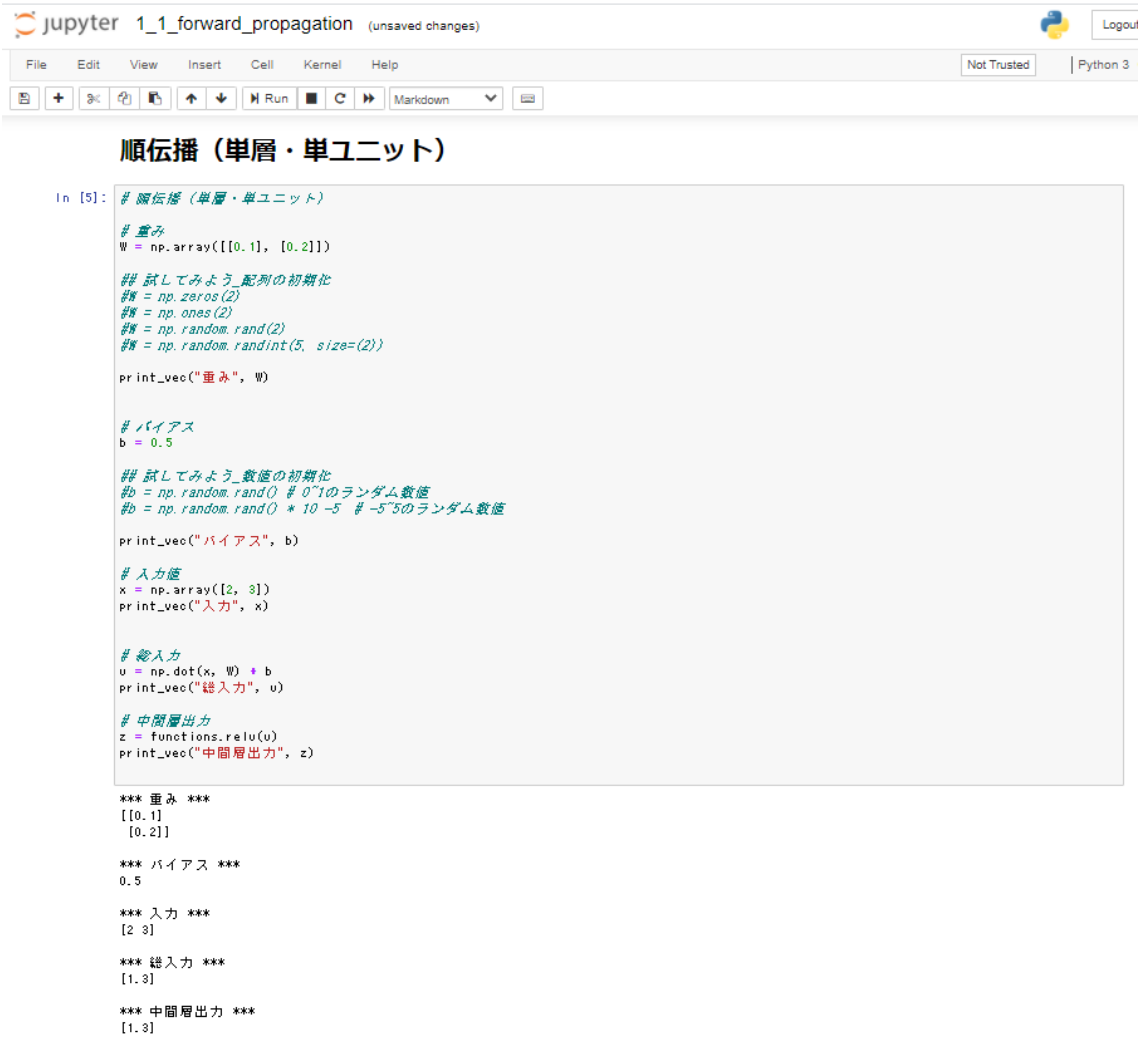
# 深層学習レポート day1

- 入力層～中間層
  - ✓ 要点のまとめ

    入力層とは、ニューラルネットワークにおける 1 層目のことである。中間層とは、入力層と出力層の間にあるすべての層のことである。一般的に、中間層が 4 層以上のニューラルネットワークをディープニューラルネットワークと呼んでいる。中間層では、入力に重みをかけ、バイアスを加算し、活性化関数による変換を施した値を次の層へと出力している。
  - ✓ 実装演習結果キャプチャと考察



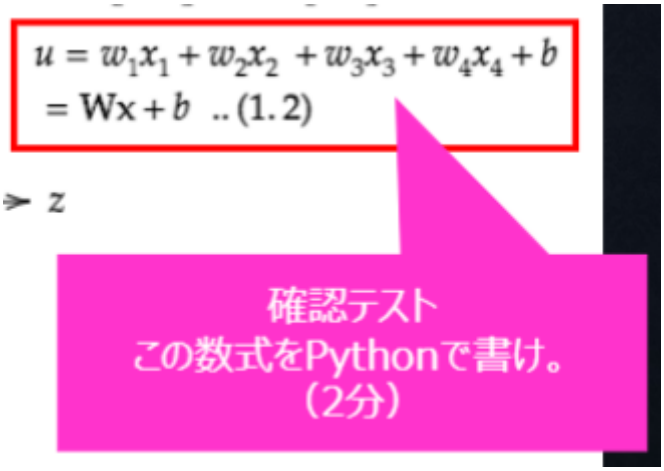

バイアスと重みの値をランダムに初期化した場合

## 順伝播（単層・単ユニット）

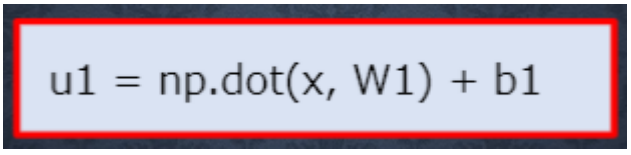
In [8]:
```
# 順伝播（単層・単ユニット）

# 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 ***
[1 1]

*** バイアス ***
-3.0787283369301433

*** 入力 ***
[2 3]

*** 総入力 ***
1.9212716630698567

*** 中間層出力 ***
1.9212716630698567
```

✓ 確認テストなど、自身の考察結果




この確認テストの回答は以下である。




np.dot(w, W1) + b1 について、Python のどのようなコードなのか調べてみた。

まず、「np」というのは python の数値計算ライブラリである Numpy のことである。この記法で用いられる場合、コードの中に「import numpy as np」という記述が必要。「np.dot」により、numpy の dot 関数と使いますという表記になる。dot 関数は行列同士の掛け算をする関数で、第一引数と第二引数にかけたい行列を記述している。

- 活性化関数
  - ✓ 要点のまとめ

    活性化関数とは、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。中間層で用いられる活性化関数の代表的なものに ReLU がある。ReLU は勾配消失問題の解決法として考案された活性化関数であり、非常に層の多いニューラルネットであっても学習を進めることができる。
  - ✓ 実装演習結果キャプチャと考察

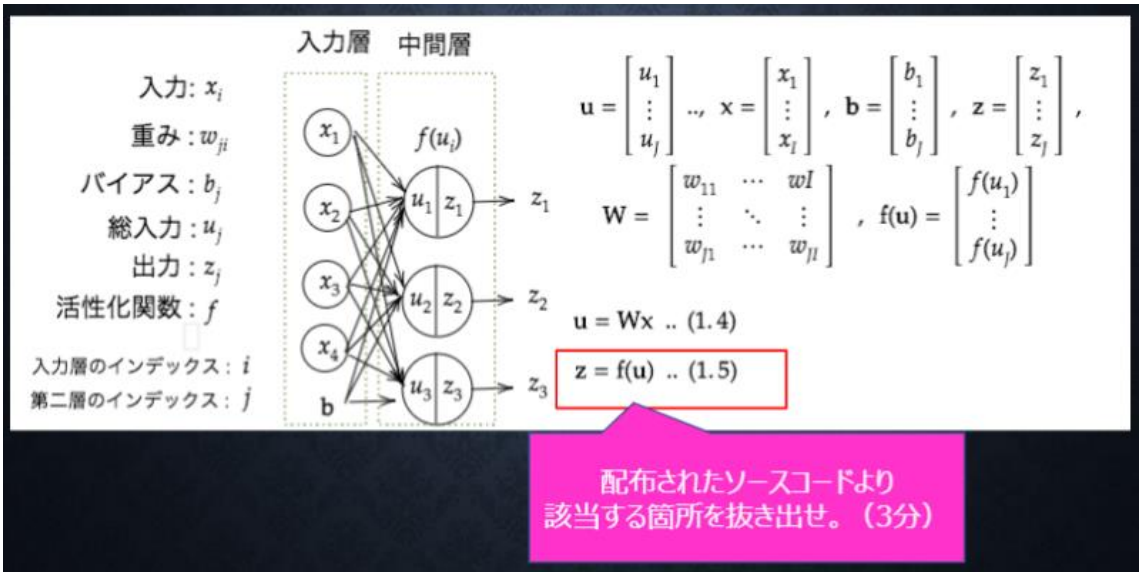
**順伝播（単層・単ユニット）**

In [9]:
```
# 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.sigmoid(u)
print_vec("中間層出力", z)
```

```
*** 重み ***
[3 3]

*** バイアス ***
-0.9937218440748694

*** 入力 ***
[2 3]

*** 総入力 ***
14.00627815592513

*** 中間層出力 ***
0.9999991736760774
```

活性化関数を ReLU からシグモイド関数に変更した。中間層出力が 0-1 の間に収まっていることがわかる。

  - ✓ 確認テストなど、自身の考察結果




回答は以下である。

```
z1 = functions.sigmoid(u)
```

functions とは、コードの中でインポートしているライブラリのことである。「from common import functions」という記述がそれに該当。

functions.py の中で、シグモイド関数が定義されているのは以下の内容である。

jupyter functions.py ✔ 1時間前

File Edit View Language

```
1  import numpy as np
2
3  # 中間層の活性化関数
4  # シグモイド関数（ロジスティック関数）
5  def sigmoid(x):
6      return 1/(1 + np.exp(-x))
7
```

- 出力層
  - ✓ 要点のまとめ

    出力層では、予測の結果と教師データとの誤差を評価する、誤差関数が用いられる。分類の問題ではクロスエントロピー誤差を用いるのが一般的である。分類における出力層では、最終的な出力結果を確率として解釈できるよう、二値分類ではシグモイド関数を、多項分類ではソフトマックス関数を用いるのが一般的である。回帰の問題では恒等写像を用いる。
  - ✓ 実装演習結果キャプチャと考察

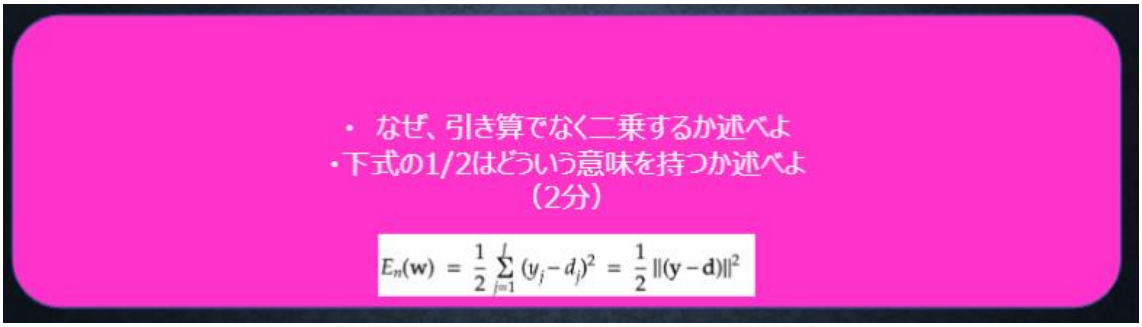
## 多クラス分類（2-3-4ネットワーク）

In [13]:
```python
# 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

# ウェイトとバイアスを設定
# ネートワークを作成
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])

    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)
```

他クラス分類を実施。重みの shape を途中で出力する仕様に変更した。
結果は以下の通り。

```
##### 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 0.7 1. ]
 [0.2 0.5 0.8 1.1]
 [0.3 0.6 0.9 1.2]]

*** バイアス1 ***
[0.1 0.2 0.3]

*** バイアス2 ***
[0.1 0.2 0.3 0.4]

##### 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 3.56 4.83]

*** 出力1 ***
[0.01602796 0.05707321 0.20322929 0.72366954]

出力合計: 1.0
*** 重みW1のshape ***
(2, 3)

*** 重みW2のshape ***
(3, 4)


##### 結果表示 #####
*** 出力 ***
[0.01602796 0.05707321 0.20322929 0.72366954]

*** 訓練データ ***
[0 0 0 1]

*** 誤差 ***
0.323420293360194 23
```

✓ 確認テストなど、自身の考察結果



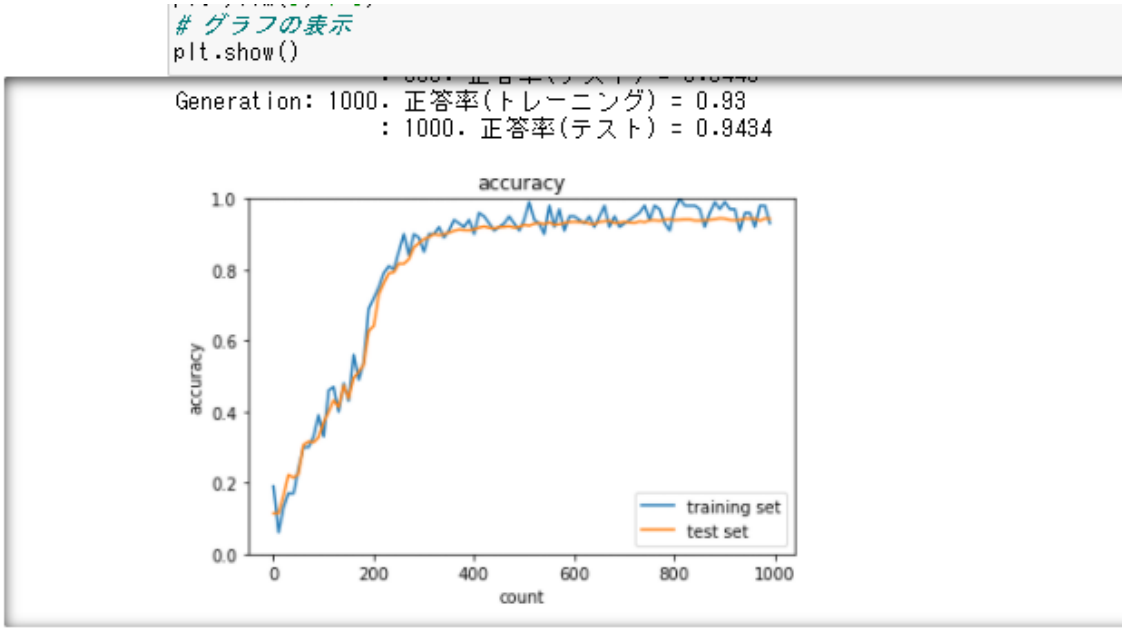

回答：

引き算では誤差が正になる場合と負になる場合が打ち消しあってしまうため。

誤差関数についている 1/2 は誤差関数を微分した場合に打ち消されて計算しやすくなるような工夫である。

- 勾配降下法
  - ✓ 要点のまとめ

    深層学習の目的は、学習を通じて、誤差 E(w)を最小にする w を見つけ出すことである。そのために用いられる手法が勾配降下法。勾配降下法の重要なパラメータとして、学習率がある。学習率が適切に設定されない場合、学習が進まない「発散」が起きたり、学習の完了に非常に長い時間がかかる場合がある。学習率を効率よく決定するためのアルゴリズムとして、「Momentum」「AdaGrad」「Adadelta」「Adam」などが知られている。
  - ✓ 実装演習結果キャプチャと考察

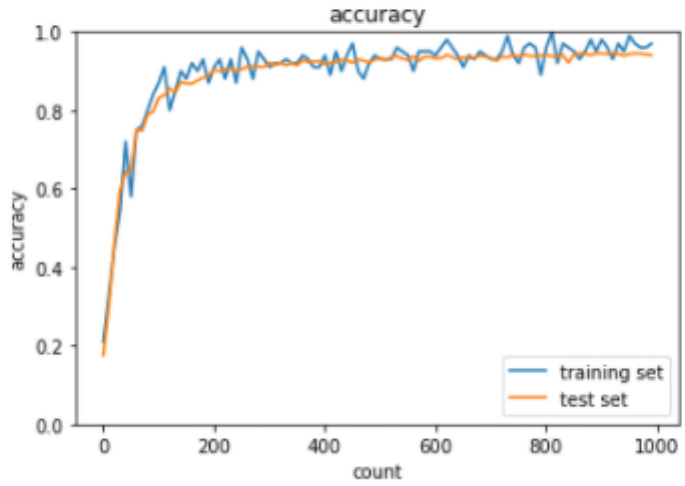



Adam で MNIST を分類した精度。まずはサンプルコードのまま。

次に、活性化関数を ReLU に変えてみる。

```
# グラフの表示
plt.show()
```

```
Generation: 1000. 正答率(トレーニング) = 0.97
                : 1000. 正答率(テスト) = 0.94
```

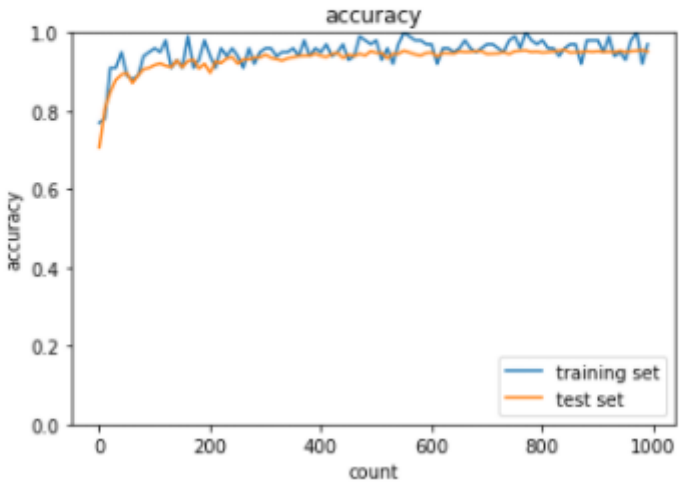



最終的な正答率には大きな差がないが、学習の収束が早くなっているのがわかる。

次に、活性化関数 ReLU のまま、He の初期値を用いる。

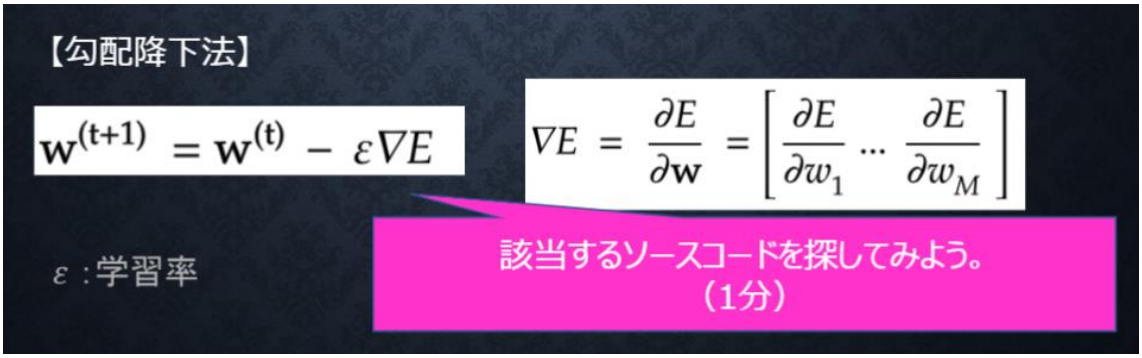
```
# グラフの表示
plt.show()
```

```
Generation: 1000. 正答率(トレーニング) = 0.97
                : 1000. 正答率(テスト) = 0.9518
```

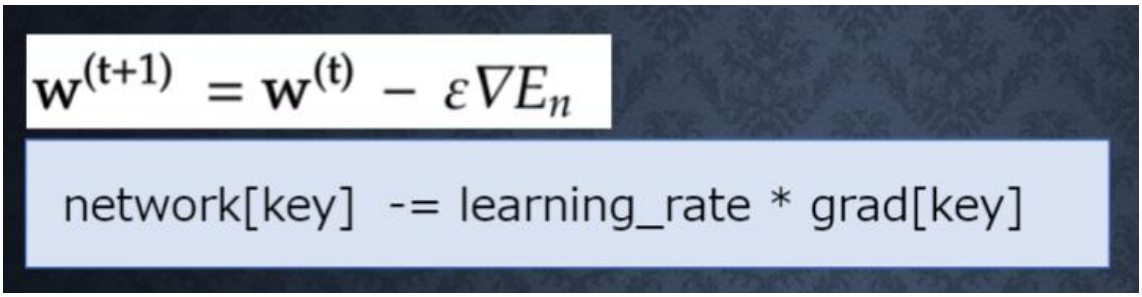



正答率は上がり、収束までの epoch 数も少なくなった。

✓ 確認テストなど、自身の考察結果




回答は以下




-=という省略の記法で記述してあるが、

network[key] = network[key] – learning_rate * grad[key] と同値。

learning_rate が ε に対応し、grad[key が ΔE に対応している。

- 誤差逆伝播法
  - ✓ 要点のまとめ

    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誤差関数の微分を算出することができる。活性化関数の微分が何度も出現するため、sigmoid 関数を何度も活性化関数に用いたニューラルネットワークでは勾配消失を起こす場合がある。
  - ✓ 実装演習結果キャプチャと考察

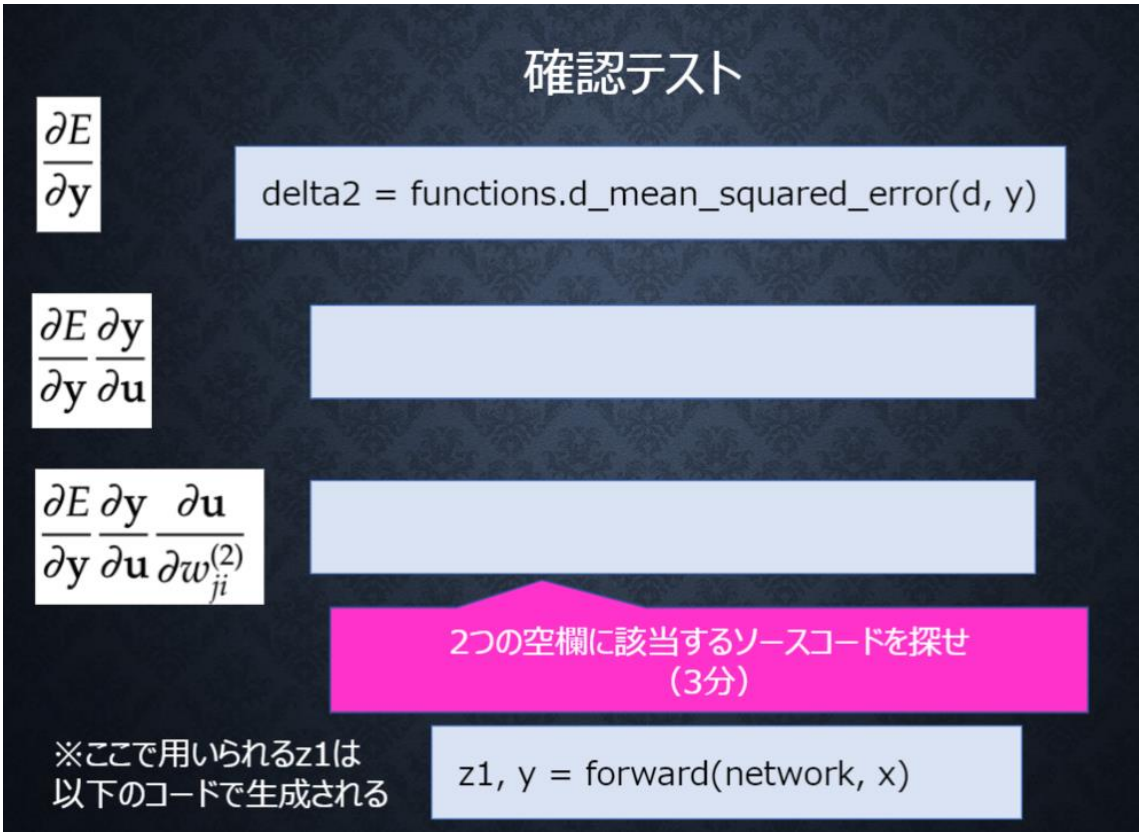
```
##### ネットワークの初期化 #####
*** 重み1 ***
[[0.1 0.3 0.5]
 [0.2 0.4 0.6]]

*** 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 ***
[[0.08708577 0.91293423]]

出力合計: 1.0

##### 誤差逆伝播開始 #####
*** 偏微分_dE/du2 ***
[[ 0.08708577 -0.08708577]]

*** 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]
 [ 0.21788443 -0.21788443]
 [ 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 ***
[ 0.08708577 -0.08708577]

##### 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]
```

誤差逆伝播の実行結果。

誤差逆伝播により重みのパラメータが調整されていることがわかる。

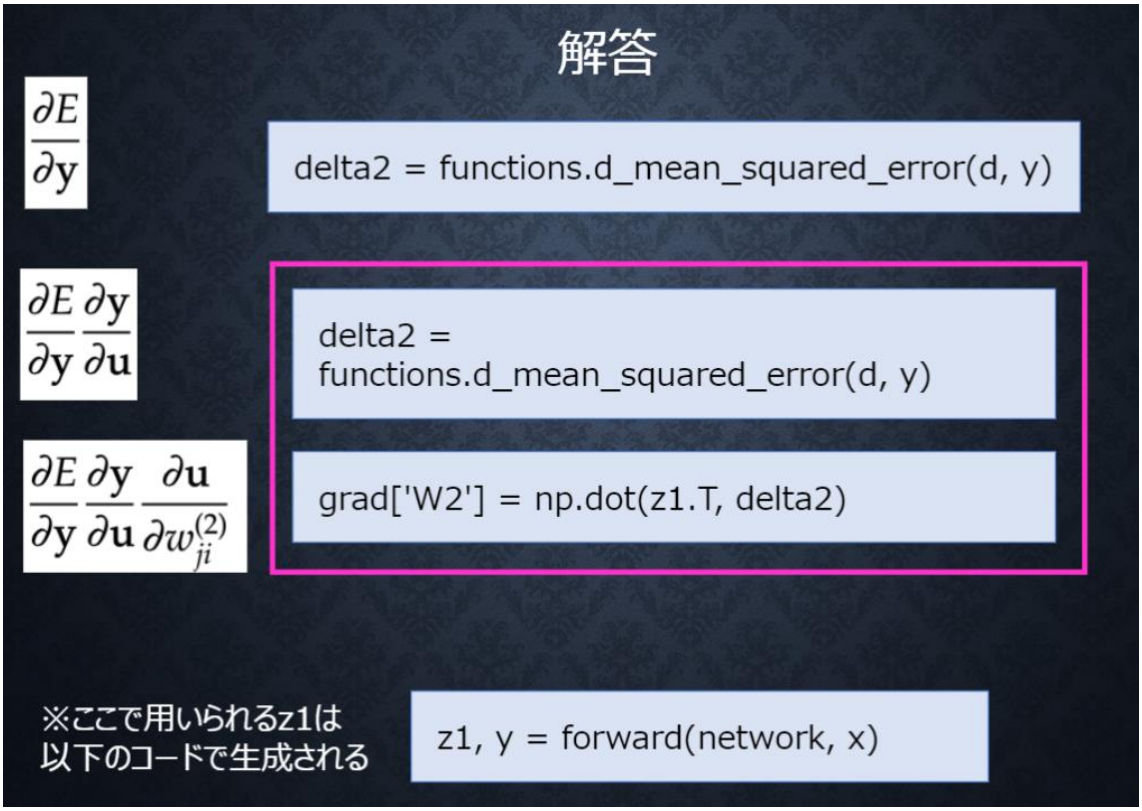
重みの shape には変化がないことがわかる。

✓ 確認テストなど、自身の考察結果


確認テスト
$\frac{\partial E}{\partial y}$
delta2 = functions.d_mean_squared_error(d, y)
$\frac{\partial E}{\partial y}\frac{\partial y}{\partial u}$
$\frac{\partial E}{\partial y}\frac{\partial y}{\partial u}\frac{\partial u}{\partial w_{ji}^{(2)}}$
2つの空欄に該当するソースコードを探せ
（3分）
※ここで用いられるz1は
以下のコードで生成される
z1, y = forward(network, x)

➢

回答は以下である。


解答
$\frac{\partial E}{\partial y}$
delta2 = functions.d_mean_squared_error(d, y)
$\frac{\partial E}{\partial y}\frac{\partial y}{\partial u}$
delta2 =
functions.d_mean_squared_error(d, y)
$\frac{\partial E}{\partial y}\frac{\partial y}{\partial u}\frac{\partial u}{\partial w_{ji}^{(2)}}$
grad['W2'] = np.dot(z1.T, delta2)
※ここで用いられるz1は
以下のコードで生成される
z1, y = forward(network, x)

ここで、functions.d_mean_squared_error(d,y)は functions.py で定義されている誤差関数の導関数である。コードの内容は以下のようになっていた。

```
# 平均二乗誤差の導関数
def d_mean_squared_error(d, y):
    if type(d) == np.ndarray:
        batch_size = d.shape[0]
        dx = (y - d)/batch_size
    else:
        dx = y - d
    return dx
```

w1.T という表記は、w1 の転置行列という意味である。
具体的には以下のようになっていた。

```
In [6]: print(z1)
        print(z1.T)

        [[1.2 2.5 3.8]]
        [[1.2]
         [2.5]
         [3.8]]
```

# 深層学習レポート day2

勾配消失問題

- ✓ 要点のまとめ

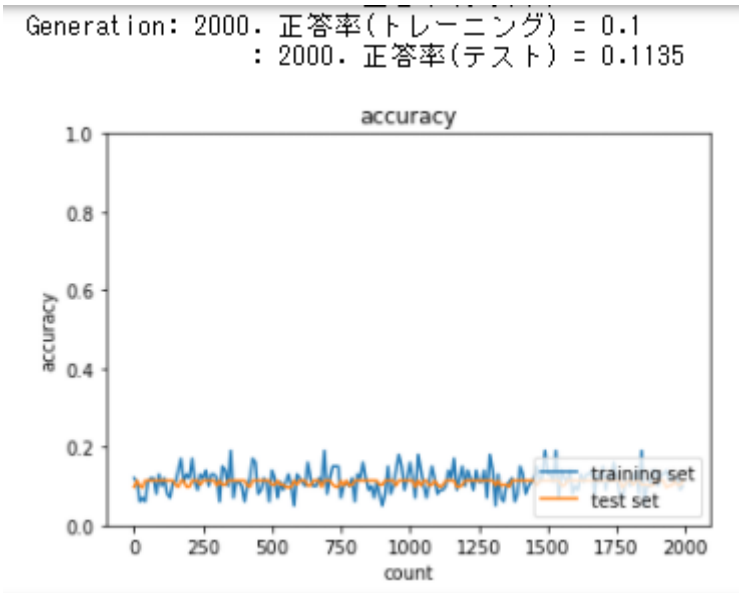
  勾配消失問題とは、誤差逆伝播法において、下位の層に逆伝播が進むにつれ、重みの更新量が少なくなること。主な原因に活性化関数の導関数の最大値が 1 未満であることがあげられる。

  勾配消失問題の解決方法として、活性化関数に ReLU など、導関数の最大値が 1 未満でないものを採用することがあげられる。また、重みの初期値に He や Xavier を用いることでも勾配消失問題が改善される。活性化関数に値を渡す前にバッチ正規化の処理を施す、バッチ正規化という手法もある。

- ✓ 実装演習結果キャプチャと考察

  活性化関数：シグモイド

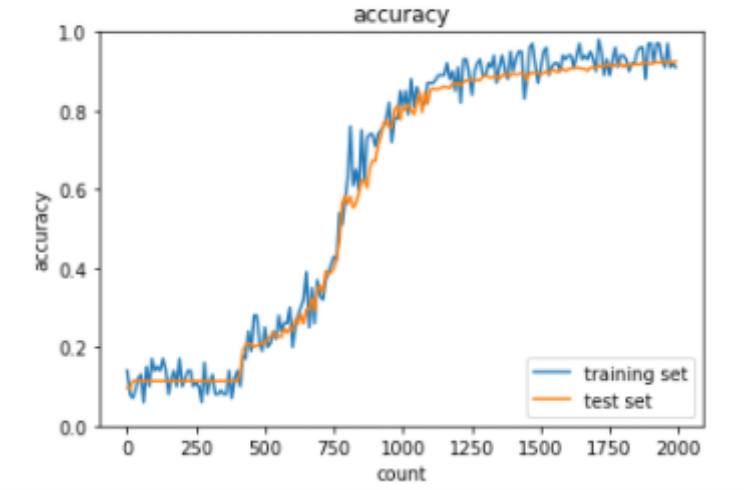
  初期値：ガウス分布




  活性化関数：ReLU

  初期値：ガウス分布

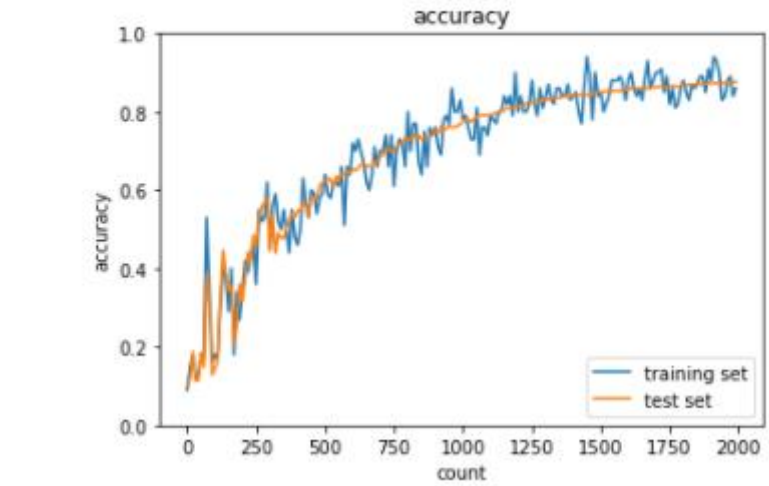
```
Generation: 2000. 正答率(トレーニング) = 0.91
                : 2000. 正答率(テスト) = 0.9253
```




活性化関数：シグモイド

初期値：Xavier

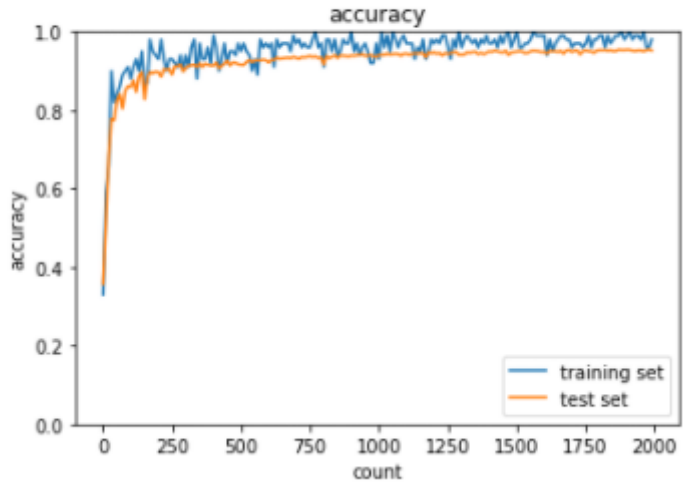
```
Generation: 2000. 正答率(トレーニング) = 0.86
                : 2000. 正答率(テスト) = 0.8762
```




活性化関数：ReLU

初期値：He

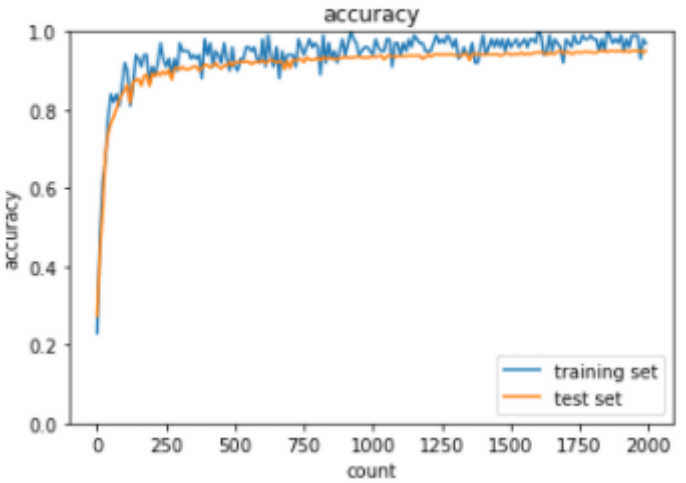
```
Generation: 2000. 正答率(トレーニング) = 0.98
                : 2000. 正答率(テスト) = 0.9523
```




活性化関数：ReLU

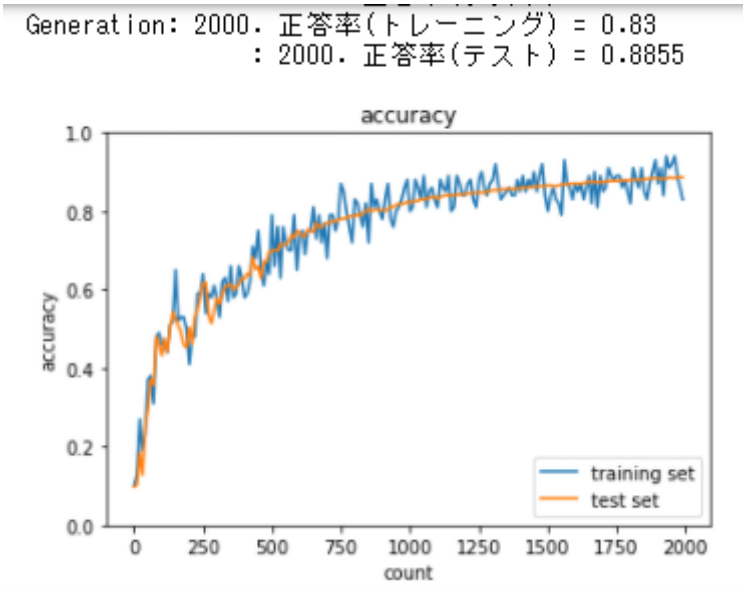
初期値：Xavier

```
Generation: 2000. 正答率(トレーニング) = 0.97
                : 2000. 正答率(テスト) = 0.9502
```




活性化関数：シグモイド

初期値：He




- ✓ 確認テストなど、自身の考察結果

  バッチ正規化の欠点は、バッチサイズをある程度（一般的には 16 以上）確保しないと使えないという点である。これは、エッジコンピューティングを行う際などに問題となる。この欠点を補うための手法として、レイヤー正規化、インスタンス正規化、グループ正規化などの手法がある。時系列データに対してよく使われる RNN に対してはレイヤーノームが有効である。

  参考：https://qiita.com/omiita/items/01855ff13cc6d3720ea4

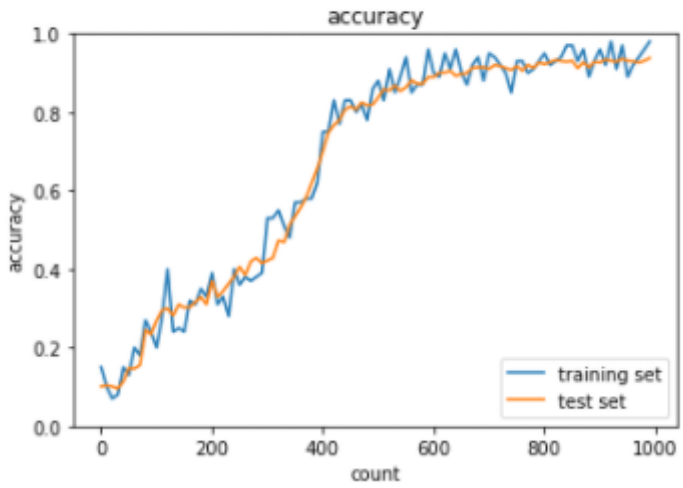
- ➢ 学習率最適化手法
  - ✓ 要点のまとめ

    学習率を適切に設定しない場合、誤差関数の値が収束しない（発散）ことや、収束までのエポック数が大きくなること、局所解に収束してしまうなどの問題が起きることがある。学習率を最適に決定するアルゴリズムとして「モメンタム」「AdaGrad」「RMSProp」「Adam」などがある。
  - ✓ 実装演習結果キャプチャと考察

    MNIST を各種学習率最適化手法で解く演習

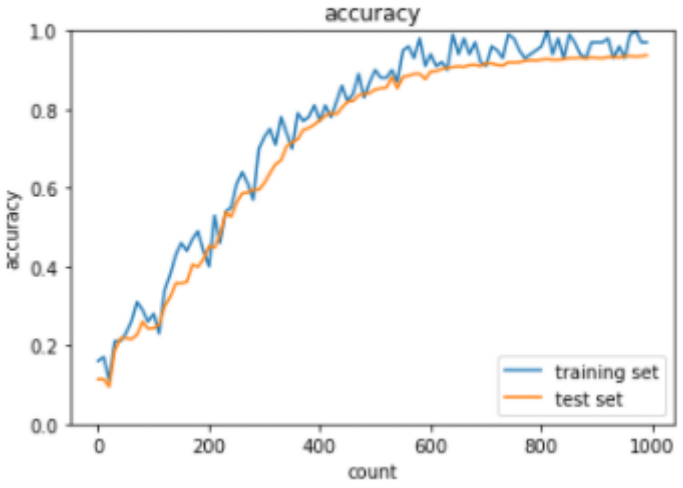
    ・モメンタム

```
Generation: 1000. 正答率(トレーニング) = 0.98
                : 1000. 正答率(テスト) = 0.938
```



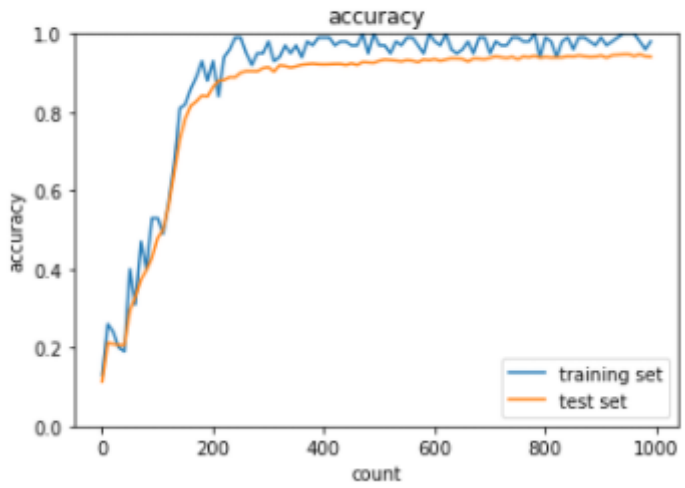

・AdaGrad

```
Generation: 1000. 正答率(トレーニング) = 0.97
                : 1000. 正答率(テスト) = 0.9375
```



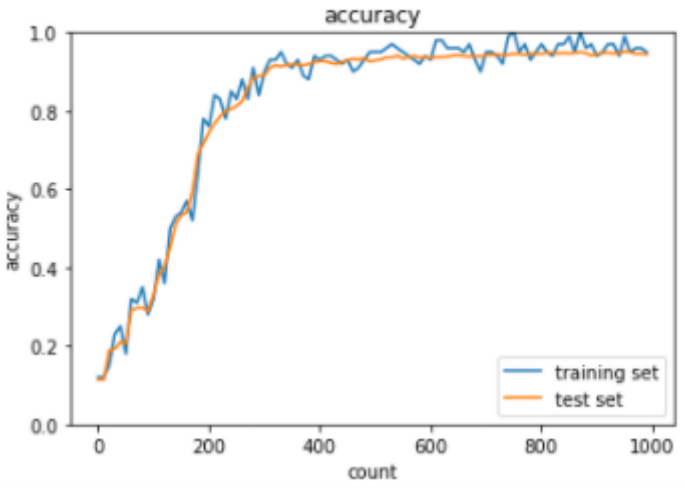

・RMSProp

```
Generation: 1000. 正答率(トレーニング) = 0.98
                : 1000. 正答率(テスト) = 0.9408
```

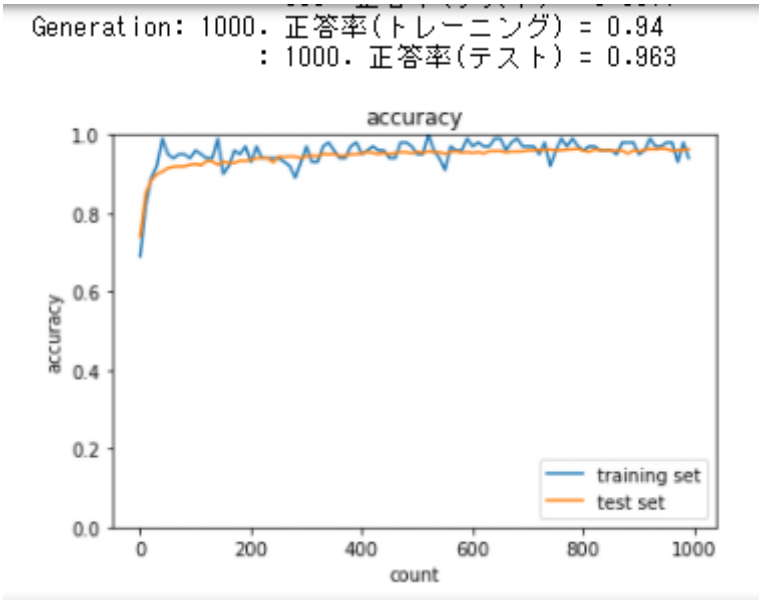



・Adam

```
Generation: 1000. 正答率(トレーニング) = 0.95
                : 1000. 正答率(テスト) = 0.9437
```




・Adam(バッチ正規化あり,活性化関数：ReLU, 重みの初期値：He)
収束が早まり、精度も向上した。




✓ 確認テストなど、自身の考察結果


・モメンタム

局所最適解になりにくい。更新量の算出に慣性項を導入することで、収束が早まっている。

・AdaGrad

誤差が凹みの極値に近づくほど、重みの更新量を減らすように、誤差関数の偏微分を 2 乗した項を使っている。誤差関数の導関数が 0 となる鞍点で学習が進まなくなる、鞍点問題を起こすことがある。

・RMSProp

勾配の大きなところで学習率を低くするような項が採用されており、学習の際に振動を起こしにくくなっている。

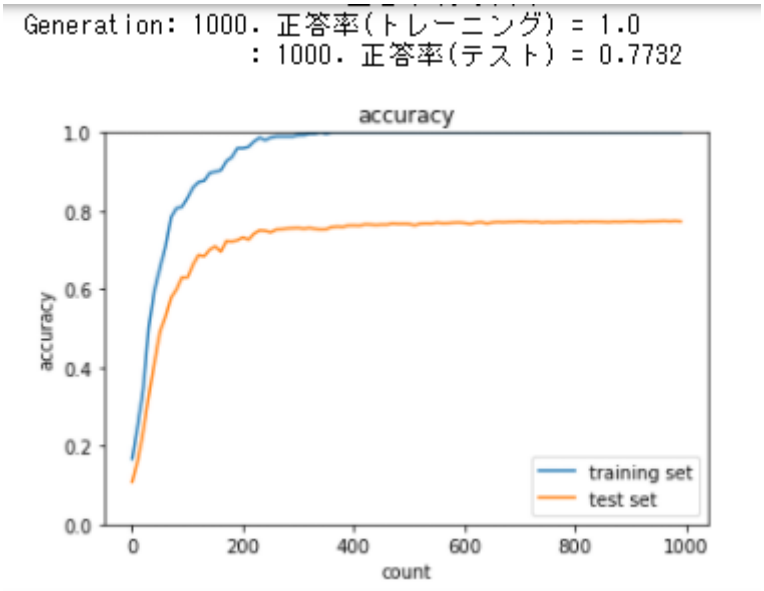
・Adam

モメンタムのメリットと、RMSProp のメリットを併せ持ったアルゴリズム

- 過学習
  - ✓ 要点のまとめ

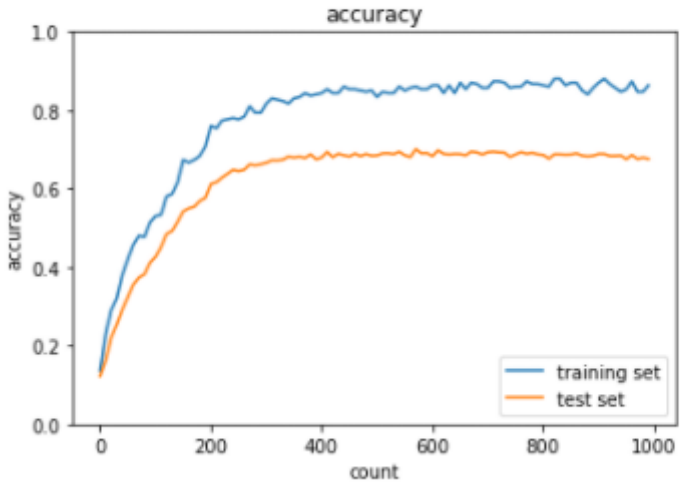
    過学習とは、訓練データに過剰に適合し、検証データやテストデータに適合しない状態のこと。汎化性能が低いとも言い換えることができる。過学習を防ぐ方法として、正則化項の導入がある。重みが大きくなることにペナルティを与えることで、汎化性能を失わない範囲で学習を進めることができる。Lasso 正則化と Ridge 正則化の 2 手法がある。
  - ✓ 実装演習結果キャプチャと考察

    正則化も dropout も設定しない場合、訓練データ正答率が 100%になってしまい、検証データの正答率と大きく乖離している。



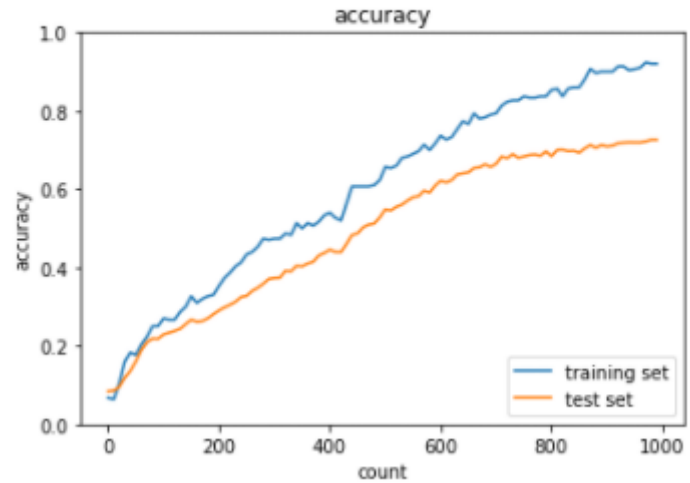

L2 正則化を導入すると、訓練と検証の正答率の乖離はやや小さくなる。600 回目の試行あたりから検証の正答率が下がり始めており、1000 回では試行回数が多すぎるように見受けられる。

```
Generation: 1000. 正答率(トレーニング) = 0.8633333333333333
                : 1000. 正答率(テスト) = 0.6759
```

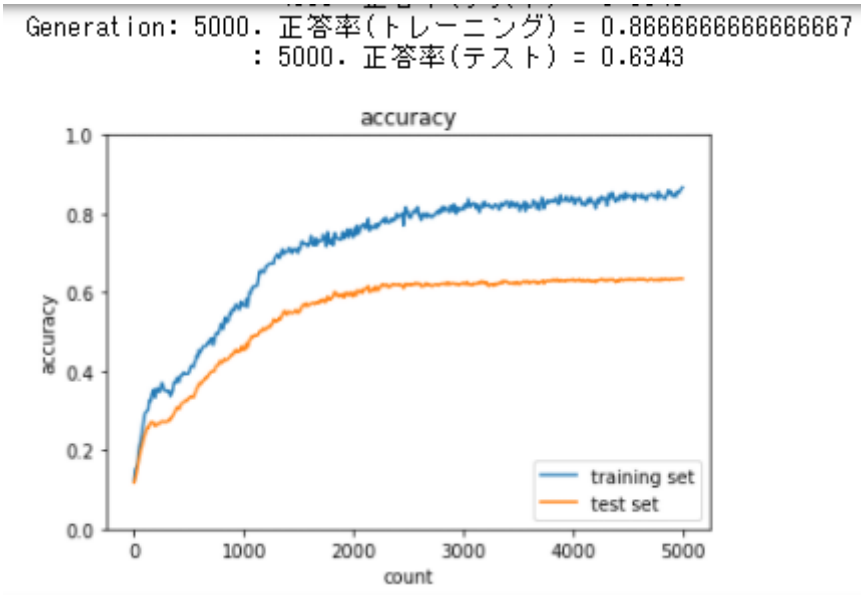



dropout を導入したモデル。訓練と検証の正答率の乖離が小さく、テストの正答率も上記のものと比べて高い。dropout には過学習を抑える効果がある一方、学習に時間がかかるというデメリットがある。1000 回の試行の段階では検証の正答率が上昇している途中であり、学習が完了しているとは言えない。

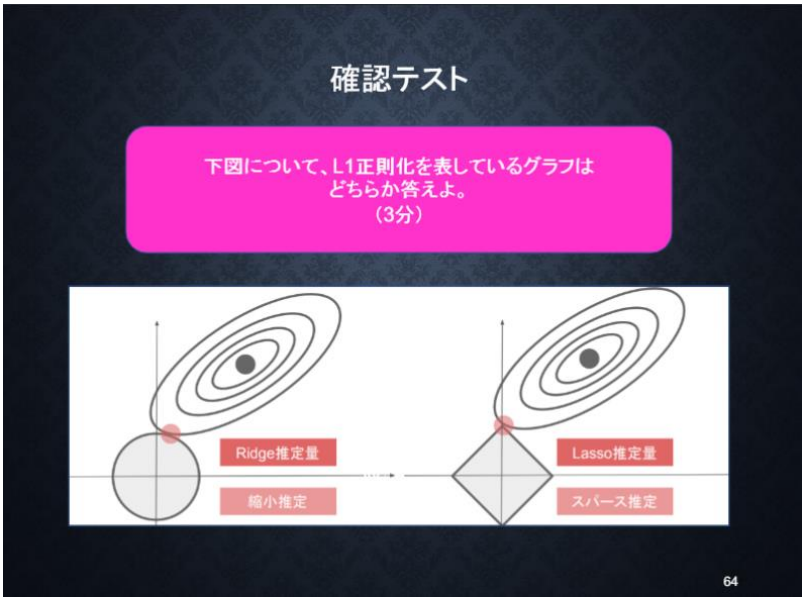
```
Generation: 1000. 正答率(トレーニング) = 0.92
                : 1000. 正答率(テスト) = 0.7261
```




L2 正則化＋dropout(0.15)+イテレーション(5000)のモデル




✓ 確認テストなど、自身の考察結果




L1 正則化を表すのは右の図。L1 正則化はパラメータの大きさをマンハッタン距離で評価する。そのため、パラメータを 2 次元と見立てた場合、等高線はひし形を描く。一方で L2 正則化はパラメータの大きさをユークリッド距離で評価する。そのため、パラメータを 2 次元と見立てた場合、等高線は円を描く。

➢ 畳み込みニューラルネットワークの概念

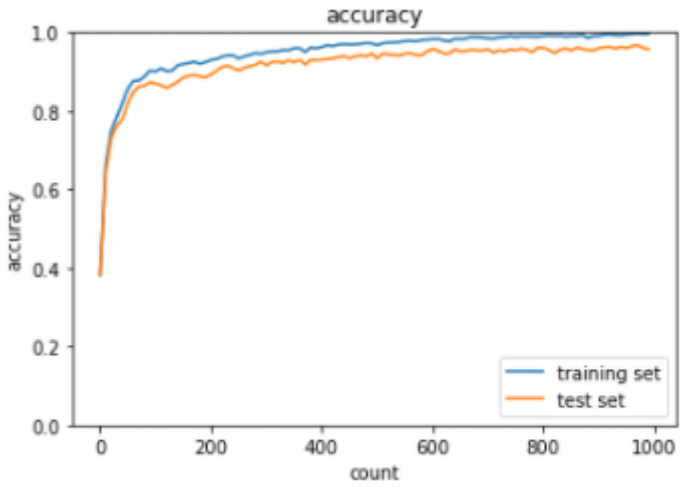
✓ 要点のまとめ

畳み込みニューラルネットワークとは、畳み込み層を含んだニューラルネットワークのことである。畳み込み層は、次元間に関係性のあるようなデータ（Ex.画像データ、時系列データ）に対して有効な手法である。入力データに対してフィルタ

ーをかける。学習の対象となるのはフィルターの重みである。フィルターをかけることで、データのサイズが小さくなってしまう問題は、パディングの手法を用いて解決することができる。代表的なのはデータの周囲を 0 で埋める zero-padding である。過学習を防ぐため、最初の 2 つの全結合層には dropout が設定されている。

✓ 実装演習結果キャプチャと考察

2_6_simple_convolution_network_after の結果




im2col の結果

```
========== input_data ===========
 [[[[48.  2. 74. 50.]
   [60.  0. 56. 22.]
   [62. 44.  0. 95.]
   [70. 60. 21. 76.]]]


 [[[56. 96. 73. 30.]
   [80. 67. 13.  1.]
   [62. 24. 85. 17.]
   [69. 32. 60. 93.]]]]
==============================
============= col ==============
 [[48.  2. 74. 60.  0. 56. 62. 44.  0.]
 [ 2. 74. 50.  0. 56. 22. 44.  0. 95.]
 [60.  0. 56. 62. 44.  0. 70. 60. 21.]
 [ 0. 56. 22. 44.  0. 95. 60. 21. 76.]
 [56. 96. 73. 80. 67. 13. 62. 24. 85.]
 [96. 73. 30. 67. 13.  1. 24. 85. 17.]
 [80. 67. 13. 62. 24. 85. 69. 32. 60.]
 [67. 13.  1. 24. 85. 17. 32. 60. 93.]]
==============================
```

im2col の結果（transpose なし）

```
========== input_data ===========
 [[[[14. 76. 97. 14.]
   [44. 48. 34. 61.]
   [ 2. 88.  9. 53.]
   [35. 91.  1. 31.]]]


 [[[96. 41. 66. 64.]
   [56. 57. 67. 31.]
   [22. 75. 63. 49.]
   [ 9. 30. 23. 17.]]]]
=============================
============= col ==============
 [[14. 76. 44. 48. 76. 97. 48. 34. 97.]
 [14. 34. 61. 44. 48.  2. 88. 48. 34.]
 [88.  9. 34. 61.  9. 53.  2. 88. 35.]
 [91. 88.  9. 91.  1.  9. 53.  1. 31.]
 [96. 41. 56. 57. 41. 66. 57. 67. 66.]
 [64. 67. 31. 56. 57. 22. 75. 57. 67.]
 [75. 63. 67. 31. 63. 49. 22. 75.  9.]
 [30. 75. 63. 30. 23. 63. 49. 23. 17.]]
=============================
```

- ✓ 確認テストなど、自身の考察結果

  講義では 3 次元データ(画像)に対する 2 次元フィルタの例が扱われたが、4 次元データに対して 3 次元フィルタを適用することも可能なのではないかと思い、調べてみた。CT 画像に対して 3 次元フィルタを用い、肺野結節影の検出に応用した新潟大学大学院の事例があった。この例では ResNet を三次元化した 3D-ResNet をを用いて学習を行っている。

  参考：https://www.jstage.jst.go.jp/article/pjsai/JSAI2020/0/JSAI2020_1C5GS1305/_article/-char/ja/

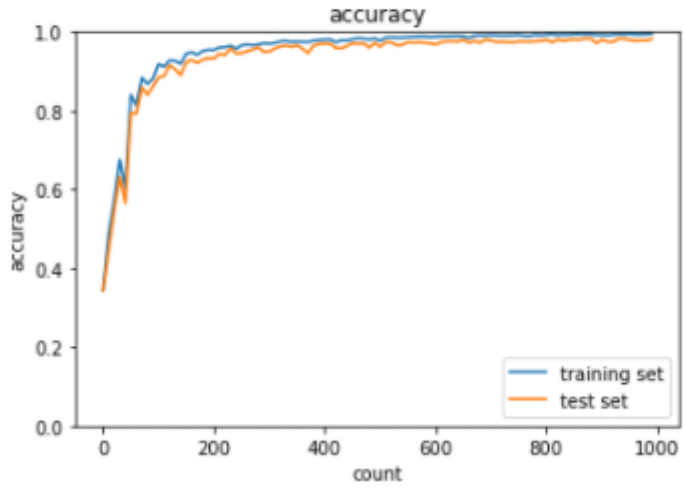
- 最新の CNN
  - ✓ 要点のまとめ

    AlexNet は、2012 年の ILSVRC において、従来の SVM などを用いた画像認識に大差をつけて優勝し、深層学習の有効性を知らしめたモデル。5 層の畳み込み層と、3 層のプーリング層、3 層の全結合層からなる。活性化関数は ReLU。プーリングの際にフィルタサイズよりも小さなストライドを用いる Overlapping Pooling が用いられている。
  - ✓ 実装演習結果キャプチャと考察

    2_8_deep_convolution_net の実行結果。

    colab ではなく、CPU のみのローカル環境で実行したところ、学習に小一時間を要した。これまでの演習で扱ってきたモデルの中で、圧倒的に高い精度 0.982 を記録した。

```
Generation: 1000. 正答率(トレーニング) = 0.995
                : 1000. 正答率(テスト) = 0.982
```




✓ 確認テストなど、自身の考察結果

自身の調べものとして 2019 年に発表された EfficientNet について調べた。

EfficientNet は 2019 年に GoogleBrain から発表されたモデル。

この論文の主題は畳み込みニューラルネットワークの深さ・広さ・解像度が精度にどう影響するか調べ、Compound Coefficient という指標を導入することで性能を上げた、というもの。これによって作られた EffientNet-B7 は今間までの SoTA モデルよりも少ないパラメータと計算量で ImageNet における SoTA を達成した。

参考：https://qiita.com/omiita/items/83643f78baabfa210ab1#