ゼロから始めるディープラーニング

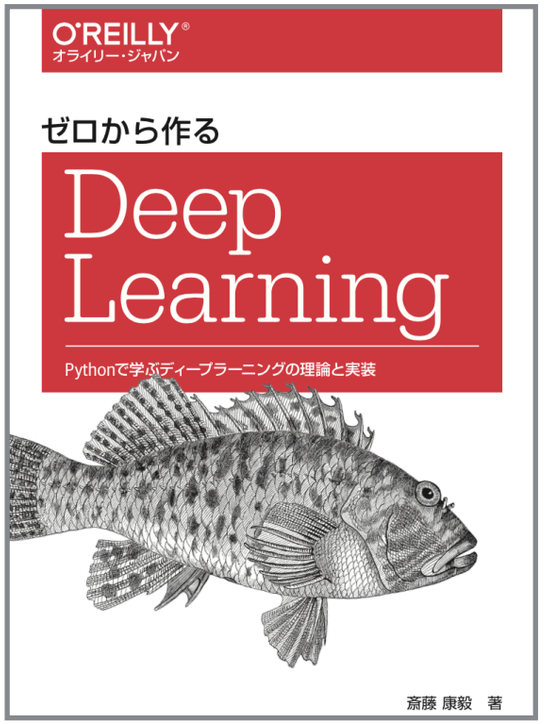
~ 4章：ニューラルネットワークの学習 ~

情報理工学院 知能情報コース

Ishida Lab. M2 Tomohiro Oishi (大石 智博)

21th, May, 2018
教科書輪講

概要



著者：斎藤 康毅

出版：オライリー・ジャパン

サポート：
https://github.com/oreilly-japan/deep-learning-from-scratch

・ニューラルネットワークの学習方法について説明

・MNISTの学習を行うニューラルネットワークを実装

・損失関数

・勾配法

・NNの重みパラメータを自動最適化

目次

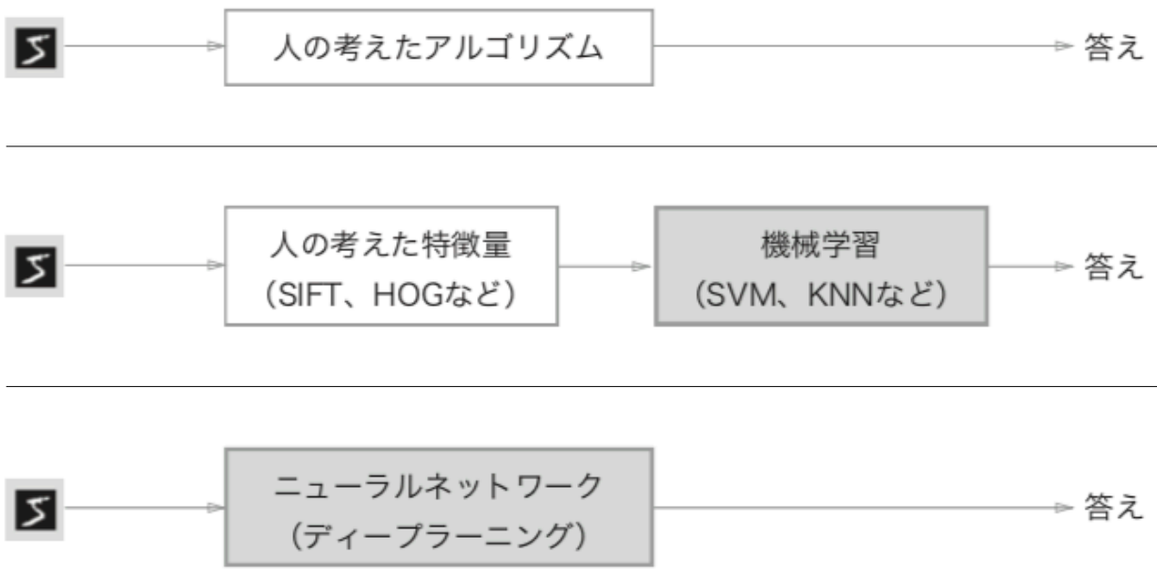
前回：5/21

今回：5/28

目次

ニューラルネットワークの特徴

5か5でないかを見分ける
プログラムを作成したい

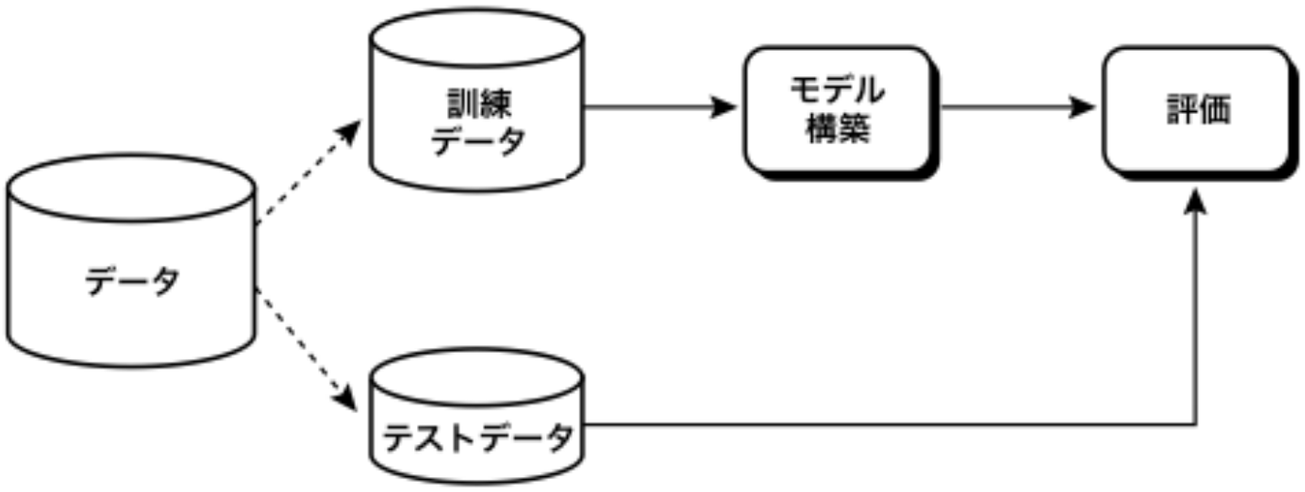


- ニューラルネットワークは，特徴量の設計が不要
  - 特徴量：入力データから本質的なデータを的確に抽出できるよう設計された変換器

- 人の手を介在せず，生データからend to endで学習することができる
  - すべての問題を同じ流れで解くことができる

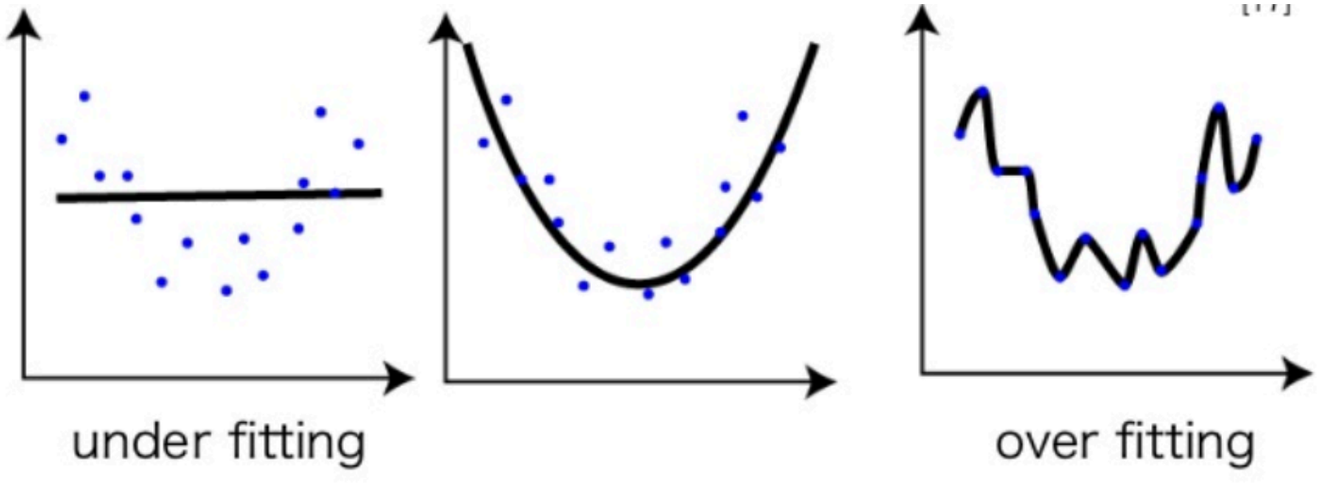
訓練データとテストデータ

・機械学習では，データを訓練データ(教師データ)とテストデータに分割
  - モデルの汎用的な能力（汎化能力）を評価するため



訓練データ　：80%
テストデータ：20%

・特定のデータセットに過度に適応した状態を過学習(over fitting)という



目次

損失関数 (loss function)

NNの重みパラメータを最適化するために損失関数という「指標」を導入

損失関数：

性能の悪さを測る指標．この値を最小化する重みパラメータの探索をおこなう．

ex)

・ 二乗和誤差

$$E = \frac{1}{2}\sum_k (y_k - t_k)^2$$

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

・ 交差エントロピー誤差

$$E = -\sum_k t_k \log y_k$$

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

二乗和誤差 (mean squared error)

$$E = \frac{1}{2}\sum_k (y_k - t_k)^2$$

y_k：ニューラルネットワークの出力

t_k：正解ラベルとなるインデックスのみ1で他は0のone-hot表現

教師データ：t = [0, 0, 1, 0, 0, 0, 0, 0, 0, 0] に対して,

NNの出力 ：y = [0.1, 0.05, 0.6, 0.0, 0.05, 0.1, 0.0, 0.1, 0.0, 0.0]

y = [0.1, 0.05, 0.1, 0.0, 0.05, 0.1, 0.0, 0.6, 0.0, 0.0] で出力をそれぞれ比較.

※tは正解ラベルのみ1としたone-hot表現.

※yはソフトマックス関数の出力で，確率に対応する.

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

# 教師データ：「2」
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]
print(mean_squared_error(np.array(y), np.array(t)))

# 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]
print(mean_squared_error(np.array(y), np.array(t)))
```

```
0.0975
0.5975
```

教師データと適合してる方が,
二乗和誤差の値が小さい.

交差エントロピー誤差 (cross entropy error)

$$E = -\sum_k t_k \log y_k$$

y_k：ニューラルネットワークの出力

t_k：正解ラベルとなるインデックスのみ1で他は0のone-hot表現

つまりこの式は，正解ラベルが1に対応するNNの出力の自然対数を計算するだけ

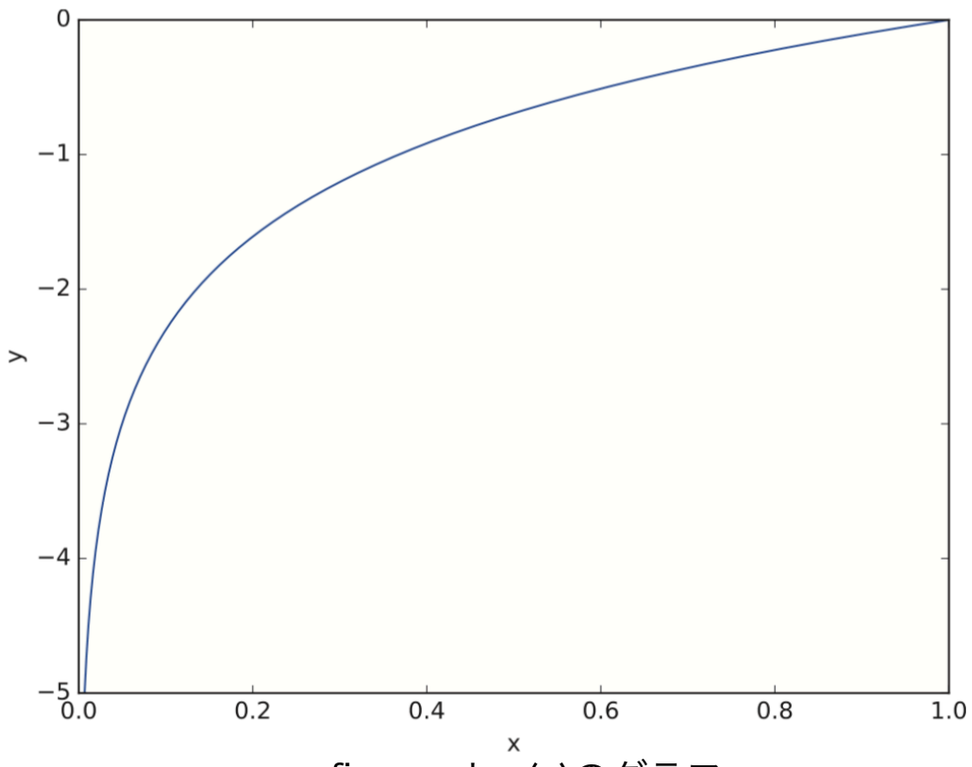


fig. y = log(x)のグラフ

ex1)
t = [0, 0, 1, 0, 0, 0, 0, 0, 0, 0]
y = [0.1, 0.05, 0.6, 0.0, 0.05, 0.1, 0.0, 0.1, 0.0, 0.0] の時
E = -log0.6 = 0.51

ex2)
t = [0, 0, 1, 0, 0, 0, 0, 0, 0, 0]
y = [0.1, 0.05, 0.1, 0.0, 0.05, 0.1, 0.0, 0.6, 0.0, 0.0] の時
E = -log0.1 = 2.30

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

・deltaを足すことでlog0 = -infを防止

ミニバッチ学習

・先程は１つのデータの損失関数を求めていた
・訓練データ全ての損失関数の和に拡張したい

$$E = -\frac{1}{N}\sum_n\sum_k t_{nk}\log y_{nk}$$

・Nで割ることで正規化
・平均の損失関数を求める

※N　：データ数

この時，全ての訓練データを学習すると時間がかかる

ランダムに選ばれた一部のデータを全体の近似として学習させる（ミニバッチ学習）

MNISTの全60,000件の訓練データから10件をバッチとして取得してみよう！

ミニバッチ学習を用いた交差エントロピー誤差の実装

In [127]:
```
import sys, os
sys.path.append(os.pardir)
import numpy as np
from dataset.mnist import load_mnist
(x_train, t_train), (x_test, t_test) = \
    load_mnist(normalize=True, one_hot_label=True)
print(x_train.shape) # (60000, 784)
print(t_train.shape) # (60000, 10)
```

```
(60000, 784)
(60000, 10)
```

In [128]:
```
train_size = x_train.shape[0]
batch_size = 10
batch_mask = np.random.choice(train_size, batch_size)
x_batch = x_train[batch_mask]
t_batch = t_train[batch_mask]
```

- 教師データtがone-hot表現の時 （例）t = [0, 0, 1, 0, 0, 0, 0, 0, 0, 0]

In [83]:
```
def cross_entropy_error(y, t):
    if y.ndim == 1:
        t = t.reshape(1, t.size)
        y = y.reshape(1, y.size)
    batch_size = y.shape[0]
    return -np.sum(t * np.log(y + 1e-7)) / batch_size
```

- 教師データtがラベルとして与えられた時　（例）t = 2

In [84]:
```
def cross_entropy_error(y, t):
    if y.ndim == 1:
        t = t.reshape(1, t.size)
        y = y.reshape(1, y.size)
    batch_size = y.shape[0]
    return -np.sum(np.log(y[np.arange(batch_size), t] + 1e-7)) / batch_size
```

なぜ損失関数を設定するのか

Question

「指標」に認識精度を用いればいいのでは？？なぜ損失関数？？

Answer

- 学習は「指標」を元に重みパラメータを更新する作業
- 「指標」を小さく(大きく)するために傾き，つまり微分を使う
- 認識精度を指標にすると殆どの場所で微分値が0になりパラメータの更新ができない
- 損失関数なら微分値を求めることができるため，損失関数を用いる

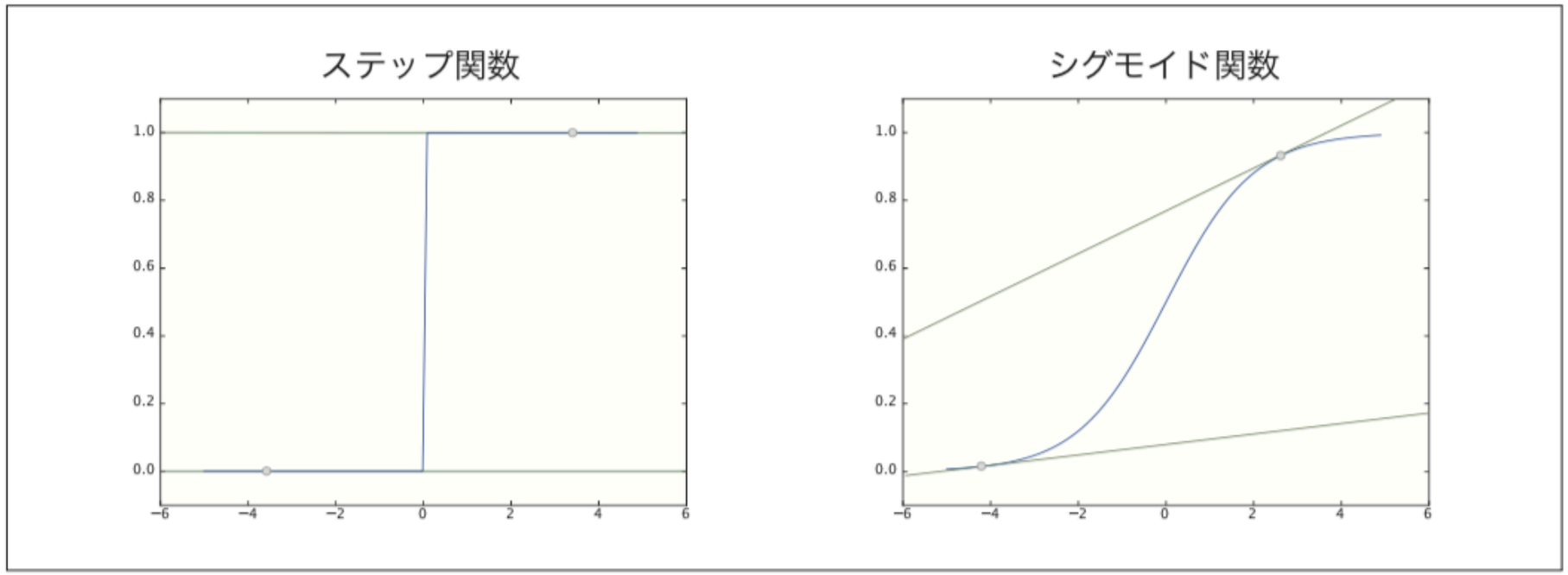


図 4-4　ステップ関数とシグモイド関数：ステップ関数はほとんどの場所で傾きは 0 であるのに対して、シグモイド関数の傾き（接線）は 0 にならない

目次

数値微分

微分とは「ある瞬間」の変化の量をあらわしたもの

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

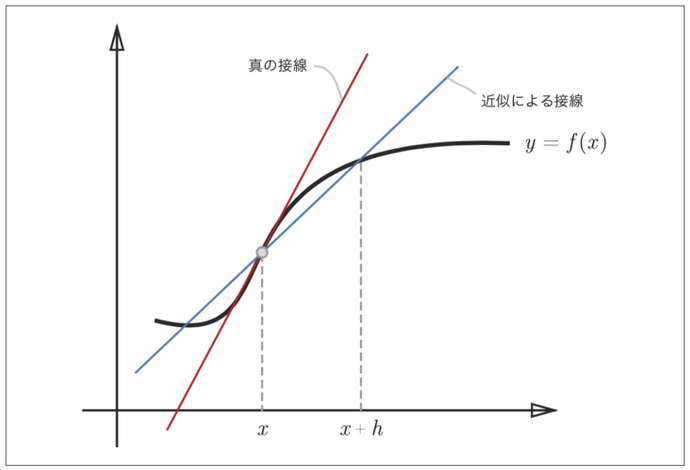


図4-5　真の微分（真の接線）と数値微分（近似による接線）の値は異なる

プログラムで実装してみる

```
def numerical_diff(f, x):
    h = 10e-50
    return (f(x+h) - f(x)) / h
```

- 丸め誤差が問題
- ▶ hが0.0に丸め込まれてしまう
- 近似解が問題

改善案

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

- (h = 1e-4) を用いた
- 中心差分を用いた
  (x + h) と (x - h) の差分

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

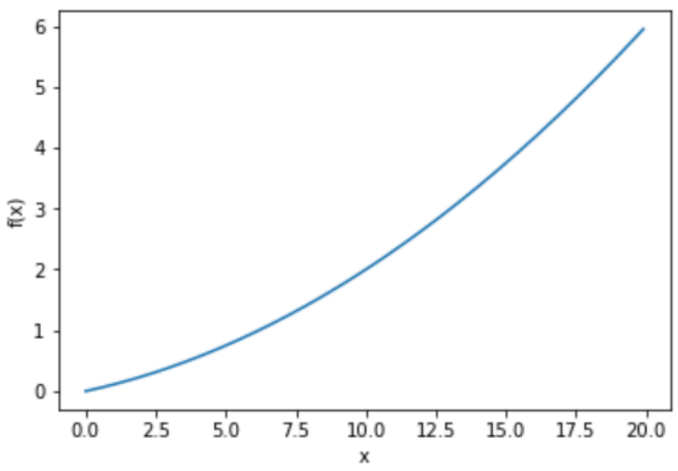
数値微分の例

exercise) 数値微分で簡単な関数を微分

$$y = 0.01x^2 + 0.1x \quad \blacktriangleright \quad \frac{df(x)}{dx} = 0.02x + 0.1$$

In [97]:
```
def function_1(x):
        return 0.01*x**2 + 0.1*x
```

In [99]:
```
import numpy as np
import matplotlib.pylab as plt
x = np.arange(0.0, 20.0, 0.1) # 0 から 20 まで、0.1 刻みの x 配列
y = function_1(x)
plt.xlabel("x")
plt.ylabel("f(x)")
plt.plot(x, y)
plt.show()
```



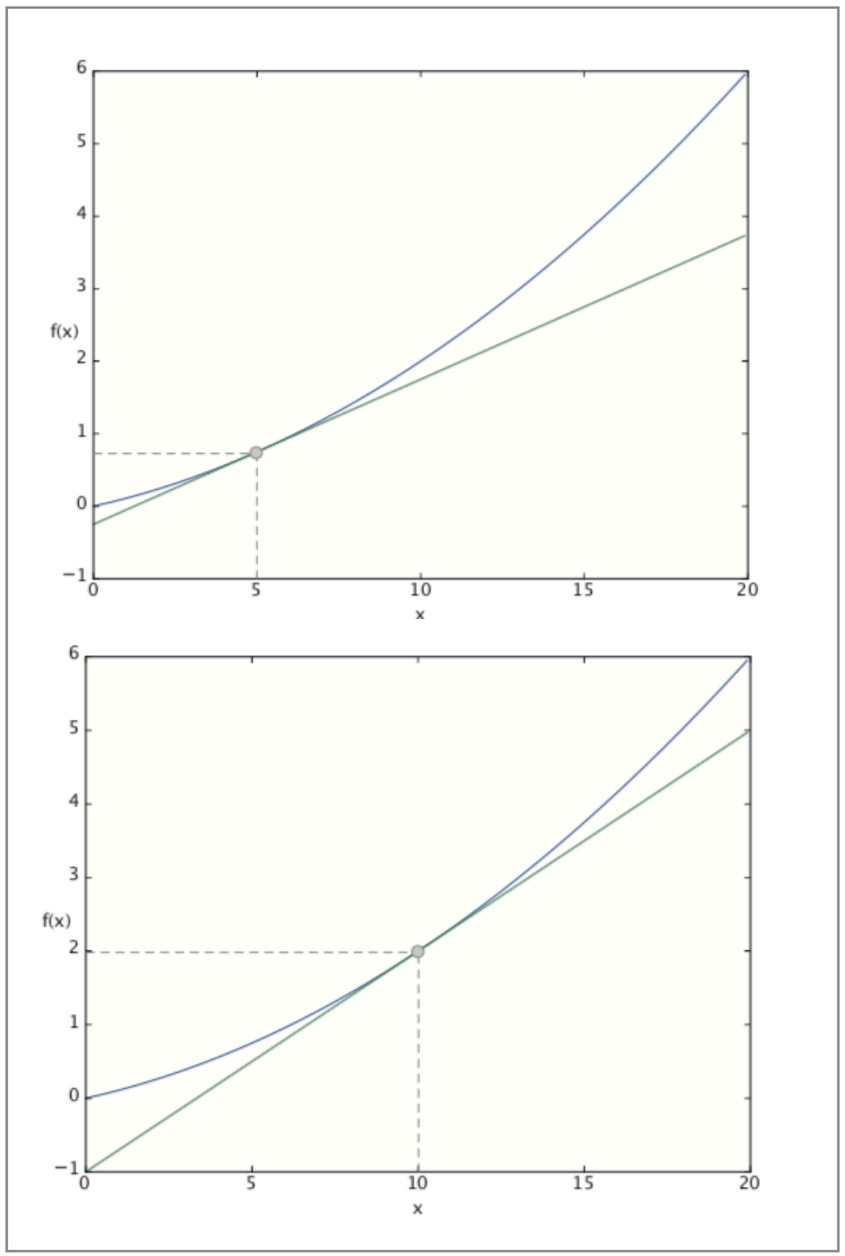
In [104]:
```
numerical_diff(function_1, 5)
```

Out[104]: 0.1999999999990898

In [105]:
```
numerical_diff(function_1, 10)
```

Out[105]: 0.2999999999986347

数値微分で求めた x = 5, x = 10 での接線



偏微分

偏微分：

複数の変数からなる関数の微分のこと

次の関数を考える

$$f(x_0, x_1) = x_0^2 + x_1^2$$

```
def function_2(x):
    return x[0]**2 + x[1]**2
    # または return np.sum(x**2)
```

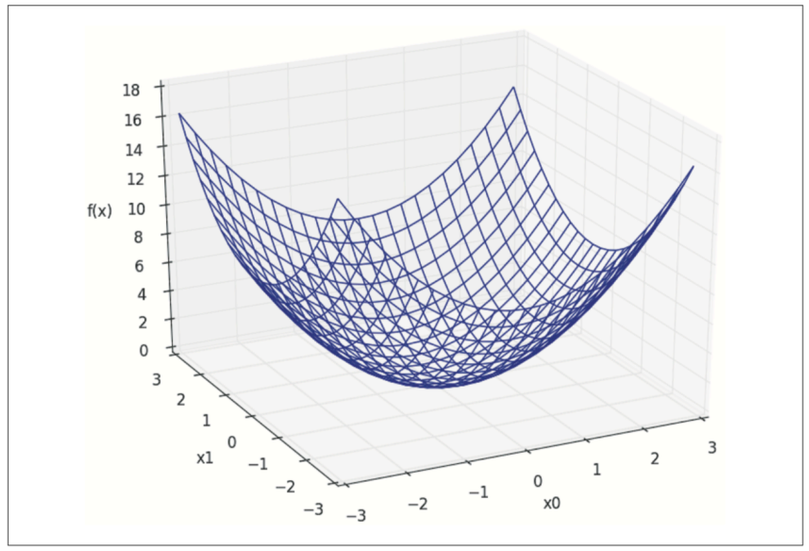


図4-8　$f(x_0, x_1) = x_0^2 + x_1^2$ のグラフ

(exercise)

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
```

(Point)
複数ある変数の中でターゲットの変数を
ひとつに絞り、他の変数はある値に固定

目次

勾配 (gradient)の実装

勾配：

すべての変数の偏微分をベクトルとしてまとめたもの $\left(\frac{\partial f}{\partial x_0}, \frac{\partial f}{\partial x_1}\right)$

```python
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
```

- np.zeros_like(x) は、xと同じ形状の配列で、その要素がすべて0の配列を生成
- numerical_gradient(f, x)関数は、引数のfは関数、xはNumPy配列

勾配を求める

点 (3, 4)、(0, 2)、(3, 0) での勾配を求める

```
>>> numerical_gradient(function_2, np.array([3.0, 4.0]))
array([ 6., 8.])
>>> numerical_gradient(function_2, np.array([0.0, 2.0]))
array([ 0., 4.])
>>> numerical_gradient(function_2, np.array([3.0, 0.0]))
array([ 6., 0.])
```

このように、(x0, x1) の各点における勾配を計算することが可能

この勾配は何を意味しているのかを理解するために、 $f(x_0, x_1) = x_0^2 + x_1^2$

の勾配を図示する（※勾配の結果にマイナスを付けたベクトルを描画）

勾配の図示

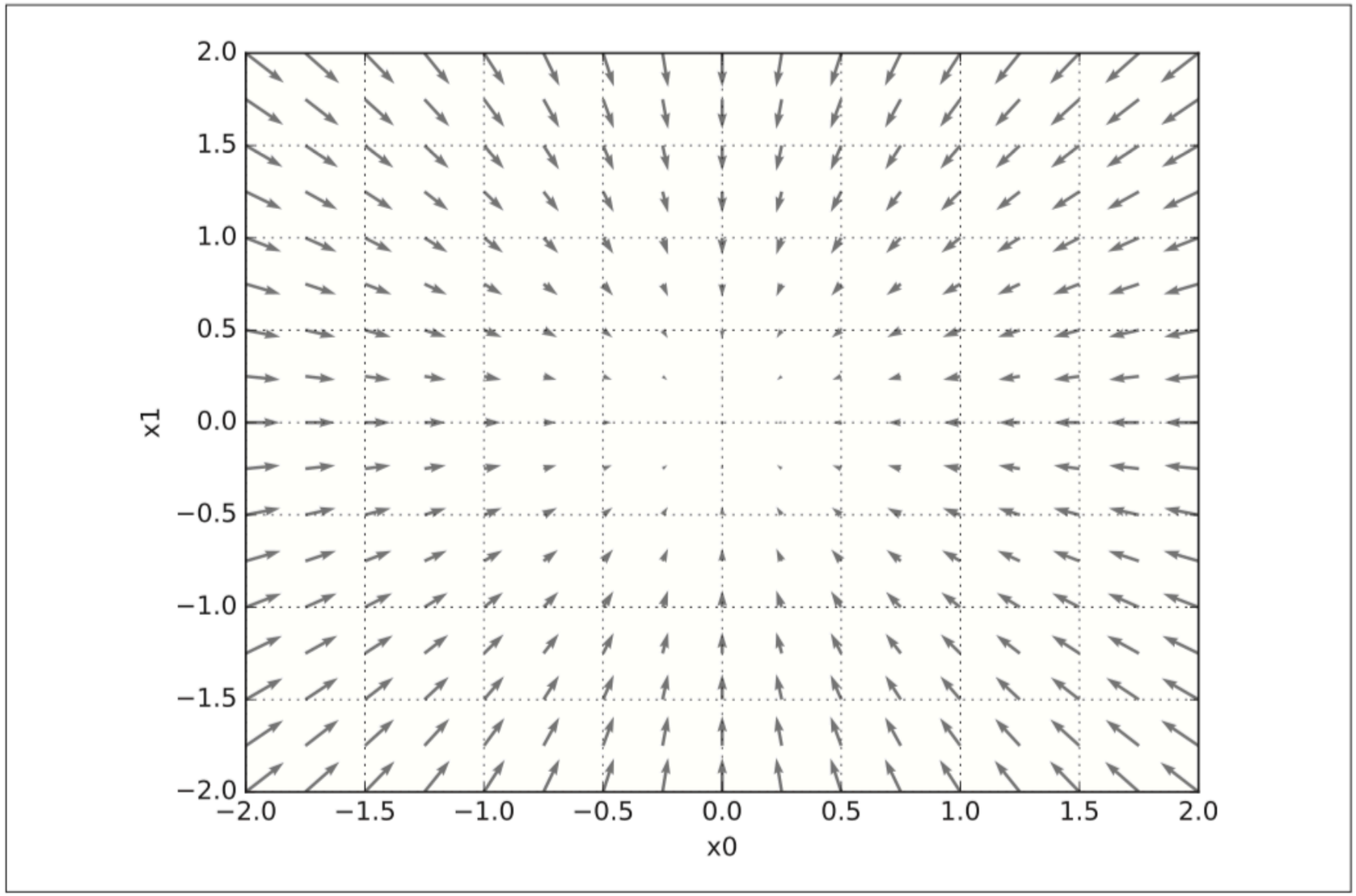


図 4-9　$f(x_0, x_1) = x_0^2 + x_1^2$ の勾配

- 勾配が示す方向は，各地点において関数の値を最も減らす方向 (最小値でない)
- 「一番低い場所」から遠く離れれば離れるほど、矢印の大きさも大きくなることが分かる

勾配法

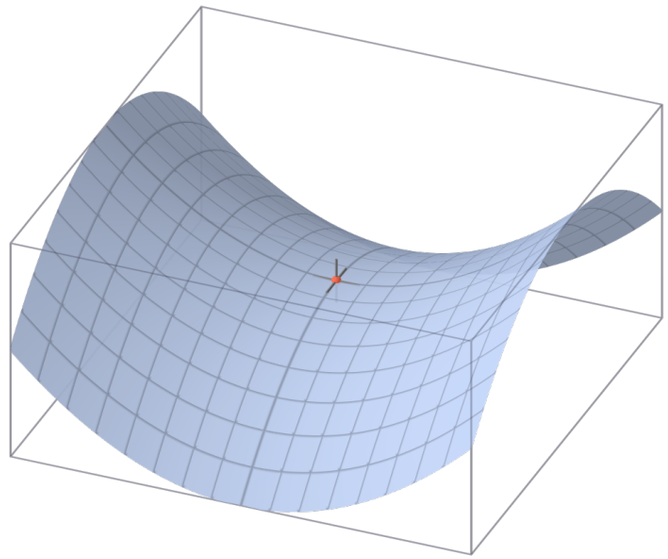
勾配法：

- 勾配をうまく利用して関数の最小値(または、できるだけ小さな値)を探索する手法

勾配法は勾配が0の場所を探すが，一般的に損失関数は複雑で，必ずしも最小値を探索できるとは限らない.

勾配が0になる点：

- **極小値**
  - 局所的な最小値
- **鞍点 (saddle point)**
  - ある方向で見れば極大値で，別の方向で見れば極小値となる点
- **最小値**



また，関数が複雑で歪な形をしていると、(ほとんど)平らな土地に入り込み、「プラトー」と呼ばれる学習が進まない停滞期に陥ることがある.

勾配法

勾配法：以下のように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

1. 現在の場所から勾配方向に一定の距離だけ進む
2. 移動した先でも同様に勾配を求める（→1に戻る）

- 勾配法は機械学習の最適化問題でよく使われる手法
- 特に，ニューラルネットワークの学習では勾配法がよく用いられる

- 最小値を探す場合：**勾配降下法(gradient descent method)**
- 最大値を探す場合：**勾配上昇法(gradient ascent method)**

※勾配が必ず最小値を指すとはかぎらないが，その方向に進むことで関数の値を最も減らせるため，最小値の場所を探す問題――もしくは，できるだけ小さな値となる関数の場所を探す問題――においては、勾配の情報を手がかりに，進む方向を決めるべき

勾配法の実装

$$x_0 = x_0 - \eta \frac{\partial f}{\partial x_0}$$

$$x_1 = x_1 - \eta \frac{\partial f}{\partial x_1}$$

η ：パラメータの更新量．学習率（learning rate）

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有

勾配降下法をpythonで実装

```
def gradient_descent(f, init_x, lr=0.01, step_num=100):
    x  = init_x

    for i in range(step_num):
        grad = numerical_gradient(f, x)
        x -= lr * grad

    return x
```

- f：最適化したい関数
- init_x：初期値
- lr：learning rate を意味する学習率
- step_num：勾配法による繰り返しの数

関数の勾配は，　numerical_gradient(f, x) で求め，
その勾配に学習率を掛けた値で更新する処理を step_num で指定された回数繰り返す

学習率を変えた勾配法の結果

問：$f(x_0, x_1) = x_0^2 + x_1^2$ の最小値を勾配法で求めよ。

```
init_x = np.array([-3.0, 4.0])
gradient_descent(function_2, init_x=init_x, lr=0.1, step_num=100)
```

```
array([ -6.11110793e-10,   8.14814391e-10])
```

```
# 学習率が大きすぎる例:lr=10.0
init_x = np.array([-3.0, 4.0])
gradient_descent(function_2, init_x=init_x, lr=10.0, step_num=100)
```

```
array([ -2.58983747e+13,  -1.29524862e+12])
```

```
# 学習率が小さすぎる例:lr=1e-10
init_x = np.array([-3.0, 4.0])
gradient_descent(function_2, init_x=init_x, lr=1e-10, step_num=100)
```

```
array([-2.99999994,  3.99999992])
```

- 学習率が大きすぎると，大きな値へと発散
- 学習率が小さすぎると，ほとんど更新されずに終了

→適切な学習率を設定することが重要

勾配法更新の可視化

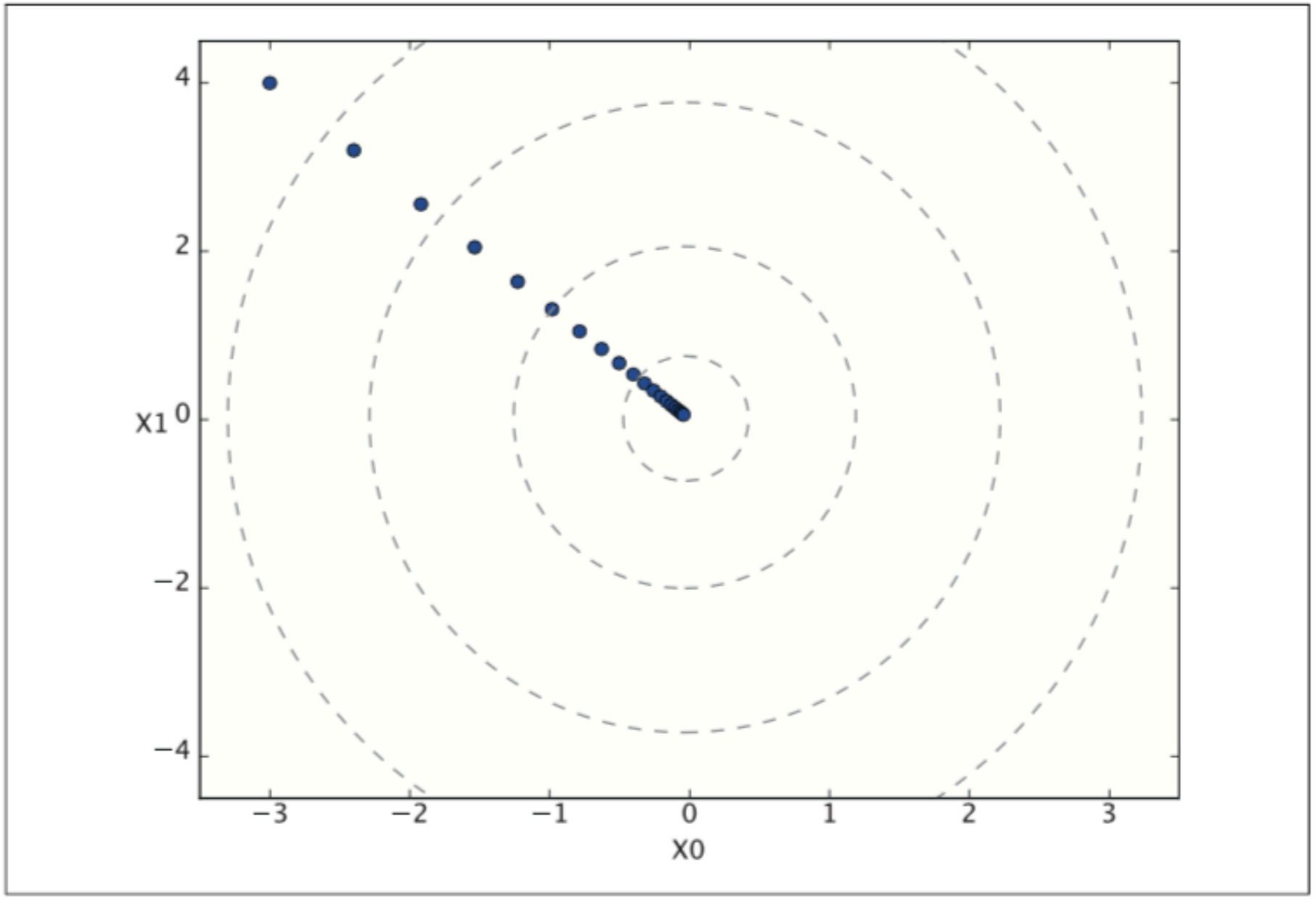


図4-10　$f(x_0, x_1) = x_0^2 + x_1^2$ の勾配法による更新のプロセス：破線は関数の等高線を示す

ハイパーパラメータ

学習率のようなパラメータはハイパーパラメータと言う.
ニューラルネットワークのパラメータ——重みやバイアス——とは性質が異なる.

ニューラルネットワークの重みパラメータ：
- 訓練データと学習アルゴリズムによって“自動”で獲得される

学習率のようなハイパーパラメータ：
- “手動”で設定が必要

一般的には，このハイパーパラメータをいろいろな値で試しながら，うまく学習できるケースを探すという作業が必要.

ニューラルネットワークに対する勾配

- NNにおいても勾配を用いる（重みパラメータに関する損失関数の勾配）

$$\mathbf{W} = \begin{pmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \end{pmatrix}$$

$$\frac{\partial L}{\partial \mathbf{W}} = \begin{pmatrix} \frac{\partial L}{\partial w_{11}} & \frac{\partial L}{\partial w_{12}} & \frac{\partial L}{\partial w_{13}} \\ \frac{\partial L}{\partial w_{21}} & \frac{\partial L}{\partial w_{22}} & \frac{\partial L}{\partial w_{23}} \end{pmatrix}$$

NNの勾配を求める実装を行っていく
まず，simpleNetクラスを実装

予測するためのメソッド predict(x),
損失関数の値を求めるためのメソッド loss(x, t)

```python
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
```

ニューラルネットワークに対する勾配


```
net = simpleNet()
net.W
```

```
array([[ 0.20416663, -0.26468752,  0.93918521],
       [-0.96143561, -0.73607466,  0.3411377 ]])
```

```
x = np.array([0.6, 0.9])
net.predict(x)
```

```
array([-0.74279207, -0.8212797 ,  0.87053506])
```

```
t = np.array([0, 0, 1]) # 正解ラベル
net.loss(x, t)
```

```
0.32455034192715981
```

ニューラルネットワークに対する勾配

- numerical_gradient(f, x) が内部で f(x) を実行するため，整合性がとれるようにf(W) を定義
- numerical_gradient(f, x) の引数 f は関数，x は関数 f への引数
- そのため，net.W を引数に取り，損失関数を計算する新しい関数 f を定義
- そして，その新しく定義した関数を，numerical_gradient(f, x) に渡す

```
1 def f(w):
2     return net.loss(x, t)
3
4 numerical_gradient(f, net.W)
```

```
array([[ 0.08640555,  0.0798831 , -0.16628865],
       [ 0.12960832,  0.11982465, -0.24943297]])
```

```
1 # これも可
2 f = lambda w: net.loss(x, t)
3 numerical_gradient(f, net.W)
```

```
array([[ 0.08640555,  0.0798831 , -0.16628865],
       [ 0.12960832,  0.11982465, -0.24943297]])
```

目次

学習アルゴリズムの実装

前提

ニューラルネットワークは、適応可能な重みとバイアスがあり、この重みとバイアスを訓練データに適応するように調整することを「学習」と呼ぶ。ニューラルネットワークの学習は次の４つの手順で行う。

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。

ミニバッチとして無作為に選ばれたデータを使用しているため,
確率的勾配降下法(stochastic gradient descent)と呼ばれる.

2層ニューラルネットワークのクラス


```python
# coding: utf-8
import sys, os
sys.path.append(os.pardir)  # 親ディレクトリのファイルをインポートするための設定
from common.functions import *
from common.gradient import numerical_gradient


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)

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

        return grads
```

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。

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。

2層ニューラルネットワークのクラス

__init__で，各層のニューロンの数を指定し，重みパラメータの初期化を行う.

※手書き数字認識を行う場合，入力画像サイズが 28 × 28 の計784個，出力は10個.
　そのため，input_size=784，output_size=10と指定，hidden_size は適当な値を設定.

重みパラメータの初期値は，ニューラルネットワークの学習を成功させる上で重要だが，
今回は，重みはガウス分布に従う乱数で初期化し，バイアスは 0 で初期化.

ミニバッチ学習の実装

ミニバッチ学習：
訓練データから無作為に一部のデータを取り出して――これをミニバッチという――，
そのミニバッチを対象に，勾配法によりパラメータを更新

実装手法：
1．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(画像データと正解ラベルデータ)を抽出

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

3．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，その値を配列に追加

この損失関数の値の推移をグラフで表示したのが次スライド

ミニバッチ学習の実装

繰り返しデータを浴びることによって、最適な重みパラメータへと徐々に近づく.

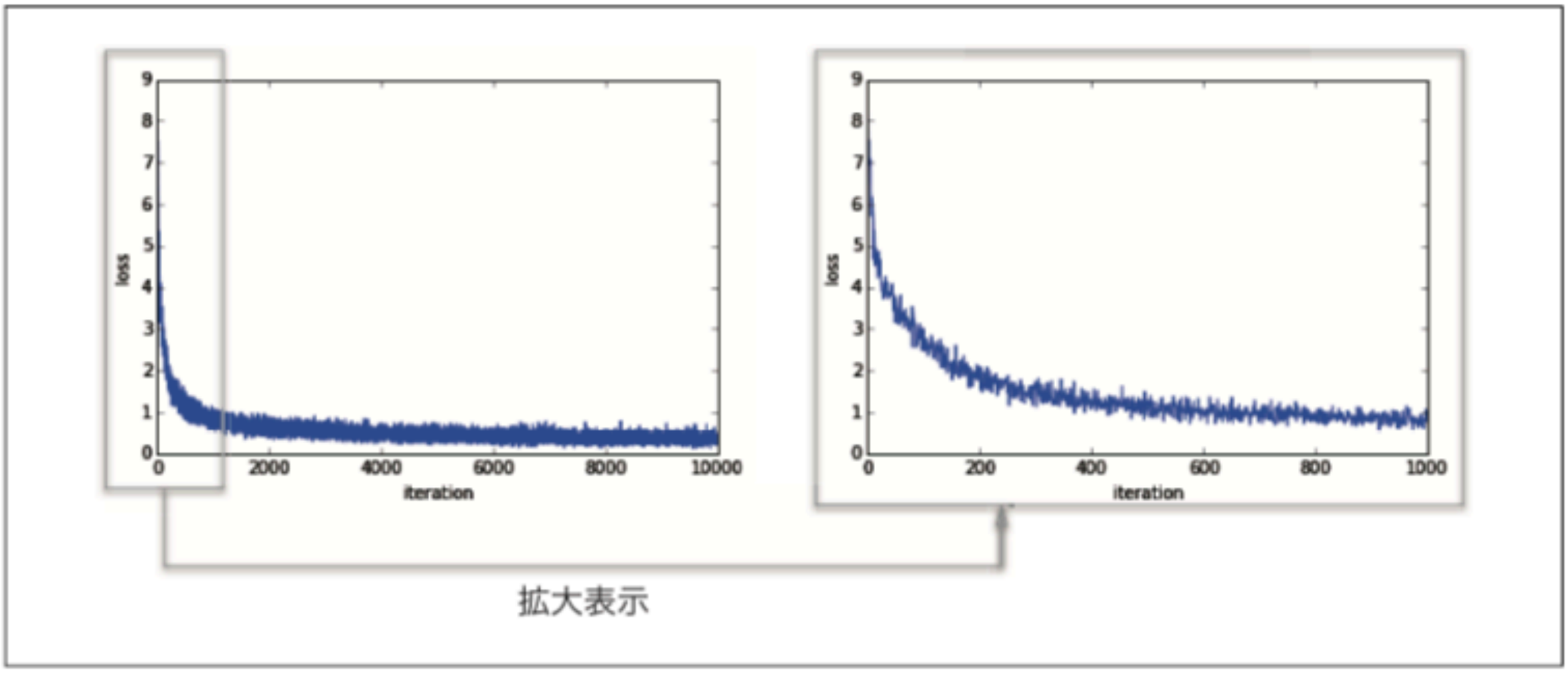


図4-11　損失関数の推移：左図は 10,000 イテレーションまでの推移、右図は 1,000 イテレーションまでの推移

テストデータで評価

学習を行う過程で，1エポック毎に訓練データとテストデータを対象に，認識精度を記録

1 エポックとは学習において訓練データをすべて使い切ったときの回数に対応

たとえば，10,000 個の訓練データに対して 100 個のミニバッチで学習する場合，
確率的勾配降下法を 100 回繰り返したら，すべての訓練データを“見た”ことになる．
この場合，100 回= 1 エポック

テストデータで評価

訓練データの 1 エポックごとに認識精度の経過を記録

2つの認識精度には差がない (2 つの線はほぼ重なっている).
→ 今回の学習では過学習が起きていないことがわかる.

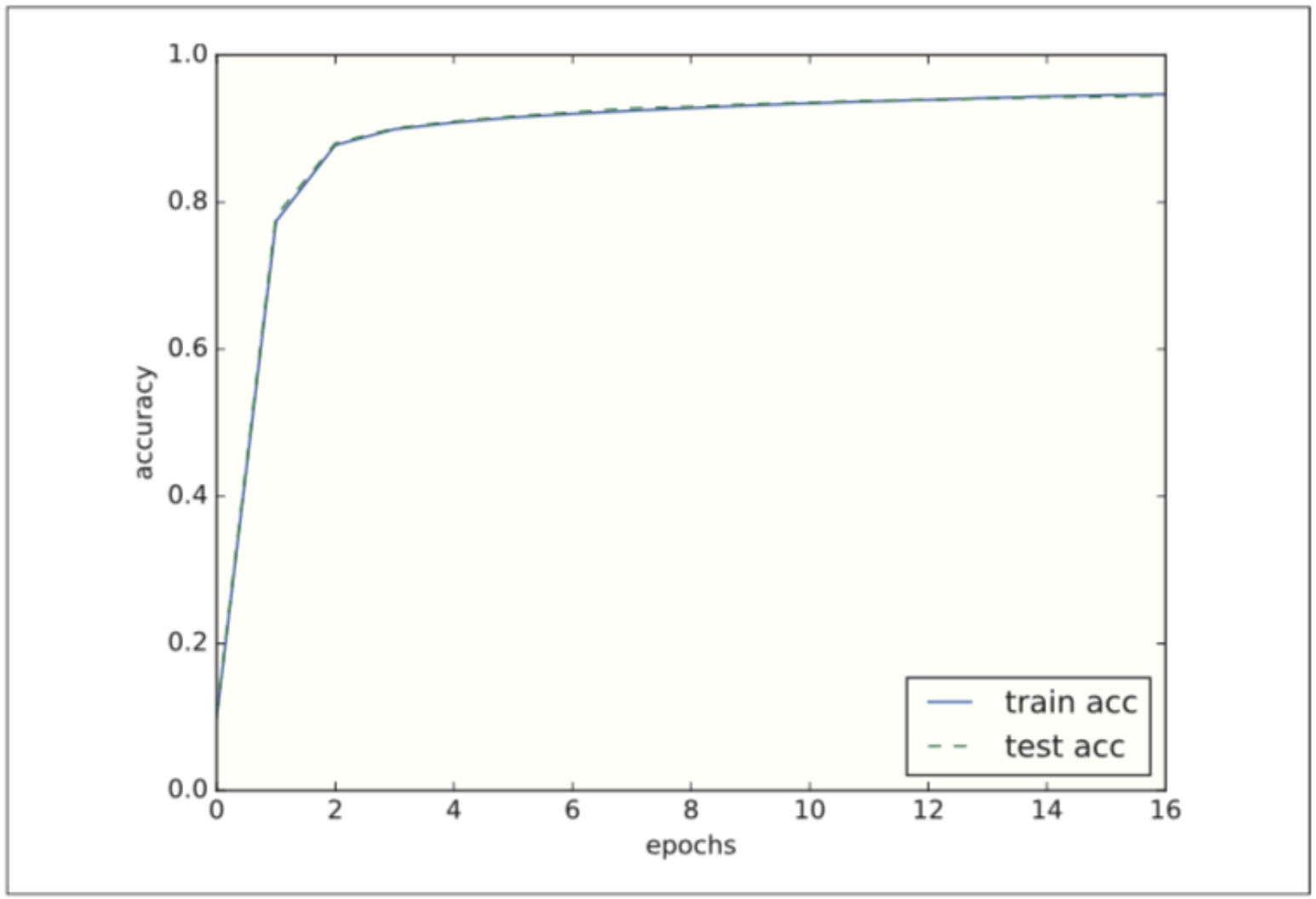


図 4-12　訓練データとテストデータに対する認識精度の推移。横軸はエポック

目次

まとめ

- 機械学習で使用するデータセットは，訓練データとテストデータに分けて使用.
- 訓練データで学習を行い，学習したモデルの汎化能力をテストデータで評価.
- ニューラルネットワークの学習は，損失関数を指標として，損失関数の値が小さくなるように，重みパラメータを更新.
- 重みパラメータを更新する際には，重みパラメータの勾配を利用して，勾配方向に重みの値を更新する作業を繰り返す.
- 微小な値を与えたときの差分によって微分を求めることを数値微分と言う.
- 数値微分によって，重みパラメータの勾配を求めることができる。.
- 数値微分による計算には時間がかかるが，その実装は簡単である．一方，次章で実装するやや複雑な誤差逆伝播法は，高速に勾配を求めることができる.