

▼ 深層学習day1

Section1: 入力層～中間層

- 入力層
 - 何らかの入力を受け取る部分 ⇒ ノード x
 - ノードにどのくらい値を使うか ⇒ 重み w
 - 入力の全体をずらす ⇒ バイアス (切片)

■ 数式で表すと

$$x = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix}, w = \begin{pmatrix} w_1 \\ \vdots \\ w_n \end{pmatrix}$$

⇒ 傾きと切片をどのようにして学習するか？

- 中間層へ値を渡す

- 中間層

- 入力層から受け取った値を処理する

■ 数式で表すと

$$\begin{aligned} u &= w_1x_1 + w_2x_2 + \dots + w_nx_n + b \\ &= Wx + b \end{aligned}$$

- 中間層で処理した値 ⇒ 出力層へ z

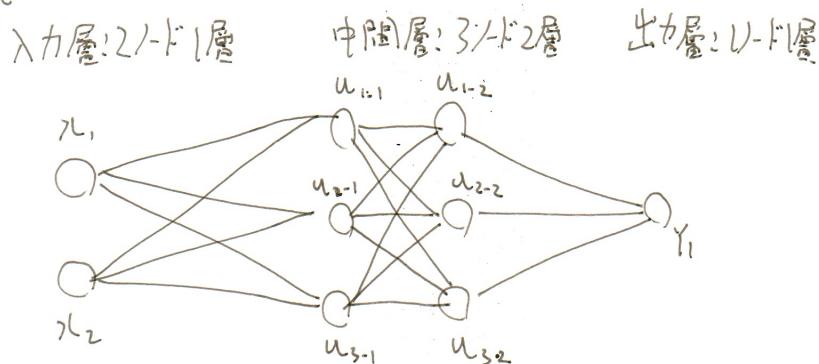
確認テスト 1.

ディープラーニングは、何をやるといいのか。
入力から出力を得るために、式を求める。

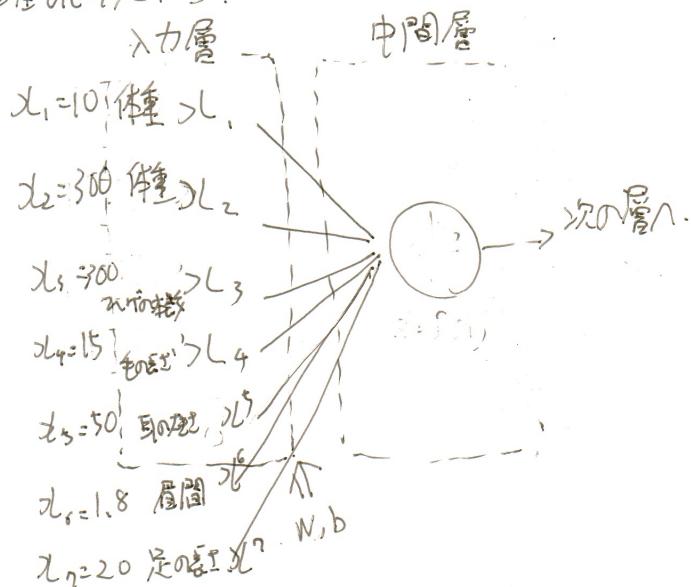
どの値の最適化が最終目的か。

- ③重み ④バイアス。

確認テスト 2.



確認テスト 3.



確認テスト 4.

$$u = \text{np.dot}(x, W) + b$$

➡ フワード・フォワード

総入力

 $u = \text{np.dot}(x, W) + b$

確認テスト5

1-1のファイルから中間層を定義しているソースを抜き出せ

```
# 2層の総入力
```

```
u2 = np.dot(z1, W2) + b2
```

```
# 2層の総出力
```

```
z2 = functions.relu(u2)
```

▼ Section2:活性化関数

ニューラルネットワークにおいて、次の層への出力の大きさを決める「**非線形の関数**」

- 線形 直線
 - 加法性、齊次性を満たす

- 非線形 曲線
 - 加法性、齊次性を満たさない

- 活性化関数

$$f^{(l)}(u^{(l)}) = \begin{bmatrix} f^{(l)}(u_1^{(l)}) & \dots & f^{(l)}(u_j^{(l)}) \end{bmatrix} l: \text{層のインデックス} j: \text{中間層ノードのインデックス}$$

l: 層のインデックス

j: 中間層ノードのインデックス

- 線形な処理に「活性化関数を通す」ことにより、パラエティに富んだ表現が可能となる。

- 活性化関数の大きな括りとして（有名なもの）

- 中間層用の活性化関数

- ReLU関数
 - シグモイド関数

- 出力層用の活性化関数

- ソフトマックス関数
 - 恒等関数
 - シグモイド関数

- 活性化関数：ステップ関数(現在はあまり使われない)

- ある値になったら値として1を出力
 - ある値になるまでは0を出力

- 活性化関数：シグモイド関数(ステップ関数の改良)

- 値の変化をなだらかになる
 - 勾配消失問題がある⇒ReLU関数へ

- 活性化関数：ReLU関数(シグモイド関数の改良)

- 勾配消失問題をある程度改善する
 - スパース化に貢献

確認テスト6：線形と非線形の図示

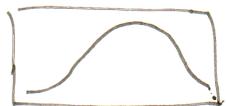
SECTION 2.

確認テスト6.

線形



非線形



確認テスト7：活性化関数を使っている個所

1層の総出力

`z1 = functions.relu(u1)`

ReLU関数

```
def relu(x):  
    return np.maximum(0, x)
```

- 実装演習

- ▼ 準備

- ▼ Google ドライブのマウント

```
from google.colab import drive
drive.mount('/content/drive')

Mounted at /content/drive
```

- ▼ sys.path の設定

以下では、Google ドライブのマイドライブ直下にDNN_code フォルダを置くことを仮定しています。必要に応じて、パスを変更してください。

```
import sys
sys.path.append('/content/drive/My Drive/')
```

- ▼ import と関数定義

```
import numpy as np
from common import functions

def print_vec(text, vec):
    print("*** " + text + " ***")
    print(vec)
    #print('shape: ' + str(x.shape))
    print("")
```

- ▼ 順伝播（単層・単ユニット）

```
# 順伝播（単層・単ユニット）

# 重み
W = np.array([[0.1], [0.2]])

## 試してみよう_配列の初期化
## W = np.zeros(2)
## W = np.ones(2)
## W = np.random.rand(2)
## W = np.random.randint(5, size=(2))

print_vec("重み", W)

# バイアス
b = 0.5

## 試してみよう_数値の初期化
## b = np.random.rand() # 0~1のランダム数値
## b = np.random.rand() * 10 - 5 # -5~5のランダム数値

print_vec("バイアス", b)

# 入力値
x = np.array([2, 3])
print_vec("入力", x)

# 総入力
u = np.dot(x, W) + b
print_vec("総入力", u)

# 中間層出力
z = functions.relu(u)
print_vec("中間層出力", z)

*** 重み ***
*** バイアス ***
*** 入力 ***
*** 総入力 ***
*** 中間層出力 ***
```

```
[[0.1]
 [0.2]]
```

```
*** バイアス ***
0.5
```

```
*** 入力 ***
[2 3]
```

```
*** 総入力 ***
[1.3]
```

```
*** 中間層出力 ***
[1.3]
```

▼ 順伝播（単層・複数ユニット）

```
# 順伝播（単層・複数ユニット）
```

```
# 重み
W = np.array([
    [0.1, 0.2, 0.3],
    [0.2, 0.3, 0.4],
    [0.3, 0.4, 0.5],
    [0.4, 0.5, 0.6]
])
```

```
## 試してみよう_配列の初期化
W = np.zeros((4,3))
print_vec("重み", W)
W = np.ones((4,3))
print_vec("重み", W)
W = np.random.rand(4,3)
print_vec("重み", W)
W = np.random.randint(5, size=(4,3))
```

```
print_vec("重み", W)
```

```
# バイアス
b = np.array([0.1, 0.2, 0.3])
print_vec("バイアス", b)
```

```
# 入力値
x = np.array([1.0, 5.0, 2.0, -1.0])
print_vec("入力", x)
```

```
# 総入力
u = np.dot(x, W) + b
print_vec("総入力", u)
```

```
# 中間層出力
z = functions.sigmoid(u)
print_vec("中間層出力", z)
```

```
*** 重み ***
[[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]
```

```
*** 重み ***
[[1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]]
```

```
*** 重み ***
[[0.13343659 0.54257807 0.23512802
 [0.41895356 0.47318455 0.02504533]
 [0.48372994 0.52941381 0.75658592]
 [0.17189697 0.70466918 0.48719085]]
```

```
*** 重み ***
[[1 3 0]
 [4 3 4]
 [2 2 0]
 [2 3 3]]
```

```
*** バイアス ***
[0.1 0.2 0.3]
```

```
*** 入力 ***
[ 1. 5. 2. -1.]
```

```

*** 総入力 ***
[23.1 19.2 17.3]

*** 中間層出力 ***
[1.          1.          0.9999997]

```

▼ 順伝播（3層・複数ユニット）

```

# 順伝播（3層・複数ユニット）

# ウェイトとバイアスを設定
# ネットワークを作成
def print_shape(vec):
    print("shape:" + str(vec.shape))

def init_network():
    print("##### ネットワークの初期化 #####")
    network = {}

    # 試してみよう
    # 各パラメータのshapeを表示
    # ネットワークの初期値ランダム生成

    """
    network['W1'] = np.array([
        [0.1, 0.3, 0.5],
        [0.2, 0.4, 0.6]
    ])
    network['W2'] = np.array([
        [0.1, 0.4],
        [0.2, 0.5],
        [0.3, 0.6]
    ])
    network['W3'] = np.array([
        [0.1, 0.3],
        [0.2, 0.4]
    ])

    network['b1'] = np.array([0.1, 0.2, 0.3])
    network['b2'] = np.array([0.1, 0.2])
    network['b3'] = np.array([1, 2])
    """

    network['W1'] = np.random.rand(2, 3)
    network['W2'] = np.random.rand(3, 2)
    network['W3'] = np.random.rand(2, 2)
    network['b1'] = np.random.rand(3)
    network['b2'] = np.random.rand(2)
    network['b3'] = np.random.randint(3, size=(2))

    print_vec("重み1", network['W1'])
    print_shape(network['W1'])
    print_vec("重み2", network['W2'])
    print_shape(network['W2'])
    print_vec("重み3", network['W3'])
    print_shape(network['W3'])
    print_vec("バイアス1", network['b1'])
    print_shape(network['b1'])
    print_vec("バイアス2", network['b2'])
    print_shape(network['b2'])
    print_vec("バイアス3", network['b3'])
    print_shape(network['b3'])

    return network

# プロセスを作成
# x : 入力値
def forward(network, x):

    print("##### 順伝播開始 #####")

    W1, W2, W3 = network['W1'], network['W2'], network['W3']
    b1, b2, b3 = network['b1'], network['b2'], network['b3']

    # 1層の総入力
    u1 = np.dot(x, W1) + b1

    # 1層の総出力
    z1 = functions.relu(u1)

    # 2層の総入力
    u2 = np.dot(z1, W2) + b2

```

```

uz = np.dot(z1, W2) + b2
# 2層の総出力
z2 = functions.relu(uz)

# 出力層の総入力
u3 = np.dot(z2, W3) + b3

# 出力層の総出力
y = u3

print_vec("総入力1", u1)
print_vec("中間層出力1", z1)
print_vec("総入力2", u2)
print_vec("出力1", z1)
print("出力合計: " + str(np.sum(z1)))

return y, z1, z2

# 入力値
x = np.array([1., 2.])
print_vec("入力", x)

# ネットワークの初期化
network = init_network()

y, z1, z2 = forward(network, x)

*** 入力 ***
[1. 2.]

##### ネットワークの初期化 #####
*** 重み1 ***
[[0.9735346 0.34390745 0.23283616]
 [0.18677336 0.75651807 0.12397265]]

shape:(2, 3)
*** 重み2 ***
[[0.65357042 0.58716275]
 [0.10512691 0.71713181]
 [0.07635605 0.04239831]]

shape:(3, 2)
*** 重み3 ***
[[0.21275785 0.78816987]
 [0.07691365 0.3860465 ]]

shape:(2, 2)
*** バイアス1 ***
[0.76847189 0.45435361 0.24362726]

shape:(3,)
*** バイアス2 ***
[0.88911249 0.99524731]

shape:(2,)
*** バイアス3 ***
[2 0]

shape:(2,)
##### 順伝播開始 #####
*** 総入力1 ***
[2.11555322 2.3112972 0.72440872]

*** 中間層出力1 ***
[2.11555322 2.3112972 0.72440872]

*** 総入力2 ***
[2.57006799 3.9256398 ]

*** 出力1 ***
[2.11555322 2.3112972 0.72440872]

出力合計: 5.151259137763849

```

▼ 多クラス分類 (2-3-4ネットワーク)

```

# 多クラス分類
# 2-3-4ネットワーク

# ! 試してみよう_ノードの構成を 3-5-4 に変更してみよう

# ウエイトとバイアスを設定
# ネットワークを作成
def init_network():
    print("##### ネットワークの初期化 #####")

```

```

#試してみよう
#_各パラメータのshapeを表示
#_ネットワークの初期値ランダム生成

"""
network = {}
network['W1'] = np.array([
    [0.1, 0.3, 0.5],
    [0.2, 0.4, 0.6]
])
network['W2'] = np.array([
    [0.1, 0.4, 0.7, 1.0],
    [0.2, 0.5, 0.8, 1.1],
    [0.3, 0.6, 0.9, 1.2]
])
network['b1'] = np.array([0.1, 0.2, 0.3])
network['b2'] = np.array([0.1, 0.2, 0.3, 0.4])
"""

network['W1'] = np.random.rand(3, 5)
network['W2'] = np.random.rand(5, 4)
network['b1'] = np.random.rand(5)
network['b2'] = np.random.rand(4)

print_vec("重み1", network['W1'])
print_shape(network['W1'])
print_vec("重み2", network['W2'])
print_shape(network['W2'])
print_vec("バイアス1", network['b1'])
print_shape(network['b1'])
print_vec("バイアス2", network['b2'])
print_shape(network['b2'])

return network

# プロセスを作成
# x : 入力値
def forward(network, x):

    print("##### 順伝播開始 #####")
    W1, W2 = network['W1'], network['W2']
    b1, b2 = network['b1'], network['b2']

    # 1層の総入力
    u1 = np.dot(x, W1) + b1

    # 1層の総出力
    z1 = functions.relu(u1)

    # 2層の総入力
    u2 = np.dot(z1, W2) + b2

    # 出力値
    y = functions.softmax(u2)

    print_vec("総入力1", u1)
    print_vec("中間層出力1", z1)
    print_vec("総入力2", u2)
    print_vec("出力1", y)
    print("出力合計: " + str(np.sum(y)))

    return y, z1

## 事前データ
# 入力値
x = np.array([1., 2., 3.])

# 目標出力
d = np.array([0, 0, 0, 1, 0])

# ネットワークの初期化
network = init_network()

# 出力
y, z1 = forward(network, x)

# 誤差
loss = functions.cross_entropy_error(d, y)

## 表示
print("\n##### 結果表示 #####")
print_vec("出力", y)
print_vec("訓練データ", d)
print_vec("誤差", loss)

```

```

##### ネットワークの初期化 #####
*** 重み1 ***
[[0.36143129 0.77593138 0.39047667 0.90802462 0.80419213]
 [0.09368066 0.03949932 0.21307025 0.5430076 0.45659999]
 [0.04964103 0.17190452 0.57804167 0.53448881 0.62714306]]

shape:(3, 5)
*** 重み2 ***
[[0.0798375 0.14092462 0.60349912 0.29840567]
 [0.10950786 0.03692371 0.8628275 0.60747951]
 [0.2056376 0.30282713 0.08953316 0.69536819]
 [0.57912239 0.20416626 0.42764695 0.70058653]
 [0.47673383 0.7603304 0.03558328 0.27240796]]

shape:(5, 4)
*** バイアス1 ***
[0.41271588 0.8647569 0.13221497 0.09943383 0.31212488]

shape:(5, )
*** バイアス2 ***
[0.35780505 0.34327934 0.21971815 0.43192193]

shape:(4, )
##### 順伝播開始 #####
*** 総入力1 ***
[1.11043157 2.23540048 2.68295714 3.69694006 3.91094619]

*** 中間層出力1 ***
[1.11043157 2.23540048 2.68295714 3.69694006 3.91094619]

*** 総入力2 ***
[5.24843101 5.12317969 4.77899069 7.64228332]

*** 出力1 ***
[0.0742763 0.06553213 0.04644885 0.81374272]

出力合計: 1.0

##### 結果表示 #####
*** 出力 ***
[0.0742763 0.06553213 0.04644885 0.81374272]

*** 訓練データ ***
[0 0 0 1 0]

*** 誤差 ***
13.125061119148114

```

▼ 回帰（2-3-2ネットワーク）

```

# 回帰
# 2-3-2ネットワーク

# !試してみよう_ノードの構成を 3-5-4 に変更してみよう

# ウエイトとバイアスを設定
# ネットワークを作成
def init_network():
    print("##### ネットワークの初期化 #####")

    network = {}
    """
    network['W1'] = np.array([
        [0.1, 0.3, 0.5],
        [0.2, 0.4, 0.6]
    ])
    network['W2'] = np.array([
        [0.1, 0.4],
        [0.2, 0.5],
        [0.3, 0.6]
    ])
    network['b1'] = np.array([0.1, 0.2, 0.3])
    network['b2'] = np.array([0.1, 0.2])
    """

    network['W1'] = np.random.rand(3, 5)
    network['W2'] = np.random.rand(5, 4)
    network['b1'] = np.random.rand(5)
    network['b2'] = np.random.rand(4)

    print_vec("重み1", network['W1'])
    print_vec("重み2", network['W2'])
    print_vec("バイアス1", network['b1'])
    print_vec("バイアス2", network['b2'])

    return network

```

```

# プロセスを作成
def forward(network, x):
    print("##### 順伝播開始 #####")

    W1, W2 = network['W1'], network['W2']
    b1, b2 = network['b1'], network['b2']
    # 隠れ層の総入力
    u1 = np.dot(x, W1) + b1
    # 隠れ層の総出力
    z1 = functions.relu(u1)
    # 出力層の総入力
    u2 = np.dot(z1, W2) + b2
    # 出力層の総出力
    y = u2

    print_vec("総入力1", u1)
    print_vec("中間層出力1", z1)
    print_vec("総入力2", u2)
    print_vec("出力1", y)
    print("出力合計: " + str(np.sum(z1)))

    return y, z1

# 入力値
x = np.array([1., 2., 3.])
network = init_network()
y, z1 = forward(network, x)
# 目標出力
d = np.array([2., 4., 3., 1.])
# 誤差
loss = functions.mean_squared_error(d, y)
## 表示
print("##### 結果表示 #####")
print_vec("中間層出力", z1)
print_vec("出力", y)
print_vec("訓練データ", d)
print_vec("誤差", loss)

#####
# ネットワークの初期化 #####
*** 重み1 ***
[[0.34851941 0.77077958 0.98086006 0.75356872 0.96663334]
 [0.56317947 0.02008674 0.58887333 0.90985946 0.7323743]
 [0.30620331 0.16893017 0.83095694 0.56796753 0.31098106]]

*** 重み2 ***
[[0.81656004 0.87480426 0.38234727 0.21264673]
 [0.46066921 0.87269718 0.9255395 0.39204393]
 [0.30910051 0.09845799 0.98931222 0.83204597]
 [0.32588475 0.50888944 0.76115847 0.46394821]
 [0.97762738 0.56687081 0.66295647 0.8775069 ]]

*** バイアス1 ***
[0.83270147 0.64477252 0.99203423 0.69450008 0.36994891]

*** バイアス2 ***
[0.05600346 0.1016697 0.04415036 0.82432213]

#####
# 順伝播開始 #####
*** 総入力1 ***
[3.22618974 1.9625161 5.64351178 4.97169031 3.73420028]

*** 中間層出力1 ***
[3.22618974 1.9625161 5.64351178 4.97169031 3.73420028]

*** 総入力2 ***
[10.60971865 9.83913513 14.93711294 12.5588079 ]

*** 出力1 ***
[10.60971865 9.83913513 14.93711294 12.5588079 ]

出力合計: 19.538108208775313

#####
# 結果表示 #####
*** 中間層出力 ***
[3.22618974 1.9625161 5.64351178 4.97169031 3.73420028]

*** 出力 ***
[10.60971865 9.83913513 14.93711294 12.5588079 ]

*** 訓練データ ***
[2. 4. 3. 1. ]

*** 誤差 ***
48.0404324571966

```

▼ 2値分類 (2-3-1ネットワーク)

```
# 2値分類
# 2-3-1ネットワーク

# !試してみよう_ノードの構成を 5-10-1 に変更してみよう

# ウエイトとバイアスを設定
# ネットワークを作成
def init_network():
    print("##### ネットワークの初期化 #####")

    network = {}
    ...

    network['W1'] = np.array([
        [0.1, 0.3, 0.5],
        [0.2, 0.4, 0.6]
    ])
    network['W2'] = np.array([
        [0.2],
        [0.4],
        [0.6]
    ])
    network['b1'] = np.array([0.1, 0.2, 0.3])
    network['b2'] = np.array([0.1])
    ...

    network['W1'] = np.random.rand(3, 10)
    network['W2'] = np.random.rand(10, 1)
    network['b1'] = np.random.rand(10)
    network['b2'] = np.random.rand(1)

    return network

# プロセスを作成
def forward(network, x):
    print("##### 順伝播開始 #####")

    W1, W2 = network['W1'], network['W2']
    b1, b2 = network['b1'], network['b2']

    # 隠れ層の総入力
    u1 = np.dot(x, W1) + b1
    # 隠れ層の総出力
    z1 = functions.relu(u1)
    # 出力層の総入力
    u2 = np.dot(z1, W2) + b2
    # 出力層の総出力
    y = functions.sigmoid(u2)

    print_vec("総入力1", u1)
    print_vec("中間層出力1", z1)
    print_vec("総入力2", u2)
    print_vec("出力1", y)
    print("出力合計: " + str(np.sum(z1)))

    return y, z1

# 入力値
x = np.array([1., 2., 3., 4., 5.])
# 目標出力
d = np.array([3])
network = init_network()
y, z1 = forward(network, x)
# 誤差
loss = functions.cross_entropy_error(d, y)

## 表示
print("\n##### 結果表示 #####")
print_vec("中間層出力", z1)
print_vec("出力", y)
print_vec("訓練データ", d)
print_vec("誤差", loss)

##### ネットワークの初期化 #####
##### 順伝播開始 #####
*** 総入力1 ***
[ 9.63719752  9.50665503  7.53595959 10.43860365  6.99452014 10.22452815
 5.9234492   6.86076354  9.97045544  6.35682631]

*** 中間層出力1 ***
```

```

[ 9.63719752 9.50665503 7.53595959 10.43860365 6.99452014 10.22452815
 5.9234492 6.86076354 9.97045544 6.35682631]

*** 総入力2 ***
[43.14608439]

*** 出力1 ***
[1.]

出力合計: 83.44895857026246

##### 結果表示 #####
*** 中間層出力 ***
[ 9.63719752 9.50665503 7.53595959 10.43860365 6.99452014 10.22452815
 5.9234492 6.86076354 9.97045544 6.35682631]

*** 出力 ***
[1.]

*** 訓練データ ***
[3]

*** 誤差 ***
-9.99999505838704e-08

```

▼ Section3: 出力層

- 出力層の役割
 - 中間層の出力 ⇒ 他の中間層の入力になる
 - 出力層 ⇒ 私たちが知りたい内容を出力する層
- ニューラルネットワークの学習
 - 実際に学習させるためには
 - 訓練データ（正解値）を用意する - 入力値に対して、ニューラルネットワークに変換をかけて出力層へ
 - 入力値に対する出力を求めて、人間がつけたラベルと比べる
 - ニューラルネットワークの値と、訓練データの答えを比較する
 - どのくらいあっていたのか（誤差関数）を調整する。
- 誤差関数
 - どのくらいあっていたかの割合
 - 講座ではまず2乗誤差を使用する。
 - *2乗誤差

$$E_n(W) = \frac{1}{2} \sum_{j=1}^J (y_j - d_j)^2 = \frac{1}{2} \|(y - d)\|^2$$

確認テスト 8.

なぜ引数算でなく計算する。
引数算だと誤差と誤差一が出てきてよく打ち消し合ってある。

このあたり..

$$E_n(W) = \frac{1}{2} \sum_{j=1}^J (y_j - d_j)^2 = \frac{1}{2} \|(y - d)\|^2$$

の $\frac{1}{2}$ はどういう意味をもつ。

⇒ 2次関数を微分すると、2乗ではなく2乗の

2を消す役割がある。

- 数式とコード

- この講座では平均 2 乗誤差を使うが、主には次を使用する。

```
loss = cross_entropy_error(d,y)
```

- 分類問題：クロスエントロピー誤差

```
loss = functions.mean_squared_error(d,y)
```

- 出力層の活性化関数

- 変換された値を私たちが扱いやすい値に変換する

- 分類問題の場合、出力層の出力は0~1の範囲に限定して、総和を 1 にする

- 回帰問題：恒等写像（2 乗誤差）

- 二値分類：シグモイド関数（交差エントロピー）

- 他クラス分類：ソフトマックス関数（交差エントロピー）

- 数式とコード

- シグモイド関数

$$f(u) = \frac{1}{1 + e^{(-u)}}$$

```
def sigmoid(x):
    return 1/(1 + np.exp(-x))
```

- ソフトマックス関数

$$\textcircled{1} \dots f(i, u) = \frac{e^{u_j} \dots \textcircled{2}}{\sum_{k=1}^K e^{u_k} \dots \textcircled{3}}$$

- 確認テスト 9

- 出力層の活性化関数

- ソフトマックス関数

```
def softmax(x):
```

... $\textcircled{1}$

```
if x.ndim == 2: # 引数の配列の次元が 2 の場合
    x = x.T # 引数を転置する
    x = x - np.max(x, axis=0) # 配列の最大値を x から引く（安定させるため）
    y = np.exp(x) / np.sum(np.exp(x), axis=0)
```

... $\frac{\textcircled{2}}{\textcircled{3}}$

```
return y.T # 転置して結果を返す
x = x - np.max(x) # オーバーフロー対策(安定させるため)
return np.exp(x) / np.sum(np.exp(x))
```

... $\frac{\textcircled{2}}{\textcircled{3}}$

- 2 乗誤差

- 式

$$E_n(W) = \frac{1}{2} \sum_{i=1}^l (y_n - d_n)^2 \dots \text{2 乗誤差}$$

- コード

```
def mean_squared_error(d,y):
    return np.mean(np.square(d-y))/2
```

- 確認問題 10（交差エントロピー）

- 式

$$\textcircled{1} \dots E_n(W) = - \sum_{i=1}^l d_i \log y_i \dots \textcircled{2} \text{ 交差エントロピー}$$

- コード def cross_entropy_error(d, y):

```
if y.ndim == 1: # 次元が 1 の場合の対策
    d = d.reshape(1, d.size)
    y = y.reshape(1, y.size)
    # 教師データが one-hot-vector の場合、正解ラベルのインデックスに変換
    if d.size == y.size:
        d = d.argmax(axis=1) # 引数の最大値
batch_size = y.shape[0]
```

```
return -np.sum(np.log(y[np.arange(batch_size), d] + 1e-7)) / batch_size #式の値を戻す
```

▼ Section4: 勾配降下法

ニューラルネットワークを学習させる方法
E(W) や W をいい感じに調整する

- 勾配降下法 (3つ)
 - 勾配降下法
 - 確率的勾配降下法
 - ミニバッチ勾配降下法
- 勾配降下法
 - 式

$$W^{(t+1)} = W^{(t)} - \varepsilon \nabla E$$
$$\nabla E = \frac{\partial E}{\partial W} = \begin{bmatrix} \frac{\partial E}{\partial w_1} & \cdots & \frac{\partial E}{\partial w_M} \end{bmatrix}$$

ε : 学習率

- 学習率が大きい: 発散する
- 学習率が小さい: 時間がかかる、極小値で終わってしまう

確認テスト 1 1 上の式のソースの該当箇所

```
# 勾配降下の繰り返し
for dataset in random_datasets:
    x, d = dataset['x'], dataset['d']
    z1, y = forward(network, x)
    grad = backward(x, d, z1, y)
    # パラメータに勾配適用
    for key in ('W1', 'W2', 'b1', 'b2'):
        network[key] -= learning_rate * grad[key]

# 誤差
loss = functions.mean_squared_error(d, y)
losses.append(loss)
```

- 学習の方法(ステップ)
 - 1. ニューラルネットワークに値が入力される
 - 2. 出力から値が出力される
 - 3. 出力値と訓練データを比較して間違いを探す
 - 4. 間違いの度合いに従って重み、バイアスを修正する
 - 5. 調整された重み、バイアスを使って次の週に反映する (エポック)
 - 6. 1 – 5 を繰り返す
- 確率的勾配降下法 (SGD)
 - 式
- 確率的勾配降下法と勾配降下法との違い
 - 確率的勾配降下法
 - ランダムに抽出したサンプルの誤差
 - 勾配降下法
 - 全サンプルの平均誤差
- 確率的勾配降下法のメリット
 - データが情報な場合の計算コスト削減
 - 望まない局所極小解に就職するリスクを減らせる
 - オンライン学習ができる

$$W^{(t+1)} = W^{(t)} - \varepsilon \nabla E \quad (\text{勾配降下法と同じ})$$

- 確認テスト1 2 オンライン学習のメリット

- 最初からデータが全部そろっていなくてもよい
- あとから訓練データを足すことができる

- ミニバッチ勾配降下法

オンライン学習のメリットを勾配降下法でできるようにする方法。

- 式

$$W^{(t+1)} = W^{(t)} - \varepsilon \nabla E_t$$

$$E_t = \frac{1}{N_t} \sum_{n \in D_t} E_n$$

$$N_t = |D_t|$$

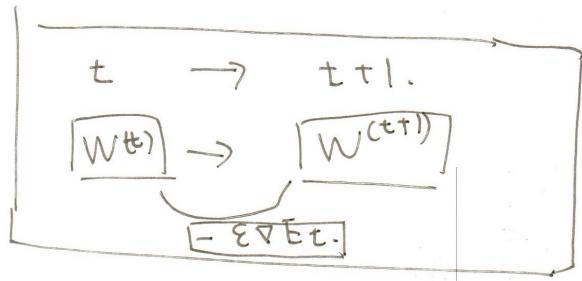
ランダムに分割したデータの集合（ミニバッチという） D_t に属するサンプルの平均誤差

- ミニバッチ勾配降下法のメリット

- 処理を小分け \Rightarrow メモリが少なくて済む
 \Rightarrow 並列で学習ができる(SIMD Single Instruction Multi Data)

- 確認テスト1 3 - 勾配降下法の意味を図示せよ

$$W^{(t+1)} = W^{(t)} - \varepsilon \nabla E_t$$



- 誤差勾配の計算

$$\nabla E = \frac{\partial E}{\partial W} = \left| \frac{\partial E}{\partial w_1} \cdots \frac{\partial E}{\partial w_M} \right|$$

\Rightarrow 誤差逆伝播法を利用する

- 演習問題

- ▼ 準備

- ▼ Google ドライブのマウント

```
from google.colab import drive
drive.mount('/content/drive')

Mounted at /content/drive
```

- ▼ sys.path の設定

以下では、Google ドライブのマイドライブ直下にDNN_code フォルダを置くことを仮定しています。必要に応じて、パスを変更してください。

```
import sys
sys.path.append('/content/drive/My Drive/')
```

- ▼ import と関数定義

```
import numpy as np
from common import functions
import matplotlib.pyplot as plt

def print_vec(text, vec):
    print("*** " + text + " ***")
    print(vec)
    #print("shape: " + str(x.shape))
    print("")
```

- ▼ メインプログラム

```
# ウェイトとバイアスを設定
# ネットワークを作成
def init_network():
    print("##### ネットワークの初期化 #####")

    network = {}
    network['W1'] = np.array([
        [0.1, 0.3, 0.5],
        [0.2, 0.4, 0.6]
    ])

    network['W2'] = np.array([
        [0.1, 0.4],
        [0.2, 0.5],
        [0.3, 0.6]
    ])

    network['b1'] = np.array([0.1, 0.2, 0.3])
    network['b2'] = np.array([0.1, 0.2])

    print_vec("重み1", network['W1'])
    print_vec("重み2", network['W2'])
    print_vec("バイアス1", network['b1'])
    print_vec("バイアス2", network['b2'])

    return network

# 順伝播
def forward(network, x):
    print("##### 順伝播開始 #####")

    W1, W2 = network['W1'], network['W2']
    b1, b2 = network['b1'], network['b2']

    u1 = np.dot(x, W1) + b1
    z1 = functions.relu(u1)
    u2 = np.dot(z1, W2) + b2
```

```

y = functions.softmax(u2)

print_vec("総入力1", u1)
print_vec("中間層出力1", z1)
print_vec("総入力2", u2)
print_vec("出力1", y)
print("出力合計: " + str(np.sum(y)))

return y, z1

# 誤差逆伝播
def backward(x, d, z1, y):
    print("\n##### 誤差逆伝播開始 #####")

    grad = []

    W1, W2 = network['W1'], network['W2']
    b1, b2 = network['b1'], network['b2']
    # 出力層でのデルタ
    delta2 = functions.d_sigmoid_with_loss(d, y)
    # b2の勾配
    grad['b2'] = np.sum(delta2, axis=0)
    # W2の勾配
    grad['W2'] = np.dot(z1.T, delta2)
    # 中間層でのデルタ
    delta1 = np.dot(delta2, W2.T) * functions.d_relu(z1)
    # b1の勾配
    grad['b1'] = np.sum(delta1, axis=0)
    # W1の勾配
    grad['W1'] = np.dot(x.T, delta1)

    print_vec("偏微分_dE/du2", delta2)
    print_vec("偏微分_dE/du1", delta1)

    print_vec("偏微分_重み1", grad['W1'])
    print_vec("偏微分_重み2", grad['W2'])
    print_vec("偏微分_バイアス1", grad['b1'])
    print_vec("偏微分_バイアス2", grad['b2'])

    return grad

# 訓練データ
x = np.array([[1.0, 5.0]])
# 目標出力
d = np.array([[0, 1]])
# 学習率
learning_rate = 0.01
network = init_network()
y, z1 = forward(network, x)

# 誤差
loss = functions.cross_entropy_error(d, y)

grad = backward(x, d, z1, y)
for key in ('W1', 'W2', 'b1', 'b2'):
    network[key] -= learning_rate * grad[key]

print("##### 結果表示 #####")

print("##### 更新後パラメータ #####")
print_vec("重み1", network['W1'])
print_vec("重み2", network['W2'])
print_vec("バイアス1", network['b1'])
print_vec("バイアス2", network['b2'])

[0.1 0.2 0.3]

*** バイアス2 ***
[0.1 0.2]

##### 順伝播開始 #####
*** 総入力1 ***
[[1.2 2.5 3.8]]

*** 中間層出力1 ***
[[1.2 2.5 3.8]]

*** 総入力2 ***
[[1.86 4.21]]

*** 出力1 ***
[[0.08706577 0.91293423]]

出力合計: 1.0
..... 80% 100% 100% 100% .....

```

```

##### 誤差逆伝播開始 #####
*** 偏微分_dE/du2 ***
[[ 0.08706577 -0.08706577]]

*** 偏微分_dE/du2 ***
[[-0.02611973 -0.02611973 -0.02611973]]

*** 偏微分_重み1 ***
[[-0.02611973 -0.02611973 -0.02611973]
 [-0.13059866 -0.13059866 -0.13059866]]

*** 偏微分_重み2 ***
[[ 0.10447893 -0.10447893]
 [ 0.21766443 -0.21766443]
 [ 0.33084994 -0.33084994]]

*** 偏微分_バイアス1 ***
[-0.02611973 -0.02611973 -0.02611973]

*** 偏微分_バイアス2 ***
[ 0.08706577 -0.08706577]

##### 結果表示 #####
##### 更新後パラメータ #####
*** 重み1 ***
[[0.1002612 0.3002612 0.5002612]
 [0.20130599 0.40130599 0.60130599]]

*** 重み2 ***
[[0.09895521 0.40104479]
 [0.19782336 0.50217664]
 [0.2966915 0.6033085 ]]

*** バイアス1 ***
[0.1002612 0.2002612 0.3002612]

*** バイアス2 ***
[0.09912934 0.20087066]

```

▼ Section5:誤差逆伝播法

数値微分ではない方法で求める方法

$$\begin{aligned}
 W^{(2)} &\cdots \frac{\partial E}{\partial y} \frac{\partial y}{\partial u^{(2)}} \frac{\partial u}{\partial u^{(2)}} \\
 b^{(2)} &\cdots \frac{\partial E}{\partial y} \frac{\partial y}{\partial u^{(2)}} \frac{\partial u}{\partial b^{(2)}} \\
 u^{(2)} &\cdots \frac{\partial E}{\partial y} \frac{\partial y}{\partial u^{(2)}} \\
 y &\cdots \frac{\partial E}{\partial y}
 \end{aligned}$$

- ソースコード（確認テスト1.4）


```

# 出力層でのデルタ
delta2 = functions.d_mean_squared_error(d, y)
# b2の勾配
grad['b2'] = np.sum(delta2, axis=0)
# W2の勾配
grad['W2'] = np.dot(z1.T, delta2)
# 中間層でのデルタ
#delta1 = np.dot(delta2, W2.T) * functions.d_relu(z1)
## 試してみよう
delta1 = np.dot(delta2, W2.T) * functions.d_sigmoid(z1)
* 太字の箇所が再利用している個所
      
```

$$\begin{aligned}
 E(y) &= \frac{1}{2} \sum_{j=1}^J (y_j - d_j)^2 = \frac{1}{2} \|y - d\|^2 : \text{誤差関数=二乗誤差関数} \\
 y &= u^{(L)} : \text{出力層の活性化関数=恒等写像} \\
 u^{(l)} &= w^{(l)} z^{(l-1)} + b^{(l)} : \text{総入力の計算}
 \end{aligned}$$

$$\begin{aligned}
 \frac{\partial E}{\partial w_{ji}^{(2)}} &= \frac{\partial E}{\partial y} \frac{\partial y}{\partial u} \frac{\partial u}{\partial w_{ji}^{(2)}} \\
 \frac{\partial E(y)}{\partial y} &= \frac{\partial}{\partial y} \frac{1}{2} \|y - d\|^2 = y - d
 \end{aligned}$$

$$\frac{\partial y(u)}{\partial u} = \frac{\partial u}{\partial u} = 1$$

$$\frac{\partial u(w)}{\partial w_{ji}} = \frac{\partial}{\partial w_{ji}}(w^{(l)}z^{(l-1)} + b^{(l)}) = \frac{\partial}{\partial w_{ji}} \left(\begin{bmatrix} w_{11} + \dots + w_{1i}z_i & + \dots + w_{1l}z_l \\ w_{j1} + \dots + w_{ji}z_i & + \dots + w_{jl}z_l \\ w_{l1} + \dots + w_{li}z_i & + \dots + w_{ll}z_l \end{bmatrix} + \begin{bmatrix} b_1 \\ b_j \\ b_l \end{bmatrix} \right) = \begin{bmatrix} 0 \\ z_i \\ 0 \end{bmatrix}$$

$$\frac{\partial E}{\partial w^{(2)}} = \frac{\partial E}{\partial y} \frac{\partial y}{\partial u} \frac{\partial u}{\partial w_{ji}^{(2)}} = (y - d) \cdot \begin{bmatrix} 0 \\ z_i \\ 0 \end{bmatrix} = (y_j - d_j)z_i$$

- コードの確認

```
# 出力層でのデルタ
delta2 = functions.d_mean_squared_error(d, y)
# b2の勾配
grad['b2'] = np.sum(delta2, axis=0)
# W2の勾配
grad['W2'] = np.dot(z1.T, delta2)
# 中間層でのデルタ
#delta1 = np.dot(delta2, W2.T) * functions.d_relu(z1)

## 試してみよう
delta1 = np.dot(delta2, W2.T) * functions.d_sigmoid(z1)

delta1 = delta1[np.newaxis, :]
# b1の勾配
grad['b1'] = np.sum(delta1, axis=0)
x = x[np.newaxis, :]
# W1の勾配
grad['W1'] = np.dot(x.T, delta1)
```

確認テスト15

$$\frac{\partial E}{\partial y}$$

$$\frac{\partial E}{\partial y} \frac{\partial y}{\partial u}$$

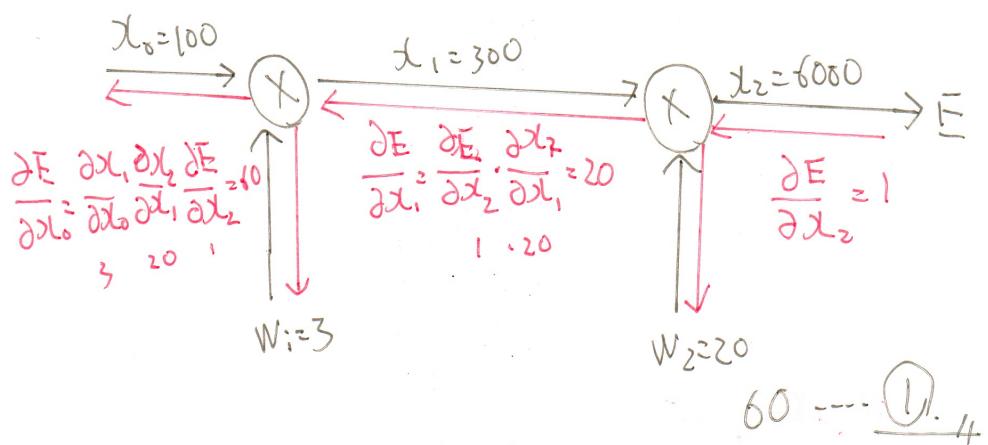
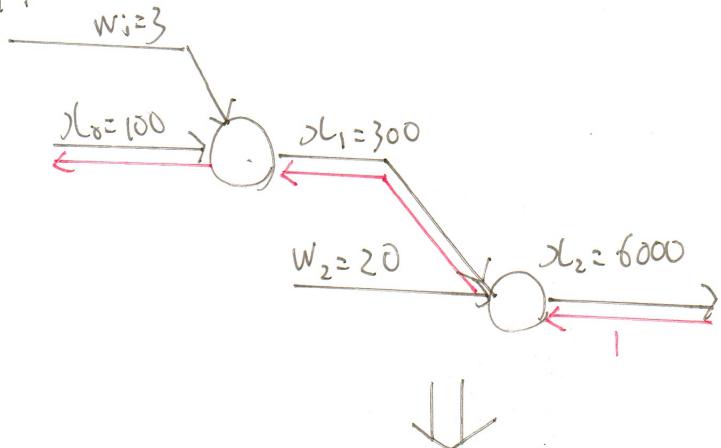
$$\frac{\partial E}{\partial y} \frac{\partial y}{\partial u} \frac{\partial u}{\partial w_{ji}^{(2)}}$$

```
delta2 = functions.d_mean_squared_error(d,y)
delta1 = np.dot(delta2, W2.T) * functions.d_sigmoid(z1)
grad['W1'] = np.dot(x.T, delta1)
```

▼ 演習問題：深層学習Day1

練習問題：深度學習 day 1.

問 1.



問 2. 條件

$$P(x_i | C_0) = 0.1$$

$$P(x_i | C_1) = 0.3$$

$$P(x_i | C_2) = 0.2$$

$$P(C_0) = 0.7$$

$$P(C_1) = 0.2$$

$$P(C_2) = 0.1$$

$$\Rightarrow P(C_1 | x_i) ?$$

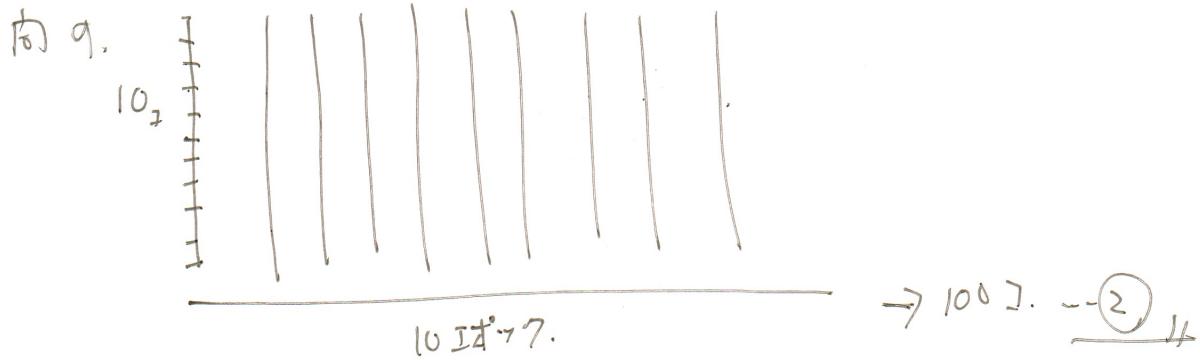
$$P(C_1 | x_i) = \frac{P(x_i | C_1) P(C_1)}{P(x_i)} \dots \text{ノーラズ定理}$$

$$P(x_i) = \sum_{k=1}^n P(x_i | C_k) P(C_k) \text{ たとえ}$$

$$P(C_1 | x_i) = \frac{0.3 \times 0.2}{0.1 \times 0.7 + 0.3 \times 0.2 + 0.2 \times 0.1} = 0.4 \quad \text{--- } \frac{\textcircled{3}}{4}$$

問 3.

$$P(C_K = 1 | x_i = 1) = \frac{12}{16} = 0.75 \quad \text{--- } \frac{\textcircled{4}}{4}$$



問10. K. 分割検証法

データを5分割.

$[S_1 | S_2 | S_3 | S_4 | S_5]$

$[S_1 | S_2 | S_3 | S_4 | S_5]$

$[S_1 | S_2 | S_3 | S_4 | S_5]$

$[S_1 | S_2 | S_3 | S_4 | S_5]$

$[S_1 | S_2 | S_3 | S_4 | S_5]$

\Rightarrow データセットを全く読み込まない。

検証, 評価 が可能。

$\sim \textcircled{3} \frac{1}{4}$

問11.

3が5が11.

$P(A|B)$ は、事象 B が起こるという条件のもとで
事象 A が起こる確率

$\sim \textcircled{3} \frac{1}{4}$

- 実装演習

- 準備

- Google ドライブのマウント

```
from google.colab import drive
drive.mount('/content/drive')

Mounted at /content/drive
```

- sys.pathの設定

以下では、Google ドライブのマイドライブ直下にDNN_codeフォルダを置くことを仮定しています。必要に応じて、パスを変更してください。

```
import sys
sys.path.append('/content/drive/My Drive/')
```

- importと関数定義

```
import numpy as np
from common import functions
import matplotlib.pyplot as plt

def print_vec(text, vec):
    print("*** " + text + " ***")
    print(vec)
    #print("shape: " + str(x.shape))
    print("")
```

- 確率勾配降下法

```
# サンプルとする関数
#yの値を予想するAI

def f(x):
    y = 3 * x[0] + 2 * x[1]
    return y

# 初期設定
def init_network():
    # print("##### ネットワークの初期化 #####")
    network = {}
    nodesNum = 10
    network['W1'] = np.random.randn(2, nodesNum)
    network['W2'] = np.random.randn(nodesNum)
    network['b1'] = np.random.randn(nodesNum)
    network['b2'] = np.random.randn()

    # print_vec("重み1", network['W1'])
    # print_vec("重み2", network['W2'])
    # print_vec("バイアス1", network['b1'])
    # print_vec("バイアス2", network['b2'])

    return network

# 順伝播
def forward(network, x):
    # print("##### 順伝播開始 #####")

    W1, W2 = network['W1'], network['W2']
    b1, b2 = network['b1'], network['b2']
    u1 = np.dot(x, W1) + b1
    z1 = functions.relu(u1)

    ## 試してみよう
    #z1 = functions.sigmoid(u1)

    u2 = np.dot(z1, W2) + b2
    y = u2
```

```

# print_vec("総入力1", u1)
# print_vec("中間層出力1", z1)
# print_vec("総入力2", u2)
# print_vec("出力1", y)
# print("出力合計: " + str(np.sum(y)))

return z1, y

# 誤差逆伝播
def backward(x, d, z1, y):
    # print("Yn##### 誤差逆伝播開始 #####")

    grad = []

    W1, W2 = network['W1'], network['W2']
    b1, b2 = network['b1'], network['b2']

    # 出力層でのデルタ
    delta2 = functions.d_mean_squared_error(d, y)
    # b2の勾配
    grad['b2'] = np.sum(delta2, axis=0)
    # W2の勾配
    grad['W2'] = np.dot(z1.T, delta2)
    # 中間層でのデルタ
    #delta1 = np.dot(delta2, W2.T) * functions.d_relu(z1)

    ## 試してみよう
    delta1 = np.dot(delta2, W2.T) * functions.d_sigmoid(z1)

    delta1 = delta1[np.newaxis, :]
    # b1の勾配
    grad['b1'] = np.sum(delta1, axis=0)
    x = x[np.newaxis, :]
    # W1の勾配
    grad['W1'] = np.dot(x.T, delta1)

    # print_vec("偏微分_重み1", grad["W1"])
    # print_vec("偏微分_重み2", grad["W2"])
    # print_vec("偏微分_バイアス1", grad["b1"])
    # print_vec("偏微分_バイアス2", grad["b2"])

    return grad

# サンプルデータを作成
data_sets_size = 100000
data_sets = [0 for i in range(data_sets_size)]

for i in range(data_sets_size):
    data_sets[i] = {}
    # ランダムな値を設定
    data_sets[i]['x'] = np.random.rand(2)

    ## 試してみよう_入力値の設定
    # data_sets[i]['x'] = np.random.rand(2) * 10 -5 # -5~5のランダム数値

    # 目標出力を設定
    data_sets[i]['d'] = f(data_sets[i]['x'])

losses = []
# 学習率
learning_rate = 0.07

# 抽出数
epoch = 1000

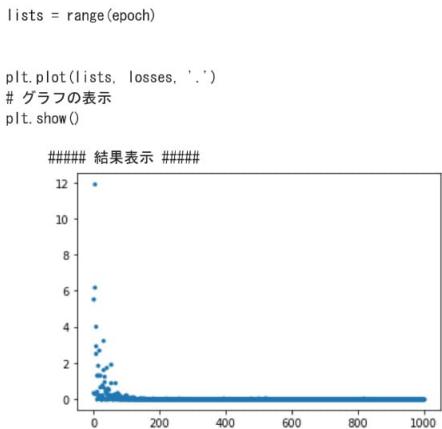
# パラメータの初期化
network = init_network()
# データのランダム抽出
random_datasets = np.random.choice(data_sets, epoch)

# 勾配降下の繰り返し
for dataset in random_datasets:
    x, d = dataset['x'], dataset['d']
    z1, y = forward(network, x)
    grad = backward(x, d, z1, y)
    # パラメータに勾配適用
    for key in ('W1', 'W2', 'b1', 'b2'):
        network[key] -= learning_rate * grad[key]

    # 誤差
    loss = functions.mean_squared_error(d, y)
    losses.append(loss)

print("##### 結果表示 #####")

```



深層学習Day2

確認テスト1：深層学習 Day2.
連鎖律を用ひて、 $\frac{dE}{dx}$ を求める。

$$E = t^2$$

$$t = x + y \quad \text{ここで}.$$

$$\frac{dE}{dx} = \frac{dE}{dt} \frac{dt}{dx}$$

$$= 2t \cdot 1$$

$$= \underline{\underline{2(x+y)}}$$

▼ Section1: 勾配消失問題

学習がうまくいかなる原因の1つで、逆伝播されるにつれて情報量が少なくなり、更新がうまくいかなくなる。起きる原因是、逆伝播するにしたがって、微分の連鎖率の項が多くなる

- ⇒ (微分値が0 - 1をとるのが多くなる) (例: シグモイド関数)
- ⇒ 値がどんどん小さくなる

- 1-1-1 活性化関数: シグモイド関数

- 式

$$f(u) = \frac{1}{1 + e^{-u}}$$

- コード


```
def sigmoid d(x):
    return (1 - sigmoid(x)) * sigmoid(x)
```

シグモイド関数を微分すると、最大値は0.25になる
⇒ 何回も微分を適用すると、0に限りなく近づく。
*これが問題である。

- 確認テスト2

シグモイド関数を微分したとき、入力値が0の時に最大値をとるがいくつか。
答え) (2) 0.25

- 勾配消失の解決方法
 - 活性化関数をどうにかする
 - 重みの初期値をどうにかする
 - バッチ正規化

- 1-1-2 活性化関数：ReLU関数

- 式

$$f(x) = \begin{cases} x & (x > 0) \\ 0 & (x \leq 0) \end{cases}$$

- コード

```
def relu(x):
    return np.maximum(0,x)
*今もっとも使われている関数
*勾配消失問題が回避
*スパース化される（モデルの効果的な動き）
```

- 1-2 初期値の設定方法

- 重みの初期値設定：Xavier

▪ 重みの要素を、前の層のノード数の平方根で割った値 * シグモイド関数のようなS字カーブの関数でうまくいく。

- 重みの初期値設定：He

▪ 重みの要素を、前の層のノード数の平方根で割った値に $\sqrt{2}$ を掛けた値
* S字カーブではない関数でうまくいく。

- 確認テスト3

▪ 重みの初期値に0を設定すると、どのような問題が発生するか。
▪ 学習の効果がない（多数の重みをもつ意味がなくなる）

- 1-3 バッチ正規化

ミニバッチ単位で、入力値のデータの偏りを抑制する手法

- 学習用データを小分けにする
⇒ 小分けにしたものミニバッチという。
- コンピュータではこのミニバッチ単位で処理をする。
- このバッチ内で正規化を行う
- 計算式

$$\begin{aligned} 1. \text{ミニバッチの平均 } \mu_t &= \frac{1}{N_t} \sum_{i=1}^{N_t} s_n i \\ 2. \text{ミニバッチの分散 } \sigma_t^2 &= \frac{1}{N_t} \sum_{i=1}^{N_t} (x_{ni} - \mu_t)^2 \\ 3. \text{ミニバッチの正規化 } \hat{x}_{ni} &= \frac{x_{ni} - \mu_t}{\sqrt{\sigma_t^2 + \Theta}} \end{aligned}$$

A 値域・分布 $\mu \pm \sigma$

▼ 準備

▼ Google ドライブのマウント

```
from google.colab import drive
drive.mount('/content/drive')
```

Mounted at /content/drive

▼ sys.path の設定

以下では、Google ドライブのマイドライブ直下にDNN_code フォルダを置くことを仮定しています。必要に応じて、パスを変更してください。

```
import sys
sys.path.append('/content/drive/My Drive/')
```

▼ vanishing gradient modified

▼ multi layer network class

```
import numpy as np
from common import layers
from collections import OrderedDict
from common import functions
from data.mnist import load_mnist
import matplotlib.pyplot as plt

class MultiLayerNet:
    ...
    input_size: 入力層のノード数
    hidden_size_list: 隠れ層のノード数のリスト
    output_size: 出力層のノード数
    activation: 活性化関数
    weight_init_std: 重みの初期化方法
    ...
    def __init__(self, input_size, hidden_size_list, output_size, activation='relu', weight_init_std='relu'):
        self.input_size = input_size
        self.output_size = output_size
        self.hidden_size_list = hidden_size_list
        self.hidden_layer_num = len(hidden_size_list)
        self.params = []

    # 重みの初期化
    self.__init_weight(weight_init_std)

    # レイヤの生成、sigmoidとreluのみ扱う
    activation_layer = {'sigmoid': layers.Sigmoid, 'relu': layers.Relu}
    self.layers = OrderedDict() # 追加した順番に格納
    for idx in range(1, self.hidden_layer_num+1):
        self.layers['Affine' + str(idx)] = layers.Affine(self.params['W' + str(idx)], self.params['b' + str(idx)])
        self.layers['Activation_function' + str(idx)] = activation_layer[activation]()

    idx = self.hidden_layer_num + 1
    self.layers['Affine' + str(idx)] = layers.Affine(self.params['W' + str(idx)], self.params['b' + str(idx)])

    self.last_layer = layers.SoftmaxWithLoss()

    def __init_weight(self, weight_init_std):
        all_size_list = [self.input_size] + self.hidden_size_list + [self.output_size]
        for idx in range(1, len(all_size_list)):
            scale = weight_init_std
            if str(weight_init_std).lower() in ('relu', 'he'):
                scale = np.sqrt(2.0 / all_size_list[idx - 1])
            elif str(weight_init_std).lower() in ('sigmoid', 'xavier'):
                scale = np.sqrt(1.0 / all_size_list[idx - 1])

            self.params['W' + str(idx)] = scale * np.random.randn(all_size_list[idx-1], all_size_list[idx])
            self.params['b' + str(idx)] = np.zeros(all_size_list[idx])
```

```

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

    return x

def loss(self, x, d):
    y = self.predict(x)

    weight_decay = 0
    for idx in range(1, self.hidden_layer_num + 2):
        W = self.params['W' + str(idx)]

    return self.last_layer.forward(y, d) + weight_decay

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

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

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

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

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

    # 設定
    grad = {}
    for idx in range(1, self.hidden_layer_num+2):
        grad['W' + str(idx)] = self.layers['Affine' + str(idx)].dW
        grad['b' + str(idx)] = self.layers['Affine' + str(idx)].db

    return grad

```

▼ vanishing sample

sigmoid - gauss

```

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

print("データ読み込み完了")

network = MultiLayerNet(input_size=784, hidden_size_list=[40, 20], output_size=10, activation='sigmoid', weight_init_std=0.01)

iters_num = 2000
train_size = x_train.shape[0]
batch_size = 100
learning_rate = 0.1

train_loss_list = []
accuracies_train = []
accuracies_test = []

plot_interval=10

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

    # 勾配
    grad = network.gradient(x_batch, d_batch)

    for key in ('W1', 'W2', 'W3', 'b1', 'b2', 'b3'):
        network.params[key] -= learning_rate * grad[key]

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

```

```

if (i + 1) % plot_interval == 0:
    accr_test = network.accuracy(x_test, d_test)
    accuracies_test.append(accr_test)
    accr_train = network.accuracy(x_batch, d_batch)
    accuracies_train.append(accr_train)

# 結果が長くなるので一部省略
if i < 100 or 1900 < i:
    print('Generation: ' + str(i+1) + '. 正答率(トレーニング) = ' + str(accr_train))
    print('           : ' + str(i+1) + '. 正答率(テスト) = ' + str(accr_test))

lists = range(0, iters_num, plot_interval)
plt.plot(lists, accuracies_train, label="training set")
plt.plot(lists, accuracies_test, label="test set")
plt.legend(loc="lower right")
plt.title("accuracy")
plt.xlabel("count")
plt.ylabel("accuracy")
plt.ylim(0, 1.0)
# グラフの表示
plt.show()

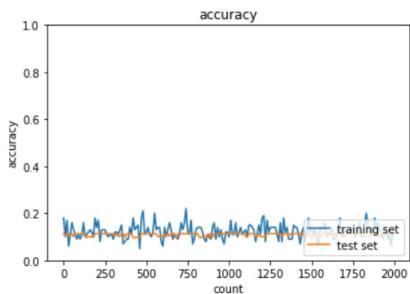
```

データ読み込み完了

```

Generation: 10. 正答率(トレーニング) = 0.18
           : 10. 正答率(テスト) = 0.1135
Generation: 20. 正答率(トレーニング) = 0.1
           : 20. 正答率(テスト) = 0.1028
Generation: 30. 正答率(トレーニング) = 0.17
           : 30. 正答率(テスト) = 0.1135
Generation: 40. 正答率(トレーニング) = 0.06
           : 40. 正答率(テスト) = 0.1135
Generation: 50. 正答率(トレーニング) = 0.11
           : 50. 正答率(テスト) = 0.1028
Generation: 60. 正答率(トレーニング) = 0.16
           : 60. 正答率(テスト) = 0.1135
Generation: 70. 正答率(トレーニング) = 0.13
           : 70. 正答率(テスト) = 0.101
Generation: 80. 正答率(トレーニング) = 0.11
           : 80. 正答率(テスト) = 0.1135
Generation: 90. 正答率(トレーニング) = 0.09
           : 90. 正答率(テスト) = 0.1135
Generation: 100. 正答率(トレーニング) = 0.11
           : 100. 正答率(テスト) = 0.1135
Generation: 1910. 正答率(トレーニング) = 0.16
           : 1910. 正答率(テスト) = 0.1135
Generation: 1920. 正答率(トレーニング) = 0.13
           : 1920. 正答率(テスト) = 0.098
Generation: 1930. 正答率(トレーニング) = 0.12
           : 1930. 正答率(テスト) = 0.1135
Generation: 1940. 正答率(トレーニング) = 0.09
           : 1940. 正答率(テスト) = 0.1135
Generation: 1950. 正答率(トレーニング) = 0.14
           : 1950. 正答率(テスト) = 0.1135
Generation: 1960. 正答率(トレーニング) = 0.14
           : 1960. 正答率(テスト) = 0.1135
Generation: 1970. 正答率(トレーニング) = 0.09
           : 1970. 正答率(テスト) = 0.1135
Generation: 1980. 正答率(トレーニング) = 0.11
           : 1980. 正答率(テスト) = 0.1135
Generation: 1990. 正答率(トレーニング) = 0.06
           : 1990. 正答率(テスト) = 0.1135
Generation: 2000. 正答率(トレーニング) = 0.13
           : 2000. 正答率(テスト) = 0.1135

```



▼ ReLU - gauss

```

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

```

```

print("データ読み込み完了")

network = MultiLayerNet(input_size=784, hidden_size_list=[40, 20], output_size=10, activation='relu', weight_init_std=0.01)

iters_num = 2000
train_size = x_train.shape[0]
batch_size = 100
learning_rate = 0.1

train_loss_list = []
accuracies_train = []
accuracies_test = []

plot_interval=10

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

    # 勾配
    grad = network.gradient(x_batch, d_batch)

    for key in ('W1', 'W2', 'W3', 'b1', 'b2', 'b3'):
        network.params[key] -= learning_rate * grad[key]

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

    if (i + 1) % plot_interval == 0:
        accr_test = network.accuracy(x_test, d_test)
        accuracies_test.append(accr_test)
        accr_train = network.accuracy(x_batch, d_batch)
        accuracies_train.append(accr_train)

    # 結果が長くなるので一部省略
    if i < 100 or 1900 < i:
        print('Generation: ' + str(i+1) + '. 正答率(トレーニング) = ' + str(accr_train))
        print('           : ' + str(i+1) + '. 正答率(テスト) = ' + str(accr_test))

lists = range(0, iters_num, plot_interval)
plt.plot(lists, accuracies_train, label="training set")
plt.plot(lists, accuracies_test, label="test set")
plt.legend(loc="lower right")
plt.title("accuracy")
plt.xlabel("count")
plt.ylabel("accuracy")
plt.ylim(0, 1.0)
# グラフの表示
plt.show()

```

```

データ読み込み完了
Generation: 10. 正答率(トレーニング) = 0.12
: 10. 正答率(テスト) = 0.098
Generation: 20. 正答率(トレーニング) = 0.17
: 20. 正答率(テスト) = 0.1135
Generation: 30. 正答率(トレーニング) = 0.13
: 30. 正答率(テスト) = 0.1135
Generation: 40. 正答率(トレーニング) = 0.16
: 40. 正答率(テスト) = 0.101
Generation: 50. 正答率(トレーニング) = 0.12
: 50. 正答率(テスト) = 0.101
Generation: 60. 正答率(トレーニング) = 0.08
: 60. 正答率(テスト) = 0.101
Generation: 70. 正答率(トレーニング) = 0.08
: 70. 正答率(テスト) = 0.101
Generation: 80. 正答率(トレーニング) = 0.11
: 80. 正答率(テスト) = 0.1028
Generation: 90. 正答率(トレーニング) = 0.11
: 90. 正答率(テスト) = 0.101
Generation: 100. 正答率(トレーニング) = 0.12
: 100. 正答率(テスト) = 0.101
Generation: 1910. 正答率(トレーニング) = 0.9
: 1910. 正答率(テスト) = 0.9134
: 1910. 正答率(テスト) = 0.9134

```

▼ sigmoid - Xavier

```

Generation: 1940 正答率(トレーニング) = 0.92
# データの読み込み
(x_train, d_train), (x_test, d_test) = load_mnist(normalize=True, one_hot_label=True)

print("データ読み込み完了")

network = MultiLayerNet(input_size=784, hidden_size_list=[40, 20], output_size=10, activation='sigmoid', weight_init_std='Xavier')

iters_num = 2000
train_size = x_train.shape[0]
batch_size = 100
learning_rate = 0.1

train_loss_list = []
accuracies_train = []
accuracies_test = []

plot_interval=10

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

    # 勾配
    grad = network.gradient(x_batch, d_batch)

    for key in ('W1', 'W2', 'W3', 'b1', 'b2', 'b3'):
        network.params[key] -= learning_rate * grad[key]

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

    if (i + 1) % plot_interval == 0:
        accr_train = network.accuracy(x_train, d_train)
        accuracies_train.append(accr_train)
        accr_test = network.accuracy(x_test, d_test)
        accuracies_test.append(accr_test)
        accr_train = network.accuracy(x_batch, d_batch)
        accuracies_train.append(accr_train)

    # 結果が長くなるので一部省略
    if i < 100 or 1900 < i:
        print('Generation: ' + str(i+1) + '. 正答率(トレーニング) = ' + str(accr_train))
        print('           : ' + str(i+1) + '. 正答率(テスト) = ' + str(accr_test))

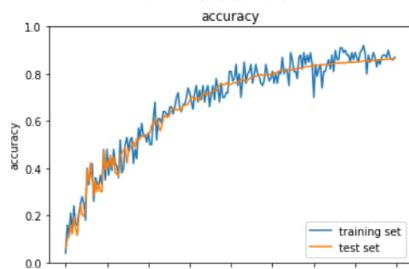
lists = range(0, iters_num, plot_interval)
plt.plot(lists, accuracies_train, label="training set")
plt.plot(lists, accuracies_test, label="test set")
plt.legend(loc="lower right")
plt.title("accuracy")
plt.xlabel("count")
plt.ylabel("accuracy")
plt.ylim(0, 1.0)
# グラフの表示
plt.show()

```

```

データ読み込み完了
Generation: 10. 正答率(トレーニング) = 0.04
: 10. 正答率(テスト) = 0.0614
Generation: 20. 正答率(トレーニング) = 0.16
: 20. 正答率(テスト) = 0.0979
Generation: 30. 正答率(トレーニング) = 0.11
: 30. 正答率(テスト) = 0.1153
Generation: 40. 正答率(トレーニング) = 0.21
: 40. 正答率(テスト) = 0.1589
Generation: 50. 正答率(トレーニング) = 0.13
: 50. 正答率(テスト) = 0.121
Generation: 60. 正答率(トレーニング) = 0.24
: 60. 正答率(テスト) = 0.1965
Generation: 70. 正答率(トレーニング) = 0.16
: 70. 正答率(テスト) = 0.1459
Generation: 80. 正答率(トレーニング) = 0.16
: 80. 正答率(テスト) = 0.1159
Generation: 90. 正答率(トレーニング) = 0.21
: 90. 正答率(テスト) = 0.1926
Generation: 100. 正答率(トレーニング) = 0.25
: 100. 正答率(テスト) = 0.2442
Generation: 1910. 正答率(トレーニング) = 0.84
: 1910. 正答率(テスト) = 0.8604
Generation: 1920. 正答率(トレーニング) = 0.87
: 1920. 正答率(テスト) = 0.8604
Generation: 1930. 正答率(トレーニング) = 0.88
: 1930. 正答率(テスト) = 0.8602
Generation: 1940. 正答率(トレーニング) = 0.88
: 1940. 正答率(テスト) = 0.8618
Generation: 1950. 正答率(トレーニング) = 0.87
: 1950. 正答率(テスト) = 0.8634
Generation: 1960. 正答率(トレーニング) = 0.9
: 1960. 正答率(テスト) = 0.8618
Generation: 1970. 正答率(トレーニング) = 0.87
: 1970. 正答率(テスト) = 0.8629
Generation: 1980. 正答率(トレーニング) = 0.86
: 1980. 正答率(テスト) = 0.8634
Generation: 1990. 正答率(トレーニング) = 0.86
: 1990. 正答率(テスト) = 0.8651
Generation: 2000. 正答率(トレーニング) = 0.87
: 2000. 正答率(テスト) = 0.8653

```



▼ ReLU - He

```

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

print("データ読み込み完了")

network = MultiLayerNet(input_size=784, hidden_size_list=[40, 20], output_size=10, activation='relu', weight_init_std='He')

iters_num = 2000
train_size = x_train.shape[0]
batch_size = 100
learning_rate = 0.1

train_loss_list = []
accuracies_train = []
accuracies_test = []

plot_interval=10

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

    # 勾配
    grad = network.gradient(x_batch, d_batch)

    for key in ('W1', 'W2', 'W3', 'b1', 'b2', 'b3'):
        network.params[key] -= learning_rate * grad[key]

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

    if i % plot_interval == 0:
        accuracy_train = network.accuracy(x_train, d_train)
        accuracy_test = network.accuracy(x_test, d_test)
        accuracies_train.append(accuracy_train)
        accuracies_test.append(accuracy_test)

```

```

train_loss_list.append(loss)

if (i + 1) % plot_interval == 0:
    accr_test = network.accuracy(x_test, d_test)
    accuracies_test.append(accr_test)
    accr_train = network.accuracy(x_batch, d_batch)
    accuracies_train.append(accr_train)

# 結果が長くなるので一部省略
if i < 100 or 1900 < i:
    print('Generation: ' + str(i+1) + '. 正答率(トレーニング) = ' + str(accr_train))
    print('                 : ' + str(i+1) + '. 正答率(テスト) = ' + str(accr_test))

lists = range(0, iters_num, plot_interval)
plt.plot(lists, accuracies_train, label="training set")
plt.plot(lists, accuracies_test, label="test set")
plt.legend(loc="lower right")
plt.title("accuracy")
plt.xlabel("count")
plt.ylabel("accuracy")
plt.ylim(0, 1.0)
# グラフの表示
plt.show()

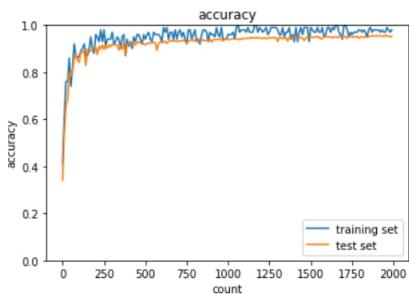
```

データ読み込み完了

```

Generation: 10. 正答率(トレーニング) = 0.41
: 10. 正答率(テスト) = 0.3407
Generation: 20. 正答率(トレーニング) = 0.61
: 20. 正答率(テスト) = 0.5417
Generation: 30. 正答率(トレーニング) = 0.76
: 30. 正答率(テスト) = 0.6605
Generation: 40. 正答率(トレーニング) = 0.76
: 40. 正答率(テスト) = 0.6828
Generation: 50. 正答率(トレーニング) = 0.86
: 50. 正答率(テスト) = 0.8021
Generation: 60. 正答率(トレーニング) = 0.74
: 60. 正答率(テスト) = 0.7828
Generation: 70. 正答率(トレーニング) = 0.82
: 70. 正答率(テスト) = 0.8059
Generation: 80. 正答率(トレーニング) = 0.92
: 80. 正答率(テスト) = 0.8587
Generation: 90. 正答率(トレーニング) = 0.87
: 90. 正答率(テスト) = 0.8747
Generation: 100. 正答率(トレーニング) = 0.86
: 100. 正答率(テスト) = 0.8505
Generation: 1910. 正答率(トレーニング) = 0.98
: 1910. 正答率(テスト) = 0.9533
Generation: 1920. 正答率(トレーニング) = 0.98
: 1920. 正答率(テスト) = 0.9547
Generation: 1930. 正答率(トレーニング) = 0.97
: 1930. 正答率(テスト) = 0.9537
Generation: 1940. 正答率(トレーニング) = 0.98
: 1940. 正答率(テスト) = 0.954
Generation: 1950. 正答率(トレーニング) = 0.97
: 1950. 正答率(テスト) = 0.9523
Generation: 1960. 正答率(トレーニング) = 0.97
: 1960. 正答率(テスト) = 0.9574
Generation: 1970. 正答率(トレーニング) = 0.99
: 1970. 正答率(テスト) = 0.9537
Generation: 1980. 正答率(トレーニング) = 0.98
: 1980. 正答率(テスト) = 0.9532
Generation: 1990. 正答率(トレーニング) = 0.97
: 1990. 正答率(テスト) = 0.9513
Generation: 2000. 正答率(トレーニング) = 0.98
: 2000. 正答率(テスト) = 0.9517

```



▼ [try] hidden_size_listの数字を変更してみよう

```
# データの読み込み
```

```

(x_train, d_train), (x_test, d_test) = load_mnist(normalize=True, one_hot_label=True)

print("データ読み込み完了")

network = MultiLayerNet(input_size=784, hidden_size_list=[40, 20], output_size=10, activation='sigmoid', weight_init_std=0.01)
network = MultiLayerNet(input_size=784, hidden_size_list=[20, 20], output_size=10, activation='sigmoid', weight_init_std=0.01)

iters_num = 2000
train_size = x_train.shape[0]
batch_size = 100
learning_rate = 0.1

train_loss_list = []
accuracies_train = []
accuracies_test = []

plot_interval=10

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

    # 勾配
    grad = network.gradient(x_batch, d_batch)

    for key in ('W1', 'W2', 'W3', 'b1', 'b2', 'b3'):
        network.params[key] -= learning_rate * grad[key]

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

    if (i + 1) % plot_interval == 0:
        accr_test = network.accuracy(x_test, d_test)
        accuracies_test.append(accr_test)
        accr_train = network.accuracy(x_batch, d_batch)
        accuracies_train.append(accr_train)

        # 結果が長くなるので一部省略
        if i < 100 or 1900 < i:
            print('Generation: ' + str(i+1) + '. 正答率(トレーニング) = ' + str(accr_train))
            print('          : ' + str(i+1) + '. 正答率(テスト) = ' + str(accr_test))

lists = range(0, iters_num, plot_interval)
plt.plot(lists, accuracies_train, label="training set")
plt.plot(lists, accuracies_test, label="test set")
plt.legend(loc="lower right")
plt.title("accuracy")
plt.xlabel("count")
plt.ylabel("accuracy")
plt.ylim(0, 1.0)
# グラフの表示
plt.show()

```

```

データ読み込み完了
Generation: 10. 正答率(トレーニング) = 0.17
: 10. 正答率(テスト) = 0.1135
Generation: 20. 正答率(トレーニング) = 0.09
: 20. 正答率(テスト) = 0.1135
Generation: 30. 正答率(トレーニング) = 0.14
: 30. 正答率(テスト) = 0.1135
Generation: 40. 正答率(トレーニング) = 0.07
: 40. 正答率(テスト) = 0.098
Generation: 50. 正答率(トレーニング) = 0.1
: 50. 正答率(テスト) = 0.1028
Generation: 60. 正答率(トレーニング) = 0.18
: 60. 正答率(テスト) = 0.1135
Generation: 70. 正答率(トレーニング) = 0.15
: 70. 正答率(テスト) = 0.1135
Generation: 80. 正答率(トレーニング) = 0.07
: 80. 正答率(テスト) = 0.1009
Generation: 90. 正答率(トレーニング) = 0.12
: 90. 正答率(テスト) = 0.1135
Generation: 100. 正答率(トレーニング) = 0.2
: 100. 正答率(テスト) = 0.101
Generation: 1910. 正答率(トレーニング) = 0.07
: 1910. 正答率(テスト) = 0.1028
Generation: 1920. 正答率(トレーニング) = 0.05
: 1920. 正答率(テスト) = 0.1135
Generation: 1930. 正答率(トレーニング) = 0.11
: 1930. 正答率(テスト) = 0.1135
Generation: 1940. 正答率(トレーニング) = 0.14

```

▼ [try] sigmoid - He と relu - Xavier についても試してみよう

```

: 1960 正答率(テスト) = 0.1135

#sigmoid - He
# データの読み込み
(x_train, d_train), (x_test, d_test) = load_mnist(normalize=True, one_hot_label=True)

print("データ読み込み完了")

network = MultiLayerNet(input_size=784, hidden_size_list=[40, 20], output_size=10, activation='sigmoid', weight_init_std='He')

iters_num = 2000
train_size = x_train.shape[0]
batch_size = 100
learning_rate = 0.1

train_loss_list = []
accuracies_train = []
accuracies_test = []

plot_interval=10

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

    # 勾配
    grad = network.gradient(x_batch, d_batch)

    for key in ('W1', 'W2', 'W3', 'b1', 'b2', 'b3'):
        network.params[key] -= learning_rate * grad[key]

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

    if (i + 1) % plot_interval == 0:
        accr_test = network.accuracy(x_test, d_test)
        accuracies_test.append(accr_test)
        accr_train = network.accuracy(x_batch, d_batch)
        accuracies_train.append(accr_train)

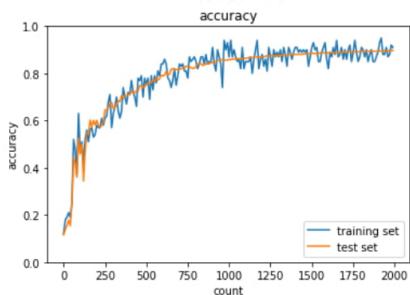
    # 結果が長くなるので一部省略
    if i < 100 or 1900 < i:
        print('Generation: ' + str(i+1) + '. 正答率(トレーニング) = ' + str(accr_train))
        print('          : ' + str(i+1) + '. 正答率(テスト) = ' + str(accr_test))

lists = range(0, iters_num, plot_interval)
plt.plot(lists, accuracies_train, label="training set")
plt.plot(lists, accuracies_test, label="test set")
plt.legend(loc="lower right")
plt.title("accuracy")
plt.xlabel("count")
plt.ylabel("accuracy")
plt.ylim(0, 1.0)
# グラフの表示
plt.show()

```

DATA_SETS/

```
データ読み込み完了
Generation: 10. 正答率(トレーニング) = 0.12
: 10. 正答率(テスト) = 0.1159
Generation: 20. 正答率(トレーニング) = 0.18
: 20. 正答率(テスト) = 0.1383
Generation: 30. 正答率(トレーニング) = 0.19
: 30. 正答率(テスト) = 0.1555
Generation: 40. 正答率(トレーニング) = 0.21
: 40. 正答率(テスト) = 0.1778
Generation: 50. 正答率(トレーニング) = 0.19
: 50. 正答率(テスト) = 0.1547
Generation: 60. 正答率(トレーニング) = 0.25
: 60. 正答率(テスト) = 0.2546
Generation: 70. 正答率(トレーニング) = 0.52
: 70. 正答率(テスト) = 0.4444
Generation: 80. 正答率(トレーニング) = 0.48
: 80. 正答率(テスト) = 0.4158
Generation: 90. 正答率(トレーニング) = 0.37
: 90. 正答率(テスト) = 0.3598
Generation: 100. 正答率(トレーニング) = 0.63
: 100. 正答率(テスト) = 0.5255
Generation: 1910. 正答率(トレーニング) = 0.89
: 1910. 正答率(テスト) = 0.8929
Generation: 1920. 正答率(トレーニング) = 0.93
: 1920. 正答率(テスト) = 0.8954
Generation: 1930. 正答率(トレーニング) = 0.95
: 1930. 正答率(テスト) = 0.8941
Generation: 1940. 正答率(トレーニング) = 0.88
: 1940. 正答率(テスト) = 0.8961
Generation: 1950. 正答率(トレーニング) = 0.88
: 1950. 正答率(テスト) = 0.8956
Generation: 1960. 正答率(トレーニング) = 0.91
: 1960. 正答率(テスト) = 0.896
Generation: 1970. 正答率(トレーニング) = 0.87
: 1970. 正答率(テスト) = 0.8951
Generation: 1980. 正答率(トレーニング) = 0.88
: 1980. 正答率(テスト) = 0.8967
Generation: 1990. 正答率(トレーニング) = 0.92
: 1990. 正答率(テスト) = 0.897
Generation: 2000. 正答率(トレーニング) = 0.91
: 2000. 正答率(テスト) = 0.8974
```



```
# relu - Xavier
# データの読み込み
(x_train, d_train), (x_test, d_test) = load_mnist(normalize=True, one_hot_label=True)

print("データ読み込み完了")

network = MultiLayerNet(input_size=784, hidden_size_list=[40, 20], output_size=10, activation='relu', weight_init_std='Xavier')

iters_num = 2000
train_size = x_train.shape[0]
batch_size = 100
learning_rate = 0.1

train_loss_list = []
accuracies_train = []
accuracies_test = []

plot_interval=10

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

    # 勾配
    grad = network.gradient(x_batch, d_batch)

    for key in ('W1', 'W2', 'W3', 'b1', 'b2', 'b3'):
        network.params[key] -= learning_rate * grad[key]
```

```

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

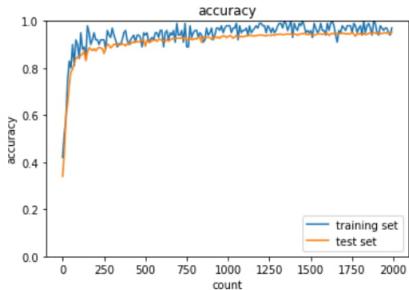
if (i + 1) % plot_interval == 0:
    accr_test = network.accuracy(x_test, d_test)
    accuracies_test.append(accr_test)
    accr_train = network.accuracy(x_batch, d_batch)
    accuracies_train.append(accr_train)

# 結果が長くなるので一部省略
if i < 100 or 1900 < i:
    print('Generation: ' + str(i+1) + '. 正答率(トレーニング) = ' + str(accr_train))
    print('           : ' + str(i+1) + '. 正答率(テスト) = ' + str(accr_test))

lists = range(0, iters_num, plot_interval)
plt.plot(lists, accuracies_train, label="training set")
plt.plot(lists, accuracies_test, label="test set")
plt.legend(loc="lower right")
plt.title("accuracy")
plt.xlabel("count")
plt.ylabel("accuracy")
plt.ylim(0, 1.0)
# グラフの表示
plt.show()

データ読み込み完了
Generation: 10. 正答率(トレーニング) = 0.42
: 10. 正答率(テスト) = 0.3402
Generation: 20. 正答率(トレーニング) = 0.52
: 20. 正答率(テスト) = 0.4367
Generation: 30. 正答率(トレーニング) = 0.58
: 30. 正答率(テスト) = 0.5978
Generation: 40. 正答率(トレーニング) = 0.75
: 40. 正答率(テスト) = 0.6458
Generation: 50. 正答率(トレーニング) = 0.83
: 50. 正答率(テスト) = 0.7276
Generation: 60. 正答率(トレーニング) = 0.8
: 60. 正答率(テスト) = 0.7842
Generation: 70. 正答率(トレーニング) = 0.9
: 70. 正答率(テスト) = 0.7939
Generation: 80. 正答率(トレーニング) = 0.81
: 80. 正答率(テスト) = 0.8216
Generation: 90. 正答率(トレーニング) = 0.92
: 90. 正答率(テスト) = 0.8403
Generation: 100. 正答率(トレーニング) = 0.9
: 100. 正答率(テスト) = 0.8463
Generation: 1910. 正答率(トレーニング) = 0.95
: 1910. 正答率(テスト) = 0.947
Generation: 1920. 正答率(トレーニング) = 0.94
: 1920. 正答率(テスト) = 0.9458
Generation: 1930. 正答率(トレーニング) = 0.98
: 1930. 正答率(テスト) = 0.9473
Generation: 1940. 正答率(トレーニング) = 0.97
: 1940. 正答率(テスト) = 0.9466
Generation: 1950. 正答率(トレーニング) = 0.96
: 1950. 正答率(テスト) = 0.9497
Generation: 1960. 正答率(トレーニング) = 0.97
: 1960. 正答率(テスト) = 0.95
Generation: 1970. 正答率(トレーニング) = 0.97
: 1970. 正答率(テスト) = 0.9482
Generation: 1980. 正答率(トレーニング) = 0.95
: 1980. 正答率(テスト) = 0.95
Generation: 1990. 正答率(トレーニング) = 0.94
: 1990. 正答率(テスト) = 0.9474
Generation: 2000. 正答率(トレーニング) = 0.97
: 2000. 正答率(テスト) = 0.9524

```



▼ Section2:学習率最適化手法

* 学習率は初期は適当な値を選んでいたが、学習率自体も最適化できないか。

- 学習率の決め方

- 初期の学習率を大きく設定して徐々に小さくしていく
- パラメータごとに学習率を可変させる
- 2-1 モメンタム
 - 誤差をパラメータで微分したものと学習率の積を減算後、現在の重みに前回の重みを減算した値と慣性の積を加算する
 - 式とコード

$$V_t = \mu V_{t-1} - \varepsilon \nabla E$$

```
self.v[key] = self.momentum * self.v[key] - self.learning_rate * grad[key]
W^{(t+1)} = W^{(t)} + V_t
params[key] += self.v[key]
```

慣性 : μ

- 2-2 AdaGrad

- 式とコード

$$h_0 = \Theta$$

```
self.h[key] = np.zeros_like(val)
h_t = h_{t-1} + (\nabla E)^2
self.h[key] += grad[key] * grad[key]
W^{(t+1)} = W^{(t)} - \varepsilon \frac{1}{\sqrt{h_t} + \Theta} \nabla E
params[key] -= self.learning_rate * grad[key] / (np.sqrt(self.h[key]) + 1e-7)
```

- メリット
 - 勾配の緩やかな斜面（誤差関数が極端でない）に対して最適解になりやすい
- デメリット
 - 暗転問題がある（どこに行けばいいかわからないとき学習がうまくいかない）

- 2-3 RMSProp

- 式とコード

$$h_t = \alpha h_{t-1} + (1 - \alpha)(\nabla E)^2 \quad * \text{どの位今回の値を使うか}$$

```
self.h[key] *= self.decay_rate
self.h[key] += (1 - self.decay_rate) * grad[key] * grad[key]
W^{(t+1)} = W^{(t)} - \varepsilon \frac{1}{\sqrt{h_t} + \Theta} \nabla E
params[key] -= self.learning_rate * grad[key] / (np.sqrt(self.h[key]) + 1e-7)
```

- メリット
 - 大域的最適解になる。
 - 暗転問題を解決したモデル。

- 2-4 Adam

モメンタムとRMSProp 2つを合体させたもの

- コード

```
m[key] += (1 - beta1) * (grad[key] - m[key])
v[key] += (1 - beta2) * (grad[key] ** 2 - v[key])
network.params[key] -= learning_rate_t * m[key] / (np.sqrt(v[key]) + 1e-7)
```

- 最適化手法 コード演習
-

▼ 準備

▼ Google ドライブのマウント

```
from google.colab import drive
drive.mount('/content/drive')

Mounted at /content/drive
```

▼ sys.pathの設定

以下では、Google ドライブのマイドライブ直下にDNN_code フォルダを置くことを仮定しています。必要に応じて、パスを変更してください。

```
import sys
sys.path.append('/content/drive/My\ Drive')
sys.path.append('/content/drive/My\ Drive/lesson_2')
```

▼ optimizer

▼ SGD

```
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 collections import OrderedDict
from common import layers
from data.mnist import load_mnist
import matplotlib.pyplot as plt
from multi_layer_net import MultiLayerNet

# データの読み込み
(x_train, d_train), (x_test, d_test) = load_mnist(normalize=True, one_hot_label=True)
print("データ読み込み完了")

# batch_normalization の設定 =====
# use_batchnorm = True
use_batchnorm = False
# =====

network = MultiLayerNet(input_size=784, hidden_size_list=[40, 20], output_size=10, activation='sigmoid', weight_init_std=0.01,
                        use_batchnorm=use_batchnorm)

iters_num = 1000
train_size = x_train.shape[0]
batch_size = 100
learning_rate = 0.01

train_loss_list = []
accuracies_train = []
accuracies_test = []

plot_interval=10

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

    # 勾配
    grad = network.gradient(x_batch, d_batch)

    for key in ('W1', 'W2', 'W3', 'b1', 'b2', 'b3'):
        network.params[key] -= learning_rate * grad[key]

    loss = network.loss(x_batch, d_batch)
    train_loss_list.append(loss)
```

```

if (i + 1) % plot_interval == 0:
    accr_test = network.accuracy(x_test, d_test)
    accuracies_test.append(accr_test)
    accr_train = network.accuracy(x_batch, d_batch)
    accuracies_train.append(accr_train)

# 結果が長くなるので一部省略
if i < 100 or 900 < i:
    print('Generation: ' + str(i+1) + '. 正答率(トレーニング) = ' + str(accr_train))
    print('           : ' + str(i+1) + '. 正答率(テスト) = ' + str(accr_test))

lists = range(0, iters_num, plot_interval)
plt.plot(lists, accuracies_train, label="training set")
plt.plot(lists, accuracies_test, label="test set")
plt.legend(loc="lower right")
plt.title("accuracy")
plt.xlabel("count")
plt.ylabel("accuracy")
plt.ylim(0, 1.0)
# グラフの表示
plt.show()

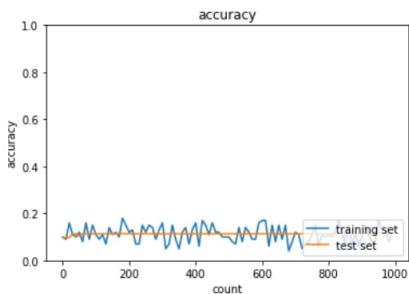
```

データ読み込み完了

```

Generation: 10. 正答率(トレーニング) = 0.1
           : 10. 正答率(テスト) = 0.0974
Generation: 20. 正答率(トレーニング) = 0.09
           : 20. 正答率(テスト) = 0.0974
Generation: 30. 正答率(トレーニング) = 0.16
           : 30. 正答率(テスト) = 0.0974
Generation: 40. 正答率(トレーニング) = 0.11
           : 40. 正答率(テスト) = 0.1135
Generation: 50. 正答率(トレーニング) = 0.1
           : 50. 正答率(テスト) = 0.1135
Generation: 60. 正答率(トレーニング) = 0.12
           : 60. 正答率(テスト) = 0.1028
Generation: 70. 正答率(トレーニング) = 0.08
           : 70. 正答率(テスト) = 0.1135
Generation: 80. 正答率(トレーニング) = 0.16
           : 80. 正答率(テスト) = 0.1135
Generation: 90. 正答率(トレーニング) = 0.09
           : 90. 正答率(テスト) = 0.1135
Generation: 100. 正答率(トレーニング) = 0.15
           : 100. 正答率(テスト) = 0.1135
Generation: 910. 正答率(トレーニング) = 0.1
           : 910. 正答率(テスト) = 0.1135
Generation: 920. 正答率(トレーニング) = 0.15
           : 920. 正答率(テスト) = 0.1135
Generation: 930. 正答率(トレーニング) = 0.11
           : 930. 正答率(テスト) = 0.1135
Generation: 940. 正答率(トレーニング) = 0.08
           : 940. 正答率(テスト) = 0.1135
Generation: 950. 正答率(トレーニング) = 0.08
           : 950. 正答率(テスト) = 0.1135
Generation: 960. 正答率(トレーニング) = 0.17
           : 960. 正答率(テスト) = 0.1135
Generation: 970. 正答率(トレーニング) = 0.13
           : 970. 正答率(テスト) = 0.1135
Generation: 980. 正答率(トレーニング) = 0.13
           : 980. 正答率(テスト) = 0.1135
Generation: 990. 正答率(トレーニング) = 0.08
           : 990. 正答率(テスト) = 0.1135
Generation: 1000. 正答率(トレーニング) = 0.11
           : 1000. 正答率(テスト) = 0.1135

```



▼ Momentum

```
# データの読み込み
```

```

# /usr/local/bin/python
(x_train, d_train), (x_test, d_test) = load_mnist(normalize=True, one_hot_label=True)

print("データ読み込み完了")

# batch_normalizationの設定 =====
# use_batchnorm = True
use_batchnorm = False
# =====

network = MultiLayerNet(input_size=784, hidden_size_list=[40, 20], output_size=10, activation='sigmoid', weight_init_std=0.01,
                        use_batchnorm=use_batchnorm)

iters_num = 1000
train_size = x_train.shape[0]
batch_size = 100
learning_rate = 0.01
# 慣性
momentum = 0.9

train_loss_list = []
accuracies_train = []
accuracies_test = []

plot_interval=10

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

    # 勾配
    grad = network.gradient(x_batch, d_batch)
    if i == 0:
        v = []
    for key in ('W1', 'W2', 'W3', 'b1', 'b2', 'b3'):
        if i == 0:
            v[key] = np.zeros_like(network.params[key])
        v[key] = momentum * v[key] - learning_rate * grad[key]
        network.params[key] += v[key]

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

    if (i + 1) % plot_interval == 0:
        accr_test = network.accuracy(x_test, d_test)
        accuracies_test.append(accr_test)
        accr_train = network.accuracy(x_batch, d_batch)
        accuracies_train.append(accr_train)

        # 結果が長くなるので一部省略
        if i < 100 or 900 < i:
            print('Generation: ' + str(i+1) + '. 正答率(トレーニング) = ' + str(accr_train))
            print('          : ' + str(i+1) + '. 正答率(テスト) = ' + str(accr_test))

lists = range(0, iters_num, plot_interval)
plt.plot(lists, accuracies_train, label="training set")
plt.plot(lists, accuracies_test, label="test set")
plt.legend(loc="lower right")
plt.title("accuracy")
plt.xlabel("count")
plt.ylabel("accuracy")
plt.ylim(0, 1.0)
# グラフの表示
plt.show()

```

```

データ読み込み完了
Generation: 10. 正答率(トレーニング) = 0.05
: 10. 正答率(テスト) = 0.0982
Generation: 20. 正答率(トレーニング) = 0.08
: 20. 正答率(テスト) = 0.1135
Generation: 30. 正答率(トレーニング) = 0.15
: 30. 正答率(テスト) = 0.1135
Generation: 40. 正答率(トレーニング) = 0.13
: 40. 正答率(テスト) = 0.1135
Generation: 50. 正答率(トレーニング) = 0.11
: 50. 正答率(テスト) = 0.1135
Generation: 60. 正答率(トレーニング) = 0.14
: 60. 正答率(テスト) = 0.1135
Generation: 70. 正答率(トレーニング) = 0.1
: 70. 正答率(テスト) = 0.1135
Generation: 80. 正答率(トレーニング) = 0.14
: 80. 正答率(テスト) = 0.1135
Generation: 90. 正答率(トレーニング) = 0.13
: 90. 正答率(テスト) = 0.1135
Generation: 100. 正答率(トレーニング) = 0.11
: 100. 正答率(テスト) = 0.1135
Generation: 910. 正答率(トレーニング) = 0.08
: 910. 正答率(テスト) = 0.0958
Generation: 920. 正答率(トレーニング) = 0.09
: 920. 正答率(テスト) = 0.0958
Generation: 930. 正答率(トレーニング) = 0.06
: 930. 正答率(テスト) = 0.0982
Generation: 940. 正答率(トレーニング) = 0.08
: 940. 正答率(テスト) = 0.1135
Generation: 950. 正答率(トレーニング) = 0.13
: 950. 正答率(テスト) = 0.1135
Generation: 960. 正答率(トレーニング) = 0.12
: 960. 正答率(テスト) = 0.1135
Generation: 970. 正答率(トレーニング) = 0.11
: 970. 正答率(テスト) = 0.1135
Generation: 980. 正答率(トレーニング) = 0.13
: 980. 正答率(テスト) = 0.1135
Generation: 990. 正答率(トレーニング) = 0.11
: 990. 正答率(テスト) = 0.1135
Generation: 1000. 正答率(トレーニング) = 0.1135

```

▼ MomentumをもとにAdaGradを作つてみよう

```

θ = 1e-4 とする
    | |
# AdaGradを作つてみよう
# データの読み込み
(x_train, d_train), (x_test, d_test) = load_mnist(normalize=True, one_hot_label=True)

print("データ読み込み完了")

# batch_normalizationの設定 =====
# use_batchnorm = True
use_batchnorm = False
# =====

network = MultiLayerNet(input_size=784, hidden_size_list=[40, 20], output_size=10, activation='sigmoid', weight_init_std=0.01,
                        use_batchnorm=use_batchnorm)

iters_num = 1000
# iters_num = 500 # 処理を短縮

train_size = x_train.shape[0]
batch_size = 100
learning_rate = 0.01

# AdaGradでは不需要
# =====

#momentum = 0.9
# =====

train_loss_list = []
accuracies_train = []
accuracies_test = []

plot_interval=10

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

    # 勾配
    grad = network.gradient(x_batch, d_batch)
    if i == 0:
        h = {}

```

```

for key in ('W1', 'W2', 'W3', 'b1', 'b2', 'b3'):

    # 変更しよう
    # =====
    ...
    if i == 0:
        h[key] = np.zeros_like(network.params[key])
        h[key] = momentum * h[key] - learning_rate * grad[key]
        network.params[key] += h[key]
    ...
    # =====

    if i == 0:
        h[key] = np.full_like(network.params[key], 1e-4)
    else:
        h[key] += np.square(grad[key])
        network.params[key] -= learning_rate * grad[key] / (np.sqrt(h[key]))

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

    if (i + 1) % plot_interval == 0:
        accr_test = network.accuracy(x_test, d_test)
        accuracies_test.append(accr_test)
        accr_train = network.accuracy(x_batch, d_batch)
        accuracies_train.append(accr_train)

        # 結果が長くなるので一部省略
        if i < 100 or 900 < i:
            print('Generation: ' + str(i+1) + '. 正答率(トレーニング) = ' + str(accr_train))
            print('           : ' + str(i+1) + '. 正答率(テスト) = ' + str(accr_test))

    lists = range(0, iters_num, plot_interval)
    plt.plot(lists, accuracies_train, label="training set")
    plt.plot(lists, accuracies_test, label="test set")
    plt.legend(loc="lower right")
    plt.title("accuracy")
    plt.xlabel("count")
    plt.ylabel("accuracy")
    plt.ylim(0, 1.0)
    # グラフの表示
    plt.show()

```

```

データ読み込み完了
Generation: 10. 正答率(トレーニング) = 0.18
: 10. 正答率(テスト) = 0.0974
Generation: 20. 正答率(トレーニング) = 0.13
: 20. 正答率(テスト) = 0.1135
Generation: 30. 正答率(トレーニング) = 0.12
: 30. 正答率(テスト) = 0.1135
Generation: 40. 正答率(トレーニング) = 0.1
: 40. 正答率(テスト) = 0.1135
Generation: 50. 正答率(トレーニング) = 0.19
: 50. 正答率(テスト) = 0.1135
Generation: 60. 正答率(トレーニング) = 0.14
: 60. 正答率(テスト) = 0.1135
Generation: 70. 正答率(トレーニング) = 0.12
: 70. 正答率(テスト) = 0.1135
Generation: 80. 正答率(トレーニング) = 0.16

```

▼ RSMprop

```

generation: 100. 正答率(トレーニング) = 0.13

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

print("データ読み込み完了")

# batch_normalizationの設定 =====
# use_batchnorm = True
use_batchnorm = False
# =====

network = MultiLayerNet(input_size=784, hidden_size_list=[40, 20], output_size=10, activation='sigmoid', weight_init_std=0.01,
                        use_batchnorm=use_batchnorm)

iters_num = 1000
train_size = x_train.shape[0]
batch_size = 100
learning_rate = 0.01
decay_rate = 0.99

train_loss_list = []
accuracies_train = []
accuracies_test = []

plot_interval=10

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

    # 勾配
    grad = network.gradient(x_batch, d_batch)
    if i == 0:
        h = {}
    for key in ('W1', 'W2', 'W3', 'b1', 'b2', 'b3'):
        if i == 0:
            h[key] = np.zeros_like(network.params[key])
        h[key] *= decay_rate
        h[key] += (1 - decay_rate) * np.square(grad[key])
        network.params[key] -= learning_rate * grad[key] / (np.sqrt(h[key]) + 1e-7)

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

    if (i + 1) % plot_interval == 0:
        accr_test = network.accuracy(x_test, d_test)
        accuracies_test.append(accr_test)
        accr_train = network.accuracy(x_batch, d_batch)
        accuracies_train.append(accr_train)

    # 結果が長くなるので一部省略
    if i < 100 or 900 < i:
        print('Generation: ' + str(i+1) + '. 正答率(トレーニング) = ' + str(accr_train))
        print('           : ' + str(i+1) + '. 正答率(テスト) = ' + str(accr_test))

lists = range(0, iters_num, plot_interval)
plt.plot(lists, accuracies_train, label="training set")
plt.plot(lists, accuracies_test, label="test set")
plt.legend(loc="lower right")
plt.title("accuracy")
plt.xlabel("count")
plt.ylabel("accuracy")
plt.ylim(0, 1.0)
# グラフの表示

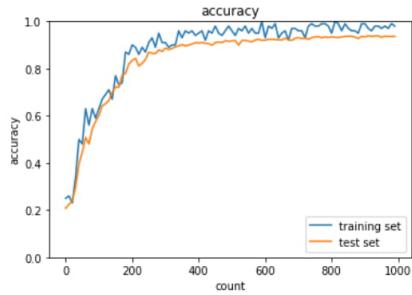
```

```

plt.show()

データ読み込み完了
Generation: 10. 正答率(トレーニング) = 0.25
: 10. 正答率(テスト) = 0.2078
Generation: 20. 正答率(トレーニング) = 0.26
: 20. 正答率(テスト) = 0.2239
Generation: 30. 正答率(トレーニング) = 0.23
: 30. 正答率(テスト) = 0.2397
Generation: 40. 正答率(トレーニング) = 0.34
: 40. 正答率(テスト) = 0.2969
Generation: 50. 正答率(トレーニング) = 0.5
: 50. 正答率(テスト) = 0.3958
Generation: 60. 正答率(トレーニング) = 0.48
: 60. 正答率(テスト) = 0.4448
Generation: 70. 正答率(トレーニング) = 0.63
: 70. 正答率(テスト) = 0.5078
Generation: 80. 正答率(トレーニング) = 0.56
: 80. 正答率(テスト) = 0.4803
Generation: 90. 正答率(トレーニング) = 0.63
: 90. 正答率(テスト) = 0.5437
Generation: 100. 正答率(トレーニング) = 0.59
: 100. 正答率(テスト) = 0.576
Generation: 910. 正答率(トレーニング) = 0.99
: 910. 正答率(テスト) = 0.9339
Generation: 920. 正答率(トレーニング) = 0.97
: 920. 正答率(テスト) = 0.9391
Generation: 930. 正答率(トレーニング) = 0.96
: 930. 正答率(テスト) = 0.9359
Generation: 940. 正答率(トレーニング) = 0.98
: 940. 正答率(テスト) = 0.9382
Generation: 950. 正答率(トレーニング) = 0.98
: 950. 正答率(テスト) = 0.9385
Generation: 960. 正答率(トレーニング) = 0.97
: 960. 正答率(テスト) = 0.9322
Generation: 970. 正答率(トレーニング) = 0.98
: 970. 正答率(テスト) = 0.9374
Generation: 980. 正答率(トレーニング) = 0.97
: 980. 正答率(テスト) = 0.9356
Generation: 990. 正答率(トレーニング) = 0.99
: 990. 正答率(テスト) = 0.9357
Generation: 1000. 正答率(トレーニング) = 0.98
: 1000. 正答率(テスト) = 0.9362

```



▼ Adam

```

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

print("データ読み込み完了")

# batch_normalizationの設定 =====
# use_batchnorm = True
use_batchnorm = False
# =====

network = MultiLayerNet(input_size=784, hidden_size_list=[40, 20], output_size=10, activation='sigmoid', weight_init_std=0.01,
                        use_batchnorm=use_batchnorm)

iters_num = 1000
train_size = x_train.shape[0]
batch_size = 100
learning_rate = 0.01
beta1 = 0.9
beta2 = 0.999

train_loss_list = []
accuracies_train = []
accuracies_test = []

```

```

plot_interval=10

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

    # 勾配
    grad = network.gradient(x_batch, d_batch)
    if i == 0:
        m = {}
        v = {}
    learning_rate_t = learning_rate * np.sqrt(1.0 - beta2 ** (i + 1)) / (1.0 - beta1 ** (i + 1))
    for key in ('W1', 'W2', 'W3', 'b1', 'b2', 'b3'):
        if i == 0:
            m[key] = np.zeros_like(network.params[key])
            v[key] = np.zeros_like(network.params[key])

        m[key] += (1 - beta1) * (grad[key] - m[key])
        v[key] += (1 - beta2) * (grad[key] ** 2 - v[key])
        network.params[key] -= learning_rate_t * m[key] / (np.sqrt(v[key]) + 1e-7)

    if (i + 1) % plot_interval == 0:
        accr_test = network.accuracy(x_test, d_test)
        accuracies_test.append(accr_test)
        accr_train = network.accuracy(x_batch, d_batch)
        accuracies_train.append(accr_train)
        loss = network.loss(x_batch, d_batch)
        train_loss_list.append(loss)

    # 結果が長くなるので一部省略
    if i < 100 or 900 < i:
        print('Generation: ' + str(i+1) + '. 正答率(トレーニング) = ' + str(accr_train))
        print('           : ' + str(i+1) + '. 正答率(テスト) = ' + str(accr_test))

lists = range(0, iters_num, plot_interval)
plt.plot(lists, accuracies_train, label="training set")
plt.plot(lists, accuracies_test, label="test set")
plt.legend(loc="lower right")
plt.title("accuracy")
plt.xlabel("count")
plt.ylabel("accuracy")
plt.ylim(0, 1.0)
# グラフの表示
plt.show()

```

```
データ読み込み完了
Generation: 10. 正答率(トレーニング) = 0.14
           : 10. 正答率(テスト) = 0.0958
Generation: 20. 正答率(トレーニング) = 0.07
           : 20. 正答率(テスト) = 0.1135
Generation: 30. 正答率(トレーニング) = 0.26
           : 30. 正答率(テスト) = 0.1982
Generation: 40. 正答率(トレーニング) = 0.29
           : 40. 正答率(テスト) = 0.2112
Generation: 50. 正答率(トレーニング) = 0.21
           : 50. 正答率(テスト) = 0.2248
Generation: 60. 正答率(トレーニング) = 0.23
           : 60. 正答率(テスト) = 0.234
Generation: 70. 正答率(トレーニング) = 0.35
           : 70. 正答率(テスト) = 0.3043
Generation: 80. 正答率(トレーニング) = 0.29
           : 80. 正答率(テスト) = 0.3156
Generation: 90. 正答率(トレーニング) = 0.43
           : 90. 正答率(テスト) = 0.3701
Generation: 100. 正答率(トレーニング) = 0.38
```

[try] 学習率を変えてみよう

```
: 920. 正答率(テスト) = 0.9512
```

[try] 活性化関数と重みの初期化方法を変えてみよう

初期状態ではsigmoid - gauss
activationはReLU、weight_init_stdは別の数値や'Xavier'・'He'に変更可能

```
: 960. 正答率(テスト) = 0.9562
```

▼ [try] バッヂ正規化をしてみよう

use_batchnormをTrueにしよう

```
----- 1000 正答率(テスト) = 0.95
# データの読み込み
(x_train, d_train), (x_test, d_test) = load_mnist(normalize=True, one_hot_label=True)
print("データ読み込み完了")

# batch_normalizationの設定 =====
use_batchnorm = True
# use_batchnorm = False
# =====

network = MultiLayerNet(input_size=784, hidden_size_list=[40, 20], output_size=10, activation='relu', weight_init_std='He',
                        use_batchnorm=use_batchnorm)

iters_num = 1000
train_size = x_train.shape[0]
batch_size = 100
learning_rate = 0.01
beta1 = 0.9
beta2 = 0.999

train_loss_list = []
accuracies_train = []
accuracies_test = []

plot_interval=10

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

    # 勾配
    grad = network.gradient(x_batch, d_batch)
    if i == 0:
        m = {}
        v = {}
    learning_rate_t = learning_rate * np.sqrt(1.0 - beta2 ** (i + 1)) / (1.0 - beta1 ** (i + 1))
    for key in ('W1', 'W2', 'W3', 'b1', 'b2', 'b3'):
        if i == 0:
            m[key] = np.zeros_like(network.params[key])
            v[key] = np.zeros_like(network.params[key])

        m[key] += (1 - beta1) * (grad[key] - m[key])
        v[key] += (1 - beta2) * (grad[key] ** 2 - v[key])
        network.params[key] -= learning_rate_t * m[key] / (np.sqrt(v[key]) + 1e-7)

    if (i + 1) % plot_interval == 0:
```

```

    if i % plot_interval == 0:
        accr_test = network.accuracy(x_test, d_test)
        accuracies_test.append(accr_test)
        accr_train = network.accuracy(x_batch, d_batch)
        accuracies_train.append(accr_train)
        loss = network.loss(x_batch, d_batch)
        train_loss_list.append(loss)

    # 結果が長くなるので一部省略
    if i < 100 or 900 < i:
        print('Generation: ' + str(i+1) + '. 正答率(トレーニング) = ' + str(accr_train))
        print('           : ' + str(i+1) + '. 正答率(テスト) = ' + str(accr_test))

```

```

lists = range(0, iter_num, plot_interval)
plt.plot(lists, accuracies_train, label="training set")
plt.plot(lists, accuracies_test, label="test set")
plt.legend(loc="lower right")
plt.title("accuracy")
plt.xlabel("count")
plt.ylabel("accuracy")
plt.ylim(0, 1.0)
# グラフの表示
plt.show()

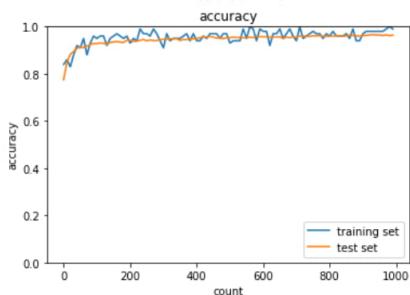
```

データ読み込み完了

```

Generation: 10. 正答率(トレーニング) = 0.84
           : 10. 正答率(テスト) = 0.7748
Generation: 20. 正答率(トレーニング) = 0.86
           : 20. 正答率(テスト) = 0.8499
Generation: 30. 正答率(トレーニング) = 0.83
           : 30. 正答率(テスト) = 0.8824
Generation: 40. 正答率(トレーニング) = 0.88
           : 40. 正答率(テスト) = 0.8947
Generation: 50. 正答率(トレーニング) = 0.92
           : 50. 正答率(テスト) = 0.9072
Generation: 60. 正答率(トレーニング) = 0.91
           : 60. 正答率(テスト) = 0.9149
Generation: 70. 正答率(トレーニング) = 0.95
           : 70. 正答率(テスト) = 0.9105
Generation: 80. 正答率(トレーニング) = 0.88
           : 80. 正答率(テスト) = 0.9172
Generation: 90. 正答率(トレーニング) = 0.93
           : 90. 正答率(テスト) = 0.9237
Generation: 100. 正答率(トレーニング) = 0.96
           : 100. 正答率(テスト) = 0.9289
Generation: 910. 正答率(トレーニング) = 0.97
           : 910. 正答率(テスト) = 0.9621
Generation: 920. 正答率(トレーニング) = 0.98
           : 920. 正答率(テスト) = 0.963
Generation: 930. 正答率(トレーニング) = 0.98
           : 930. 正答率(テスト) = 0.9648
Generation: 940. 正答率(トレーニング) = 0.98
           : 940. 正答率(テスト) = 0.9659
Generation: 950. 正答率(トレーニング) = 0.98
           : 950. 正答率(テスト) = 0.9648
Generation: 960. 正答率(トレーニング) = 0.98
           : 960. 正答率(テスト) = 0.9643
Generation: 970. 正答率(トレーニング) = 0.98
           : 970. 正答率(テスト) = 0.9625
Generation: 980. 正答率(トレーニング) = 0.99
           : 980. 正答率(テスト) = 0.9652
Generation: 990. 正答率(トレーニング) = 1.0
           : 990. 正答率(テスト) = 0.9618
Generation: 1000. 正答率(トレーニング) = 0.99
           : 1000. 正答率(テスト) = 0.964

```



▼ Section3:過学習

学習用データにマッチしすぎてしまって、ほかのデータに対してあまりうまくいかなくなること。 例) 自分で買った問題集ばかりやつて、問題集は完璧だが、問題集しかあまり解けないとか

- 原因
 - パラメータの数が多い
 - パラメータの値が適切でない
 - ノードが多いなど
 - ニューラルネットワークに対して自由度が高い

- 3-1 L1正則化、L2正則化

- 正則化とは ニューラルネットワークの自由度をそぐ方法
- Weight decay (荷重減衰)

- 過学習の原因

- 重みが大きい値をとることで、過学習が発生することがある
(一部の値に極端に反応している)

- 過学習の解決策

- 誤差に対して、正則化項を加算することで、重みを抑制

- L1正則化(ラッソ回帰)、L2正則化 (リッジ回帰)

- 式とコード

$$E_n(W) + \frac{1}{p} \lambda ||x||_p : \text{誤差関数に、 } p \text{ ノルムを加える}$$

```
weight_decay += weight_decay_lambda
*np.sum(np.abs(network.params['W' + str(idx)]))
loss = network.loss(x_batch,d_batch) + weight_decay
```

$$||x||_p = (|x_1|^p + \dots + |x_n|^p)^{\frac{1}{p}} : p \text{ ノルムの計算}$$

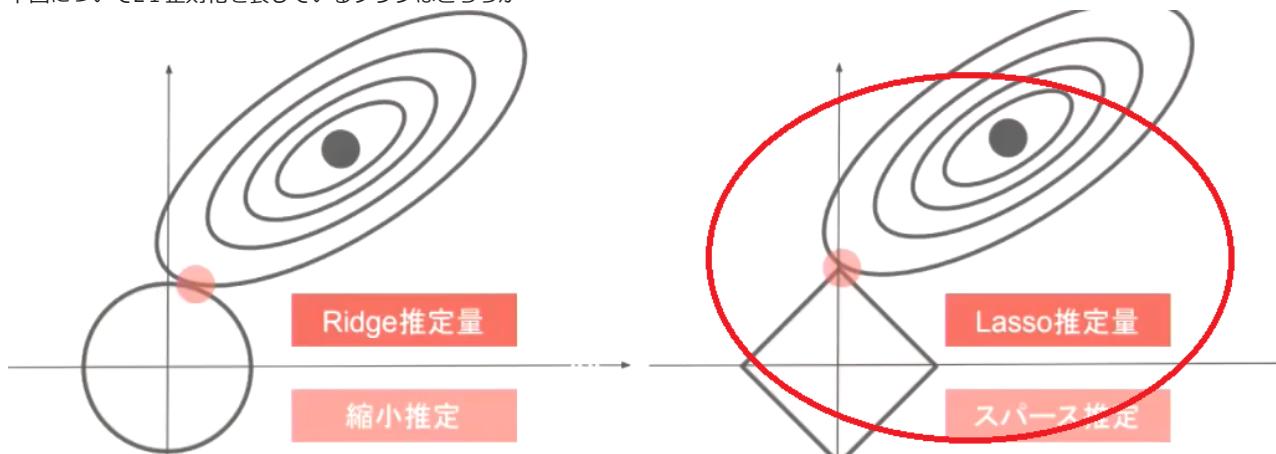
```
np.sum(np.abs(network.params['W' + str(idx)]))
```

p=1の場合、L1正則化と呼ぶ

p=2の場合、L2正則化と呼ぶ

- 確認テスト4

下図についてL1正則化を表しているグラフはどちらか



⇒右の図

- 3-1-2 例題

- 問5 (4)

L2ノルムは $||param||^2$ なので微分した値、 $2 * param$ (2は係数に吸収される)

- 問6 (3) L1ノルムは、 $|param|$ の微分なので $sign(param)$ §§

$$np.sign(x) = \begin{cases} 1 & (x > 0) \\ 0 & (x = 0) \\ -1 & (x < 0) \end{cases}$$

§

- 3-2 ドロップアウト

ノードを切断して、学習させること。

- プログラム演習
-

- ▼ 準備

- ▼ Google ドライブのマウント

```
from google.colab import drive
drive.mount('/content/drive')

Mounted at /content/drive
```

- ▼ sys.path の設定

以下では、Google ドライブのマイドライブ直下にDNN_code フォルダを置くことを仮定しています。必要に応じて、パスを変更してください。

```
import sys
sys.path.append('/content/drive/My Drive')
sys.path.append('/content/drive/My Drive/lesson_2')
```

- ▼ overfitting

```
import numpy as np
from collections import OrderedDict
from common import layers
from data.mnist import load_mnist
import matplotlib.pyplot as plt
from multi_layer_net import MultiLayerNet
from common import optimizer

(x_train, d_train), (x_test, d_test) = load_mnist(normalize=True)

print("データ読み込み完了")

# 過学習を再現するために、学習データを削減
x_train = x_train[:300]
d_train = d_train[:300]

network = MultiLayerNet(input_size=784, hidden_size_list=[100, 100, 100, 100, 100], output_size=10)
optimizer = optimizer.SGD(learning_rate=0.01)

iters_num = 1000
train_size = x_train.shape[0]
batch_size = 100

train_loss_list = []
accuracies_train = []
accuracies_test = []

plot_interval=10

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

    grad = network.gradient(x_batch, d_batch)
    optimizer.update(network.params, grad)

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

    if (i+1) % plot_interval == 0:
        accr_train = network.accuracy(x_train, d_train)
        accr_test = network.accuracy(x_test, d_test)
        accuracies_train.append(accr_train)
        accuracies_test.append(accr_test)

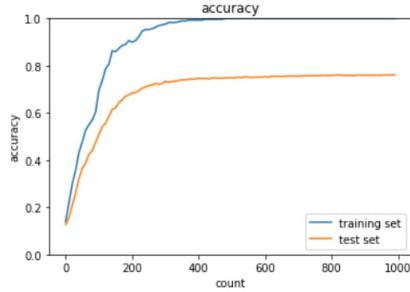
    # 結果が長くなるので一部省略
    if i < 50 or 950 < i:
        print('Generation: ' + str(i+1) + ', 正答率(トレーニング) = ' + str(accr_train))
        print('                 : ' + str(i+1) + ', 正答率(テスト) = ' + str(accr_test))
```

```

lists = range(0, iters_num, plot_interval)
plt.plot(lists, accuracies_train, label="training set")
plt.plot(lists, accuracies_test, label="test set")
plt.legend(loc="lower right")
plt.title("accuracy")
plt.xlabel("count")
plt.ylabel("accuracy")
plt.ylim(0, 1.0)
# グラフの表示
plt.show()

データ読み込み完了
Generation: 10. 正答率(トレーニング) = 0.14
: 10. 正答率(テスト) = 0.1271
Generation: 20. 正答率(トレーニング) = 0.22
: 20. 正答率(テスト) = 0.1561
Generation: 30. 正答率(トレーニング) = 0.3
: 30. 正答率(テスト) = 0.2082
Generation: 40. 正答率(トレーニング) = 0.35666666666666667
: 40. 正答率(テスト) = 0.2609
Generation: 50. 正答率(トレーニング) = 0.4333333333333335
: 50. 正答率(テスト) = 0.3221
Generation: 960. 正答率(トレーニング) = 1.0
: 960. 正答率(テスト) = 0.7602
Generation: 970. 正答率(トレーニング) = 1.0
: 970. 正答率(テスト) = 0.7605
Generation: 980. 正答率(トレーニング) = 1.0
: 980. 正答率(テスト) = 0.7611
Generation: 990. 正答率(トレーニング) = 1.0
: 990. 正答率(テスト) = 0.7618
Generation: 1000. 正答率(トレーニング) = 1.0
: 1000. 正答率(テスト) = 0.7607

```



▼ weight decay

L2

```

from common import optimizer

(x_train, d_train), (x_test, d_test) = load_mnist(normalize=True)

print("データ読み込み完了")

# 過学習を再現するために、学習データを削減
x_train = x_train[:300]
d_train = d_train[:300]

network = MultiLayerNet(input_size=784, hidden_size_list=[100, 100, 100, 100, 100, 100], output_size=10)

iters_num = 1000
train_size = x_train.shape[0]
batch_size = 100
learning_rate=0.01

train_loss_list = []
accuracies_train = []
accuracies_test = []

plot_interval=10
hidden_layer_num = network.hidden_layer_num

# 正則化強度設定 =====
weight_decay_lambda = 0.1
# =====

for i in range(iters_num):

```

```

batch_mask = np.random.choice(train_size, batch_size)
x_batch = x_train[batch_mask]
d_batch = d_train[batch_mask]

grad = network.gradient(x_batch, d_batch)
weight_decay = 0

for idx in range(1, hidden_layer_num+1):
    grad['W' + str(idx)] = network.layers['Affine' + str(idx)].dW + weight_decay_lambda * network.params['W' + str(idx)]
    grad['b' + str(idx)] = network.layers['Affine' + str(idx)].db
    network.params['W' + str(idx)] -= learning_rate * grad['W' + str(idx)]
    network.params['b' + str(idx)] -= learning_rate * grad['b' + str(idx)]
    weight_decay += 0.5 * weight_decay_lambda * np.sqrt(np.sum(network.params['W' + str(idx)] ** 2))

loss = network.loss(x_batch, d_batch) + weight_decay
train_loss_list.append(loss)

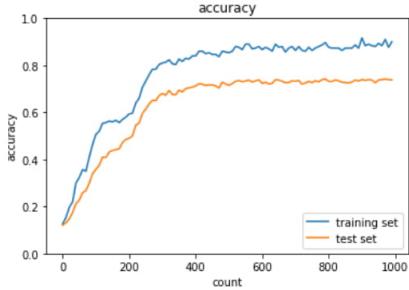
if (i+1) % plot_interval == 0:
    accr_train = network.accuracy(x_train, d_train)
    accr_test = network.accuracy(x_test, d_test)
    accuracies_train.append(accr_train)
    accuracies_test.append(accr_test)

# 結果が長くなるので一部省略
if i < 50 or 950 < i:
    print('Generation: ' + str(i+1) + '. 正答率(トレーニング) = ' + str(accr_train))
    print('           : ' + str(i+1) + '. 正答率(テスト) = ' + str(accr_test))

lists = range(0, iters_num, plot_interval)
plt.plot(lists, accuracies_train, label="training set")
plt.plot(lists, accuracies_test, label="test set")
plt.legend(loc="lower right")
plt.title("accuracy")
plt.xlabel("count")
plt.ylabel("accuracy")
plt.ylim(0, 1.0)
# グラフの表示
plt.show()

データ読み込み完了
Generation: 10. 正答率(トレーニング) = 0.12666666666666668
           : 10. 正答率(テスト) = 0.1209
Generation: 20. 正答率(トレーニング) = 0.1533333333333332
           : 20. 正答率(テスト) = 0.1306
Generation: 30. 正答率(トレーニング) = 0.19666666666666666
           : 30. 正答率(テスト) = 0.1476
Generation: 40. 正答率(トレーニング) = 0.22
           : 40. 正答率(テスト) = 0.1747
Generation: 50. 正答率(トレーニング) = 0.3
           : 50. 正答率(テスト) = 0.2123
Generation: 960. 正答率(トレーニング) = 0.8933333333333333
           : 960. 正答率(テスト) = 0.7378
Generation: 970. 正答率(トレーニング) = 0.8833333333333333
           : 970. 正答率(テスト) = 0.7392
Generation: 980. 正答率(トレーニング) = 0.91
           : 980. 正答率(テスト) = 0.7417
Generation: 990. 正答率(トレーニング) = 0.8766666666666667
           : 990. 正答率(テスト) = 0.7389
Generation: 1000. 正答率(トレーニング) = 0.9
           : 1000. 正答率(テスト) = 0.7383

```



▼ L1

```

(x_train, d_train), (x_test, d_test) = load_mnist(normalize=True)

print("データ読み込み完了")

# 過学習を再現するために、学習データを削減

```

```

x_train = x_train[:300]
d_train = d_train[:300]

network = MultiLayerNet(input_size=784, hidden_size_list=[100, 100, 100, 100, 100, 100], output_size=10)

iters_num = 1000
train_size = x_train.shape[0]
batch_size = 100
learning_rate=0.1

train_loss_list = []
accuracies_train = []
accuracies_test = []

plot_interval=10
hidden_layer_num = network.hidden_layer_num

# 正則化強度設定 =====
weight_decay_lambda = 0.05
# =====

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

    grad = network.gradient(x_batch, d_batch)
    weight_decay = 0

    for idx in range(1, hidden_layer_num+1):
        grad['W' + str(idx)] = network.layers['Affine' + str(idx)].dW + weight_decay_lambda * np.sign(network.params['W' + str(idx)])
        grad['b' + str(idx)] = network.layers['Affine' + str(idx)].db
        network.params['W' + str(idx)] -= learning_rate * grad['W' + str(idx)]
        network.params['b' + str(idx)] -= learning_rate * grad['b' + str(idx)]
        weight_decay += weight_decay_lambda * np.sum(np.abs(network.params['W' + str(idx)]))

    loss = network.loss(x_batch, d_batch) + weight_decay
    train_loss_list.append(loss)

    if (i+1) % plot_interval == 0:
        accr_train = network.accuracy(x_train, d_train)
        accr_test = network.accuracy(x_test, d_test)
        accuracies_train.append(accr_train)
        accuracies_test.append(accr_test)

    # 結果が長くなるので一部省略
    if i < 50 or 950 < i:
        print('Generation: ' + str(i+1) + '. 正答率(トレーニング) = ' + str(accr_train))
        print('           : ' + str(i+1) + '. 正答率(テスト) = ' + str(accr_test))

lists = range(0, iters_num, plot_interval)
plt.plot(lists, accuracies_train, label="training set")
plt.plot(lists, accuracies_test, label="test set")
plt.legend(loc="lower right")
plt.title("accuracy")
plt.xlabel("count")
plt.ylabel("accuracy")
plt.ylim(0, 1.0)
# グラフの表示
plt.show()

```

```
データ読み込み完了
Generation: 10. 正答率(トレーニング) = 0.28666666666666667
          : 10. 正答率(テスト) = 0.2488
Generation: 20. 正答率(トレーニング) = 0.13
          : 20. 正答率(テスト) = 0.1178
Generation: 30. 正答率(トレーニング) = 0.13
          : 30. 正答率(テスト) = 0.1135
Generation: 40. 正答率(トレーニング) = 0.13
          : 40. 正答率(テスト) = 0.1135
Generation: 50. 正答率(トレーニング) = 0.13
          : 50. 正答率(テスト) = 0.1135
Generation: 960. 正答率(トレーニング) = 0.13
          : 960. 正答率(テスト) = 0.1135
Generation: 970 正答率(トレーニング) = 0.13
```

[try] weight_decay_lambdaの値を変更して正則化の強さを確認しよう

```
: 990. 正答率(テスト) = 0.1135
```

▼ Dropout

```
|           |
class Dropout:
    def __init__(self, dropout_ratio=0.5):
        self.dropout_ratio = dropout_ratio
        self.mask = None

    def forward(self, x, train_flg=True):
        if train_flg:
            self.mask = np.random.rand(*x.shape) > self.dropout_ratio
            return x * self.mask
        else:
            return x * (1.0 - self.dropout_ratio)

    def backward(self, dout):
        return dout * self.mask

from common import optimizer
(x_train, d_train), (x_test, d_test) = load_mnist(normalize=True)

print("データ読み込み完了")

# 過学習を再現するために、学習データを削減
x_train = x_train[:300]
d_train = d_train[:300]

# ドロップアウト設定 =====
use_dropout = True
dropout_ratio = 0.15
# =====

network = MultiLayerNet(input_size=784, hidden_size_list=[100, 100, 100, 100, 100], output_size=10,
                        weight_decay_lambda=weight_decay_lambda, use_dropout=use_dropout, dropout_ratio=dropout_ratio)
optimizer = optimizer.SGD(learning_rate=0.01)
# optimizer = optimizer.Momentum(learning_rate=0.01, momentum=0.9)
# optimizer = optimizer.AdaGrad(learning_rate=0.01)
# optimizer = optimizer.Adam()

iters_num = 1000
train_size = x_train.shape[0]
batch_size = 100

train_loss_list = []
accuracies_train = []
accuracies_test = []

plot_interval=10

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

    grad = network.gradient(x_batch, d_batch)
    optimizer.update(network.params, grad)

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

    if (i+1) % plot_interval == 0:
        accr_train = network.accuracy(x_train, d_train)
        accr_test = network.accuracy(x_test, d_test)
        accuracies_train.append(accr_train)
```

```

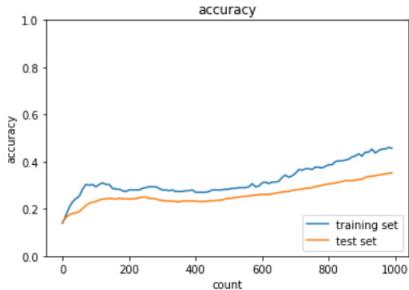
    accuracies_test.append(accr_test)

    # 結果が長くなるので一部省略
    if i < 50 or 950 < i:
        print("Generation: " + str(i+1) + ". 正答率(トレーニング) = " + str(accr_train))
        print("          : " + str(i+1) + ". 正答率(テスト) = " + str(accr_test))

    lists = range(0, iters_num, plot_interval)
    plt.plot(lists, accuracies_train, label="training set")
    plt.plot(lists, accuracies_test, label="test set")
    plt.legend(loc="lower right")
    plt.title("accuracy")
    plt.xlabel("count")
    plt.ylabel("accuracy")
    plt.ylim(0, 1.0)
    # グラフの表示
    plt.show()

データ読み込み完了
Generation: 10. 正答率(トレーニング) = 0.14
          : 10. 正答率(テスト) = 0.1449
Generation: 20. 正答率(トレーニング) = 0.17333333333333334
          : 20. 正答率(テスト) = 0.163
Generation: 30. 正答率(トレーニング) = 0.20666666666666667
          : 30. 正答率(テスト) = 0.1748
Generation: 40. 正答率(トレーニング) = 0.23
          : 40. 正答率(テスト) = 0.1809
Generation: 50. 正答率(トレーニング) = 0.24333333333333335
          : 50. 正答率(テスト) = 0.1827
Generation: 960. 正答率(トレーニング) = 0.44666666666666666
          : 960. 正答率(テスト) = 0.3431
Generation: 970. 正答率(トレーニング) = 0.4533333333333333
          : 970. 正答率(テスト) = 0.3455
Generation: 980. 正答率(トレーニング) = 0.4533333333333333
          : 980. 正答率(テスト) = 0.3478
Generation: 990. 正答率(トレーニング) = 0.46
          : 990. 正答率(テスト) = 0.3505
Generation: 1000. 正答率(トレーニング) = 0.45666666666666667
          : 1000. 正答率(テスト) = 0.3527

```



[try] dropout_ratioの値を変更してみよう

[try] optimizerとdropout_ratioの値を変更してみよう

▼ Dropout + L1

```

from common import optimizer
(x_train, d_train), (x_test, d_test) = load_mnist(normalize=True)

print("データ読み込み完了")

# 過学習を再現するために、学習データを削減
x_train = x_train[:300]
d_train = d_train[:300]

# ドロップアウト設定 =====
use_dropout = True
dropout_ratio = 0.08
# =====

network = MultiLayerNet(input_size=784, hidden_size_list=[100, 100, 100, 100, 100, 100], output_size=10,
                        use_dropout = use_dropout, dropout_ratio = dropout_ratio)

iters_num = 1000

```

```

train_size = x_train.shape[0]
batch_size = 100
learning_rate=0.01

train_loss_list = []
accuracies_train = []
accuracies_test = []
hidden_layer_num = network.hidden_layer_num

plot_interval=10

# 正則化強度設定 =====
weight_decay_lambda=0.004
# =====

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

    grad = network.gradient(x_batch, d_batch)
    weight_decay = 0

    for idx in range(1, hidden_layer_num+1):
        grad['W' + str(idx)] = network.layers['Affine' + str(idx)].dW + weight_decay_lambda * np.sign(network.params['W' + str(idx)])
        grad['b' + str(idx)] = network.layers['Affine' + str(idx)].db
        network.params['W' + str(idx)] -= learning_rate * grad['W' + str(idx)]
        network.params['b' + str(idx)] -= learning_rate * grad['b' + str(idx)]
        weight_decay += weight_decay_lambda * np.sum(np.abs(network.params['W' + str(idx)]))

    loss = network.loss(x_batch, d_batch) + weight_decay
    train_loss_list.append(loss)

    if (i+1) % plot_interval == 0:
        accr_train = network.accuracy(x_train, d_train)
        accr_test = network.accuracy(x_test, d_test)
        accuracies_train.append(accr_train)
        accuracies_test.append(accr_test)

    # 結果が長くなるので一部省略
    if i < 50 or 950 < i:
        print('Generation: ' + str(i+1) + '. 正答率(トレーニング) = ' + str(accr_train))
        print('          : ' + str(i+1) + '. 正答率(テスト) = ' + str(accr_test))

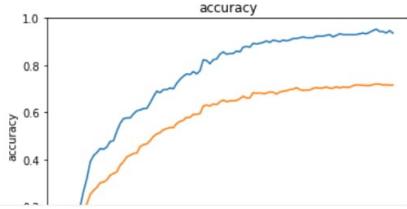
lists = range(0, iters_num, plot_interval)
plt.plot(lists, accuracies_train, label="training set")
plt.plot(lists, accuracies_test, label="test set")
plt.legend(loc="lower right")
plt.title("accuracy")
plt.xlabel("count")
plt.ylabel("accuracy")
plt.ylim(0, 1.0)
# グラフの表示
plt.show()

```

データ読み込み完了
Generation: 10. 正答率(トレーニング) = 0.09666666666666666
: 10. 正答率(テスト) = 0.0928
Generation: 20. 正答率(トレーニング) = 0.13
: 20. 正答率(テスト) = 0.106
Generation: 30. 正答率(トレーニング) = 0.16333333333333333

編集するにはダブルクリックするか Enter キーを押してください

: 40. 正答率(テスト) = 0.1294
Generation: 50. 正答率(トレーニング) = 0.19
: 50. 正答率(テスト) = 0.1482
Generation: 960. 正答率(トレーニング) = 0.9433333333333334
: 960. 正答率(テスト) = 0.7201
Generation: 970. 正答率(トレーニング) = 0.9433333333333334
: 970. 正答率(テスト) = 0.7158
Generation: 980. 正答率(トレーニング) = 0.93666666666666666
: 980. 正答率(テスト) = 0.7173
Generation: 990. 正答率(トレーニング) = 0.94666666666666667
: 990. 正答率(テスト) = 0.7153
Generation: 1000. 正答率(トレーニング) = 0.93666666666666666
: 1000. 正答率(テスト) = 0.7159



▼ Section4: 置み込みニューラルネットワークの概念

画像によく使われるが、汎用的なニューラルネットワーク

- 4-1 置み込み層

置み込み層の全体像

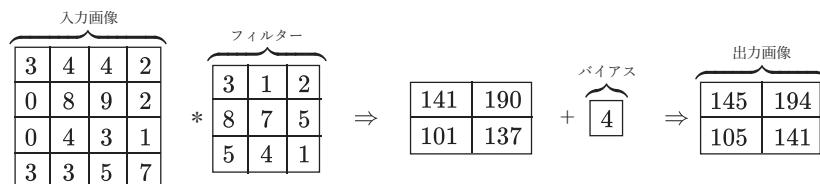
入力値 * フィルター (全結合でいう重み) ⇒ 出力値 +バイアス ⇒ 活性化関数 ⇒ 出力値

入力値に対して、フィルターをずらしながら出力値を計算する

置み込み層では、画像の場合、縦、横、チャンネルの3次元のデータをそのまま学習し、次に伝えることができる。

結論：3次元の空間情報も学習できるような層が置み込み層である。

- 4-1-1 バイアス



$$3 \times 3 + 4 \times 1 + 4 \times 2 + 0 \times 8 + 8 \times 7 + 9 \times 5 + 0 \times 5 + 4 \times 4 + 3 \times 1 = 141$$

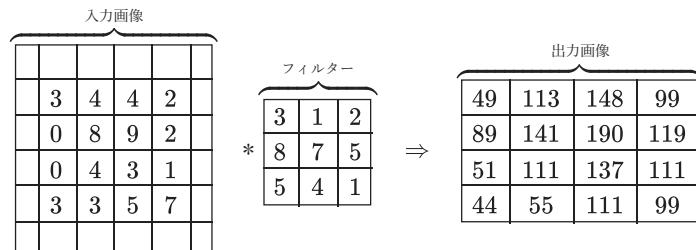
$$4 \times 3 + 4 \times 1 + 2 \times 2 + 8 \times 8 + 9 \times 7 + 2 \times 5 + 4 \times 5 + 3 \times 4 + 1 \times 1 = 190$$

$$8 \times 3 + 9 \times 1 + 2 \times 2 + 4 \times 8 + 3 \times 7 + 1 \times 5 + 3 \times 5 + 4 \times 4 + 7 \times 1 = 137$$

- 4-1-2 パディング

入力画像に対して出力画像はちょっと小さくなる

入力データの周りを広げると同じサイズにできる



- 4-1-3 ストライド

指定数ずらす。

ストライド：2の場合

1	2	4	7	8	1	4
8	8	0	8	7	3	0
8	9	0	0	2	1	9
4	5	9	3	3	4	2
3	4	5	6	4	1	1
5	5	6	8	2	8	4
1	3	7	0	1	8	4

ストライド：2

⇒

1	2	4	7	8	1	4
8	8	0	8	7	3	0
8	9	0	0	2	1	9
4	5	9	3	3	4	2
3	4	5	6	4	1	1
5	5	6	8	2	8	4
1	3	7	0	1	8	4

- 4-1-4 チャンネル

チャンネル：フィルターの数

- 4-2 プーリング層

- 畳み込み演算と組みあわせて使用される場合が多い
- バリエーションがある（Maxプーリング、Avgプーリングなど）
- 畳み込み層と同じく、ずらしながら計算するが

Maxプーリング—枠内の最大値をとる など

計算方法が異なる

- 重みもない

- 確認テスト

- サイズ 6×6 の入力画像をサイズ 2×2 のフィルタで畳み込んだ時の出力画像のサイズは？（ストライドとパディングは1とする）
 $\Rightarrow 7 \times 7$

- 置込み処理 実装演習

▼ 準備

▼ Google ドライブのマウント

```
from google.colab import drive
drive.mount('/content/drive')

Mounted at /content/drive
```

▼ sys.pathの設定

以下では、Google ドライブのマイドライブ直下にDNN_code フォルダを置くことを仮定しています。必要に応じて、パスを変更してください。

```
import sys
sys.path.append('/content/drive/My Drive/')
```

▼ simple convolution network

▼ image to column

```
import pickle
import numpy as np
from collections import OrderedDict
from common import layers
from common import optimizer
from data.mnist import load_mnist
import matplotlib.pyplot as plt

# 画像データを2次元配列に変換
...
input_data: 入力値
filter_h: フィルターの高さ
filter_w: フィルターの横幅
stride: ストライド
pad: パディング
...
def im2col(input_data, filter_h, filter_w, stride=1, pad=0):
    # N: number, C: channel, H: height, W: width
    N, C, H, W = input_data.shape
    # 切り捨て除算
    out_h = (H + 2 * pad - filter_h) // stride + 1
    out_w = (W + 2 * pad - filter_w) // stride + 1

    img = np.pad(input_data, [(0, 0), (0, 0), (pad, pad), (pad, pad)], 'constant')
    col = np.zeros((N, C, filter_h, filter_w, out_h, out_w))

    for y in range(filter_h):
        y_max = y + stride * out_h
        for x in range(filter_w):
            x_max = x + stride * out_w
            col[:, :, y, x, :, :] = img[:, :, y:y_max:stride, x:x_max:stride]

    col = col.transpose(0, 4, 5, 1, 2, 3) # (N, C, filter_h, filter_w, out_h, out_w) -> (N, filter_w, out_h, out_w, C, filter_h)

    col = col.reshape(N * out_h * out_w, -1)
    return col
```

▼ [try] im2colの処理を確認しよう

- ・関数内で transpose の処理をしている行をコメントアウトして下のコードを実行してみよう
- ・input_data の各次元のサイズやフィルターサイズ・ストライド・パディングを変えてみよう

```
# im2colの処理確認
input_data = np.random.rand(2, 1, 4, 4)*100//1 # number, channel, height, widthを表す
print('===== input_data =====\n', input_data)
print('=====')
```

```

-----
filter_h = 3
filter_w = 3
stride = 1
pad = 0
col = im2col(input_data, filter_h=filter_h, filter_w=filter_w, stride=stride, pad=pad)
print('===== col =====\n', col)
print('=====')

===== input_data =====
[[[ 3.  8. 12. 72.]
 [37. 29. 64. 40.]
 [61. 86. 98.  8.]
 [24. 40.  8. 82.]]

[[[24. 31.  3. 82.]
 [69.  8. 78. 87.]
 [99. 97. 26. 51.]
 [32. 64. 40. 65.]]

===== col =====
[[ 3.  8. 12. 37. 29. 64. 61. 86. 98.]
 [ 8. 12. 72. 29. 64. 40. 86. 98.  8.]
 [37. 29. 64. 61. 86. 98. 24. 40.  8.]
 [29. 64. 40. 86. 98.  8. 40.  8. 82.]
 [24. 31.  3. 69.  8. 78. 99. 97. 26.]
 [31.  3. 82.  8. 78. 87. 97. 26. 51.]
 [69.  8. 78. 99. 97. 26. 32. 64. 40.]
 [ 8. 78. 87. 97. 26. 51. 64. 40. 65.]]
=====
```

▼ column to image

```

# 2次元配列を画像データに変換
def col2im(col, input_shape, filter_h, filter_w, stride=1, pad=0):
    # N: number, C: channel, H: height, W: width
    N, C, H, W = input_shape
    # 切り捨て除算
    out_h = (H + 2 * pad - filter_h)//stride + 1
    out_w = (W + 2 * pad - filter_w)//stride + 1
    col = col.reshape(N, out_h, out_w, C, filter_h, filter_w).transpose(0, 3, 4, 5, 1, 2) # (N, filter_h, filter_w, out_h, out_w, C)

    img = np.zeros((N, C, H + 2 * pad + stride - 1, W + 2 * pad + stride - 1))
    for y in range(filter_h):
        y_max = y + stride * out_h
        for x in range(filter_w):
            x_max = x + stride * out_w
            img[:, :, y:y_max:stride, x:x_max:stride] += col[:, :, y, x, :, :]

    return img[:, :, pad:H + pad, pad:W + pad]
```

▼ [try] col2imの処理を確認しよう

- im2colの確認で出力したcolをimageに変換して確認しよう

```

# ここにcol2imでの処理を書こう
img = col2im(col, input_shape=input_data.shape, filter_h=filter_h, filter_w=filter_w, stride=stride, pad=pad)
print(img)

[[[ 3. 16. 24. 72.]
 [ 74. 116. 256. 80.]
 [122. 344. 392. 16.]
 [ 24. 80. 16. 82.]]

[[[ 24. 62.  6. 82.]
 [138. 32. 312. 174.]
 [198. 388. 104. 102.]
 [ 32. 128. 80. 65.]]]
```

▼ convolution class

```

class Convolution:
    # W: フィルター, b: バイアス
    def __init__(self, W, b, stride=1, pad=0):
        self.W = W
        self.b = b
        self.stride = stride
        self.pad = pad
```

```

# 中間データ (backward時に使用)
self.x = None
self.col = None
self.col_W = None

# フィルター・バイアスパラメータの勾配
self.dW = None
self.db = None

def forward(self, x):
    # FN: filter_number, C: channel, FH: filter_height, FW: filter_width
    FN, C, FH, FW = self.W.shape
    N, C, H, W = x.shape
    # 出力値のheight, width
    out_h = 1 + int((H + 2 * self.pad - FH) / self.stride)
    out_w = 1 + int((W + 2 * self.pad - FW) / self.stride)

    # xを行列に変換
    col = im2col(x, FH, FW, self.stride, self.pad)
    # フィルターをxに合わせた行列に変換
    col_W = self.W.reshape(FN, -1).T

    out = np.dot(col, col_W) + self.b
    # 計算のために変えた形式に戻す
    out = out.reshape(N, out_h, out_w, -1).transpose(0, 3, 1, 2)

    self.x = x
    self.col = col
    self.col_W = col_W

    return out

def backward(self, dout):
    FN, C, FH, FW = self.W.shape
    dout = dout.transpose(0, 2, 3, 1).reshape(-1, FN)

    self.db = np.sum(dout, axis=0)
    self.dW = np.dot(self.col.T, dout)
    self.dW = self.dW.transpose(1, 0).reshape(FN, C, FH, FW)

    dcol = np.dot(dout, self.col_W.T)
    # dcolを画像データに変換
    dx = col2im(dcol, self.x.shape, FH, FW, self.stride, self.pad)

    return dx

```

▼ pooling class

```

class Pooling:
    def __init__(self, pool_h, pool_w, stride=1, pad=0):
        self.pool_h = pool_h
        self.pool_w = pool_w
        self.stride = stride
        self.pad = pad

        self.x = None
        self.arg_max = None

    def forward(self, x):
        N, C, H, W = x.shape
        out_h = int(1 + (H - self.pool_h) / self.stride)
        out_w = int(1 + (W - self.pool_w) / self.stride)

        # xを行列に変換
        col = im2col(x, self.pool_h, self.pool_w, self.stride, self.pad)
        # プーリングのサイズに合わせてリサイズ
        col = col.reshape(-1, self.pool_h*self.pool_w)

        #maxプーリング
        arg_max = np.argmax(col, axis=1)
        out = np.max(col, axis=1)
        out = out.reshape(N, out_h, out_w, C).transpose(0, 3, 1, 2)

        self.x = x
        self.arg_max = arg_max

        return out

    def backward(self, dout):
        dout = dout.transpose(0, 2, 3, 1)

```

```

pool_size = self.pool_h * self.pool_w
dmax = np.zeros((dout.size, pool_size))
dmax[np.arange(self.arg_max.size), self.arg_max.flatten()] = dout.flatten()
dmax = dmax.reshape(dout.shape + (pool_size,))

dcol = dmax.reshape(dmax.shape[0] * dmax.shape[1] * dmax.shape[2], -1)
dx = col2im(dcol, self.x.shape, self.pool_h, self.pool_w, self.stride, self.pad)

return dx

```

▼ simple convolution network class

```

class SimpleConvNet:
    # conv - relu - pool - affine - relu - affine - softmax
    def __init__(self, input_dim=(1, 28, 28), conv_param={'filter_num':30, 'filter_size':5, 'pad':0, 'stride':1},
                 hidden_size=100, output_size=10, weight_init_std=0.01):
        filter_num = conv_param['filter_num']
        filter_size = conv_param['filter_size']
        filter_pad = conv_param['pad']
        filter_stride = conv_param['stride']
        input_size = input_dim[1]
        conv_output_size = (input_size - filter_size + 2 * filter_pad) / filter_stride + 1
        pool_output_size = int(filter_num * (conv_output_size / 2) * (conv_output_size / 2))

        # 重みの初期化
        self.params = {}
        self.params['W1'] = weight_init_std * np.random.randn(filter_num, input_dim[0], filter_size, filter_size)
        self.params['b1'] = np.zeros(filter_num)
        self.params['W2'] = weight_init_std * np.random.randn(pool_output_size, hidden_size)
        self.params['b2'] = np.zeros(hidden_size)
        self.params['W3'] = weight_init_std * np.random.randn(hidden_size, output_size)
        self.params['b3'] = np.zeros(output_size)

        # レイヤの生成
        self.layers = OrderedDict()
        self.layers['Conv1'] = layers.Convolution(self.params['W1'], self.params['b1'], conv_param['stride'], conv_param['pad'])
        self.layers['Relu1'] = layers.Relu()
        self.layers['Pool1'] = layers.Pooling(pool_h=2, pool_w=2, stride=2)
        self.layers['Affine1'] = layers.Affine(self.params['W2'], self.params['b2'])
        self.layers['Relu2'] = layers.Relu()
        self.layers['Affine2'] = layers.Affine(self.params['W3'], self.params['b3'])

        self.last_layer = layers.SoftmaxWithLoss()

    def predict(self, x):
        for key in self.layers.keys():
            x = self.layers[key].forward(x)
        return x

    def loss(self, x, d):
        y = self.predict(x)
        return self.last_layer.forward(y, d)

    def accuracy(self, x, d, batch_size=100):
        if d.ndim != 1 : d = np.argmax(d, axis=1)

        acc = 0.0

        for i in range(int(x.shape[0] / batch_size)):
            tx = x[i*batch_size:(i+1)*batch_size]
            td = d[i*batch_size:(i+1)*batch_size]
            y = self.predict(tx)
            y = np.argmax(y, axis=1)
            acc += np.sum(y == td)

        return acc / x.shape[0]

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

        # backward
        dout = 1
        dout = self.last_layer.backward(dout)
        layers = list(self.layers.values())

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

        # 終了

```

```

# 計算
grad = []
grad['W1'] = self.layers['Conv1'].dW, self.layers['Conv1'].db
grad['W2'] = self.layers['Affine1'].dW, self.layers['Affine1'].db
grad['W3'] = self.layers['Affine2'].dW, self.layers['Affine2'].db

return grad

from common import optimizer

# データの読み込み
(x_train, d_train), (x_test, d_test) = load_mnist(flatten=False)
print("データ読み込み完了")

# 処理に時間のかかる場合はデータを削減
x_train, d_train = x_train[:500], d_train[:500]
x_test, d_test = x_test[:100], d_test[:100]

network = SimpleConvNet(input_dim=(1, 28, 28), conv_param = {'filter_num': 30, 'filter_size': 5, 'pad': 0, 'stride': 1},
                        hidden_size=100, output_size=10, weight_init_std=0.01)

optimizer = optimizer.Adam()

iters_num = 1000
train_size = x_train.shape[0]
batch_size = 100

train_loss_list = []
accuracies_train = []
accuracies_test = []

plot_interval=10

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

    grad = network.gradient(x_batch, d_batch)
    optimizer.update(network.params, grad)

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

    if (i+1) % plot_interval == 0:
        accr_train = network.accuracy(x_train, d_train)
        accr_test = network.accuracy(x_test, d_test)
        accuracies_train.append(accr_train)
        accuracies_test.append(accr_test)

    # 結果が長くなるので一部省略
    if i < 50 or 950 < i:
        print('Generation: ' + str(i+1) + '. 正答率(トレーニング) = ' + str(accr_train))
        print('                 : ' + str(i+1) + '. 正答率(テスト) = ' + str(accr_test))

lists = range(0, iters_num, plot_interval)
plt.plot(lists, accuracies_train, label="training set")
plt.plot(lists, accuracies_test, label="test set")
plt.legend(loc="lower right")
plt.title("accuracy")
plt.xlabel("count")
plt.ylabel("accuracy")
plt.ylim(0, 1.0)
# グラフの表示
plt.show()

```

データ読み込み完了

Generation: 10. 正答率(トレーニング) = 0.36
: 10. 正答率(テスト) = 0.31

Generation: 20. 正答率(トレーニング) = 0.654
: 20. 正答率(テスト) = 0.65

Generation: 30. 正答率(トレーニング) = 0.804
: 30. 正答率(テスト) = 0.73

Generation: 40. 正答率(トレーニング) = 0.808
: 40. 正答率(テスト) = 0.75

Generation: 50. 正答率(トレーニング) = 0.848
: 50. 正答率(テスト) = 0.79

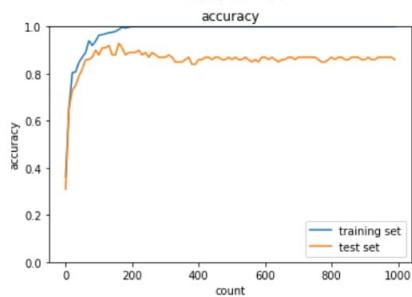
Generation: 960. 正答率(トレーニング) = 1.0
: 960. 正答率(テスト) = 0.87

Generation: 970. 正答率(トレーニング) = 1.0
: 970. 正答率(テスト) = 0.87

Generation: 980. 正答率(トレーニング) = 1.0
: 980. 正答率(テスト) = 0.87

Generation: 990. 正答率(トレーニング) = 1.0
: 990. 正答率(テスト) = 0.87

Generation: 1000. 正答率(トレーニング) = 1.0
: 1000. 正答率(テスト) = 0.86



Section5: 最新のCNN

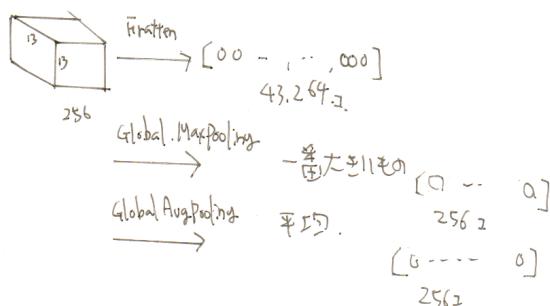
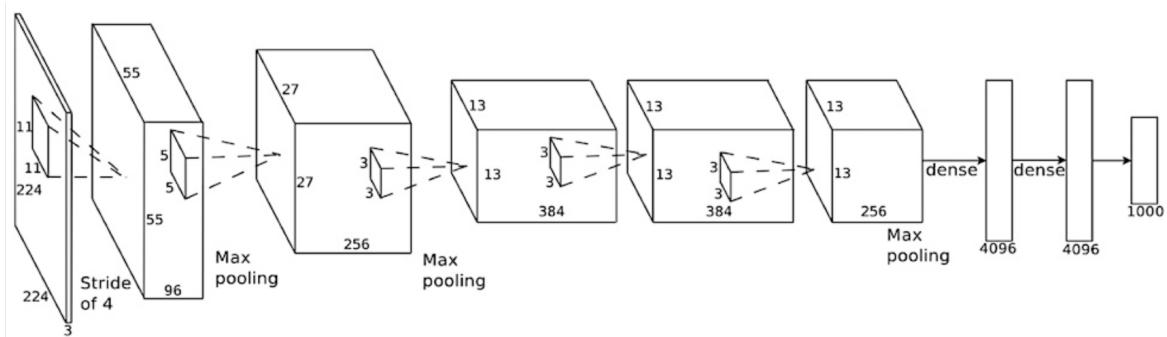
- 5-1 AlexNet (初期のころのCNN)

 - データセット—IImageNetをとくためのCNN

 - ImageNet-Mechanical Turkを用いて手動で1000個のクラスにラベル付けされた1400万枚以上の画像を含む大規模なデータセット

 - ILSVRCで2012年にトロント大学がAlexNetを用いてこの大会に他チームと40%以上の大差をつけて優勝した際に用いたCNN
 - モデルの構造

 - 5層の畳み込み層とプーリング層など、それに続く3層の全結合層から構成



- 【参考】 AlexNetの論文

- ImageNet Classification with Deep Convolutional Neural Networks
- 2012年
- URL : <https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf>

終了テストについて

- 終了テストを合格してから、深層学習Day1-2のレポートをまとめていくつか気付いた点についてまとめました。
 - テスト内容と、動画解説が一致していない。
深層学習のテストだと思って受けてみたが、機械学習、特に動画で解説していないSVMの出題が多くあった。
自己学習が必要とあったが、機械学習動画は後半がかなり駆け足だったため、調べて学習するのが時間を要した。
 - テスト後に改めてレポートをまとめていて気付いたこと
前回と同様、設問の意味が分からぬるものもあったが、テスト合格後に改めて動画と講義資料を確認したら、かなりの設問事項について、きちんと記載しており、確認が足りていないことを痛感した。
 - 検索してもわからない問題がある
やはりトレース問題が自信をもって回答できないと胃があった。問題集を購入し、繰り返し学習が必要を感じた。いまのところは克服できた。