

# 深層学習 day1

## 1 入力層～中間層

$\mathbf{W}$  を  $m \times d$  行列、 $\mathbf{b} \in \mathbb{R}^d$  とし、 $f: \mathbb{R}^d \rightarrow \mathbb{R}^d$  を関数とする。このとき、関数  $FN(\mathbf{W}, \mathbf{b}, f): \mathbb{R}^m \rightarrow \mathbb{R}^d$  を

$$FN(\mathbf{W}, \mathbf{b}, f)(\mathbf{x}) = f(\mathbf{W}\mathbf{x}^T + \mathbf{b})$$

で定義する。 $FN(\mathbf{W}, \mathbf{b}, f)$  を順伝搬型ニューラルネットワーク (**feedforward neural network**) といい、 $\mathbf{W}$  を  $FN(\mathbf{W}, \mathbf{b}, f)$  の重み、 $\mathbf{b}$  を  $FN(\mathbf{W}, \mathbf{b}, f)$  のバイアス、 $f$  を  $FN(\mathbf{W}, \mathbf{b}, f)$  の活性化関数という。 $FN(\mathbf{W}^{(1)}, \mathbf{b}^{(1)}, f_1), \dots, FN(\mathbf{W}^{(n)}, \mathbf{b}^{(n)}, f_n)$  を順伝搬型ニューラルネットワークとし、 $FN(\mathbf{W}^{(l)}, \mathbf{b}^{(l)}, f_l)$  の値域の次元が  $FN(\mathbf{W}^{(l+1)}, \mathbf{b}^{(l+1)}, f_{l+1})$  の定義域の次元に等しいとする。このとき、 $m$  を  $FN(\mathbf{W}^{(1)}, \mathbf{b}^{(1)}, f_1)$  の定義域の次元、 $d$  を  $FN(\mathbf{W}^{(n)}, \mathbf{b}^{(n)}, f_n)$  の値域の次元とし、関数  $FN(\mathbf{W}^{(1)}, \dots, \mathbf{W}^{(n)}, \mathbf{b}^{(1)}, \dots, \mathbf{b}^{(n)}, f_1, \dots, f_n): \mathbb{R}^m \rightarrow \mathbb{R}^d$  を

$$FN(\mathbf{W}^{(1)}, \dots, \mathbf{W}^{(n)}, \mathbf{b}^{(1)}, \dots, \mathbf{b}^{(n)}, f_1, \dots, f_n)(\mathbf{x}) := FN(\mathbf{W}^{(n)}, \mathbf{b}^{(n)}, f_n)(\dots(FN(\mathbf{W}^{(1)}, \mathbf{b}^{(1)}, f_1)(\mathbf{x}))\dots)$$

で定義する。 $FN(\mathbf{W}^{(1)}, \dots, \mathbf{W}^{(n)}, \mathbf{b}^{(1)}, \dots, \mathbf{b}^{(n)}, f_1, \dots, f_n)$  を多層順伝搬型ニューラルネットワークという。

例 1.1. (コードは FN.ipynb)

```
import numpy as np

# W, b を設定
W = np.array([[1, 2, 3], [4, 5, 6]])
b = np.array([7, 8])

# f を設定
def f(x):
    return 2*x

# FN を設定
def FN(W, b, f, x):
    x = x.T
    Z = np.dot(W, x) + b
    return f(Z)
```

```
#出力
x = np.array([1,2,3])
FN(W,b,f,x)
>array([42, 80])
```

## 2 活性化関数

順伝搬型ニューラルネットワーク  $FN(\mathbf{W}, \mathbf{b}, f)$  を考える。 $FN(\mathbf{W}, \mathbf{b}, f)$  の活性化関数  $f$  には単調増加する非線形関数が用いられることが多い。ここでは、いくつかの活性化関数を紹介する。

- (1) シグモイド関数  $\sigma$  は  $\sigma(x) = 1/(1 + e^{-x})$  で定義される。
- (2) 双曲線正接関数  $\tanh$
- (3) **ReLU (Rectified Linear Unit)** 関数は  $\text{ReLU}(x) = \max(x, 0)$  で定義される。

例 2.1. (コードは活性化関数.ipynb)

```
import numpy as np

# W,b を設定
W = np.random.randn(2,3)
b = np.random.randn(1,2)
W,b
>(array([[ -0.3275861 ,  0.35679815, -0.93635564],
        [ -0.3966737 ,  2.45340843,  0.80369067]]),
 array([[ 0.73728049, -0.99336306]]))

# f を設定
def f(x):
    return 1/1+np.exp(-x)

# FN を設定
def FN(W,b,f,x):
    x = x.T
    Z = np.dot(W,x)+b
    return f(Z)

#出力
x = np.array([0.1,0.2,0.3])
FN(W,b,f,x)
```

```
>array([[1.60958626, 2.35153575]])
```

### 3 出力層

$n$  を自然数とし、 $\mathbf{x} = (x_1, \dots, x_n) \in \mathbb{R}^n$  とする。ある  $1 \leq i \leq n$  が存在して、 $x_i = 1$  であり、他の全ての  $1 \leq j \leq n$  に対して、 $x_j = 0$  であるとき、 $\mathbf{x}$  を  $n$  次元 **one-hot** ベクトルという。 $\{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^N$  をラベル付けされたデータとし、 $N$  をデータの数、 $\mathbf{x}_i$  を  $m$  次元特徴ベクトル、 $\mathbf{y}_i$  を  $\mathbf{x}_i$  のラベルとする。多層順伝搬型ニューラルネットワークを用いて、未知の  $m$  次元特徴ベクトル  $\mathbf{x}$  に対して、 $\mathbf{x}$  のラベル  $\mathbf{y}$  を予測することを試みる。 $FN(\mathbf{W}^{(1)}, \dots, \mathbf{W}^{(n)}, \mathbf{b}^{(1)}, \dots, \mathbf{b}^{(n)}, f_1, \dots, f_n) : \mathbb{R}^m \rightarrow \mathbb{R}^d$  を多層順伝搬型ニューラルネットワークとする。このとき、 $\mathbf{W}^{(n+1)}$  を  $d \times k$  行列、 $\mathbf{b}^{(n+1)} \in \mathbb{R}^k$  とし、 $f_{n+1} : \mathbb{R}^k \rightarrow \mathbb{R}^l$  を関数とする。順伝搬型ニューラルネットワーク  $FN(\mathbf{W}^{(n+1)}, \mathbf{b}^{(n+1)}, f_{n+1}) : \mathbb{R}^d \rightarrow \mathbb{R}^l$  を以下のように定義する。 $\mathbf{y}_i$  の値が任意の実数値をとる場合と、 $\mathbf{y}_i$  が  $r$  次元 one-hot ベクトルの値しかとらない場合で定義が異なる。

- (1)  $\mathbf{y}_i$  の値が任意の実数値をとる場合： $k = l = 1$  とし、 $f_{n+1} : \mathbb{R} \rightarrow \mathbb{R}$  を恒等関数とする。
- (2)  $\mathbf{y}_i$  が  $r$  次元 one-hot ベクトルの値しかとらない場合： $l = r$  とし、

$$f_{n+1}(z_1, \dots, z_r) = (e^{z_1} / \sum_{i=1}^r e^{z_i}, \dots, e^{z_r} / \sum_{i=1}^r e^{z_i})$$

と定義する。 $f_{n+1}$  をソフトマックス関数という。

$FN(\mathbf{W}^{(n+1)}, \mathbf{b}^{(n+1)}, f_{n+1})$  を  $FN(\mathbf{W}^{(1)}, \dots, \mathbf{W}^{(n)}, \mathbf{b}^{(1)}, \dots, \mathbf{b}^{(n)}, f_1, \dots, f_n)$  の出力層という。結果として、多層順伝搬型ニューラルネットワーク  $FN(\mathbf{W}^{(1)}, \dots, \mathbf{W}^{(n+1)}, \mathbf{b}^{(1)}, \dots, \mathbf{b}^{(n+1)}, f_1, \dots, f_{n+1}) : \mathbb{R}^m \rightarrow \mathbb{R}^l$  を得る。 $\mathbf{y} = FN(\mathbf{W}^{(1)}, \dots, \mathbf{W}^{(n+1)}, \mathbf{b}^{(1)}, \dots, \mathbf{b}^{(n+1)}, f_1, \dots, f_{n+1})$  とおく。未知の  $m$  次元特徴ベクトル  $\mathbf{x}$  に対して、 $\mathbf{x}$  のラベル  $\mathbf{y} = \mathbf{y}(\mathbf{x})$  を予測することが目的であるから、最適な  $\mathbf{W}^{(1)}, \dots, \mathbf{W}^{(n+1)}, \mathbf{b}^{(1)}, \dots, \mathbf{b}^{(n+1)}$  は以下の方法で求めることができる。まず、誤差関数を以下のように定義する。 $\mathbf{y}_i$  の値が任意の実数値をとる場合と、 $\mathbf{y}_i$  が  $r$  次元 one-hot ベクトルの値しかとらない場合で定義が異なる。

- (1)  $\mathbf{y}_i$  の値が任意の実数値をとる場合： $1 \leq i \leq n$  に対して、

$$E_i(\mathbf{W}^{(1)}, \dots, \mathbf{W}^{(n+1)}, \mathbf{b}^{(1)}, \dots, \mathbf{b}^{(n+1)}) = \frac{1}{2} \|\mathbf{y}_i - \mathbf{y}(\mathbf{x}_i)\|^2$$

と定義し、

$$E(\mathbf{W}^{(1)}, \dots, \mathbf{W}^{(n+1)}, \mathbf{b}^{(1)}, \dots, \mathbf{b}^{(n+1)}) = \sum_{i=1}^N E_i(\mathbf{W}^{(1)}, \dots, \mathbf{W}^{(n+1)}, \mathbf{b}^{(1)}, \dots, \mathbf{b}^{(n+1)})$$

と定義する。

- (2)  $\mathbf{y}_i$  が  $r$  次元 one-hot ベクトルの値しかとらない場合： $\mathbf{y}_i = (y_{i1}, \dots, y_{ir})$ ,  $\mathbf{y}(\mathbf{x}_i) = (\mathbf{y}(\mathbf{x}_i)_1, \dots, \mathbf{y}(\mathbf{x}_i)_r)$  とおき、 $1 \leq i \leq n$  に対して、

$$E_i(\mathbf{W}^{(1)}, \dots, \mathbf{W}^{(n+1)}, \mathbf{b}^{(1)}, \dots, \mathbf{b}^{(n+1)}) = - \sum_{j=1}^r y_{ij} \log \mathbf{y}(\mathbf{x}_i)_j$$

と定義し、

$$E(\mathbf{W}^{(1)}, \dots, \mathbf{W}^{(n+1)}, \mathbf{b}^{(1)}, \dots, \mathbf{b}^{(n+1)}) = \sum_{i=1}^N E_i(\mathbf{W}^{(1)}, \dots, \mathbf{W}^{(n+1)}, \mathbf{b}^{(1)}, \dots, \mathbf{b}^{(n+1)})$$

と定義する。

このとき、最適な  $\mathbf{W}^{(1)}, \dots, \mathbf{W}^{(n+1)}, \mathbf{b}^{(1)}, \dots, \mathbf{b}^{(n+1)}$  は

$$\min_{\mathbf{W}^{(1)}, \dots, \mathbf{W}^{(n+1)}, \mathbf{b}^{(1)}, \dots, \mathbf{b}^{(n+1)}} E(\mathbf{W}^{(1)}, \dots, \mathbf{W}^{(n+1)}, \mathbf{b}^{(1)}, \dots, \mathbf{b}^{(n+1)})$$

を解くことによって得られる。

例 3.1. (コードは出力層.ipynb)

```
import numpy as np

# W_1, b_1 を設定
W_1 = np.random.randn(2,3)
b_1 = np.random.randn(1,2)
W_1, b_1
>(array([[ -0.85226586,  -0.74838968,  -0.47477658],
         [ 0.9893178 ,  -0.17452691,   0.08049864]]),
 array([[ -0.1930833 ,  -1.26376165]]))

# f_1 を設定
def f_1(x):
    return 1/(1+np.exp(-x))

# FN を設定
def FN(W,b,f,x):
    x = x.T
    Z = np.dot(W,x)+b
    return f(Z)

# W_2, b_2 を設定
W_2= np.random.randn(1,2)
b_2= np.random.randn()
W_2, b_2
>(array([[ -0.79262461,  -1.11520968]]), 1.4734070700523327)

#f_2 を設定
def f_2(x):
    return x
```

```
#出力
x = np.array([0.1,0.2,0.3])
x = FN(W_1,b_1,f_1,x)
FN(W_2,b_2,f_2,x)
>array([[ -5.44991305]])
```

## 4 勾配降下法

$\{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^N$  をラベル付けされたデータとし、 $N$  をデータの数、 $\mathbf{x}_i$  を  $m$  次元特徴ベクトル、 $\mathbf{y}_i$  を  $\mathbf{x}_i$  のラベルとする。多層順伝搬型ニューラルネットワークを用いて、未知の  $m$  次元特徴ベクトル  $\mathbf{x}$  に対して、 $\mathbf{x}$  のラベル  $\mathbf{y}$  を予測することを試みる。 $FN(\mathbf{W}^{(1)}, \dots, \mathbf{W}^{(n)}, \mathbf{b}^{(1)}, \dots, \mathbf{b}^{(n)}, f_1, \dots, f_n) : \mathbb{R}^m \rightarrow \mathbb{R}^d$  を多層順伝搬型ニューラルネットワークとする。3 章で述べたように、多層順伝搬型ニューラルネットワーク  $FN(\mathbf{W}^{(1)}, \dots, \mathbf{W}^{(n+1)}, \mathbf{b}^{(1)}, \dots, \mathbf{b}^{(n+1)}, f_1, \dots, f_{n+1}) : \mathbb{R}^m \rightarrow \mathbb{R}^l$  を構成し、誤差関数  $E(\mathbf{W}^{(1)}, \dots, \mathbf{W}^{(n+1)}, \mathbf{b}^{(1)}, \dots, \mathbf{b}^{(n+1)})$  を構成する。記法の簡略化のため、行列のリスト  $\mathbf{W}^{(1)}, \dots, \mathbf{W}^{(n+1)}, \mathbf{b}^{(1)}, \dots, \mathbf{b}^{(n+1)}$  を  $\mathbf{W}$  で略記する。3 章で述べたように、最適な  $\mathbf{W}$  は

$$\min_{\mathbf{W}} E(\mathbf{W}) \quad (1)$$

解くことによって求まる。勾配降下法とは、(1) を解くためのアルゴリズムであり、以下のように定義される。

- (1) ある正の正数  $\epsilon$  をとる。この  $\epsilon$  を学習率という。
- (2)  $\mathbf{W}$  を適当に決める。
- (3)  $\nabla E(\mathbf{W})$  を計算する。
- (4)  $\mathbf{W}$  を  $\mathbf{W} - \epsilon \nabla E(\mathbf{W})$  で置き換える。
- (5) (3) に戻る。

誤差関数  $E(\mathbf{W})$  は一般に下に凸ではないので、勾配降下法の収束値は必ずしも大域的極小値ではなく、局所的極小値である可能性がある。そのため、勾配降下法にはいくつかの改良版が存在する。それらをここで紹介しておく。

- (a) 確率的勾配降下法：このアルゴリズムは以下のように定義される。

- (1) 学習係数  $\epsilon$  をとる。
- (2)  $\mathbf{W}$  を適当に決める。
- (3) 無作為に  $\mathbf{x}_i$  をとる。
- (4)  $\nabla E_i(\mathbf{W})$  を計算する。
- (5)  $\mathbf{W}$  を  $\mathbf{W} - \epsilon \nabla E_i(\mathbf{W})$  で置き換える。
- (6) (3) に戻る。

確率的勾配降下法の利点は以下である。

- (i) データが冗長な場合の計算コストの削減
- (ii) 望まない局所的極小値に収束するリスクの削減
- (iii) オンライン学習ができる。

(b) ミニバッチ勾配降下法：このアルゴリズムは以下のように定義される。

- (1) 学習係数  $\epsilon$  をとる。
- (2) データ集合  $\{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^N$  をいくつかのミニバッチ  $B_1, \dots, B_k$  に分割する。
- (3)  $\mathbf{W}$  を適当に決める。
- (4) 無作為に  $B_j$  をとる。
- (5)  $\nabla \frac{1}{|B_j|} \sum_{\mathbf{x}_i \in B_j} E_i(\mathbf{W})$  を計算する。
- (6)  $\mathbf{W}$  を  $\mathbf{W} - \epsilon \nabla \frac{1}{|B_j|} \sum_{\mathbf{x}_i \in B_j} E_i(\mathbf{W})$  で置き換える。
- (7) (4) に戻る。

ミニバッチ勾配降下法の利点は、確率的勾配降下法のメリットを損なわず、計算機の計算資源を有効利用できる点にある。

例 4.1. (コードは勾配降下法.ipynb) mnist データを用いて、ミニバッチ勾配降下法を実装する。

```
from common.functions import *
from common.gradient import numerical_gradient
import numpy as np

# 2層のニューラルネットを設定
class TwoLayerNet:

    def __init__(self, input_size, hidden_size, output_size, weight_init_std=0.01):
        # 重みの初期化
        self.params = {}
        self.params['W1'] = weight_init_std * np.random.randn(input_size, hidden_size)
        self.params['b1'] = np.zeros(hidden_size)
        self.params['W2'] = weight_init_std * np.random.randn(hidden_size, output_size)
        self.params['b2'] = np.zeros(output_size)

    def predict(self, x):
        W1, W2 = self.params['W1'], self.params['W2']
        b1, b2 = self.params['b1'], self.params['b2']

        a1 = np.dot(x, W1) + b1
        z1 = sigmoid(a1)
        a2 = np.dot(z1, W2) + b2
        y = softmax(a2)

        return y

# x:入力データ, t:教師データ
def loss(self, x, t):
```

```

y = self.predict(x)

return cross_entropy_error(y, t)

def accuracy(self, x, t):
    y = self.predict(x)
    y = np.argmax(y, axis=1)
    t = np.argmax(t, axis=1)

    accuracy = np.sum(y == t) / float(x.shape[0])
    return accuracy

# x:入力データ, t:教師データ
def numerical_gradient(self, x, t):
    loss_W = lambda W: self.loss(x, t)

    grads = {}
    grads['W1'] = numerical_gradient(loss_W, self.params['W1'])
    grads['b1'] = numerical_gradient(loss_W, self.params['b1'])
    grads['W2'] = numerical_gradient(loss_W, self.params['W2'])
    grads['b2'] = numerical_gradient(loss_W, self.params['b2'])

    return grads

def gradient(self, x, t):
    W1, W2 = self.params['W1'], self.params['W2']
    b1, b2 = self.params['b1'], self.params['b2']
    grads = {}

    batch_num = x.shape[0]

    # forward
    a1 = np.dot(x, W1) + b1
    z1 = sigmoid(a1)
    a2 = np.dot(z1, W2) + b2
    y = softmax(a2)

    # backward
    dy = (y - t) / batch_num
    grads['W2'] = np.dot(z1.T, dy)

```

```

    grads['b2'] = np.sum(dy, axis=0)

    dz1 = np.dot(dy, W2.T)
    da1 = sigmoid_grad(a1) * dz1
    grads['W1'] = np.dot(x.T, da1)
    grads['b1'] = np.sum(da1, axis=0)

    return grads

#ミニバッチ勾配降下法を実装
from dataset.mnist import load_mnist
import matplotlib.pyplot as plt

# データの読み込み
(x_train, t_train), (x_test, t_test) = load_mnist(normalize=True, one_hot_label=True)

network = TwoLayerNet(input_size=784, hidden_size=50, output_size=10)

iters_num = 1000 # 繰り返しの回数を適宜設定する
train_size = x_train.shape[0]
batch_size = 100
learning_rate = 0.1

train_loss_list = []
train_acc_list = []
test_acc_list = []

iter_per_epoch = max(train_size / batch_size, 1)

for i in range(iters_num):
    batch_mask = np.random.choice(train_size, batch_size)
    x_batch = x_train[batch_mask]
    t_batch = t_train[batch_mask]

    # 勾配の計算
    #grad = network.numerical_gradient(x_batch, t_batch)
    grad = network.gradient(x_batch, t_batch)

    # パラメータの更新
    for key in ('W1', 'b1', 'W2', 'b2'):

```



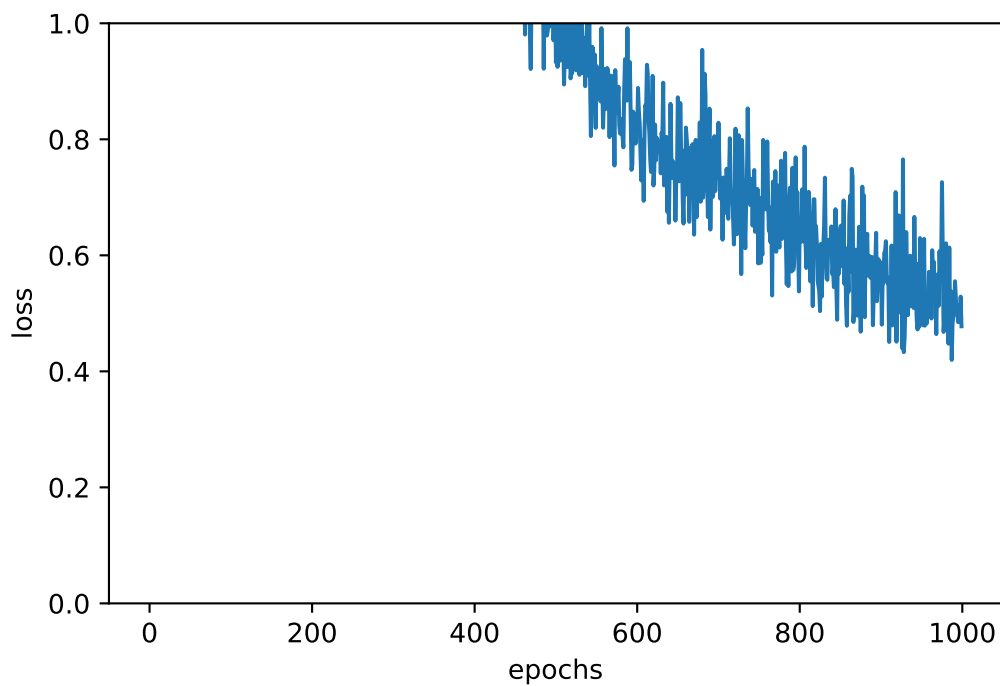
```

network.params[key] -= learning_rate * grad[key]

loss = network.loss(x_batch, t_batch)
train_loss_list.append(loss)

# グラフの描画
x = np.arange(len(train_loss_list))
plt.plot(x, train_loss_list)
plt.xlabel("epochs")
plt.ylabel("loss")
plt.ylim(0, 1.0)
plt.show()

```



## 5 誤差逆伝搬法

$\{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^N$  をラベル付けされたデータとし、 $N$  をデータの数、 $\mathbf{x}_i$  を  $m$  次元特徴ベクトル、 $\mathbf{y}_i$  を  $\mathbf{x}_i$  のラベルとする。多層順伝搬型ニューラルネットワークを用いて、未知の  $m$  次元特徴ベクトル  $\mathbf{x}$  に対して、 $\mathbf{x}$  のラベル  $\mathbf{y}$  を予測することを試みる。 $FN(\mathbf{W}^{(1)}, \dots, \mathbf{W}^{(n)}, \mathbf{b}^{(1)}, \dots, \mathbf{b}^{(n)}, f_1, \dots, f_n) : \mathbb{R}^m \rightarrow \mathbb{R}^d$  を多層順伝搬型ニューラルネットワークとする。3 章で述べたように、多層順伝搬型ニューラルネットワーク  $FN(\mathbf{W}^{(1)}, \dots, \mathbf{W}^{(n+1)}, \mathbf{b}^{(1)}, \dots, \mathbf{b}^{(n+1)}, f_1, \dots, f_{n+1}) : \mathbb{R}^m \rightarrow \mathbb{R}^l$  を構成し、誤

差関数  $E(\mathbf{W}^{(1)}, \dots, \mathbf{W}^{(n+1)}, \mathbf{b}^{(1)}, \dots, \mathbf{b}^{(n+1)})$  を構成する。記法の簡略化のため、行列のリスト  $\mathbf{W}^{(1)}, \dots, \mathbf{W}^{(n+1)}, \mathbf{b}^{(1)}, \dots, \mathbf{b}^{(n+1)}$  を  $\mathbf{W}$  で略記する。表記を簡素化するために、 $\mathbf{b}^{(1)} = \dots = \mathbf{b}^{(n+1)} = 0$  とする。このように仮定しても一般性を失わない ([6], p42 を参照) からである。3 章で述べたように、最適な  $\mathbf{W}$  は

$$\min_{\mathbf{W}} E(\mathbf{W}) \quad (2)$$

解くことによって求まる。4 章では、(2) を勾配降下法（確率的勾配降下法、ミニバッチ勾配降下法）で求める方法について述べたが、これを行うためには、全ての  $1 \leq p \leq N$  に対して、 $\nabla E_p(\mathbf{W}) = \partial E_p(\mathbf{W}) / \mathbf{W}$  を計算する必要がある。これを行うためのアルゴリズムが誤差逆伝搬法である。このアルゴリズムは以下のように定義される。

- (1)  $1 \leq l \leq n+1$  に対して、 $\mathbf{u}^{(l)} = \mathbf{W}^{(l)} \mathbf{x}_p + \mathbf{b}^{(l)}$ ,  $\mathbf{z}^{(l)} = f_l(\mathbf{u}^{(l)})$  とおく。 $\mathbf{u}^{(l)}, \mathbf{z}^{(l)}$  の第  $j$  成分をそれぞれ  $u_j^{(l)}, z_j^{(l)}$  で表すことにする。
- (2)  $1 \leq l \leq n+1$  に対して、 $\mathbf{W}^{(l)}$  の代  $(j, i)$  成分を  $w_{ji}^{(l)}$  で表すことにする。
- (3) 全ての  $1 \leq l \leq n+1$  に対して、 $\mathbf{u}^{(l)}, \mathbf{z}^{(l)}$  を計算する。
- (4)  $\delta_j^{(n+1)} = \partial E_p / \partial u_j^{(n+1)}$  を計算する。
- (5)  $1 \leq l \leq n$  に対して、 $\delta_j^{(l)} = \sum_k \delta_k^{(n+1)} (w_{kj}^{(l+1)} f'_l(u_j^{(l)}))$  を順番に計算する。
- (6)  $\partial E_p / \partial w_{ji}^{(l)} = \delta_j^{(l)} z_i^{(l-1)}$  とおき、 $\nabla E_p(\mathbf{W}) = (\partial E_p / \partial w_{ji}^{(l)})$  とおく。

例 5.1. (コードは誤差逆伝搬法.ipynb)

```
import numpy as np

#関数を設定

class Relu:
    def __init__(self):
        self.mask = None

    def forward(self, x):
        self.mask = (x <= 0)
        out = x.copy()
        out[self.mask] = 0

        return out

    def backward(self, dout):
        dout[self.mask] = 0
        dx = dout

        return dx
```

```

class Affine:
    def __init__(self, W, b):
        self.W = W
        self.b = b

        self.x = None
        self.original_x_shape = None
        # 重み・バイアスパラメータの微分
        self.dW = None
        self.db = None

    def forward(self, x):
        # テンソル対応
        self.original_x_shape = x.shape
        x = x.reshape(x.shape[0], -1)
        self.x = x

        out = np.dot(self.x, self.W) + self.b

        return out

    def backward(self, dout):
        dx = np.dot(dout, self.W.T)
        self.dW = np.dot(self.x.T, dout)
        self.db = np.sum(dout, axis=0)

        dx = dx.reshape(*self.original_x_shape) # 入力データの形状に戻す (テンソル対応)
        return dx

class SoftmaxWithLoss:
    def __init__(self):
        self.loss = None
        self.y = None # softmax の出力
        self.t = None # 教師データ

    def forward(self, x, t):
        self.t = t
        self.y = softmax(x)
        self.loss = cross_entropy_error(self.y, self.t)

```

```

        return self.loss

def backward(self, dout=1):
    batch_size = self.t.shape[0]
    if self.t.size == self.y.size: # 教師データが one-hot-vector の場合
        dx = (self.y - self.t) / batch_size
    else:
        dx = self.y.copy()
        dx[np.arange(batch_size), self.t] -= 1
        dx = dx / batch_size

    return dx

# 2層のニューラルネットを設定
import sys
sys.path.append("/content/drive/MyDrive/Colab Notebooks")
from common.gradient import numerical_gradient
from collections import OrderedDict

class TwoLayerNet:

    def __init__(self, input_size, hidden_size, output_size, weight_init_std = 0.01):
        # 重みの初期化
        self.params = {}
        self.params['W1'] = weight_init_std * np.random.randn(input_size, hidden_size)
        self.params['b1'] = np.zeros(hidden_size)
        self.params['W2'] = weight_init_std * np.random.randn(hidden_size, output_size)
        self.params['b2'] = np.zeros(output_size)

        # レイヤの生成
        self.layers = OrderedDict()
        self.layers['Affine1'] = Affine(self.params['W1'], self.params['b1'])
        self.layers['Relu1'] = Relu()
        self.layers['Affine2'] = Affine(self.params['W2'], self.params['b2'])

        self.lastLayer = SoftmaxWithLoss()

    def predict(self, x):
        for layer in self.layers.values():
            x = layer.forward(x)

```

```

        return x

# x:入力データ, t:教師データ
def loss(self, x, t):
    y = self.predict(x)
    return self.lastLayer.forward(y, t)

def accuracy(self, x, t):
    y = self.predict(x)
    y = np.argmax(y, axis=1)
    if t.ndim != 1 : t = np.argmax(t, axis=1)

    accuracy = np.sum(y == t) / float(x.shape[0])
    return accuracy

# x:入力データ, t:教師データ
def numerical_gradient(self, x, t):
    loss_W = lambda W: self.loss(x, t)

    grads = {}
    grads['W1'] = numerical_gradient(loss_W, self.params['W1'])
    grads['b1'] = numerical_gradient(loss_W, self.params['b1'])
    grads['W2'] = numerical_gradient(loss_W, self.params['W2'])
    grads['b2'] = numerical_gradient(loss_W, self.params['b2'])

    return grads

def gradient(self, x, t):
    # forward
    self.loss(x, t)

    # backward
    dout = 1
    dout = self.lastLayer.backward(dout)

    layers = list(self.layers.values())
    layers.reverse()
    for layer in layers:
        dout = layer.backward(dout)

```

```
# 設定
grads = {}
grads['W1'], grads['b1'] = self.layers['Affine1'].dW, self.layers['Affine1'].db
grads['W2'], grads['b2'] = self.layers['Affine2'].dW, self.layers['Affine2'].db

return grads
```

## 参考文献

- [1] Andriy Burkov. (2019). The hundred-page machine learning book.
- [2] Francois Chollet. (2018). Deep learning with python. Manning Publications Co.
- [3] Marc Peter Deisenroth., A. Aldo Faisal., Cheng Soon Ong. (2020). Mathematics for machine learning. Cambridge University Press.
- [4] Aurélien Geron. (2019). Hands-on machine learning with Scikit-Learn, Keras & TensorFlow. 2nd Edition. Oreilly.
- [5] 小縣信也., 斎藤翔汰., 溝口聡., 若杉一幸. (2021). ディープラーニング E 資格エンジニア問題集. インプレス.
- [6] 岡谷貴之. (2015). 深層学習. 講談社.
- [7] Sebastian Raschka., Vahid Mirjalili. (2019). Python machine learning. Third Edition. Packt.
- [8] 斎藤康毅. (2016). ゼロから作る deep learning. オライリージャパン.
- [9] 斎藤康毅. (2018). ゼロから作る deep learning 2. オライリージャパン.
- [10] 瀧雅人. (2017). これならわかる深層学習. 講談社.