, n = configuration
        if len(configuration) == 1:
            m = None; n, = configuration
        self.config = m, n
        super().__init__(**kwargs)
```

NeuronLayer は BaseLayer を継承する. リスト 4.1.4 は NeuronLayer のコントラクタだ. 少々面倒だが、その構成を *configuration で与えるようにしている. 構成として指定するのは、入力の数とニューロンの数、あるいは、ニューロンの数だけを指定する. それというのも入力の数は、わざわざ予め指定しなくても、入力を行うときに決まる. 逆に言うなら、コントラクタで入力の数を指定しておいて、それと異なる形状の入力を行うと矛盾してしまうから、指定しないで済ませるほうが便利な場合がある. ともかく、ここでは構成をあらわす入力数とニューロン数を m, n として、これをインスタンス変数 self.config に保存する. 後は super().__init__(**kwargs)で BaseLayer のコントラクタを呼び出すだけだ.

やっと準備ができたので、式 4.1.1～式 4.1.6 で示したニューロンの信号伝達を作ろう. そのために、リスト 4.1.4 に示した NeuronLayer のクラスに、forward―すなわち順伝播のメソッドを定義する.

リスト 4.1.5：NeuronLayer の順伝播

```
class NeuronLayer(BaseLayer):

    #( コントラクタはリスト 4.1.4 に掲載 )

    def forward(self, x):
        if None in self.config:
            print('NeuronLayer.input.shape', x.shape)
            self.fix_configuration(x.shape)
        if self.w is None or self.b is None:
            m, n = self.config   # m:入力数、n:ニューロン数
            self.init_parameter(m, n)
        self.x = x
        u = np.dot(self.x, self.w) + self.b                # affine
        y = self.activator.forward(u)                      # 活性化関数
        return y
```

リスト 4.1.5 に示す forward メソッドで、式 4.1.1～式 4.1.6 に対応するのは、return の前の 2 行である．すなわち、
u = np.dot(self.x, self.w) + self.b で中間値を求め、y = self.activator.forward(u)で 活性化関数によって変換して出力を返り値とする．数式との対応は言うまでもないだろう．
forward メソッドの核となる重みづけ和の計算について先に説明したが、この計算に先立ってやらなくてはならないのは、入力 x に応じて構成 self.config を確定することと、その構成に従って重み w とバイアス b を初期化することだ．構成を確定する役割は別のメソッド、fix_configuration()に委ねることとして、self.config に None があるかどうかを見ている．
また、重み w とバイアス b の初期化も別のメソッド init_parameter()に委ねることとして、self.w と self.b のいずれかが None を見ている．

ではまず fix_configuration()だが、これは class NeuronLayer のメソッドとして定義する．
これをリスト 4.1.6 に示す．

リスト 4.1.6：NeuronLayer の fix_configuration メソッド

```
class NeuronLayer(BaseLayer):

    #( コントラクタはリスト 4.1.4 に、forward メソッドはリスト 4.1.5 に掲載 )

    def fix_configuration(self, shape):
        m = 1
        for i in shape[1:]:  # shape[0]はバッチサイズ
            m *= i
        self.config = m, self.config[1]
```

構成の入力数とニューロン数のうち、ニューロン数はコントラクタの実行時に確定しているから、入力数を入力の形状に合わせれば良い．式 4.1.5 に示すように入力が行列の形状とすれば、その列の数＝1 次元目の大きさが入力数となる．またいっぽう、NeuronLayer で定義する基本的なニューロンでは様々なデータを扱うことが出来る．例えば白黒の画像データならば 1 つの画像で 2 次元のデータとなっているから、それを制約しないように、入力数は 1 次元目以降を掛け合わせた値としている．そしてこれを self.config に設定して完了する．

次に init_parameter メソッドをリスト 4.1.7 に示す．重み w とバイアス b の初期化は、リスト 4.1.5、リスト 4.1.6 に示す基本的なニューロンとは違う種類のものでも共通にできるから、init_parameter メソッドは BaseLayer のメソッドとして定義する．

リスト 4.1.7：BaseLayer の init_parameter メソッド

```
import numpy as np

class BaseLayer:

    #( コントラクタはリスト 4.1.1 に掲載 )

    def init_parameter(self, m, n):
        width = self.width
        if width is not None:
            width = float(width)
        else:
            width = 0.01
        self.w = width * np.random.randn(m, n).astype(Config.dtype)
        self.b = np.zeros(n, dtype=Config.dtype)
        print(self.__class__.__name__, 'init_parameters', m, n)
```

話があとさきになるが、冒頭で import numpy as np により、numpy を使えるようにしておく．init_parameter メソッドが呼ばれる際には入力数 m とニューロン数 n が確定していなければならない．すなわち、式 4.1.3、式 4.1.4、式 4.1.5 に示すベクトルと行列の積ないし行列積を計算するためには、重み w やバイアス b の形が重要であり、重み w やバイアス b の形は、入力数 m とニューロン数 n によって決まる．重み w は m 行 n 列の行列、すなわち行の数が入力の数であり、列の数がニューロンの数となる．いっぽう、バイアス b はニューロン数に対応する要素数 n 個のベクトルとして定義するが、バイアスの加算の際には、numpy の機能として式 4.1.5 で示す拡張が行われる前提だ．初期化にあたり、重み w は乱数、バイアス b は値 0 とする．ここで重み w の乱数の広がりの大きさは、コントラクタの実行時、すなわちク

ラスのインスタンス化の際に設定して、インスタンス変数 self.w で指定するか、その際に設定せずに None となっていたら、デフォルト値 0.01 とする．重み w やバイアス b の初期値は非常に重要だが、ここでは特に指定しなかったら width = 0.01 としている．そして、
self.w = width * np.random.randn(m, n).astype(Config.dtype)
で numpy の乱数を使って初期化する．np.random.randn は平均 0、標準偏差 1 の乱数だから、重み w は、平均 0、標準偏差が width で指定した値の乱数となる．
いっぽうバイアス b は、
self.b = np.zeros(n, dtype=Config.dtype)
によって、値 0 で初期化する．いずれも、大きなニューラルネットワークでは数が多くなり、メモリを圧迫するから astype(Config.dtype)や dtype=Config.dtype で精度を指定している．このためにクラス変数だけのクラスを作っておき、この Config.dtype の値を変えれば、あちこちで指定する精度を一括して変えられるようにしておく．これをリスト 4.1.8 に示す．これはファイル Neuron.py の冒頭に入れておくのが良いだろう．

リスト 4.1.8：精度指定

```
class Config:
    dtype = 'f4'
```

元のリスト 4.1.5 に戻って、構成 self.config、そして、重み w とバイアス b の初期化を行うメソッド fix_configuration()、そしてメソッド init_parameter()は、初めてデータを流す際に 1 回だけ実行すれば、あとは同じ入力数である限り、そのままデータ x を入力して出力 y を取り出すことが出来る．次節で説明するニューラルネットワークで機械学習を行う際には、データ x を入力しては出力 y を取り出す動作を非常に多くの回数繰り返し行うため、この点は重要である．

これで層としてのニューロンの集まりが記述できたので動作を確かめておこう．先にも実行した Neuron.py を開いて Run Module で実行し、エラーせずに、
numpy is running in the background of common_function. [0.36329094].
などと出力されれば準備ができている．ここで、スクリプトの画面からニューロンの層を作ってみよう．例えば図 4.1.2 と同じ形で、入力数 2、ニューロン数 3 ならば、
>>> layer = NeuronLayer(2, 3)
と打ち込めば、
Initialize NeuronLayer

と返ってきて、ニューロンの層が作られたことがわかる．ここで例えば、
>>> x = np.array([[1, 2],[3, 4]])
として入力データ x を用意して、
>>> y = layer.forward(x)
と打ち込めば、すでに layer としてインスタンス化されている NeuronLayer の forward メソッドが実行されて、出力 y が得られるはずだ．このときメッセージは、
NeuronLayer init_parameters 2 3
と出るが、これはリスト 4.1.7 に示す BaseLayer クラスの init_parameter メソッドで print 指定したものだ．いっぽう先の Initialize NeuronLayer のメッセージは、リスト 4.1.1 の BaseLayer クラスのコントラクタ__init__の print 文によるものだ．これらの print 文は無くても動くが、どの段階まで進んだのかを確かめながら進めていくのに大いに役立つから、ぜひ入れておいてほしい．
さてここまでくれば、ニューロンの層としての動作はできているはずだが、その処理内容についても見ておこう．まずは、出力 y の値を確認しよう．
>>> y　と入力すれば、
array([[0.01618334, -0.01371857, 0.00875722],
       [0.0550257 , -0.04350725, 0.01615163]])
などと出力の値が見える．この数値は重み w が乱数となっているから、場合によって異なる．しかし、2 行 3 列の行列となっていることがわかる．これは入力 x を 2 行 2 列の行列にし、3 つのニューロンからなる層を作ったのだから、適切な形だ．でもこれでは処理が正しいかどうかまではわからない．そこで乱数で初期化されている重み w の値を例えばすべて 1 に変えてみよう．まず元の値を確認しておく．同じくスクリプトの画面から、
>>> layer.w　と打ち込めば、
array([[0.02265902, -0.0160701 , -0.00136281],
       [-0.00323784, 0.00117576, 0.00506002]], dtype=float32)
などと帰ってくるはずだ、むろん値は乱数だから、その都度違ってよい．ここで、
>>> layer.w[...]=1
とすれば、この値をすべて 1 に変えられる．この指定で、[...]は詳しく説明しないが、行列の要素をメモリ上の位置を変えずに指定する．結果的に元の形のまま値だけが置き換わることになる．ここでもう一度、
>>> layer.w　と打ち込めば、
array([[1., 1., 1.],
       [1., 1., 1.]], dtype=float32)
と重みの値がすべて 1 となっていることが確認できる．

いちおうバイアスも layer.b で見ておくと、
array([0., 0., 0.], dtype=float32)
とすべて 0 になっているはずだ.
この重みとバイアスに対して、もう一度
>>> y = layer.forward(x)
で forward メソッドを実行して出力 y を確認すると、
array([[3., 3., 3.],
[7., 7., 7.]])
となるはずだ. 数式通りであることは、式 4.1.2～式 4.1.6 の計算をすれば、活性化関数が恒等関数 Identity だから $\boldsymbol{Y} = \boldsymbol{U}$ であり、

$$\begin{pmatrix} y_{00} & y_{01} & y_{02} \\ y_{10} & y_{11} & y_{12} \end{pmatrix} = \begin{pmatrix} u_{00} & u_{01} & u_{02} \\ u_{10} & u_{11} & u_{12} \end{pmatrix} = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix} + (0 \quad 0 \quad 0) = \begin{pmatrix} 3 & 3 & 3 \\ 7 & 7 & 7 \end{pmatrix} \tag{4.1.7}$$

出力 y は、式 4.1.7 の結果と一致していることを確認することができる. このように動作の確認は煩わしいが、確認しながら一歩ずつ積み上げていくことは、達成への早道である.

ここで活性化関数 Activators にステップ関数を追加しておこう.

リスト 4.1.9：Activators にステップ関数を定義する

```
class Step:
    def __init__(self, t=0):
        self.t = t

    def forward(self, x):
        y = x > self.t
        return y
```

コントラクタではステップ関数の閾値をデフォルト引数 t=0 で与える. そして forward メソッドで、入力とその閾値の大小識別を行って、その結果を返り値とする. 初期化の際に何も指定しなければ閾値は 0 で、入力が 0 より大きいならば True、そうでなければ False になる. 比較を行う次の文、
y = x > self.t
は入力 x と t を比較するが、x が numpy の行列ならば、比較結果 y も numpy の行列となる.

4.1.3 簡単なパーセプトロン

パーセプトロンに必要な材料がそろったので、さっそくニューロン１個からなる簡単なパーセプトロンを動かしてみよう.

リスト 4.1.10：簡単なパーセプトロン

```
import numpy as np
import Neuron

model = Neuron.NeuronLayer(1, activate='Step')

x = np.array([[0, 0],[0, 1],[1, 0],[1, 1]])
print('input¥n', x)

y = model.forward(x)
print('random¥n', y)

model.w[...] = 1
y = model.forward(x)
print('OR¥n', y)

model.b[...] = -1
y = model.forward(x)
print('AND¥n', y)
```

リスト 4.1.10 に示すのはごくごく簡単なパーセプトロンだ.

model = Neuron.NeuronLayer(1, activate='Step')

でニューロン層を使い、ニューロン１つ、活性化関数はステップ関数、と指定して model としてインスタンス化する. データは 0 と 1 の 2 つの値の組み合わせとする. そしてこのデータをはじめはそのまま、その後に重みとバイアスの値を設定して model.forward で順伝播を行って結果を表示する. そうすると、そのままでは実行のたびに結果が違うが、その後に重みを 1 にした場合には論理和、それに加えてバイアスを-1 にした場合には論理積が得られる. これ以外にも、重みとバイアスを変えれば、また違った論理をとることができるから、やってみると良い.

4.2 簡単なデータの準備

前節で、ニューロン１個で論理演算を行うパーセプトロンを作った．重みとバイアスを適切に設定すれば、論理和を出力したり論理積を出力したりと、動作が変わることも見てきた。ここではパーセプトロンの可能性を探るためのデータを用意していくことにする．

4.2.1 アヤメのデータ

これまでの範囲では Python が動く環境であり、数値計算ライブラリ numpy があればよかったが、機械学習用のモジュール scikit-learn をインストールして欲しい．scikit-learn がインストールされていれば、そこに含まれる機械学習用のデータを使うことができる．以下はこれが使えることが前提となる．

Python のスクリプトの画面で、

```
>>> from sklearn import datasets
```

と打ち込んでから、

```
>>> iris = datasets.load_iris()
```

とすると、iris にアヤメのデータや正解ラベルなどが得られる．

```
>>> type(iris)
```

と打ち込むと、

```
<class 'sklearn.utils.Bunch'>
```

と返ってくるが、iris は「辞書もどき」のようになっている．すなわち、

```
>>> iris.keys()
```

とすると、

```
dict_keys(['data', 'target', 'target_names', 'DESCR', 'feature_names', 'filename'])
```

と返ってきて、iris はキーを使ってアクセスできる 6 つの要素から成っていて、

```
>>> iris['data']
```

では、

```
array([[5.1, 3.5, 1.4, 0.2],
       [4.9, 3. , 1.4, 0.2],
       [4.7, 3.2, 1.3, 0.2],
          (途中略)
       [5.9, 3. , 5.1, 1.8]])
```

これは、iris.data としてもアクセスできる．そこで、楽な方の'.xx'を使って、

```
>>> type(iris.data)
```

で、

```
<class 'numpy.ndarray'>
```

さらに、

>>> iris.data.shape　　で、
(150, 4)
と返ってくるから、150×4 の numpy の行列になっていることがわかる.
>>> iris.target　　では、
```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2])
```
と返ってきて、アヤメの品種の ID を 0～2 で示していて、その 3 つの品種名は、
>>> iris.target_names　　によって、
```
array(['setosa', 'versicolor', 'virginica'], dtype='<U10')
```
と知ることができる. いっぽう、
>>> iris.feature_names　　で、
```
['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal width (cm)']
```
と返ってきて、先に示した iris.data の 4 つの数値の並びが、
「がく片の長さ」、「がく片の幅」、「花弁の長さ」、「花弁の幅」だとわかる.
ではこれを使うのに便利なようにプログラムにしておこう.

リスト 4.2.1：アヤメのデータ

```
import numpy as np
from sklearn import datasets

def get_data():
    data    = datasets.load_iris().data
    target = datasets.load_iris().target
    correct = np.eye(3)[target]
    return data, correct, target

def label_list():
    labels = datasets.load_iris().target_names
    return labels
```

リスト 4.2.1 を簡単に説明する. 冒頭 sklearn の dataset をインポートして関数 get_data()の中で読み込む. 入力に使う data は読込んだ値そのものだが、正解値は one_hot の correct と正解番号 target の 2 通りを用意する. target が示す正解番号は 0～2 の値をとるが、one_hot は

正解番号が 0 ならば 100、正解番号が 1 ならば 010、正解番号が 2 ならば 001 と、番号の範囲の正解のところだけ'1'で他は'0'のベクトルに変換したものだ. この one_hot の正解値 correct は、numpy の関数 eye()を使って target から変換して得る. get_data()の返り値は data と correct と target だ. もう 1 つ関数 label_list()を用意し、品種の番号に対応する名前を得られるようにしておく. この内容は Neuron.py を置いたのと同じディレクトリに Iris.py として入れておこう.

[補足：one_hot について]
たとえばアヤメのデータで正解が 0～2 のうち、'1'とすれば、この'1'には 0 より大きくて 2 より小さいといったような数の大小関係は関係あるだろうか？ そうではなくて、これはあくまでもラベルないしは識別子であって、'0'とも'2'とも違う'1'だということを示すだけのはずだ. いっぽうで数値計算で結果を出そうとする際には、算数、数学そのものに従わなくてはならないから、正解値が 0、1、2 のように数値のままでは具合が悪い. そこで、取りうる値の大小関係を排除して、横並びの数値のどれか 1 つが '1'として、どのカテゴリーに属するのかを値の並びで直に示す one_hot が都合が良い. one_hot であれば、それぞれのニューロンはただ自身が割り当てられたカテゴリーに属する可能性の高低に応じた値を出力するだけで良くなる.

4.2.2 アヤメのデータを使ってみる

データを可視化したりグラフを描画したりするために、グラフ描画ライブラリ Mayplotlib が必要だ. 以下はこれが使えるのが前提となる.
さっそくアヤメのデータの中から「花びらの長さ」と「花びらの幅」の関係をグラフにしてみよう.

リスト 4.2.2：アヤメのデータから花びらの長さと花びらの幅の関係を可視化

```
import matplotlib.pyplot as plt
import Iris

# データを用意
data, correct, target = Iris.get_data()

x = data[:, 2:3] # 花弁の長さ
t = data[:, 3:4] # 花弁の幅

# グラフ表示
plt.scatter(x, t, marker='x')
plt.show()
```

リスト 4.2.2 は、冒頭で matplotlib.pyplot を plt と呼び変え、先にリスト 4.2.1 で作った Iris.py とともにインポートする．Iris.py の関数 get_data()でデータを読込み、その data の中の花弁の長さと花弁の幅の項を x = data[:, 2:3]と t = data[:, 3:4]で抽出する．なお x = [:, 2]、t = [:, 3]としてもデータは抽出できる．しかしそうすると x、t の次元が 1 つ減ってベクトルになって、あとでパーセプトロンで使う際に都合が悪いから、次元を維持するようにしている．抽出した花弁の長さと花弁の幅は、matplotlib の散布図 scatter()を使って表示する．matplotlib は実際に描画する際に show()を行う．この結果を図 4.2.1 に示す．

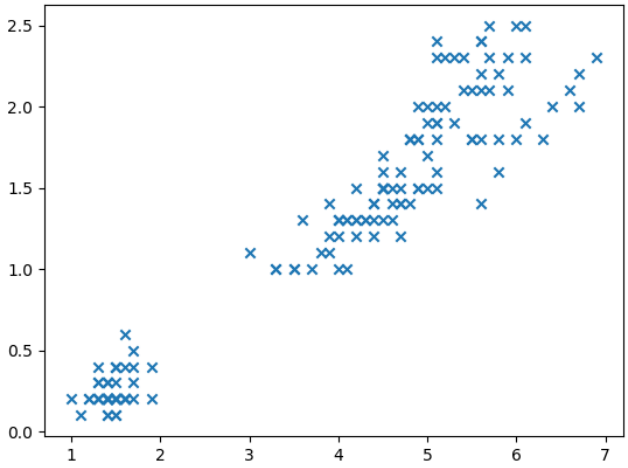

図 4.2.1 アヤメの花弁の長さと幅

図 4.2.1 を見るとアヤメの花弁の長さと幅がほぼ直線的に並んでいることがわかる．そこで図 4.2.2 に示すように、花弁の長さを入力したら花弁の幅を出力するようにパーセプトロンを働かせてみよう．

図 4.2.2 パーセプトロンでアヤメの花弁の長さから花弁の幅を回帰

リスト 4.2.3：アヤメのデータを近似する

```
import matplotlib.pyplot as plt
import Neuron, Iris

# データを用意
data, correct, target = Iris.get_data()

# モデルを作って近似を行う
model = Neuron.NeuronLayer(1, 1)

model.w = [[0.4]]
model.b = [-0.3]

x = data[:, 2:3] # 花弁の長さ
t = data[:, 3:4] # 花弁の幅
y = model.forward(x)

# グラフ表示
plt.scatter(x, t, marker='x')
plt.plot(x, y)
plt.show()
```

リスト 4.2.3 はリスト 4.2.2 に加えてパーセプトロンの入出力の関係をグラフに表示する．このため numpy と Neuron もインポートする．データを用意し、花弁の長さと花弁の幅を表示するのはリスト 4.2.2 と同じだ．いっぽうパーセプトロンは入力 1 出力 1 で大きさをはじめから指定し、活性化関数はデフォルトの恒等関数のまま、model = Neuron.NeuronLayer(1, 1) でインスタンス化する．重みとバイアスはちょっと置いておいて、花弁の幅 x をパーセプトロンの入力にするが、この入力は 1 つでも 2 次元配列ないし縦ベクトルにしておく必要がある．そして重みとバイアスだが、まずは指定をコメントアウトしたまま実行し、あとからコメントを外して指定して実行してみよう．ともかく、y = model.forward(x) でパーセプトロンの出力 y を得て、入力の x との関係を plt.plot(x, y) でグラフにする．最後に plt.show() で描画する．

この実行結果を図 4.2.3 に示す．図に示すのは上記の重みとバイアスの指定を行った場合だ．このように重みもバイアスも 1 つずつで簡単な場合には、目分量で重みとバイアスの値を決めるのは、簡単だし結果もまあまあ妥当なのではないだろうか？ ともかくも適切な重みとバイアスが得られる限りにおいて、パーセプトロンで回帰問題を扱うことができそうだ．

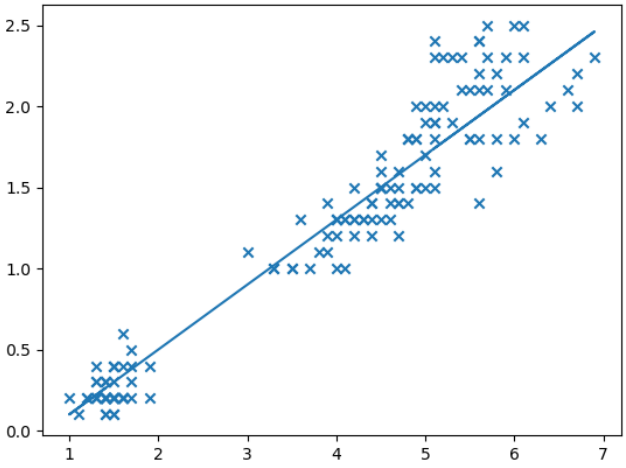

図 4.2.3 アヤメの花弁の長さと幅の近似

4.2.3 アヤメの分類

先に述べたように、アヤメのデータは「がく片の長さ」、「がく片の幅」、「花弁の長さ」、「花弁の幅」の値とともに、その品種が、'setosa', 'versicolor', 'virginica'のいずれであるかが示されている. そこで図 2.2.3 に示すようにパーセプトロンで、「がく片の長さ」、「がく片の幅」、「花弁の長さ」、「花弁の幅」の値から品種を分類する、ということができないだろうか？

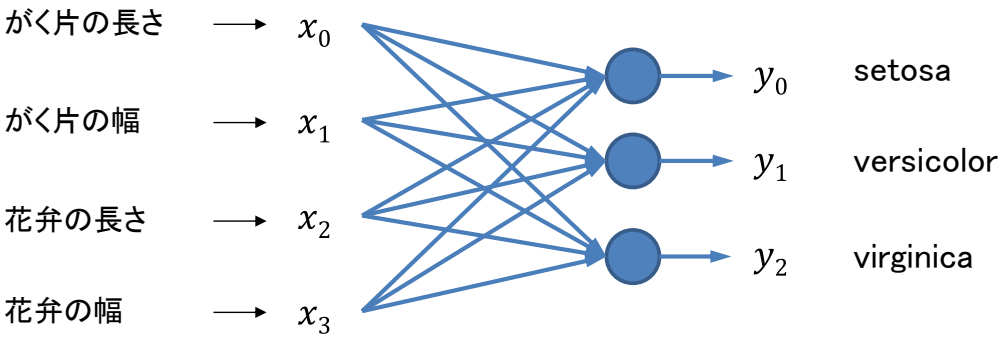

図 4.2.4 パーセプトロンでアヤメの品種を分類

ではさっそくやってみよう.

リスト 4.2.4：アヤメの品種を分類する

```
import numpy as np
import Neuron
import Iris

# データを用意
x, c, t = Iris.get_data()

# モデルを作る
model = Neuron.NeuronLayer(3)

'''# 重みとバイアスの設定
model.w = np.array( [[ 0.1,  0.0,  0.0],
                     [ 0.2, -0.4,  0.2],
                     [-0.2,  0.2,  0.0],
                     [-0.0, -0.5,  0.6]])
model.b = np.array([ 0.1,  1.6, -0.7])
#'''#
# 順伝播
y = model.forward(x)

# 結果確認
for i in range(len(x)):
    print(i, np.argmax(y[i])==t[i])
```

リスト 4.2.4 は、アヤメのデータを読み込んで品種を分類し、その正誤を出力するプログラムだ. 冒頭 Neuron.py と Iris.py が必要だが numpy も使うから、これらをインポートする. モデルはニューロン数＝出力数＝3 を指定し、活性化関数はデフォルトの恒等関数だ. 続く重みとバイアスの設定は '''#と#'''#で囲まれた範囲をまとめて無効化する. そしてこうしておけば、はじめの'''#を頭に#をつけて#'''#に変えれば、囲まれた範囲が有効化できる. はじめは囲まれた範囲を無効化したままやってみよう. そうすると、順伝播の際に重みとバイアスを乱数で初期化するから、続く結果確認では、鉛筆倒しと同じように 3 択だから 3 分の 1 だけ正解するだろう. それから今度は重みとバイアスの設定を活かしてやってみよう. 全部が正解にはならないものの、正解率は大幅に向上するはずだ. パーセプトロンは、適切な重みとバイアスが得られる限りにおいて、回帰問題だけでなく分類問題を扱うこともできそうだ.

4.3 損失と勾配

前節でアヤメのデータを用意し、これを使って、パーセプトロンを回帰問題や分類問題に適用できるのではないかというところまで見てきた．いずれの場合も、重みやバイアスに設定する値が鍵となるのだが、そのためにはそもそもパーセプトロンが出力する結果の良し悪しを評価する尺度が必要だ．そして次にその尺度に従って、重みやバイアスの値をどうすれば良くなるのかを知ることができれば、重みやバイアスの値を適切なものにすることができるはずだ．そこで本節では、評価尺度としての損失関数、そしてその損失関数によって算出される損失とその勾配について考えていくことにする．

4.3.1 損失と損失関数

図 4.2.3 はアヤメの花弁の長さと花弁の幅の関係を示すが、これをあらわす直線の両側に実際のデータは程よく散らばっている．もともとばらつきがあって誤差が 0 にはならないような場合に、これを評価する尺度として二乗和誤差がある．これについては第 2 章の 2.5 節 正規分布、ガウス分布で説明した．

いっぽうここでは、モデルの重みとバイアスを調整する目的に対して、出力と正解値との隔たりの大きさの定量的な尺度を損失(誤差ともいう)、そしてこれを与える関数を損失関数(誤差関数ともいう)とし、その代表的な 1 つとして、二乗和誤差をサンプル数で割った Mean Squared Error(略して MSE)、平均二乗誤差を定義する．これは出力 y、正解値 t とすれば、k 個のサンプルに対し次式であらわされる．

$$L = \frac{1}{2k}\sum^{k}(y-t)^2 \tag{4.3.1}$$

式 4.3.1 で $y-t$ は出力と正解の差だが、二乗しているから、複数サンプルに対して y と t の大小関係によらず隔たりの大きさが合算される．またサンプル数 k で割って平均としているが、さらにその 2 分の 1 とするのは、後述するが損失関数を微分した場合に都合が良いからだ．

さてそれでは、図 4.2.4 のアヤメの品種を分類するパーセプトロンではどうだろう？　この場合、どの品種かに応じて、3 つのニューロンから出力する．そしてその正解値は one_hot で'100'、'010'、'001'とした．そして例えば'setosa'が正解ならば、正解値 100 に対して、1 つ目のニューロンは 1 にどれだけ近い値を出力するか、2 つ目と 3 つ目のニューロンは 0 にどれだけ近い値を出力するかを、同時に評価すべきだろう．そして他のデータについても、同様に正解値の 3 つの値に対して 3 つの出力の隔たりを同時に評価する必要がある．そこで

出力と正解値のベクトルどうしの引き算の 3 つの要素の二乗和をとって、それを複数のデータにわたって平均をとって損失とすれば良さそうだ．このような分類の場合にも式 4.3.1 がそのまま使える．

リスト 4.3.1：Mean Squared Error

```
import numpy as np

class MeanSquaredError:
    def forward(self, y, t):
        self.k = y.size // y.shape[-1]
        l = 0.5 * np.sum((y - t) ** 2)
        return l / self.k
```

リスト 4.3.1 は式 4.3.1 をそのまま記述したものとなっている．サンプル数 k を求めるのに

self.k = y.size // y.shape[-1]

としている．ここでいうサンプル数はニューラルネットワークに 1 度に与えるデータの組の数、すなわちバッチ処理のバッチの大きさだ．k の求め方が少々ややこしいが、出力が one_hot のベクトルとなっている場合も、こうしておけば扱え、時系列データのようにさらに次元数が多い場合にも対応する．リスト 4.3.1 の内容は、LossFunctions.py とファイル名をつけて、Neuron.py と同じディレクトリに格納しておこう．

図 4.3.1 には入力 $\boldsymbol{X}$、出力 $\boldsymbol{Y}$、正解値 $\boldsymbol{T}$、損失 L としたときのパーセプトロンと損失関数の関係を簡単に図示する．

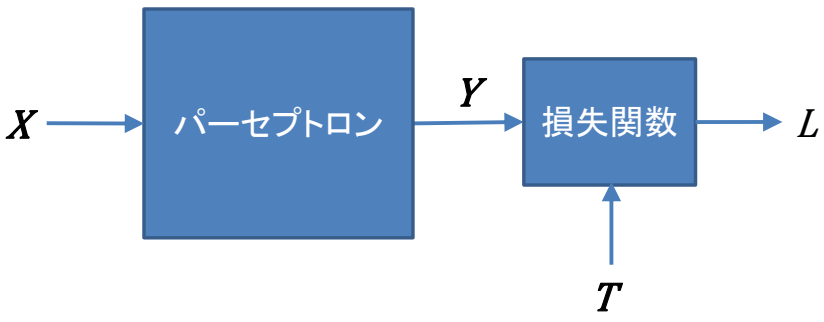

図 4.3.1 パーセプトロンと損失関数

4.3.2 損失を評価する

では Neuron.py、Iris.py、LossFunctions.py を使って損失関数を組み込んでモデルを作り評価してみよう.

リスト 4.3.2：アヤメのデータを近似する～その 2

```
import numpy as np
import matplotlib.pyplot as plt
import Neuron, LossFunctions, Iris

# データを用意
data, correct, target = Iris.get_data()

# モデルを作って近似を行う
model = Neuron.NeuronLayer(1, 1)
loss_function = LossFunctions.MeanSquaredError()

model.w = [[0.4]]
model.b = [-0.3]

x = data[:, 2:3]
t = data[:, 3:4]
y = model.forward(x)
loss = loss_function.forward(y, t)
print(loss)

# グラフ表示
plt.scatter(x, t, marker='x')
plt.plot(x, y)
plt.show()
```

リスト 4.3.2 は損失関数以外はリスト 4.2.3 と同じだ. 冒頭インポート対象に LossFunctions を加え、モデルを作成する際に loss_function = LossFunctions.MeanSquaredError()で損失関数もインスタンス化する. そして出力 y と正解値 t を入力し、
loss = loss_function.forward(y, t)で損失を得て print(loss)で表示する.
さて重みとバイアスを初期値のままとした場合と、設定した場合で損失の値が変わることが確認できるはずだ.

もう 1 つアヤメの品種分類にも損失を組み込もう.
リスト 4.3.3 は損失関数以外はリスト 4.2.4 と同じだ. また損失関数の組み込み方はリスト 4.3.2 と同じだから、もう説明の必要はないだろう. この場合も重みとバイアスを初期値のままとした場合と、設定した場合で損失の値が変わることを観察して欲しい.

リスト 4.3.3：アヤメの品種を分類する～その 2

```
import numpy as np
import Neuron, LossFunctions, Iris

# データを用意
x, c, t = Iris.get_data()

# モデルを作る
model = Neuron.NeuronLayer(3)
loss_function = LossFunctions.MeanSquaredError()

#'''# 重みとバイアスの設定
model.w = np.array( [[ 0.1,  0.0,  0.0],
                     [ 0.2, -0.4,  0.2],
                     [-0.2,  0.2,  0.0],
                     [-0.0, -0.5,  0.6]])
model.b = np.array([ 0.1,  1.6, -0.7])
#'''#
# 順伝播
y = model.forward(x)
loss = loss_function.forward(y, c)
print(loss)

# 結果確認
for i in range(len(x)):
    print(i, np.argmax(y[i])==t[i])
```

リスト 4.3.2、リスト 4.3.3 いずれの場合も、重みやバイアスが乱数や'0'で初期化されただけのパーセプトロンは、入力と出力の関係に意味があるとはいいがたいだろう．そしてその際の損失の大きさはどうだろうか？　いっぽう重みやバイアスに値を設定すると、回帰や分類で、それなりに働くようになるが、その際の損失の大きさはどうだろう？ 当然ではあるけれども、損失の大小とパーセプトロンの働きとの関連がわかるのではないだろうか？

4.3.3 勾配とは

モデルの働きの良し悪しを客観的に評価する指標として損失関数を組み込んだ. そして、ニューロンの重みやバイアスの値の大小と、損失 L の大小との関係を知ることができれば、それにもとづいて重みやバイアスの値を調節し、モデルの働きを良くすることが期待できる. このニューロンの重みやバイアスの値の大小と、損失 L の大小との関係は、損失 L をそれぞれの重みやバイアスで偏微分することによって求めることが出来る. そしてこの偏微分の値を勾配という.

[補足]

一般に多変数のスカラー値関数 $f(\boldsymbol{x}) = f(x_0, \dots, x_n)$ の関数行列またはヤコビ行列は、

$$\frac{\partial f}{\partial \boldsymbol{x}} = \left(\frac{\partial f}{\partial x_0}, \dots, \frac{\partial f}{\partial x_n}\right) \tag{4.3.2}$$

となり、これを f の勾配という. 損失関数 L も多変数のスカラー値関数だから、同様に勾配が求められる. しかし損失関数の何を変数とみて微分するかによって、異なる勾配が得られる. そこでここでは数学的に正しい言い方になるかどうかはさておき、これらを区別するために、重みの勾配、バイアスの勾配、入力の勾配など、微分される関数ではなくて、それを微分する変数を区別して、何某の勾配と呼んでいる.

4.3.4 勾配を求める

では具体的に図 4.2.2 の入力 1 つ出力 1 つのパーセプトロンで考えてみよう．損失関数は式 4.2.1 で示す平均二乗誤差で、データは全体では 150 組だが、その中の 1 組に着目する．すると入力 x、中間値 u、出力 y いずれもスカラーとなって順伝播は次式であらわされる．

$$y = u = x \times w + b \tag{4.3.3}$$

そしてこのとき正解値を t とすれば損失は式 4.3.1 から、

$$L = \frac{1}{2}(y - t)^2 = \frac{1}{2}(x \times w + b - t)^2 \tag{4.3.4}$$

式 4.3.4 で損失 L を、重み w、バイアス b、そして入力 x でそれぞれ微分する．まず w について微分すれば、

$$\frac{\partial L}{\partial w} = \frac{\partial}{\partial w}\left\{\frac{1}{2}(x \times w + b - t)^2\right\} = (x \times w + b - t)x = (y - t)x \tag{4.3.5}$$

ここで $\delta = y - t$ とおき、ほかの変数についても同様にして、

$$\frac{\partial L}{\partial w} = \delta x \qquad \frac{\partial L}{\partial b} = \delta \qquad \frac{\partial L}{\partial x} = \delta w \tag{4.3.6}$$

このようにして勾配を求めることが出来る．では続いて複数組のデータで考えてみよう．アヤメのデータは全体では 150 組だが、すべて一度に使うとは限らないし、ほかの場合にはデータの数も違うだろうから、 k 組のデータとする．入力 $\boldsymbol{X}$ の形状は、式 4.1.5 で入力が 1 つの場合となるから、列数が 1 の行列、すなわち、縦ベクトルになる．同様に中間値 $\boldsymbol{U}$ と出力 $\boldsymbol{Y}$ も縦ベクトルになる．いっぽう入力 1 つニューロン 1 つだから、重み w、バイアス b はいずれもスカラーだ． すると順伝播は次式であらわされる．

$$\boldsymbol{Y} = \boldsymbol{U} = \boldsymbol{X} * w + b = \begin{pmatrix} x_0 \\ x_1 \\ \vdots \end{pmatrix} w + b = \begin{pmatrix} x_0 w + b \\ x_1 w + b \\ \vdots \end{pmatrix} \tag{4.3.7}$$

損失関数は同じく平均二乗誤差で、

$$L = \frac{1}{2k}\sum(y-t)^2 = \frac{1}{2k}\{(y_0-t_0)^2 + (y_1-t_1)^2 + \cdots\}$$
$$= \frac{1}{2k}\{(x_0w+b-t_0)^2 + (x_1w+b-t_1)^2 + \cdots\} \qquad (4.3.8)$$

Lがスカラーであるのに対して、出力 $\boldsymbol{Y}$ や中間値 $\boldsymbol{U}$ は k 個の要素をもつ縦ベクトルとなっている．正解値も k 個の要素をもつ縦ベクトルだ．
ここで中間値の勾配を求めておこう．出力 $\boldsymbol{Y}$ や中間値 $\boldsymbol{U}$ の i 番目の要素について微分をすると、

$$\frac{\partial L}{\partial u_i} = \frac{\partial L}{\partial y_i}\frac{\partial y_i}{\partial u_i} = \frac{\partial L}{\partial y_i}\frac{\partial f(u_i)}{\partial u_i}$$
$$= \frac{\partial}{\partial y_i}\left[\frac{1}{2k}\sum^{k}(y-t)^2\right]$$
$$= \frac{\partial}{\partial y_i}\left[\frac{1}{2k}\{(y_0-t_0)^2 + \cdots + (y_i-t_i)^2 + \cdots\}\right] = \frac{1}{k}(y_i-t_i) \qquad (4.3.9)$$

この勾配を求める際、損失関数のΣの中の二乗の各項のうち y_i が関係するものだけが残り、偏微分の際に他は常数とみなす．とまれ中間値の勾配を中間値 $\boldsymbol{U}$ と同じ形状に並べて、

$$\frac{\partial L}{\partial \boldsymbol{U}} = \begin{pmatrix} \frac{\partial L}{\partial u_0} \\ \frac{\partial L}{\partial u_1} \\ \vdots \end{pmatrix} = \frac{1}{k}\begin{pmatrix} y_0-t_0 \\ y_1-t_1 \\ \vdots \end{pmatrix} = \begin{pmatrix} \delta_0 \\ \delta_1 \\ \vdots \end{pmatrix} = \boldsymbol{\delta} \qquad (4.3.10)$$

式4.3.6で δ としたのはデータ1組の場合の中間値の勾配にほかならない．そしてその δ は式4.3.9に示すように損失関数の出力による微分 $\frac{\partial L}{\partial y}$ と、出力の中間値による微分 $\frac{\partial y}{\partial u}$ との要素ごとの積だが、ここでは活性化関数が恒等関数で $\frac{\partial y}{\partial u} = 1$ だから $\delta = \frac{\partial L}{\partial y}$ となっている．

いっぽう、式4.3.8を重み w について微分して重みの勾配 $\frac{\partial L}{\partial w}$ を求めると、

$$\begin{aligned}\frac{\partial L}{\partial w} &= \frac{\partial}{\partial w}\left[\frac{1}{2k}\{(x_0 w + b - t_0)^2 + (x_1 w + b - t_1)^2 + \cdots\}\right] \\ &= \frac{1}{k}(x_0 w + b - t_0)x_0 + \frac{1}{k}(x_1 w + b - t_1)x_1 + \cdots \\ &= \frac{1}{k}(y_0 - t_0)x_0 + \frac{1}{k}(y_1 - t_1)x_1 + \cdots \\ &= \delta_0 x_0 + \delta_1 x_1 + \cdots = (x_{00} \quad x_{10} \quad \cdots)\begin{pmatrix}\delta_0 \\ \delta_1 \\ \vdots\end{pmatrix} = {}^t\boldsymbol{X}\,\boldsymbol{\delta}\end{aligned} \tag{4.3.11}$$

続いて式4.3.8をバイアス b について微分すれば、

$$\begin{aligned}\frac{\partial L}{\partial b} &= \frac{1}{k}\{(x_0 w + b - t_0) + (x_1 w + b - t_1) + \cdots\} \\ &= \frac{1}{k}\{(y_0 - t_0) + (y_1 - t_1) + \cdots\} \\ &= \delta_0 + \delta_1 + \cdots = \sum \delta\end{aligned} \tag{4.3.12}$$

最後に入力 x の勾配を求める．まず式4.3.8の損失 L を x_0 について微分すれば、

$$\frac{\partial L}{\partial x_0} = \frac{1}{k}(x_0 w + b - t_0)w = \frac{1}{k}(y_0 - t_0)w = \delta_0 w \tag{4.3.13}$$

入力 x の他のデータに対しても同様にして、

$$\frac{\partial L}{\partial x_1} = \delta_1 w \quad \cdots \tag{4.3.14}$$

だから、

$$\frac{\partial L}{\partial \boldsymbol{X}} = \begin{pmatrix} \frac{\partial L}{\partial x_0} \\ \frac{\partial L}{\partial x_1} \\ \vdots \end{pmatrix} = \begin{pmatrix} \delta_0 w \\ \delta_1 w \\ \vdots \end{pmatrix} = \begin{pmatrix} \delta_0 \\ \delta_1 \\ \vdots \end{pmatrix} w = \boldsymbol{\delta}\, w \tag{4.3.15}$$

ではさらに進めて、図 4.2.4 のパーセプトロンで考えていこう．この場合には入力が 4 つとなっている．簡単のため 3 つあるニューロンの 1 つに着目することとする．そうすると活性化関数は恒等関数で順伝播は次式であらわされる．

$$\begin{aligned} \boldsymbol{Y} = \boldsymbol{U} = \boldsymbol{X} * \boldsymbol{W} + b &= \begin{pmatrix} x_{00} & x_{01} & x_{02} & x_{03} \\ x_{10} & x_{11} & x_{12} & x_{13} \\ \vdots & \vdots & \vdots & \vdots \end{pmatrix} \begin{pmatrix} w_0 \\ w_1 \\ w_2 \\ w_3 \end{pmatrix} + b \\ &= \begin{pmatrix} x_{00}w_0 + x_{01}w_1 + x_{02}w_2 + x_{03}w_3 + b \\ x_{10}w_0 + x_{11}w_1 + x_{12}w_2 + x_{13}w_3 + b \\ \vdots \end{pmatrix} \end{aligned} \tag{4.3.16}$$

損失関数は平均二乗誤差だから、データの組の数を k として、

$$\begin{aligned} L = \frac{1}{2k}\sum^{k}(y-t)^2 &= \frac{1}{2k}\{(y_0 - t_0)^2 + (y_1 - t_1)^2 + \cdots\} \\ &= \frac{1}{2k}\{(x_{00}w_0 + x_{01}w_1 + x_{02}w_2 + x_{03}w_3 + b - t_0)^2 \\ &\quad +(x_{10}w_0 + x_{11}w_1 + x_{12}w_2 + x_{13}w_3 + b - t_1)^2 + \cdots\} \end{aligned} \tag{4.3.17}$$

ここではニューロン 1 つに着目しているため、この値は複数の出力にわたる全体の損失のうち、着目した 1 つを取り出したもの、つまり一部分だということを覚えておこう．
それはさておき、ここから中間値の勾配との関係に着目しつつ、それぞれの勾配を求めていこう．中間値の勾配 $\boldsymbol{\delta}$ は、入力が複数になってもニューロン 1 つに着目するならば、中間値や出力は 1 つずつであって式 4.3.10 がそのまま成り立つ．それをふまえて、式 4.3.17 を重み $\boldsymbol{W}$ について微分して重みの勾配 $\frac{\partial L}{\partial \boldsymbol{W}}$ を求める．まず、w_0 について微分すれば、

$$\frac{\partial L}{\partial w_0} = \frac{\partial}{\partial w_0}\left[\frac{1}{2k}\{(x_{00}w_0 + x_{01}w_1 + x_{02}w_2 + x_{03}w_3 + b - t_0)^2 + (x_{10}w_0 + x_{11}w_1 + x_{12}w_2 + x_{13}w_3 + b - t_1)^2 + \cdots\}\right]$$

$$= \frac{1}{k}(x_{00}w_0 + x_{01}w_1 + x_{02}w_2 + x_{03}w_3 + b - t_0)x_{00} + \frac{1}{k}(x_{10}w_0 + x_{11}w_1 + x_{12}w_2 + x_{13}w_3 + b - t_1)x_{10} + \cdots$$

$$= \frac{1}{k}(y_0 - t_0)x_{00} + \frac{1}{k}(y_1 - t_1)x_{10} + \cdots$$

$$= \delta_0 x_{00} + \delta_1 x_{10} + \cdots \tag{4.3.18}$$

同様にして、

$$\frac{\partial L}{\partial w_1} = \delta_0 x_{01} + \delta_1 x_{11} + \cdots \quad \frac{\partial L}{\partial w_2} = \delta_0 x_{02} + \delta_1 x_{12} + \cdots \quad \frac{\partial L}{\partial w_3} = \delta_0 x_{03} + \delta_1 x_{13} + \cdots \tag{4.3.19}$$

これを $\boldsymbol{W}$ と同じ形状に並べて、

$$\frac{\partial L}{\partial \boldsymbol{W}} = \begin{pmatrix} \frac{\partial L}{\partial w_0} \\ \frac{\partial L}{\partial w_1} \\ \frac{\partial L}{\partial w_2} \\ \frac{\partial L}{\partial w_3} \end{pmatrix} = \begin{pmatrix} \delta_0 x_{00} + \delta_1 x_{10} + \cdots \\ \delta_0 x_{01} + \delta_1 x_{11} + \cdots \\ \delta_0 x_{02} + \delta_1 x_{12} + \cdots \\ \delta_0 x_{03} + \delta_1 x_{13} + \cdots \end{pmatrix} = \begin{pmatrix} x_{00} & x_{10} & \cdots \\ x_{01} & x_{11} & \cdots \\ x_{02} & x_{12} & \cdots \\ x_{03} & x_{13} & \cdots \end{pmatrix} \begin{pmatrix} \delta_0 \\ \delta_1 \\ \vdots \end{pmatrix} = {}^t\boldsymbol{X}\,\boldsymbol{\delta} \tag{4.3.20}$$

続いてバイアスの勾配は、式 4.3.17 をバイアス b について微分し、式 4.3.10 により $\frac{1}{k}(y - t)$ を δ で置換えて次式となる.

$$\frac{\partial L}{\partial b} = \frac{1}{k}\{(x_{00}w_0 + x_{01}w_1 + x_{02}w_2 + x_{03}w_3 + b - t_0) + (x_{10}w_0 + x_{11}w_1 + x_{12}w_2 + x_{13}w_3 + b - t_1) + \cdots\}$$

$$= \frac{1}{k}\{(y_0 - t_0) + (y_1 - t_1) + \cdots\} = \delta_0 + \delta_1 + \cdots = \sum \delta \tag{4.3.21}$$

最後に入力 $\boldsymbol{X}$の勾配を求める．まず式 4.3.17 の損失 L を x_{00} について微分すれば、

$$\frac{\partial L}{\partial x_{00}} = (x_{00}w_0 + x_{01}w_1 + x_{02}w_2 + x_{03}w_3 + b - t_0)w_0 = (y_0 - t_0)w_0 = \delta_0 w_0 \qquad (4.3.22)$$

入力 $\boldsymbol{X}$の他の要素も同様にして、

$$\begin{array}{cccc} \frac{\partial L}{\partial x_{00}} = \delta_0 w_0 & \frac{\partial L}{\partial x_{01}} = \delta_0 w_1 & \frac{\partial L}{\partial x_{02}} = \delta_0 w_2 & \frac{\partial L}{\partial x_{03}} = \delta_0 w_3 \\ \frac{\partial L}{\partial x_{10}} = \delta_1 w_0 & \frac{\partial L}{\partial x_{11}} = \delta_1 w_1 & \frac{\partial L}{\partial x_{12}} = \delta_1 w_2 & \frac{\partial L}{\partial x_{13}} = \delta_1 w_3 \\ \vdots & \vdots & \vdots & \vdots \end{array} \qquad (4.3.23)$$

だから、

$$\frac{\partial L}{\partial \boldsymbol{X}} = \begin{pmatrix} \frac{\partial L}{\partial x_{00}} & \frac{\partial L}{\partial x_{01}} & \frac{\partial L}{\partial x_{02}} & \frac{\partial L}{\partial x_{03}} \\ \frac{\partial L}{\partial x_{10}} & \frac{\partial L}{\partial x_{11}} & \frac{\partial L}{\partial x_{12}} & \frac{\partial L}{\partial x_{13}} \\ \vdots & \vdots & \vdots & \vdots \end{pmatrix} = \begin{pmatrix} \delta_0 w_0 & \delta_0 w_1 & \delta_0 w_2 & \delta_0 w_3 \\ \delta_1 w_0 & \delta_1 w_1 & \delta_1 w_2 & \delta_1 w_3 \\ \vdots & \vdots & \vdots & \vdots \end{pmatrix}$$

$$= \begin{pmatrix} \delta_0 \\ \delta_1 \\ \vdots \end{pmatrix} (w_0 \quad w_1 \quad w_2 \quad w_3) = \boldsymbol{\delta}\ {}^t\boldsymbol{W} \qquad (4.3.24)$$

式 4.3.20、式 4.3.21、式 4.3.24 により、重みの勾配、バイアスの勾配、入力の勾配を中間値の勾配 $\boldsymbol{\delta}$ との関係で表す式が得られ、入力の数によらず、同様にあらわせることがわかった．ここでまで見てきたように、中間値の勾配が得られれれば、重み、バイアス、そして入力の勾配が算出できる．その中間値の勾配は式 4.3.9 に示すように、微分の連鎖律により、損失関数の微分と活性化関数の微分(この場合は恒等関数の微分なので 1)の積となっている．だから、損失関数とニューラルネットワーク本体を分け、さらに活性化関数と、入力の重み付け和の計算を分けて、それぞれ微分が求められれば良いことになる．

ここまでニューロン 1 つについて求めてきたが、式 4.1.5 ないし式 4.1.6 によって順伝播があらわされる一般的な場合に対応しておこう．活性化関数も恒等関数に限らないものとする．ニューロンが 1 つに限らず複数の場合にも、各ニューロンそれぞれで同様の関係が成り立つはずだ．　まず中間値の勾配 $\boldsymbol{\delta}$ だが、式 4.3.10 では縦ベクトルになっていて、その各元はデータに対応する．その層のニューロンが複数からなる場合には、各ニューロンそれぞれについて $\boldsymbol{\delta}$ はそれぞれ縦ベクトルとなり、それがニューロン分だけあるはずだ．そうして行方向はデータに対応し、列方向はニューロンに対応する行列を作れば、それは式 4.1.5 に示したニューロン層の中間値ないし出力の並びそのものだ．すなわち、 $\boldsymbol{\delta}$ はその層の中間値 $\boldsymbol{U}$や出力 $\boldsymbol{Y}$と同じ形になる．そして活性化関数は形状を変えないから中間値 $\boldsymbol{U}$と出力 $\boldsymbol{Y}$とは同じ形状であり、恒等関数の種類によらず一般的にそうなる．これを踏まえて、ニューロンの数を n、データを k組として、 $\boldsymbol{\delta}$ を一般化して次式で表すことにしよう．

$$\frac{\partial L}{\partial \boldsymbol{U}} = \frac{\partial L}{\partial \boldsymbol{Y}} \frac{\partial \boldsymbol{Y}}{\partial \boldsymbol{U}} = \boldsymbol{\delta} = \begin{pmatrix} \delta_{00} & \delta_{01} & \cdots & \delta_{0n} \\ \delta_{10} & & & \vdots \\ \vdots & & & \vdots \\ \delta_{k0} & \cdots & \cdots & \delta_{kn} \end{pmatrix} \tag{4.3.25}$$

さて重み $\boldsymbol{W}$の勾配だが、1 つのニューロンをあらわす式 4.3.21 は縦ベクトルとなっている．これが各列各ニューロンに対応して列方向に並ぶ．すなわち行方向に各データ、列方向に各ニューロンが並んだ行列となって、

$$\begin{aligned} \frac{\partial L}{\partial \boldsymbol{W}} &= \begin{pmatrix} \dfrac{\partial L}{\partial w_{00}} & \dfrac{\partial L}{\partial w_{01}} & \cdots & \dfrac{\partial L}{\partial w_{0n}} \\ \dfrac{\partial L}{\partial w_{10}} & & & \vdots \\ \vdots & & & \vdots \\ \dfrac{\partial L}{\partial w_{m0}} & \cdots & \cdots & \dfrac{\partial L}{\partial w_{mn}} \end{pmatrix} \\ &= {}^t\boldsymbol{X}\,\boldsymbol{\delta} = \begin{pmatrix} x_{00} & x_{10} & \cdots & x_{k0} \\ x_{01} & & & \vdots \\ \vdots & & & \vdots \\ x_{0m} & \cdots & \cdots & x_{km} \end{pmatrix} \begin{pmatrix} \delta_{00} & \delta_{01} & \cdots & \delta_{0n} \\ \delta_{10} & & & \vdots \\ \vdots & & & \vdots \\ \delta_{k0} & \cdots & \cdots & \delta_{kn} \end{pmatrix} \end{aligned} \tag{4.3.26}$$

次はバイアスだ．その勾配はニューロン 1 つの場合には、式 4.3.22 に示すように $\boldsymbol{\delta}$の縦ベクトルの元を加え合わせてスカラーを得たのだから、重み同様にニューロン毎に列方向に、これが並んでベクトルとなり、

$$\frac{\partial L}{\partial \boldsymbol{B}} = \left(\frac{\partial L}{\partial b_0} \quad \frac{\partial L}{\partial b_1} \quad \cdots \quad \frac{\partial L}{\partial b_n}\right) = \left(\sum_{i=0}^{k} \delta_{i0} \quad \sum_{i=0}^{k} \delta_{i1} \quad \cdots \quad \sum_{i=0}^{k} \delta_{in}\right) \tag{4.3.27}$$

このようにバイアスの勾配は、行列 $\boldsymbol{\delta}$ の行方向の和をとる必要がある.

最後に入力 $\boldsymbol{X}$ の勾配だ. ニューロンが複数になると、重み $\boldsymbol{W}$ は列方向に拡張される. そうすると転置した行列 ${}^t\boldsymbol{W}$ は行方向に拡張されて、ニューロンの数の行数となる. いっぽう $\boldsymbol{\delta}$ は式 4.3.26 に示すように複数のニューロンに対応して列方向に拡張される. そして $\boldsymbol{\delta}$ と ${}^t\boldsymbol{W}$ の積として次式により、$\frac{\partial L}{\partial \boldsymbol{X}}$ を求めることが出来る.

$$\begin{aligned}\frac{\partial L}{\partial \boldsymbol{X}} &= \begin{pmatrix} \frac{\partial L}{\partial x_{00}} & \frac{\partial L}{\partial x_{01}} & \cdots & \frac{\partial L}{\partial x_{0m}} \\ \frac{\partial L}{\partial x_{10}} & & & \vdots \\ \vdots & & & \vdots \\ \frac{\partial L}{\partial x_{k0}} & \cdots & \cdots & \frac{\partial L}{\partial x_{km}} \end{pmatrix} \\ &= \boldsymbol{\delta}\ {}^t\boldsymbol{W} = \begin{pmatrix} \delta_{00} & \delta_{01} & \cdots & \delta_{0n} \\ \delta_{10} & & & \vdots \\ \vdots & & & \vdots \\ \delta_{k0} & \cdots & \cdots & \delta_{kn} \end{pmatrix} \begin{pmatrix} w_{00} & w_{10} & \cdots & w_{m0} \\ w_{01} & & & \vdots \\ \vdots & & & \vdots \\ w_{0n} & \cdots & \cdots & w_{mn} \end{pmatrix}\end{aligned} \tag{4.3.28}$$

これで一般的な場合に、重み、バイアス、入力の勾配を求めることが出来る.

[補足]

損失 L と各勾配の関係について振り返っておこう. 損失 L は複数のニューロンの出力全体の値だ. たとえば図 4.2.4 のパーセプトロンでは 3 つの one_hot の出力があって、たとえば、そのあるデータの正解値が 010 とすれば 3 つのニューロンの 1 つ目が 0 からどれだけ隔たり、2 つ目が 1 からどれだけ隔たり、3 つ目が 0 からどれだけ隔たるかの総和だ. しかし少なくとも重みやバイアスが出力に影響を与えるのは、その重みやバイアスがかかわる、そのニューロンの出力だけであって、重みやバイアスがニューロンごとに与えられる限りは、その重みやバイアスが関係するニューロンの損失だけを見ればよいことは明らかだ. じっさい式 4.3.17 で損失 L は $(y-t)^2$ の総和によるが、重みやバイアスの勾配を求める際に行う偏微分で、 y や t は着目する重みやバイアスのかかわるニューロンの以外の項は定数とみなされて、結果

に影響しない．そして式 4.3.26 と式 4.3.27 で重みの勾配とバイアスの勾配は、ニューロンごとに互いに独立に算出した式 4.3.20 と式 4.3.21 が列方向に並ぶだけだ．いっぽう入力に関しては、入力がすべてのニューロンに等しく配られていることから、出力の全体の損失が勾配の計算にかかわることも明らかだ．そういう意味では式 4.3.24 で示すのは $\boldsymbol{X}$ の勾配の中の、着目したニューロンに関する部分を抜き出したものというべきだろう．そして式 4.3.28 で入力の勾配は、各ニューロンの中間値の勾配と転置した重みの行列積になっていて、各ニューロンに対応する $\boldsymbol{\delta}$ が各ニューロンの重みによる重み付け和として合算されている．これは損失 L が複数のニューロンの出力全体の値だということと符合する．とまれここまで式 4.1.5 で示される一般的な場合について、重み、バイアス、入力の勾配を求めるのに、行列の行や列の並びに着目して説明したが、ヤコビ行列の連鎖律として得ることが出来る．詳しくは線形代数に関する書籍等を参照されたし．

[補足]

活性化関数では中間値を出力に変換する際に形状は変わらない．つまり式 4.1.5 に示すように中間値が k 行 n 列の場合であれば出力も k 行 n 列だ．そしてその変換は要素ごとに独立に行われる．だから出力の勾配から中間値の勾配 $\boldsymbol{\delta}$ には、スカラー値の微分の連鎖律がそのまま当てはまる．そういう理由から、まず $\boldsymbol{\delta}$ を求め、それからその $\boldsymbol{\delta}$ を使って積和部分を計算するのは一般的に行われる．

4.4 逆伝播と学習

前節で損失関数をニューロンの重み、バイアス、入力の各項で偏微分することによって、その勾配を求めた．その中で活性化関数を通過する前の中間値の勾配 δ を求め、各勾配はそれとの関係によって求めることができることを示した．このことは、入力から出力そして損失関数と順に伝わるのとは逆に、損失関数のパーセプトロン本体の出力に対する微分→活性化関数の微分→ニューロンの積和のところの微分と辿りながら、重み、バイアス、入力の勾配を求める方法を示唆している．この逆方向に辿る信号伝播を逆伝播という．
ここでは勾配を求める手段として逆伝播を、そして重みやバイアスを調整する方法として勾配降下法による更新を組み込む．そして実際にパーセプトロンの学習を行う．
またここで重みとバイアスを求める別のアプローチとして、直接法による解を求め、第 3 章線形回帰を振り返ることにする．

4.4.1 逆伝播を組み込む

ここからは損失関数、活性化関数、ニューロン層に逆伝播を組み込んでいくことにしよう．
もちろん、前節で求めた勾配を求める数式に従って進めていくのだ．ではまず、損失関数の Mean Squared Error ないし平均二乗誤差に逆伝播を組み込もう．これをリスト 4.4.1 に示す．

リスト 4.4.1：逆伝播を備えた Mean Squared Error

```
import numpy as np

class MeanSquaredError:
    def forward(self, y, t):
        self.y = y
        self.t = t
        self.k = y.size // y.shape[-1]
        l = 0.5 * np.sum((y - t) ** 2)
        return l / self.k

    def backward(self, gl=1):
        y = self.y
        t = self.t
        gy = gl * (y - t)
        return gy / self.k
```

リスト 4.2.1 で forward メソッドを実装済みだが、リスト 4.4.1 の forward メソッドは、逆伝播で順伝播の際の出力と正解値が必要だから、self.y = y と self.t = t の 2 つを追記して、インスタンス変数に保存する．backward メソッドでは、これを取り出して、式 4.3.9 に

従って勾配を求めるが、その際にデフォルト引数 gl=1 で受けとった勾配を算出した値に掛け合わせて、後で出てくるニューロン層の逆伝播などと実装を揃えている．しかし通常は引数は与えずに gl=1 のままとして、中で算出した値そのままの勾配にする．

次は活性化関数の恒等関数だ．これをリスト 4.4.2 に示す．

リスト 4.4.2：逆伝播を備えた恒等関数

```
import numpy as np

class Identity:
    def forward(self, x):
        return x

    def backward(self, gy):
        return gy
```

リスト 4.4.2 で forward メソッドはリスト 4.1.2 そのものだ．backward メソッドを追加するが、恒等関数なので逆伝播でも受け取った勾配をそのまま流すだけなので、引数の gy をそのまま返り値とする．しかしここで活性化関数が恒等関数に限らず、一般の場合について考えておく必要があるだろう．これまで説明してきたように、ニューロン層において $\boldsymbol{\delta}$ は出力の勾配と活性化関数の微分の積で $\boldsymbol{\delta} = \frac{\partial L}{\partial \boldsymbol{Y}} \frac{\partial \boldsymbol{Y}}{\partial \boldsymbol{U}}$ だ．活性化関数の backward メソッドは引数として、gy すなわち $\frac{\partial L}{\partial \boldsymbol{Y}}$ を受け取り、自身の微分 $\frac{\partial \boldsymbol{Y}}{\partial \boldsymbol{U}}$ を求めて、$\boldsymbol{\delta} = \frac{\partial L}{\partial \boldsymbol{Y}} \frac{\partial \boldsymbol{Y}}{\partial \boldsymbol{U}}$ を計算して返り値とする．引数で受け取る勾配と自身の微分の積を返り値とするこの操作は、リスト 4.4.1 の誤差関数の逆伝播でも同様だ．

続いてニューロン層の逆伝播だ．これをリスト 4.4.3 に示す．

リスト 4.4.3：NeuronLayer の逆伝播

```
class NeuronLayer(BaseLayer):

    (コントラクタはリスト 4.1.4、forward メソッドはリスト 4.1.5、
     fix_configuration メソッドはリスト 4.1.6 に掲載 )

    def backward(self, grad_y, **kwargs):
        delta = self.activator.backward(grad_y, **kwargs)
        self.grad_w = np.dot(self.x.T, delta)
        self.grad_b = np.sum(delta, axis=0)
        grad_x = np.dot(delta, self.w.T)
        return grad_x
```

リスト 4.4.3 の中に記したように NeuronLayer クラスには既に、コントラクタ(リスト 4.1.4)、forward メソッド(リスト 4.1.5)、fix_configuration メソッド(リスト 4.1.6)が実装済みだが、これらはそのままで、backward メソッドを加える. backward メソッドの引数は grad_y と**kwargs としている. ここで辞書型の可変長引数**kwargs は、backward メソッドの機能の追加に備えて用意しているが、活性化関数に引数として渡すだけで、ここでは使わない. いっぽう引数 grad_y は下流側からの勾配として与えるが、損失関数の場合と違ってデフォルト値を与えない. 下流側に損失関数が置かれる場合には、そこからの勾配を grad_y に与えるようにする. また下流側が別のニューロン層の場合には、その下流側のニューロン層の入力の勾配を grad_y に与えるようする. どちらの場合も grad_y を出力の勾配として各勾配を求める起点とする. backward メソッドではじめに求めるのは $\boldsymbol{\delta}$ で、プログラムでは $\boldsymbol{\delta}$ はこの文字が使えないから delta としている. 活性化関数はリスト 4.1.5 の forward メソッドでは最後に通るが、逆伝播である backward メソッドでは最初に通る.

delta = self.activator.backward(grad_y, **kwargs)

で、activator が Identity 恒等関数ならば、delta には grad_y がそのまま返ってくるが、一般には出力の勾配と活性化関数の微分の積で $\boldsymbol{\delta} = \frac{\partial L}{\partial \boldsymbol{Y}} \frac{\partial \boldsymbol{Y}}{\partial \boldsymbol{U}}$ だ. delta が得られたら、次の 3 行で重みとバイアスと入力の勾配を求める.

```
self.grad_w = np.dot(self.x.T, delta)
self.grad_b = np.sum(delta, axis=0)
grad_x = np.dot(delta, self.w.T)
```

これらは、それぞれ式 4.3.26、式 4.3.27、式 4.3.28 をそのままプログラムしたものだ. バイアスの勾配は式 4.3.27 に従い、numpy の関数 sum で足し合わせる際の axis の指定により、行方向での足し合わせとなっている. grad_w、grad_b はインスタンス変数として保存するが、これらは、後で説明する重みやバイアスの更新に使う.

4.4.2 勾配降下法

前節で損失関数を起点に誤差逆伝播によってニューラルネットワークの重みやバイアスの勾配を求める方法を見てきた. 勾配がわかったならば最小値を探すことは、その勾配に従えば自然に行える. とはいえその勾配が単調なんものとは限らず、どこにいるのかによって勾配は変わることが普通だから、いっぺんに最小値の場所に至ることは難しい. そこで今いるところを起点にして、そこでの勾配を求め、その勾配にしたがって少し動かし、そして動いた先でまた勾配を求め、そしてまた少し動かすということを繰り返すことによって、最小値の場所に到達することを図る. これを勾配降下法という.

[補足]

勾配は、重みやバイアスのうち着目するものを変数、そのほかを定数とみなした偏微分によって求められる. しかし重みやバイアスはいずれも変数なのであって変わるものだ. そしてそれが変わった場合には、偏微分の値を算出した際の前提が変わることになるから、当然ながら着目したものの勾配もまた変わる. つまり求められる勾配はあくまで、その時点での値ーその値の組み合わせに対するものであって、それを受けて調整して値そのものが変われば、これは変わるから求め直さなければならない.

勾配降下法を行うために作らなければならないものは極めて単純だ. すなわち、重みとバイアスの更新を BaseLayer に組み込むだけで良い. これをリスト 4.4.4 に示す.

リスト 4.4.4：重みとバイアスの更新

```
class BaseLayer:

    #(コントラクタはリスト 4.1.1 に、init_parameter メソッドはリスト 4.1.7 に掲載)

    def update(self, eta=0.01, **kwargs):
        self.w -= eta * self.grad_w
        self.b -= eta * self.grad_b
```

BaseLayer に update メソッドを定義する. 重みとバイアスの勾配はインスタンス変数の self.grad_w と self.grad_b に、それぞれ逆伝播の際に保存されている. そして学習率をデフォルト引数 eta=0.01 で与えて学習率×勾配だけ重みやバイアスを減ずる. 勾配が正ならば値が大きくなると損失が大きくなるから値を小さくし、勾配が負ならば値が大きくなると損失が小さくなるのだから値を大きくする、と考えれば引き算となっている意味が分かるのでは

ないだろうか？ 説明が後先になったが、update メソッドには辞書型の可変長引数も用意してあるが、将来の拡張のためでここでは使わない.

4.4.3 便利な機能の追加

いよいよパーセプトロンを学習させるために必須の機能が用意できた. しかしここで今後のために便利な機能をいくつか作って、common_function に置いておこう.

まずは正解率を取得する機能を関数にしよう. リスト 4.2.4 やリスト 4.3.3 に示すような分類問題では、ニューラルネットワークの出力は、行方向は各データに対応するが、列方向は各カテゴリーの番号に対応する. そして列方向で値が最大のものを機械の判定とする. つまりニューラルネットワークの出力は one_hot の正解値の形状と同じであり、ニューラルネットワークの出力を one_hot の正解値と比較して、one_hot の値が 1 のところの値が最大となっている場合が正解、そうでない場合が不正解だ. これをふまえて正解率を求めるための関数を用意しよう.

リスト 4.4.5：正解率の取得

```
def get_accuracy(y, t, mchx=False):
    '''
    y:順伝播の結果と、対応する t:正解とを与え、分類の正解率を返す
    mchx=True では正誤表も返す
    '''
    result = np.argmax(y, axis=-1)
    if y.shape == t.shape: # 正解が one_hot の場合
        correct = np.argmax(t, axis=-1)
    elif y.ndim == t.ndim + 1:
        correct = t
    else:
        raise Exception('Wrong dimension of t')
    errata = result == correct
    accuracy = float(np.sum(errata) / len(y))
    if mchx:
        return accuracy, errata
    else:
        return accuracy
```

リスト 4.4.5 に関数 get_accuracy()を示す. 引数の y にはニューラルネットワークの出力を与え、引数の t には正解値を与える. すでに述べたように y は列方向で最大の値のところが機械の判定だから、result = np.argmax(y, axis=-1)で結果としている. このとき axis=-1 は末尾の次元であって、y が行列ならば列方向を指す. いっぽう t は one_hot で与えても、カテゴリ

一番号で与えても良いように、形状を y との比較でチェックして、one_hot ならば y 同様に numpy の関数 argmax で最大のところのインデクスを求め、もともとカテゴリー番号ならば、そのままの値を正解値としている.

errata = result == correct

は result が correct と一致していたら True、さもなくば False を errata にベクトルとして並べる. その正解の合計をデータの数 len(y)で割って、

accuracy = float(np.sum(errata) / len(y))

で正解率を求める. あとは引数の mchx の指定に従って、正解率もしくは正解率と errata を返り値とする.

つづいて学習経過のグラフ表示だ. 学習中にはエラーや正解率を求め、これを学習完了後にグラフ表示して、学習がうまくいっているのかなどの確認を行う. グラフ表示は matplotlib を使って比較的簡単にできるが、関数を用意しておくと便利だ.

リスト 4.4.6：学習の進捗のグラフ

```
def graph_for_error(*data, **kwargs):
    labels = kwargs.pop('label',  None)
    xlabel = kwargs.pop('xlabel', None)
    ylabel = kwargs.pop('ylabel', None)
    legend = True

    if labels is None: # labels がない場合は None を data 分並べる
        labels = (None,) * len(data)
        legend = False

    elif len(data)==1 and type(labels) is str:
        labels = labels,

    elif type(labels) in(list, tuple) and len(data)==len(labels):
        pass

    else:
        raise Exception('length of data and label mismatch.')

    for d, l in zip(data, labels):
        plt.plot(d, label=l)
    if legend:
        plt.legend()
    plt.xlabel(xlabel)
    plt.ylabel(ylabel)
    plt.show()
```

リスト 4.4.6 を簡単に説明しよう．関数名は graph_for_error() だ．引数は何が来ても取り敢えず受け取るように*args, **kwargs としておいて、kwargs から、

```
labels = kwargs.pop('label',  None)
xlabel = kwargs.pop('xlabel', None)
ylabel = kwargs.pop('ylabel', None)
```

で、個々のデータのラベル、x 軸ラベル、y 軸ラベルを kwargs から取り出す．matplotlib では label=None はラベル非表示なので、ラベルの指定の有無に応じて、labels がデータと同数になるようにする．あとはデータとラベルを組にして、

```
for d, l in zip(data, labels):
    plt.plot(d, label=l)
```

で回してプロットし、軸ラベルを表示あるいは非表示して、グラフを描画する．

4.4.4 パーセプトロンの学習

ニューロンの重みやバイアスの更新を組み込んだ．学習を進めるのに便利な機能も作った．これでパーセプトロンを学習させることができる．学習の手順はこれまで何度か述べてきたとおりだが、あらためて整理しておくと次のようになる．データを用意し、モデルと損失関数を初期化する．初期化の時点に重みは乱数でバイアスは 0 だ．その状態でデータをモデルに入力して順伝播を行って、その時点での状態を確定しつつ出力する．その出力と正解値を損失関数に入力して損失を求める．次に今度は損失関数の逆伝播で出力の勾配を求め、続いて活性化関数そしてニューロンの積和と逆伝播して、重みとバイアスの勾配を求める．勾配が求まったら、それを使って重みとバイアスの更新を行う．そうしたらまたデータの入力に戻って、この手順を繰り返す．

ではさっそくアヤメの回帰の例に当てはめて、この手順を組み込んでみよう.

リスト 4.4.7：アヤメのデータを近似する～その 3

```
import numpy as np
import matplotlib.pyplot as plt
import Neuron, LossFunctions, Iris
import common_function as cf

# データを用意
data, correct, target = Iris.get_data()

# モデルを作る
model = Neuron.NeuronLayer(1)
loss_function = LossFunctions.MeanSquaredError()

x = data[:, 2:3]
t = data[:, 3:4]

# 学習
n_learn = 101
errors = []
for i in range(n_learn):
    y = model.forward(x)
    l = loss_function.forward(y, t)
    gy = loss_function.backward()
    model.backward(gy)
    model.update()
    errors.append(l)
    print('{:6d} {:6.3f}'.format(i, float(l)))

# 学習後に経過表示
cf.graph_for_error(errors)

# グラフ表示
plt.scatter(x, t, marker='x')
plt.plot(x, y)
plt.show()
```

リスト 4.4.7 はリスト 4.3.2 と多くが共通で、違うのは学習とその経過表示だ. 学習は回数を n_learn で指定し for ループで繰り返す. errors = [] は経過の記録用だ. for ループの中で以下に示す、順伝播→損失→損失の逆伝播→逆伝播→更新は先に述べた手順通りだ.

```
y = model.forward(x)
l = loss_function.forward(y, t)
gy = loss_function.backward()
model.backward(gy)
model.update()
```

毎回これを行うたびに、errors.append(l) で損失の値を記録していく．同時に print 文で回数と損失を表示しておく．あとはリスト 4.4.6 で作った関数 graph_for_error()に記録した損失をまとめて渡して表示する．最後にデータとその近似を可視化するのはリスト 4.3.2 と同じだ．どうだろう学習の様子が観察でき、その結果としての回帰もうまくいったのではないだろうか？
つづいてアヤメの品種の分類をやってみよう．

リスト 4.4.8：アヤメの品種を分類する～その 3

```
import numpy as np
import Neuron, LossFunctions, Iris
import common_function as cf

# データを用意
x, c, t = Iris.get_data()

# モデルを作る
model = Neuron.NeuronLayer(3)
loss_function = LossFunctions.MeanSquaredError()

# 学習
n_learn = 1001
errors, accuracy = [], []
for i in range(n_learn):
    y = model.forward(x)
    loss = loss_function.forward(y, c)
    gy = loss_function.backward()
    model.backward(gy)
    model.update()
    acc = cf.get_accuracy(y, c)
    errors.append(loss)
    accuracy.append(acc)

    if i%100==0:
        print('{:6d} {:6.3f} {:6.3f}'.format(i, float(loss), acc))

# 学習後に経過表示
cf.graph_for_error(errors, accuracy, label=('error','accuracy'), xlabel='n_learn')
```

リスト 4.4.8 はリスト 4.3.3 と多くが共通で、学習とその経過表示はリスト 4.4.7 とほとんど同じだ．ただこれは分類なので、学習中にリスト 4.4.5 で用意した関数 get_accuracy()を使って、正解率を求めて、損失と併せて記録ならびに表示している．また、学習回数が多いので、学習途中の表示は if i%100==0:と条件を付けて、100 回に 1 回に減らしている．分類もまたうまくいったのではないだろうか？

もう１つ、リスト 4.1.10 で作った論理演算のパーセプトロンも学習させてみよう.

リスト 4.4.9：簡単なパーセプトロン～その 2

```
import numpy as np
import Neuron, LossFunctions
import common_function as cf

# 教師データの用意
x = np.array([[0, 0],[0, 1],[1, 0],[1, 1]])
t = np.array([[0],[1],[1],[1]])

# モデルを作る
model = Neuron.NeuronLayer(1)
loss_function = LossFunctions.MeanSquaredError()

# 学習
n_learn = 1001
errors = []
for i in range(n_learn):
    y = model.forward(x)
    loss = loss_function.forward(y, t)
    gy = loss_function.backward()
    model.backward(gy)
    model.update()
    errors.append(loss)

    if i%100==0:
        print('{:6d} {:6.3f}'.format(i, float(loss)))

# 学習後に経過表示
cf.graph_for_error(errors, label='error', xlabel='n_learn')

# 学習済での順伝播と結果
print('input¥n', x)
y = model.forward(x)
z = y > 0.5
print('result¥n', z)
```

ここでは教師データとして、入力 x = np.array([[0, 0],[0, 1],[1, 0],[1, 1]]) に対し、正解値 t = np.array([[0],[1],[1],[1]]) を用意した. ここでは入力と正解値の組み合わせは「論理和」になっているが、この正解値は「論理積」ならば、[[0],[0],[0],[1]] というようにすればよく、いろいろ変えてやってみてほしい. リスト 4.1.10 の場合には活性化関数にステップ関数を指定したが、ここでは指定せずにデフォルトの恒等関数でモデルを作る. これは学習が必要だから、微分できず逆伝播が定義できないステップ関数は都合が悪いからだ. 学習はリスト 4.4.7 やリスト 4.4.8 と同様だ. 最後に学習済での順伝播と結果だが、活性化

関数を恒等関数にしたので z = y > 0.5 で判定して z に結果を得て表示するようにしている.
パーセプトロンによる論理演算も、論理和や論理積など、うまくい合ったのではないだろうか？ ではははたして排他的論理和はどうだろうか？ これも是非試してみてほしいのだが、実はこのパーセプトロンでは排他的論理和を表現する能力はない. もちろん解決の方法はあるのだが、ここでは後日への課題としておこう.

さてここまで、ニューロン層が 1 層だけからなる簡単なパーセプトロンを作っていろいろ見てきた. ニューロンで行われる演算で、重みやバイアスを初期化しただけの、いわば生まれたばかりの神経細胞は、何も意味のある結果がえられないのに対し、正解値との隔たりをあらわす損失を評価して、この値を小さくするように重みとバイアスを自動的に調節することこそ機械学習にほかならない. すなわちデータを用いて結果を出し、損失を求めて、それを小さくすべく、勾配にもとづいて重みとバイアスを調節するのであり、いうならばモデルが自分でデータから学ぶようにするのである. そしてその際に鍵となるのが損失だったのだ.
いずれにせよ、ここで取り上げたのは回帰と分類と論理演算のごく簡単な例に過ぎないが、それでもデータから学習させることによって、単純なパーセプトロンが用途に応じた様々な働きをすることがわかったのではないだろうか？

4.4.5 線形性の確認

数学では、写像 f が線型であれば、 f について加法性と斉次性（作用との可換性）が満たされるとされる．加法性とは、任意の x, y に対して、

$$f(x+y) = f(x) + f(y) \tag{4.4.1}$$

が成り立つことであり、斉次性とは、任意の x, α に対して、

$$f(\alpha x) = \alpha f(x) \tag{4.4.2}$$

が成り立つことである．いっぽう、リスト 4.4.7 やリスト 4.4.8 のパーセプトロンは、入力と重みおよびバイアスの線形結合にほかならないから、線形性が保たれているはずだ．そこでこれを確かめておくことにしよう．

リスト 4.4.10：パーセプトロンの線形性の確認

```
import numpy as np
import Neuron

m, n, k = 4, 3, 5  # 入出力の大きさ
a = 2              # 斉次性確認の係数

x = np.random.rand(k, m)
y = np.random.rand(k, m)

f = Neuron.NeuronLayer(m, n).forward # 対象の関数の設定

print(np.abs(f(x)+f(y)-f(x+y))<1e-7) # 加法性
print(np.abs(a*f(x)-f(a*x))<1e-7)    # 斉次性
```

入出力の大きさは、これまでと同じく、m が入力、n が出力の数であり、k はデータの組の数とした．また斉次性確認の係数は a として、これらを与えるようにした．そして x と y にこれで指定された大きさの乱数を生成する．いっぽう線形性を確認する対象の関数は、パーセプトロンの順伝播なので、f = Neuron.NeuronLayer(m, n).forward でモデルを生成するとともに、その順伝播を f と読みかえておく．これで準備が整うから、あとは線形性の定義にそって確認する．このとき加法性をみるのに f(x)+f(y)==f(x+y) のようにすると、一致すべきところで計算誤差によるごくわずかな不一致となることがあるから、np.abs(f(x)+f(y)-f(x+y))<1e-7 として、微小値 1×10^{-7} より小さければ一致したものとして表示するようにした．斉次性も同様だ．とまれこれを実行すると、True が並んだ行列が出力されるはずだ．

[補足]
リスト 4.4.10 でパーセプトロンが線形であることを確認した．このとき重みやバイアスは、リスト 4.1.7 に示す BaseLayer の init_parameter()メソッドによって、重みは乱数、バイアスは'0'で初期化されている．実はここで、バイアスに何らかの値を設定すると線形性は崩れ、結果はリスト 4.4.10 とは全く違うものになる．これは $y = ax + b$ の形の方程式が $b = 0$ に限って線形で、$b \neq 0$ では線形でないのと同じであり、線形回帰では $y = ax + b$ の形の方程式も $y = (1 \quad x)\begin{pmatrix} b \\ a \end{pmatrix}$ の形にすることで定数項をかけ算に組み入れて線形にしたのである．

さてここで少々乱暴なことを言わせてもらうと、「これは表現方法が違うだけだから、$y = ax + b$ がもともと線形性を持っていた」と考えるのは自然なことで、演算としての足し算 $+$ が、その表現方法も含めて難しいということの一端が伺い知れるのではないだろうか？

4.4.6 線形回帰とのつながり

ニューロン 1 層だけのパーセプトロンで、活性化関数を恒等関数とした場合には、出力は入力の重み付け和であって線形結合にほかならない．だから重みやバイアスの値は直接法で求めることが出来る．ここでは「第 3 章 線形回帰」で詳しく説明したことをふまえて、ごく簡単に説明することにしよう．

まず簡単のために、入力は m 個とするが、ニューロン 1 つ、データの組も 1 つとする．式 4.1.1～式 4.1.4 を見ながら入出力の関係を式であらわすと、活性化関数が恒等関数ならば、ニューロン 1 層だけのパーセプトロンにおいて、

$$y = x_0 \times w_0 + \cdots + x_m \times w_m + b = (1 \quad x_0 \quad \cdots \quad x_m)\begin{pmatrix} b \\ w_0 \\ \vdots \\ w_m \end{pmatrix} \tag{4.4.3}$$

ニューロン n 個、データの組を k 個として一般化すれば、

$$\begin{pmatrix} y_{00} & \cdots & y_{0n} \\ \vdots & & \vdots \\ y_{k0} & \cdots & y_{kn} \end{pmatrix} = \begin{pmatrix} 1 & x_{00} & \cdots & x_{0m} \\ \vdots & \vdots & & \vdots \\ 1 & x_{k0} & \cdots & x_{km} \end{pmatrix}\begin{pmatrix} b_0 & \cdots & b_n \\ w_{00} & \cdots & w_{0n} \\ \vdots & & \vdots \\ w_{m0} & \cdots & w_{mn} \end{pmatrix} \tag{4.4.4}$$

当然ながら、これは式 4.1.5 で $y = u$ としたものと同じだ．ただしバイアス項を重みの行列に組み入れ、それに対応して入力の行列に'1'の列を加えている．ここで入力の行列は線形回帰にならって $\boldsymbol{\Phi}$、重みとバイアスの行列はバイアスも含めて $\boldsymbol{W}$ とおく．

$$\boldsymbol{\Phi} = \begin{pmatrix} 1 & x_{00} & \cdots & x_{0m} \\ \vdots & \vdots & & \vdots \\ 1 & x_{k0} & \cdots & x_{km} \end{pmatrix} \qquad \boldsymbol{W} = \begin{pmatrix} b_0 & \cdots & b_n \\ w_{00} & \cdots & w_{0n} \\ \vdots & & \vdots \\ w_{m0} & \cdots & w_{mn} \end{pmatrix} \tag{4.4.5}$$

そうすると、式 4.4.4 は次式となる.

$$\boldsymbol{Y} = \boldsymbol{\Phi W} \tag{4.4.6}$$

ここで、$\boldsymbol{Y}$ は出力であり $\boldsymbol{W}$ が求める対象の変数、そして $\boldsymbol{\Phi}$ は入力 x と定数'1'が並んでいて、入力 x は $\boldsymbol{W}$ を求めるにあたってデータとして与えられる. したがってこれは「第 3 章線形回帰」で扱ったのとまったく同じだ. ただし、第 3 章では入力が 1 つのいわゆる単回帰を扱い、その際には 1 つの入力 x に対して、複数の関数 $\varphi_0(x), \ldots, \varphi_1(x)$ の値を並べた. しかし式 4.4.4 ではもともと別の入力である点が異なるが、 $\boldsymbol{\Phi}$ の 1 列目がすべて'1'で、それ以外はデータによって決まる値が並んだ行列となっているから、計算上はまったく同じように扱うことが出来る.

出力 $\boldsymbol{Y}$ は、正解値 $\boldsymbol{T}$ に隔たり $\boldsymbol{E}$ が加わったものとして、

$$\boldsymbol{Y} = \boldsymbol{\Phi W} = \boldsymbol{T} + \boldsymbol{E} \tag{4.4.7}$$

この各辺の各項の左から ${}^t\boldsymbol{\Phi}$をかけて、

$${}^t\boldsymbol{\Phi Y} = {}^t\boldsymbol{\Phi\Phi W} = {}^t\boldsymbol{\Phi T} + {}^t\boldsymbol{\Phi E} \tag{4.4.8}$$

ここで詳細は省略するが、損失 L を式 4.3.1 で定義して、損失が最小となるときの勾配は 0 だからすなわち、 $\dfrac{\partial L}{\partial \boldsymbol{W}} = \boldsymbol{0}$ となって、

$${}^t\boldsymbol{\Phi E} = \boldsymbol{0} \tag{4.4.9}$$

この詳細は第 3 章ならびに補足に詳述したので必要に応じて参照されたし. とまれこれにより、

$${}^t\boldsymbol{\Phi\Phi W} = {}^t\boldsymbol{\Phi T} \tag{4.4.10}$$

$$\therefore \boldsymbol{W} = \left({}^{t}\boldsymbol{\Phi}\boldsymbol{\Phi} \right)^{-1} {}^{t}\boldsymbol{\Phi}\boldsymbol{T} \tag{4.4.11}$$

正則化項も加えておくことにする．これも計算の詳細は第3章に詳述した．
とまれ m 行 m 列の単位行列を $\boldsymbol{I}$ とおいて、

$$\boldsymbol{W} = \left({}^{t}\boldsymbol{\Phi}\boldsymbol{\Phi} + \lambda\boldsymbol{I} \right)^{-1} {}^{t}\boldsymbol{\Phi}\boldsymbol{T} \tag{4.4.12}$$

ではこれを、NeuronLayer に組み込んでおこう．

リスト 4.4.11：直接法で重みやバイアスを求める

```
class NeuronLayer(BaseLayer): # 基本的なニューロンの機能

    (コントラクタ、fix_configuration、forward、backward の
     各メソッドはリスト 4.1.4～4.1.6、4.4.3 に掲載 )

    def direct_linear_regression(self, x, t, alpha=0.0):
        '''
        線形回帰 直接法による解
        活性化関数が恒等関数の場合のみに対応
        alpha は正則化係数
        '''
        if self.activator.__class__.__name__ != 'Identity':
            raise Exception("Activator should be 'Identity'.")

        Pi = np.ones((len(x),1), dtype='f4')
        Pi = np.hstack((Pi, x))
        tPiPi = np.dot(Pi.T, Pi)
        I  = np.eye(len(tPiPi))
        tPiPi += alpha * I        # 正則化項を加える
        tPiPiI = np.linalg.inv(tPiPi)
        PiI = np.dot(tPiPiI, Pi.T)
        wb = np.dot(PiI, t)
        self.w = wb[1:, :]
        self.b = wb[0]
        print('Parameters are fixed.')
```

線形回帰の直接法で重みやバイアスを求める手段をクラス NeuronLayer のメソッドの一つ direct_linear_regression() として組み込む．

正則化係数は式 4.4.12 ではλだが、これを英語にした lambda は python で特別な意味を持つから、ここでは alpha にしている. 正解値は出力と同じ形状であることには注意が必要だ. すなわち分類問題などでは one hot の正解値だ.
また以下の 5 行、

```
tPiPi = np.dot(Pi.T, Pi)
I  = np.eye(len(tPiPi))
tPiPi += alpha * I          # 正則化項を加える
tPiPiI = np.linalg.inv(tPiPi)
PiI = np.dot(tPiPiI, Pi.T)
```

はムーア・ペンローズの一般逆行列を使えば、

```
PiI = np.linalg.pinv(Pi)
```

の 1 行に簡略化でき、正則化項を加えなくとも numpy の関数 linalg.pinv()が答えを返してくれる.

さっそく直接法を試してみよう. まずはリスト 4.2.3、4.3.2、4.4.7 と扱ってきたアヤメの花弁の長さと花弁の幅の関係だ.

リスト 4.4.12：アヤメのデータを近似する～その 4

```
import matplotlib.pyplot as plt
import Neuron, LossFunctions, Iris

# データを用意
data, correct, target = Iris.get_data()

# モデルを作る
model = Neuron.NeuronLayer(1)
loss_function = LossFunctions.MeanSquaredError()

x = data[:, 2:3]
t = data[:, 3:4]

model.direct_linear_regression(x, t)

y = model.forward(x)
l = loss_function.forward(y, t)
print('loss =', l)

# グラフ表示
plt.scatter(x, t, marker='x')
plt.plot(x, y)#, marker='o')
plt.show()
```

リスト 4.4.12 ではリスト 4.4.7 で学習に費やしていた部分をすっぽり次の 1 行、
model.direct_linear_regression(x, t)
に置き換えた．これで重みもバイアスも適切な値に設定されるから、あとは 1 回だけ順伝播と損失の計測を行って、出力と正解値をグラフに表示すればよい．
勾配法では 100 回学習させてもまだ、重みやバイアスが最適にならないのに対し、瞬時に最適値となる．

続いて分類もやっておこう．リスト 4.2.4、4.3.3、4.4.8 でアヤメの品種の分類を扱っているから、これを直接法でやってみよう．

リスト 4.4.13：アヤメの品種を分類する～その 4

```
import Neuron, LossFunctions, Iris

# データを用意
x, c, t = Iris.get_data()

# モデルを作る
model = Neuron.NeuronLayer(3)
loss_function = LossFunctions.MeanSquaredError()

model.direct_linear_regression(x, c)

y = model.forward(x)
l = loss_function.forward(y, c)
print('loss =', l)
```

これも学習に費やしていた部分をすっぽり次の 1 行、
model.direct_linear_regression(x, c)
に置き換えればよい．ただし正解値としては出力と同じ形、すなわち one hot の c を渡す．
こちらも勾配法で 1000 回学習したよりも小さな損失が一瞬で得られる．

直接法は論理演算のパーセプトロンにも使える．リスト 4.4.9 で行った勾配法の結果と比べてみてほしい．

リスト 4.4.14：簡単なパーセプトロン～その 3

```
import numpy as np
import Neuron, LossFunctions

# 教師データの用意
x = np.array([[0, 0],[0, 1],[1, 0],[1, 1]])
t = np.array([[0],[0],[0],[1]])

# モデルを作る
model = Neuron.NeuronLayer(1)
loss_function = LossFunctions.MeanSquaredError()

model.direct_linear_regression(x, t)

y = model.forward(x)
l = loss_function.forward(y, t)
print('loss =', l)

# 学習済での順伝播と結果
print('input¥n', x)
y = model.forward(x)
z = y > 0.5
print('result¥n', z)
```

[補足:損失と勾配からの正規方程式の導出]

簡単のためニューロン 1 つに着目する. 出力 $\boldsymbol{Y}$、正解 $\boldsymbol{T}$ でデータの組は k 個として、

$$\boldsymbol{Y} = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_k \end{pmatrix} \qquad \boldsymbol{T} = \begin{pmatrix} t_0 \\ t_1 \\ \vdots \\ t_k \end{pmatrix} \tag{4.4.13}$$

とおけば、損失 L は式 4.3.1 にしたがって、

$$L = \frac{1}{2k}\sum^{k}(y-t)^2 = \frac{1}{2k}\sum^{k}(y^2 - 2yt + t^2) = \frac{1}{2k}\left(\sum^{k} y^2 - 2\sum^{k} yt + \sum^{k} t^2\right) \tag{4.4.14}$$

いっぽう、

$$\sum^{k} y^2 = {y_0}^2 + {y_1}^2 + \cdots + {y_k}^2 = \begin{pmatrix} y_0 & y_1 & \cdots & y_k \end{pmatrix}\begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_k \end{pmatrix} = {}^t\boldsymbol{Y}\,\boldsymbol{Y} \tag{4.4.15}$$

同様に、

$$\sum^{k} t^2 = {}^t\boldsymbol{T}\,\boldsymbol{T} \qquad \sum^{k} yt = {}^t\boldsymbol{Y}\,\boldsymbol{T} \tag{4.4.16}$$

したがって、

$$L = \frac{1}{2k}\left({}^t\boldsymbol{Y}\,\boldsymbol{Y} - 2\,{}^t\boldsymbol{Y}\,\boldsymbol{T} + {}^t\boldsymbol{T}\,\boldsymbol{T}\right) \tag{4.4.17}$$

ここで、$\boldsymbol{Y} = \boldsymbol{\Phi W}$ という関係があるならば、これを式 4.4.17 に代入して、

$$\begin{aligned} L &= \frac{1}{2k}\left({}^t(\boldsymbol{\Phi W})\boldsymbol{\Phi W} - 2\,{}^t(\boldsymbol{\Phi W})\,\boldsymbol{T} + {}^t\boldsymbol{T}\,\boldsymbol{T}\right) \\ &= \frac{1}{2k}\left({}^t\boldsymbol{W}\,{}^t\boldsymbol{\Phi}\boldsymbol{\Phi W} - 2\,{}^t\boldsymbol{W}\,{}^t\boldsymbol{\Phi T} + {}^t\boldsymbol{T}\,\boldsymbol{T}\right) \end{aligned} \tag{4.4.18}$$

ここで、L の $\boldsymbol{W}$ に対する勾配を求めると、

$$\frac{\partial L}{\partial \boldsymbol{W}} = \frac{\partial}{\partial \boldsymbol{W}}\left\{\frac{1}{2k}\left({}^{t}\boldsymbol{W}\ {}^{t}\boldsymbol{\Phi}\boldsymbol{\Phi}\boldsymbol{W} - 2\ {}^{t}\boldsymbol{W}\ {}^{t}\boldsymbol{\Phi}\boldsymbol{T} + {}^{t}\boldsymbol{T}\,\boldsymbol{T}\right)\right\}$$

$$= \frac{1}{2k}\left(\frac{\partial}{\partial \boldsymbol{W}}\ {}^{t}\boldsymbol{W}\ {}^{t}\boldsymbol{\Phi}\boldsymbol{\Phi}\boldsymbol{W} - 2\frac{\partial}{\partial \boldsymbol{W}}\ {}^{t}\boldsymbol{W}\ {}^{t}\boldsymbol{\Phi}\boldsymbol{T} + \frac{\partial}{\partial \boldsymbol{W}}\ {}^{t}\boldsymbol{T}\,\boldsymbol{T}\right)$$

$$= \frac{1}{2k}\left[\left\{ {}^{t}\boldsymbol{\Phi}\boldsymbol{\Phi} + {}^{t}\left({}^{t}\boldsymbol{\Phi}\boldsymbol{\Phi}\right)\right\}\boldsymbol{W} - 2\ {}^{t}\boldsymbol{\Phi}\boldsymbol{T}\right]$$

$$= \frac{1}{2k}\left\{\left({}^{t}\boldsymbol{\Phi}\boldsymbol{\Phi} + {}^{t}\boldsymbol{\Phi}\boldsymbol{\Phi}\right)\boldsymbol{W} - 2\ {}^{t}\boldsymbol{\Phi}\boldsymbol{T}\right\}$$

$$= \frac{1}{k}\left({}^{t}\boldsymbol{\Phi}\boldsymbol{\Phi}\boldsymbol{W} - {}^{t}\boldsymbol{\Phi}\boldsymbol{T}\right) \qquad (4.4.19)$$

損失 L を最小にする $\boldsymbol{W}$ は、その勾配が 0 すなわち、

$$\frac{\partial L}{\partial \boldsymbol{W}} = 0$$

だから式 4.4.19 により、

$${}^{t}\boldsymbol{\Phi}\boldsymbol{\Phi}\boldsymbol{W} - {}^{t}\boldsymbol{\Phi}\boldsymbol{T} = \boldsymbol{0} \qquad (4.4.20)$$

したがって、

$${}^{t}\boldsymbol{\Phi}\boldsymbol{\Phi}\boldsymbol{W} = {}^{t}\boldsymbol{\Phi}\boldsymbol{T} \qquad (4.4.10\ \text{再掲})$$

$$\therefore \boldsymbol{W} = \left({}^{t}\boldsymbol{\Phi}\boldsymbol{\Phi}\right)^{-1}\ {}^{t}\boldsymbol{\Phi}\boldsymbol{T} \qquad (4.4.11\ \text{再掲})$$

以上により、損失 L の $\boldsymbol{W}$ に対する勾配から、正規方程式が得られた. ところで、さらに $\frac{\partial L}{\partial \boldsymbol{W}}$ の式の変形を行えば、

$$\frac{\partial L}{\partial \boldsymbol{W}} = \frac{1}{k}\left({}^{t}\boldsymbol{\Phi}\boldsymbol{\Phi}\boldsymbol{W} - {}^{t}\boldsymbol{\Phi}\boldsymbol{T}\right) = \frac{1}{k}\ {}^{t}\boldsymbol{\Phi}(\boldsymbol{\Phi}\boldsymbol{W} - \boldsymbol{T}) = \frac{1}{k}\ {}^{t}\boldsymbol{\Phi}(\boldsymbol{Y} - \boldsymbol{T}) = \frac{1}{k}\ {}^{t}\boldsymbol{\Phi}\boldsymbol{E} \qquad (4.4.21)$$

$\frac{\partial L}{\partial \boldsymbol{W}} = \boldsymbol{0}$ ならば ${}^{t}\boldsymbol{\Phi}\boldsymbol{E} = \boldsymbol{0}$ となることも判明した. $\boldsymbol{E} = \boldsymbol{0}$ にはならないかわりに ${}^{t}\boldsymbol{\Phi}\boldsymbol{E} = \boldsymbol{0}$ となっていると考えると面白い. なおニューロン 1 つについて成り立つ上述の関係が、ニューロンが複数個で層となっている場合にも、それぞれのニューロンで成り立つとすれば層全体について成り立つ. なぜならば入力は共通であるものの、重み $\boldsymbol{W}$、出力 $\boldsymbol{Y}$、そして与えられる正解値 $\boldsymbol{T}$ はいずれもニューロンごとに独立だからだ.

4.5 パーセプトロンで画像認識

パーセプトロンが回帰や分類に使えることを見てきた．アヤメの品種分類で入力はがく片の長さ、がく片の幅、花弁の長さ、花弁の幅の 4 つの数値だったが、これらの数値のかわりに、画像を入力としたらどうだろうか？というのも画像もまた数値の集まりにほかならないからだ．そしてその数値に応じた分類を行って、何らかのカテゴリーに分類すれば、それはすなわち、その画像を認識することにほかならない．もう少し具体的に見てみよう．たとえば図 4.5.1 に示すように 8×8=64 個の明暗からなる手書き数字の画像データを入力とする．そして数字の 0～9 に対応して、ニューロン 1 つを 1 つのカテゴリーに割り当てた 10 個のニューロンから成るパーセプトロンで 10 個の出力を行って、その値の大小関係を判定結果として 0～9 に分類すれば、これは 1 種の文字認識あるいは画像認識と言えるのではないだろうか？

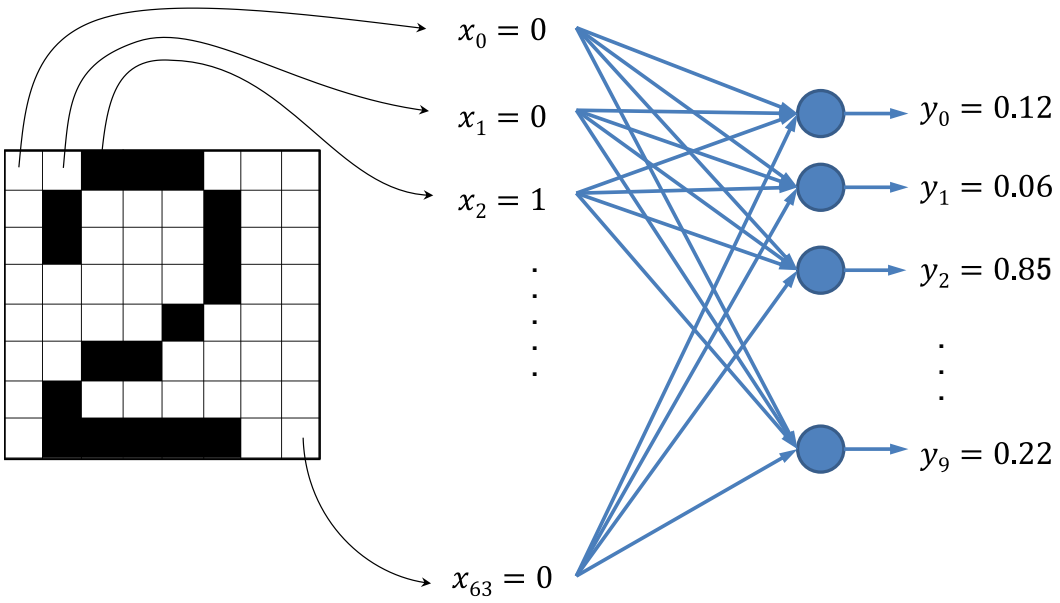

図 4.5.1 パーセプトロンで文字認識(概念図)

このようにすれば、様々な筆跡の手書き数字を人が読むように 0～9 のどれかを判別することもできるだろうし、さまざまな表情や向きの顔写真が誰かを見分けることなどもできそうだ．そこで対象とするデータを準備して、それをパーセプトロンに学習させて画像認識に取り組んでいくことにしよう．

4.5.1 データの準備

scikit-learn には、アヤメのデータのほかにも、様々なデータが用意されているので、その中から手書き数字のデータと顔写真のデータを用意しよう.

Python のスクリプトの画面で、

```
>>> from sklearn import datasets
```

と打ち込んでから、

```
>>> digits = datasets.load_digits()
```

とすると、digits に手書き数字のデータや正解ラベルなどが得られる.

```
>>> type(digits)
```

と打ち込むと、

```
<class 'sklearn.utils.Bunch'>
```

と返ってくるが、digits は「辞書もどき」のようになっている. すなわち、

```
>>> digits.keys()
```

とすると、

```
dict_keys(['data', 'target', 'target_names', 'images', 'DESCR'])
```

と返ってきて、digits はキーを使ってアクセスできる 5 つの要素から成っていて、

```
>>> digits['data']
```

では、

```
array([[ 0.,  0.,  5., ...,  0.,  0.,  0.],
       [ 0.,  0.,  0., ..., 10.,  0.,  0.],
       [ 0.,  0.,  0., ..., 16.,  9.,  0.],
       ...,
       [ 0.,  0.,  1., ...,  6.,  0.,  0.],
       [ 0.,  0.,  2., ..., 12.,  0.,  0.],
       [ 0.,  0., 10., ..., 12.,  1.,  0.]])
```

これは、digits.data としてもアクセスできる. そこで、楽な方の'.xx'を使って、

```
>>> type(digits.data)
```

で、

```
<class 'numpy.ndarray'>
```

さらに、

```
>>> digits.data.shape
```

で、

```
(1797, 64)
```

と返ってくるから、1797×64 の numpy の行列になっていることがわかる. ここで×64 は画像の大きさで 8×8 をベクトルに並べたものとなっている. そこで、

```
>>> import matplotlib.pyplot as plt
```

で matplotlib をインポートしてから、画像の大きさが 8×8 であることを考慮して、

>>> plt.imshow(digits.data[0].reshape(8,8)) とすれば、
<matplotlib.image.AxesImage object at 0x000001C03D582048>
などと返ってくるから、続いて、
>>> plt.show() とすれば、
0番目のデータの内容を表示して、手書き数字の'0'となっていることが確認できる.
このほか、
digits.target では、その画像データが0～9のうち、どの文字かが与えられる.
digits.images には、上記の digits.data と同じ内容が8×8の画像のまま入っていて、全体では(1797, 8, 8)の形状となっている.

手書き数字データがわかったので、顔の画像データも見ておこう. 手書き数字と同様に
>>> from sklearn import datasets
>>> of = datasets.fetch_olivetti_faces()
とすれば、of に顔の画像データや正解ラベルなどが得られる.
>>> type(of) で、
<class 'sklearn.utils.Bunch'> となっているから、
>>> of.keys()
dict_keys(['data', 'images', 'target', 'DESCR'])
そこで、
>>> of.data.shape
(400, 4096)
>>> of.images.shape
(400, 64, 64)
>>> of.target.shape
(400,)
など確認しておこう. そして、
>>> plt.imshow(of.data[0].reshape(64,64))
<matplotlib.image.AxesImage object at 0x000001C03D7EDB88>
>>> plt.show()
で画像が見えるはずだ.

それでは、データの読み込みと表示を関数にしておこう. 手書き数字データはリスト4.5.1に、顔画像データはリスト4.5.2に、それぞれ示す.

リスト 4.5.1：手書き数字データ

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import datasets

def get_data():
    data    = datasets.load_digits().data
    target = datasets.load_digits().target
    correct = np.eye(10)[target]
    return data, correct, target

def show_sample(data, label=None):
    max_picel = np.max(data); min_picel = np.min(data) # 画素データを 0～1 に補正
    rdata = (data - min_picel)/(max_picel - min_picel)
    rdata = rdata.reshape(8, 8)
    plt.imshow(rdata.tolist(), cmap='gray')
    plt.title(label)
    plt.show()
```

リスト 4.5.1 とリスト 4.5.2 の共通の内容からみていく．冒頭 sklearn の dataset をインポートし、関数 get_data()の中で data と target を読み込む．どちらも、関数 get_data()で画像データとそれに対する正解を返し、関数 show_sample()で、画像の表示を行う．関数 get_data()で返す正解値は one_hot と正解ラベルの 2 通りを用意する．正解ラベルは、各データに含まれる正解の番号を示すラベルそのものだ．いっぽう one_hot というのは、取りうる番号の範囲の正解のところだけ'1'で他は'0'のベクトルに変換したもので、全体ではそれがデータの分だけ並んだ行列になる．one_hot の正解値 correct は、numpy の関数 eye()を使って target から変換して得て、返り値として data、target とともに返す．

いっぽう関数 show_sample()では、画像がもともと白黒であることとともに、機械学習の際にデータの持つ数値の大きさを調整することも考えておくことにする．それというのも、matplotlib で表示する際の各画素の値の大きさには制約がある一方、ニューラルネットワークに入力された各画素の値は重み付け和がとられるから、それが適切な値の範囲に収まることが望ましいからだ．とまれここでは値が 0～1 の範囲になるように調整する．また画像データは表示のために numpy の関数 reshape()を使って縦×横の 2 次元に変換する．表示の際には白黒画像に変な色がつくのは好ましくないから、cmap='gray'を指定する．また表示の際のラベルは引数 label を使うが、そのデフォルトは None なので、何も指定しなければラベルを表示しない．

リスト 4.5.2：顔画像データ

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import datasets

def get_data():
    data = datasets.fetch_olivetti_faces().data
    target = datasets.fetch_olivetti_faces().target
    correct = np.eye(40)[target]
    return data, correct, target

def label_list(): # *は女性
    name = 'Jake', 'Tucker', 'Oliver', 'Cooper', 'Duke', 'Buster', 'Buddy', 'Kate*',¥
           'Sam', 'Lora*', 'Toby', 'Cody', 'Ben', 'Baxter', 'Oscar', 'Rusty', 'Gizmo', ¥
           'Ted', 'Murphy', 'Cooper', 'Bentley', 'Wiston', 'William', 'Alex', 'Aaron', ¥
           'Colin','Daniel','Cooper','Connor','Devin', 'Henry','Sadie*','Ian','James', ¥
           'Gracie*','Jordan','Joseph','Kevin','Kyle','Luke'
    print('名簿を提供します')
    return name

def show_sample(data, label=None):
    max_picel = np.max(data); min_picel = np.min(data) # 画素データを 0～1 に補正
    rdata = (data - min_picel)/(max_picel - min_picel)
    rdata = rdata.reshape(64, 64)
    plt.imshow(rdata.tolist(), cmap='gray')
    plt.title(label)
    plt.show()
```

リスト 4.5.2 の顔画像の方には、もう 1 つ関数 label_list()を用意した．元のデータは各人の誰かを ID で示すが、ID 番号では誰だかピンとこないので、勝手に犬につける名前を借用しておいた．顔画像を提供してくださった方々には大変申し訳ないが、やはり具体的な名前があると分かり易いので、ご容赦いただきたい．

以下、ここで取り上げた手書き数字の例を図 4.5.2 に、顔画像の例を図 4.5.3 に示す.

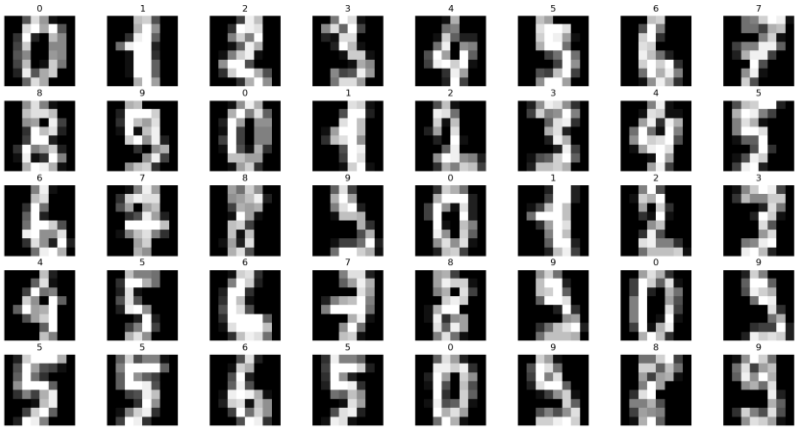

図 4.5.2 sklearn の Digits 手書き文字の例

図 4.5.3 sklearn の OlivettiFaces 顔画像の例

4.5.2 結果確認のための準備

分類問題でサンプルデータを選んで機械の判定を確認するための関数を用意しよう.

リスト 4.5.3：サンプルで確認

```
def test_sample(show, func, x, t, label_list=None):
    '''show:画像表示の関数, func:順伝播の関数,  X:入力, t:正解'''
    print('¥n― テスト開始 ―')
    print('データ範囲内の数字を入力、その他は終了')
    while True:
        try:
            i = int(input('¥n テストしたいデータの番号'))
            sample_data  = x[i]
            sample_id    = t[i]
            sample_label = label_list[sample_id] if label_list is not None else sample_id
        except:
            print('― テスト終了 ―')
            break
        print(' ― 選択されたサンプルを表示します ―')
        print('このサンプルの正解は =>', sample_label)
        if input('サンプルの画像を表示しますか？(y/n)') in ('y', 'Y'):
            show(sample_data, sample_label) # 画像の表示

        # サンプル表示の後にいったん問合せた方が何をやっているか分りやすい
        if input('機械の判定を行いますか？(y/n)') not in ('y', 'Y'):
            continue

        print('― サンプルを機械で判定します ―')
        y = func(sample_data)              # 順伝播(入力の次元保存)
        print('ニューラルネットワークの出力¥n', y)

        estimation = int(np.argmax(y))    # ニューラルネットワークの判定結果
        if label_list is not None:
            estimate_label = label_list[estimation]
        else:
            estimate_label = estimation

        print('機械の判定は          =>', estimate_label)
```

リスト 4.5.3 はインタラクティブにサンプルデータを選んで、それを表示し、ニューラルネットワークにそのデータを入力して、出力を表示するとともに機械の判定を表示する. 判定の表示の際には、カテゴリーにラベルがあれば、番号でなくラベルに置き換える.
やっていることは単純だが手順がだらだらとすこし長い. しかしそうであればますます、関数にしておくのが便利だ. これも common_function.py に入れておこう.

4.5.3 手書き数字の認識

はじめに扱うのはリスト 4.5.1 で用意した手書き数字だ．画像としては 8×8 で小さいが、これを読み取って 0～9 のどの数字かを識別できるようにパーセプトロンを学習させてみよう．

リスト 4.5.4：手書き数字の認識

```
import Neuron, LossFunctions
import Digits
import common_function as cf

# データを用意
x, c, t = Digits.get_data()

# モデルを作る
model = Neuron.NeuronLayer(10)
loss_function = LossFunctions.MeanSquaredError()

# 学習
n_learn = 1001
errors, accuracy = [], []
for i in range(n_learn):
    y = model.forward(x)
    l = loss_function.forward(y, c)
    gy = loss_function.backward()
    model.backward(gy)
    model.update(eta=0.0001)
    acc = cf.get_accuracy(y, c)
    errors.append(l)
    accuracy.append(acc)
    if i%100==0:
        print('{:6d} {:6.3f} {:6.3f}'.format(i, float(l), acc))

# 学習後に経過表示と文字認識のテスト
cf.graph_for_error(errors, accuracy, label=('error','accuracy'), xlabel='n_learn')
cf.test_sample(Digits.show_sample, model.forward, x, t)
```

リスト 4.5.4 に手書き数字の認識のプログラムを示す．データはリスト 4.5.1 の内容を Digits.py に置いたので、その関数 get_data()で、画像データとそれに対する正解を変数 x, c, t に入手する．もちろん x，c，t いずれもしかるべき大きさの行列やベクトルだ．

アヤメの分類に取り組んだリスト 4.4.8 と多くの部分が共通だが、ニューロンの数は 10 個だ．なぜならば、判別する 0～9 のどの数字かの 1 つ 1 つにそれぞれニューロンを割り当てるからだ．すなわち、ニューロン 1 つを 1 つの数字に対応させ、これを並べた 10 個の値が、one_hot の正解値に近づくように学習させるのだ．活性化関数はデフォルトの恒等関数、損失関数は平均二乗誤差だ．

そしていよいよ学習だ．学習回数は n_learn = 1001 で指定した．途中経過の記録のために errors と accuracy を空のリストで用意し、for ループで回して学習していく．その際に common_function に用意したリスト 4.4.5 の関数 get_accuracy()を使って、正解率 acc を求めて誤差 l と併せて記録する．また誤差と正解率は i%100 == 0 で 100 回ごとに画面に出力する．
学習後の確認は、common_function に用意した関数 graph_for_error()で学習経過をグラフ表示し、同じく common_function に用意した関数 test_sample()でインタラクティブに確認する．
図 4.5.4 には学習経過のグラフを示すが、順調に誤差が減ると同時に正解率が上がる様子がわかる．学習率は eta=0.0001 にしているが、これを指定しないとすぐにオーバーフローしてしまうようだ．
学習がうまくいったことを確認したら、common_function.py に作ったリスト 4.5.3 の関数 test_sample()を使って、インタラクティブに結果を確認しよう．

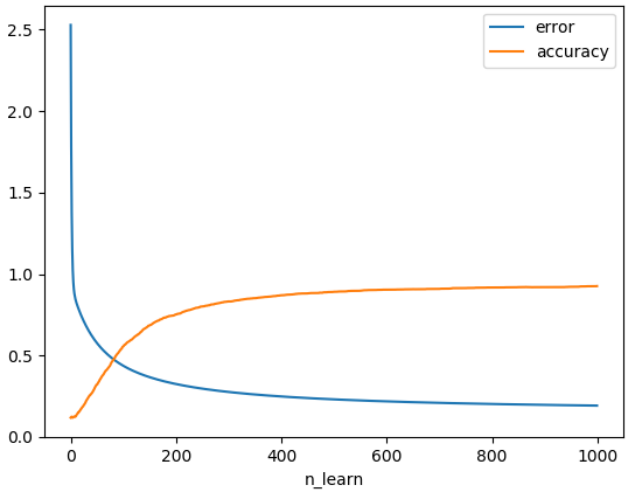

図 4.5.4 手書き数字の認識の学習経過

4.5.4 顔認識

もう少し画像らしい画像として図 4.5.3 に例を示す顔画像を認識させてみよう. こちらは画像サイズが 64×64 と少し大きく、また、40 人の顔画像なので 40 通りに分類する.

リスト 4.5.5：顔認識

```
import Neuron, LossFunctions
import OlivettiFaces as OF
import common_function as cf

# データを用意
x, c, t = OF.get_data()
label_list = OF.label_list()

# モデルを作る
model = Neuron.NeuronLayer(40)
loss_function = LossFunctions.MeanSquaredError()

# 学習
n_learn = 2001
errors, accuracy = [], []
for i in range(n_learn):
    y = model.forward(x)
    l = loss_function.forward(y, c)
    gy = loss_function.backward()
    model.backward(gy)
    model.update(eta=0.001)
    acc = cf.get_accuracy(y, c)
    errors.append(l)
    accuracy.append(acc)
    if i%100==0:
        print('{:6d} {:6.3f} {:6.3f}'.format(i, float(l), acc))

# 学習後に経過表示と文字認識のテスト
cf.graph_for_error(errors, accuracy, label=('error','accuracy'), xlabel='n_learn')
cf.test_sample(OF.show_sample, model.forward, x, t, label_list)
```

リスト 4.5.5 に顔認識のプログラムを示す. リスト 4.5.4 とほとんど同じだが、40 通りの分類だからニューロンは 40 個だ. またリスト 4.5.2 に示すように名前を用意したので、これを関数 label_list()で読み込んで、関数 test_sample()に渡すようにした.

学習率は eta=0.001 としたがリスト 4.5.4 の手書き文字に比べて大きな値にしないと学習が遅い. しかし大きすぎるとやはりオーバーフローしてしまう. とまれ eta=0.001 ではむしろ急激に誤差が減り、そして順調に正解率が上がって 1 に近づくようすが観測される. これも最後に common_function.py の関数 test_sample()で顔認識がうまくいっていることを確認しよう.

学習経過は図 4.5.5 に示す.

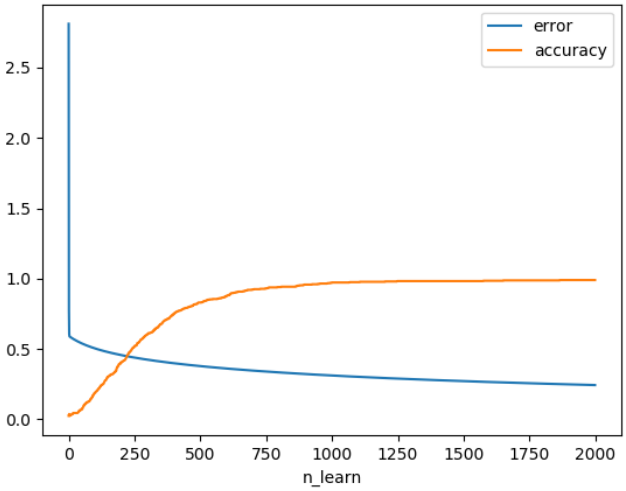

図 4.5.5 顔認識の学習経過

4.6 多層化と活性化関数

ニューロン1層だけのパーセプトロンで画像認識まで扱えてしまうことに、むしろ戸惑いすら感じる．それというのも単なる入力の重み付け和によって画像認識がなされているからだ．そして排他的論理和の論理演算を例外として、ここまで扱ってきた例では回帰でも分類でも、パーセプトロンの能力がその例外によって制約されることを見てとることは難しい．むしろ非線形性を含まないが故に容易に学習し的確ですらある．しかしそれをかなぐり捨ててなお、これを超えた先に、より高度な、文字通り高次元の能力が秘められている．

4.6.1 多層化とニューラルネットワーク

パーセプトロンは、図4.2.2や図4.2.4に示すように、1つまたは複数のニューロンで構成される．しかしいずれの場合も、ニューロンに外から入力が入ると、そのニューロンの出力がそのままパーセプトロンの出力となっていた．これに対して、外から入力がつながるニューロンの出力をそのまま取り出すのではなく、また別のニューロンに入れて、そこで処理を行ってから外に取り出すことが考えられる．これを図にすると図4.6.1のようになる．そして全体がネットワーク状につながっているから、これをニューラルネットワークと呼ぶ．図4.6.1では入力も1つの「層」と考えて、ニューロン層の2層と併せて全部で3層としている．入力層から中間層の各ニューロンに信号が伝わり、その中間層の各ニューロンの出力が出力層の各ニューロンの入力となって、出力層の各ニューロンの出力がニューラルネットワークの出力となっている．図4.6.1に示すのはごく単純なニューラルネットワークだが、中間層のニューロンの入出力はいずれも外には直接つながっておらず、ニューロンの数も含めて、ニューラルネットワークの外からは観測されないので、これを「かくれ層」と呼ぶことがある．かくれ層は1つとは限らず、複数にすることも考えられる．すなわち、入力層からはじめのかくれ層に信号が入り、そのかくれ層の出力がまた次のかくれ層に入り、というように繰り返し、そして最後に出力層に入ってニューラルネットワークの出力になるといったように．この場合にはかくれ層が何層あるのかさえ、外からは直接観測されない．

[補足]

図4.6.1で、入力層としている層は図にニューロンを示す〇が示されるが、実際には何もしないで通過して中間層に信号を伝えるだけである．何もしないのに層とするのはどうかと思うが、これを層として数えるのが通例となっている．大事なことは、その入力のそれぞれに重みがかけられて中間層のニューロンに伝えられることであり、図でいうならば矢印の部分ーすなわち層と層の間の部分こそ、機能の中心という見方ができる．実際、モデルの元とな

った神経細胞で言うならば、軸索端末からシナプスを介して樹状突起そして細胞体に至る経路そのものが、神経のネットワークとしてのはたらきの主役である．それをふまえれば、何もしないのに入力層という捉え方はわからなくもない．

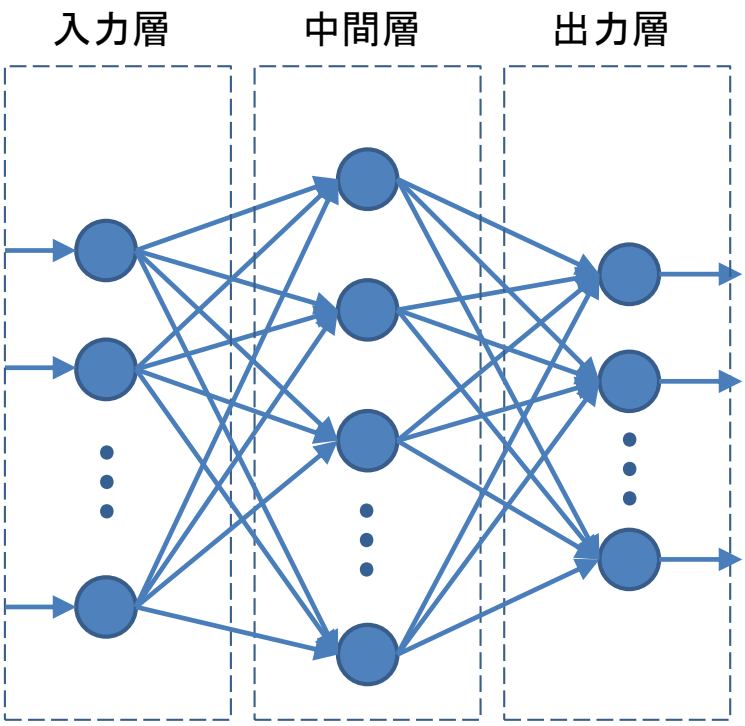

図 4.6.1 ニューラルネットワーク

ニューロン層が 1 つだけのパーセプトロンについて順伝播と逆伝播をすでに説明したが、ニューラルネットワークでも、入力から出力に信号が順に伝わっていく動作を順伝播といい、その逆に出力に与えられた勾配が層を逆に辿りながら入力に向かって伝えられていく動作を逆伝播という．

ニューロン層の順伝播は式 4.1.5 ないし式 4.1.6 で示され、ニューロン層が 1 つだけのパーセプトロンの順伝播はこれがそのまま当てはまるが、ニューラルネットワークでは「各層において」同式が当てはまる．たとえば図 4.6.1 のように 3 層のニューラルネットワークならば、入力層から中間層に信号が入る際に中間層の重みやバイアスによって、式 4.1.5 ないし

式 4.1.6 に示す積和の操作がなされ、このとき同式に示すように中間層の活性化関数にしたがって中間層の出力が決定される．そしてその中間層の出力が出力層の入力となって、出力層の重みやバイアスと活性化関数により式 4.1.5 ないし式 4.1.6 にしたがって出力層の出力、すなわちニューラルネットワークの出力が決定される．このとき入力から出力の変換の式は共通だが、中間層は中間層の、出力層は出力層の、それぞれ重み、バイアス、活性化関数が独立に与えられ、適切な活性化関数を選べば、ニューロン層が 1 つだけのパーセプトロンでは表すことができなかった入力と出力の複雑な関係をあらわすことができる．とまれこのようにニューラルネットワークの順伝播では、入力から出力に至るまで式 4.1.5 ないし式 4.1.6 に従う変換が繰り返し層ごとに順に行われていく．

つづいて逆伝播を考えていこう．ニューロン層の逆伝播は損失関数を起点として、式 4.3.25～4.3.28 でそれぞれ、中間値の勾配、重みの勾配、バイアスの勾配、入力の勾配が求められる．ニューロン層が 1 つだけのパーセプトロンではこれがそのまま当てはまり、逆伝播の目的の、重みとバイアスの勾配を求めて、パーセプトロンを学習させた．では図 4.6.1 に示すような、ニューロン層が 2 層、あるいは、それ以上あるような場合はどうだろうか？ ニューラルネットワークの出力層については、式 4.3.25～4.3.28 がそのまま当てはまることは自明だろう．なぜなら図 4.3.1 と図 4.6.2 を見比べればわかるように、ニューラルネットワークの出力層と損失関数の関係は、パーセプトロンと損失関数の関係と同じだからだ．したがって、ニューラルネットワークの出力層の中間値の勾配が式 4.3.25 となり、出力層の重みとバイアスの勾配 $\frac{\partial L}{\partial \boldsymbol{W}}$ と $\frac{\partial L}{\partial \boldsymbol{B}}$ が式 4.3.26 と式 4.3.27 で求められる．このとき式 4.3.28 では、出力層の入力の勾配 $\frac{\partial L}{\partial \boldsymbol{X}}$ が得られる．出力層の入力は、その上流の中間層の出力にほかならない．そこで次は、出力層の入力の勾配 $\frac{\partial L}{\partial \boldsymbol{X}}$ を中間層の出力の勾配 $\frac{\partial L}{\partial \boldsymbol{Y}}$ として中間層での勾配の計算の起点とすれば、中間層の中間値の勾配 $\delta = \frac{\partial L}{\partial \boldsymbol{Y}} \frac{\partial \boldsymbol{Y}}{\partial \boldsymbol{U}}$ が得られる．

それから中間層の重みとバイアスの勾配 $\frac{\partial L}{\partial \boldsymbol{W}}$ と $\frac{\partial L}{\partial \boldsymbol{B}}$ をふたたび式 4.3.26 と式 4.3.27 で求める．図 4.6.1 の 3 層のニューラルネットワークではこれで終わりだ．しかし層の数がもっと多い場合には、これを同じように繰り返せば、あたかも勾配を下流から上流へと伝えながら遡るようにして各層の重みやバイアスの勾配を求めることが出来る．このように勾配を下流から上流へと遡るように伝えていくことは逆伝播にほかならないが、ニューロン層が 1 つ

だけのパーセプトロンの場合よりも、逆伝播という言葉がしっくりくるのではないだろうか？

このように損失関数を起点として、逆伝播によって勾配を伝えていけば、ニューラルネットワークの出力を損失関数で評価した損失あるいは誤差を小さくするために、各層の重みやバイアスをどう調節すればよいかを知ることが出来る．すなわち、下流側の入力の勾配を求めれば、それはその上流側の出力の勾配であり、出力の勾配が分かれば、その層の重みとバイアスの勾配が求められる、ということが繰り返し行われるだけであって、ニューロン層を複数重ねて多層化していく道筋を容易に描くことができる．

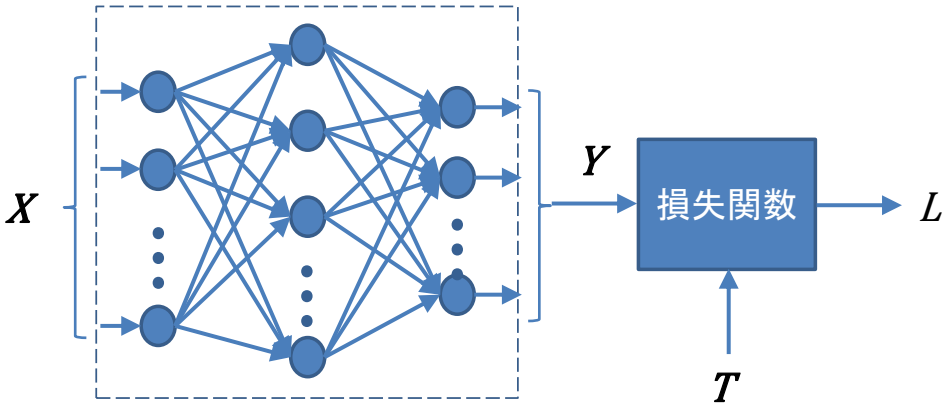

図 4.6.2 ニューラルネットワークと損失関数

図 4.6.2 に入力 X、出力 Y、正解値 T、損失 Lとしたときのニューラルネットワークと損失関数の関係を簡単に図示するが、図 4.3.1 のパーセプトロンをニューラルネットワークに置き換えただけだ．

4.6.2 ニューラルネットワークの実装

多層化したニューラルネットワークを容易に使えるように実装していこう．もちろんその基本となる部品はすでに作ってきたニューロン層だ．そしてニューロン層の実装同様に、基底クラスと派生クラスに分けて作っていく．基底クラスはクラス NN_CNN_Base として、そのコントラクタから作ろう．

リスト 4.6.1：ニューラルネットワークの基底クラスのコントラクタ

```
import LossFunctions
import common_function as cf

class NN_CNN_Base:
    def __init__(self, *args, **kwargs):
        print(args, kwargs)
        # 入出力の判定
        if len(args) == 2:
            In, Out = args
        elif len(args) == 0:
            In = None; Out = None
        else:
            In = None; Out = args[-1]
        print(args, 'number of output =', Out)
        self.layers = []

        # 損失関数
        loss_function_name = kwargs.pop('loss', 'MeanSquaredError')
        self.loss_function = cf.eval_in_module(loss_function_name, LossFunctions)

        return In, Out
```

リスト 4.6.1 に NN_CNN_Base のコントラクタを示す．可変長引数 *args，**kwargs を受け取るが、*args には入出力の信号数が指定されるものとして、簡単な識別を行っている．リスト 4.1.4 の NeuronLayer のところで説明したように、データから入力の信号数を決める場合もあるし、両方とも指定される場合も、どちらも指定されない場合もあるものとした．
self.layers は空のリストで初期化しておくが、これは NN_CNN_Base を継承する派生クラスで層構成に応じて情報を入れる．キーワード'loss'で kwargs により指定する損失関数を組み込む．組み込む方法はリスト 4.1.1 で活性化関数を組み込んだのと同じで、リスト 4.1.3 で作った関数 eval_in_module() を使う．損失関数のデフォルトはリスト 4.3.1 で作った平均二乗誤差 MeanSquaredError で、ここまでの範囲ではこれが唯一だが、LossFunctions.py に組み込めば、別の損失関数も指定できるようにしている．あとは、入出力数を返り値として終わる．

つづいて順伝播のメソッドを作ろう.

リスト 4.6.2：ニューラルネットワークの順伝播

```
class NN_CNN_Base:

    （コントラクタはリスト 4.6.1 に掲載）

    def forward(self, x, t=None):
        y = x
        for layer in self.layers:
            y = layer.forward(y)
        if t is None:
            return y
        elif hasattr(self, 'loss_function'):
            l = self.loss_function.forward(y, t)
            return y, l
        else:
            raise Exception("Can't get loss by forward.")
```

まず入力 x からの流れを見ていこう．派生クラスによって self.layers には層が上流から下流へと順にまとめられているものとして、そこから層を取り出しては 1 つ前の出力を入力として layer―すなわちニューロン層の forward メソッドを実行していく．そして最後の layer の出力はニューラルネットワークの出口なので返り値となる.

forward メソッドの引数に入力 x のほかに t=None がある．このメソッドを呼出す際に t に何も指定しなければ、if 文で t is None が成立して、ニューロン層を順伝播した結果の y を返り値として終わる．いっぽう、t に何か値を設定してこのメソッドを呼んだ場合には、損失関数が定義されていることを hasattr(self, 'loss_function') で確かめて、順伝播の結果の y と引数で与えられた正解値 t を損失関数に与えて、損失関数の forward メソッドを実行する．そして得られた損失あるいは誤差の値を、順伝播の結果の y とともに返り値とする．最後にもしも、t に値が与えられたのに損失関数が未定義の場合には例外を引き起こすようにしている.

次は逆伝播のメソッドだ.

リスト 4.6.3：ニューラルネットワークの逆伝播

```
class NN_CNN_Base:

    （コントラクタはリスト 4.6.1 に、forward メソッドはリスト 4.6.2 に掲載）

    def backward(self, gy=None, gl=1):
        if gy is not None:
            pass
        elif self.loss_function.t is not None:
            gy = self.loss_function.backward(gl)
        else:
            raise Exception("Can't get gradient for backward.", 'gy =', gy, 'gl =', gl)
        gx = gy
        for layer in reversed(self.layers):
            gx = layer.backward(gx)
        return gx
```

逆伝播は損失関数から辿る. その場合には引数は不要で、gl=1 はリスト 4.4.1 で説明した通りだ. しかし外から勾配を与えることもできるようにデフォルト引数 gy=None としている. 外から勾配を与える場合には、それを起点にするから if 文が成立して pass で何もせずに、elif、else を全部スキップする. いっぽう外から勾配を与えない場合には、loss_function.t が None でないことをチェックしたうえで、損失関数の逆伝播を行い、ニューラルネットワーク本体の出力の勾配 gy を得る. チェック対象の loss_function.t には損失関数の forward メソッドで正解値を保存する. だからそれが None だったら、まだ損失関数の forward メソッドを実行していないので例外にしている. とまれいずれかにより起点の gy が決まると、あとは self.layers から逆順に層を取り出して、ニューロン層を勾配を伝えながら逆順に辿る. 返り値は最後に得られた gx だが、これは最上流のニューロン層の入力の勾配となる.

更新のメソッドも必要だ.

リスト 4.6.4：ニューラルネットワークの更新

```
class NN_CNN_Base:

    （コントラクタ、forward、backward はリスト 4.6.1～リスト 4.6.3 に掲載）

    def update(self, **kwargs):
        for layer in self.layers:
            if hasattr(layer, 'update'):
                layer.update(**kwargs)
```

更新は self.layers から各層を取り出して順に更新すればよい. これまでのところでは、更新しない層はないが、将来に備えて更新を伴う層かどうかの判断を行ったうえで更新する.

NN_CNN_Base の最後に構成を一括して確認する手段として、メソッド summary()も作っておこう.

リスト 4.6.5：ニューラルネットワークの構成を確認

```
class NN_CNN_Base:

   (コントラクタ、forward、backward、update の各メソッドはリスト 4.6.1～リスト 4.6.4 に掲載)

    def summary(self):
        print('～～ model summary of', self.__class__.__name__, '～'*24)
        for i, layer in enumerate(self.layers):
            print('layer', i, layer.__class__.__name__)
            print(' configuration =', layer.config, end='')
            if hasattr(layer, 'activator'):
                print('¥n activate =', layer.activator.__class__.__name__, end=' ')
            print('¥n' + '-'*72)
        if hasattr(self, 'loss_function'):
            print('loss_function =', self.loss_function.__class__.__name__)
        print('～～ end of summary ' + '～'*28 + '¥n')
```

自分自身、層、活性化関数、損失関数のクラス名は __class__.__name__ で得る. そして、config の入力数、ニューロン数と併せて表示する. 表示に際して、活性化関数や損失関数を伴わないニューロン層やニューラルネットワークでもエラーしないように、hasattr を使って該当する属性の有無をチェックしている.

あれこれ作ってきたが、NN_CNN_Base を検証する意味もあるから、継承する派生クラスとして、まずはこれまで動かしてきたニューロン 1 層だけのパーセプトロンを定義しておこう.

リスト 4.6.6：NN_0

```
import Neuron as neuron

class NN_0(NN_CNN_Base):
    def __init__(self, *args, **kwargs):
        In, Out = super().__init__(*args, **kwargs)
        self.layer = neuron.NeuronLayer(Out, **kwargs)
        self.layers.append(self.layer)
```

基底クラス NN_CNN_Base を定義したことで、その派生クラスとしてはコントラクタで構成を指定するだけで良い.

In, Out = super().__init__(*args, **kwargs)

によって、基底クラスのコントラクタを呼び出して入出力数を受け取る. そして、

self.layer = neuron.NeuronLayer(Out, **kwargs)

でニューロン層のインスタンスを生成し、それを、

self.layers.append(self.layer)

でニューラルネットワーク全体の層の情報 self.layers に入れる.

機能的に新しいものは何もないが、確認のためにリスト 4.5.5 の顔認識を NN_0 で動かしてみよう.

リスト 4.6.7：NN_0 で顔認識

```
import NN
import OlivettiFaces as OF
import common_function as cf

# データを用意
x, c, t = OF.get_data()
label_list = OF.label_list()

# モデルを作る
model = NN.NN_0(40)
model.summary()

# 学習
n_learn = 2001
errors, accuracy = [], []
for i in range(n_learn):
    y, l = model.forward(x, c)
    model.backward()
    model.update(eta=0.001)
    acc = cf.get_accuracy(y, c)
    errors.append(l)
    accuracy.append(acc)
    if i%100==0:
        print('{:6d} {:6.3f} {:6.3f}'.format(i, float(l), acc))

# 学習後に経過表示と文字認識のテスト
cf.graph_for_error(errors, accuracy, label=('error','accuracy'), xlabel='n_learn')
cf.test_sample(OF.show_sample, model.forward, x, t, label_list)
```

損失関数も組み込んであるから少し簡単になること以外は、リスト 4.5.5 と変わらない.
関数 summary()を実行しているから、モデルを作ったところで次のような情報が出力される.

```
~~ model summary of NN_0 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
layer 0 NeuronLayer
 configuration = (None, 40)
 activate = Identity
----------------------------------------------------------------
loss_function = MeanSquaredError
~~ end of summary ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

途中経過も、関数 test_sample()でのインタラクティブなテストも含めて、リスト 4.5.5 と同じ結果になるはずだ. ここまで動けば、リスト 4.6.1～リスト 4.6.6 の作りこみができていることも確認できたことになる.

4.6.3 シグモイド関数

これまで活性化関数としては恒等関数とステップ関数を定義したが、ニューラルネットワークの活性化関数としてとてもポピュラーなシグモイド関数を定義しよう．シグモイド関数の入出力の関係をあらわす式を示すと、

$$y = \frac{1}{1 + e^{-x}} \tag{4.6.1}$$

となっている．これがどんな特性かはちょっと置いておいて、式 4.6.1 であらわされるシグモイド関数の微分を求めよう．式 4.6.1 で $z = 1 + e^{-x}$ とおけば $y = z^{-1}$ だから、

$$\begin{aligned}
\frac{dy}{dx} = \frac{dy}{dz}\frac{dz}{dx} &= \frac{d(z^{-1})}{dz}\frac{d(1 + e^{-x})}{dx} \\
&= -z^{-2}(-e^{-x}) \\
&= (1 + e^{-x})^{-2}e^{-x} \\
&= (1 + e^{-x})^{-2}(1 + e^{-x} - 1) \\
&= (1 + e^{-x})^{-1}\{1 - (1 + e^{-x})^{-1}\} \\
&= y(1 - y)
\end{aligned} \tag{4.6.2}$$

式が分かったのでさっそくこれを活性化関数の 1 つとして、Activators.py に加えていこう．

リスト 4.6.8：シグモイド関数

```
class Sigmoid:
    def forward(self, x):
        y = 1 / (1 + np.exp(-x))
        self.y = y
        return y

    def backward(self, gy):
        y = self.y
        gx = y * (1 - y) * gy
        return gx
```

リスト 4.6.8 に活性化関数としてのシグモイド関数を示す．順伝播の forward メソッドは式 4.6.1 を、逆伝播の backward メソッドは式 4.6.2 を、そのままインプリメントしたものになっている．ただし forward メソッドでは算出した y をインスタンス変数 self.y に保存し、

backward メソッドで使う．また backward メソッドは引数に下流からの勾配を引数 gy として受け取り、式 4.6.2 の $\frac{dy}{dx}$ とこれとの積を返り値とする．

活性化関数としてニューラルネットワークで使う前に特性を見ておこう．これまでに Activators.py に定義したステップ関数とシグモイド関数の 2 つを並べて視覚化しよう．

リスト 4.6.9：ステップ関数とシグモイド関数の視覚化

```
import numpy as np
import matplotlib.pyplot as plt
import Activators

x = np.linspace(-5, 5, 1000)
y1 = Activators.Step().forward(x)
y2 = Activators.Sigmoid().forward(x)

plt.plot(x, y1)
plt.plot(x, y2)
plt.show()
```

リスト 4.6.9 の実行結果を図 4.6.3 に示す．

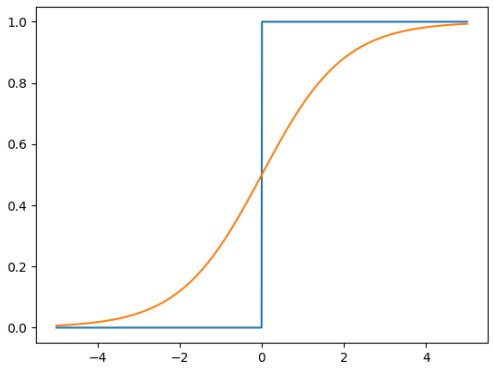

図 4.6.3 ステップ関数とシグモイド関数の応答

このようにシグモイド関数は、閾値 0 のステップ関数を閾値付近で滑らかに変化するようにした形をしている．そして先に式 4.6.2 で求めたように微分可能だからリスト 4.6.8 に示したように逆伝播を定義することができるのだ．シグモイド関数は生物の神経細胞が持つ性質をモデル化したものといわれるが、そのことはさておいても、ニューラルネットワークで使うのに都合が良い．次にリスト 4.6.10 で、シグモイド関数の逆伝播を順伝播と併せて視覚化して

おこう.

リスト 4.6.10：シグモイド関数の順伝播と逆伝播

```
import numpy as np
import matplotlib.pyplot as plt
import Activators

x = np.linspace(-5, 5, 1000)
func = Activators.Sigmoid()

y = func.forward(x)
dydx = func.backward(gy=1)

plt.plot(x, y)
plt.plot(x, dydx)
plt.show()
```

リスト 4.6.10 では活性化関数の backward メソッドを呼出す際に gy=1 を指定する．ニューラルネットワークの活性化関数としてシグモイド関数を使うときは、下流側からの勾配を gy に与えるが、ここではシグモイド関数単体の逆伝播の特性を見たいから、こうしている．とまれこの実行結果を図 4.6.4 に示す．

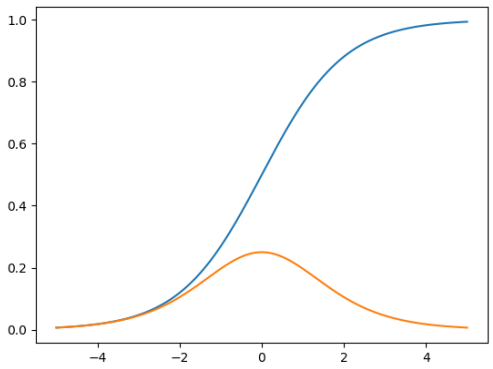

図 4.6.4 シグモイド関数の順伝播と逆伝播

図 4.6.4 は $x = -5 \sim 5$ の範囲で式 4.6.1 の値と式 4.6.2 の値を描いた．逆伝播の特性を見ると、シグモイド関数は $x = 0$ の付近で程よい勾配だが、0 から遠くなると勾配が小さくなり、ほとんど 0 になることがわかる．勾配が 0 になるということは、どちらに行けばよいか分からなくなって学習が進まなくなることを意味し、深いニューラルネットワークでしばしば問題となる．それを避ける様々な活性化関数が考案されている．

ここで念のため、シグモイド関数が非線形であることを確認しておこう．リスト 4.4.10 でパーセプトロンが線形であることを確かめた．同じ要領でシグモイド関数を見てみよう．

リスト 4.6.11：シグモイド関数が非線形であることの確認

```
import numpy as np
import Activators

m, n, k = 4, 3, 5  # 入出力の大きさ
a = 2              # 斉次性確認の係数

x = np.random.rand(k, m)
y = np.random.rand(k, m)

f = Activators.Sigmoid().forward      # 対象の設定

print(np.abs(f(x)+f(y)-f(x+y))<1e-7) # 加法性
print(np.abs(a*f(x)-f(a*x))<1e-7)    # 斉次性
```

リスト 4.6.11 は Activators.py に定義したシグモイド関数を

f = Activators.Sigmoid().forward

で対象となる関数に設定している．ここを

f = Neuron.NeuronLayer(m, n, activate='Sigmoid').forward

とすれば書き換えれば、活性化関数をシグモイド関数としたパーセプトロンが対象になる．ただし冒頭に、import Neuron を追記するのを忘れずに．とまれシグモイド関数は非線形なので当然ながら False が並んだ行列が出力される．

4.6.4 ニューラルネットワークの新たな派生クラス

シグモイド関数の威力を確認していくことにしよう. そのための準備として、ニューラルネットワークに新たな派生クラスを組み込んで、ニューラルネットワークを簡便に構成できるようにしておこう.

リスト 4.6.12：NN_m のコントラクタ

```
class NN_m(NN_CNN_Base):
    def __init__(self, *args, **kwargs):
        In, Out = super().__init__(*args, **kwargs)
        ml_nn = kwargs.pop('ml_nn', 3)
        opt_for_ml = {}
        opt_for_ol = {}
        opt_for_ml['activate'] = kwargs.pop('ml_act', 'Identity')
        opt_for_ol['activate'] = kwargs.pop('ol_act', 'Identity')
        opt_for_ml.update(kwargs)
        opt_for_ol.update(kwargs)

        # ― 各層の初期化 ―
        self.middle_layer  = neuron.NeuronLayer(ml_nn, **opt_for_ml)
        self.output_layer  = neuron.NeuronLayer(Out, **opt_for_ol)

        # ― layer のまとめ ―
        self.layers.append(self.middle_layer)
        self.layers.append(self.output_layer)
```

NN_m のコントラクタをリスト 4.6.12 に示す. これはリスト 4.6.1～リスト 4.6.6 と同じくファイル NN.py に記述する. NN_m は図 4.6.1 ないし図 4.6.2 に示す 3 層のニューラルネットワークだ. 中間層のニューロン数は場合によって適切な大きさが違うので、
ml_nn = kwargs.pop('ml_nn', 3) によってデフォルトを 3 として、キーワード'ml_nn'で指定できるようにした. また、中間層と出力層の活性化関数は独立に指定できるようにしたい. そこで opt_for_ml と opt_for_ol を辞書型の変数として用意し、引数 kwargs から、'ml_act'と'ol_act'のキーワードで活性化関数の指定を受け取って、それぞれニューロン層に渡す. デフォルトの活性化関数は恒等関数とした. あとは各層を middle_layer、output_layer として初期化して、self.layers にまとめておく. このとき middle_layer、output_layer という名前は必須ではないが、あると便利なので敢えてインスタンス変数にしている.

ではさっそくパーセプトロンで解けなかった問題を解いてみよう．リスト 4.6.13 に、NN_m を用いて排他的論理和 XOR を学習するプログラムを示す．これはリスト 4.4.9 のパーセプトロンの学習とほとんど同じだ．いくつか指定が必要ではあるものの、あっさりと XOR を学習することが確認できるはずだ．

リスト 4.6.13：XOR の学習

```
import numpy as np
import NN
import common_function as cf

# 教師データの用意
x = np.array([[0, 0],[0, 1],[1, 0],[1, 1]])
t = np.array([[0],[1],[1],[0]])

# モデルを作る
model = NN.NN_m(1, ml_nn=10, ml_act='Sigmoid', width=2.0)
model.summary()

# 学習
n_learn = 1001
errors = []
for i in range(n_learn):
    y, l = model.forward(x, t)
    model.backward()
    model.update()
    errors.append(l)

    if i % 100==0:
        print('{:6d} {:6.3f}'.format(i, float(l)))

# 学習後に経過表示
cf.graph_for_error(errors, label='error', xlabel='n_learn')

# 学習済での順伝播と結果
print('input¥n', x)
y = model.forward(x)
z = y > 0.5
print('result¥n', z)
```

データは、x = np.array([[0, 0],[0, 1],[1, 0],[1, 1]])に対し、t = np.array([[0],[1],[1],[0]])で、排他的論理和にしている．もちろんこれ以外の論理演算も扱える．モデルは次行で、

model = NN.NN_m(1, ml_nn=10, ml_act='Sigmoid', width=2.0)

NN_m を使い、出力は 1 つで、中間層の活性化関数にシグモイド関数を指定する．このほかに中間層のニューロン数、重みの初期値の広がりも指定し、試行錯誤の結果うまくいった値にしているが、これらはいろいろ変えて試してみると良い．そして中間層の活性化関数の指定

ml_act='Sigmoid' をやめてしまうと元の木阿弥で排他的論理和ができなくなることがわかる.

次に回帰問題もやっておこう. ここでは直線でなぞれないケースのデータを用意しよう.

リスト 4.6.14：インフルエンザの流行

```
data = [[ 1, 0.26],
        [ 2, 0.25],
        [ 3, 0.19],
        [ 4, 0.34],
        [ 5, 0.89],
        [ 6, 1.78],
        [ 7, 2.51],
        [ 8, 4.81],
        [ 9, 10.97],
        [10, 18.02],
        [11, 19.28],
        [12, 32.42],
        [13, 62.12],
        [14, 62.18],
        [15, 62.6 ],
        [16, 49.18],
        [17, 32.57],
        [18, 25.93],
        [19, 21.17],
        [20, 14.68]]
```

リスト 4.6.14 は、第 3 章 線形回帰でも使ったもので、経過週と定点あたりの報告数の対になったインフルエンザの流行に関する 20 個のデータだ. data_flu.py と名前をつけておこう.

ではインフルエンザの流行がうまくなぞれるかどうかを試してみよう.

リスト 4.6.15：NN_m で回帰

```
import numpy as np
import matplotlib.pyplot as plt
import NN
import data_flu
import common_function as cf

# データを用意
data = np.array(data_flu.data)
x = data[:, :1]     # 入力
t = data[:, 1:]     # 正解値

# モデルを作る
model = NN.NN_m(1, ml_nn=100, ml_act='Sigmoid')
model.summary()

# 学習
n_learn = 100001
errors = []
for i in range(n_learn):
    y, l = model.forward(x, t)
    model.backward()
    model.update()
    errors.append(l)
    if i % 1000==0:
        print('{:6d} {:6.3f}'.format(i, float(l)))

# 経過表示
cf.graph_for_error(errors)

# 学習結果のグラフ表示
plt_x = np.linspace(min(x)-0.5, max(x)+0.5, 1000)
plt_y = model.forward(plt_x.reshape(-1, 1))
plt.scatter(x, t, marker='x', s=100)          # 入力
plt.scatter(plt_x, plt_y, marker='.', s=1)    # 近似
plt.show()
```

リスト 4.6.15 に NN_m で回帰を行うプログラムを示す．これはリスト 4.4.7 のパーセプトロンでの回帰とだいたい同じ流れだ.

data_flu.py に格納したリスト 4.6.14 のデータを numpy の行列にして、入力と正解値を縦ベクトルで切り出す．いっぽうモデルは、

model = NN.NN_m(1, ml_nn=100, ml_act='Sigmoid')

で NN_m を中間層のニューロン数、中間層の活性化関数をシグモイド関数にして生成する．学習回数は n_learn = 100001 とかなり多くし、進捗の表示も 1000 回に 1 回にしたが、学習の流

れはこれまでと変わらない．学習結果のグラフ表示では、入力 x を±0.5 で外挿するデータを numpy の linspace()で ply_x として 1000 個用意し、学習済みのモデルで順伝播して plt_y を得る．そしてこれを入出力データと併せてグラフ表示する．

さっそく学習経過の一例を図 4.6.5 に、また学習結果の一例を図 4.6.6 に示す．

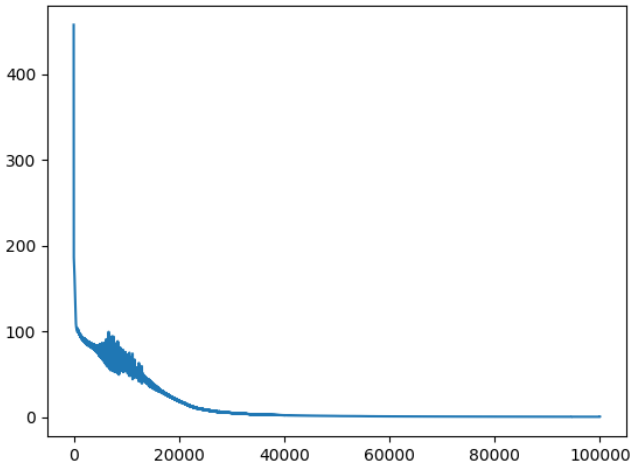

図 4.6.5 NN_m で回帰～学習経過

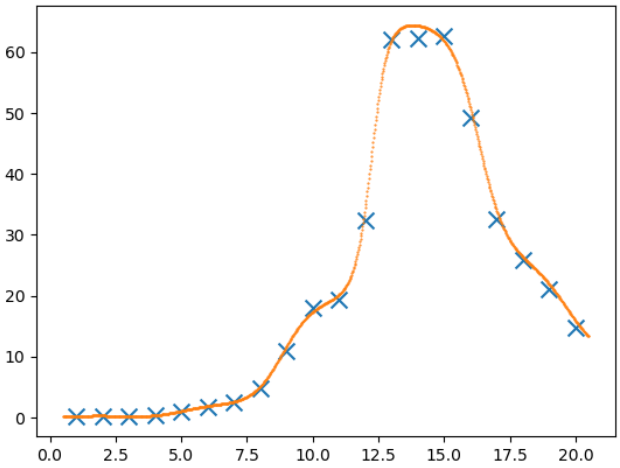

図 4.6.6 NN_m で回帰～インフルエンザの流行

図 4.6.5 の学習経過を見ると、学習回数が 10000 回くらいの時に少し暴れている. これは学習率を少し小さくすると状況が変わる. しかしそうすると図 4.6.6 の学習結果の方も影響がある. データの数が少ない中で、何が良いかは一概には言えない. いずれにせよシグモイド関数の威力を確認するには十分な結果といえるだろう.

ここでも中間層のニューロン数 ml_nn、重みやバイアスの広がり width、学習率 eta など、いろいろ変えてみてほしい.

試みにリスト 4.6.15 で中間層の活性化関数の指定 ml_act='Sigmoid'をやめた場合には、どうなるだろうか？この場合の結果を図 4.6.7 に示す. ただしこの場合には、更新の際に model.update(eta=0.001)のように小さめの学習率を指定しないとオーバーフローしてしまうようだ.

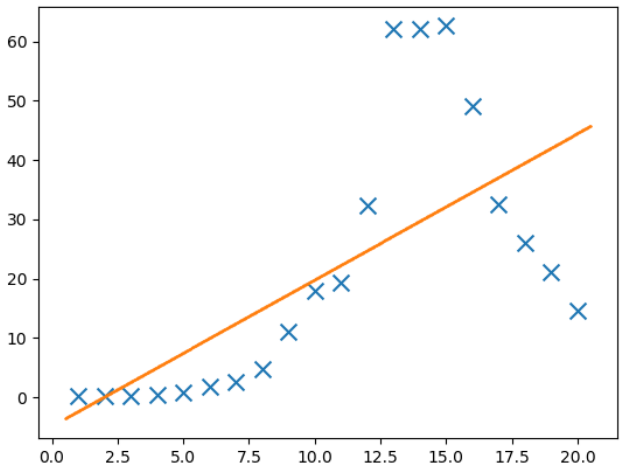

図 4.6.7 恒等関数で回帰～インフルエンザの流行～

ニューロン層を積上げても活性化関数が恒等関数であるとニューラルネットワークは直線しか描けないことが確かめられた. これでは多層化の意味はない. いっぽう多層化と併せてシグモイド関数を使うことによって、ニューラルネットワークは大きな表現力を獲得することがわかった.

4.7 データと学習

学習によって適切なパラメータを獲得し、ニューラルネットワークは論理演算や回帰や分類など様々な問題を扱うことができることを見てきた．この学習の過程は、どのような問題を扱うにしても、データを用いて結果を出し、損失を求めて、それを小さくすべくパラメータを調節する．それによって、乱数や’0’でパラメータを初期化したばかりのときには、意味のある結果は出せなかったニューラルネットワークが、狙った通りの動作をするようになった．これはいうならばモデルが「データから学ぶ」ということにほかならないだろう．このことについて少し掘り下げて、ニューラルネットワークの可能性を考えていこう．

4.7.1 学習と評価

ニューラルネットワークはデータから学ぶのだが、これまでこの学習に使うデータと、進捗を評価する際に使うデータ、さらに学習が完了した後で、ニューラルネットワークの動作を確認する際に使うデータは、区別してこなかった．

もちろん何らかのデータによって学習させて、ニューラルネットワークの入力と出力の関係がなにがしかの対応関係をもつようにし、その意図したとおりであることをそのデータで評価したり確認することは、データから学んだことを見るうえで大きな意味がある．しかしもし仮に、学習の際に使うデータでしか狙った動作をしないとすればどうだろうか？ たとえばリスト 4.5.4 の手書き数字の認識や、リスト 4.5.5 ないしリスト 4.6.7 の顔認識で、学習に使ったデータでは正解できても、少し違う筆跡の数字や、少し表情の違う顔写真をうまく認識、分類できないとすれば、そのニューラルネットワークは役に立つ機械だろうか？ いっぽうもし仮に筆跡の違う数字や、表情の違う顔写真を正しく認識、分類できたとしたら、ニューラルネットワークは様々な場面で応用の可能性がひらかれるのではないだろうか．

これは人が勉強するときに例えるならば、丸暗記して暗記したことには正解できるけれど、問題が少し違っただけで間違えてしまうのと、そうではなくて、違う問題に対しても覚えたことを応用して対応できるのとの違いのようなもので、ちゃんと覚えているかを確認することに意味がないわけではないけれども、勉強したことが身についたかどうかは、それとは別であることと似ている．

逆にいうならば、画像として見たことのないものを、きちんと認識、分類できるならば、それは、より高次の「概念」といったものを獲得したということができるのではないだろうか？

そこでこれまで区別をしてこなかったデータを学習用と評価用に分けて、評価用はあくまでも評価に使い、これによってパラメタが影響を受けないようにして、ニューラルネットワークを学習させるようにしよう．

まずはデータを学習用と評価用に分けるプログラムを用意しよう.

リスト 4.7.1：データを学習用と評価用に分ける関数

```
import numpy as np

def split_train_test(*data, **kwargs):
    '''
    リストかタプルで与えたデータそれぞれを rate で指定した割合で train と test に分割
    shuffle=True にすればランダムに train と test に振分ける
    '''
    rate    = kwargs.pop('rate', None)    # 分割割合
    shuffle = kwargs.pop('shuffle', False) # 分割の際にデータをシャッフルするか否か
    seed    = kwargs.pop('seed', None)    # 乱数のシード値
    n_data  = len(data[0])                # データ数(各データの長さ)

    if rate == None or rate == 0:
        return data
    elif rate<0 or rate>1:
        raise Exception('rate should be in range 0 to 1.')

    print('分割割合 =', rate, 'シャッフル =', shuffle)

    index = np.arange(n_data)
    if seed is not None:
        np.random.seed(seed)
    if shuffle:
        np.random.shuffle(index)

    n_test  = int(n_data * rate)
    n_train = n_data - n_test

    index_train = index[:n_train].tolist()
    index_test  = index[n_train:].tolist()
    train, test = [], []
    for d in data:
        train.append(d[index_train])
        test.append (d[index_test])

    return train + test # リストの結合
```

データを学習用と評価用に分けるのだが、データには入力データと正解値が含まれていたり、その正解値としては one_hot の正解値と正解のカテゴリ番号のいずれか、あるいは両方が含まれていたりとさまざまだ. そこでリスト 4.7.1 の関数 split_train_test()は、可変長の仮引数*data として、そこから一つずつ取り出して、train、test に分けるようにしている. また分割割合、分割の際にデータの並びをシャッフルするかどうか、シャッフルの際の乱数のシード値は、辞書型のキーワード引数**kwargs でそれぞれキー'rate'、'shuffle'、'seed'で指定するようにしている. データの大きさは、data に与えられるデータのはじめの一つ

data[0]の長さで判断し n_data とする．kwargs に’rate’を指定しない場合にはデフォルト値を None として、引数 data をそのまま返り値とする．また rate が 0 以上 1 以下であることもチェックする．0～n_data の並びを index に作って、kwargs の’shuffle’の指定にしたがって並べ替える．train と test の数は rate に従い、その比率で index を分けておいて、data の中の各要素に適用して分割する．分割したものは返り値にする際にリストとして結合するから、train の中に元の data に含まれていた学習用のものが並び、続いて test の中に同じ並びの評価用のものが並ぶ．たとえば data に X，C，T を与えて、返り値として X_train，C_train，T_train，X_test，C_test，T_test というように受け取ることができる．　この関数は common_function.py に作っておこう．

4.7.2 評価のための準備

評価は学習とは別に行うことを踏まえて、測定のための関数を作っておこう．測定した結果は後で経過を表示するのが良いから、クラスとして定義し、インスタンス変数に測定結果を保存するようにする．

リスト 4.7.2：測定用の関数

```
class Mesurement:
    def __init__(self, model, get_acc=None):
        self.model = model
        if get_acc is not None:
            self.get_acc = get_acc
        else:
            self.get_acc = get_accuracy # get_accuracy は common_function 内に定義
        self.error, self.accuracy = [], []

    def __call__(self, x, t, mchx=False):
        y = self.model.forward(x)
        l = self.model.loss_function.forward(y, t)
        self.error.append(l)
        if mchx:
            acc, errata = self.get_acc(y, t, mchx)
            self.accuracy.append(acc)
            return l, acc, errata
        else:
            acc = self.get_acc(y, t)
            self.accuracy.append(acc)
            return l, acc

    def progress(self):
        return self.error, self.accuracy

def mesurement(model, x, t):
    return Mesurement(model)(x, t)
```

コントラクタ__init__では、引数 model に評価対象を指定する．ここではリスト 4.6.1～リスト 4.6.5 を基底クラスとするニューラルネットワークを前提にする．それで forward メソッドが定義されており、損失関数は self.loss_function としてインスタンス変数になっている．正解率測定の関数は、リスト 4.4.5 で定義した関数 get_accuracy()をデフォルトにして、いちおう引数で指定もできるようにしている．
特殊メソッド__call__では、コントラクタで設定した対象の forward()メソッドにデータ x を与えて出力を得、それを正解 t とともに損失関数の forward()メソッドにより損失 l を得る．いっぽう正解率はコントラクタで設定した関数を使うが、引数 muchx により場合を分けている．これはリスト 4.4.5 で引数 mchx で errata を得る指定があるからだが、このメソッドからも必要に応じて errata が得られる．とまれ得られた損失 l と正解率 acc はそれぞれ、self.error と self.accuracy に蓄積するとともに、返り値とする．なお特殊メソッドで__call__としているからクラス名だけで呼出せる．
メソッド progress()は、__call__で蓄積した損失 l と正解率 acc を返すだけだ．
この Mesurement クラスは、単なる測定にも使えるように、関数 mesurement()も用意した．この関数は蓄積した損失や正解率をあとから読みだすようなことは期待しない場合に便利だ．この関数は common_function.py に加えておこう．

4.7.3 ソフトマックス関数

ニューラルネットワークで画像認識を含め分類問題を扱う際、出力層には分類するカテゴリーの数だけニューロンを並べて、活性化関数は恒等関数としてきた．これまで見てきたように、これでも分類問題が扱えないわけではない．しかし出力として得たいのは、すべてのカテゴリーの中で、特定の 1 つがどういう割合なのか、つまりその確率が欲しいのだ．たとえば、この画像は猫である確率は 0.7 で、車である確率は 0.01 といったように．つまり数値として、全部の和が 1 で、それぞれの値が 0 以上 1 以下の実数となるようにしたいのだ．いっぽうの正解は one_hot だから、正解のところが 1 それ以外は 0 であって全部の和が 1 だ．だからそのようにすれば、出力と正解との比較は、より直接的だともいえるだろう．それを実現する都合の良い関数として、次式であらわされるソフトマックス関数がある．

$$y = \frac{e^x}{\sum e^x} \tag{4.7.1}$$

分子は指数関数で 0 以上の実数であり、分母はその和となっている．だから先に述べた全部の和が 1 となっているし、その値が 0 以上 1 以下の実数になって、まさにうってつけだ．そこ

でこれを活性化関数のひとつに加えよう. ただし、指数関数が非常に大きな値になることがあってオーバーフローしてしまうから、その対策が必要だ. e^xがあまり大きな値にならないようにするには、x の最大値を max_x として、$x - \max\_x \leqq 0$ だから、$e^{x-\max\_x}$ ならばオーバーフローしない. そして、

$$c = \frac{1}{e^{\max\_x}} \tag{4.7.2}$$

とおけば、

$$e^{x-\max\_x} = e^x e^{-\max\_x} = \frac{e^x}{e^{\max\_x}} = ce^x \tag{4.7.3}$$

そして式4.7.1で、 e^x を $e^{x-\max\_x}$ に置き換えると、

$$y = \frac{e^{x-\max\_x}}{\sum e^{x-\max\_x}} = \frac{ce^x}{\sum ce^x} = \frac{ce^x}{c\sum e^x} = \frac{e^x}{\sum e^x} \tag{4.7.4}$$

このように分子と分母に同じ定数 c を掛けただけで、元に戻って式4.7.1と同じになる.
活性化関数としてのソフトマックス関数の順伝播は、式4.7.4にしたがって実装することができる.
いっぽう逆伝播も必要だから、これも求めておこう. 算出過程は後述することとして、ここでは式だけ示しておく.

$$\frac{\partial y}{\partial x} = y * \left(gy - \sum^{n} y * gy \right) \tag{4.7.5}$$

順伝播も逆伝播もわかったので、Activators.py にソフトマックス関数を実装しよう.

リスト 4.7.3：ソフトマックス関数

```
class Softmax:
    def forward(self, x):
        max_x = np.max(x, axis=-1, keepdims=True)
        exp_a = np.exp(x - max_x)  # オーバーフロー対策
        sum_exp_a = np.sum(exp_a, axis=-1, keepdims=True)
        self.y = exp_a / (sum_exp_a + 1e-7)
        return self.y

    def backward(self, gy):
        y = self.y
        gx = y * gy
        sumgx = np.sum(gx, axis=-1, keepdims=True)
        gx -= y * sumgx
        return gx
```

x の最大値を求める際に軸の指定は必須だ. なぜならば、引数 x には、データが行方向、入出力が列方向の行列が与えられることが想定されるからだ. その際に、最大値はあくまで 1 組のデータの中の最大値を得たいから、axis=-1 で末尾の次元の中で最大値をとって、それをデータ分だけ並べる. このとき keepdims=True で次元が維持される. また、式 4.7.4 の分母の計算でも、末尾の次元を指定して和をとる. これもデータごとにソフトマックス関数の結果を求めたいのだから必要な指定だ.

[補足：ソフトマックスの計算グラフ]

計算グラフを使えばソフトマックス関数の逆伝播を比較的容易に導き出すことができる．計算グラフそのものの説明は省略するが、計算グラフは計算を単純な要素に分解して、その過程をグラフによってあらわしたものだ．だからここで行うように、順伝播の計算を単純な処理に分解してグラフ化すれば、その単純な処理を逆に辿ることによって、簡単には得られない逆伝播の計算式を得ることができる．

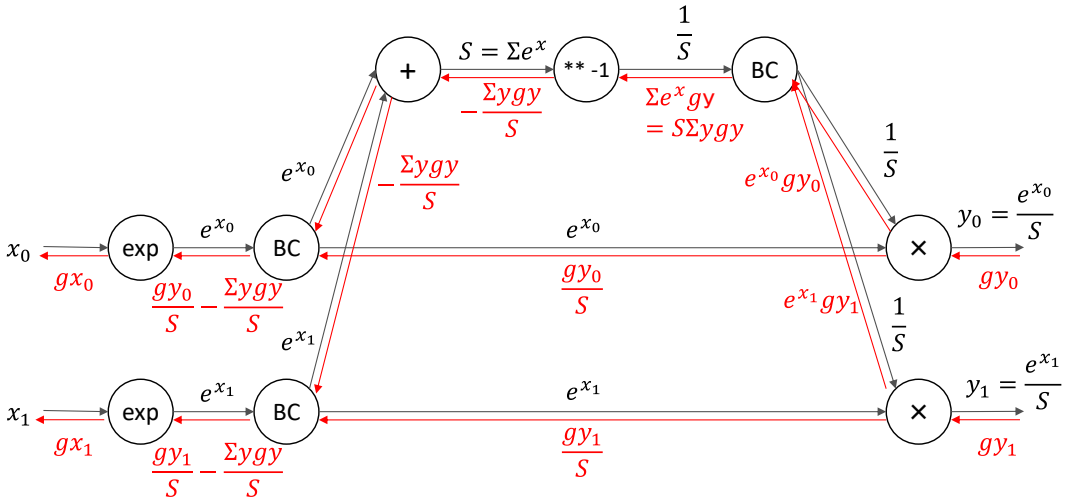

図 4.7.1 ソフトマックス関数の計算グラフ

まず計算グラフの順伝播からみていく．

はじめの exp ノードで、入力 x の指数がとられて e^x となる．

次の BC(分配)ノードで、図中で水平に進んで出力にまっすぐ向かうものと、図中で上方向に向かい合算するものに分かれる．

加算ノードには各 BC ノードから e^x が送られて合算され、$S = \Sigma e^x$となる．

次の**-1 は －1 乗するノードで $S^{-1} = \frac{1}{S}$ となる．

次の BC ノードから $\frac{1}{S}$ が各出力手前のかけ算ノードに配られる．

かけ算ノードに入ると、始めの BC ノードからまっすぐに送られる e^x と、図中の上方向にある BC ノードから送られる $\frac{1}{S}$ とが掛け合わされて $y = \frac{e^x}{S}$ が算出される．

逆伝播についてみる前に、各ノードの処理に関する微分の法則をおさらいしておこう．

まずは足し算だが、

$z = x + y$ を x と y でそれぞれ微分すれば $\frac{\partial z}{\partial x} = 1$ ， $\frac{\partial z}{\partial y} = 1$

かけ算は、

$z = x\,y$ を x と y でそれぞれ微分すれば $\frac{\partial z}{\partial x} = y$ ， $\frac{\partial z}{\partial y} = x$

べき乗は、

$y = x^a$ を微分すれば $\frac{dz}{dx} = ax^{a-1}$で、 $y = x^{-1}$ ならば $\frac{dy}{dx} = -x^{-2}$

指数は、

$y = e^x$ を微分すれば $\frac{dy}{dx} = e^x$

ではこれらを踏まえて、計算グラフの逆伝播を見ていこう．逆伝播では右端の出力から左端の入力へと辿っていく．出力端で下流からの与えられる勾配は、

$$gy = \frac{\partial L}{\partial y} \tag{4.7.6}$$

$y = e^x \times \frac{1}{S}$ ならば上述の微分の法則により、$\frac{\partial y}{\partial\left(\frac{1}{S}\right)} = e^x$ ， $\frac{\partial y}{\partial e^x} = \left(\frac{1}{S}\right)$ となるから、かけ算ノードで下流から gy が与えられたときの逆伝播は、

$$g\left(\frac{1}{S}\right) = \frac{\partial L}{\partial\left(\frac{1}{S}\right)} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial\left(\frac{1}{S}\right)} = e^x gy$$

$$ge^x = \frac{\partial L}{\partial e^x} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial e^x} = \frac{gy}{S} \tag{4.7.7}$$

BC ノードは分配するだけだが、逆伝播では分配先からの勾配が集まって、

$$g\left(\frac{1}{S}\right) = \sum e^x gy = S\sum \frac{e^x}{S} gy = S\sum y\, gy \tag{4.7.8}$$

S は入力の並びの方向で e^x を足し合わせていることに注意して **-1 ノードの逆伝播は、

$$gs = \frac{\partial L}{\partial S} = \frac{\partial L}{\partial \left(\frac{1}{S}\right)} \frac{\partial \left(\frac{1}{S}\right)}{\partial S} = g\left(\frac{1}{S}\right)(-s^{-2}) = -\frac{1}{S}\sum y\, gy \tag{4.7.9}$$

Σは各項の加算であり、加算ノードからの逆伝播は微分の法則に従って元のまま通過し、

$$ge^x = gs = -\frac{1}{S}\sum y\, gy \tag{4.7.10}$$

式 4.7.7 と式 4.4.10 で経路の違う ge^x が２つ算出された．順伝播では e^x が BC ノードで分配されたのだから、逆伝播では分配先からの勾配の和となって、

$$ge^x = \frac{gy}{S} - \frac{1}{S}\sum y\, gy = \frac{1}{S}\left(gy - \sum y\, gy\right) \tag{4.7.11}$$

exp ノードの逆伝播は、微分の法則により $\frac{\partial e^x}{\partial x} = e^x$ だから、

$$\begin{aligned} gx = \frac{\partial L}{\partial x} = \frac{\partial L}{\partial e^x}\frac{\partial e^x}{\partial x} = ge^x\, e^x &= \frac{1}{S}\left(gy - \sum y\, gy\right)e^x \\ &= \frac{e^x}{S}\left(gy - \sum y\, gy\right) \\ &= y\left(gy - \sum y\, gy\right) \end{aligned} \tag{4.7.12}$$

いささか手数はかかるものの、この以上の手順でソフトマックス関数の逆伝播の式を得ることができる.

4.7.4 交差エントロピー誤差

損失関数としては、式 4.3.1 の示す平均二乗誤差をこれまでずっと用いてきた．これはリスト 4.3.1 およびリスト 4.4.1 に示した．いっぽう分類や画像認識においては、出力層のソフトマックス関数とともに、損失関数として交差エントロピー誤差が使われることが多い．
その交差エントロピーは 2 つの確率分布の間の乖離を評価する尺度として定義される．
pは真の確率分布、qは推定した確率分布として式であらわせば、

$$H(p,q) = -\sum p(x)\log q(x) \tag{4.7.13}$$

となっていて、推定した確率分布が真の確率分布とどれだけ離れているかを表す尺度だ．これをニューラルネットワークの損失関数として定義するために、出力 $\boldsymbol{Y}$、正解値 $\boldsymbol{T}$ とし、その要素をそれぞれ y, t とすれば、k 個のサンプルに対して、

$$L = -\frac{1}{k}\sum^{k} t\ \log y \tag{4.7.14}$$

ここで正解値 $\boldsymbol{T}$ は、分類するカテゴリーに対応する幅 n の one_hot のベクトルがサンプル数 k だけ行方向に並んだ行列であり、出力 $\boldsymbol{Y}$ も同じ大きさの行列だ．サンプル数の大小によらない評価とするため、全体は k で割って 1 サンプル当たりの値にしている．老婆心ながら $\boldsymbol{Y}$ と $\boldsymbol{T}$ の要素数は $k \times n$ なので間違えないようにしよう．損失関数として実装するにあたり、順伝播は式 4.7.14 の通りだが、逆伝播も求めておこう．

$$\frac{\partial L}{\partial \boldsymbol{Y}} = \frac{\partial}{\partial \boldsymbol{Y}}\left(-\frac{1}{k}\sum^{k} t\ \log y\right) = -\frac{\boldsymbol{T}}{k\boldsymbol{Y}} \tag{4.7.15}$$

計算の詳細は省略するが、$z = \log y$ のとき、$\frac{dz}{dy} = \frac{1}{y}$ で、$\frac{\partial L}{\partial \boldsymbol{Y}}$ と $\boldsymbol{Y}$ と $\boldsymbol{T}$ が同じ形状の行列であることに注意して、各要素ごとに対応させて展開すれば上式は自ずと導かれる．
ではこれを損失関数の一つとして LossFunctions.py に組み込もう．

リスト 4.7.4：交差エントロピー誤差

```
class CrossEntropyError:
    def forward(self, y, t):
        self.y = y
        self.t = t
        self.k = y.size // y.shape[-1]
        l = - np.sum(t * np.log(y + 1e-7))
        return l / self.k

    def backward(self, gl=1):
        y = self.y
        t = self.t
        gy = - gl * t / (y + 1e-7)
        return gy / self.k
```

交差エントロピー誤差を誤差関数に定義するリスト 4.7.4 は、式 4.7.14 と式 4.7.15 をそのまま順伝播と逆伝播のメソッドに記述したものになっている．サンプル数の求め方や逆伝播の引数などはリスト 4.4.1 の平均二乗誤差と同様だ．

4.7.5 データの正規化

データの正規化については「第 3 章 線形回帰　3.7 正規化と正則化を含めた勾配の可視化」で正規化によって学習がうまくいく理由をさぐった．とまれデータの正規化はニューラルネットワークの学習においても有効だ．そこでここではニューラルネットワークの学習に使えるように、正規化の際の値の範囲を指定できるようにして、データを正規化する関数を作って common_function.py に置いておこう．

リスト 4.7.5：データの正規化

```
def normalize(data, method='standard'):
    if method is None:
        pass
    elif not method:
        pass
    elif method in('0to1', 'minmax01', 'range01'):
        data_min = np.min(data); data_max = np.max(data)
        data = (data - data_min) / (data_max - data_min)  #  0～1 の範囲
        print('データは最小値=0,最大値=1 に標準化されます')
    elif method in('-1to1', 'minmax-11','range-1to1'):
        data_min = np.min(data); data_max = np.max(data)
        data = (data - data_min) / (data_max - data_min)  #  0～1 の範囲
        data = data * 2 - 1                               # -1～1 の範囲
        print('データは最小値=-1,最大値=1 に標準化されます')
    else: # standard
        data = (data - np.average(data)) / np.std(data)
        print('データは平均値=0,標準偏差=1 に標準化されます')
    return data
```

正規化後の値の範囲は 3 通り用意し、それを引数 method で指定する．リスト 4.7.5 の中の print 文にある通り「平均値=0,標準偏差=1 に標準化」をデフォルトとして、'0to1', 'minmax01', 'range01'のいずれかを指定すれば、「最小値=0,最大値=1 に標準化」、'-1to1', 'minmax-11','range-1to1'のいずれかを指定すれば「最小値=-1,最大値=1 に標準化」として正規化する．

なお冒頭の if method is None:と elif not method:で pass としているのは、関数が呼ばれたけれど正規化せずにそのまま data を返すようにするためだ．

4.7.6 学習の進め方

ニューラルネットワークの学習は、これまで実践してきたように、データを入力して順伝播をおこなって出力を得、それと正解との乖離を損失関数で評価し、それに基づいて逆伝播を行って勾配を得、その勾配によってパラメータを更新する、ということを繰り返して、少しずつ出力が正解に近くなるようにして進める．これをもう少し考えてみよう．そもそも学習の際に少しずつパラメータを調整するのは、パラメータの数が多く、個々のパラメータが出力に対して互いに影響しあうような関係にあるために、いっぺんに最適なパラメータに到達できないからだ．これまでは繰り返し行う学習の際に毎回同じデータを使ってきた．それはそれで、データに限りがあるのだから、その限りあるデータを使わざるを得ないという理由はある．しかしデータをいっぺんに全部使うのではなく、その一部分を用い、それを変えながら学習していく、という方法も考えられる．そうして、あるひとかたまりのデータで、パ

ラメータを調整したのちに、また別のひとかたまりのデータで、と進めるならば、毎回違った観点でパラメータの調整が行われるよな具合になって、学習がよりよく進むことが考えられはしないだろうか？　もちろん毎回使うデータの一群になにがしかの大きな偏りがあるならば、かえって学習を妨げかねないから注意は必要だろう．そこで乱数を使って、一部のデータを切り出して１回の学習を行うようにし、データを使い切ったら、またデータを切り出す乱数そのものをトランプを切るように入れ替えて、切り出される部分が変わるようにすれば万全だろう．

これまでのように学習の際にデータを毎回すべて使うやり方はバッチ学習と呼ばれる．これに対し、ここで説明したように毎回の学習ではデータを一度にすべて使わずに、一部データを切り出して行うのはミニバッチ学習などと呼ばれる．そしてこれにより、確率的にうまく勾配降下を行うことが期待でき、これを確率的勾配降下法という．

4.7.7 学習用と評価用を分けた画像認識

データを学習用と評価用に分け、測定用の関数も作った．そして、ソフトマックス関数を活性化関数に加え、クロスエントロピー誤差を損失関数に加えた．さらにミニバッチ学習も説明したから、これらを活かして手書き文字認識と顔認識を動かしてみよう．もちろんこれらは比較的容易に学習することが、リスト 4.5.4、リスト 4.5.5 リスト 4.6.7 ですでに実証済みだ．ただし評価用に分けたデータで、はたして高い認識率が得られるかどうかはおおきに興味あることだろう．

手書き数字の認識をリスト 4.7.6 に顔認識をリスト 4.7.7 に示す．プログラムは少し長くなるが、やっていることはこれまでとほとんど同じだ．冒頭必要なものをインポートし、データを用意し、モデルを生成して、学習させ、学習後に経過表示と認識のテストを行う．拡張した点として、測定は学習用データとテスト用データ双方を対象にし、Mesurement クラスを用いて記録も併せて行うこと、モデルは NN_m として、中間層のニューロン数を 200、活性化関数は中間層に Sigmoid、出力層に Softmax とし、損失関数は CrossEntropyError を指定した．モデルは NN_0 でもやってみてほしいし、NN_m でも中間層のニューロン数、活性化関数、損失関数など、いろいろ変えて試してみると良い．なおリスト 4.7.6 で、重みの初期値の広がりを width=0.1 で指定しているが、これも学習の進み具合に影響するから、変えてみると良い．学習率 eta もまたしかりだ．ともかくうまくいく場合だけでなく、うまくいかない場合も試して、何がどう影響するかを見ることは無駄ではないはずだ．

リスト 4.7.6：手書き数字の認識

```
import NN
import Digits
import common_function as cf
import numpy as np

# データを用意
X, C, T = Digits.get_data()
X, C, T, X_test, C_test, T_test ¥
        = cf.split_train_test(X, C, T, rate=0.1, shuffle=True)

# モデルを作る
model = NN.NN_m(10, ml_nn=200,
                ml_act='Sigmoid',
                ol_act='Softmax',
                loss_f='CrossEntropyError'
               )
model.summary()

# 学習
epoch = 301
batch_size = 50
index_random = np.arange(len(X))
mesurement = cf.Mesurement(model)
mesurement_test = cf.Mesurement(model)
for i in range(epoch):
    np.random.shuffle(index_random)
    for j in range(0, len(X), batch_size):
        idx = index_random[j:j+batch_size]
        x = X[idx, :]
        c = C[idx, :]
        y, _ = model.forward(x, c)
        model.backward()
        model.update(eta=0.03)

    if i==0:
        print(' epoch |error train test | accuracy train test')
    if i%10==0:
        l, acc = mesurement(X, C)
        l_test, acc_test = mesurement_test(X_test, C_test)
        print('{:6d} |   {:6.3f} {:6.3f} |       {:6.3f} {:6.3f}'¥
              .format(i, float(l), float(l_test), acc, acc_test))

# 学習後に経過表示と認識のテスト
errors, accuracy = mesurement.progress()
errors_test, accuracy_test = mesurement_test.progress()
cf.graph_for_error(errors, accuracy, errors_test, accuracy_test,
                   label=('error','accuracy', 'error_test', 'accuracy_test'),
                   xlabel='n_learn')
cf.test_sample(Digits.show_sample, model.forward, X_test, T_test)
```

リスト 4.7.7：顔認識

```
import NN
import OlivettiFaces as OF
import common_function as cf
import numpy as np

# データを用意
X, C, T = OF.get_data()
X, C, T, X_test, C_test, T_test ¥
        = cf.split_train_test(X, C, T, rate=0.1, shuffle=True)
label_list = OF.label_list()

# モデルを作る
model = NN.NN_m(40, ml_nn=200,
                width=0.1,
                ml_act='Sigmoid',
                ol_act='Softmax',
                loss_f='CrossEntropyError'
                )
model.summary()

# 学習
epoch = 501
batch_size = 50
index_random = np.arange(len(X))
mesurement = cf.Mesurement(model)
mesurement_test = cf.Mesurement(model)
for i in range(epoch):
    np.random.shuffle(index_random)
    for j in range(0, len(X), batch_size):
        idx = index_random[j:j+batch_size]
        x = X[idx, :]
        c = C[idx, :]
        y, _ = model.forward(x, c)
        model.backward()
        model.update(eta=0.03)

    if i==0:
        print(' epoch |error train test | accuracy train test')
    if i%10==0:
        l, acc = mesurement(X, C)
        l_test, acc_test = mesurement_test(X_test, C_test)
        print('{:6d} |   {:6.3f} {:6.3f} |       {:6.3f} {:6.3f}'¥
              .format(i, float(l), float(l_test), acc, acc_test))

# 学習後に経過表示と文字認識のテスト
errors, accuracy = mesurement.progress()
errors_test, accuracy_test = mesurement_test.progress()
cf.graph_for_error(errors, accuracy, errors_test, accuracy_test,
                   label=('error','accuracy', 'error_test', 'accuracy_test'),
                   xlabel='n_learn')
cf.test_sample(OF.show_sample, model.forward, X_test, T_test, label_list)
```

4.8 物体認識への挑戦と課題

ニューラルネットワークが画像認識で大きな能力を有することを見ることができたはずだ．しかしより実際的なデータではどうなんだろうか？　なぜなら手書き数字も顔画像もそれぞれ 8×8 ないし 64×64 の画像の枠の中に位置も大きさも揃えてきれいに並べられており、余計なものは含まれていない．しかし実際に写真を写せば、画像の向きや大きさもばらばらだし、そのうえ背景にはさまざまなものが映り込むだろう．そこでここでは「4.7 データと学習」で行った手書き文字や顔認識に比べて、より実践的な物体認識に取り組んでいくことにしよう．それはかなり難しく、がっかりさせられることが予想される．しかし課題を明らかにして更なる発展を展望していくためには避けて通れないことだと思う．

4.8.1 物体認識のデータ

物体認識でしばしば使われるデータセットとして下記から入手可能な CIFAR-10 がある．

https://www.cs.toronto.edu/~kriz/cifar.html

図 4.8.1 に含まれる画像の一部を示すが、CIFER-10 には訓練用 5 万枚＋テスト用 1 万枚の画像とそれに対する正解ラベルが収録されている．

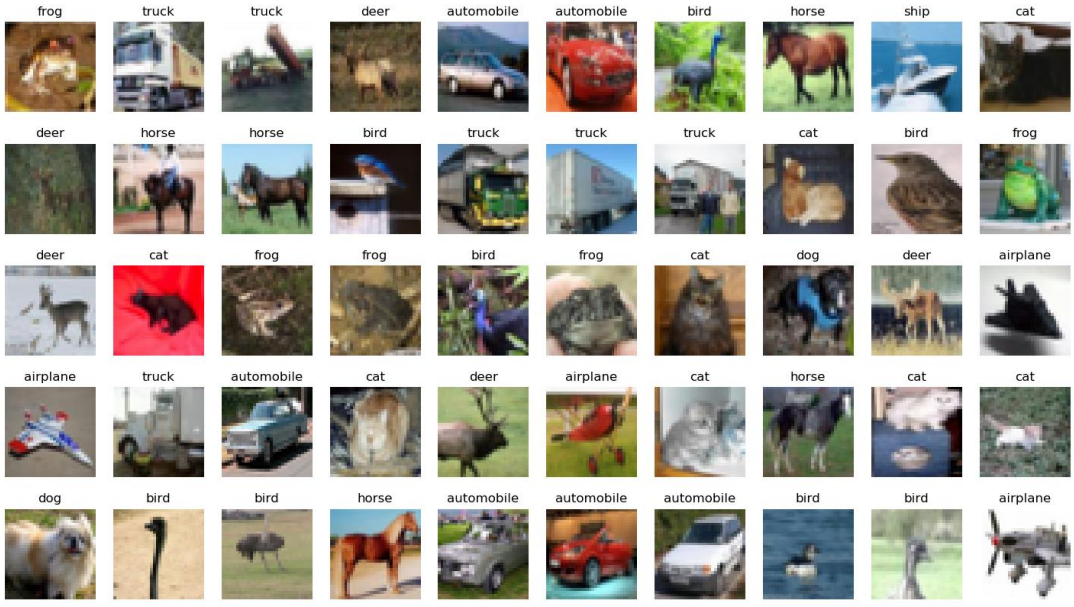

図 4.8.1 CIFER10 の画像の一例

これを入手するには、上記にアクセスして、その中のダウンロードへのリンク
CIFAR-10 python version
をクリックすれば良い．そしてダウンロードされる
Cifar-10-python.tar.gz
を解凍すれば、フォルダ cifar-10-batchs-py の下に
batches.meta，data_batch_1～data_batch_5，readme.html，test_batch の 8 つのファイルが得られる．
これらのファイルを読むためのコードも記載されているから参考にしつつ、データを入手するプログラムを作ろう．

リスト 4.8.1：CIFER-10 のデータを入手～その 1

```
import numpy as np

def unpickle(file):
    import pickle
    with open(file, 'rb') as fo:
        dict = pickle.load(fo, encoding='bytes')
    return dict

def read_batch(file):
    print(file)
    dict = unpickle(file)
    data   = dict[b'data']
    labels = dict[b'labels']
    return data, labels

def load_data(datapath=''):
    data1, labels1 = read_batch(datapath + 'data_batch_1')
    data2, labels2 = read_batch(datapath + 'data_batch_2')
    data3, labels3 = read_batch(datapath + 'data_batch_3')
    data4, labels4 = read_batch(datapath + 'data_batch_4')
    data5, labels5 = read_batch(datapath + 'data_batch_5')
    x_train = np.vstack((data1, data2, data3, data4, data5))
    t_train = np.hstack((labels1, labels2, labels3, labels4, labels5)).astype('uint8')
    x_test, t_test = read_batch(datapath + 'test_batch')
    x_test = np.array(x_test)
    t_test = np.array(t_test, dtype='uint8')
    return x_train, t_train, x_test, t_test
```

関数 unpickle()は、上記のウェブページで示されたものそのままだ．この返り値 dict は辞書型で、データと正解ラベルのキーは、バイト型の b'data' と b'labels' だ．そこで関数 read_batch()は、関数 unpickle()を使って dict を得、そこからデータと正解ラベルを取り出して、返り値 data と labels にする．関数 load_data()は、この一連の処理を data_batch_1～

data_batch_5 に行って、データは numpy の関数 vstack()で、labels は関数 hstack()で、まとめてそれぞれ x_train と t_train とする．このとき t_train はデータ型を明示的に指定して元の labels の各要素のデータ型に合わせておく．いっぽう test_batch は関数 read_batch()での処理は同じだが、マージする必要はなく、numpy の関数 array()で整えるだけだ．
これで CIFER-10 のデータが使えるのだが、もう一工夫しておこう．CIFER-10 のデータには labes で正解の番号が与えられるが、それが何であるかを言葉どおりに得る関数 label_list()を用意しておこう．

リスト 4.8.2：CIFER-10 のデータを入手～その 2

```
def label_list():
    label = ['airplane',   # 0
             'automobile', # 1
             'bird',       # 2
             'cat',        # 3
             'deer',       # 4
             'dog',        # 5
             'frog',       # 6
             'horse',      # 7
             'ship',       # 8
             'truck']      # 9
    return label
```

続いて画像を表示するための関数 show_sample()を用意しよう．

リスト 4.8.3：CIFER-10 のデータを入手～その 3

```
import numpy as np
import matplotlib.pyplot as plt

def show_sample(data, label):
    rdata = data[0] if data.ndim==4 else data
    rdata = data.reshape(3, 32, 32) if data.ndim<=2 else rdata
    rdata = rdata.transpose(1, 2, 0) if rdata.shape[0]==3 else rdata
    max_picel = np.max(rdata); min_picel = np.min(rdata)
    rdata = (rdata - min_picel)/(max_picel - min_picel)  # 画素データを 0～1 に補正
    plt.imshow(rdata.tolist())
    plt.title(label)
    plt.show()
```

厄介なのはデータの次元と並びが場合によって違うことだ．そして縦横 32×32 のカラーの画像で、その 3 原色の次元を縦横の前にした方がニューラルネットワークでは扱いやすい一方、画像の表示では縦横の後に色の次元を置く必要がある．そのうえニューラルネットワークの入出力で、縦横と色があわせて 1 次元でベクトルにされて、元の並びがわからない場合もある．だから、すべてのケースに対応することは出来ず、取ってつけたような対応にならざるを得ないが表示に適した形にするようにしている．また、CIFER-10 の元のデータは 0～255 の 'uint8' 形式だが、ニューラルネットワークで扱うために、正規化するとともに、浮動小数点型にする．そこで matplotlib で表示するには値を 0～1 の範囲に調整する必要もある．

最後に、アヤメ、手書き文字、顔画像など一連の sklearn のデータ同様のインタフェースを提供するための関数 get_data()を作っておこう．ほとんどのことはリスト 4.8.1 でできているから、データの軸の切り出しや入替え、データの標準化、そしてデータの素性を表示するなど、便利な機能を加えておこう．

リスト 4.8.4：CIFER-10 のデータを入手～その 4

```
def get_data(path = '../cifar-10-batches-py/', **kwargs):
    x_train, t_train, x_test, t_test = load_data(path)
    print('CIFER-10 のデータが読み込まれました')

    # ― データの軸の切り出し ―
    if kwargs.pop('image', False):
        x_train = x_train.reshape(-1, 3, 32, 32)
        x_test  = x_test .reshape(-1, 3, 32, 32)

    # ― データの軸の入替え ―
    if kwargs.pop('transpose', False):
        print('データの軸を画像表示用に入れ替えます (B, C, Ih, Iw) -> (B, Ih, Iw, C)')
        x_train = x_train.transpose(0, 2, 3, 1)
        x_test  = x_test .transpose(0, 2, 3, 1)

    # ― データの標準化 ―
    normalize = kwargs.pop('normalize', True)
    if normalize is not None: # True, 0to1, -1to1 など指定可能
        x_train = cf.normalize(x_train, normalize)
        x_test  = cf.normalize(x_test,  normalize)

    # ― 正解を one-hot 表現に ―
    c_train = np.eye(10)[t_train].astype(Config.dtype)
    c_test  = np.eye(10)[t_test] .astype(Config.dtype)

    print('訓練用の入力と正解値のデータが用意できました')
    print('入力データの形状', x_train.shape, '正解値の形状', c_train.shape)
    print('データの素性 最大値 {:6.3f} 最小値 {:6.3f} 平均値 {:6.3f} 分散 {:6.3f}' ¥
        .format(float(np.max(x_train)), float(np.min(x_train))¥
              , float(np.mean(x_train)), float(np.var(x_train))))
    print('評価用の入力と正解値のデータが用意できました')
    print('入力データの形状', x_test.shape, '正解値の形状', c_test.shape)
    print('データの素性 最大値 {:6.3f} 最小値 {:6.3f} 平均値 {:6.3f} 分散 {:6.3f}' ¥
        .format(float(np.max(x_test)), float(np.min(x_test))¥
              , float(np.mean(x_test)), float(np.var(x_test))))

    return x_train, c_train, t_train, x_test, c_test, t_test
```

4.8.2 物体認識の実行

CIFER-10 のデータの準備ができた.「4.7 データと学習」では、ニューラルネットワークをよりよく学習させるようにしたはずだ. そこで物体認識を動かしてみよう. リスト 4.7.5 やリスト 4.7.6 とほとんど同じだが、データが大きく学習に時間がかかることが予想されるから、time モジュールを使って経過時間を表示するようにしておこう.

リスト 4.8.5：物体認識

```
import NN
import CIFER10
import common_function as cf
import numpy as np
import time

# データを用意
X, C, T, X_test, C_test, T_test = CIFER10.get_data()
label_list = CIFER10.label_list() # 名簿

# モデルを作る
model = NN.NN_m(10, ml_nn=200,
                ml_act='Sigmoid',
                ol_act='Softmax',
                loss_f='CrossEntropyError'
                )
model.summary()

# 学習
epoch = 101
batch_size = 100
start = time.time()
index_random = np.arange(len(X))
mesurement = cf.Mesurement(model)
mesurement_test = cf.Mesurement(model)
for i in range(epoch):
    np.random.shuffle(index_random)
    for j in range(0, len(X), batch_size):
        idx = index_random[j:j+batch_size]
        x = X[idx, :]
        c = C[idx, :]
        y, _ = model.forward(x, c)
        model.backward()
        model.update(eta=0.01)

    if i==0:
        print(' epoch elapse |error train test | accuracy train test')
    if i%10==0:
        l, acc = mesurement(X, C)
        l_test, acc_test = mesurement_test(X_test, C_test)
        print('{:6d} {:6.3f} |   {:6.3f} {:6.3f} |       {:6.3f} {:6.3f}'¥
              .format(i, time.time()-start, float(l), float(l_test), acc, acc_test))
        start = time.time()

# 学習後に経過表示と認識のテスト
errors, accuracy = mesurement.progress()
errors_test, accuracy_test = mesurement_test.progress()
cf.graph_for_error(errors, accuracy, errors_test, accuracy_test,
                   label=('error','accuracy', 'error_test', 'accuracy_test'),
                   xlabel='n_learn')
cf.test_sample(CIFER10.show_sample, model.forward, X_test, T_test, label_list)
```

さてそれでは結果を見てみよう．経過が表示されて、うまくいけば学習が進む様子が見られる．学習が完了するとグラフで経過が表示されるが、一例は次のようになっている．

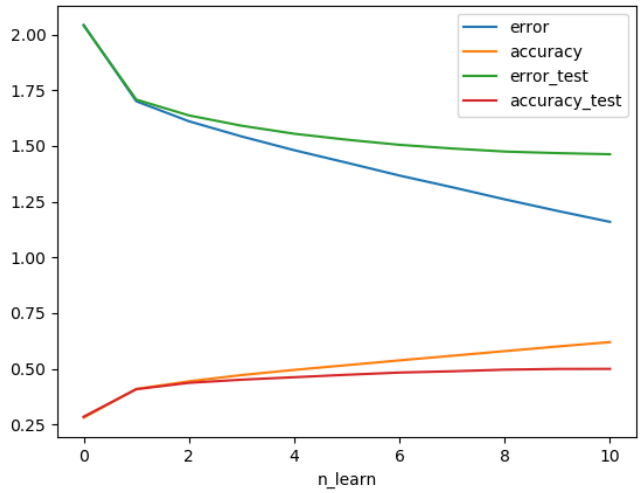

図 4.8.2 CIFER10 の学習

エラーは減少し正解率は上昇する．しかしエラーは小さくならず、正解率は 50%くらいと芳しくない．これは予想通りであって、課題の難しさを表しているにほかならない．しかしまだエラーは減少傾向、正解率は上昇傾向だ．次第に学習データとテストデータで乖離し始めているが、ともかく、学習回数を 1000 回に増やしてみると次のようになる．

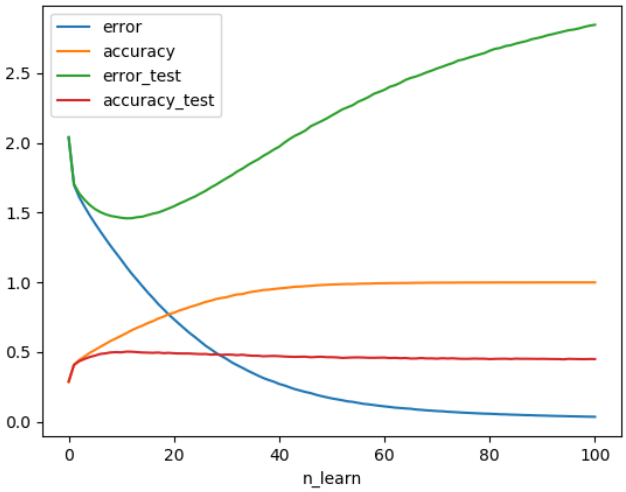

図 4.8.3 CIFER10 の学習

学習を繰り返すことで、学習データについては、エラーが減少し正解率が上がり、ついに正解

率は 100%に到達する．しかし、なんということだろう？ テストデータでは、エラーは途中から露骨にエラーが上昇し、正解率は頭打ちとなったのち少し劣化する．やはり難しさを突き付けられたかたちだ．同じ事物でも色も形も様々だし、画像の中に占める対象物の大きさも、どの部分かも違う．首だけ写されたダチョウや、保護色に埋もれた蛙やら、を鳥や蛙に分類するのは人には容易でも確かに難しそうだ．それでも学習データについては正解率が 100%になるのだ．いっぽうでテストデータでは状況が全く異なり、残念ながら、ここまでの範囲では、これを大幅に向上させる策はない．しかしここで明らかにしておきたいのは次の点だ．すなわち、学習データとテストデータの結果が乖離していくということこそが、何とかしなければならない真の問題だということだ．

4.9 敵対的生成ネットワーク

前節ではがっかりさせられたことだろう．しかしこれを何とかするには、まだまだやらなければならないことが多い．そこでここでは視点を変えて、ニューラルネットワークで生成を行うことに挑戦して締めくくりとすることにしよう．そしてその代表的な一つ、敵対的生成ネットワーク GAN（Genarative Adversarial Network）を動かしてみよう．GAN を動かすにあたり、少しばかり新たな機能を加えなければならないが、いずれもここまでの範囲を理解していれば簡単だ.

4.9.1 ランプ関数（Rectified linear function）

活性化関数としてはこれまで、恒等関数(リスト 4.4.2)、シグモイド関数(リスト 4.6.8)、ソフトマックス関数(リスト 4.7.3)を作ってきた．ここでもう一つ、活性化関数としてとてもよく使われるランプ関数を導入しよう．ランプ関数は場合分けにより次式であらわされる.

$$y = \begin{cases} x, & x \geqq 0 \\ 0, & x < 0 \end{cases} \tag{4.9.1}$$

ランプ関数は原点 0 では不連続で微分できない．しかし、それ以外のところでは問題なく、次式であらわされる.

$$\frac{dy}{dx} = \begin{cases} 1, & x > 0 \\ 0, & x < 0 \end{cases} \tag{4.9.2}$$

簡単なので Activators.py に活性化関数の一つ ReLU クラスとして実装してしまおう．ただし実装するにあたり逆伝播で $x = 0$ が定義されないのは困るから、式 4.8.1 にならい場合分けの一方に入れておくことにする.

リスト 4.9.1：ReLU

```
class ReLU:
    def forward(self, x):
        y = np.where(x>=0, x, 0)
        self.x = x
        return y

    def backward(self, gy):
        x  = self.x
        gx = gy * np.where(x>=0, 1, 0)
        return gx
```

ついでながら、式4.9.1で $x < 0$ の時に0ではなく、$y = x$ ではないけれど、何某かの係数 c $(0 < c < 1.0)$ で傾きを与えて $x < 0$ でも x が無視されないようにすることが考えられる．これを活性化関数LReLUとして実装しよう．

リスト4.9.2：LReLU

```
class LReLU():
    def __init__(self, C=0.01):
        self.c = c

    def forward(self, x):
        y = np.where(x>=0, x, self.c * x)
        self.x = x
        return y

    def backward(self, gy):
        x  = self.x
        gx = gy * np.where(x>=0, 1, self.c)
        return gx
```

リスト4.9.2のLReLUは、 $x < 0$ の時の係数 c をデフォルト0.01として、コントラクタの引数で指定もできるようにしている．

リスト4.9.1のReLUも、リスト4.9.2のLReLUも簡単なので間違えることもないとは思うが、リスト4.6.10でシグモイド関数で行ったのと同じやり方で、この特性を確認しておこう．

リスト4.9.3：順伝播と逆伝播の確認

```
import numpy as np
import matplotlib.pyplot as plt
import Activators

x = np.linspace(-2, 2, 1000)
func = Activators.ReLU()
#func = Activators.LReLU()
#func = Activators.Tanh()

y = func.forward(x)
dydx = func.backward(gy=1)

plt.plot(x, y)
plt.plot(x, dydx)
```

リスト 4.9.3 を実行すると次図を得る.

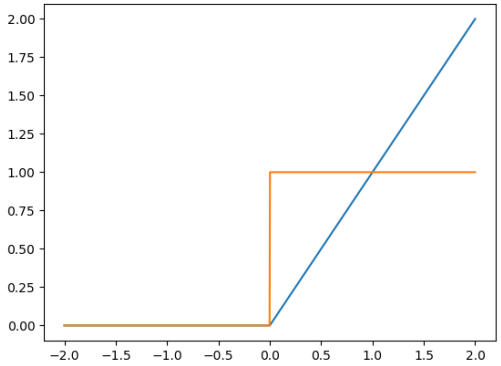

図 4.9.1 ReLU の順伝播と逆伝播

順伝播がランプ関数、逆伝播がステップ関数となっていることが確認できる.
リスト 4.9.3 でコメントアウトしている LReLU では次図となる.

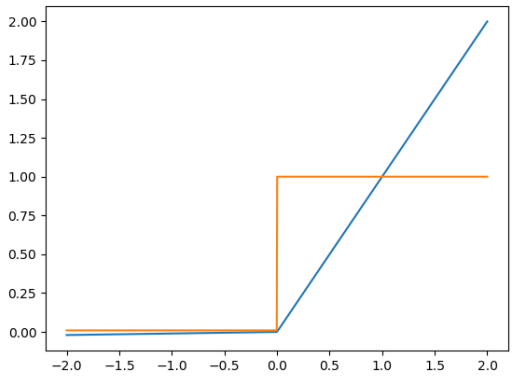

図 4.9.2 LReLU の順伝播と逆伝播

LReLU も ReLU とほとんど同じだが、 $x < 0$ でもわずかに勾配を持つことが確認できる.

4.9.2 ハイパボリックタンジェント関数（Hyperbolic tangent function, tanh）

活性化関数に加えておきたい関数がもう一つある．それは次式であらわされる双曲線正接またはハイパボリックタンジェントと呼ばれる関数だ．

$$y = \tanh x = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{4.9.3}$$

これは実装してしまってから特性を確認するが、 $x \to \infty$ で $y \to 1$ ， $x \to -\infty$ で $y \to -1$ で $-1 < y < 0$ であって、原点を通りかつ原点対称だ．また微分可能であり、

$f(x) = e^x + e^{-x}$ および $g(x) = e^x - e^{-x}$ とおけば、$y = \frac{g(x)}{f(x)}$ となり、商の微分の公式から、

$$\frac{dy}{dx} = \frac{d}{dx}\left(\frac{g}{f}\right) = \frac{g'f - f'g}{f^2}$$

ここで、

$$f' = \frac{df(x)}{dx} = \frac{d}{dx}(e^x + e^{-x}) = e^x - e^{-x}$$

$$g' = \frac{dg(x)}{dx} = \frac{d}{dx}(e^x - e^{-x}) = e^x + e^{-x}$$

なんとなれば $z = e^x$ とおけば、

$$\frac{d}{dx}e^{-x} = \frac{d}{dz}z^{-1}\frac{d}{dx}e^x = -z^{-2}e^x = -e^{-2x}e^x = -e^{-x}$$

したがって、

$$\begin{aligned}\frac{dy}{dx} = \frac{d}{dx}\left(\frac{e^x - e^{-x}}{e^x + e^{-x}}\right) &= \frac{(e^x + e^{-x})^2 - (e^x - e^{-x})^2}{(e^x + e^{-x})^2} \\ &= \frac{(e^x + e^{-x})^2}{(e^x + e^{-x})^2} - \frac{(e^x - e^{-x})^2}{(e^x + e^{-x})^2} \\ &= 1 - y^2 \end{aligned} \tag{4.9.5}$$

それではハイパボリックタンジェント関数を活性化関数の一つとして実装しよう．

リスト 4.9.4：Tanh

```
class Tanh:
    def forward(self, x):
        y = np.tanh(x)
        self.y = y
        return y

    def backward(self, gy):
        y = self.y
        gx = (1 - y * y) * gy
        return gx
```

この特性はリスト 4.9.3 でコメントアウトしている func = Activators.Tanh() を生かせば確認できる．なお図 4.9.3 ではリスト 4.9.3 の入力 x の値の範囲を少し広げて、
x = np.linspace(-3, 3, 1000) として描いた．

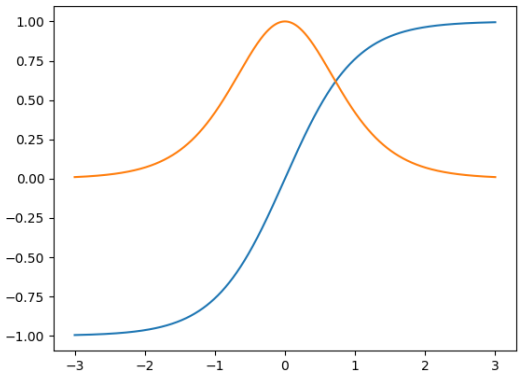

図 4.9.3 Tanh の順伝播と逆伝播

図 4.9.3 に示すハイパボリックタンジェント関数は、図 4.6.4 のシグモイド関数に似ているが、原点を中心に対称となっていて、値の範囲が -1～1 となっていることがわかる．

4.9.3 二値の交差エントロピー誤差

「4.7 データと学習」では手書き数字認識と顔認識、「4.8 物体認識への挑戦と課題」ではCIFER10の画像認識に取り組んだが、たとえば手書き数字ではそれが0～9のいずれの数字なのかを識別したように、いずれも画像をいくつかのカテゴリーに分類する問題であり、すなわちこれらは多値分類だった．これをニューラルネットワークで扱う場合には、出力層に分類するカテゴリーに一対一で対応させたニューロンを設けた．だから分類のカテゴリーの数＝出力層のニューロンの数だった．そして、ソフトマックス関数を活性化関数に用い、交差エントロピーを損失関数に用いることで、多値分類を扱うモデルを作ることができることを見てきた．

それでは白黒、yes／no、0／1、正誤のように二値に分類する場合はどうだろうか？これまでのやり方を踏襲するならば、出力層のニューロンは2つとして、多値分類と同じように構成すればよい．しかし、二値分類では、2つの値の一方が決まればもう一方はその反対というようにするのが自然であり、わざわざ2つの出力を取り出すまでもないだろう．そこで出力層のニューロンを1つにして出力を一つだけ取り出すとする．その出力 y が分類の2つのうちどちらか一方である確率を表すとすれば、そうでない方である確率は $1-y$ ということになる．こういう出力 y を生成するのに都合の良い関数はすでに実装済みだ．リスト4.6.8のシグモイド関数がそれだ．そして多値分類で損失関数は交差エントロピーだったが、これに y と $1-y$ との両方を入れるようにすれば良いのだから、正解値も出力に合わせて対で用意して t と $1-t$ として、損失の式は式4.7.14を変形して次のようになる．ここで y，t は任意の対となる出力と正解をあらわす．

$$L = -\frac{1}{k}\sum^{k} t\ \log y + (1-t)\ \log(1-y) \tag{4.9.6}$$

この式4.9.6であらわされる関数を二値の交差エントロピー誤差という．逆伝播も式4.7.15をもとにして求める．

$$\frac{\partial L}{\partial y} = -\frac{t}{ky} + \frac{1-t}{k(1-y)} = \frac{-t(1-y)+y(1-t)}{ky(1-y)} = \frac{y-t}{ky(1-y)} \tag{4.9.7}$$

これをデータに合わせて並べ、ベクトルとすれば次式となる．$\boldsymbol{Y}$ も $\boldsymbol{T}$ も同じ形状で、値'1'がこれらと同じ形状に並んだベクトルを $\mathbf{1}$ とすれば、

$$\frac{\partial L}{\partial \boldsymbol{Y}} = \frac{\boldsymbol{Y} - \boldsymbol{T}}{k\boldsymbol{Y}(\boldsymbol{1} - \boldsymbol{Y})} \tag{4.9.8}$$

ではさっそく LossFunctions.py に実装しよう.

リスト 4.9.5：二値のクロスエントロピー誤差

```
class CrossEntropyError2():
    def forward(self, y, t):
        self.y = y
        self.t = t
        self.k = y.size // y.shape[-1]  # 時系列データで必須
        l = -np.sum(t * np.log(y + 1e-7) + (1 - t) * np.log(1 - y + 1e-7))
        return l / self.k

    def backward(self, gl=1):
        y = self.y
        t = self.t
        gy = gl * (y - t) / (y * (1 - y) + 1e-7)
        return gy / self.k
```

4.9.4 GAN の実装

この期におよんであれこれ作ったが、いずれも先々使うことが想定される．そのいっぽうで、ディープラーニングに歩を進めるには、まだまだ作らなければならないものが多い．とまれごく基本的な材料は揃ったので容易に学習した手書き数字のデータを使って GAN を動かしてみよう．GAN は 2 つのニューラルネットワークを、一つは生成器、もう一つは識別器として組み合わせて構成する．GAN を数学的に説明しようとすれば難しい．しかしごく簡単に言うならば次のようになるだろう．すなわち生成器が生成する偽物と、データによって与えられる本物を、識別器に入れて見分ける．このとき識別器は両者をより的確に見分けられるように学習するのに対し、生成器は識別器が誤りやすくなるように学習する．そして両者が拮抗する中で、生成器はより本物に近いものを生成できるようになるということだ．とまれ作って動かしてみるのが理解への早道だ．ニューラルネットワークはクラスとして作ったが、これを部品として使って、GAN のクラスとしよう．そして「はじめの一歩」の意味を込めてファイル名は GAN0.py としておこう.

リスト 4.9.6：GAN の実装

```
import numpy as np
import NN, LossFunctions

class GAN_m_m:
    def __init__(self, nz, out, **kwargs):
        opt_for_gen, opt_for_dsc = {}, {}
        keys_kwargs = list(kwargs.keys()) # iterator(ループ内変更不可)
        for k in keys_kwargs:
            if k[:4]='gen_':
                opt_for_gen[k[4:]]=kwargs.pop(k)
            if k[:4]='dsc_':
                opt_for_dsc[k[4:]]=kwargs.pop(k)
        # 生成器のオプション
        opt_for_gen['ml_nn']  = opt_for_gen.pop('ml_nn',       256 )
        opt_for_gen['ml_act'] = opt_for_gen.pop('ml_act',    'ReLU')
        opt_for_gen['ol_act'] = opt_for_gen.pop('ol_act',    'Tanh')
        opt_for_gen['width']  = opt_for_gen.pop('width',      0.01 )

        # 判別器のオプション
        opt_for_dsc['ml_nn']  = opt_for_dsc.pop('ml_nn',       256 )
        opt_for_dsc['ml_act'] = opt_for_dsc.pop('ml_act',   'LReLU')
        opt_for_dsc['ol_act'] = opt_for_dsc.pop('ol_act', 'Sigmoid')
        opt_for_dsc['width']  = opt_for_dsc.pop('width',      0.01 )

        # 生成機と判別器
        print(opt_for_gen, opt_for_dsc)
        self.gen = NN.NN_m(out, **opt_for_gen)
        self.dsc = NN.NN_m(1,   **opt_for_dsc)
        self.loss_function = LossFunctions.CrossEntropyError2()
        self.nz = nz # ノイズの大きさ
        self.gen.summary()
        self.dsc.summary()
        print('loss_function =', self.loss_function.__class__.__name__, '¥n')
```

リスト 4.9.6 にクラス GAN_m_m のコントラクタを示す．引数 nz は画像生成の元となるノイズの大きさ、引数 out は生成する画像の大きさだ．また生成機と識別器それぞれ何らかの設定を**kwargs で設定できるようにしている．このため**kwargs からキーを抽出し、そのキーが 'gen_' で始まるか、'dsc_' で始まるかに応じて、キーの頭の文字を取り除いたうえで振分けて、opt_for_gen と opt_for_dsc とする．続いてそれぞれの中間層のニューロン数、活性化関数、出力層の活性化関数、それから重みの初期値の広がり width のデフォルトを与えつつ、設定する．なおこの時にリスト 4.6.12 の NN_m のコントラクタやリスト 4.1.1 の BaseLayer のコントラクタなどにこれらの設定値が渡される．そのあと確認のため opt_for_gen と opt_for_dsc を出力し、self.gen と self.dsc として NN_m をオプションを指定しつつインスタンス化する．これらとは独立にリスト 4.9.5 で作った二値のクロスエントロピー誤差を損失

関数に指定する．nz は self.nz に保存し、生成器と識別器の構成と損失関数を出力して完了する．

GAN は訓練が難しく、こうすれば必ずうまくいくというものは見いだせない．それ故にこそ訓練の進行具合を把握することは欠かせない．そこで GAN の事情に合わせた測定用の関数をを作ることにしよう．

リスト 4.9.7：GAN の測定用の関数

```
class Mesurement_for_GAN:
    def __init__(self, model):
        self.model = model
        self.error, self.accuracy = [], []

    def get_accuracy(self, y, t):
        correct = np.sum(np.where(y<0.5, 0, 1) == t)
        return correct / len(y)

    def __call__(self, x, t):
        y = self.model.dsc.forward(x)
        loss = self.model.loss.forward(y, t)
        acc  = self.get_accuracy(y, t)
        loss = float(loss)
        acc  = float(acc)
        self.error.append(loss)
        self.accuracy.append(acc)
        return loss, acc

    def progress(self, moving_average=None):
        if moving_average is None:
            return self.error, self.accuracy
        else:
            r = int(moving_average)
            n = len(self.error) - r
            err_ma = []
            acc_ma = []
            for i in range(n):
                err_ma.append(float(np.average(self.error[i:i+r])))
                acc_ma.append(float(np.average(self.accuracy[i:i+r])))
            return err_ma, acc_ma
```

測定用の関数はリスト 4.7.2 で作成済みだが、それは損失関数も一体として組み込まれた一般的なニューラルネットワークを前提としている．これに対し GAN は、リスト 4.9.6 で示すように、生成器と識別器とをそれぞれニューラルネットワークとし、損失関数はそのいずれとも独立に設けているから、そのままでは使えない．GAN にも対応するようにできなくもないが、かえって煩雑になるから GAN 専用として別に用意した．さらに正解率取得の関数はリ

スト 4.4.5 で作成済みだが、GAN に必要な二値の場合には対応していないから、これも別に用意し、メソッドの一つとして組み込んだ.
正解率の求め方だが、二値分類だから、出力 y の値が 0.5 より小さいならば'0'、大きければ'1'として、それと正解値 t を比較する. 比較によりできるベクトルは、正解が'1'不正解が'0'の値の要素が並ぶ. このベクトルの'1'の合計を出力の要素数で割れば、このベクトルの'1'の割合を求めることになる. すなわち正解率となる. メソッド__call__では識別器の順伝播、損失関数など呼び出し方をリスト 4.9.6 の構成に合わせているが、正誤表は不要だから、リスト 4.7.2 のような場合分けはない.
そして progress()メソッドでは、引数の指定で移動平均を返すように拡張した. 実は GAN の学習ではエラーも正解率も激しく変化するので、この拡張をしておくと便利なのだ. 移動平均は株価の変動など変化の激しいものの傾向を見るのに使われることが多いが、ここでは self.error と self.accuracy にある結果から、引数 moving_average で指定する区間を順に抜き出して、その算術平均をとって、それをあらたに err_ma と acc_ma にリストとして並べる.
リスト 4.9.7 の GAN の測定用の関数も、リスト 4.7.2 と同じく common_function.py に加えておこう.

それではリスト 4.5.4 やリスト 4.7.5 で画像認識として取り上げた手書き数字のデータを使って GAN を訓練してみよう．プログラムは少し長いのでリスト 4.9.8、リスト 4.9.9 の 2 つに分けて掲載する．

リスト 4.9.8：GAN の学習～その 1

```
import common_function as cf
import Digits as  DT
import GAN0
import time
import numpy as np

# ― 各設定値 ―
epoch      = 501    # 学習回数
nz         = 10     # 潜在変数の数
interval   = 10     # 経過の表示間隔
batch_size = 100    # バッチサイズ
eta        = 0.02   # 学習率

# ― 訓練データ ―
X, C, T = DT.get_data()
X = cf.normalize(X, '-1to1')

# ― モデルの生成 ―
model = GAN0.GAN_m_m(nz, 64, gen_ml_nn=100, dsc_ml_nn=100, gen_width=0.1)
mesurement_real = cf.Mesurement_for_GAN(model)
mesurement_fake = cf.Mesurement_for_GAN(model)
```

まず最初の部分は必要なモジュールのインポートに始まり、主に学習の設定値、そして訓練データを持ち込む．データは値が-1～1 の範囲になるように正規化する．これはリスト 4.9.6 で GAN の生成器の出力層の活性化関数をハイパボリックタンジェント tanh としていることと対応する．すなわち tanh が生成する値は-1～1 の範囲だから、生成器の作る偽画像の画素の値は-1～1 の範囲となる．GAN は、この偽画像と実画像を照合しながら学習するから、両者の値の範囲が一致している必要があるのだ．あとは GAN を model としてインスタンス化し、併せてリスト 4.9.7 の GAN の測定用のクラスをインスタンス化する．

続いて学習の部分だ.

リスト 4.9.9：GAN の学習～その 2

```
# ― GAN の学習 ―
start = time.time()
n_itr = len(X) // batch_size
index_rand = np.arange(len(X))
for i in range(epoch):
    np.random.shuffle(index_rand)
    rand = np.random.randn(len(X), nz)                  # 生成器入力ノイズ
    for j in range(n_itr):
        # データの準備、ノイズの準備、バッチ切り出し、本物画像の選択
        batch_idx = index_rand[j*batch_size:(j+1)*batch_size]
        rand2 = np.random.rand(batch_size, 1)           # 実偽の選択乱数
        imgs_real = X[batch_idx]
        imgs_fake = model.gen.forward(rand[batch_idx])

        # ― 識別器の訓練：本物画像と生成器が作った偽画像で訓練
        t = np.where(rand2>0.5, 1, 0)
        x = imgs_real * t + imgs_fake * (1 - t)
        y = model.dsc.forward(x)
        model.loss_function.forward(y, t)
        gy = model.loss_function.backward()
        model.dsc.backward(gy)
        model.dsc.update(eta=eta)

        # ― 生成器の訓練：すべて実画像と識別器を騙して逆伝播し生成器を訓練
        y = model.dsc.forward(imgs_fake)
        model.loss_function.forward(y, 1)               # 偽を本物と偽る
        gy = model.loss_function.backward()
        gx = model.dsc.backward(gy)
        gx = model.gen.backward(gx)          # 識別器からもらった勾配を生成器に逆伝播
        model.gen.update(eta=eta)

    # 誤差を測定して一定間隔で表示
    err_real, acc_real = mesurement_real(imgs_real, 1)
    err_fake, acc_fake = mesurement_fake(imgs_fake, 0)

    if i % interval == 0:
        print('{:5d} {:5.1f} Error real{:6.3f} fake{:6.3f} | Accuracy real{:6.3f} fake{:6.3f}' ¥
           .format(i,  time.time()-start, err_real, err_fake, acc_real, acc_fake))
        start = time.time()

err_real_rec, acc_real_rec = mesurement_real.progress()
err_fake_rec, acc_fake_rec = mesurement_fake.progress()
cf.graph_for_error(err_real_rec, acc_real_rec, err_fake_rec, acc_fake_rec, ¥
        label=('error_real', 'accuracy_real', 'error_fake', 'accuracy_fake'))
cf.generate_random_images(model.gen.forward, nz, 1, 8, 8, 256)
```

学習を始める前に記録用の変数を確保するなどしてから、リスト 4.7.6、リスト 4.7.7、リスト 4.8.5 と同様に二重の for ループで学習を進める．GAN の学習の進め方はいろいろ考えられ、なかなか難しい．それ故にさまざまなことが言われているが、とりあえずやってみて、うまく結果が得られた方法を示しておくから、是非いろいろやってみてほしい．とまれここでは、識別器の訓練と生成器の訓練に分け、以下の手順で学習させる．

すなわち、識別器の訓練の際に、確率的に 0 と 1 が半々になるように乱数で作った正解値 t を用意する．いっぽう、データで与えられる本物画像と、生成器が乱数から作った偽画像も用意しておく．そして正解値 t を用いて、本物画像と偽画像を選択して識別器の入力とし、識別器の出力 y を正解値 t と照合して損失を求める．あとは逆伝播して勾配を求めて識別器を更新する．

続いて生成器の訓練だが、先に用意した生成器による偽画像を、今度は全部識別器に入れて出力を求める．そして損失を求める際に、正解値は偽画像だから'0'であるべきところを'1'とする．そして識別器を逆伝播して、恰も識別器を騙すかのようにして得られる勾配を、生成器に伝えて逆伝播し、その勾配により生成器を更新する．

これを繰り返して、識別器と生成器の両方を学習させる．そして以上の過程で得られる損失とは別に、エポック毎に本物画像のと偽画像のそれぞれについてエラーと正解率を測定する．すなわち前者の本物画像では本物を本物と判別する識別器の能力が観測される．いっぽう後者の偽画像では偽物を偽物と判別する識別器の能力を観測し、前者と併せて識別器の能力となる．しかし識別能力が高いのに偽画像を入力して本物と誤るというのは、生成器が本物に近い画像を生成できているということにほかならない．だから両者の拮抗する状態こそが GAN がうまく学習していることの証になる．訓練の際にも損失は得られるが、解釈が難しいこともあり、エポック毎に集約することも兼ねてこうした．

学習が終わると途中経過をグラフ表示し、最後に common_function.py に用意した関数 generate_random_images()で生成器の能力を確かめて終わる．この関数は後で示す．

[補足：GAN の学習について]

GAN の生成器の学習の際に $t = 1$ としているが、二値の交差エントロピー誤差を表す式 4.9.6 に $t = 1$ を入れれば、$L = -\frac{1}{k}\sum \log y$ となる．これを最小化することは、符号の取れた $\frac{1}{k}\sum \log y$ を最大化することにほかならない．このことは GAN の最初の論文で述べられていることとも矛盾しないと考えられる．詳しくは以下の論文を参照されたし．

Generative Adversarial Nets, Ian J.Goodfellow, et al. 2014

では一旦ここでリスト 4.9.8 とリスト 4.9.9 の GAN の学習の経過のグラフを示そう.

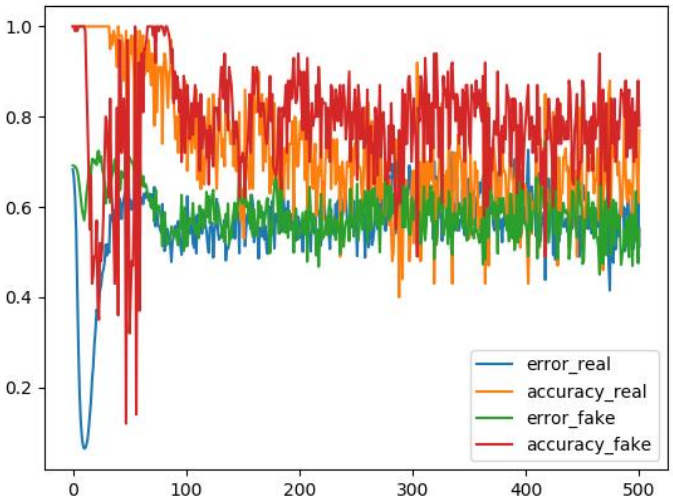

図 4.9.4 GAN の学習経過～その 1

図 4.9.4 は、GAN の学習の経過の一例になるが、見たとおりのぐしゃぐしゃのグラフとなった. とくに 100 エポックあたりまでは酷い. ここままではよくわからないので、リスト 4.9.7 の progress()メソッドで作った移動平均を使って描くと次のグラフになる.

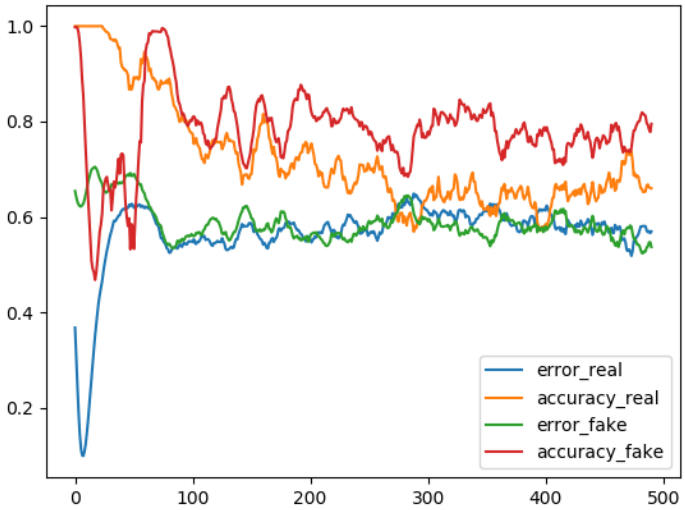

図 4.9.5 GAN の学習経過～その 2

図 4.9.5 では全体の流れがつかみやすく、以下のように読み取れる.
本物画像に対するエラーerror_real は、一旦小さくなってすぐに上昇し、偽画像のエラーerror_fake と 0.6 あたりで重なると、それからジグザグを繰り返しながらも安定している.
そして本物画像の正解率 accuracy_real は当初の 1.0 から少しずつ下がっているいっぽう、偽画像の正解率 accuracy_fake は、はじめ大きく変動する. しかし、100 エポックあたりから両者が少し絡むようになっている.
これらの動きは、はじめ識別器は識別能力がなく、生成器も無意味なノイズしか出力できないこと、そして次第に識別器が識別能力を獲得していき、それを追うように生成器が画像を生成するようになって、やがて両者が拮抗していく、まさにその GAN の成長過程を示していると言えるのではないだろうか？

では後回しにしておいた画像を目で見て確認するための関数を用意しよう.

リスト 4.9.10：乱数から画像を生成

```
def generate_random_images(func, nz, C, Ih, Iw, n=81, reverse=False, rate=1.0):
    # 画像の生成
    n_rows = int(n ** 0.5)  # 行数
    n_cols = n // n_rows    # 列数
    # 入力となるノイズ
    noise = np.random.randn(n_rows*n_cols, nz) * rate
    # 画像を生成して 0-1 の範囲に調整
    if C <= 1:
        g_imgs = func(noise).reshape(n, Ih, Iw)
    else:
        g_imgs = func(noise).reshape(n, C, Ih, Iw).transpose(0, 2, 3, 1)
    g_imgs = (g_imgs - np.min(g_imgs)) / (np.max(g_imgs) - np.min(g_imgs))
    g_imgs = 1 - g_imgs if reverse==True else g_imgs
    Ih_spaced = Ih + 2; Iw_spaced = Iw + 2
    if C <= 1:
        matrix_image = np.empty((Ih_spaced*n_rows, Iw_spaced*n_cols))    # 全体の画像
    else:
        matrix_image = np.empty((Ih_spaced*n_rows, Iw_spaced*n_cols, C)) # 全体の画像
    matrix_image[...] = 1.0 if reverse==True else 0.0
    #  生成された画像を並べて一枚の画像にする
    for i in range(n_rows):
        for j in range(n_cols):
            g_img = g_imgs[i*n_cols + j]
            top  = i*Ih_spaced
            left = j*Iw_spaced
            matrix_image[top : top+Ih, left : left+Iw] = g_img

    plt.figure(figsize=(9, 9))
    if C <= 1:
        plt.imshow(matrix_image.tolist(), cmap='Greys_r')
    else:
        plt.imshow(matrix_image.tolist())
    plt.tick_params(labelbottom=False, labelleft=False, bottom=False, left=False)
    plt.show()
```

リスト 4.9.10 の関数 generate_random_images()は、カラー画像を含めて、ある程度汎用的に使えるようにしておいた. 引数 func は生成器の順伝播、nz は画像の元となる入力ノイズの大きさ、C, Ih, Iw は生成画像のチャネル数と縦横の大きさだ. そして n=81 は生成画像を並べる数で、それを縦横に並べるから n_rows, n_cols で正方形に近くなるように切り出す. reverse=False は画像の白黒反転、rate=1.0 は入力ノイズの広がりの調整だ. 入力ノイズは nz の大きさのノイズを n_rows×n_cols 個、すなわち(n_rows*n_cols, nz)の形で用意して、表示する n 個の画像をいっぺんに生成する. 画像の画素の値の調整をした上で、生成画像から取り出して、空行列で用意した matrix_image に順に並べていく. このとき見やすさのために画

像に値'2'の隙間をあけている．最後に白黒ならグレースケールの指定をして、ラベルやメモリを消して表示する.

ではリスト 4.9.8、リスト 4.9.9 の GAN の学習の結果を、リスト 4.9.10 を使ってみてみよう.

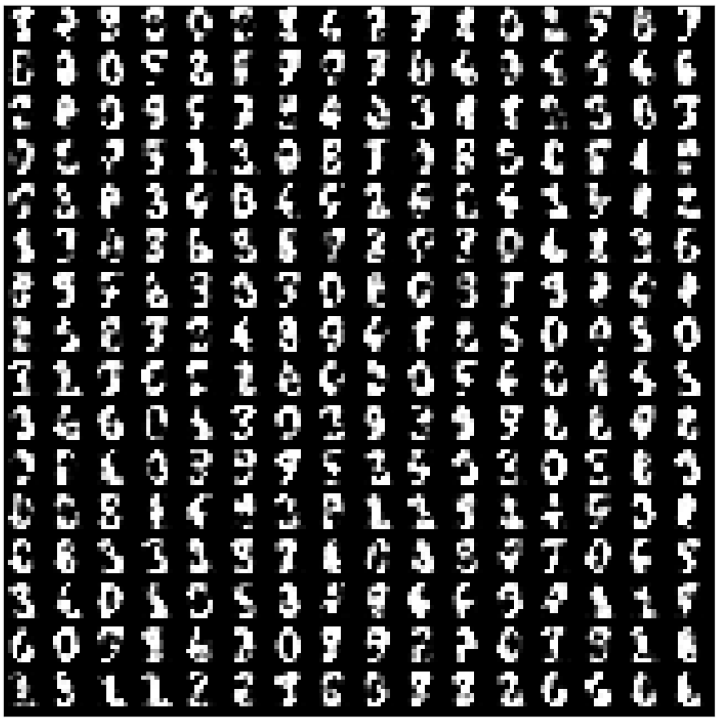

図 4.9.6 GAN0 が生成した手書き数字画像

どうだろうか？ 図 4.5.2 で示した本物画像の手書き数字と比べて、劣化は認められるものの、ほぼそれらしい画像が得られているのではないだろうか？
なによりも、この生成の過程では、もはや本物画像は参照しておらず、純粋に乱数から作ったノイズをもとにしているのだ.
最後に、同じプログラムで学習に使うデータを図 4.5.3 に示した顔画像に差し替えた場合の結果を示しておこう．これはぜひ自身でやってみてほしい．感想は皆さんにお任せする.

図 4.9.7 GAN0 が生成した顔画像

ここまで長い道のりとなってしまった．しかし数式の裏付けも含め、肝心なところを何一つ隠さず、すべて追えるようにしたつもりだ．この真にホワイトボックスといえる手作りのニューラルネットワークで、画像識別から GAN に至るまで取り組んできた．まだこの先やらねばならないことはたくさんあるが、冒頭に述べたように、これから先への足掛かりが得られたならば幸いだ．

ファイル	クラス 関数	メソッド	リスト番号
Neuron.py			
	class Config		4.1.8
	class BaseLayer		
		def __init__()	4.1.1
		def init_parameter()	4.1.7
		def update()	4.4.4
	class NeuronLayer		
		def __init__()	4.1.4
		def fix_configuration()	4.1.6
		def forward()	4.1.5
		def backward()	4.4.3
NN.py			
	class NN_CNN_Base		
		def __init__()	4.6.1
		def forward()	4.6.2
		def backward()	4.6.3
		def summary()	4.6.5
		def update()	4.6.4
	class NN_0		
		def __init__()	4.6.6
	class NN_m		
		def __init__()	4.6.12
GAN0.py			
	class GAN_m_m		
		def __init__()	4.9.6

ファイル	クラス 関数	メソッド	リスト番号
Activators.py			
	class Identity		
		def forward()	4.1.2
		def backward()	4.4.2
	class Step		
		def __init__()	4.1.9
		def backward()	4.1.9
	class Sigmoid		
		def forward()	4.6.8
		def backward()	4.6.8
	class Softmax		
		def forward()	4.7.3
		def backward()	4.7.3
	class ReLU		
		def forward()	4.9.1
		def backward()	4.9.1
	class LReLU		
		def __init__()	4.9.2
		def forward()	4.9.2
		def backward()	4.9.2
	class Tanh		
		def forward()	4.9.4
		def backward()	4.9.4
LossFunctions.py			
	class MeanSquaredError		
		def forward()	4.3.1→4.4.1
		def backward()	4.4.1
	class CrossEntropyError		
		def forward()	4.7.4
		def backward()	4.7.4
	class CrossEntropyError2		
		def forward()	4.9.5
		def backward()	4.9.5

ファイル	クラス 関数	メソッド	リスト番号
common_function.py			
	def eval_in_module()		4.1.3
	def get_accuracy()		4.4.5
	def graph_for_error()		4.4.6
	def test_sample()		4.5.3
	def split_train_test()		4.5.3
	def normalize()		4.7.5
	class Mesurement		
		def __init__()	4.7.2
		def __call__()	4.7.2
		def progress()	4.7.2
	def mesurement()		4.7.2
	class Mesurement_for_GAN		
		def __init__()	4.9.7
		def __call__()	4.9.7
		def progress()	4.9.7
	def generate_random_images()		4.9.10

ファイル	クラス 関数	メソッド	リスト番号
Iris.py			
	def get_data()		4.2.1
	def label_list()		4.2.1
Digits.py			
	def get_data()		4.5.1
	def show_sample()		4.5.1
OlivettiFaces.py			
	def get_data()		4.5.2
	def label_list()		4.5.2
	def show_sample()		4.5.2
CIFER10.py			
	def unpickle()		4.8.1
	def read_batch()		4.8.1
	def load_data()		4.8.1
	def get_data()		4.8.4
	def label_list()		4.8.2
	def show_sample()		4.8.3