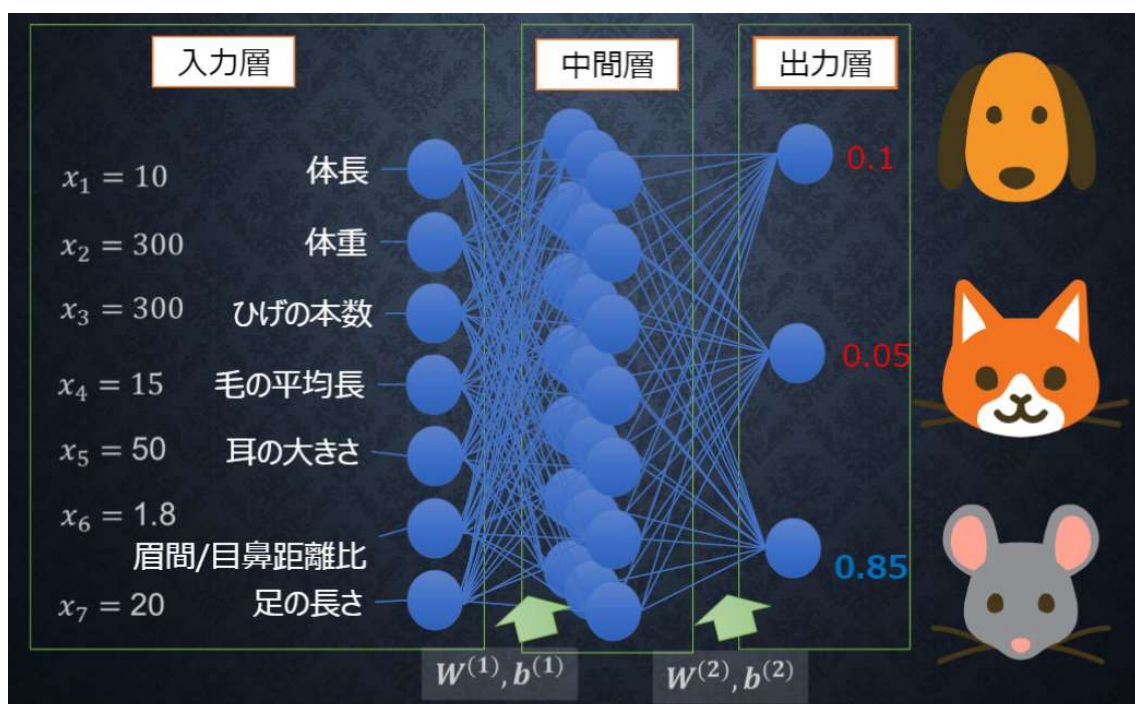


【E 資格学習レポート】深層学習前半(day 1、day2)レポート

深層学習前半(day1)については以下 5 つの科目でレポートする

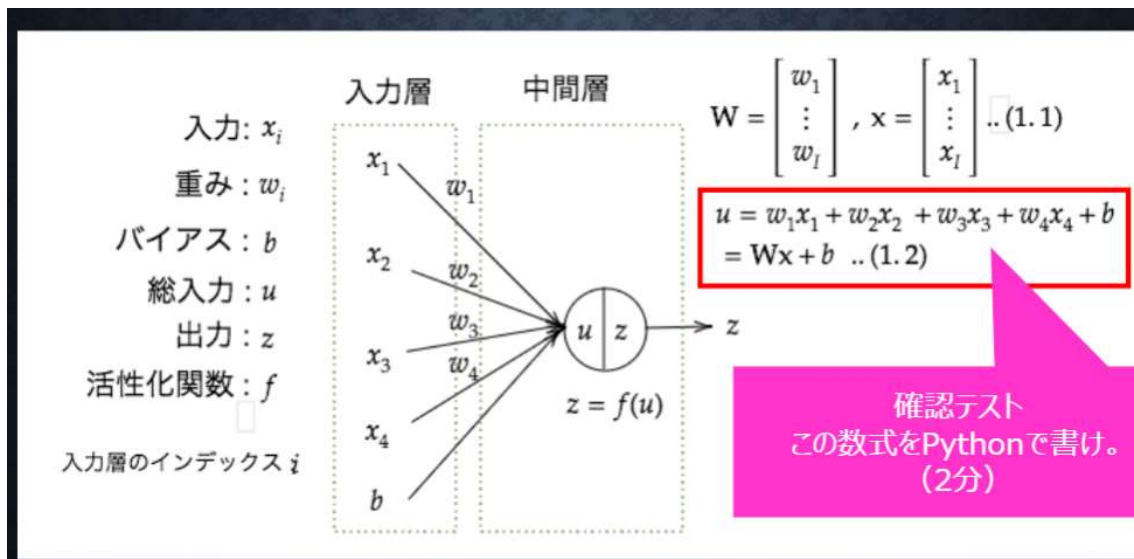
ニューラルネットワークの全体像



Section 1 : 入力層～中間層

- ・「識別モデルと生成モデル」
- ・識別 (discriminative, backward) (データ→クラス)
- ・生成 (generative, forward) (クラス→データ) (犬という情報から犬らしい画像を生成する事)
- ・機械学習・深層学習モデルのそれぞれで識別・生成を行う手法が存在する。
- ・生成モデルは「データのクラス条件付き密度」まで学習する (「データがクラスに属する確率」にとどまる事なく))
- ・識別モデルによる識別 (確率的識別モデル⇔決定的識別モデル)
- ・推論 (入力データ⇒事後確率)

- ・ 決定（事後確率⇒識別結果）
- ・ 間違いの程度を測る事ができる。
- ・ 推論結果の取り扱いを決められる（棄却など）
- ・ 万能近似定理と深さ
- ・ 「ニューラルネットワーク（全体像）」
- ・ 識別モデル：変換器である。できる事：回帰、分類
- ・ 深層ニューラルネットワークモデル：4つ以上中間層を持つ。
- ・ 入力が数字であり、出力が数字であれば何でも学習できる。
- ・ 「入力層から中間層」
 - w は傾きを変える事ができる。
 - b は切片を変える事ができる。
- 傾きと切片が決まれば、1 次関数の式は一意に決定される。
- ・ 数式コード



```
u1 = np.dot(x, W1) + b1
```

実行結果：

- ・ 順伝播（単層・単ユニット）

```
*** 重み ***  
[[0.1]  
 [0.2]]  
shape: (2, 1)
```

```
*** バイアス ***  
0.5  
shape: ()
```

```
*** 入力 ***  
[2 3]  
shape: (2,)
```

```
*** 総入力 ***  
[1.3]  
shape: (1,)
```

・順伝播（単層・複数ユニット）

```
[33] *** 重み ***  
[[0.1 0.2 0.3 0. ]  
 [0.2 0.3 0.4 0.5]  
 [0.3 0.4 0.5 1. ]]  
shape: (3, 4)
```

```
*** バイアス ***  
[0.1 0.2 0.3]  
shape: (3,)
```

```
*** 入力 ***  
[ 1.  5.  2. -1.]  
shape: (4,)
```

```
*** 総入力 ***  
[1.8 2.2 2.6]  
shape: (3,)
```

```
*** 中間層出力 ***  
[0.85814894 0.90024951 0.93086158]  
shape: (3,)
```

・順伝播 (3層・複数ユニット)

```

*** 層み2 ***
[[0.91438857 0.19018789 0.81320185 0.58847771 0.73350378]
 [0.97382203 0.03090181 0.42082118 0.90784218 0.93341542]
 [0.87082874 0.22200934 0.15501838 0.53370853 0.11305811]
 [0.50524041 0.85719982 0.85432811 0.28778534 0.1185785 ]
 [0.75740213 0.09110927 0.0358131 0.59394884 0.808989 ]
 [0.49272341 0.18532451 0.01745725 0.87779507 0.84814888]
 [0.88048492 0.581081 0.24395482 0.89217789 0.34755144]
 [0.37814882 0.20278853 0.39245007 0.15743877 0.91098018]
 [0.20548572 0.71985182 0.45498145 0.84424087 0.13887473]
 [0.88908757 0.75770985 0.21078394 0.13927977 0.98324952]]
shape: (10, 5)

*** 層み3 ***
[[0.43070477 0.34379179 0.41177549 0.81321178]
 [0.34033892 0.8888228 0.47352021 0.9045348 ]
 [0.9878571 0.75588845 0.05513157 0.54944437]
 [0.87002492 0.29738834 0.78291442 0.20795981]
 [0.13791303 0.35119298 0.8177744 0.5990705 ]]
shape: (5, 4)

*** バイアス1 ***
[0.45953432 0.49552451 0.91102513 0.17245202 0.94840338 0.98859842
 0.32192385 0.34940418 0.35310089 0.51743309]
shape: (10, )

*** バイアス2 ***
[0.84085453 0.81745884 0.85002747 0.4402298 0.10428327]
shape: (5, )

*** バイアス3 ***
[0.49583504 0.55212873 0.30311988 0.81934792]
shape: (4, )

##### 順伝播開始 #####
*** 総入力1 ***
[5.88154041 4.15287847 3.08981075 5.83770359 2.82289088 3.17482502
 4.83748815 5.04828084 2.43887352 2.80851822]
shape: (10, )

*** 中間層出力1 ***
[5.88154041 4.15287847 3.08981075 5.83770359 2.82289088 3.17482502
 4.83748815 5.04828084 2.43887352 2.80851822]
shape: (10, )

*** 中間層出力2 ***
[28.29153882 15.93259388 15.21549378 21.58715088 21.98538103]
shape: (5, )

*** 総入力2 ***
[28.29153882 15.93259388 15.21549378 21.58715088 21.98538103]
shape: (5, )

```

・多クラス分類 (2-3-4 ネットワーク)

```

*** 重み1 ***
[[0.08830593 0.18588431 0.5319753 0.31890437 0.34815288 0.78990159
 0.83378058 0.98715475 0.12727925 0.52119841 0.5062853 0.92278588
 0.20388411 0.72282204 0.17743718 0.15938189 0.54973895 0.82955048
 0.3951821 0.30583412 0.27831807 0.54078592 0.21180458 0.02548397
 0.10004599 0.12818089 0.24884079 0.82885788 0.23454188 0.59480318
 0.84455241 0.07743105 0.93227927 0.84790945 0.50201792 0.18638824
 0.29395595 0.2947148 0.97382929 0.78440399 0.88580312 0.78748097
 0.80128813 0.24733214 0.28757821 0.92993999 0.13578801 0.31155482
 0.23941831 0.87127883]
[0.48815051 0.15224278 0.9808835 0.89790453 0.5315328 0.27983859
 0.18808801 0.25455225 0.74880598 0.11357359 0.23882138 0.48883923
 0.15181184 0.82123312 0.24902873 0.78338783 0.2793515 0.83012203
 0.08002489 0.80154273 0.78800378 0.84941308 0.87588889 0.98044889
 0.04037974 0.98881829 0.83944588 0.50830833 0.83143955 0.57350351
 0.08748135 0.84512794 0.21355798 0.39238888 0.08534429 0.40859577
 0.27831388 0.73355885 0.28419497 0.35582994 0.05784859 0.83304989
 0.72508133 0.29385434 0.88705315 0.95252438 0.92742903 0.10853448
 0.13721948 0.53834848]
[0.8488983 0.14825282 0.09742702 0.58849199 0.07935078 0.58329292
 0.20282482 0.13883872 0.87505208 0.83520425 0.40545408 0.3991458
 0.71137552 0.19809887 0.27808818 0.01119219 0.38730518 0.80013052
 0.50182787 0.31708881 0.81382414 0.31880048 0.27253882 0.40743253
 0.21858811 0.58252513 0.8118887 0.58838558 0.59383788 0.27783874
 0.85455885 0.42877083 0.0041589 0.09297871 0.83583117 0.99402135
 0.0720077 0.92938882 0.03857253 0.31594157 0.4839885 0.30818838
 0.85587851 0.39244988 0.91409428 0.88889852 0.949849 0.9905848
 0.78751971 0.32910974]]
shape: (3, 50)

```

```

*** 重み2 ***
[[0.95785528 0.38378249 0.92002845 0.49118912 0.55792877 0.85088202]
[0.82025843 0.78970187 0.11232128 0.41493128 0.8229888 0.28025421]
[0.83835802 0.73814459 0.15787847 0.03139311 0.38848772 0.9304958 ]
[0.15588357 0.52985492 0.09174977 0.14471428 0.29097581 0.04987142]
[0.73744358 0.13942975 0.14895512 0.42808582 0.25182878 0.82719012]
[0.39274885 0.82558537 0.13058021 0.08894888 0.09143905 0.8179478 ]
[0.84890219 0.80743454 0.25737578 0.40981143 0.8871944 0.10159294]
[0.01514534 0.54159125 0.78295411 0.87371728 0.34545888 0.5852394 ]
[0.39923994 0.70958782 0.72152107 0.82837078 0.44893585 0.89542289]
[0.52791317 0.13301382 0.40088129 0.90594819 0.93415531 0.99437448]
[0.22424028 0.51132422 0.85444185 0.07208582 0.21598382 0.18480778]
[0.15781108 0.42948891 0.18554423 0.84810888 0.05838033 0.21219579]
[0.384847 0.78398772 0.08709182 0.42157728 0.15999893 0.04751528]
[0.93509487 0.0810033 0.2092554 0.35455431 0.88200718 0.04701535]
[0.71548859 0.34028551 0.44359504 0.41775532 0.89582291 0.48789072]
[0.84813018 0.50883244 0.29270552 0.15912285 0.48357547 0.90285558]
[0.59843238 0.93404402 0.40197225 0.54137887 0.79917858 0.07812559]
[0.28482712 0.92090044 0.01883978 0.45427544 0.50183881 0.88983717]
[0.50485537 0.9438029 0.40749841 0.52028229 0.75027588 0.20293898]
[0.5871597 0.19828381 0.07450581 0.30289222 0.44588084 0.1383241 ]
[0.02801992 0.41957909 0.49928515 0.21503382 0.57543834 0.5778898 ]
[0.3141114 0.97985875 0.84835328 0.37983985 0.88011828 0.99070488]
[0.82427927 0.88194358 0.38822831 0.82745549 0.72588701 0.8908445 ]
[0.95955488 0.738328 0.91357418 0.04353999 0.24525207 0.45352223]
[0.40222838 0.38309808 0.14107741 0.75340111 0.0187129 0.098087 ]
[0.70788587 0.35957895 0.51780578 0.85589522 0.58924487 0.21854787]
[0.78550418 0.87981841 0.80539557 0.24845915 0.84949808 0.44282818]
[0.91855548 0.784287 0.82354921 0.47154209 0.8949584 0.48747855]
[0.28838448 0.92588823 0.35878248 0.47179335 0.83350272 0.12873438]
[0.23859353 0.28323595 0.00422121 0.78121084 0.81008293 0.52958723]
[0.28782059 0.87498583 0.97412413 0.25925222 0.04135848 0.14400324]

```

・ 回帰 (2-3-2 ネットワーク)

ネットワークの初期化

*** 重み1 ***

```
[[2.38375179e-01 1.16467337e-01 8.11442993e-01 8.00558295e-01
 5.78792322e-01 8.91439584e-01 1.21527472e-01 3.28151001e-02
 7.97248403e-01 4.01376340e-01 7.36422553e-01 8.16048811e-01
 5.45917026e-01 2.95781013e-01 6.92331895e-01 6.81567852e-01
 3.92716909e-01 9.28920029e-01 8.64840526e-01 8.90428418e-01
 7.93967796e-01 2.40407765e-01 3.20554572e-01 2.11427425e-01
 3.01076334e-01 9.05898057e-01 4.75353936e-01 4.85632441e-01
 1.67064435e-01 2.83249656e-01 2.64419802e-01 7.58972440e-01
 6.10980621e-01 9.72417961e-01 7.16913388e-02 2.56844295e-01
 4.16928176e-01 7.41635364e-01 4.43201548e-01 8.32768487e-02
 9.05732138e-01 9.76180565e-01 7.74620596e-01 2.20381391e-02
 6.97876339e-01 3.31858050e-01 8.93123119e-01 2.13060411e-01
 2.99886435e-01 5.94064181e-01]
[6.52378624e-01 2.90348362e-01 2.98128463e-01 6.38994867e-01
 5.95735580e-01 6.27820081e-01 5.29604544e-01 3.09971804e-01
 2.51083402e-02 2.55306674e-01 1.91095157e-01 1.46742691e-01
 8.10500953e-01 7.08998752e-01 4.13410970e-01 3.41204903e-01]
```

・2 値分類 (2-3-1 ネットワーク)

ネットワークの初期化

順伝播開始

*** 入力1 ***

```
[2.04781037 4.9489324 7.92722718 2.38886252 5.11038348 7.09578187
 2.37770277 5.07352012 7.18107183 1.82488999]
shape: (10,)
```

*** 中間層出力1 ***

```
[2.04781037 4.9489324 7.92722718 2.38886252 5.11038348 7.09578187
 2.37770277 5.07352012 7.18107183 1.82488999]
shape: (10,)
```

*** 入力2 ***

```
[26.35895934 31.8938422 20.78591108 20.04743543 28.08478898 27.82139834
 29.91331888 28.37285781 13.8880387 28.37048018 20.55454781 28.87491444
 11.75794057 29.84897009 28.00383928 22.39095087 28.39518891 30.95804812
 28.05895312 21.57188939]
shape: (20,)
```

*** 出力1 ***

```
[1.]
shape: (1,)
```

出力合計: 1.0

結果表示

*** 中間層出力 ***

```
[2.04781037 4.9489324 7.92722718 2.38886252 5.11038348 7.09578187
 2.37770277 5.07352012 7.18107183 1.82488999]
shape: (10,)
```

*** 出力 ***

```
[1.]
```

参照コード:

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

import sys

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

```

import numpy as np
from common import functions

def print_vec(text, vec):
    print("*** " + text + " ***")
    print(vec)
    print("shape: " + str(vec.shape))
    print("")
# 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 = np.array(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)

# 順伝播(単層・複数ユニット)

# 重み
W = np.array([
    [0.1, 0.2, 0.3, 0],
    [0.2, 0.3, 0.4, 0.5],
    [0.3, 0.4, 0.5, 1],
])

## 試してみよう_配列の初期化
#W = np.zeros((4,3))
#W = np.ones((4,3))
#W = np.random.rand(4,3)
#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(W, x) + b
print_vec("総入力", u)

```



```

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

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

    input_layer_size = 3
    hidden_layer_size_1=10
    hidden_layer_size_2=5
    output_layer_size = 4

    #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.random.rand(input_layer_size, hidden_layer_size_1)
    network['W2'] = np.random.rand(hidden_layer_size_1,hidden_layer_size_2)
    network['W3'] = np.random.rand(hidden_layer_size_2,output_layer_size)

    network['b1'] = np.random.rand(hidden_layer_size_1)
    network['b2'] = np.random.rand(hidden_layer_size_2)
    network['b3'] = np.random.rand(output_layer_size)

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

```

```

print_vec("バイアス 3", 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

    # 2 層の総出力
    z2 = functions.relu(u2)

    # 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("中間層出力 2", z2)
    print_vec("総入力 2", u2)
    print_vec("出力", y)

```

```

        print("出力合計: " + str(np.sum(y)))

    return y, z1, z2

# 入力値
x = np.array([1., 2., 4.])
print_vec("入力", x)

# ネットワークの初期化
network = init_network()

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

# 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 3-5-6 に変更してみよう

# 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 #####")

    #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 = {}

    input_layer_size = 3
    hidden_layer_size=50
    output_layer_size = 6

    #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.random.rand(input_layer_size, hidden_layer_size)
network['W2'] = np.random.rand(hidden_layer_size, output_layer_size)

network['b1'] = np.random.rand(hidden_layer_size)
network['b2'] = np.random.rand(output_layer_size)

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

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, 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("交差エントロピー誤差", loss)

# 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 #####")

    input_layer_size = 3
    hidden_layer_size=50
    output_layer_size = 2

    #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.random.rand(input_layer_size, hidden_layer_size)
    network['W2'] = np.random.rand(hidden_layer_size, output_layer_size)

    network['b1'] = np.random.rand(hidden_layer_size)
    network['b2'] = np.random.rand(output_layer_size)

    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(y)))

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.])
# 誤差
loss = functions.mean_squared_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("二乗誤差", loss)

# 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 5-10-20-1 に変更してみよう

# 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, 0.1, 0.3, 0.5, 0.1, 0.3, 0.5, 0.1],
    [0.1, 0.3, 0.5, 0.1, 0.3, 0.5, 0.1, 0.3, 0.5, 0.1],
    [0.1, 0.3, 0.5, 0.1, 0.3, 0.5, 0.1, 0.3, 0.5, 0.1],
    [0.1, 0.3, 0.5, 0.1, 0.3, 0.5, 0.1, 0.3, 0.5, 0.1],
    [0.1, 0.3, 0.5, 0.1, 0.3, 0.5, 0.1, 0.3, 0.5, 0.1]
])
network['W2'] = np.random.rand(10, 20)
network['W3'] = np.random.rand(20, 1)

network['b1'] = np.random.rand(10)
network['b2'] = np.random.rand(20)
network['b3'] = 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, W3 = network['W1'], network['W2'], network['W3']
    b1, b2, b3 = network['b1'], network['b2'], network['b3']

    # 隠れ層の総入力
    u1 = np.dot(x, W1) + b1
    # 隠れ層 1 の総出力
    z1 = functions.relu(u1)
    # 隠れ層 2 層への総入力
    u2 = np.dot(z1, W2) + b2
    # 隠れ層 2 の出力
    z2 = functions.relu(u2)

    u3 = np.dot(z2, W3) + b3

```



```

    z3 = functions.sigmoid(u3)
    y = z3
    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

# 入力値
x = np.array([1., 2., 2., 4., 5.])

# 目標出力
d = np.array([1])
network = init_network()
y, z1 = forward(network, x)

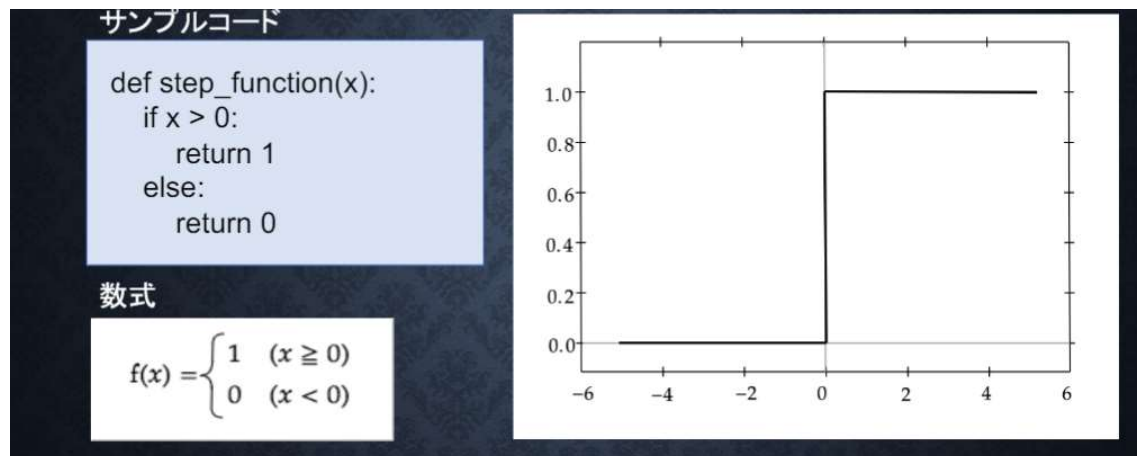
# 誤差
loss = functions.cross_entropy_error(d, y)

## 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("交差エントロピー誤差", loss)

```

Section 2 : 活性化関数

- ・ 活性化関数は非線形である事が大切。
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。
入力値の値によって、次の層への信号の ON/OFF や強弱を定める働きをもつ。
- ・ 線形な関数は加法性 (additivity) と斉次性 (homogeneity) を満たす。
- ・ 非線形な関数はそれらを満たさない。
- ・ ステップ関数。課題：0 - 1 の間を表現できず、線形分離可能なものしか学習できなかった。
- ・ シグモイド関数。課題：勾配消失問題。
- ・ RELU 関数。勾配消失問題の回避とスパース化 (モデルの中身がシンプルになる) に貢献する。
- ・ 活性化関数: ステップ関数



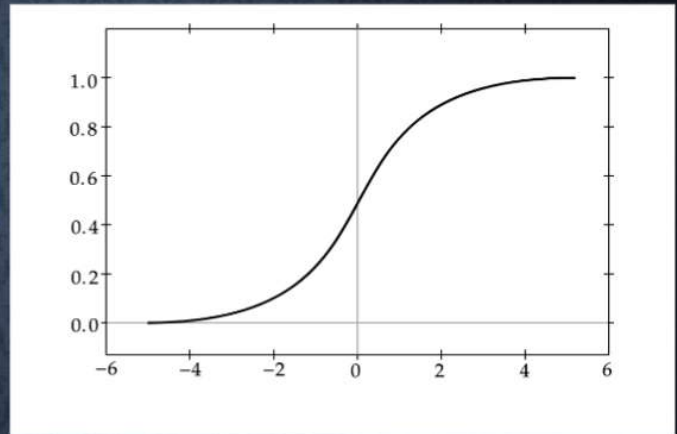
- ・ しきい値を超えたら発火する関数であり、出力は常に 1 か 0。パーセプトロン (ニューラルネットワークの前身) で利用された関数。課題 0 -1 間の間を表現できず、線形分離可能なものしか学習できなかった
- ・ 活性化関数: シグモイド関数

サンプルコード

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

数式

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



- ・ 0 ~ 1 の間を緩やかに変化する関数で、ステップ関数では ON/OFF しかない状態に対し、信号の強弱を伝えられるようになり、予想ニューラルネットワーク普及のきっかけとなった。課題：大きな値では出力の変化が微小なため、勾配消失問題を引き起こす事があった。

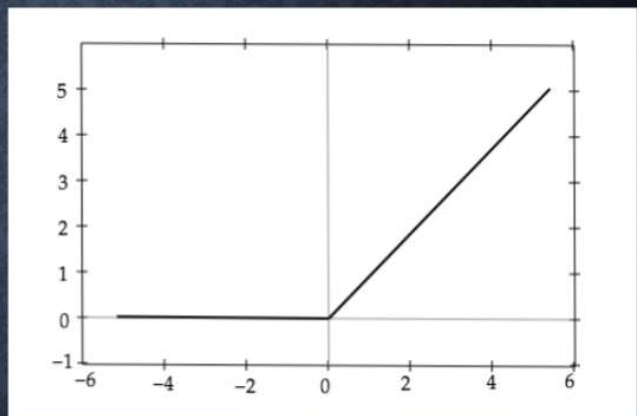
- ・ 活性化関数: RELU 関数

サンプルコード

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

数式

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



- ・ 今最も使われている活性化関数勾配消失問題の回避とスパース化に貢献することで良い成果をもたらしている。

実行結果：

```

##### ネットワークの初期化 #####
*** 重み1 ***
[[0.1 0.3 0.5]
 [0.2 0.4 0.6]]

*** 重み2 ***
[[0.1 0.4]
 [0.2 0.5]
 [0.3 0.6]]

*** 重み3 ***
[[0.1 0.3]
 [0.2 0.4]]

*** バイアス1 ***
[0.1 0.2 0.3]

*** バイアス2 ***
[0.1 0.2]

*** バイアス3 ***
[1 2]

##### 順伝播開始 #####
*** 総入力1 ***
[0.6 1.3 2. ]

*** 中間層出力1 ***
[0.6 1.3 2. ]

*** 総入力2 ***
[1.02 2.29]

*** 出力1 ***
[0.6 1.3 2. ]

```

参照コード：

活性化関数: RELU 関数、シグモイド関数、

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

# 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

    # 2 層の総出力
    z2 = functions.relu(u2)

    # 出力層の総入力
    u3 = np.dot(z2, W3) + b3

```

```
# 出力層の総出力
y = u3

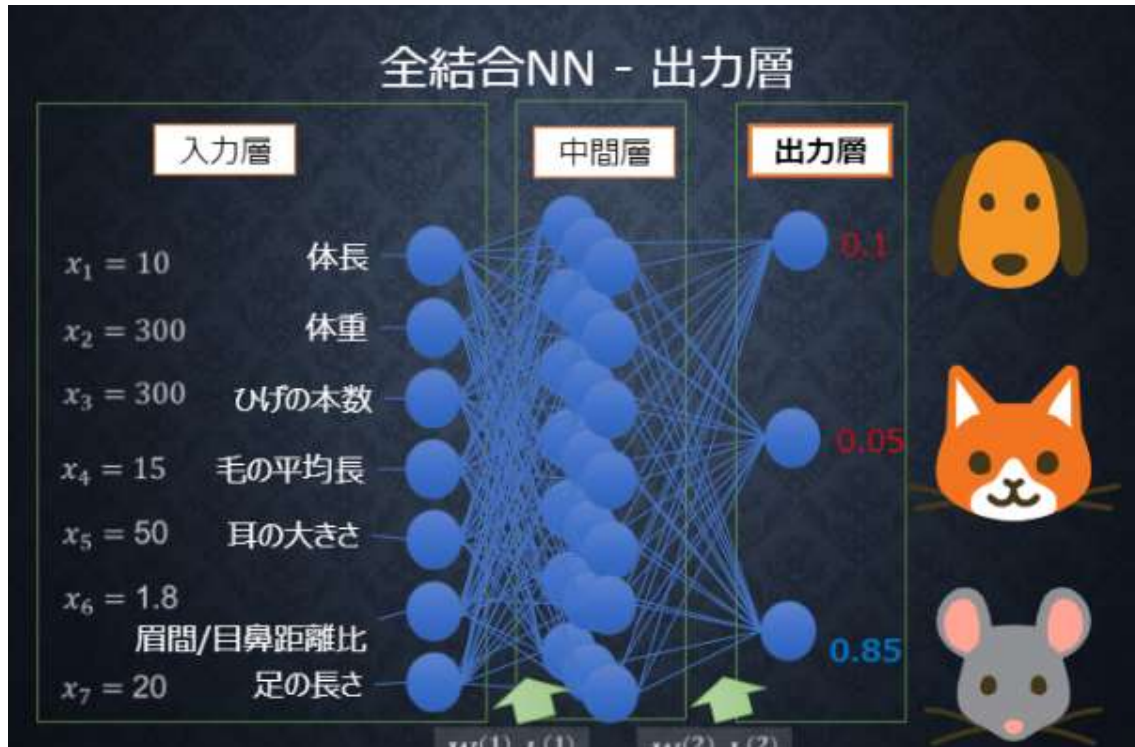
print_vec("総入力 1", u1)
print_vec("中間層出力 1", z1)
print_vec("総入力 2", u2)
print_vec("出力 1", z1)
print("出力合計: " + str(np.sum(z1)))

return y, z1, z2
```

Section 3：出力層

3-1 誤差関数

- ・全結合 NN-出力層のイメージ図



- ・中間層は次の層の入力として適切なものを出力する。
- ・出力層の出力は目的に合致したものである必要がある（例えば、各クラスの「確率」）。
- ・「誤差関数」はそれぞれのニューラルネットワークの出力結果と正解データを比較する事によって「どのくらいあっていったか」を見る。
- ・クロスエントロピー誤差コードは `loss = cross_entropy_error(d, y)`
- ・平均二乗誤差コードは `loss = functions.mean_squared_error(d, y)`

3-2 出力層の活性化関数

- ・「活性化関数」は出力層と中間層のものとは目的が違う。
 - 中間層：しきい値の前後で信号の強弱を調整
 - 出力層：信号の大きさ（比率）はそのままに変換
- 分類問題の場合、出力層の出力は0～1の範囲に限定し、総和を1とする必要がある
- ・恒等写像 回帰 二乗誤差
- ・シグモイド関数 二値分類 交差エントロピー
- ・ソフトマックス関数 多クラス分類 交差エントロピー
- ・交差エントロピーは分類問題の誤差関数としては非常に良く使用する。

・全結合 NN-出力層の種類

	回帰	二値分類	多クラス分類
活性化関数	恒等写像 $f(u) = u$	シグモイド関数 $f(u) = \frac{1}{1 + e^{-u}}$	ソフトマックス関数 $f(i, u) = \frac{e^{u_i}}{\sum_{k=1}^K e^{u_k}}$
誤差関数	二乗誤差	交差エントロピー	

実行結果：

ネットワークの初期化

*** 重み 1 ***

```
[[0.1 0.3 0.5]
 [0.2 0.4 0.6]]
```

*** 重み 2 ***

```
[[0.1 0.4]
 [0.2 0.5]
 [0.3 0.6]]
```

*** バイアス 1 ***

```
[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

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]
```

参照コード:

```
from google.colab import drive
drive.mount('/content/drive')
import sys
sys.path.append('/content/drive/My Drive/DNN_code')
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("")

# 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/du2", 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)

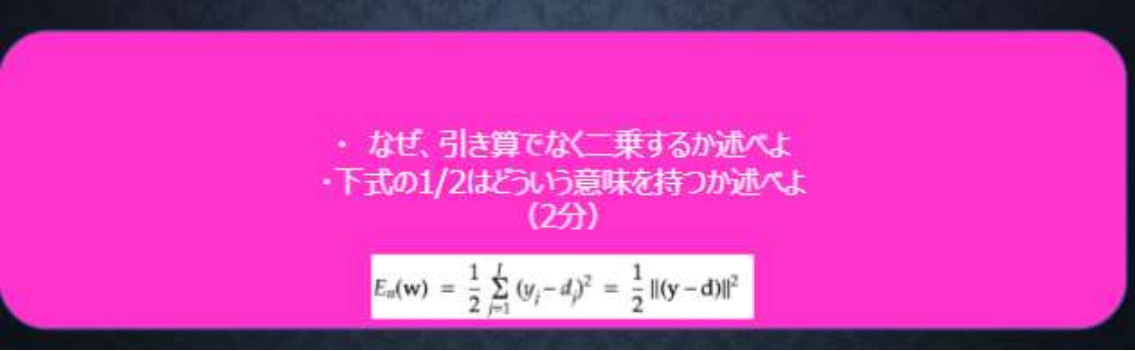
grad = backward(x, d, z1, y)
for key in ('W1', 'W2', 'b1', 'b2'):
    network[key] -= learning_rate * grad[key]

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'])

```

確認テスト 1



・なぜ、引き算でなく二乗するか述べよ
 ・下式の1/2はどのような意味を持つか述べよ
 (2分)

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

- ・全体でどれ位誤差があったかを知りたいとき、引き算したものの総和はゼロになる。各々を二乗したものを足し合わせる事によって、それを防ぐ事ができる。
- ・誤差逆伝播の計算で、誤差関数を微分する必要があるのだが、その際に 1/2 があると係数が相殺される計算式が簡単になるため。

確認テスト 2

ソフトマックス関数

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

①～③の数式に該当するソースコードを示し、一行ずつ処理の説明をせよ。
(5分)

```
def softmax(x):
    if x.ndim == 2:
        x = x.T
        x = x - np.max(x, axis=0)
        y = np.exp(x) / np.sum(np.exp(x), axis=0)
        return y.T
    x = x - np.max(x) # オーバーフロー対策
    return np.exp(x) / np.sum(np.exp(x))
```

本質の部分は $\text{np.exp}(x) / \text{np.sum}(\text{np.exp}(x))$

確認テスト 3

① $E_n(\mathbf{w}) = - \sum_{i=1}^I d_i \log y_i$ ② .. 交差エントロピー

①～②の数式に該当するソースコードを示し、一行ずつ処理の説明をせよ。
(5分)

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

本質の部分は $-\text{np.sum}(\text{np.log}(y[\text{np.arange}(\text{batch_size}), d] + 1e-7))$

- ・対数関数の性質上、0 に近づくと $-\infty$ に落ちてしまう。それを避けるために小さな値 ($1e-7$) を付与している。

Section 4 : 勾配降下法

・「勾配降下法」

ニューラルネットワークを学習させる手法の事。

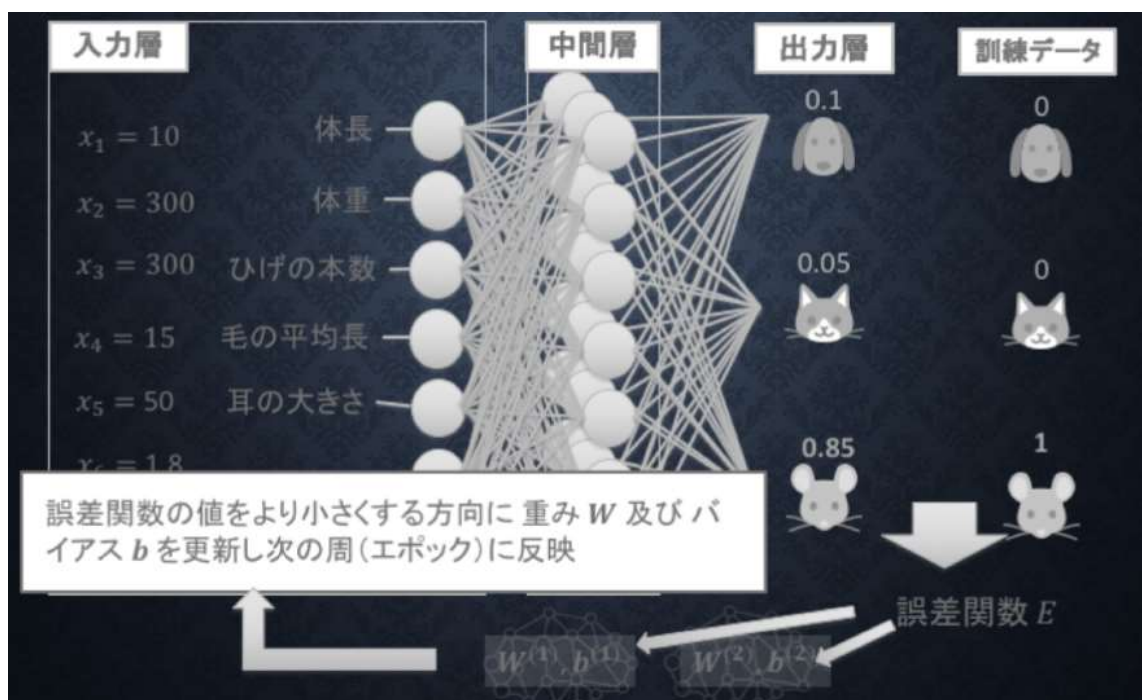
学習率 ε がどのように学習に影響するのか。

学習率が大きすぎた場合、最小値にいつまでもたどり着かず発散してしまう。

学習率が小さい場合発散することはないが、小さすぎると収束するまでに時間がかかってしまう。

エポック：重み更新一回のサイクルの事。

イメージ図



・「確率的勾配降下法 (SGD)」

望まない局所極小解に収束するリスクの軽減。

オンライン学習ができる (⇔バッチ学習)

データが冗長な場合の計算コストの軽減

イメージ図

【確率的勾配降下法】	【勾配降下法】
$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \varepsilon \nabla E_n$	$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \varepsilon \nabla E$
確率的勾配降下法	勾配降下法
ランダムに抽出したサンプルの誤差	全サンプルの平均誤差

・「ミニバッチ勾配降下法」

オンライン学習の特徴を上手くバッチ学習で利用できる様にした手法。

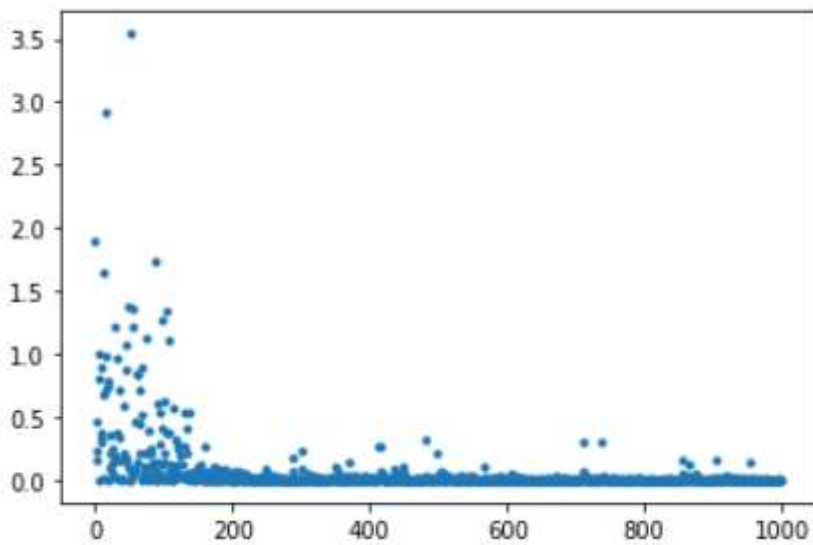
誤差は誤差の和をミニバッチの数で割ったもの。

CPU を利用したスレッドの並列化や GPU を利用した SIMD (Single Instruction Multiple Data) 並列化。

イメージ図

【ミニバッチ勾配降下法】	【確率的勾配降下法】
$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \varepsilon \nabla E_t$	$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \varepsilon \nabla E_n$
$E_t = \frac{1}{N_t} \sum_{n \in D_t} E_n$	
$N_t = D_t $	
ミニバッチ勾配降下法	確率的勾配降下法
ランダムに分割したデータの集合(ミニバッチ) D_t に属するサンプルの平均誤差	ランダムに抽出したサンプルの誤差

実行結果：



参照コード：

```
from google.colab import drive
drive.mount('/content/drive')
import sys
sys.path.append('/content/drive/My Drive/DNN_code')
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("")
# 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("\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_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 #####")
lists = range(epoch)

plt.plot(lists, losses, '.')
# グラフの表示
plt.show()

```

確認テスト1

【勾配降下法】

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \varepsilon \nabla E$$

$$\nabla E = \frac{\partial E}{\partial \mathbf{w}} = \left[\frac{\partial E}{\partial w_1} \dots \frac{\partial E}{\partial w_M} \right]$$

ε : 学習率

該当するソースコードを探してみよう。
(1分)

```

# パラメータに勾配適用
for key in ('w1', 'w2', 'b1', 'b2'):
    network[key] -= learning_rate * grad[key]

# 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 #####")
lists = range(epoch)
```

確認テスト 2

オンライン学習とは何か
2行でまとめよ（2分）

学習データが入ってくるたびに都度パラメータを更新し、学習を進めていく方法。一方、バッチ学習では一度にすべての学習データを使ってパラメータ更新を行う。

確認テスト 3

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

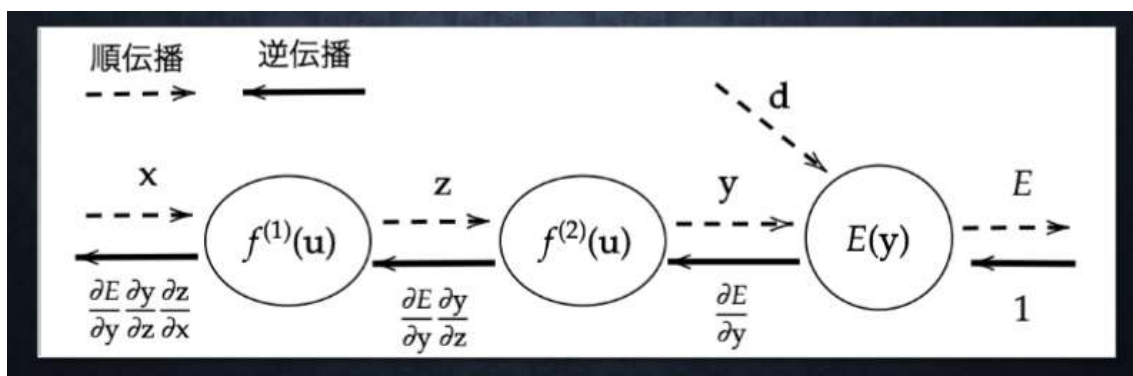
・この数式の意味を図に書いて説明せよ。
（3分）

$t \Rightarrow$ エポック $w \Rightarrow$ 重み

$\mathbf{W}_t \Rightarrow \mathbf{W}_{t+1} \Rightarrow \mathbf{W}_{t+2}$

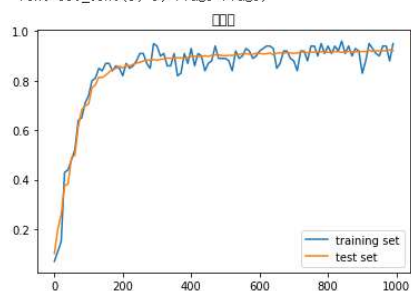
Section 5 : 誤差逆伝播法

- ・ 誤差勾配の計算
- ・ 数値微分⇒計算量が非常に大きくなる。
- ・ 誤差逆伝播法⇒算出される誤差を、出力側から順に微分し、前の層へと伝播。最小限の計算で各パラメータでの微分値を解析的に計算する方法。
- ・ 計算結果 (= 誤差) から微分を逆算することで、不要な再帰的計算を避けて微分を算出できる。
- ・ イメージ図



実行結果：

```
/usr/local/lib/python3.7/dist-packages/matplotlib/backends/backend_agg.py:214: RuntimeWarning: Glyph 27491 missing from current font.  
font.set_text(s, 0.0, flags=flags)  
/usr/local/lib/python3.7/dist-packages/matplotlib/backends/backend_agg.py:214: RuntimeWarning: Glyph 31572 missing from current font.  
font.set_text(s, 0.0, flags=flags)  
/usr/local/lib/python3.7/dist-packages/matplotlib/backends/backend_agg.py:214: RuntimeWarning: Glyph 29575 missing from current font.  
font.set_text(s, 0.0, flags=flags)  
/usr/local/lib/python3.7/dist-packages/matplotlib/backends/backend_agg.py:183: RuntimeWarning: Glyph 27491 missing from current font.  
font.set_text(s, 0, flags=flags)  
/usr/local/lib/python3.7/dist-packages/matplotlib/backends/backend_agg.py:183: RuntimeWarning: Glyph 31572 missing from current font.  
font.set_text(s, 0, flags=flags)  
/usr/local/lib/python3.7/dist-packages/matplotlib/backends/backend_agg.py:183: RuntimeWarning: Glyph 29575 missing from current font.  
font.set_text(s, 0, flags=flags)
```



参照コード：

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

```

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

import numpy as np
from data.mnist import load_mnist
import pickle
from common import functions
import matplotlib.pyplot as plt

# mnist をロード
(x_train, d_train), (x_test, d_test) = load_mnist(normalize=True, one_hot_label=True)
train_size = len(x_train)

print("データ読み込み完了")

# 重み初期値補正係数
wieght_init = 0.01 # 変更してみよう
#入力層サイズ
input_layer_size = 784 # 変更してみよう
#中間層サイズ
hidden_layer_size = 40 # 変更してみよう
#出力層サイズ
output_layer_size = 10 # 変更してみよう
# 繰り返し数
iters_num = 1000 # 変更してみよう
# ミニバッチサイズ
batch_size = 100 # 変更してみよう
# 学習率
learning_rate = 0.1 # 変更してみよう
# 描写頻度
plot_interval=10

# 初期設定
def init_network():
    network = {}
    network['W1'] = wieght_init * np.random.randn(input_layer_size,
hidden_layer_size)

```



```

    network['W2'] = wieght_init * np.random.randn(hidden_layer_size
, output_layer_size)

    # 試してみよう_Xavierの初期値
    # network['W1'] = np.random.randn(input_layer_size, hidden_laye
r_size) / np.sqrt(input_layer_size)
    # network['W2'] = np.random.randn(hidden_layer_size, output_lay
er_size) / np.sqrt(hidden_layer_size)

    # 試してみよう_Heの初期値
    # network['W1'] = np.random.randn(input_layer_size, hidden_laye
r_size) / np.sqrt(input_layer_size) * np.sqrt(2)
    # network['W2'] = np.random.randn(hidden_layer_size, output_lay
er_size) / np.sqrt(hidden_layer_size) * np.sqrt(2)

    network['b1'] = np.zeros(hidden_layer_size)
    network['b2'] = np.zeros(output_layer_size)

    return network

```

順伝播

```

def forward(network, x):
    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)

    return z1, y

```

誤差逆伝播

```

def backward(x, d, z1, y):
    grad = {}

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

```

```

# 出力層でのデルタ
delta2 = functions.d_softmax_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)
# 1 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)

return grad

# パラメータの初期化
network = init_network()

accuracies_train = []
accuracies_test = []

# 正答率
def accuracy(x, d):
    z1, y = forward(network, 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

for i in range(iters_num):
    # ランダムにバッチを取得
    batch_mask = np.random.choice(train_size, batch_size)
    # ミニバッチに対応する教師訓練画像データを取得
    x_batch = x_train[batch_mask]
    # ミニバッチに対応する訓練正解ラベルデータを取得する
    d_batch = d_train[batch_mask]

```

```

z1, y = forward(network, x_batch)
grad = backward(x_batch, d_batch, z1, y)

if (i+1)%plot_interval==0:
    accr_test = accuracy(x_test, d_test)
    accuracies_test.append(accr_test)

    accr_train = accuracy(x_batch, d_batch)
    accuracies_train.append(accr_train)

    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))

# パラメータに勾配適用
for key in ('W1', 'W2', 'b1', 'b2'):
    network[key] -= learning_rate * grad[key]

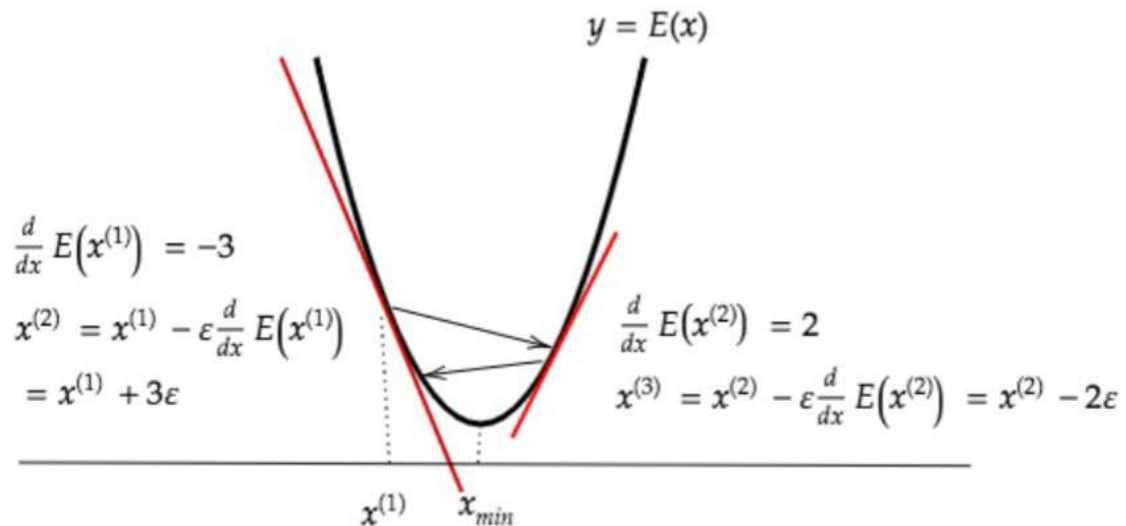
lists = range(0, iters_num, plot_interval)
plt.plot(lists, accuracies_train, label="training set")
plt.plot(lists, accuracies_test, label="test set")
plt.legend(loc="lower right")
plt.title("正答率")
# グラフの表示
plt.show()

```

深層学習前半(day2)については以下 5 つの科目でレポートする

Section 1 : 勾配消失問題

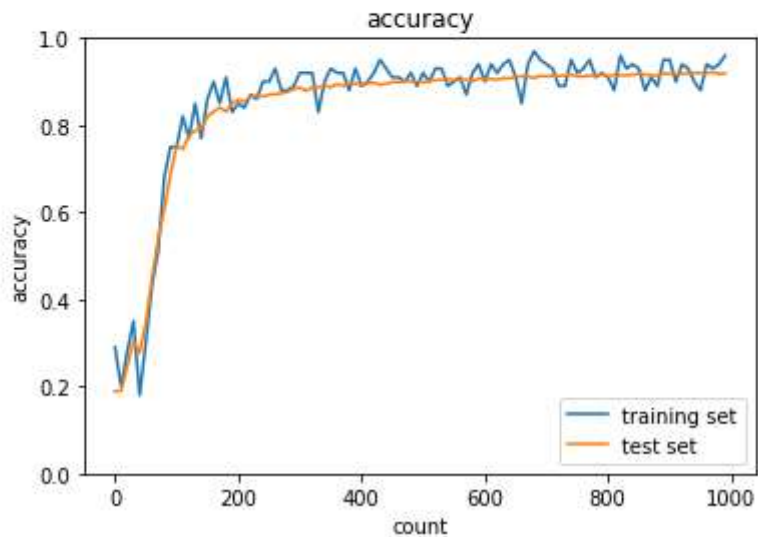
- ・勾配降下法のイメージ図



- ・勾配消失問題：誤差逆伝播法が下位層に進んでいくに連れて、勾配がどんどん緩やかになっていく。そのため、勾配降下法による、更新では下位層のパラメータはほとんど変わらず、訓練は最適値に収束しなくなる。
- ・勾配消失の解決方法
活性化関数の選択：ReLU 関数、勾配消失問題の回避とスパース化に貢献する事で良い結果をもたらしている。
重みの初期値設定
バッチ正規化
- ・バッチアルゴリズムとミニバッチアルゴリズム
深層学習では一般に要するデータが多く、メモリなどの都合ですべてまとめてバッチで計算することはできない。そのためデータを少数のまとまりであるミニバッチにして計算を行う。

実行結果：

Generation: 970. 正答率(トレーニング) = 0.94
 : 970. 正答率(テスト) = 0.9194
 Generation: 980. 正答率(トレーニング) = 0.93
 : 980. 正答率(テスト) = 0.9209
 Generation: 990. 正答率(トレーニング) = 0.94
 : 990. 正答率(テスト) = 0.9182
 Generation: 1000. 正答率(トレーニング) = 0.96
 : 1000. 正答率(テスト) = 0.9189



参照コード：

```

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

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

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

# ReLU layer
class Relu:
    def __init__(self):
        self.mask = None
  
```

```

def forward(self, x):
    # mask.shape = x.shape
    # True or False を要素として持つ
    self.mask = (x <= 0)
    out = x.copy()
    # True の箇所を 0 にする
    out[self.mask] = 0

    return out

def backward(self, dout):
    # True の箇所を 0 にする
    dout[self.mask] = 0
    dx = dout

    return dx

# Affine layer(全結合 layer)
class Affine:

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

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

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

        return out

    def backward(self, dout):

```

```

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

    return dx

class TwoLayerNet:
    '''
    input_size: 入力層のノード数
    hidden_size: 隠れ層のノード数
    output_size: 出力層のノード数
    weight_init_std: 重みの初期化方法
    '''

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

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

        self.lastLayer = layers.SoftmaxWithLoss()

    # 順伝播
    def predict(self, x):
        for layer in self.layers.values():

```

```

        x = layer.forward(x)

    return x

# 誤差
def loss(self, x, d):
    y = self.predict(x)
    return self.lastLayer.forward(y, d)

# 精度
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

# 勾配
def gradient(self, x, d):
    # forward

    self.loss(x, d)

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

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

    # 設定
    grad = {}

```



```

        grad['W1'], grad['b1'] = self.layers['Affine1'].dW, self.layers['Affine1'].db
        grad['W2'], grad['b2'] = self.layers['Affine2'].dW, self.layers['Affine2'].db

    return grad

# 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 = TwoLayerNet(input_size=784, hidden_size=40, 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

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', 'b1', 'b2'):
        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)

    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()

```

確認テスト1

連鎖律の原理を使い、 dz/dx を求めよ。
(2分)

$$z = t^2$$
$$t = x + y$$

$$dz/dx = dz/dt \cdot dt/dx = 2t \cdot 1 = 2(x+y)$$

確認テスト 2

シグモイド関数を微分した時、入力値が0の時に最大値をとる。その値として正しいものを選択肢から選べ。
(3分)

- (1) 0.15
- (2) 0.25
- (3) 0.35
- (4) 0.45

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

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

```
sigmoid(0)=(1/(1+1))=0.5
```

```
sigmoid_d(0)=(1-0.5)*0.5=0.25
```

回答 : (2)0.25

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。簡潔に説明せよ。
(1分)

重みを0で初期化すると正しい学習が行えない

確認テスト 4

一般的に考えられるバッチ正規化の効果を2点挙げよ。
(2分)

- ・ミニバッチ単位で、入力値のデータの偏りを抑制する
- ・過学習を押さえる事ができる

Section 2：学習率最適化手法

- ・学習率の値が大きい場合
最適値にいつまでもたどり着かず発散してしまう。
- ・学習率の値が小さい場合
発散することはないが、小さすぎると収束するまでに時間がかかってしまう。
大域局所最適値に収束しづらくなる
- ・学習率最適化手法
モメンタム：SDG のジグザグ運転に対して株価の移動平均のような動きをする。

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

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \epsilon \nabla E$$

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} + V_t$$

慣性: μ

モメンタム	勾配降下法
誤差をパラメータで微分したものと学習率の積を減算した後、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	誤差をパラメータで微分したものと学習率の積を減算する

モメンタムのメリット

- ・局所的最適解にはならず、大域的最適解となる。
- ・谷間についてから最も低い位置(最適値)にいくまでの時間が早い。

AdaGrad のようにハイパーパラメータの調整が必要な場合が少ない。学習率が徐々に小さくなるので、鞍点問題を引き起こす事があった

$$h_0 = \theta$$

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \epsilon \nabla E$$

$$h_t = h_{t-1} + (\nabla E)^2$$

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \epsilon \frac{1}{\sqrt{h_t + \theta}} \nabla E$$

AdaGrad	勾配降下法
誤差をパラメータで微分したものと再定義した学習率の積を減算する	誤差をパラメータで微分したものと学習率の積を減算する

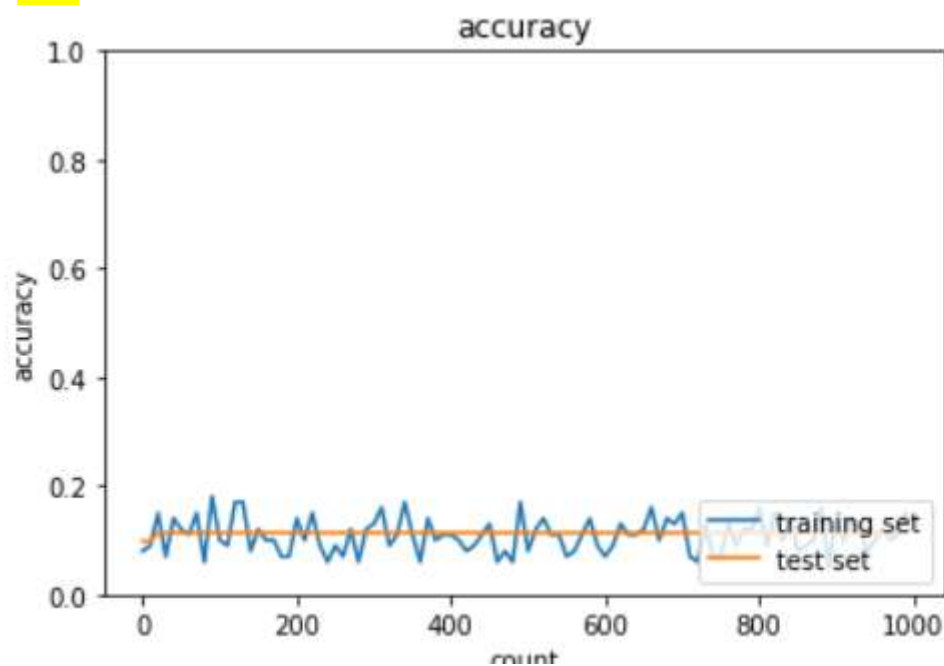
RMSProp (AdaGrad を改良)；鞍点問題をスムーズに解消できるようにした試みになる。

【RMSProp】	【勾配降下法】
$h_t = \alpha h_{t-1} + (1 - \alpha) (\nabla E)^2$ $\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \epsilon \frac{1}{\sqrt{h_t + \theta}} \nabla E$	$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \epsilon \nabla E$
RMSProp	勾配降下法
誤差をパラメータで微分したものと再定義した学習率の積を減算する	誤差をパラメータで微分したものと学習率の積を減算する

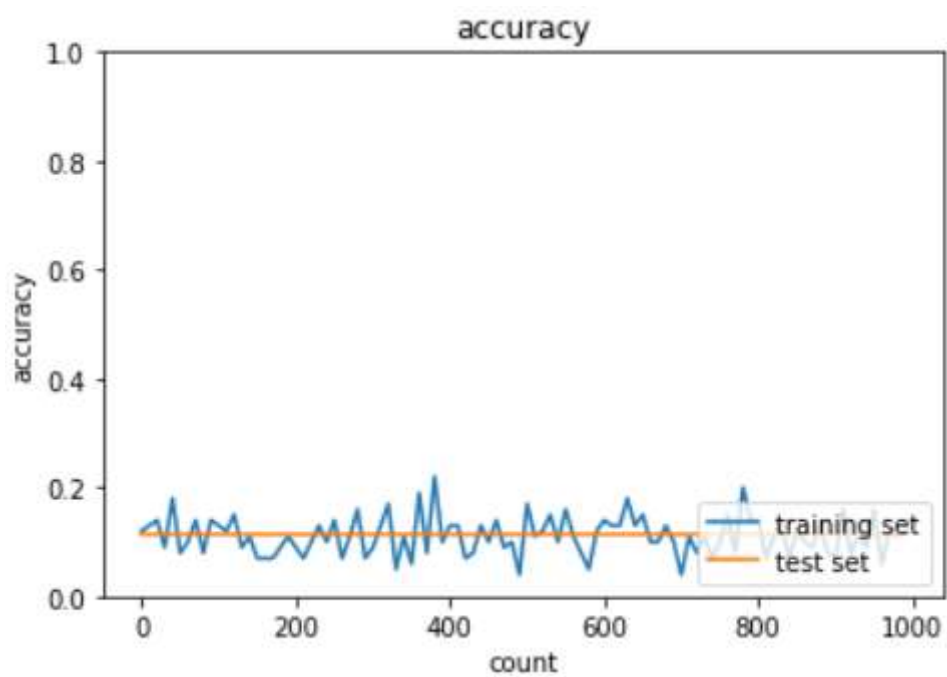
Adam：優秀な最適化手法（Optimizer）モメンタム及び RMSProp のメリットを学んだアルゴリズムである。

実行結果：

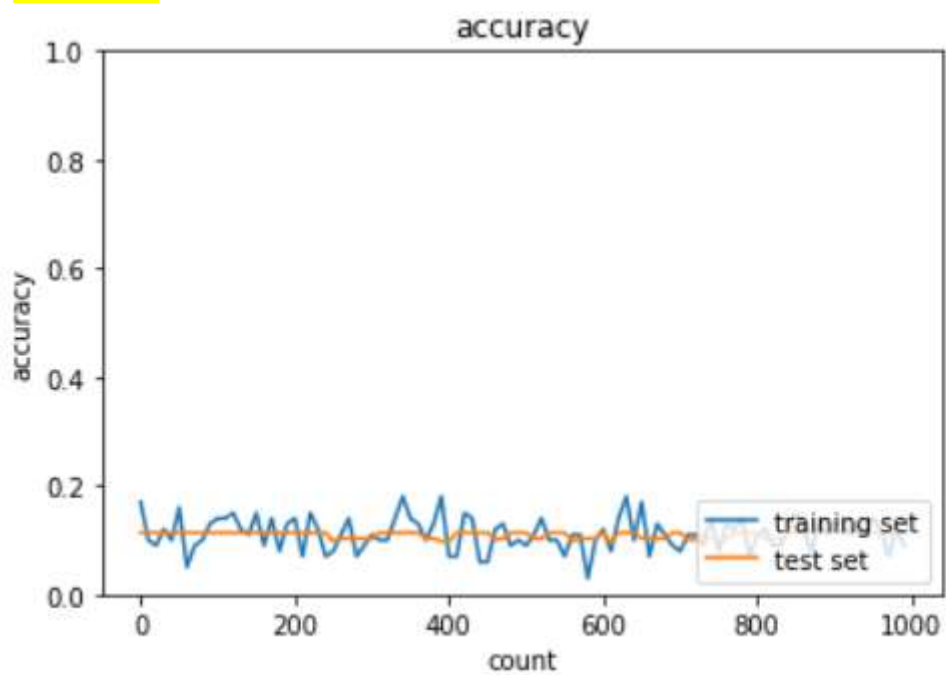
SDG



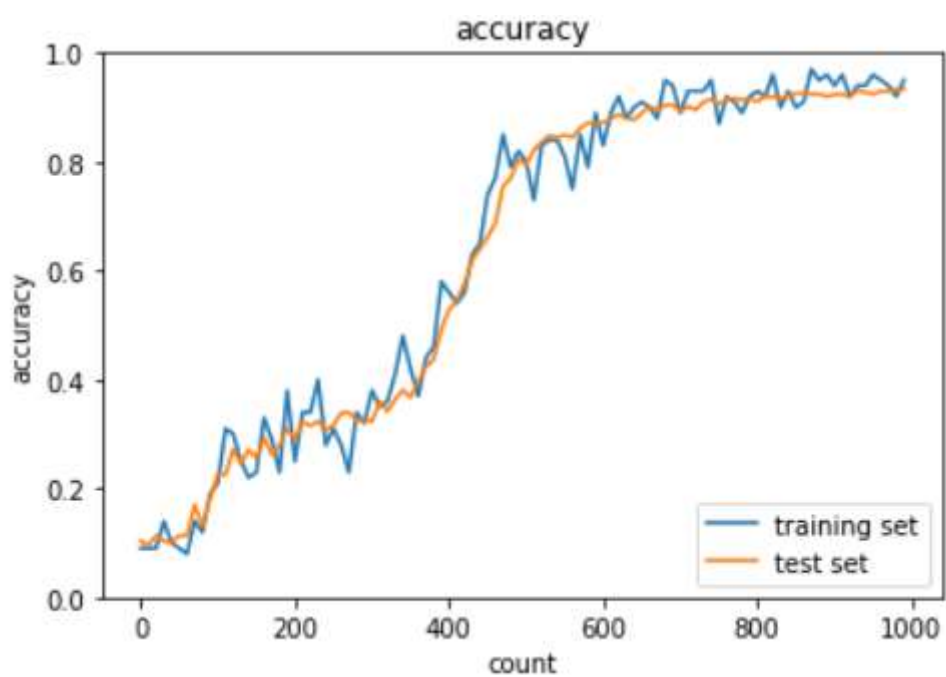
最適化（optimize）後



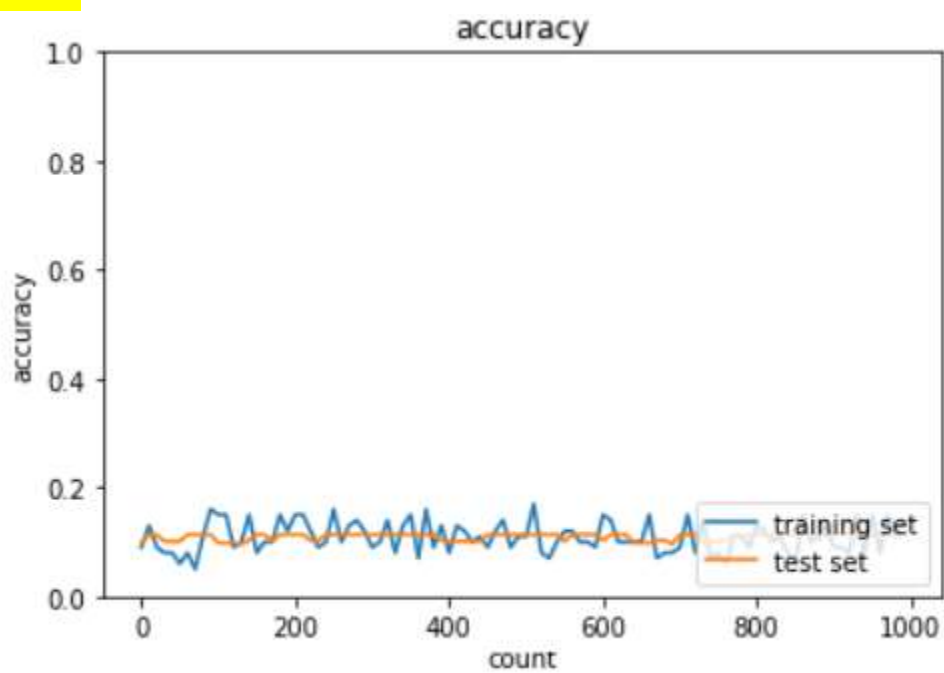
Momentum



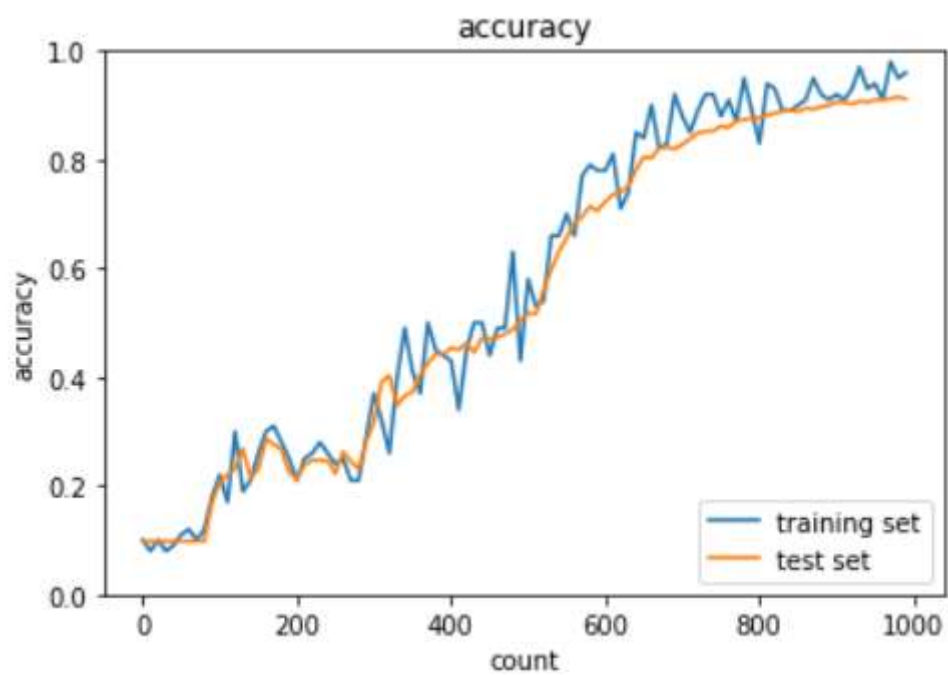
最適化 (optimize) 後



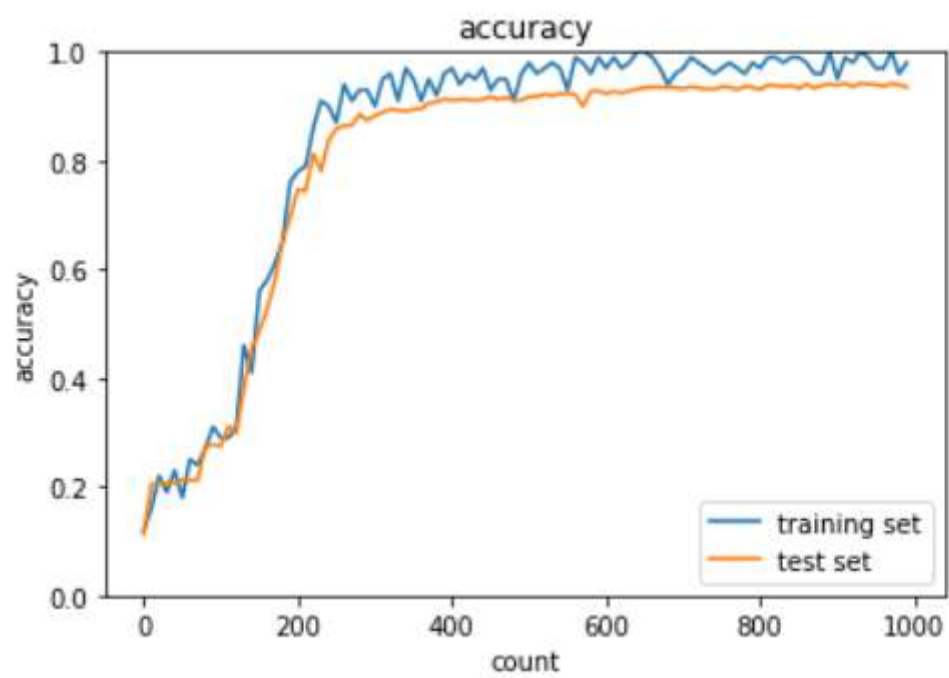
AdaGrad



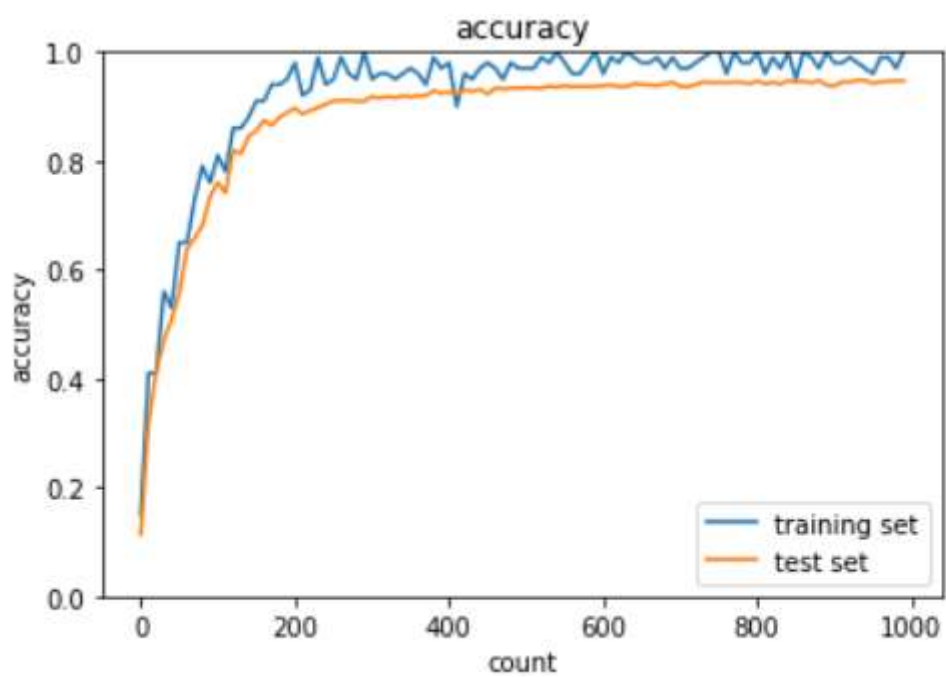
最適化 (optimize) 後



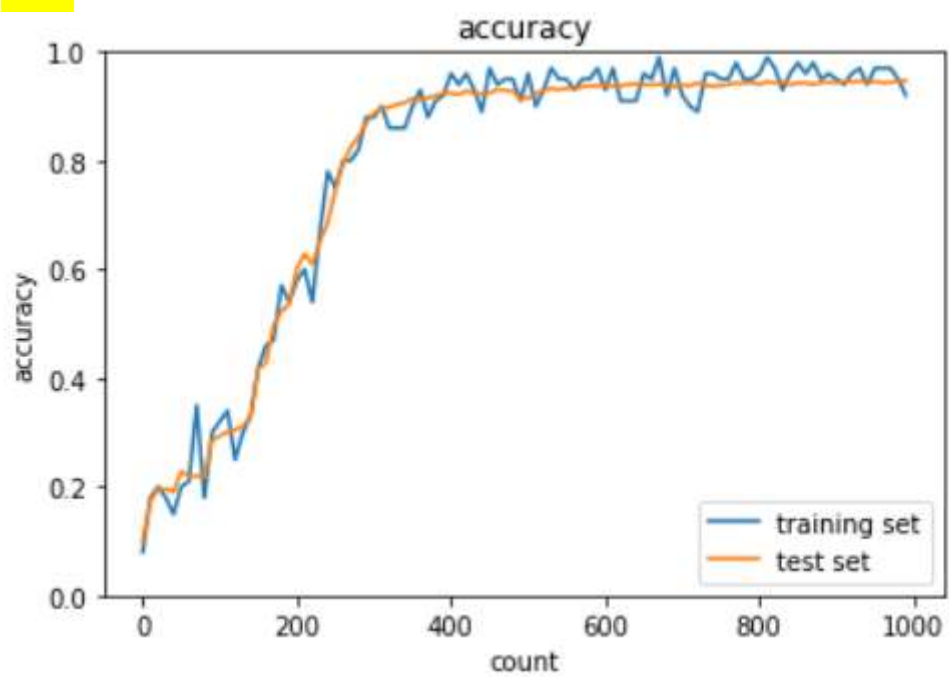
RSMprop



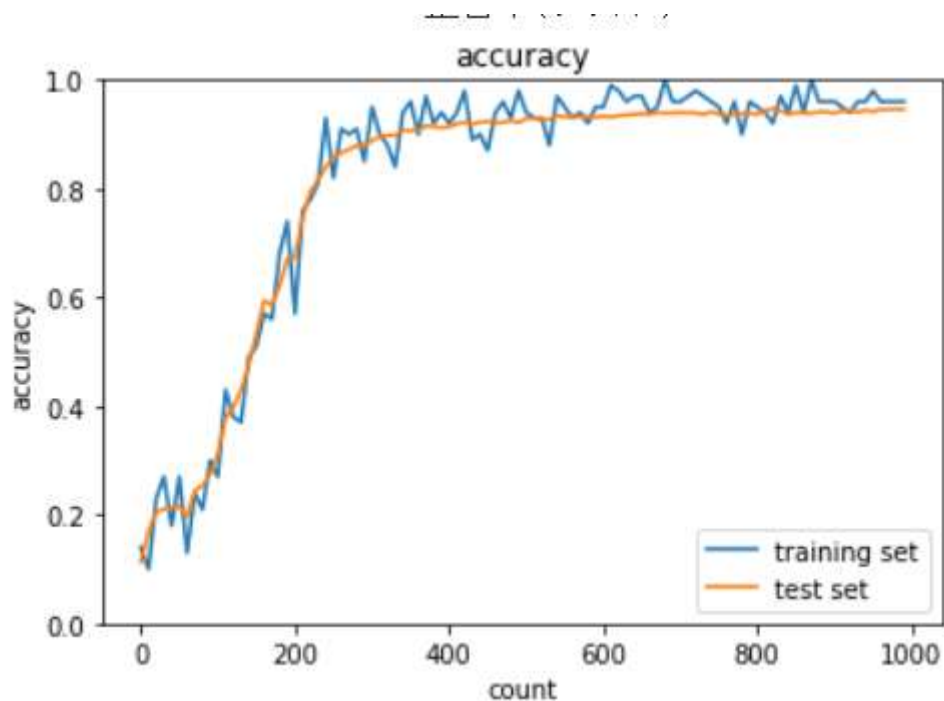
最適化 (optimize) 後



Adam



最適化 (optimize) 後



参照コード:

```
from google.colab import drive
drive.mount('/content/drive')
import sys
sys.path.append('/content/drive/My Drive/DNN_code')
sys.path.append('/content/drive/My Drive/DNN_code/lesson_2')
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, o
ne_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)

        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
# 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)

        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()

# AdaGradを作ってみよう
# 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 では不必要
# =====

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]

# =====

    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)

        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)

        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)

        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()

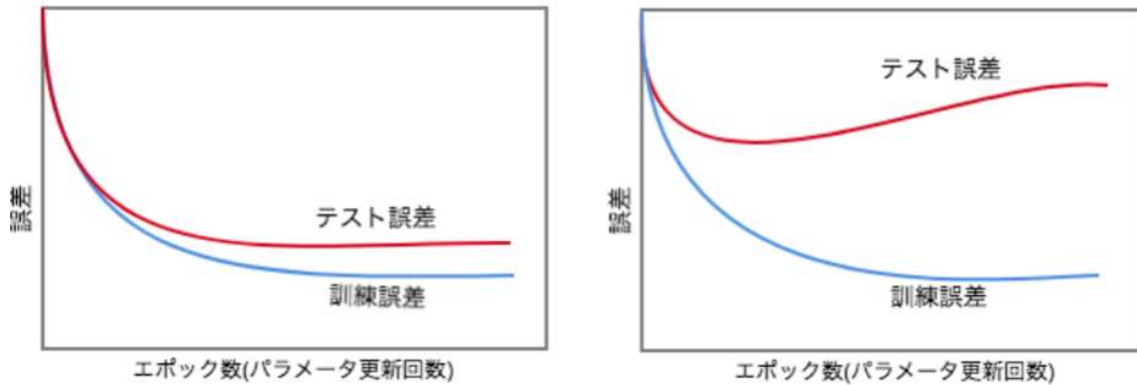
```

モメンタム・AdaGrad・RMSPropの特徴を
それぞれ簡潔に説明せよ。
(3分)

- モメンタムのメリット
局所的最適解にはならず、大域的最適解となる。
谷間についてから最も低い位置(最適値)に行くまでの時間が早い。
- AdaGrad のメリット
勾配の緩やかな斜面に対して、最適値に近づける。
- RMSProp のメリット
局所的最適解にはならず、大域的最適解となる。
ハイパーパラメータの調整が必要な場合が少ない。

Section 3 : 過学習

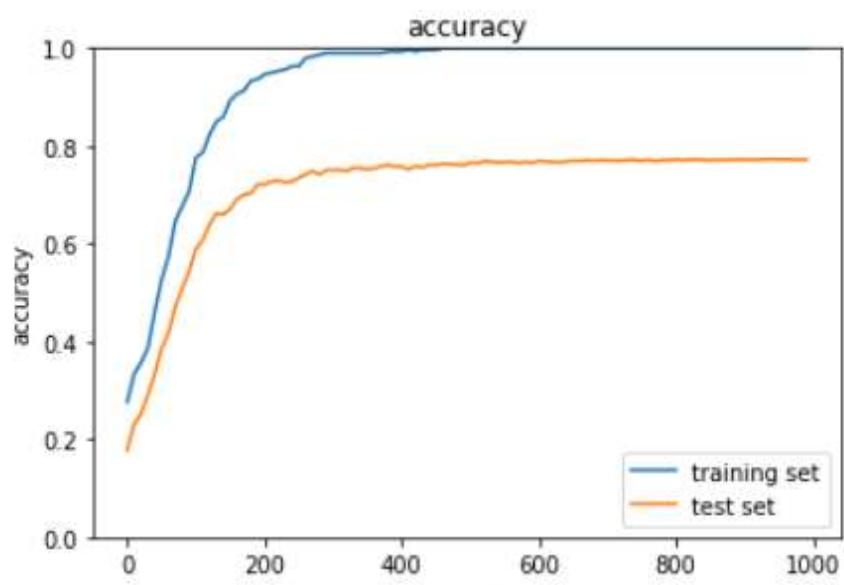
- ・ 過学習：テスト誤差と訓練誤差とで学習曲線が乖離すること
特定の訓練サンプルに対して、特化して学習する



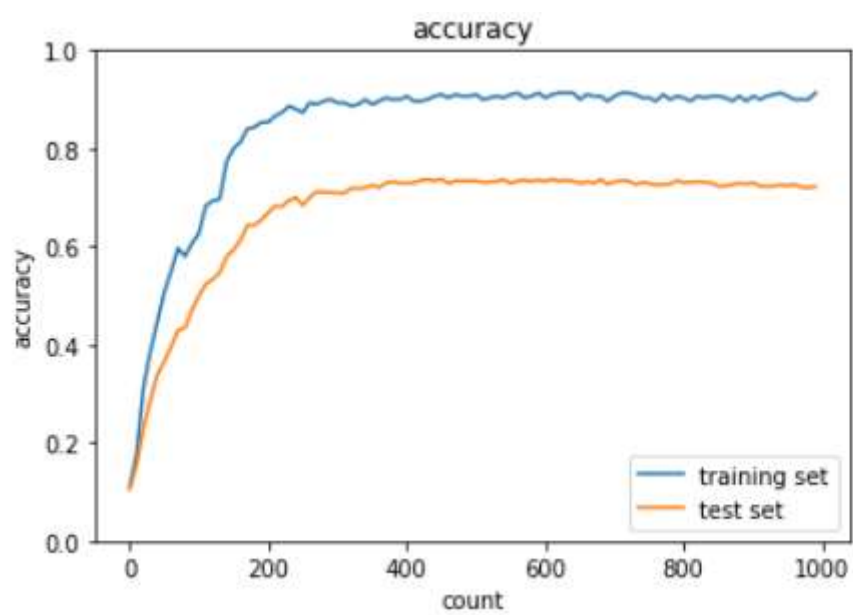
- ・ 正則化：ネットワークの自由度（層数、ノード数、パラメータの値等）を制約すること
正則化手法を利用して過学習を抑制する
L1 正則化（Lasso 回帰）マンハッタン距離（P1 ノルム）
L2 正則化（Ridge 回帰）ユークリッド距離（P2 ノルム）
ドロップアウト（ランダムにノードを削除して学習させること）
- ・ Weight decay（荷重減衰）
重みが大きい値は、学習において重要な値であるが、重みが大きいと過学習が起こる。
過学習がおこりそうな重みの大きさ以下で重みをコントロールし、かつ重みの大きさにはばらつきを出す必要がある。

実行結果：

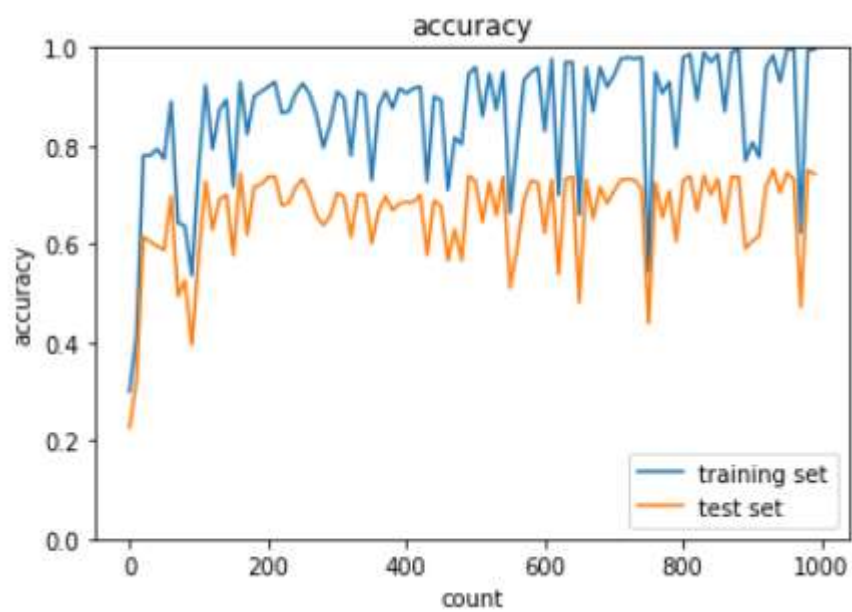
Overfitting



L2

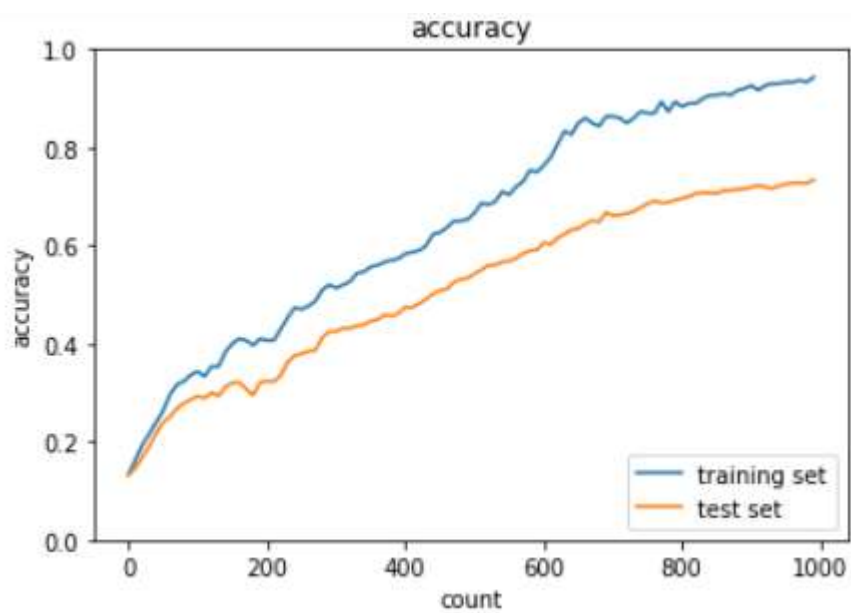


L1

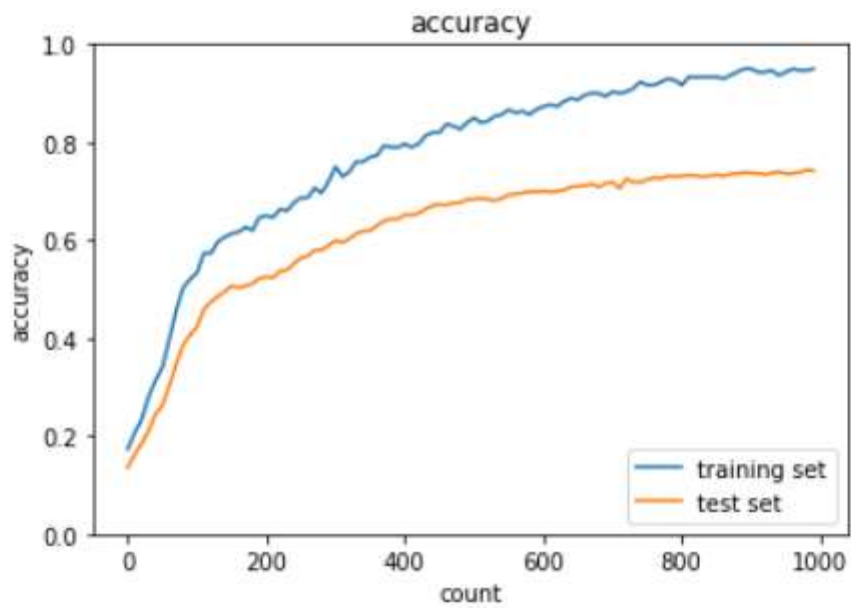


Dropout

[13]



Dropout + L1



参照コード:

```
from google.colab import drive
drive.mount('/content/drive')
import sys
sys.path.append('/content/drive/My Drive/DNN_code')
sys.path.append('/content/drive/My Drive/DNN_code/lesson_2')
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, 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)

        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()
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)

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)

```

```
        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()
```

```
(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)
```

```
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 = 0.005
# =====

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)

```

```

        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()

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, 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)

    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()

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)

        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()

```

確認テスト 1

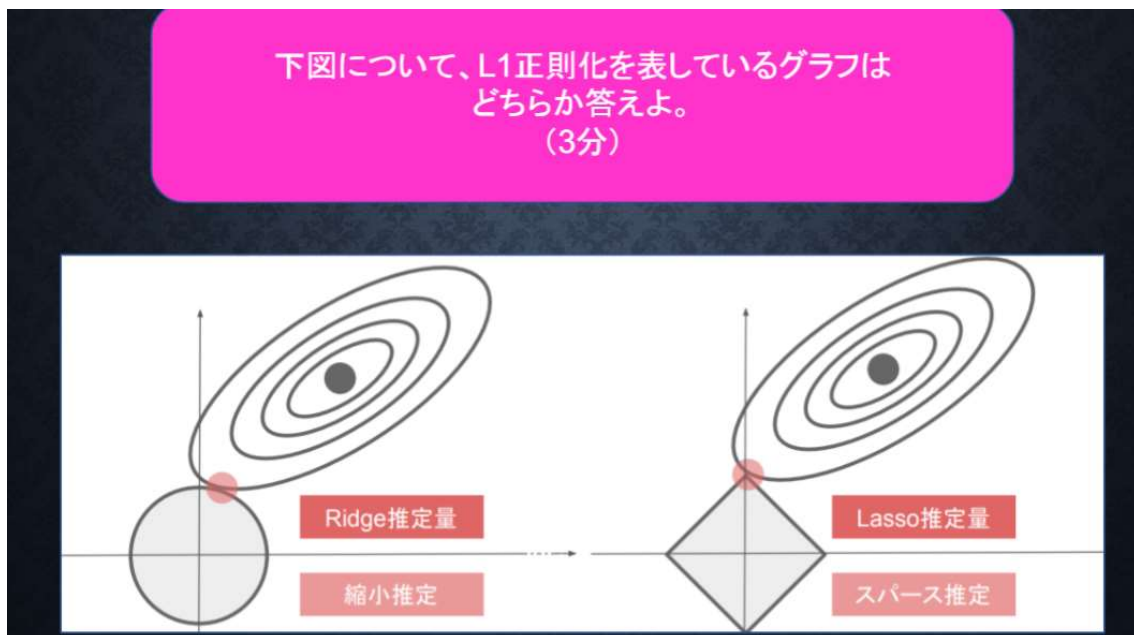
機械学習で使われる線形モデル(線形回帰,主成分分析…etc)の正則化は、モデルの重みを制限することで可能となる。
前述の線形モデルの正則化手法の中にリッジ回帰という手法があり、その特徴として正しいものを選択しなさい。

- (a)ハイパーパラメータを大きな値に設定すると、すべての重みが限りなく0に近づく
- (b)ハイパーパラメータを0に設定すると、非線形回帰となる。
- (c)バイアス項についても、正則化される
- (d)リッジ回帰の場合、隠れ層に対して正則化項を加える

回答：(a)

確認テスト 2

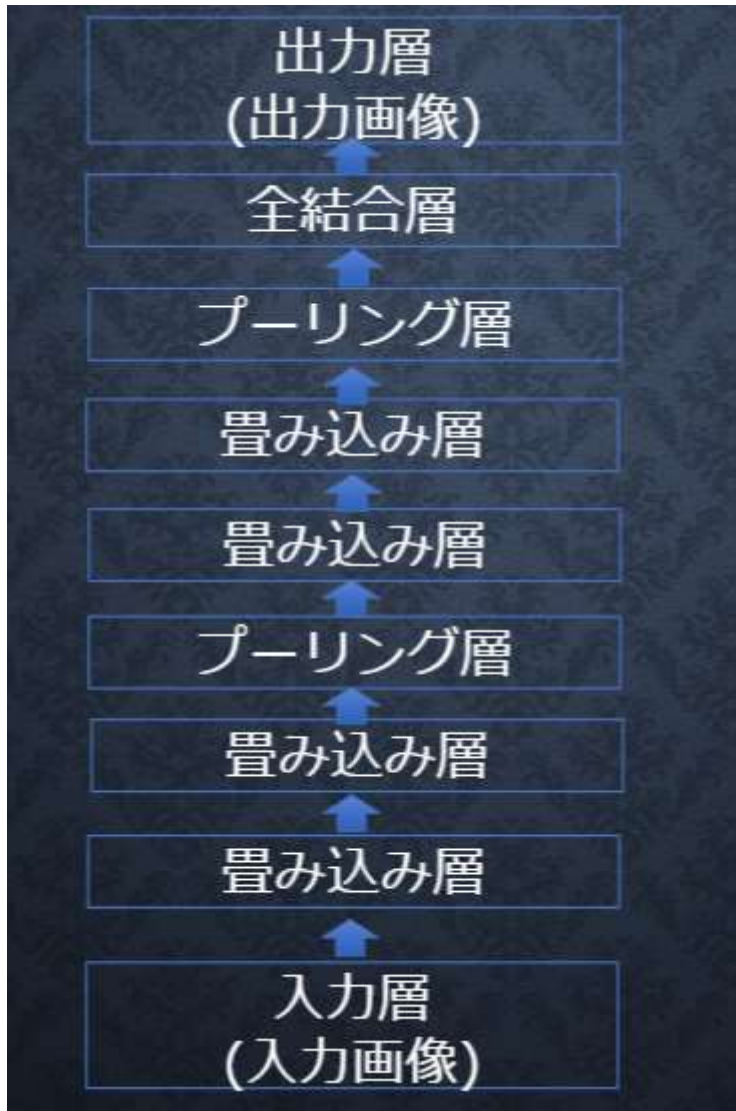
下図について、L1正則化を表しているグラフは
どちらか答えよ。
(3分)



回答は 右の Lasso 推定量

Section 4：畳み込みニューラルネットワークの概念

・CNN 構造図(例)



- ・ CNN：画像処理を行う時に良く用いられるニューラルネットワーク。次元間で繋がりのあるデータならなんでも扱える。
- ・ LeNet：畳み込みニューラルネットワークの代表的なものの1つ。
- ・ 畳み込み層：畳み込み層では、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。
全結合層のデメリット画像の場合、縦、横、チャンネルの3次元データだが、1次元のデータとして処理される。
畳み込み層：3次元の空間情報も学習できるような層が畳み込み層である。
フィルター（全結合でいう重み）
バイアス（畳み込みの演算概念）

パディング（畳み込み演算を何回も繰り返すと画像が小さくなってしまう）

ストライド（ストライドの数はダイレクトに出力画像を小さくする要因になる）

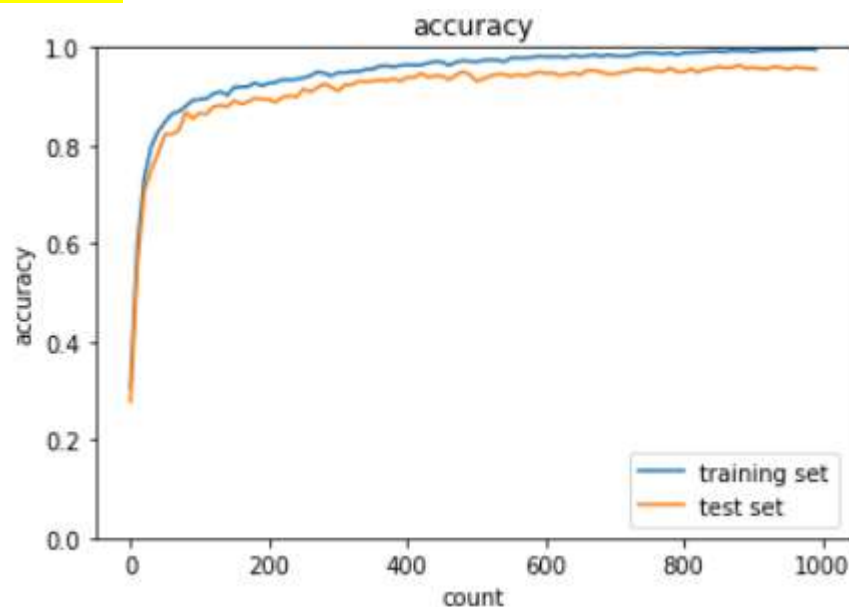
チャンネル（フィルターの数）

畳み込み演算をしたくなった理由：全結合では、縦、横、チャンネルの3次元データ（特徴）が1次元のデータとして処理される。それでは上手く画像が持つ情報から特徴を抽出する事ができない。

全結合層のデメリット画像の場合、縦、横、チャンネルの3次元データだが、1次元のデータとして処理される

im2col: 画像認識において用いられている関数です。動作としては多次元配列を2次元配列へ、可逆的に変換します。

実行結果：



参照コード：

```
from google.colab import drive
drive.mount('/content/drive')
import sys
sys.path.append('/content/drive/My Drive/DNN_code')
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
    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

# 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('=====')
# 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]
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を戻す
    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.p
ad)

    return dx

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)

# 行ごとに最大値を求める
arg_max = np.argmax(col, axis=1)
out = np.max(col, axis=1)
# 整形
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.sh
ape[2], -1)
    dx = col2im(dcol, self.x.shape, self.pool_h, self.pool_w, s
elf.stride, self.pad)

    return dx

class SimpleConvNet:
    # conv - relu - pool - affine - relu - affine - softmax
    def __init__(self, input_dim=(1, 28, 28), conv_param={'filter_n
um':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_p
ad) / 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(filte
r_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(hidde
n_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, s
tride=2)
        self.layers['Affine1'] = layers.Affine(self.params['W2'], s
elf.params['b2'])
        self.layers['Relu2'] = layers.Relu()
        self.layers['Affine2'] = layers.Affine(self.params['W3'], s
elf.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)

    # 設定

```

```

        grad = {}
        grad['W1'], grad['b1'] = self.layers['Conv1'].dW, self.layers['Conv1'].db
        grad['W2'], grad['b2'] = self.layers['Affine1'].dW, self.layers['Affine1'].db
        grad['W3'], grad['b3'] = 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[:5000], d_train[:5000]
x_test, d_test = x_test[:1000], d_test[:1000]

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)

        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()

```


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を答えよ。
なおストライドとパディングは1とする。
(3分)

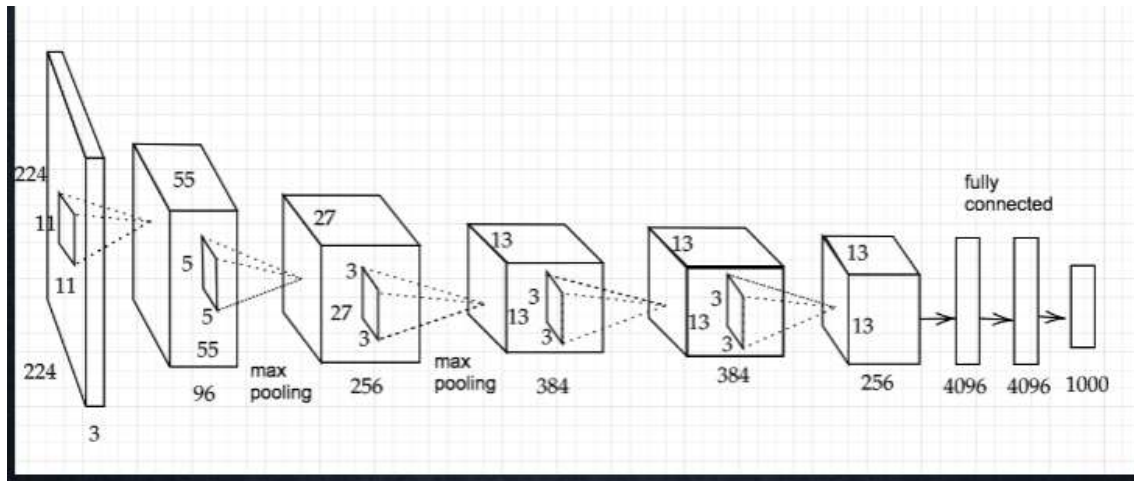
$$(6 + 2 * 1 - 2) / 1 + 1 = 7$$

$$(6 + 2 * 1 - 2) / 1 + 1 = 7 \quad \text{結果：} 7 * 7$$

Section 5 : 最新のCNN

- ・ AlexNet : 過学習を防ぐ施策・ サイズ 4096 の全結合層の出力にドロップアウトを使用している

イメージ図



モデルの構造：5層の畳み込み層、及びプーリング層等、それに続く3層の全結合層から構成される。

畳み込み演算の部分から全結合層に至る部分について：[13, 13, 256]の画像

Flatten：横1列にずらっと並べるだけ[$13 \times 13 \times 256 = 43264$]。初期のニューラルネットワークでの1列の並べ替えでは非常に良く行われている。

Global Max Pooling：[13*13]をあたかも1つのフィルタのように見立て、1番大を使用する。[256]にまで圧縮される。

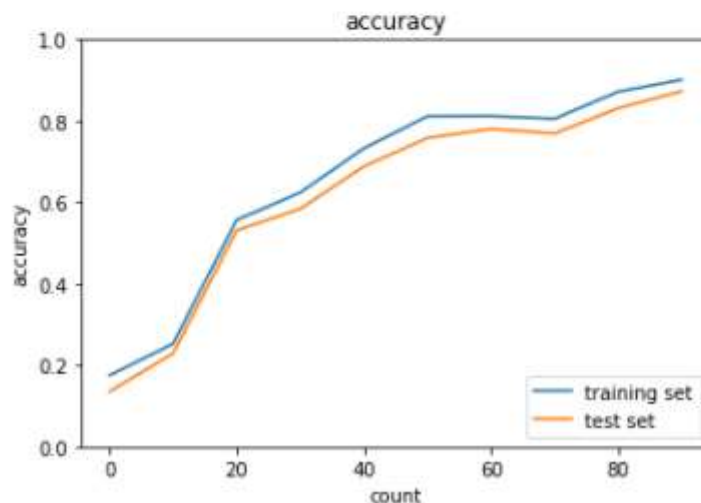
Global Avg Pooling：[13*13]をあたかも1つのフィルタのように見立て、平均を使用する。[256]にまで圧縮される。

後2つは何故か非常に上手くいく。一気に数値を減らせる割に非常に効率的に特徴量を抽出して認識精度の向上をはかる事ができる。

実行結果：(iters_num = 100) で実施

☞ (1024, 50)

```
Generation: 10. 正答率(トレーニング) = 0.1746  
               : 10. 正答率(テスト) = 0.135  
Generation: 20. 正答率(トレーニング) = 0.2526  
               : 20. 正答率(テスト) = 0.229  
Generation: 30. 正答率(トレーニング) = 0.5564  
               : 30. 正答率(テスト) = 0.531  
Generation: 40. 正答率(トレーニング) = 0.624  
               : 40. 正答率(テスト) = 0.584  
Generation: 50. 正答率(トレーニング) = 0.7324  
               : 50. 正答率(テスト) = 0.688  
Generation: 60. 正答率(トレーニング) = 0.8112  
               : 60. 正答率(テスト) = 0.758  
Generation: 70. 正答率(トレーニング) = 0.8118  
               : 70. 正答率(テスト) = 0.78  
Generation: 80. 正答率(トレーニング) = 0.8052  
               : 80. 正答率(テスト) = 0.769  
Generation: 90. 正答率(トレーニング) = 0.8716  
               : 90. 正答率(テスト) = 0.831  
Generation: 100. 正答率(トレーニング) = 0.9008  
                : 100. 正答率(テスト) = 0.873
```



参照コード：

```
from google.colab import drive  
drive.mount('/content/drive')  
  
import sys  
sys.path.append('/content/drive/My Drive/DNN_code')  
  
import pickle  
  
import numpy as np  
  
from collections import OrderedDict  
from common import layers  
from data.mnist import load_mnist  
import matplotlib.pyplot as plt
```

```

from common import optimizer

class DeepConvNet:
    '''
    認識率 99%以上の高精度な ConvNet

    conv - relu - conv- relu - pool -
    conv - relu - conv- relu - pool -
    conv - relu - conv- relu - pool -
    affine - relu - dropout - affine - dropout - softmax
    '''
    def __init__(self, input_dim=(1, 28, 28),
                  conv_param_1 = {'filter_num':16, 'filter_size':3,
                                'pad':1, 'stride':1},
                  conv_param_2 = {'filter_num':16, 'filter_size':3,
                                'pad':1, 'stride':1},
                  conv_param_3 = {'filter_num':32, 'filter_size':3,
                                'pad':1, 'stride':1},
                  conv_param_4 = {'filter_num':32, 'filter_size':3,
                                'pad':2, 'stride':1},
                  conv_param_5 = {'filter_num':64, 'filter_size':3,
                                'pad':1, 'stride':1},
                  conv_param_6 = {'filter_num':64, 'filter_size':3,
                                'pad':1, 'stride':1},
                  hidden_size=50, output_size=10):
        # 重みの初期化=====
        # 各層のニューロンひとつあたりが、前層のニューロンといくつのつながりがあるか
        pre_node_nums = np.array([1*3*3, 16*3*3, 16*3*3, 32*3*3, 32
                                *3*3, 64*3*3, 64*4*4, hidden_size])
        wight_init_scales = np.sqrt(2.0 / pre_node_nums) # He の初期
        値

        self.params = {}
        pre_channel_num = input_dim[0]
        for idx, conv_param in enumerate([conv_param_1, conv_param_
2, conv_param_3, conv_param_4, conv_param_5, conv_param_6]):

```

```

        self.params['W' + str(idx+1)] = wight_init_scales[idx]
        * np.random.randn(conv_param['filter_num'], pre_channel_num, conv_p
aram['filter_size'], conv_param['filter_size'])
        self.params['b' + str(idx+1)] = np.zeros(conv_param['fi
lter_num'])

        pre_channel_num = conv_param['filter_num']
        self.params['W7'] = wight_init_scales[6] * np.random.randn(
pre_node_nums[6], hidden_size)
        print(self.params['W7'].shape)
        self.params['b7'] = np.zeros(hidden_size)
        self.params['W8'] = wight_init_scales[7] * np.random.randn(
pre_node_nums[7], output_size)
        self.params['b8'] = np.zeros(output_size)

# レイヤの生成=====
self.layers = []
self.layers.append(layers.Convolution(self.params['W1'], se
lf.params['b1'],
                                conv_param_1['stride'], conv_param_1['pa
d']))

self.layers.append(layers.Relu())
self.layers.append(layers.Convolution(self.params['W2'], se
lf.params['b2'],
                                conv_param_2['stride'], conv_param_2['pa
d']))

self.layers.append(layers.Relu())
self.layers.append(layers.Pooling(pool_h=2, pool_w=2, strid
e=2))

self.layers.append(layers.Convolution(self.params['W3'], se
lf.params['b3'],
                                conv_param_3['stride'], conv_param_3['pa
d']))

self.layers.append(layers.Relu())
self.layers.append(layers.Convolution(self.params['W4'], se
lf.params['b4'],

```

```

        conv_param_4['stride'], conv_param_4['padding']))
        self.layers.append(layers.ReLU())
        self.layers.append(layers.Pooling(pool_h=2, pool_w=2, stride=2))
        self.layers.append(layers.Convolution(self.params['W5'], self.params['b5'],
        conv_param_5['stride'], conv_param_5['padding']))
        self.layers.append(layers.ReLU())
        self.layers.append(layers.Convolution(self.params['W6'], self.params['b6'],
        conv_param_6['stride'], conv_param_6['padding']))
        self.layers.append(layers.ReLU())
        self.layers.append(layers.Pooling(pool_h=2, pool_w=2, stride=2))
        self.layers.append(layers.Affine(self.params['W7'], self.params['b7']))
        self.layers.append(layers.ReLU())
        self.layers.append(layers.Dropout(0.5))
        self.layers.append(layers.Affine(self.params['W8'], self.params['b8']))
        self.layers.append(layers.Dropout(0.5))

        self.last_layer = layers.SoftmaxWithLoss()

    def predict(self, x, train_flg=False):
        for layer in self.layers:
            if isinstance(layer, layers.Dropout):
                x = layer.forward(x, train_flg)
            else:
                x = layer.forward(x)
        return x

    def loss(self, x, d):

```

```

        y = self.predict(x, train_flg=True)
        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, train_flg=False)
        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)

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

    # 設定
    grads = {}
    for i, layer_idx in enumerate((0, 2, 5, 7, 10, 12, 15, 18))
:
        grads['W' + str(i+1)] = self.layers[layer_idx].dW
        grads['b' + str(i+1)] = self.layers[layer_idx].db

```

```

        return grads
(x_train, d_train), (x_test, d_test) = load_mnist(flatten=False)

# 処理に時間のかかる場合はデータを削減
x_train, d_train = x_train[:5000], d_train[:5000]
x_test, d_test = x_test[:1000], d_test[:1000]

print("データ読み込み完了")

network = DeepConvNet()
optimizer = optimizer.Adam()

iters_num = 100
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)

    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()

```

参考資料

YouTube (Able Programming) [はじめて学ぶ深層学習](https://www.youtube.com/watch?v=O3ohRBi5-Og)

<https://www.youtube.com/watch?v=O3ohRBi5-Og>

【深層学習】深層学習とは？ | ディープラーニングの意味、ニューラルネットワーク

【深層学習】活性化関数 | ReLU、シグモイド関数、ソフトマックス関数

【深層学習】損失関数/勾配降下法 | 交差エントロピー誤差、ミニバッチ勾配降下法

【深層学習】誤差逆伝播法 | バックプロパゲーション

【深層学習】深層学習の実装 | 深層学習フレームワーク、Keras