４章
ニューラルネットワークの学習

赤嶺研
澤岻成弥・高角諒・新垣結梨

4章の目的

・ニューラルネットワークの学習について学ぶ

※「学習」とは訓練データから適切なパラメータの値を自動で獲得する事を指す

4.1 データから学習する

ニューラルネットワークの特徴

→「重み」のパラメータをデータから自動で決定できる

実際のニューラルネットワーク、ディープラーニングでは数千数万のパラメー　タが必要であり、手作業ですることは現実的では無い

4.1.1データ駆動

・機械学習に必須なモノ　→　**データ**

・何かのパターンを見つける問題

　　ー 人が解決する場合
　　　直感、経験に基づいて試行錯誤する

　　ー 機械学習
　　　データの中からパターンを探し出す

　　ー ニューラルネットワーク/ディープラーニング
　　　機械学習以上に極力人の介入を避ける

4.1.1データ駆動

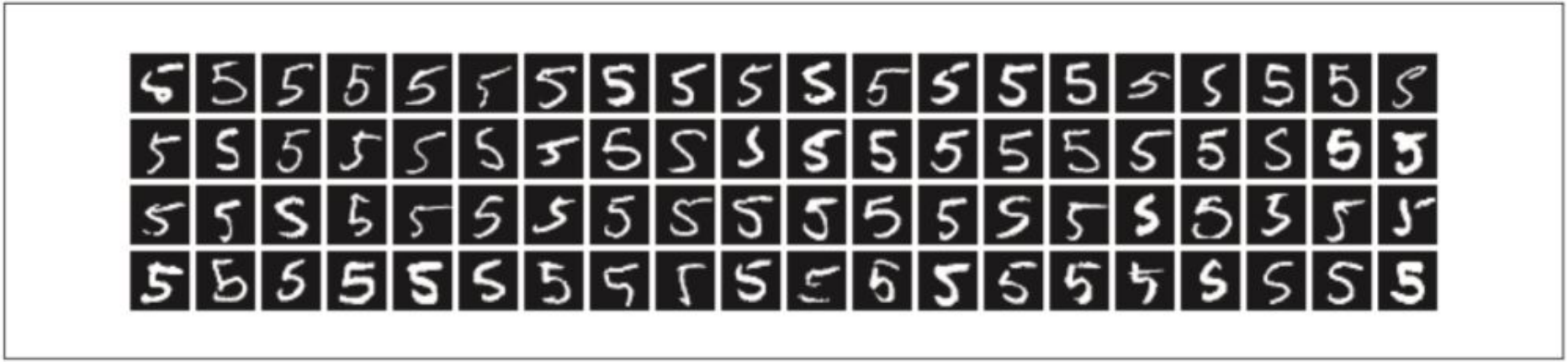


図4-1　手書き数字の「5」の例：人によってさまざまな書き方（クセ）がある

ゴール：5か/5でないかを分類するプログラムを作る

4.1.1データ駆動

ゴールに向けてのアプローチ

1. 人が5を判別するアルゴリズムを考える

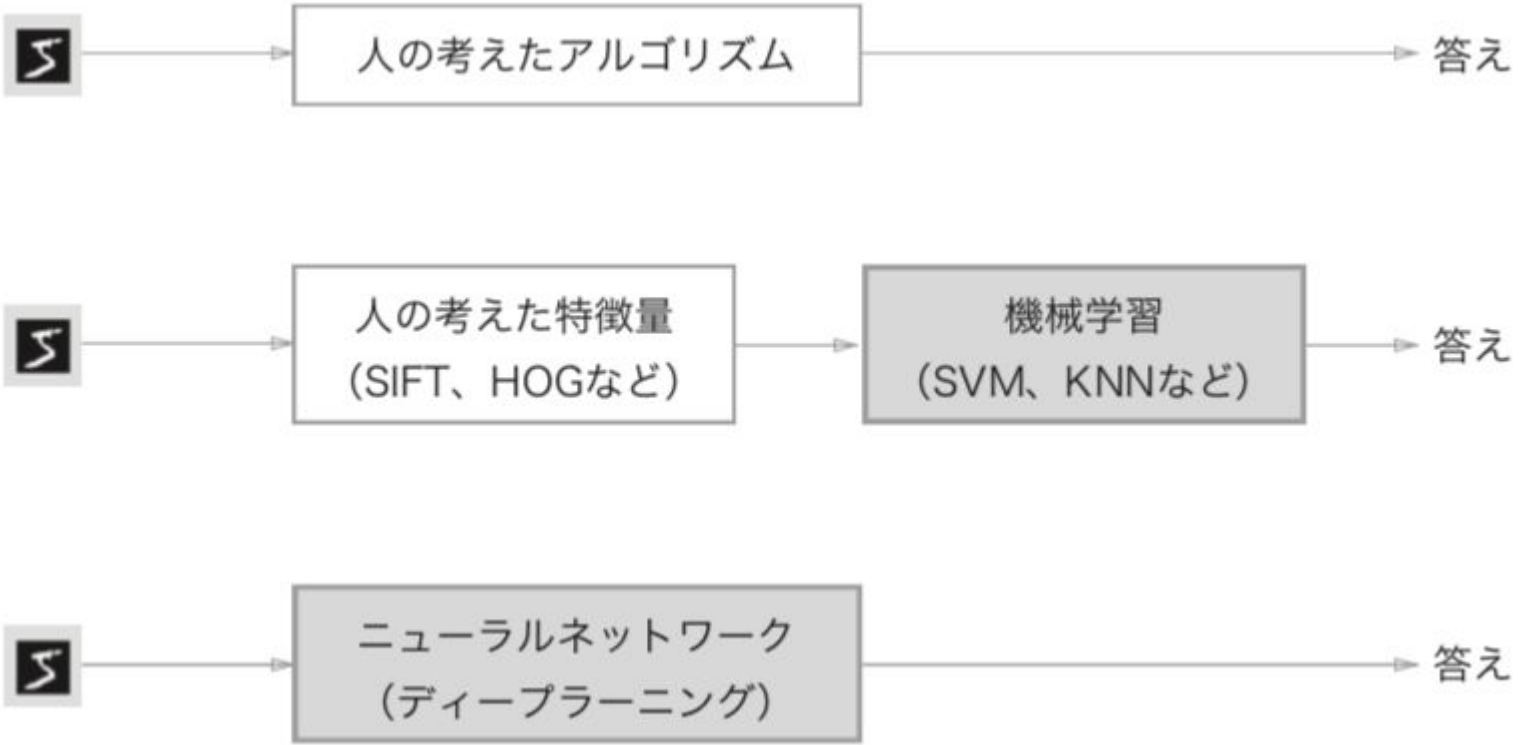
2. 人が考えた「特徴量」を用いて、機械学習を行う

   特徴量：入力画像から重要なデータを抽出する変換器
   画像の特徴量は通常ベクトルで記述される

   有名な特徴量：SIFT 、SURF、HOG

3. ニューラルネットワークを用いる
   人が介入しない、特徴量も機械が「学習」データの中から獲得する

4.1.1データ駆動


人の考えたアルゴリズム
答え
人の考えた特徴量
（SIFT、HOGなど）
機械学習
（SVM、KNNなど）
答え
ニューラルネットワーク
（ディープラーニング）
答え

4.1.2 訓練データとテストデータ

・機械学習の問題では訓練データとテストデータに分けて学習・実験を行う

・訓練データ(**教師データ**ともいう)で学習を行い、最適なパラメータを探索
　その後テストデータで訓練モデルを評価する

なぜ訓練データとテストデータを分ける？

4.1.2 訓練データとテストデータ

・**汎化能力**を評価するため

　　― 汎化能力とは訓練データに含まれないデータに対する能力

・これの獲得が機械学習の最終目標

・ひとつのデータセットで学習と評価を行ってしまうと、正しい評価が行えない

・特定のデータセットに過度に対応した状態、**過学習**を招いてしまう

・過学習を避けることも機械学習の重要課題である

4.2　損失関数

・ニューラルネットワークの学習ではある「指標」を元に現在の状態を表す

・その指標を元に最適な重みパラメータを探索する

・この「指標」のことを**損失関数**と呼ぶ

・損失関数は任意の関数であるが、一般に2乗和誤差や交差エントロピー誤差な　どが用いられる

4.2.1　2乗和誤差

$$E = \frac{1}{2}\sum_{k}(y_k - t_k)^2 \quad (4.1)$$

ykはニューラルネットワークの出力、tkは教師データを表す

「3.6 手書き数字認識」の例では次のようになる

```
>>> y = [0.1, 0.05, 0.6, 0.0, 0.05, 0.1, 0.0, 0.1, 0.0, 0.0]
>>> t = [0, 0, 1, 0, 0, 0, 0, 0, 0, 0]
```

この配列の要素は、最初のインデックスから順に数字の「0」「1」「2」…に対応

4.2.1 2乗和誤差

```
>>> y = [0.1, 0.05, 0.6, 0.0, 0.05, 0.1, 0.0, 0.1, 0.0, 0.0]
```

ニューラルネットワークの出力yはソフトマックス関数の出力

ソフトマックス関数の出力は確率と解釈できる

「0」の確率は0.1、「1」の確率0.05、「2」の確率0,6

```
>>> t = [0, 0, 1, 0, 0, 0, 0, 0, 0, 0]
```

tは教師データで、正解となるラベルを1、それ以外を0とする

ここではラベルの「2」が1なので、「2」が正解であることを表す

正解ラベルを1、それ以外を0で表す表記法を**one-hot表現**という

2乗和誤差をPythonで実装すると

```
def mean_squared_error(y, t):
        return 0.5 * np.sum((y-t)**2)
```

ここでの引数y,tはNumPyの配列、式(4,1)の通り

```
>>> #「2」を正解とする
>>> t = [0, 0, 1, 0, 0, 0, 0, 0, 0, 0]
```

```
>>> # 例 1:「2」の確率が最も高い場合(0.6)
>>> y = [0.1, 0.05, 0.6, 0.0, 0.05, 0.1, 0.0, 0.1, 0.0, 0.0] >>>
mean_squared_error(np.array(y), np.array(t))
0.09750000000000031
```

```
>>> # 例 2:「7」の確率が最も高い場合(0.6)
>>> y = [0.1, 0.05, 0.1, 0.0, 0.05, 0.1, 0.0, 0.6, 0.0, 0.0] >>>
mean_squared_error(np.array(y), np.array(t))
0.5975000000000003
```

4.2.2 交差エントロピー誤差

$$E = -\sum_k t_k \log y_k \quad (4.2)$$

ykはニューラルネットワークの出力、tkは正解ラベル(tkはone-hot表現)

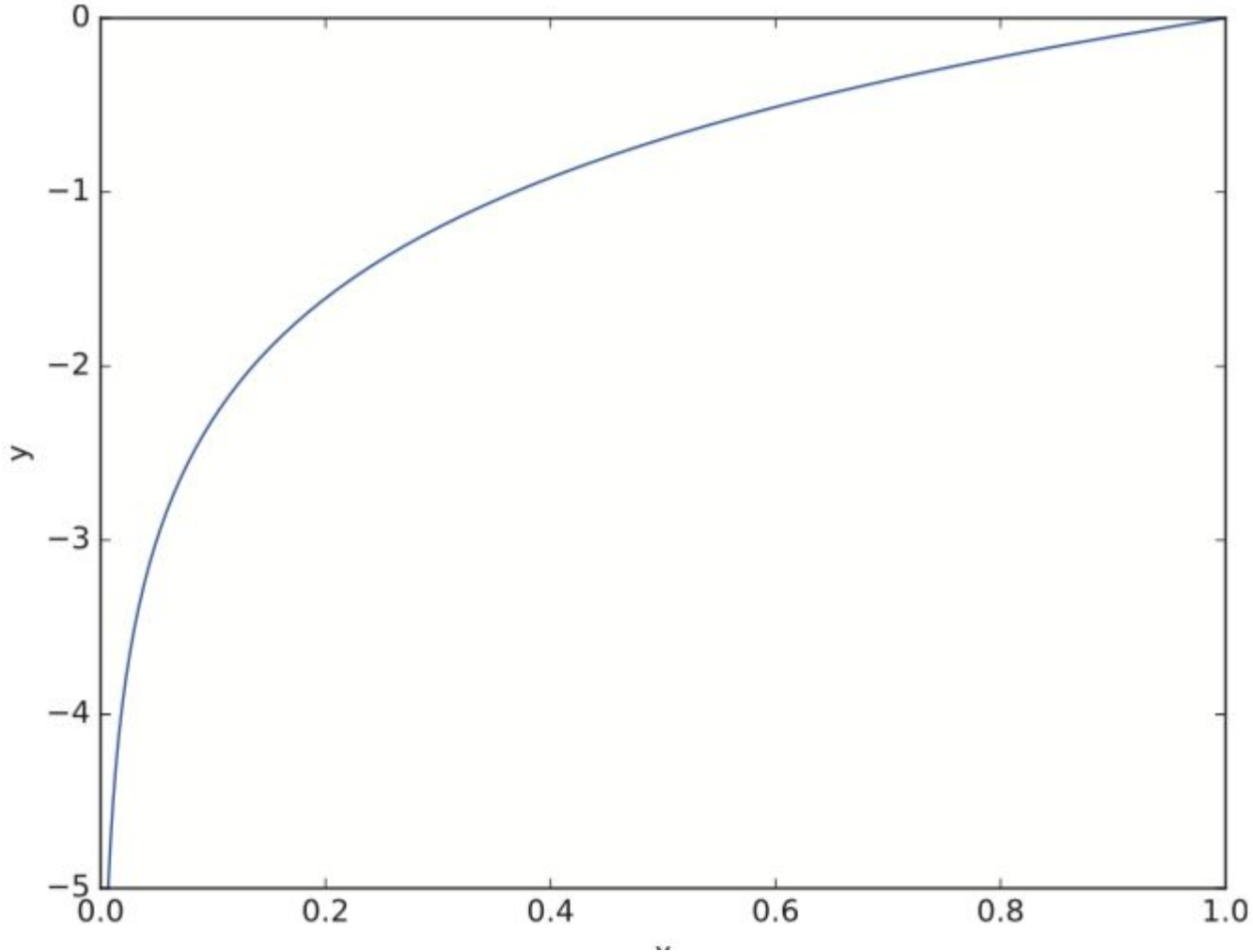
この式 (4.2) は正解ラベル1に対応する出力の自然対数を計算する

「2」が正解ラベルのインデックスの時、ykが0.6なら交差エントロピー誤差は

-log0.6=0.51

「2」の出力が0.1 の場合は− log 0.1 = 2.30

交差エントロピー誤差は正解ラベルの出力の結果で値が決まる



自然対数 y = logx

交差エントロピー誤差の実装

```
def cross_entropy_error(y, t):
    delta = 1e-7
    return -np.sum(t * np.log(y + delta))    ※1e-7=1*10^-7
```

np.log の 計算時に、微小な値 delta を足して計算している

np.log(0) のような計算の場合np.log(0) はマイナス無限大を表す-inf となり

計算できない

その防止として、極小な値を追加して、マイナス無限大になることを防ぐ

cross_entropy_error の計算については割愛

4.2.3 ミニバッチ学習

機械学習問題は訓練データを用いて学習を行うが、正確には訓練データに対する

損失関数を求め、その値をできるだけ小さくするようなパラメータを探し出す

損失関数は全訓練データを対象ににする必要がある。

訓練データが 100 個あれば、その 100 個の損失関数の和を指標とする

全訓練データの損失関数の和交差エントロピー誤差を求める場合

$$E = -\frac{1}{N}\sum_{n}\sum_{k} t_{nk}\log y_{nk} \quad (4.3)$$

データがN個、tnkはn個目のデータのk次元目の値を意味する

ynkはニューラルネットワークの出力、tnkは教師データ

Nで割ることで「平均の損失関数」を求めデータの数に関係なく統一された指標が得られる

全てのデータを対象として損失関数を計算 するのは時間がかかり非現実的

そこで、データののの一部を選び出し、それを全体の「近似」として利用する

この訓練データから一部を選び出す、これを「**ミニバッチ(小さな塊)**」という

そのミニバッチ毎に学習を行う学習法を「**ミニバッチ学習**」と呼ぶ

訓練データから指定個数のデータをランダムに呼び出す機能の実装

その前にMNISTデータセットの読み込み

コードは省略(教科書見て)

引数 one_hot_label=True でone-hot表現のデータ構造として取得できる

読み込んだ訓練データからどうやってランダムに10枚だけ抜き出す?

関数: np.random.choice()

詳細は教科書
np.random.choiceでランダムに選ばれたインデックスを指定、ミニバッチを取り出して、
損失関数を計算する

4.2.4 [バッチ対応版]交差エントロピー誤差の実装

バッチデータに対応した交差エントロピー誤差の実装

教科書ではデータが一つの場合、データがバッチとしてしてまとめられて入力される場合の両方のケースに対応するように実装

実装の方法に関しては教科書参照

4.2.5 なぜ損失関数を設定するのか?

数字認識の場合、「認識精度」を指標にした方が高精度なニューラルネットワークを得られるのでは？

ニューラルネットワークの学習では最適なパラメータ(重みとバイアス)を探索する際に、損失関数の値ができるだけ小さくなるようパラメータを探します

この際パラメータの微分勾配を計算、その値を手がかりにパラメータを更新する

ニューラルネットワークの学習の際に、認識精度を“指標”にしてはいけない。

理由は認識精度を指標にすると、パラメータの微分(勾配)がほとんど0に

なってしまい、パラメータが更新されず学習できない

上記を説明する具体例：

ニューラルネットワークが100枚の訓練データ中32枚を認識できていると仮定

この時の認識精度は32%

認識精度が指標の場合、重みパラメータの多少の調整では精度が改善されず、仮に改善されても不連続になる

※連続不連続よく分からん場合は「関数連続性」あたりのワードでググろう

損失関数を指標とした場合、現在の損失関数の値は 0.92543…のように表されるパラメータの値を少し変化させると、損失関数も 0.93432…のように連続的に変化する

認識精度は3章で説明のあった活性化関数「ステップ関数」と同様に不連続に変化する「ステップ関数」の微分、接線の勾配はほとんどの場所で0

同じく3章で紹介されていた、「シグモイド関数」
「シグモイド関数」の微分、接線の勾配は0にならない

次のスライドに図で示す

ニューラルネットワークの学習に置いて、この**「傾きが0にならない」**

という性質は正しい学習の為に非常に重要

微分（勾配)については次節で説明する

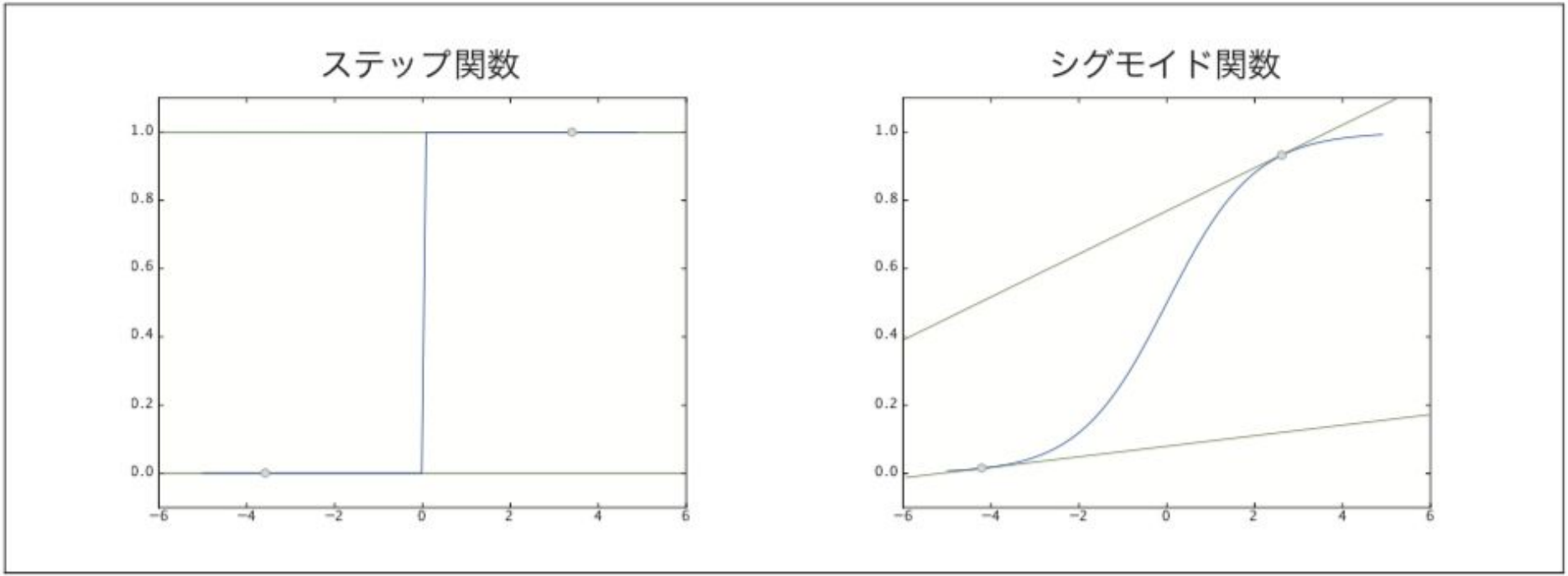


図4-4　ステップ関数とシグモイド関数：ステップ関数はほとんどの場所で傾きは0であるのに対して、シグモイド関数の傾き（接線）は0にならない

4.3 数値微分

4.3.1 微分

微分・・・ある瞬間の変化量を表したもの。

微分の定義式・・・

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h} \tag{4.4}$$

x の「小さな変化」によって、関数 f(x) の値がどれだけ変化するかということを意味する。その際、「小さな変化」である h を限りなく 0 に近づける。

```
# 悪い実装例
def numerical_diff(f, x):
    h = 10e-50
    return (f(x+h) - f(x)) / h
```

hにはできるだけ小さな値(10e-50)を代入

←改善すべきポイントが2つある

- 改善点その1：丸め誤差の問題

  1e-50 を float32 型(32 ビットの浮動小数点数)で表すと、0.0となり、正しく表現できない。

  →微小な値hとして10^{-4}を用いる

  （この程度であれば良い結果が得られることがわかっている）

- 改善点その2：関数 f の差分

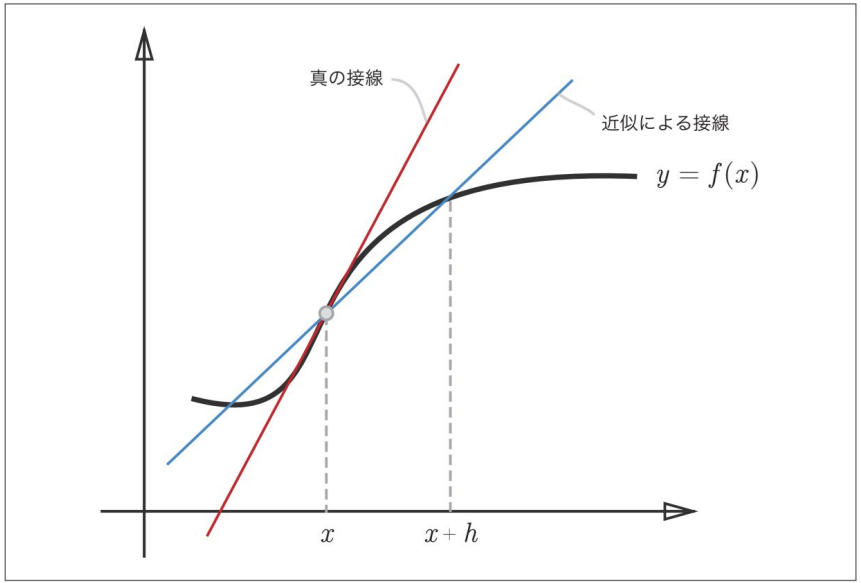
  数値微分には誤差が生じる

（hを無限に0へと近づけることができないため）

→(x + h) と (x − h) での関数 f の差分を計算

することで、誤差を減らすことができる。

（中心差分）



改善後の数値微分（数値勾配）の実装

```
def numerical_diff(f, x):
    h = 1e-4 # 0.0001
    return (f(x+h) - f(x-h)) / (2*h)
```

ここで行っているように、微小な差分によって微分を求めることを**数値微分**

(numerical differentiation)と言う。

また、数式の展開によって微分を求めることを「**解析的**に解く」などと言う。

4.3.2 数値微分の例

$$y = 0.01x^2 + 0.1x \tag{4.5}$$

この式をPythonで実装すると、

```
def function_1(x):
    return 0.01*x**2 + 0.1*x
```

この関数の微分を、x=5 と x=10 のときで、それぞれ計算してみると..

```
>>> numerical_diff(function_1, 5)
0.1999999999990898
>>> numerical_diff(function_1, 10)
0.2999999999986347
```

解析的な解は、df(x)/dx = 0.02x + 0.1

そのためx = 5, 10での「真の微分」は0.2、0.3であり、上の数値微分とは厳密には一致しないが、その誤差は非常に小さい。

4.3.3 偏微分

偏微分・・・複数の変数からなる関数の微分（ $\frac{\partial f}{\partial x_0}$、$\frac{\partial f}{\partial x_1}$

1変数の微分と同じで、ある場所の傾きを求める。

$$f(x_0, x_1) = x_0^2 + x_1^2 \qquad (4.6)$$

問 1：$x_0 = 3$、$x_1 = 4$ のときの x_0 に対する偏微分 $\frac{\partial f}{\partial x_0}$ を求めよ。

```
>>> def function_tmp1(x0):
...     return x0*x0 + 4.0**2.0
...
>>> numerical_diff(function_tmp1, 3.0)
6.00000000000378
```

問 2：$x_0 = 3$、$x_1 = 4$ のときの x_1 に対する偏微分 $\frac{\partial f}{\partial x_1}$ を求めよ。

```
>>> def function_tmp2(x1):
...     return 3.0**2.0 + x1*x1
...
>>> numerical_diff(function_tmp2, 4.0)
7.999999999999119
```

複数ある変数の中でターゲットとする変数をひとつに絞り、他の変数はある値に固定する。

←値を固定するために新しい関数を定義し、数値微分の関数に渡している。

4.4 勾配

$\left(\frac{\partial f}{\partial x_0}, \frac{\partial f}{\partial x_1}\right)$ のように、全ての変数の偏微分をベクトルとしてまとめたものを**勾配**(gradient)と言う。

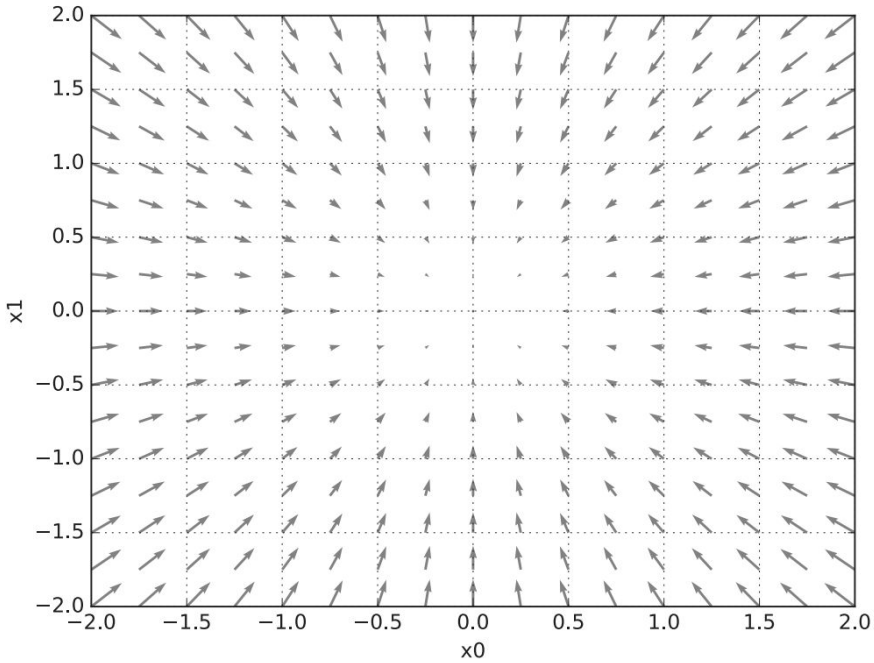
勾配の実装

```
def numerical_gradient(f, x):
    h = 1e-4 # 0.0001
    grad = np.zeros_like(x) # x と同じ形状の配列を生成

    for idx in range(x.size):
        tmp_val = x[idx]
        # f(x+h) の計算
        x[idx] = tmp_val + h
        fxh1 = f(x)

        # f(x-h) の計算
        x[idx] = tmp_val - h
        fxh2 = f(x)

        grad[idx] = (fxh1 - fxh2) / (2*h)
        x[idx] = tmp_val # 値を元に戻す

    return grad
```

numerical_gradient(f, x) 関数は、引数の f は関数、x は NumPy 配列であるとして、NumPy 配列 x の各要素に対して数値微分を求める。

$f(x_0, x_1) = x_0^2 + x_1^2$ の勾配を、
向きをもったベクトル（矢印）として
図で表す。

（ソースコードはch04/gradient_2d.py）



右図より、勾配は関数(x_0, x_1) の最小値を
指している。

正確には...

勾配が示す方向は、各場所において
関数の値を最も減らす方向！

4.4.1　勾配法

ニューラルネットワークは最適なパラメータ(重みとバイアス)を学習時に探索する。

最適なパラメータとは損失関数が最小値を取る時の値 → 勾配を利用して関数の最小値を探索したい

※各地点における関数の値を最も減らす方向を示すのが勾配

→ 勾配が指す先が本当に関数の最小値なのかどうか保証できない.

勾配法・・・勾配方向へ進むことを繰り返すことで、関数の値を徐々に減らす手法。

(最小値を探す場合を**勾配降下法**、最大値を探す場合を**勾配上昇法**と呼ぶ)

勾配法を数式で表すと…

$$x_0 = x_0 - \eta \frac{\partial f}{\partial x_0}$$
$$x_1 = x_1 - \eta \frac{\partial f}{\partial x_1} \qquad (4.7)$$

式 (4.7) の η は更新の量を表す。これは、ニューラルネットワークの学習においては、学習率 (learning rate)と呼ばれ、1 回の学習で、どれだけ学習すべきか、どれだけパラメータを更新するかを決定する。

学習率の値は0.01や0.001など、前もって何らかの値に決める必要がある。

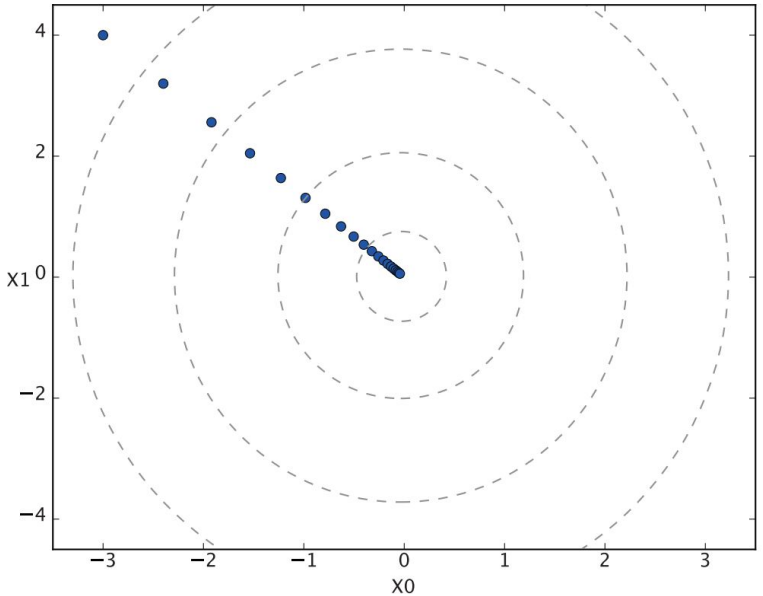
学習率は大きすぎても小さすぎても良い結果が得られない。

大きすぎると、大きな値へと発散し、逆に小さすぎると、値がほとんど更新されない。

学習率の値を変更しながら正しく学習できているかどうか、確認作業を行うのが一般的。

勾配法による更新のプロセスを図示すると、右図のようになる。

図から、最も低い場所である原点に徐々に近づいていることがわかる。



$f(x_0, x_1) = x_0^2 + x_1^2$ の勾配法による更新のプロセス：破線は関数の等高線を示す

4.4.2　ニューラルネットワークに対する勾配

ニューラルネットワークの学習における勾配は、重みパラメータに関する損失関数の勾配を指す。

例えば、形状が2×3の重みWだけを持つニューラルネットワークがあり、損失関数をLで表す場合、
勾配は ∂L/∂W と表すことができる。数式で表すと...

$$\mathbf{W} = \begin{pmatrix} w_{11} & w_{21} & w_{31} \\ w_{12} & w_{22} & w_{32} \end{pmatrix}$$

$$\frac{\partial L}{\partial \mathbf{W}} = \begin{pmatrix} \frac{\partial L}{\partial w_{11}} & \frac{\partial L}{\partial w_{21}} & \frac{\partial L}{\partial w_{31}} \\ \frac{\partial L}{\partial w_{12}} & \frac{\partial L}{\partial w_{22}} & \frac{\partial L}{\partial w_{32}} \end{pmatrix} \tag{4.8}$$

∂W の各要素は、それぞれの要素に関する偏微分から構成される。ここで1行1列目の要素である $\partial L/\partial w_{11}$ は、w_{11} を少し変化させると損失関数 L がどれだけ変化するかを表す。

ここで大切な点は、∂L/∂W の形状は W と同じであるということ。

実際に勾配を求めるsimpleNetクラスを実装する。

```
import sys, os
sys.path.append(os.pardir)
import numpy as np
from common.functions import softmax, cross_entropy_error
from common.gradient import numerical_gradient

class simpleNet:
    def __init__(self):
        self.W = np.random.randn(2,3) # ガウス分布で初期化

    def predict(self, x):
        return np.dot(x, self.W)

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

        return loss
```

形状が2×3の重みパラメータをひとつだけインスタンス変数として持つ。

予測するためのメソッドpredict(x)、
損失関数を求めるためのメソッドloss(x, t)がある。

引数のxには入力データ、t には正解ラベルが入力されるものとする。

simpleNetを使う。

```
>>> net = simpleNet()
>>> print(net.W) # 重みパラメータ
[[ 0.47355232,  0.9977393 ,  0.84668094],
 [ 0.85557411,  0.03563661,  0.69422093]])
>>>
>>> x = np.array([0.6, 0.9])
>>> p = net.predict(x)
>>> print(p)
[ 1.13282549  0.66052348  1.20919114]
>>> np.argmax(p) # 最大値のインデックス
2
>>>
>>> t = np.array([0, 0, 1]) # 正解ラベル
>>> net.loss(x, t)
0.92806853663411326
```

続いて勾配を求める

```
>>> def f(W):
...     return net.loss(x, t)
...
>>> dW = numerical_gradient(f, net.W)
>>> print(dW)
[[ 0.21924763  0.14356247 -0.36281009]
 [ 0.32887144  0.2153437  -0.54421514]]
```

numerical_gradient(f, net.W) の結果は dW となり、形状が 2×3 の 2 次元配列になる。

dW の中身を見ると、たとえば、W の w_{11} はおよそ 0.2 ということが分かる。

これは、w_{11} を h だけ増やすと損失関数の値は 0.2h だけ増加するということを意味する。

また、w_{23} はおよそ −0.5 だが、これは w_{23} を h だけ増やすと損失関数の値は 0.5h だけ減少する。

そのため、損失関数を減らすという観点からは、w_{23} はプラス方向へ更新し、w_{11} はマイナス方向へ更新するのが良いことが分か る。また、更新の度合いについても、w_{23}のほうが w_{11} よりも大きく貢献するということが分かる。

ニューラルネットワークの勾配を求めれば、後は勾配法に従って、重みパラメータを更新する。

4.5 学習アルゴリズムの実装

ステップ1(ミニバッチ)

訓練データの中からランダムに一部のデータを選び出す。その選ばれたデータをミニバッチと言い、ここでは、そのミニバッチの損失関数の値を減らすことを目的とする。

ステップ2(勾配の算出)

ミニバッチの損失関数を減らすために、各重みパラメータの勾配を求める。
勾配は、損失関数の値を最も減らす方向を示す。

ステップ3(パラメータの更新)

重みパラメータを勾配方向に微小量だけ更新する。

ステップ4(繰り返す)

ステップ 1、ステップ 2、ステップ 3 を繰り返す。

4.5　学習アルゴリズムの実装

確率勾配降下法 (stochastic gradient descent)

確率的は、
「確率的に無作為に選び出した」という意味。

確率的勾配降下法は
「無作為に選び出したデータに対して行う勾配降下法」という意味。

4.5.1　2層ニューラルネットワークのクラス

ソースコードは、ch04/two_layer_net.py にあります。

4.5.1　2層ニューラルネットワークのクラス

TwoLayerNet クラスで使用する変数とメソッド

表 4-1　TwoLayerNet クラスで使用する変数

変数	説明
params	ニューラルネットワークのパラメータを保持するディクショナリ変数（インスタンス変数）。 params['W1'] は 1 層目の重み、params['b1'] は 1 層目のバイアス。 params['W2'] は 2 層目の重み、params['b2'] は 2 層目のバイアス。
grads	勾配を保持するディクショナリ変数（numerical_gradient() メソッドの返り値）。 grads['W1'] は 1 層目の重みの勾配、grads['b1'] は 1 層目のバイアスの勾配。 grads['W2'] は 2 層目の重みの勾配、grads['b2'] は 2 層目のバイアスの勾配。

4.5.1　2層ニューラルネットワークのクラス

TwoLayerNet クラスで使用する変数とメソッド

表4-2　TwoLayerNet クラスのメソッド

メソッド	説明
`__init__(self, input_size, hidden_size, output_size)`	初期化を行う。 引数は頭から順に、入力層のニューロンの数、隠れ層のニューロンの数、出力層のニューロンの数。
`predict(self, x)`	認識（推論）を行う。 引数の `x` は画像データ。
`loss(self, x, t)`	損失関数の値を求める。 引数の `x` は画像データ、`t` は正解ラベル（以下の 3 つのメソッドの引数についても同様）。
`accuracy(self, x, t)`	認識精度を求める。
`numerical_gradient(self, x, t)`	重みパラメータに対する勾配を求める。
`gradient(self, x, t)`	重みパラメータに対する勾配を求める。 `numerical_gradient()` の高速版！ 実装は次章で行う。

4.5.2 ミニバッチの実装

ソースコードは ch04/train_neuralnet.py にあります。

4.5.2 ミニバッチの実装

ミニバッチのサイズを100として、
毎回60,000個の訓練データからランダムに
100個のデータを抜き出す

100個のミニバッチを対象に勾配を求め、
確率勾配降下(SDG)によりパラメータを更新

勾配法による更新の回数――繰り返し(iteration)の回数を10,000回として、更新するごとに、訓練データに対する損失関数を計算し、値を配列に追加

```
import numpy as np
from dataset.mnist import load_mnist
from two_layer_net import TwoLayerNet

(x_train, t_train), (x_test, t_test) = load_mnist(normalize=True)

train_loss_list = []

# ハイパーパラメータ
iters_num = 10000
train_size = x_train.shape[0]
batch_size = 100
learning_rate = 0.1

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

for i in range(iters_num):
    # ミニバッチの取得
    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) # 高速版!

    # パラメータの更新
    for key in ('W1', 'b1', 'W2', 'b2'):
        network.params[key] -= learning_rate * grad[key]

    # 学習経過の記録
    loss = network.loss(x_batch, t_batch)
    train_loss_list.append(loss)
```

4.5.2 ミニバッチの実装

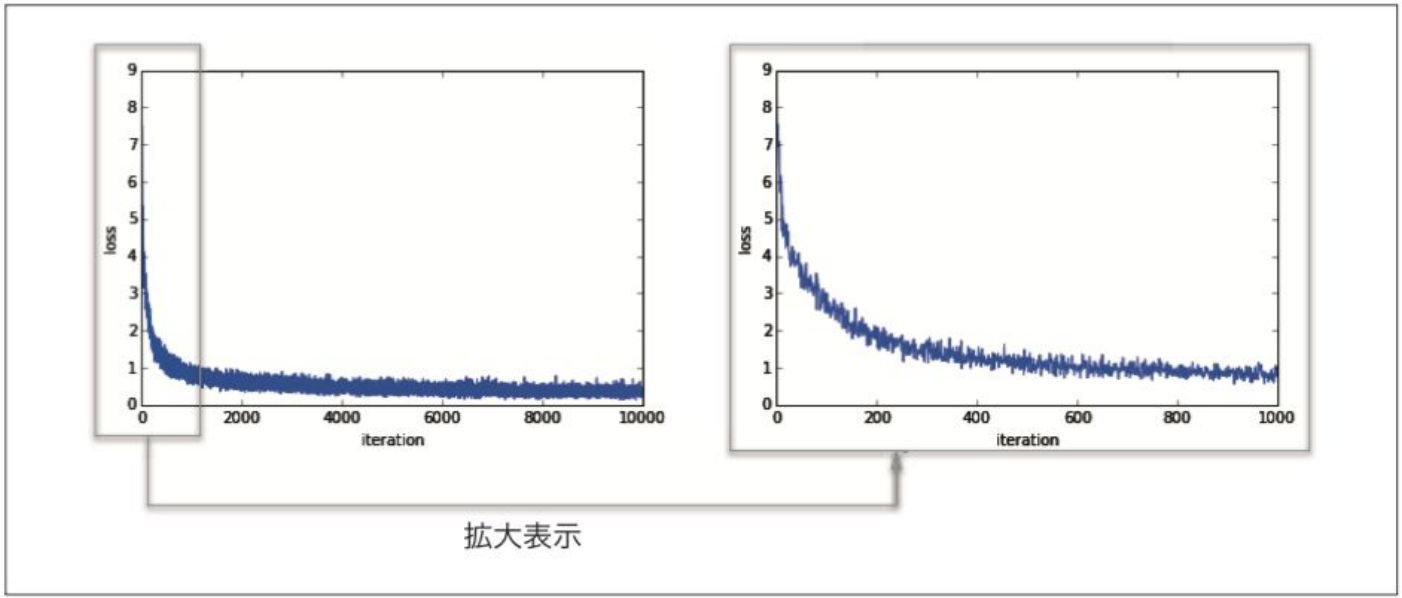


図4-11　損失関数の推移：左図は 10,000 イテレーションまでの推移、右図は 1,000 イテレーションまでの推移

4.5.3　テストデータで評価

学習を繰り返し行うことで損失関数が徐々に下がっていく。

損失関数の値とは、「訓練データのミニバッチに対する損失関数」の値。

訓練データ以外のデータを正しく認識できるかどうかを確認する必要がある。
----「過学習」を起こしていないかの確認。

過学習を起こすとは、
訓練データに含まれている数字画像では正しく見分けられるが
訓練データに含まれない数字画像は認識できない

4.5.3　テストデータで評価

ニューラルネットワークの学習での目標：汎化能力を身に着けること

汎化的な能力を評価するには
訓練データに含まれないデータを使って評価

次の実装では、学習を行う過程で、定期的に訓練データとテストデータを対象に認識制度を記録

4.5.3 テストデータで評価

```
train_loss_list = []
train_acc_list = []
test_acc_list = []
# 1エポックあたりの繰り返し数
iter_per_epoch = max(train_size / batch_size, 1)

# 1エポックごとに認識精度を計算
if i % iter_per_epoch == 0:
    train_acc = network.accuracy(x_train, t_train)
    test_acc = network.accuracy(x_test, t_test)
    train_acc_list.append(train_acc)
    test_acc_list.append(test_acc)
    print("train acc, test acc | " + str(train_acc) + ", " + str(test_acc))
```

4.5.3　テストデータで評価

訓練データとテストデータを使って評価した認識制度は両方とも向上している。

認識制度には差がない。

過学習が起きていない。

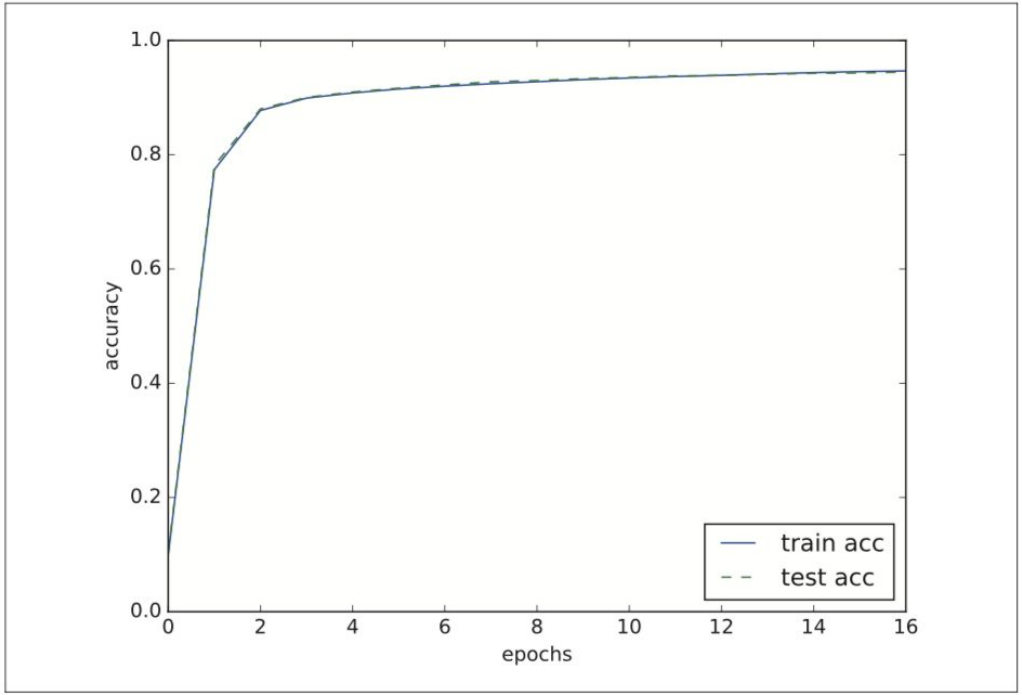


図4-12　訓練データとテストデータに対する認識精度の推移。横軸はエポック

4.6 まとめ

・ニューラルネットワークが学習を行えるようにするために、
損失関数という「指標」を導入

・損失関数を基準として、値が最も小さくなる重みのパラメータを
探し出すことが学習の目標

・小さな損失関数の値を探し出すための勾配法と呼ばれる関数の
傾きを使った手法がある