# 実装演習レポート（深層学習Day1&Day2）



owadatk edited this page 6 minutes ago · 10 revisions

# 深層学習day1

## Section 1: 入力層～中間層

### 要点のまとめ

- 入力から入ってきた数字に重みを乗じて中間層に渡す。その際にバイアスも追加する。総入力u = Wx + b で表現する。（Wとxは重みと入力を表す行列）
- 学習の目的は最適なWとxを求めること
- 総入力uを活性化関数に通して次の層へ渡す

### 実装演習

## 順伝播（単層・単ユニット）

```
# 順伝播（単層・単ユニット）

# 重み
W = np.array([[0.1], [0.2]])

## 試してみよう_配列の初期化
##W = np.zeros(2)
```

```
#W = np.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)
```

```
*** 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,)

*** バイアスh ***
```

```
*** 中間層出力 ***
[1.3]
shape: (1,)
```

```
0.5
shape: ()

バイアスb1 0.4736667655136967
バイアスb2 2.2198464080761635
```

- 配列の初期化
  - np.zeros(2) は2次元配列で要素はいずれも0
  - np.ones(2) は2次元配列で要素はいずれも1
  - np.random.rand(2) は2次元配列で要素は浮動小数点の乱数
  - np.random.randint(5, size=(2)) は2次元配列で要素は1～5までの整数
- 乱数の生成
  - random.rand() で0～1の乱数を生成することができる
  - print_vec でshapeを表示させようとするとエラーになる

## 順伝播（単層・複数ユニット）

```
# 順伝播（単層・複数ユニット）

# 重み
W = np.array([
    [0.1, 0.2, 0.3,0],
    [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=(3,4))

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)
```

```
*** 重み ***
[[4 0 3 1]
 [0 0 0 0]
 [2 1 4 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 ***
[ 9.1  0.2 14.3]
shape: (3,)

*** 中間層出力 ***
[0.99988835 0.549834   0.99999938]
shape: (3,)
```

- 配列のshapeを指定して初期化することができる
- Wとxのドット積をとる際、shapeが合っていないとエラーになるので、Wはsize=(3,4)で初期化する
- 活性化関数にsigmoidを使っているので、中間層出力は0～1の間の数値になる

# 多クラス分類（2-3-4ネットワーク）

```
# 多クラス分類
# 2-3-4ネットワーク

# ！試してみよう_ノードの構成を 3-5-6 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 = {}

    input_layer_size = 3
    hidden_layer_size= 5
    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
```

- ノードの構成を 3-5-6 に変更するには次のように初期値を変える
  - input_layer_size = 3 # 入力値の配列shapeが一致しているか確認
  - hidden_layer_size = 5

    - outpu_layer_size = 6
- 回帰のコード例でも同様
- 他クラス分類の誤差関数は交差エントロピーだが、回帰の誤差関数は二乗誤差MSEを使っている

## 2値分類（2-3-1ネットワーク）

```
# 2値分類
# 2-3-1ネットワーク

# ！試してみよう_ノードの構成を 5-10-20-1 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 ######")

    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
```

- ノードの構成を 5-10-20-1 に変更してみようとのことだが、既に 5-10-20-1 になっていた **★ここで'after'のファイルを見ていたことに気づく（⇒見ないでできるか後日再トライ）**

    - 入力は x = np.array([1.,2.,2.,4.,5.]) なので、5

- W1 shapeは(10,)なので入力から中間層1に渡されるのはノード数 10
- W2 shapeは(10,20)なので、次の中間層2に渡されるのはノード数 20
- W3 shapeは(20,1)なので、出力層に渡されるのはノード数 1

### 確認テスト等考察

- 問「DLは結局何をやろうとしているか2行以内で述べよ。また、次の中のどの値の最適化が最終目的か全て選べ①入力値②出力値③重み④バイアス⑤総入力⑥中間層入力⑦学習率」
  - 解答「明示的なプログラムの代わりに多数の中間層を持つNNを用いて、入力値から目的とする出力値に変換する数学モデルを構築すること」「最適化するのは重みとバイアス」
- 問「次のネットワークを紙にかけ。入力層：2ノード1層、中間層：3ノード2層、出力層：1ノード1層」
- 問「入力層から中間層への図式に動物分類の実例を入れてみよう」
- 問「NNを表す線形式u=Wx+bをPythonで書け」
  - u1 = np.dot(x, W1) + b1 # xとW1のドット積にバイアスb1を加える式
- 問「1-1のファイルから中間層の出力を定義しているソースを抜き出せ（3層の場合）」
  - z2 = functinos.relu(u2) # 活性化関数としてReLUを用いている

## Section 2: 活性化関数

### 要点のまとめ

- NN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。**（非線形であることが重要！）**
- 非線形の返還をすることでより変化に富んだ出力を作ることができる
- 活性化関数の種類（20種類ぐらいある）
  - 中間層用：ReLU関数、シグモイド（ロジスティック）関数、ステップ関数
  - 出力層用：ソフトマックス関数、恒等写像、シグモイド（ロジスティック）関数

- **ステップ関数**：0から1の動きが極端で線形分離可能な場合しか使えないので、現在は使われていない
- **シグモイド関数**：0から1の間の出力。なだらかに変化し、微分可能。勾配消失問題を起こすことがある。
- **ReLU関数**：いま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（モデルの中身がシンプルになる）に貢献することでよい結果を出す

## 実装演習

- 順伝播のコード例では、中間層の活性化関数にReLU、出力層にsoftmax関数を使っている。

```
# 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
```

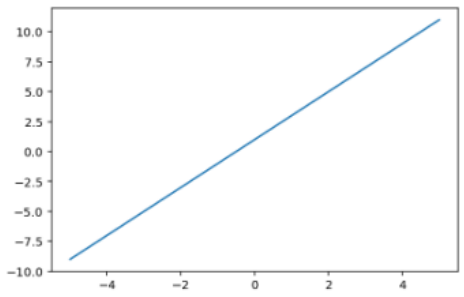
- 逆伝播のコード例では、中間層の活性化関数にReLU、主力層にシグモイドとクロスエントロピーの複合導関数（d_sigmoid_with_loss = d - y）を使っている。

```python
# 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
```

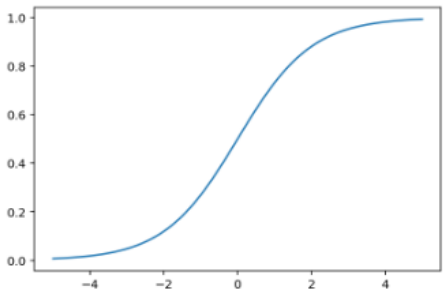
```python
# シグモイドとクロスエントロピーの複合導関数
def d_sigmoid_with_loss(d, y):
    return y - d
```

## 確認テスト等考察

- 問「線形と非線形の違いを図に書いて簡単に説明せよ」



線形な関数



非線形な関数

線形な関数は

- 加法性: $f(x+y) = f(x) + f(y)$
- 斉次性: $f(kx) = kf(x)$

を満たす

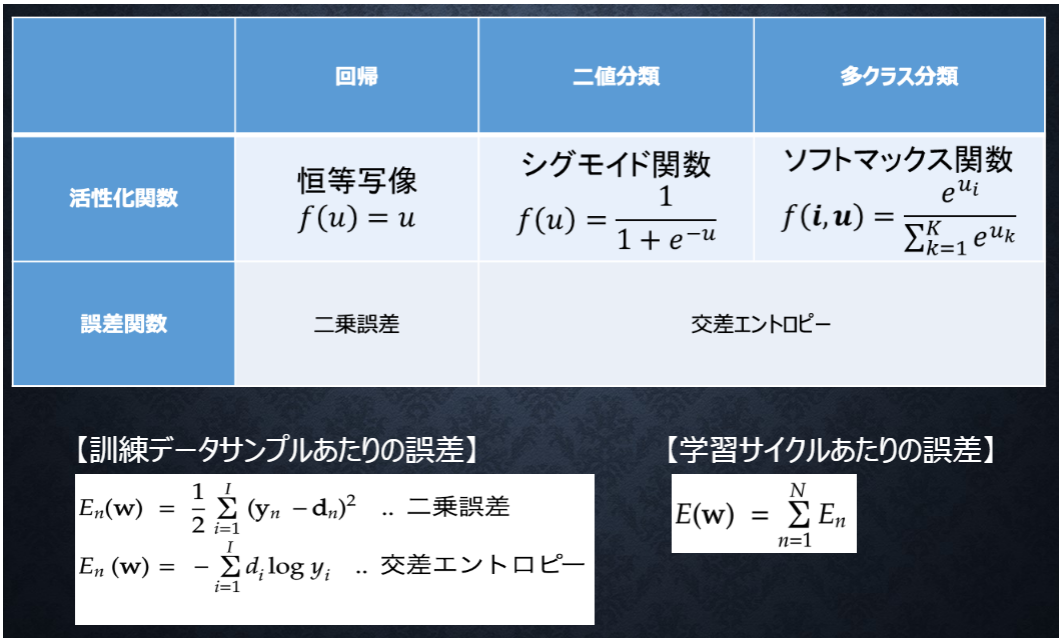
非線形な関数は加法性·斉次性を満たさない

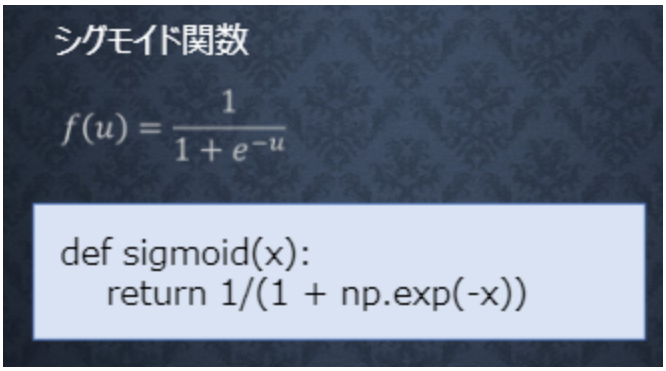
- 問「配布されたソースコードより該当する箇所を抜き出せ（順伝播3層・複数ユニット）」
- z1 = functions.relu(u1) と z2 = functions.relu(u2) の部分。functionsは別ファイルに定義されているので、それを参照すると処理の中身がわかる。
-
```
# ReLU関数
def relu(x):
    return np.maximum(0, x)
```

# Section 3: 出力層

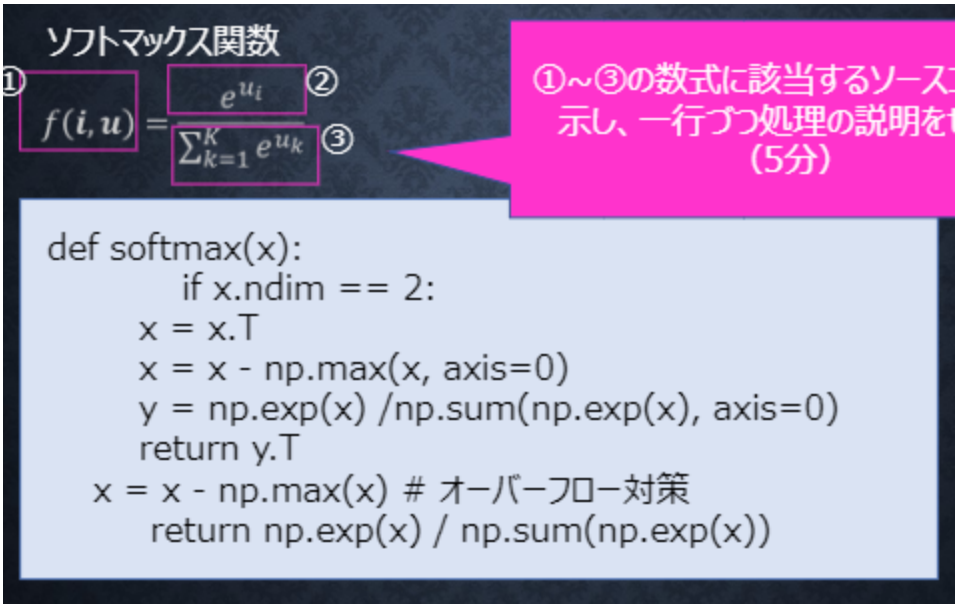
## 要点のまとめ

- 出力層の役割：中間層から受け取った数字を人間が欲しい形の出力に変換
- 誤差関数：NN全体をどうやって学習させるか？
  - 訓練データ使って、予測と正解との誤差を最小化するようにNNを学習させる
  - 誤差を表現するのが誤差関数
  - 解く問題によって使われる誤差関数はある程度決まっている。分類問題の場合はクロスエントロピー誤差を用いる。回帰問題の場合はMSE。
- 出力層の活性化関数
  - 出力層の活性化関数の目的は中間層と異なる

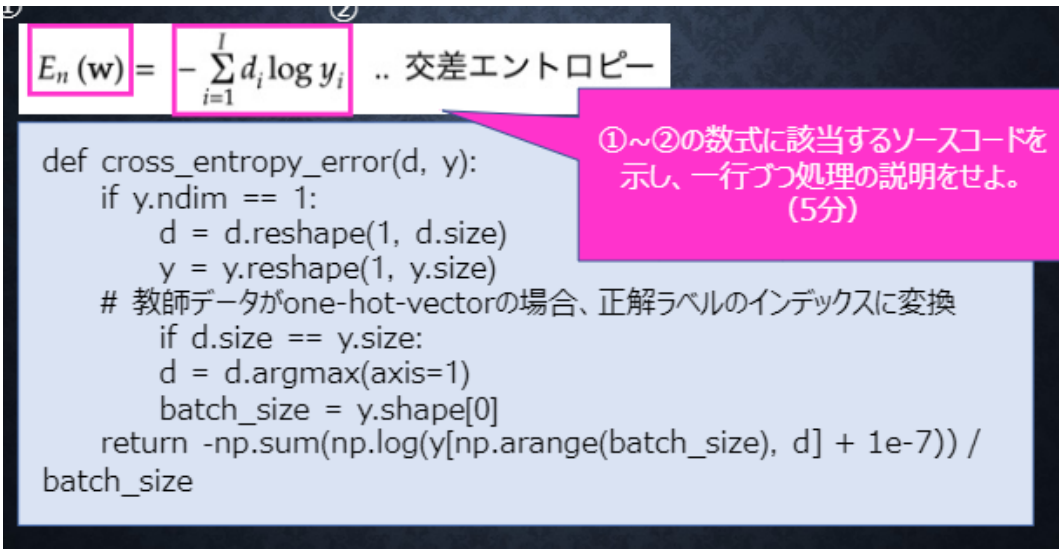
- 信号の大きさ（比率）はそのままに変換
- 分類問題の場合、出力層の出力は0～1の範囲に限定し、総和を1とする必要がある
- 

- シグモイド関数： 

- ソフトマックス関数：




- 交差エントロピー：




## 実装演習

- 順伝播のコード例では、中間層の活性化関数にReLU、出力層にsoftmax関数を使っている。

```
# 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
```

- 逆伝播のコード例では、中間層の活性化関数にReLU、主力層にシグモイドとクロスエントロピーの複合導関数（d_sigmoid_with_loss = d - y）を使っている。

```
# 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
```

```
# シグモイドとクロスエントロピーの複合導関数
def d_sigmoid_with_loss(d, y):
    return y - d
```

## 確認テスト等考察

- 問「なぜ引き算ではなく二乗するか。1/2はどういう意味をもつか」
  - 引き算を行うだけでは各ラベルでの誤差で正負両方の値が発生し、全体の誤差を正しく表すのに都合が悪い。二乗してそれぞれのラベルでの誤差を正の値になるようにする。
  - 1/2にするのは、実際にネットワークを学習するときに行う誤差逆伝播の計算で誤差関数の微分を用いる際の計算式を簡単にするため。本質的な意味はない。

- ソフトマックス関数について、数式に該当するソースコードを示し、一行づつ処理の説明をせよ

```python
# ソフトマックス関数
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))
```

- 本質的な処理は最後の1行。上のif分はミニバッチの際に利用（行列の処理をしている）
- 交差エントロピーに該当するソースコードを示し、一行づつ処理の開設をせよ

```python
# クロスエントロピー
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
```

- 本質的な処理は最後の一行の中、-np.sum(np.log(y[np.arange(batch_size), d] + 1e-7)) の部分
- 最後の 1e-7 はマイナス無限大に落ちないようにするためごく小さい値を足している

# Section 4: 勾配降下法

## 要点のまとめ

- 3種類の勾配降下法：勾配降下法、確率的勾配降下法、ミニバッチ勾配降下法
- パラメータを最適化する（誤差を最小化するパラメータを探索する）
- 学習率の値によって学習の効率が大きく異なる

    - 学習率ε 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ε が小さい場合、発散することはないが収束するまでに時間がかかってしまう
- 学習率の決定、収束性向上のためのアルゴリズムが複数利用されている
    - Momentum
    - AdaGrad
    - Adadelta
    - Adam # このアルゴリズムは非常によく使われる
- 確率的勾配降下法（SGD）
    - メリット：計算コストの削減、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
- ミニバッチ勾配降下法
    - ランダムに分割したデータの集合（ミニバッチ）を用いて学習（メモリに乗りきらないデータ量でも効率的に学習）
    - 非常に一般的な手法
    - メリット：確率的勾配降下法のメリットを損なわず、計算機の計算資源を有効利用できる（CPUを利用したスレッド並列化やGPUを利用したSIMD並列化）
- 誤差勾配の計算 - ∇Eをどのように計算するか？
    - 数値微分を使う：プログラムで微小な数値を生成し、疑似的に微分を計算する手法
    - 数値微分のデメリット：順伝播の計算を繰り返すため計算量が多くなってしまう。解決策が誤差逆伝播法

## 実装演習

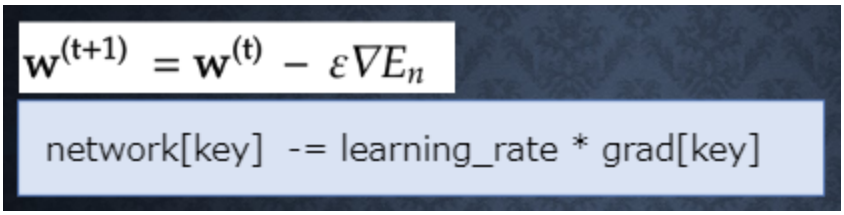
- NNのノード数を変えて試してみたが、ノード数と収束のスピードにはあまり関係がない（ノード数2でも早く収束する場合もある）
- 中間層の∇関数をReLUからシグモイドに変えると性能が落ちる

## 確認テスト等考察

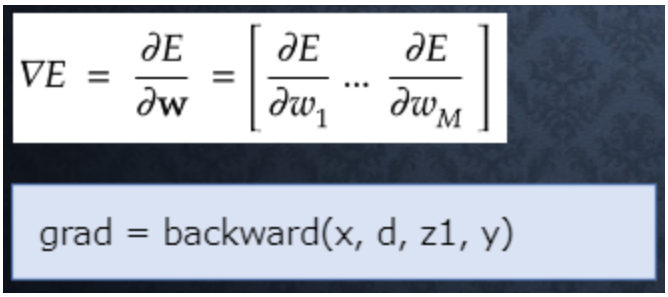
- 問「勾配降下法の数式に該当するコードを探してみよう」

```
# 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]
```

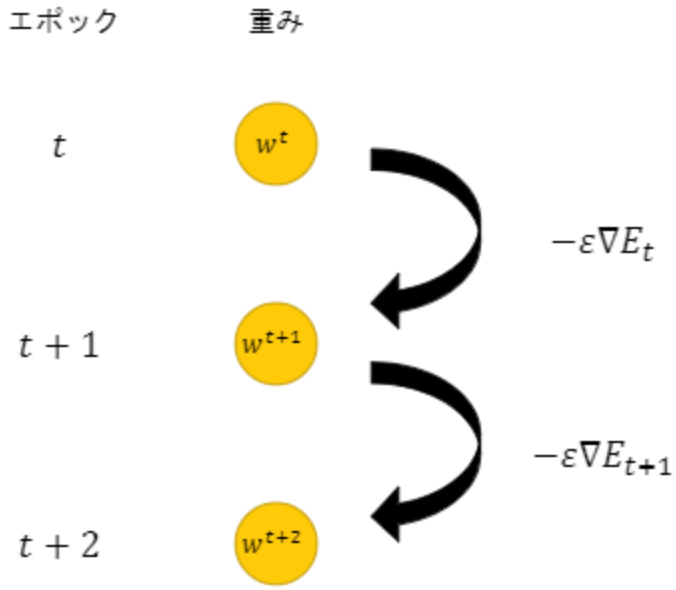
- wの更新：




- ▽Eの部分：




- 問「オンライン学習とは何か」
  - オンライン学習とは、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。
- 問「計算式の意味を図に書いて説明せよ」




## Section 5: 誤差逆伝播法

### 要点のまとめ

- 数値微分ではない方法で更新量▽Eを求める手法
- 算出された誤差を出力層側から順に微分し、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**計算する手法
- 微分の連鎖律を利用（一回計算した内容を次の計算で再利用できるので効率的）

### 実装演習

- 入力値を-5～5の範囲のランダム値にすると収束しない
- 学習率を大きくすると（例えば0.4）収束しない。逆に小さすぎても（例えば0.001）収束しない。

### 確認テスト等考察

- 問「誤差逆伝播法では不要な再帰的処理を避けることができる。既に行った計算結果を保持しているソースコードを抽出せよ」

```
    # 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)
```

  - delta2の使われ方に注目。一度計算されたdelta2が下の方のdelta1の中で再度利用されている
- 問「2つの空欄に該当するソースコードを探せ（1-3のファイル参照）」
  - dE/dy：delta2 = functions.d_mean_squared_error(d, y)
  - dE/dy * dy/du：delta1 = np.dot(delta2, W2.T) * functions.d_sigmoid(zl)
  - dE/dy * dy/du * du/dw：grad['W1'] = np.dot(x.T, delta1)

# 深層学習day2

## Section 1: 勾配消失問題

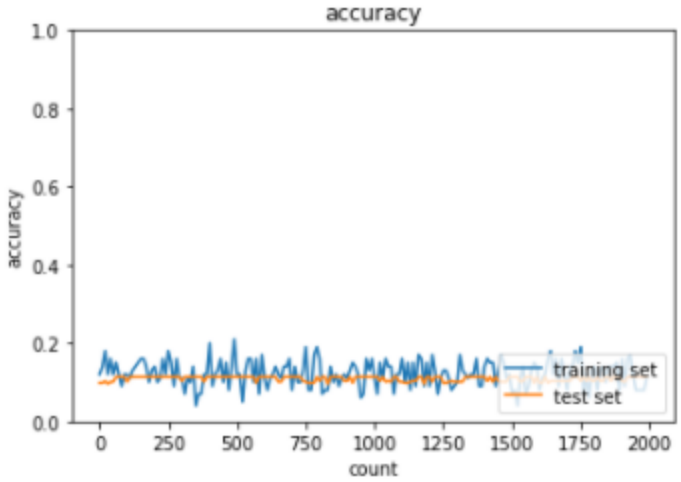
### 要点のまとめ

- 画像のような複雑な対象を処理するには中間層を増やす必要がある
- 中間中間層が増えると生じる問題：勾配消失問題（誤差逆伝播法が下位層に進んでいくにつれて勾配が緩やかになり、下位層のパラメータ更新がほとんど行われず訓練が最適値に収束しなくなる）
- なぜ生じるか：下位層ほど微分の連鎖律が長くなる。活性化関数にシグモイド関数を用いる場合、微分値の最大は0.25なため何度も掛け合わせると勾配消失問題を引き起こす
- どのように対処するか
  - **活性化関数の選択**：ReLU関数を使う（勾配消失問題の回避とスパース化に貢献（0未満の微分値は0なので、不要な重みを削減）することでよい成果をもたらしている
  - **重みの初期値設定**

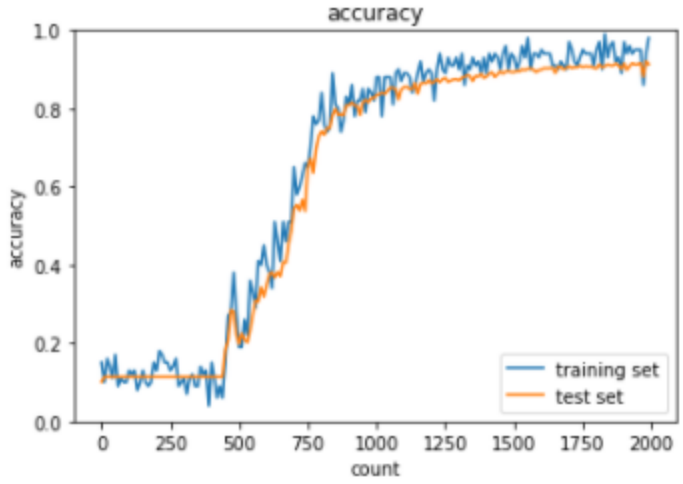
        - Xavierの初期値（重みの要素を前の層のノード数の平方根で除算した値）ReLU、シグモイド、tanh ⇒S字カーブの関数
        - Heの初期値（重みの要素を前の層のノード数の平方根で除算した値に対し√2を掛け合わせた値）ReLU
        - 何恺明 (Kaiming He) カィミン・フゥと読む？
    - **バッチ正規化**：ミニバッチ単位で入力値のデータの偏りを抑制する手法。活性化関数に値を渡す前後にバッチ正規化の処理を加える。
        - 中間層の重みの更新が安定化する。その結果学習が速く進む
        - 過学習を抑制する

## 実装演習

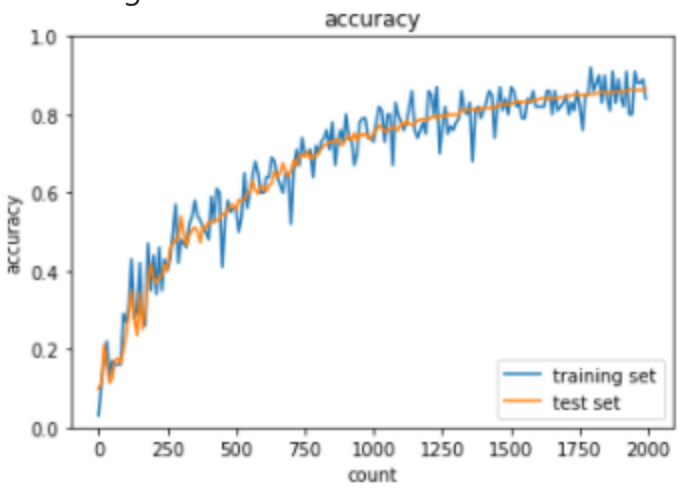
- 勾配消失 - sigmoid - gauss

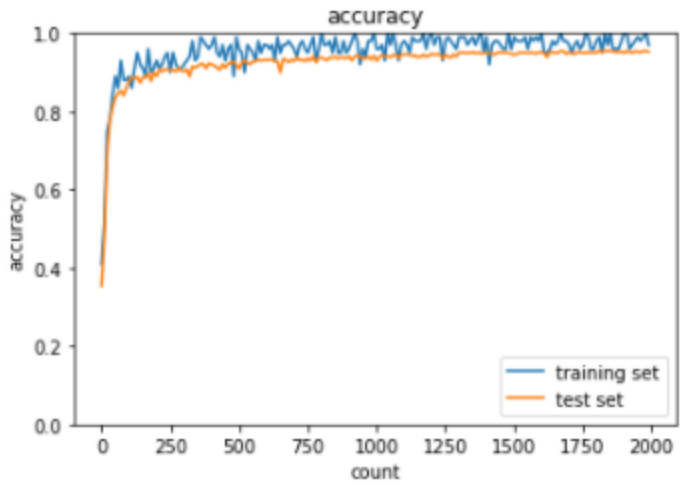



- 勾配消失 - ReLU - gauss

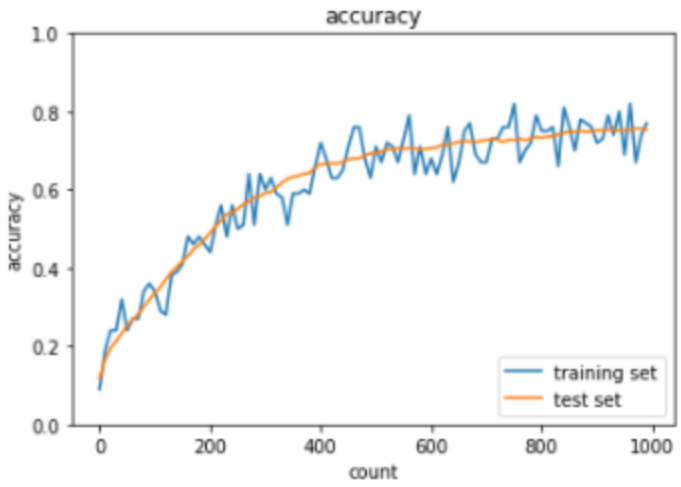


- 勾配消失 - sigmoid - Xavier




- 勾配消失 - ReLU - He




- バッチ正規化

```
40          if train_flg:
41              mu = x.mean(axis=0) # 平均
42              xc = x - mu # xをセンタリング
43              var = np.mean(xc**2, axis=0) # 分散
44              std = np.sqrt(var + 10e-7) # スケーリング
45              xn = xc / std
```

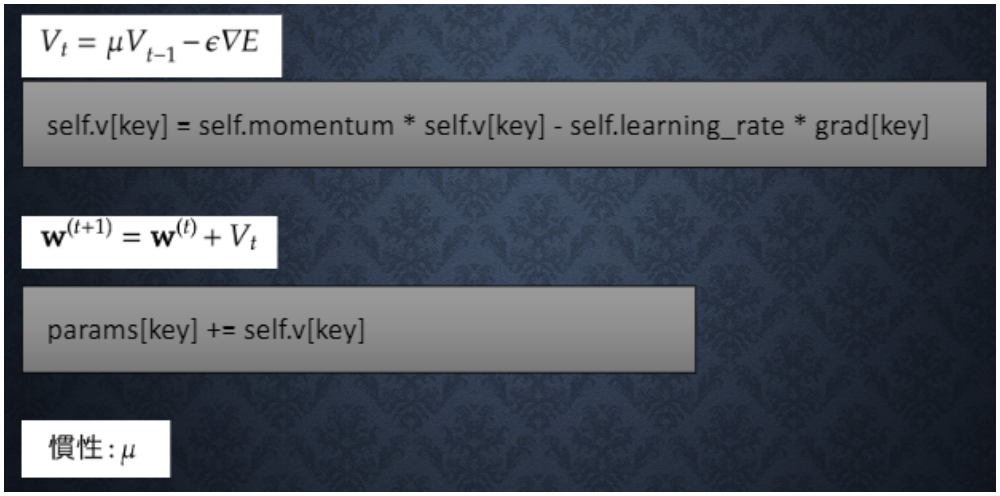
### 確認テスト等考察

- 連鎖律の原理を使いdz/dxを求めよ（連鎖律の計算の復習）
- シグモイド関数を微分したとき、入力値が0のときに最大値をとる。その値として正しいものを選択しから選べ（知識問題：0.25）
- 重みの初期値を0に設定するとどのような問題が発生するか。
  - 重みを0で初期化すると正しい学習が行えない。（すべての重みの値が均一に更新されるため、多数の重みをもつ意味がなくなる。）
- 一般的に考えられるバッチ正規化の効果を2点挙げよ。
  - 中間層の重みの更新が安定化する。その結果学習が速く進む
  - 過学習を抑制する
- E資格ではコードの穴埋め問題はよく出題される。解くのに時間がかかる場合があるので、解ける問題を先に解答してから取り組むとよい。

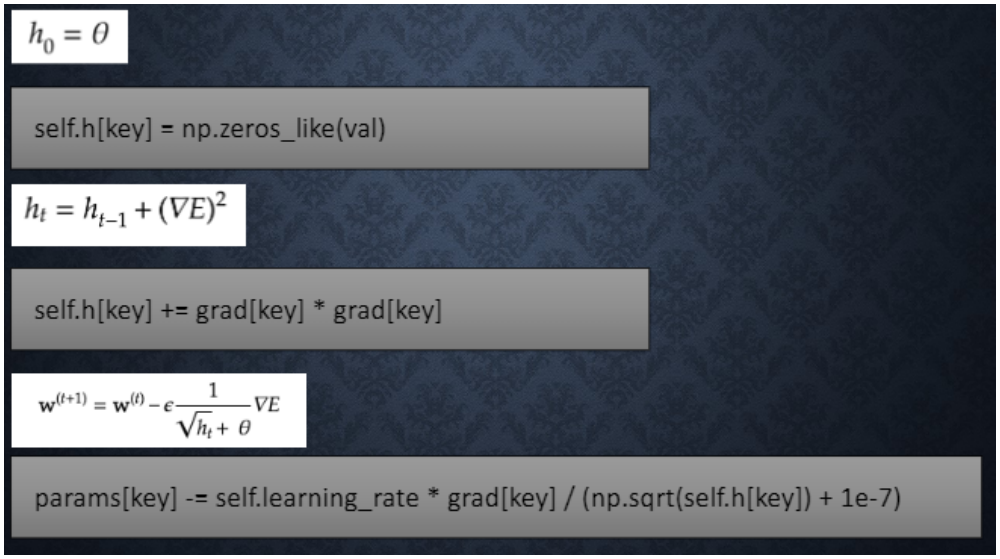
## Section 2: 学習率最適化手法

### 要点のまとめ

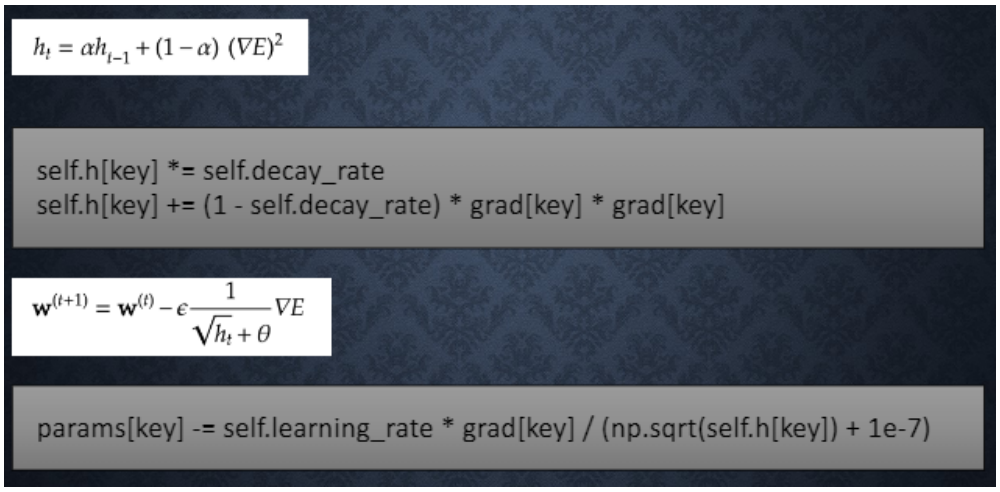
- 学習率が大きい場合：最適値にいつまでもたどり着かず発散してしまう

- 学習率が小さい場合：収束までに時間がかかる。大域的最適解に収束しづらくなる
- 学習率設定方法の指針：初期の学習率を大きく設定し、徐々に学習率を小さくしていく。パラメータごとに学習率を変化させる
- 学習率 ε を最適化する手法（Optimizer）
    - モメンタム
        - 誤差をパラメータで微分したものと学習率の積を減算した後、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
        - メリット：局所的最適解にならず、大域的最適解となる。谷間についてから最も低い位置に行くまでの時間が早い
        - 

    - AdaGrad
        - 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。計算した勾配を蓄積していく
        - 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
        - デメリット：大域的最適解にたどり着きにくい＝学習率が徐々に小さくなるので鞍点問題を引き起こすことがある




- RMSProp
  - 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。重みの更新はAdaGradと同じ式だが、経験を蓄積する度合いを調整できる。
  - メリット：局所的最適解に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

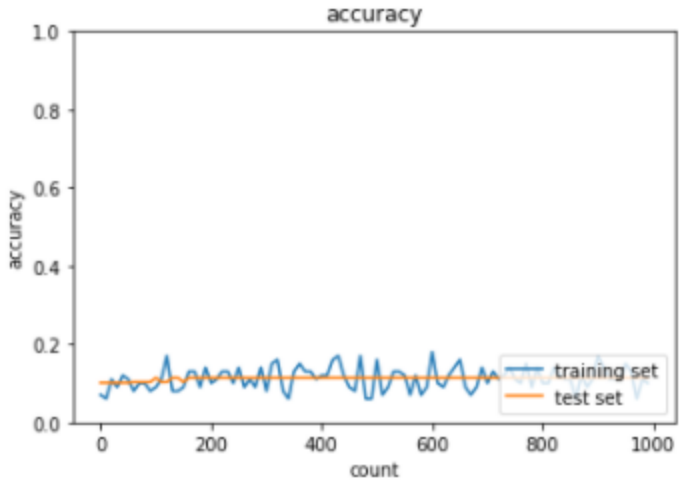



- Adam
  - モメンタムの過去の勾配の指数関数的減衰平均、RMSPropの過去の勾配の2乗の指数関数的減衰平均のそれぞれの要素を含んだ最適化アルゴリズム

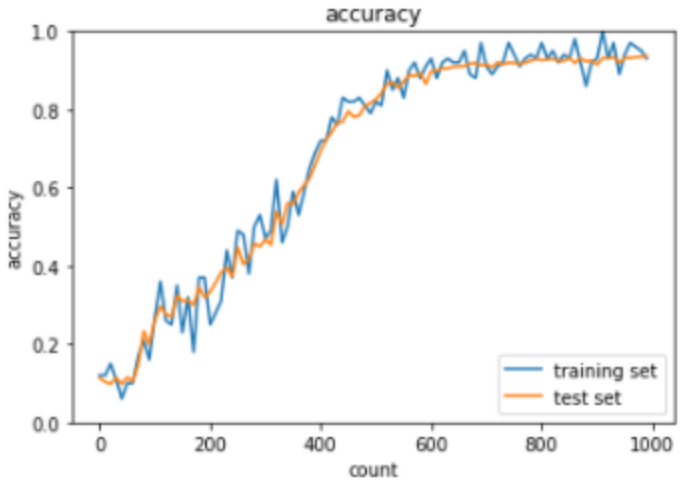
- メリット：モメンタムとRMSPropの両方のメリットを享受できる

## 実装演習

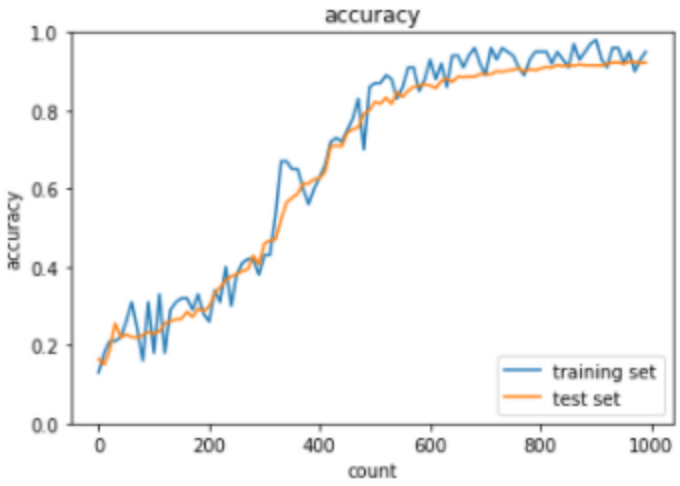
- SGD




- Momentum

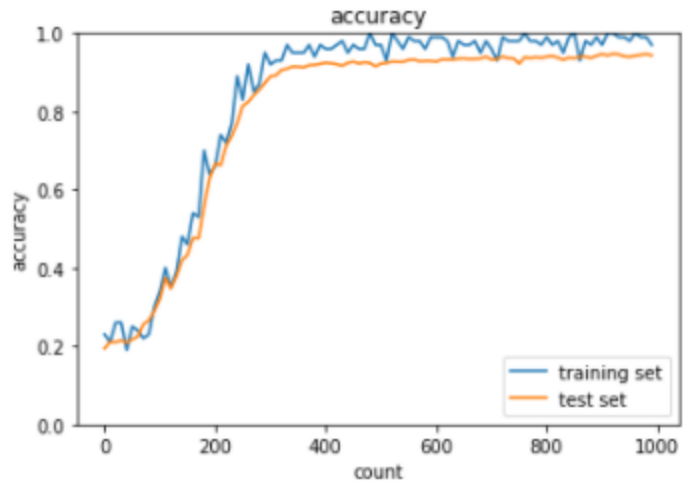



```
    # 勾配
    grad = network.gradient(x_batch, d_batch)
    if i == 0:
        v = {}
    for key in ('W1', 'W2', 'W3', 'b1', 'b2', 'b3'):
        if i == 0:
            v[key] = np.zeros_like(network.params[key])
        v[key] = momentum * v[key] - learning_rate * grad[key]
        network.params[key]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



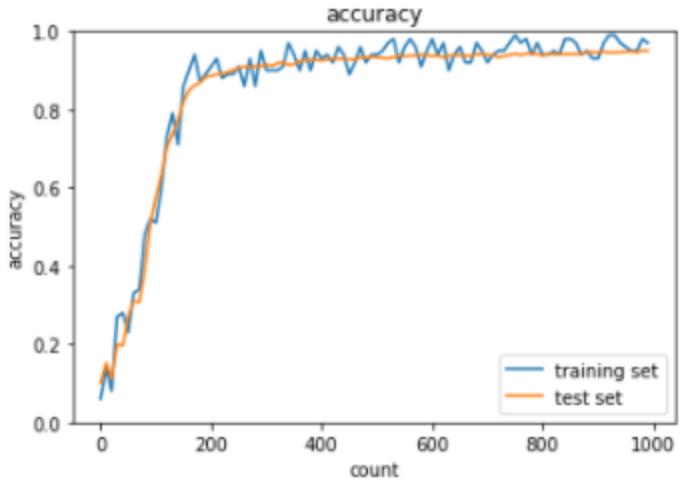

- AdaGrad

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




- RMSProp

```
    # 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)
```




- Adam

```
    # 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)

```

## 確認テスト等考察

- モメンタム・AdaGrad・RMSPropの特徴をそれぞれ簡潔に説明せよ。
  - モメンタム：勾配が同じ方向を向いている次元に向けて更新を増加させ、勾配が方向を変える次元に向けて更新を減少させる。そのため収束が早まり振動を抑える。
  - AdaGrad：稀なパラメータに対して大きな更新を、頻出のパラメータに対して小さな更新をするため、スパースなデータを扱うのに適している。手動で学習率を調整する必要がないという利点はあるものの、分母の二乗勾配が蓄積するため累積和は増加し続け、最終的にほとんど学習しなくなる。
  - RMSProp：AdaGradの急速に学習が低下する問題を解決する手法。

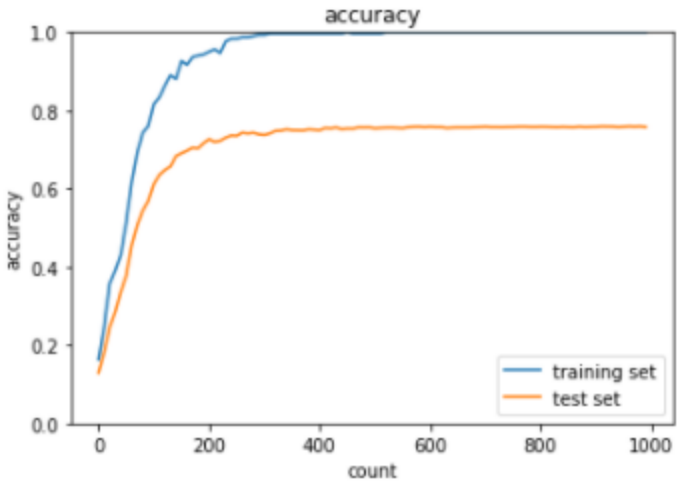
# Section 3: 過学習

## 要点のまとめ

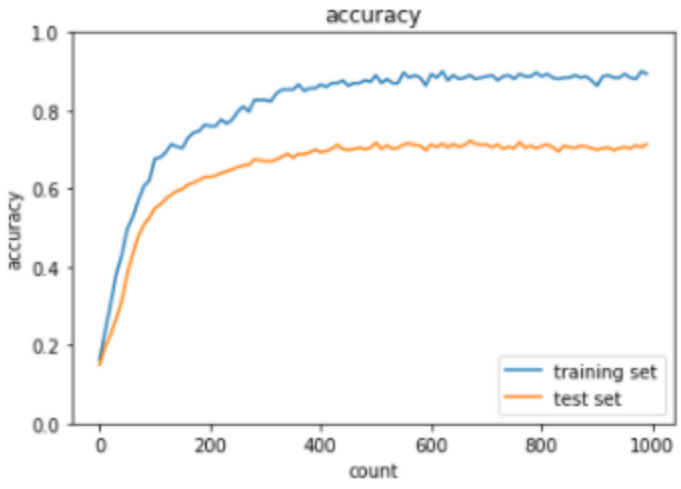
- 特定の訓練サンプルに対して特化して学習してしまうこと
- 原因：パラメータの数が多い、パラメータの値が適切でない、ノードが多いなど……**ネットワークの自由度が高い**
- 過学習を抑制する方法：正則化（NNの自由度を抑えること）
  - L1正則化（Lasso回帰、Pノルムが1　マンハッタン距離）
  - L2正則化（Ridge回帰、pノルムが2　ユークリッド距離）
  - ドロップアウト（ランダムにノードを削除して学習させる）
    - データ量を変化させずに異なるモデルを学習させていると解釈できる
    - データのバリエーションが大きいほど過学習は起きにくい

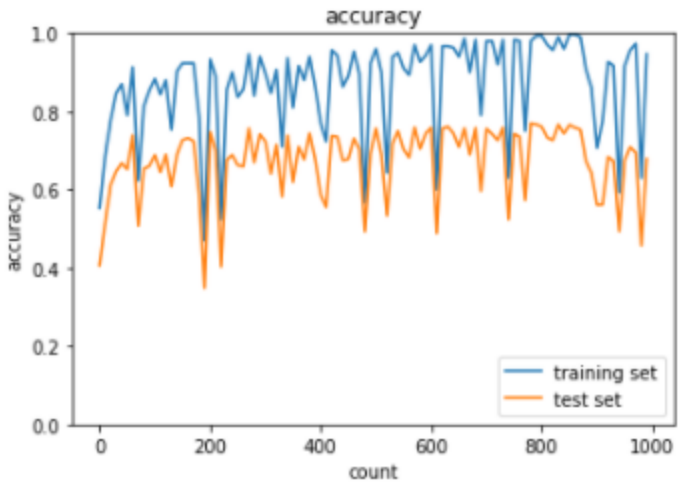
## 実装演習

- Overfitting

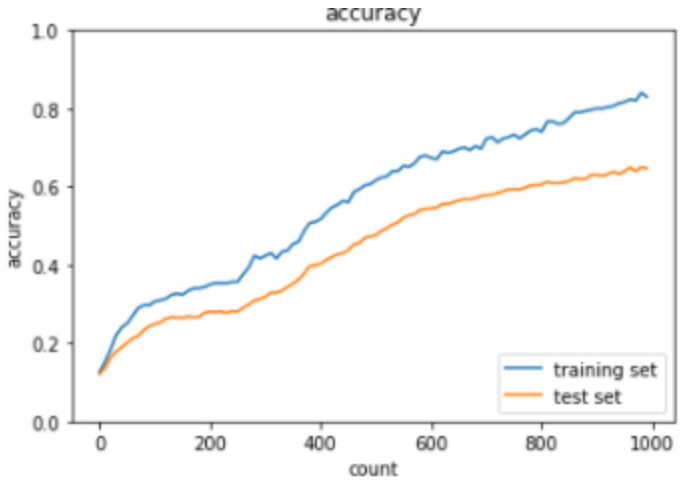



- L2

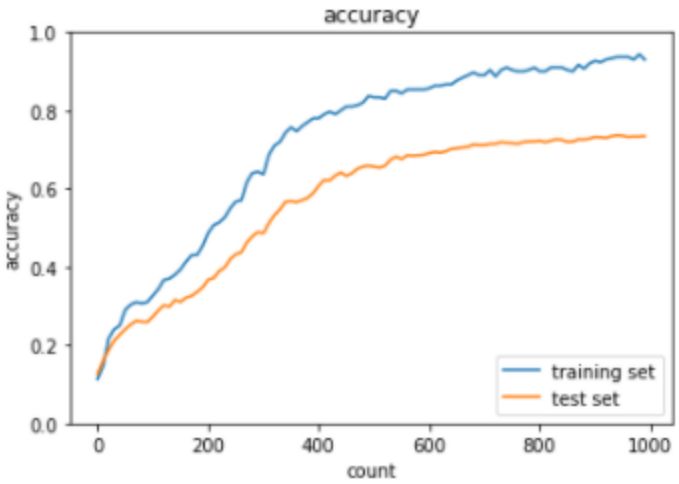





- L1




- Dropout




- Dropout + L1

## 確認テスト等考察

- 問「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」
  - 誤差関数の等高線のイメージとLasso推定量、Ridge推定量の等高線のイメージで考えるとわかりやすい
  - Lassoはスパース推定、Ridgeは縮小推定
  - Lasso推定量と誤差関数は軸上で交わる⇒ゼロになる
- 例題チャレンジ（コード穴埋め問題）
  - L2ノルムは||param||^2なので、微分は2*paramだが、係数2は正則化の係数に吸収されても変わらない
  - L1ノルムは|param|なので、微分はsign(param)　（signは符号関数）

# Section 4: 畳み込みニューラルネットワークの概念

## 要点のまとめ

- 画像の識別によく用いられるが、汎用性が高く適用できるデータは画像に限らない
- 次元間でつながりのあるデータを扱える（音声、CTスキャン画像、アニメのスケルトン、カラー画像、動画）
- CNNの構造図（入力層・畳み込み層・プーリング層・全結合層・出力層）
  - LeNet (32x32)の画像を最終的に10種類のアウトプットに変換する
  - 畳み込み層（フィルターを通した出力にバイアスを加えて、活性化関数を通して次に渡す）
  - 全結合層（次元のつながりを保ったまま特徴量を抽出する部分から入力を受け、人間が欲しい出力に変換する）
- 畳み込み層
  - 畳み込み層では3次元の空間情報も学習できる
  - フィルター、バイアス、パディング、ストライド、チャンネル
- 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⇒各チャンネルの間の関連性が学習に反映されない
- プーリング層
  - 重みはない
  - 対象領域のMax値または平均値を取得（max pooling, average pooling）

# 実装演習

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

        self.last_layer = layers.SoftmaxWithLoss()
```

- 高速演算の工夫：image to column

```
# im2colの処理確認
input_data = np.random.rand(2, 1, 4, 4)*100//1 # number, channel, height, widthを表す
print('========== input_data ===========¥n', input_data)
print('================================')
filter_h = 3
filter_w = 3
stride = 1
pad = 0
col = im2col(input_data, filter_h=filter_h, filter_w=filter_w, stride=stride, pad=pad)
print('============= col ==============¥n', col)
print('================================')
```

```
========== input_data ===========
 [[[[66. 96. 29. 78.]
   [73. 81. 87. 97.]
   [31. 63.  5. 17.]
   [78. 67. 55. 50.]]]


 [[[53. 52. 54. 27.]
   [10. 20. 59. 77.]
   [80. 93. 95. 68.]
   [11. 81. 52. 96.]]]]
================================
============= col ==============
 [[66. 96. 29. 73. 81. 87. 31. 63.  5.]
 [96. 29. 78. 81. 87. 97. 63.  5. 17.]
 [73. 81. 87. 31. 63.  5. 78. 67. 55.]
 [81. 87. 97. 63.  5. 17. 67. 55. 50.]
 [53. 52. 54. 10. 20. 59. 80. 93. 95.]
 [52. 54. 27. 20. 59. 77. 93. 95. 68.]
 [10. 20. 59. 80. 93. 95. 11. 81. 52.]
 [20. 59. 77. 93. 95. 68. 81. 52. 96.]]
================================
```

- プーリング層

```
        # 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
```

### 確認テスト等考察

- 問「サイズ6x6の入力画像を、サイズ2x2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」
  - 公式：O(H) = [（画像の高さ＋2パディング高 - フィルター高さ）／ストライド] +1　;O(W) =[（画像の幅＋2パディング幅 - フィルター幅）／ストライド] +1
  - 公式で解いてもよいし、図で考えてもよい

## Section 5: 最新のCNN（初期のNN）

### 要点のまとめ

- AlexNet
  - ImageNetを処理するためのCNN
  - 5層の畳み込み層及びプーリング層など、それに続く3層の全結合層から構成される
  - 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

### 実装演習

（実装演習なし）

## 確認テスト等考察

（確認テストなし）