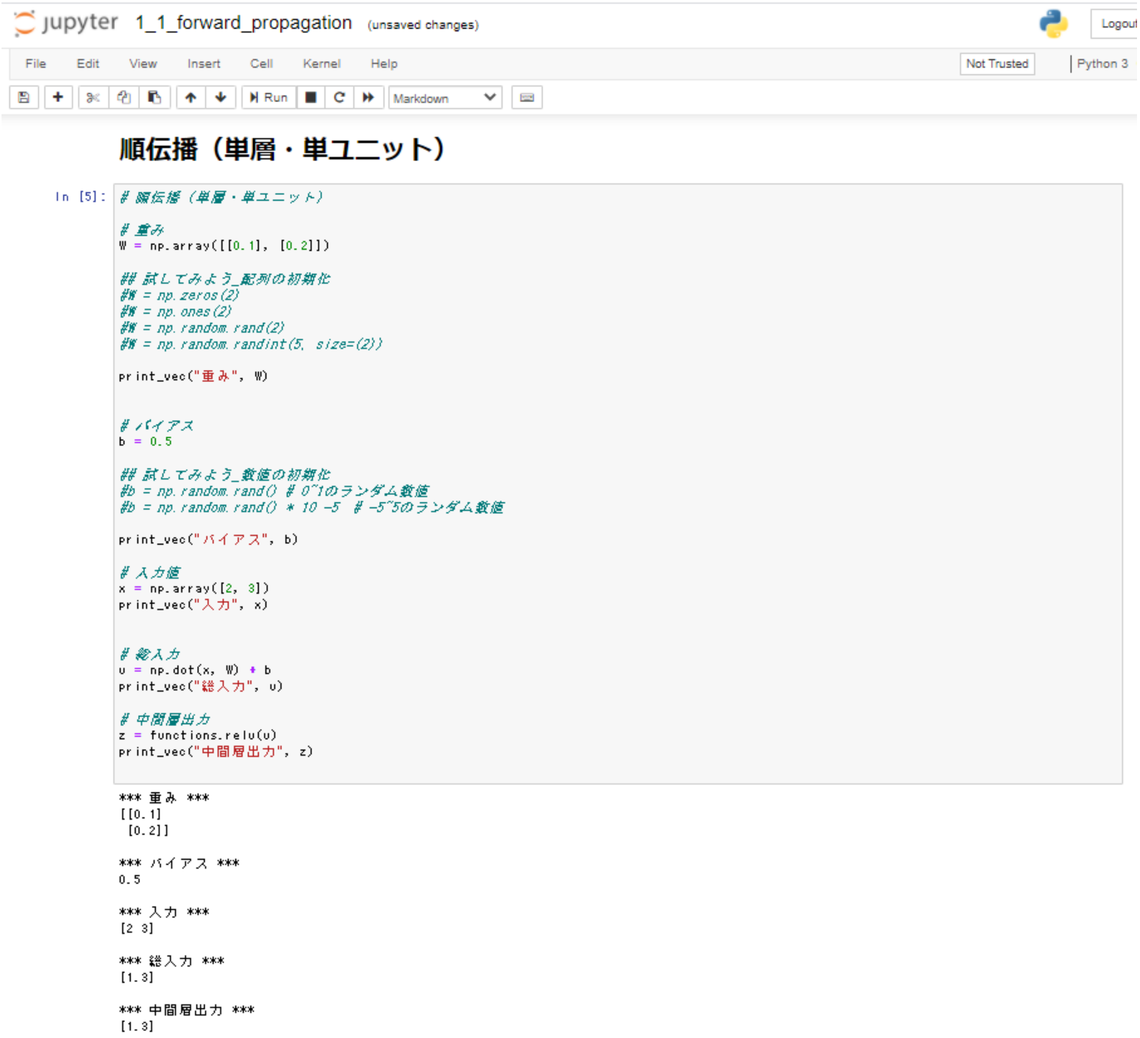
- 入力層～中間層
  - ✓ 要点のまとめ

    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



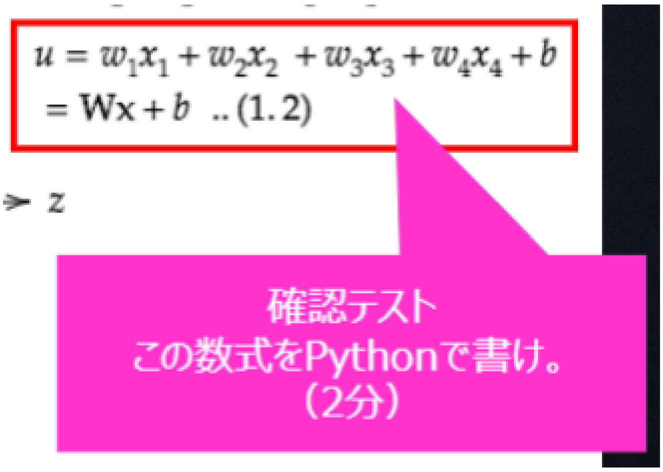
バイアスと重みの値をランダムに初期化した場合

順伝播（単層・単ユニット）

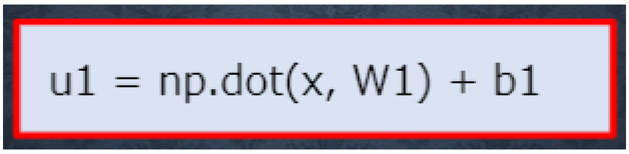
```
In [8]: # 順伝播（単層・単ユニット）

        # 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

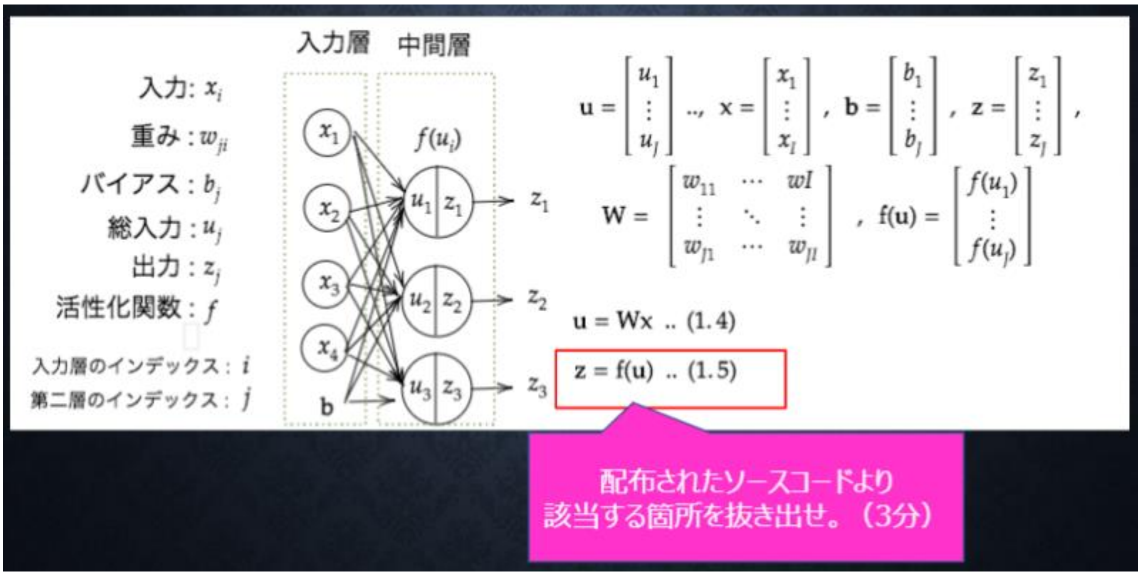
順伝播（単層・単ユニット）

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。

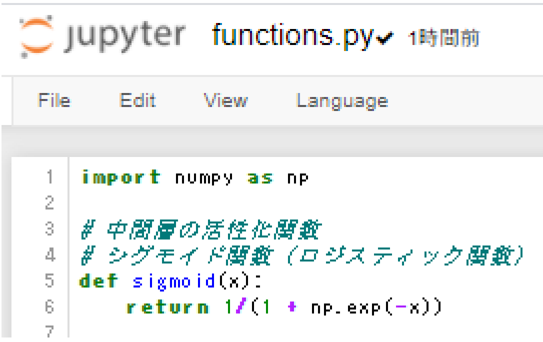
  - ✓ 確認テストなど、自身の考察結果



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。




```
import numpy as np

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

- 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）

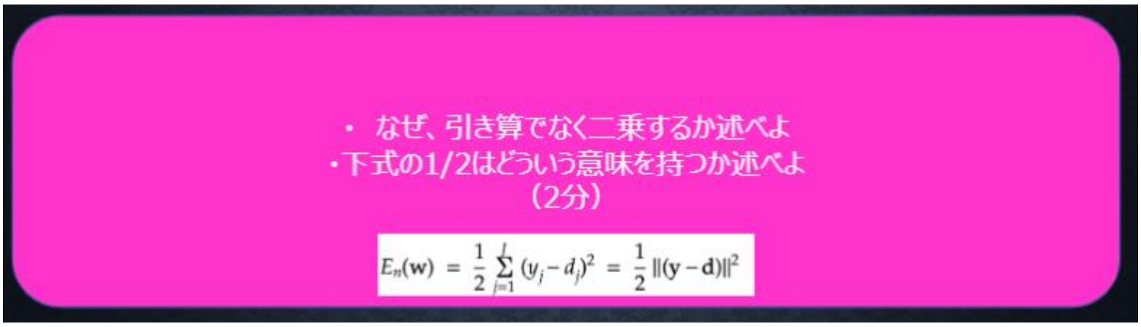
In [13]:
```
# 多クラス分類
# 2-3-4ネットワーク

# ！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 ***
0.32342029336019423
```

- ✓ 確認テストなど、自身の考察結果



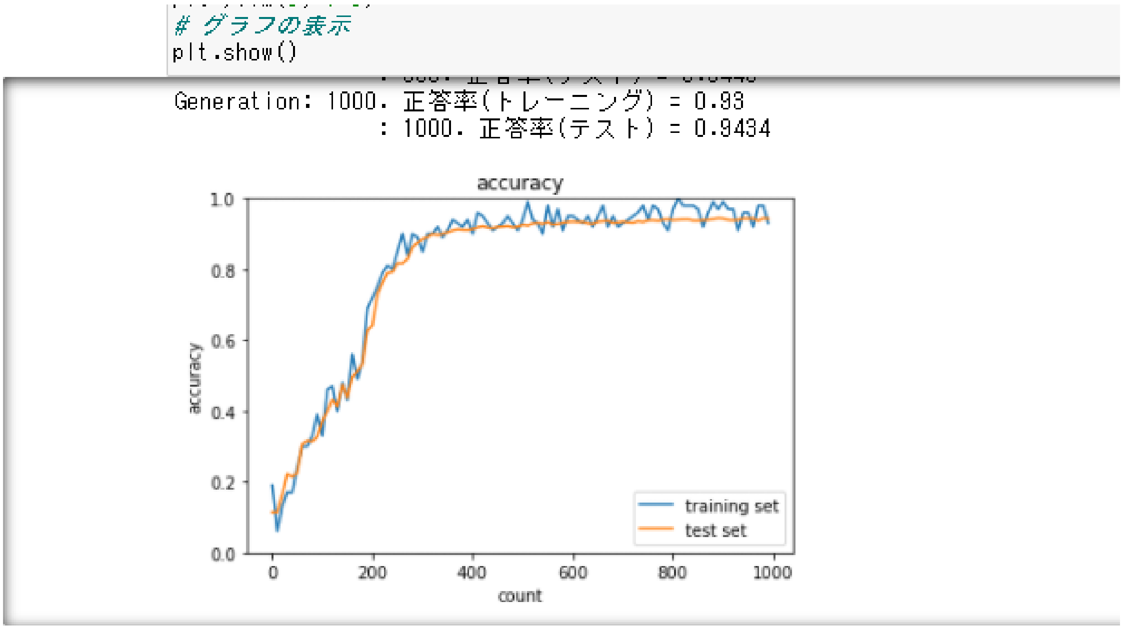
回答：

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

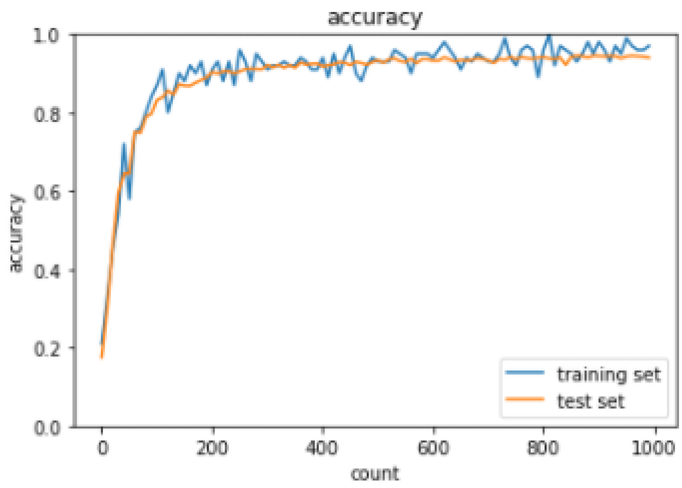


Adam で MNIST を分類した精度。まずはサンプルコードのまま。

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
```

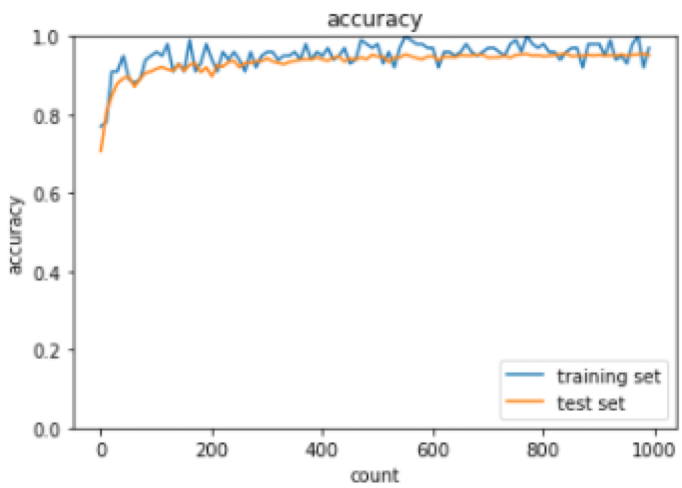


最終的な正答率には大きな差がないが、学習の収束が早くなっているのがわかる。

次に、活性化関数 ReLU のまま、He の初期値を用いる。

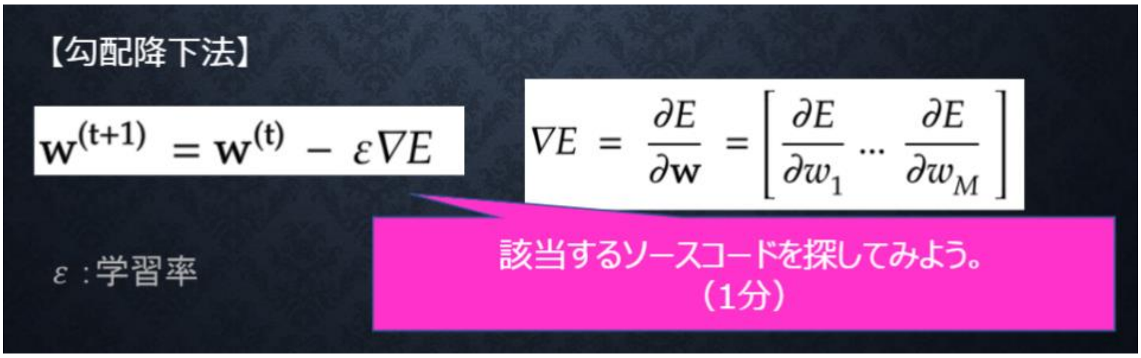
```
# グラフの表示
plt.show()
```

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

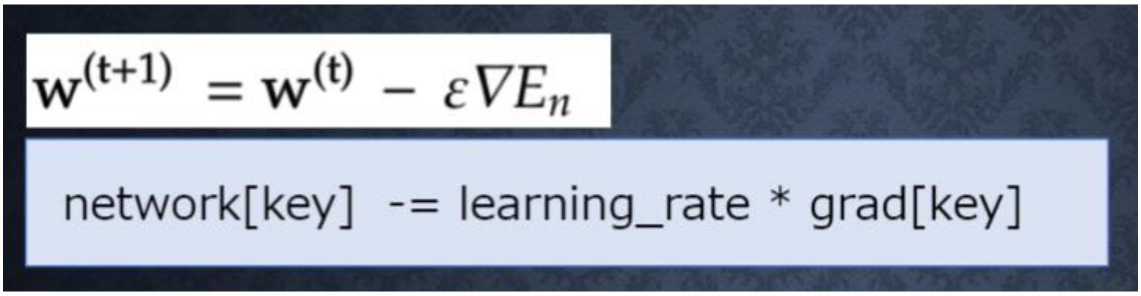


正答率は上がり、収束までの epoch 数も少なくなった。

- ✓ 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 ΛE 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

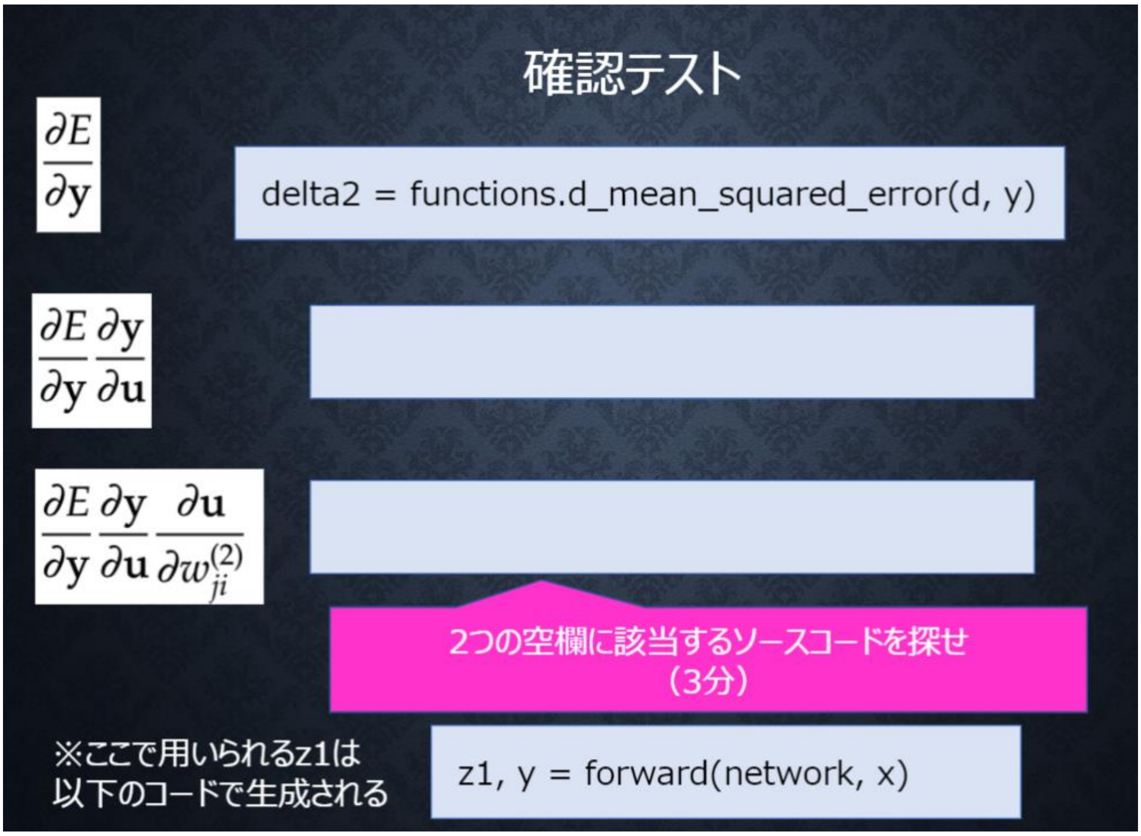
```
##### ネットワークの初期化 #####
*** 重み1 ***
[[0.1 0.3 0.5]
 [0.2 0.4 0.6]]

*** 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 ***
[[0.08706577 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]
```

誤差逆伝播の実行結果。

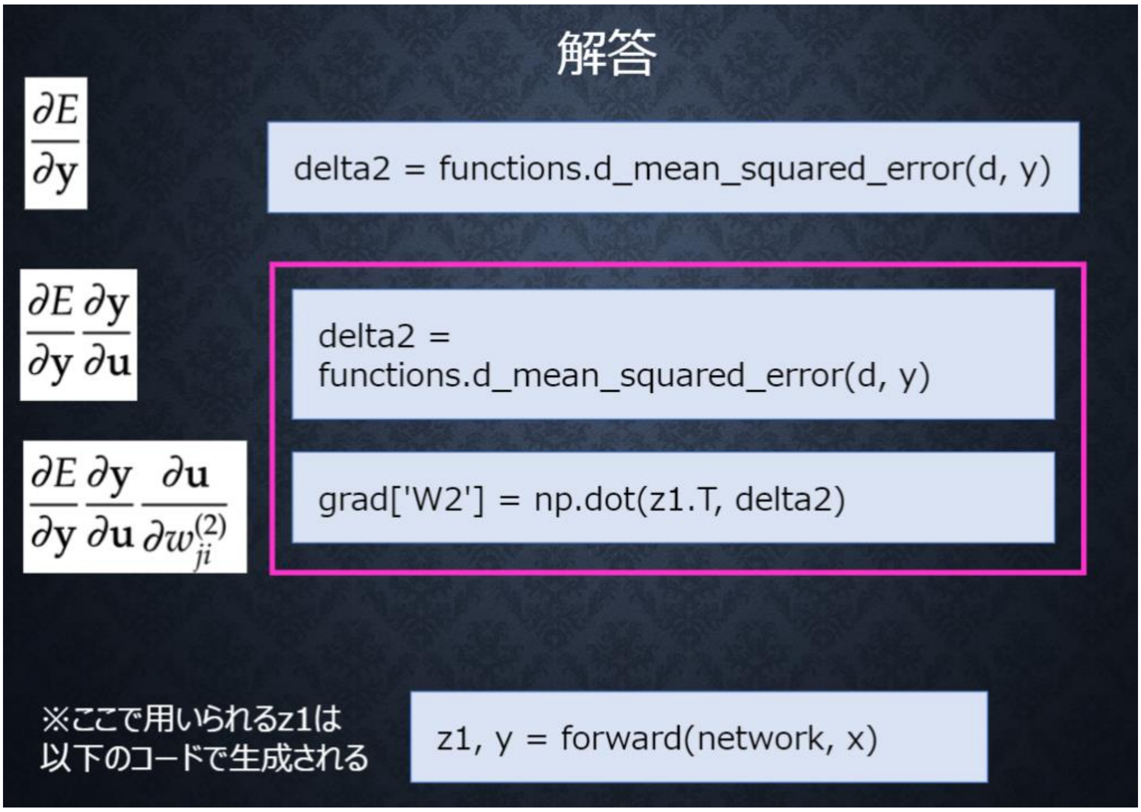
誤差逆伝播により重みのパラメータが調整されていることがわかる。

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
∂E/∂y
delta2 = functions.d_mean_squared_error(d, y)
∂E/∂y ∂y/∂u
∂E/∂y ∂y/∂u ∂u/∂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
∂E/∂y
delta2 = functions.d_mean_squared_error(d, y)
∂E/∂y ∂y/∂u
delta2 =
functions.d_mean_squared_error(d, y)
∂E/∂y ∂y/∂u ∂u/∂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]]
```