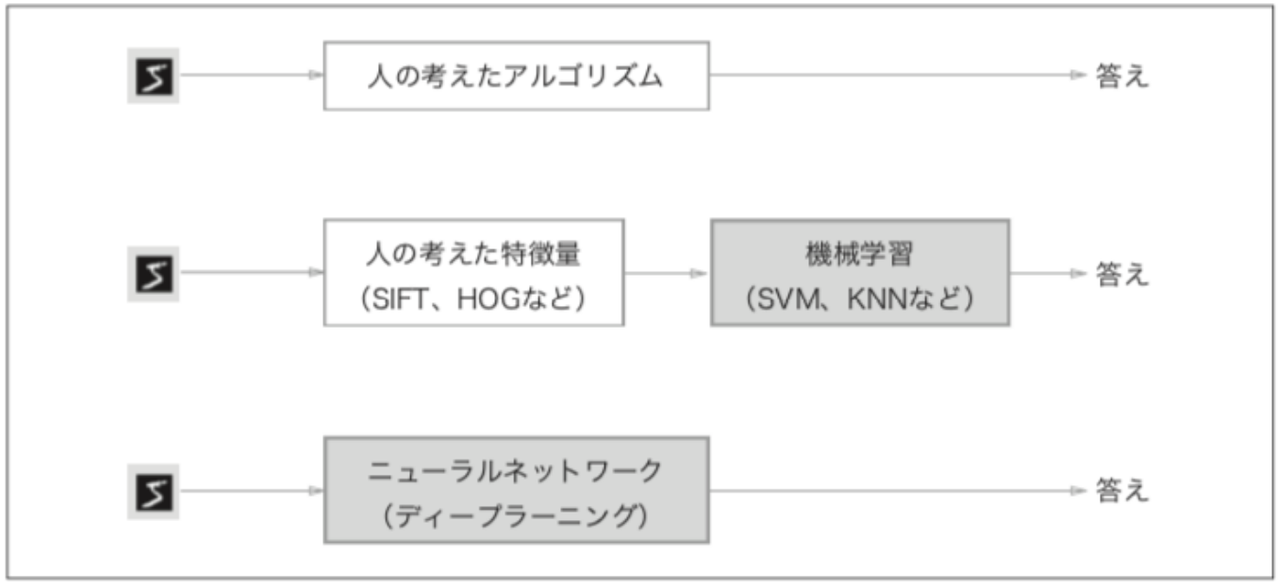
Chapter 4

28,May,2020 藤本

ニューラルネットワークの学習

- データから学習してパラメータを決める！



データセット

訓練データ（教師データ）

テストデータ

※ あるデータセットにだけ過度に対応した状態を過学習という

損失関数 (loss function)

- ニューラルネットワークの性能の指標

２乗和誤差

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

NNの出力　教師データ

```
 1 import numpy as np
 2
 3 y = [0.1, 0.05, 0.6, 0.0, 0.05, 0.1, 0.0, 0.1, 0.0, 0.0]
 4 y1 = [0.1, 0.05, 0.1, 0.0, 0.05, 0.1, 0.0, 0.6, 0.0, 0.0]
 5 t = [0, 0, 1, 0, 0, 0, 0, 0, 0, 0]
 6
 7 def mean_squared_error(y, t):
 8         return 0.5 * np.sum((y-t)**2)
 9
10 results = mean_squared_error(np.array(y), np.array(t))
11 print(results)
12
13 results = mean_squared_error(np.array(y1), np.array(t))
14 print(results)
15
```

```
0.0975
0.5975
```

y(1個目)の方が損失関数が小さい
＝　誤差が小さい
＝　出力が教師データにより適合

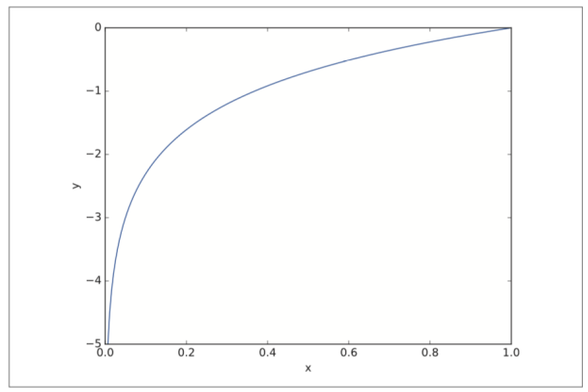
- 損失関数が小さくなるように、重みパラメータを調節する。

損失関数 (loss function)

- ニューラルネットワークの性能の指標

交差エントロピー誤差　$$E = -\sum_k t_k \log y_k$$

正解ラベル (正解は1)　　NNの出力




```python
import numpy as np

y = [0.1, 0.05, 0.6, 0.0, 0.05, 0.1, 0.0, 0.1, 0.0, 0.0]
y1 = [0.1, 0.05, 0.1, 0.0, 0.05, 0.1, 0.0, 0.6, 0.0, 0.0]
t = [0, 0, 1, 0, 0, 0, 0, 0, 0, 0]

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

results = cross_entropy_error(np.array(y), np.array(t))
print(results)

results = cross_entropy_error(np.array(y1), np.array(t))
print(results)
```

NNの出力が大きいほど0に近くなる

```
0.510825457099
2.30258409299
```

- 損失関数が小さくなるように、重みパラメータを調節する。

ミニバッチ学習

- 訓練データの全てに対して損失関数を求めなければならない

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

np.random.choice(60000, 10)

和をとって、正規化する。「平均の損失関数を求める」

- 無作為にサンプリングして学習を行う→ミニバッチ学習

ミニバッチ学習

```python
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)
#ここまででデータ読み込み

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]
#ランダムに取り出す

print(x_batch)
print(t_batch)


```

```
[21872 14743 17033 17534 14096 28520 19748 26841 32
[[ 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.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  1.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  1.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  1.  0.  0.  0.]
 [ 1.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  1.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  1.  0.  0.]
 [ 1.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  1.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  1.  0.  0.  0.  0.  0.  0.]]
```

ミニバッチ対応版
交差エントロピー誤差

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

```
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
```

このサイトおもしろい

```
In [1]: import numpy as np

In [2]: a = np.arange(12) # 1つ1次元配列を生成。

In [3]: a
Out[3]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])

In [4]: b = np.reshape(a, (3, 4)) # 3×4の2次元配列に変形。

In [5]: b  # しっかり変形ができているか確認。
Out[5]:
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

yはNNの出力、tは教師データ
yの次元が1、つまりデータひとつあたりの誤差を求める場合は、データの形状を整形する。
で、バッチの枚数で正規化し、1枚あたりの誤差を計算する

損失関数の意義

- 認識精度じゃなくて、損失関数を指標に

  認識精度は不連続で、重みパラメータで微分してもほとんど0になってしまう
  (認識精度 = 100枚のうち32枚正しい、で32%)

  損失関数は連続で、微分が0にならない

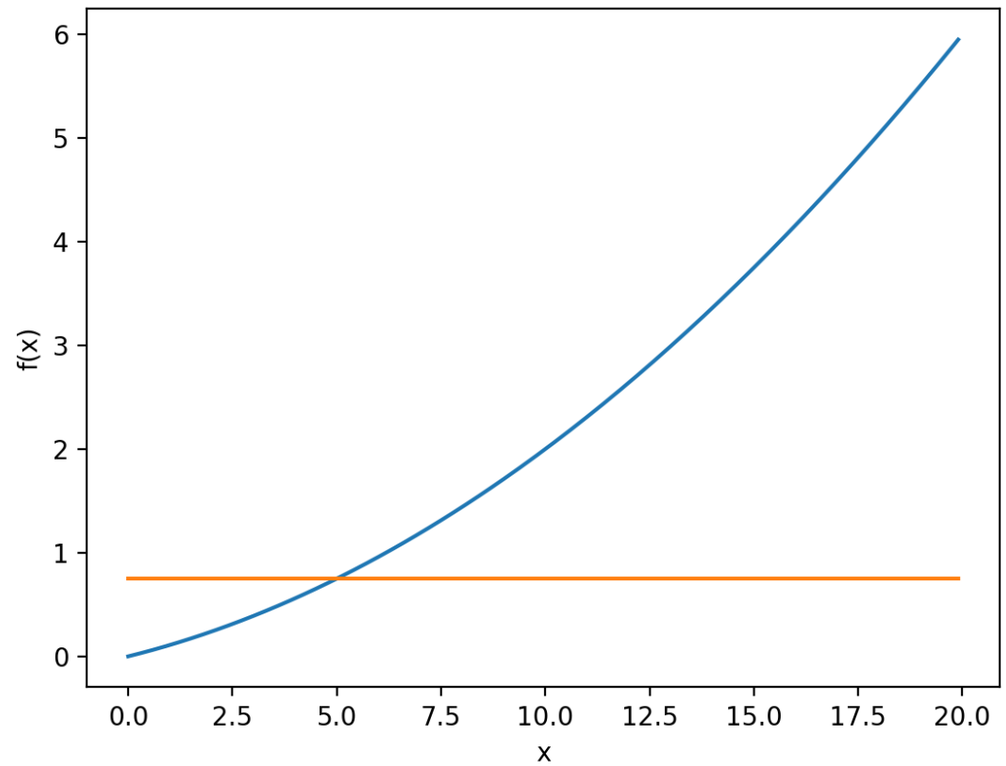
- NNで重みパラメータを更新する際、重みパラメータの勾配を利用して、勾配方向に重みの値を更新する作業を繰り返す

微分を使う

- 微分を実装する

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

```python
import numpy as np
import matplotlib.pylab as plt

def numerical_diff(f, x):
    h = 10e-50 #(1)ここがダメ、差が小さすぎ
    return (f(x+h) - f(x)) / h

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

def tangent_line(f, x):
    d = numerical_diff(f, x)
    print(d)
    y = f(x) - d*x
    return lambda t: d*t + y

x = np.arange(0.0, 20.0, 0.1)
y = function_1(x)
plt.xlabel("x")
plt.ylabel("f(x)")

print(x)
print(y)

tf = tangent_line(function_1, 5)
y2 = tf(x)

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



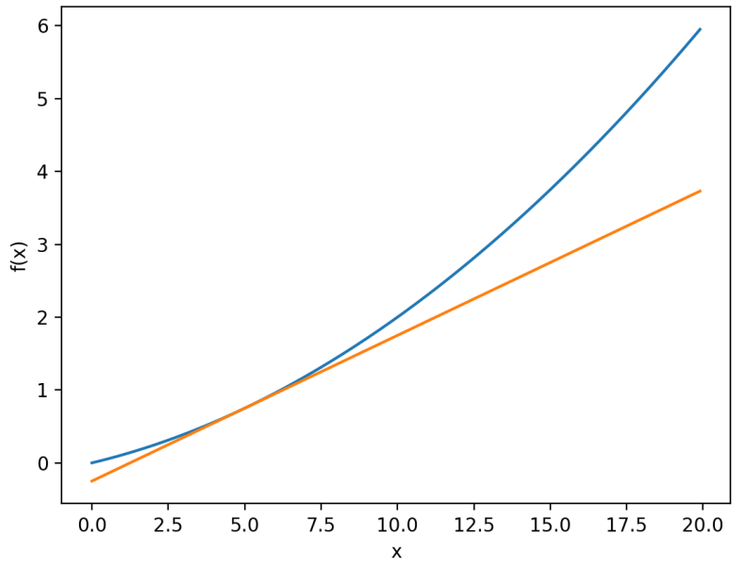
hは、丸め誤差を考えて **10e-4**くらいを使う

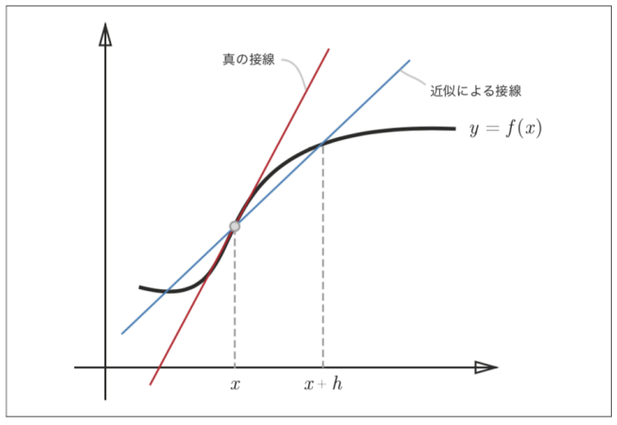
微分を使う

- 微分を実装する

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

```python
import numpy as np
import matplotlib.pylab as plt

def numerical_diff(f, x):
    h = 1e-4
    return (f(x+h) - f(x)) / h #(2)ここがダメ、差分の計算方法を工夫する

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

def tangent_line(f, x):
    d = numerical_diff(f, x)
    print(d)
    y = f(x) - d*x
    return lambda t: d*t + y

x = np.arange(0.0, 20.0, 0.1)
y = function_1(x)
plt.xlabel("x")
plt.ylabel("f(x)")

print(x)
print(y)

tf = tangent_line(function_1, 5)
y2 = tf(x)

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





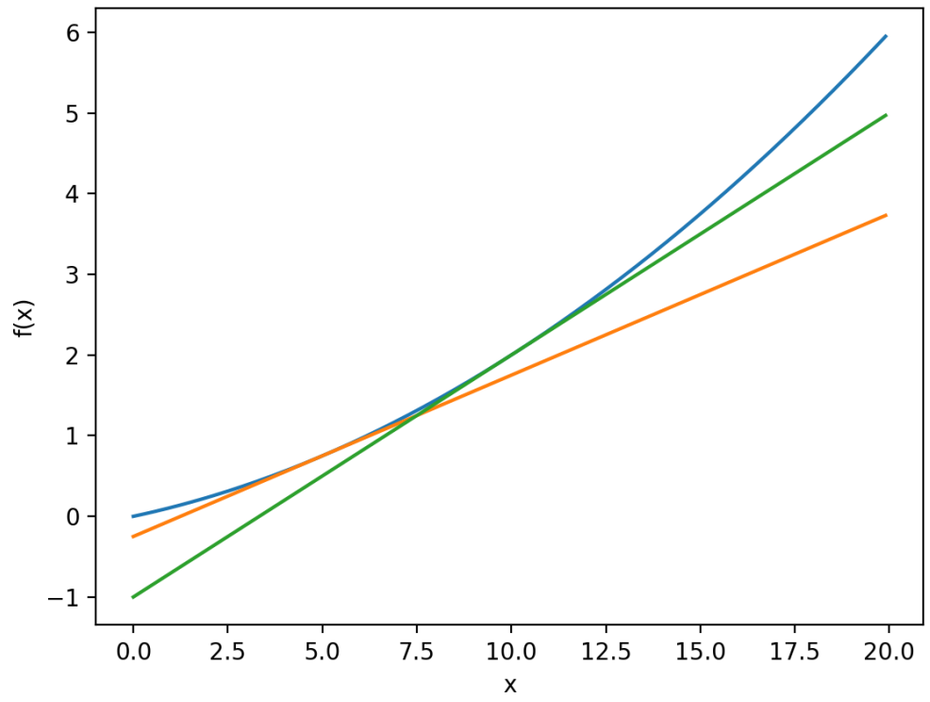
中心差分をとる。

微分を使う

- 微分を実装する

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

```python
# coding: utf-8
import numpy as np
import matplotlib.pylab as plt


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


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


def tangent_line(f, x):
    d = numerical_diff(f, x)
    print(d)
    y = f(x) - d*x
    return lambda t: d*t + y

x = np.arange(0.0, 20.0, 0.1)
y = function_1(x)
plt.xlabel("x")
plt.ylabel("f(x)")

tf = tangent_line(function_1, 5)
tf2 = tangent_line(function_1, 10)
y2 = tf(x)
y3 = tf2(x)

plt.plot(x, y)
plt.plot(x, y2)
plt.plot(x, y3)
plt.show()
```

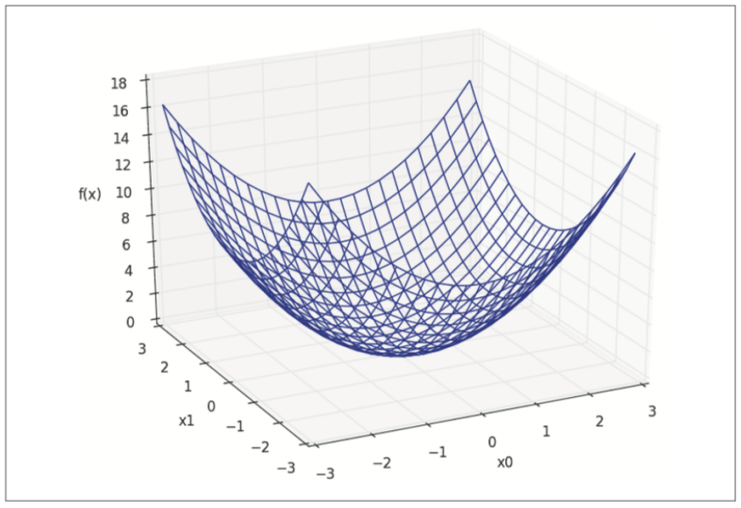



```
0.1999999999990898
0.2999999999986347
```

解析的に微分した場合(2.0,3.0)に近い値

偏微分

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



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)という

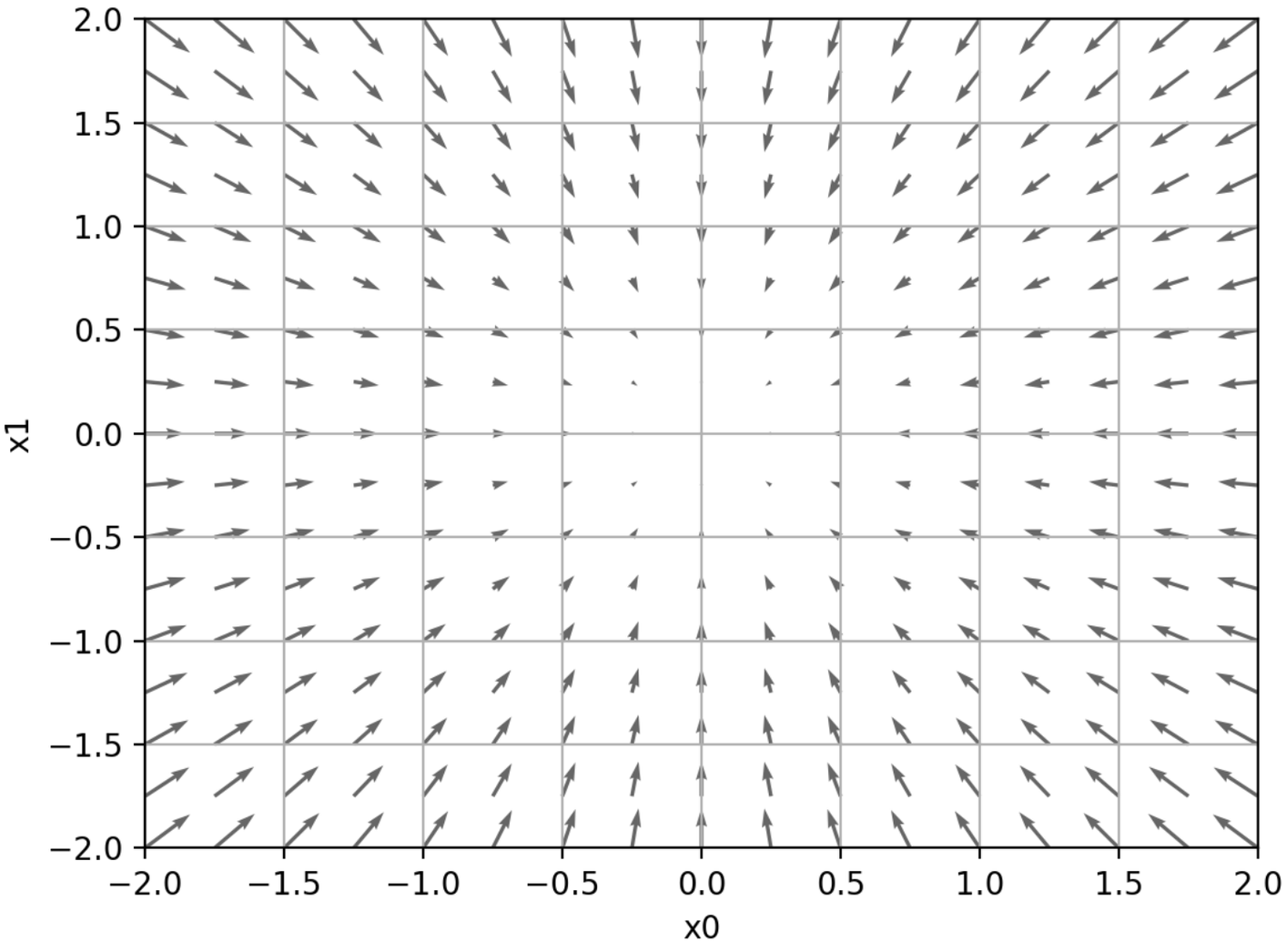
```python
import numpy as np
import matplotlib.pylab as plt
from mpl_toolkits.mplot3d import Axes3D

def function_2(x):
    return np.sum(x**2)

def numerical_gradient(f, x):
    h = 1e-4  # 0.0001
    grad = np.zeros_like(x) #xと同じ形状で、要素が0の配列をつくっておく

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

        x[idx] = tmp_val - h
        fxh2 = f(x)  # f(x-h)
        grad[idx] = (fxh1 - fxh2) / (2*h) #つくっておいたgradに計算結果を入れる

        x[idx] = tmp_val  # 値を元に戻す

    return grad


print(numerical_gradient(function_2, np.array([3.0, 4.0])))

```

```
2
3.0
3.0001
4.0
4.0001
[ 6.  8.]
```

勾配

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

それぞれのx0,x1での勾配の図 by gradient_2d.py
勾配は、低いところを指す




```
51
52 if __name__ == '__main__':
53     x0 = np.arange(-2, 2.5, 0.25)
54     x1 = np.arange(-2, 2.5, 0.25)
55     X, Y = np.meshgrid(x0, x1)
56
57     X = X.flatten()
58     Y = Y.flatten()
59
60     grad = numerical_gradient(function_2, np.array([X, Y]).T).T
61
62     plt.figure()
63     plt.quiver(X, Y, -grad[0], -grad[1],  angles="xy",color="#666666")
64     plt.xlim([-2, 2])
65     plt.ylim([-2, 2])
66     plt.xlabel('x0')
67     plt.ylabel('x1')
68     plt.grid()
69     plt.draw()
70     plt.show()
71
```

[1.5,-2.0]だと[3,-4]
ベクトルのスケールは自動調節されている

勾配法

- 勾配の方向に一定値進むことを繰り返して、関数の最小値か、より小さい値を探す

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

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

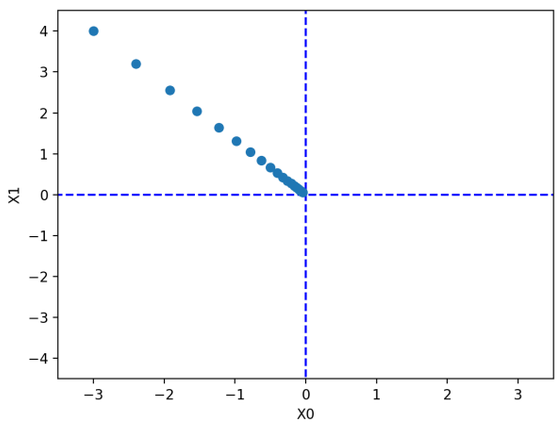
学習率 → 適切な値を探す必要がある。大きすぎると発散するし小さすぎると進まない

```python
import numpy as np

def function_2(x):
    return x[0]**2 + x[1]**2


def numerical_gradient(f, x):
    h = 1e-4  # 0.0001
    grad = np.zeros_like(x) #xと同じ形状で、要素が0の配列をつくっておく

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

        x[idx] = tmp_val - h
        fxh2 = f(x)  # f(x-h)
        grad[idx] = (fxh1 - fxh2) / (2*h) #つくっておいたgradに計算結果を入れる

        x[idx] = tmp_val  # 値を元に戻す

    return grad

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


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

```
[ -6.11110793e-10   8.14814391e-10]
```

ほぼゼロ (正しい)

※gradient_method.pyにもコードがある



NNにおける勾配

- 重みパラメータに関する損失関数の勾配

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

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

```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
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)

    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

x = np.array([0.6, 0.9])
t = np.array([0, 0, 1])

net = simpleNet()

f = lambda w: net.loss(x, t)
dW = numerical_gradient(f, net.W)

print(dW)
```

gradient_simplenet.pyで実装

NNにおける勾配

- 重みパラメータに関する損失関数の勾配

```
>>> dW = numerical_gradient(f, net.W)
>>> print(dW)
[[ 0.21924763  0.14356247 -0.36281009]
 [ 0.32887144  0.2153437  -0.54421514]]
```

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

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

2行3列の配列

つまり、w_11をhだけ増やすと、損失関数の値は0.2hだけ増える
損失関数を減らしたいので、w_11はマイナス方向に更新するのが良い

学習アルゴリズム

- 確率的勾配降下法 (stochastic gradient descent)

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。

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する。
- 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する。
- 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する。
- 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。