

# 深層学習（前編1）

「Section1：入力層～中間層」

- ・NN全体図

入力層 … 1層

中間層 … 複数層

出力層 … 1層

- ・確認テスト DLとは？

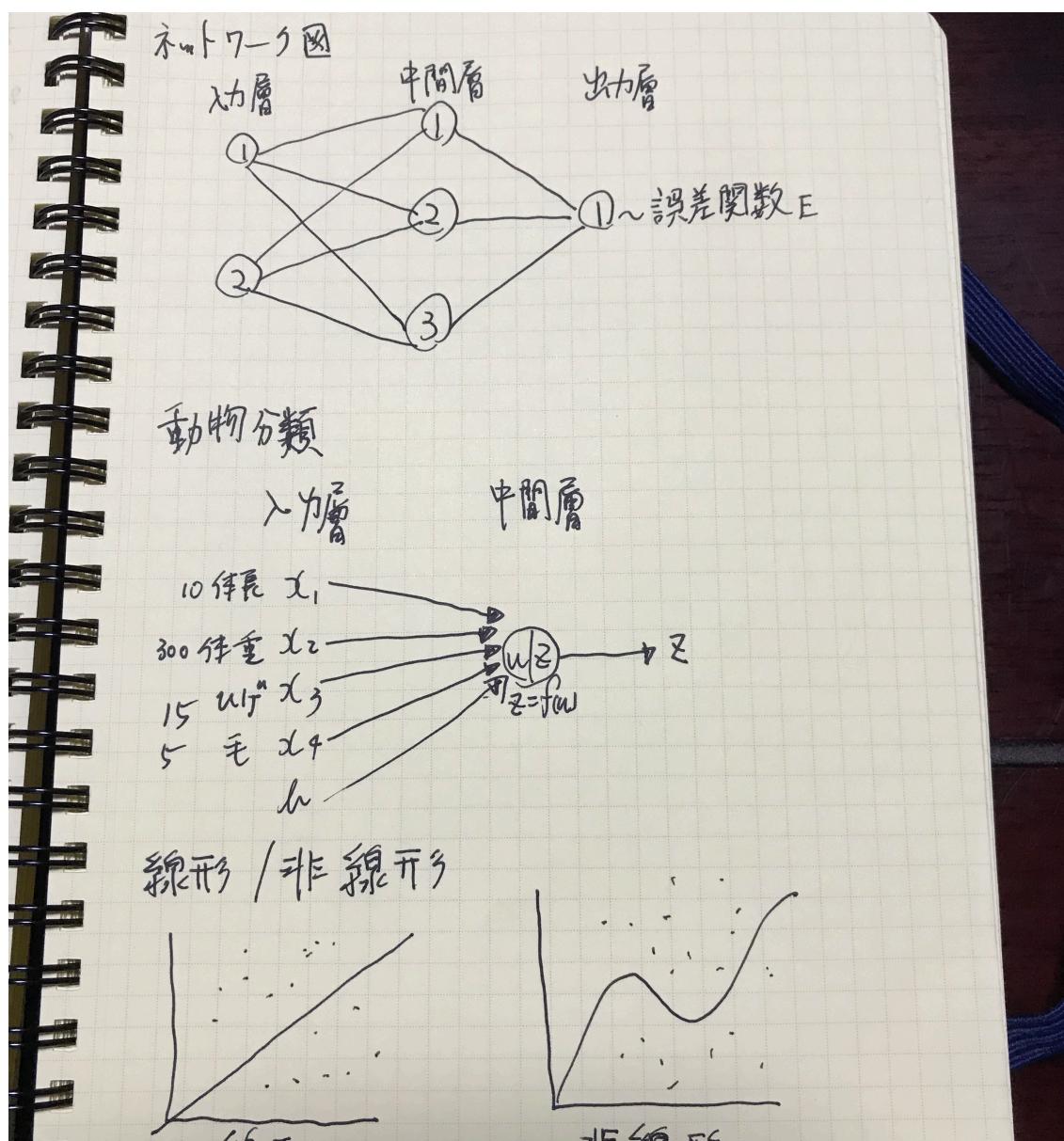
誤差を最小化するパラメータ（重み、バイアス）を探すこと、

この2個を最適化すること

- ・パラメータ：n個のw（重み）とn個のb（バイアス）

- ・活性化関数 > 中間層出力 > 総入力 > 出力

- ・確認テスト ネットワーク図（写真で）



- ・NNでできること

回帰～予測（売上、株価）、ランキング（人気、競馬予想）

分類～手書き認識、写真判別

実用例：株自動売買、チャットボット（コールセンター）、翻訳、音声認識、囲碁・将棋

## 「Section2：活性化関数」

- ・中間層で使用

- ・確認テスト 動物分類の入力層、中間層

**写真で**

- ・確認テスト 総入力の式をPythonで

`u1 = np.dot(x,w1) + b1`

- ・確認テスト 中間層の出力定義

`z = functions.relu(u)`

w,bのコメントアウト部分を外して実行すると出力結果が変わる

- ・確認テスト 線形、非線形の図

**写真で**

- ・活性化関数 次の層への出力の大きさを決める非線形関数

入力値で次の層への信号のon/offや強弱を定める

中間層用：ReLU, Sigmoid, Step(DLでは使わない)

出力層用：Softmax, Sigmoid, 恒等関数

- ・ステップ関数 閾値を超えたなら1、超えなかったら0を返す

線形分離できるものにしか対応できない、微細な表現ができない

- ・シグモイド関数 強弱を表現できる。課題は小数点で表現されるので勾配が消失してしまう。

0～1の連続値を返す

- ・ReLU関数 今一番使われている。勾配消失問題に対応。スペース化に貢献。

0より大きい値はそのまま伝えてゆく。強弱を伝えやすい。

\*シグモイド、ReLUをモデルによって使い分ける

- ・単層、複数ノード 中間層のノードが増えたパターン

ノードの数だけw,bが必要。情報伝播の強弱がより強くなるが、計算に時間がかかる。ノードを増やした分だけ正確なモデルになるわけではない。

- ・確認テスト 中間層  $z=f(u)$ に対応するソースコード

`z = functions.sigmoid(u)`

### 「Section3：出力層」

- ・中間層は人間が見てもわからない数値だが、出力層は人間にもわかる統計論理値が出力される
- ・算出した値から誤差を取る（誤差関数）
- ・確認テスト 2乗するのは二乗誤差を0にしないため

1/2するのは微分の計算を簡単にするため

- ・出力層の活性化関数

値の強弱の違いで中間層、出力層の関数を使い分ける

中間層は強弱をそのまま出力したい

出力層の出力は総和=1にする必要がある

	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 $f(u) = 1 / 1 + \exp(-h)$	Softmax $f(u) = \sum_{k=1}^n e^{u_k} / \sum_{k=1}^n e^{u_k}$
誤 差 関 数	二 乘 誤 差		交 差 エ ン ト ロ ピ ー

- ・シグモイド関数

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

- ・Softmax関数

SoftmaxはSigmoidの多項式（ベクトル）版

確認テスト Softmax関数の実装コード部分

```
# ソフトマックス関数
def softmax(x):
    if x.ndim == 2:
        x = x.T
        # 次元数が2次元だった場合
    x = x - np.max(x, axis=0)
        # 転置
        # 最大値を各要素から減算
    y = np.exp(x) / np.sum(np.exp(x), axis=0) # Softmax関数の公式、確率値算出
    return y.T
        # 転置

x = x - np.max(x) # オーバーフロー対策
return np.exp(x) / np.sum(np.exp(x))
```

- ・二乗誤差

```
# 平均二乗誤差
```

```

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

・交差エントロピー

確認テスト

# クロスエントロピー

def cross_entropy_error(d, y):
    # 教師データの型によって配列の形の変換を行う

    if y.ndim == 1:                                # 一次元
        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 #公式の処理部分
    *1e-7は微小な値を与えて0にならないようにしている

```

## 「Section4：勾配降下法」

- ・勾配下降法  $w(t+1) = w(t) - \varepsilon \nabla E_n$

全データに対して計算を行う。データが多い場合計算コストがかかる。

確認テスト 公式に該当するコード

```
network[key] -= learning_rate * grad[key]
```

- ・学習率 大きくしすぎると大域的極小解にいつまでも辿り着けない。歩幅が広い状態。

小さすぎると発散することはないが、収束するまでに時間がかかる。

- ・アルゴリズム Momentum, AdGrad, Adadelta, Adamなどがある

誤差関数の値をより小さくしてw,bを更新し、次の学習に反映

- ・確率的勾配下降法 (SGD)

ランダムに抽出したサンプルの誤差を求める

データが多い時の計算のコストを軽減できる

オンライン学習で使用できる

局所極小解に陥るリスクを権限できる

確認テスト オンライン学習とは？

新規の登録顧客などの、新しく入ってきたリアルタイムデータを  
学習に使用すること。

- ミニバッチ勾配下降法

ランダムに分割したデータの集合（ミニバッチ）をサンプルに平均誤差を求める

SGDのメリットに加え、計算資源を有効活用できる

確認テスト  $w(t+1) = w(t) - \epsilon \nabla E_n$  の意味の説明

$N_t = |D_t|$  ...サンプリング分割。tは分割する数。

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

- 誤差勾配の計算

数値微分...プログラムで微小な数値を生成し擬似的に微分する一般的な手法

各パラメータをそぞろに毎回算出するのは計算負荷が大きいので、

誤差逆伝播法を使用する

「Section5：誤差逆伝播法」

- 算出された誤差を出力層から順に微分し、前の層更に前の層へ伝播してゆく。

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計算できる。

確認テスト ソースコードで計算結果を保持している部分

```
# 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)
```

・計算方法

$$E(\mathbf{y}) = \frac{1}{2} \sum_{j=1}^J (y_j - d_j)^2 = \frac{1}{2} \|\mathbf{y} - \mathbf{d}\|^2 \quad : \text{誤差関数} = \text{二乗誤差関数}$$

$$\mathbf{y} = \mathbf{u}^{(L)} \quad : \text{出力層の活性化関数} = \text{恒等写像}$$

$$\mathbf{u}^{(l)} = \mathbf{w}^{(l)} \mathbf{z}^{(l-1)} + \mathbf{b}^{(l)} \quad : \text{総入力の計算}$$

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

確認テスト ReLUからシグモイド関数に中間層を変える

# 順伝播

`z1 = functions.relu(u1)`

を

`z1 = functions.sigmoid(u1)`

# 誤差逆伝播

# 中間層でのデルタ

`delta1 = np.dot(delta2, W2.T) * functions.d_relu(z1)`

を

`delta1 = np.dot(delta2, W2.T) * functions.d_sigmoid(z1)`

ReLUの結果のほうが、シグモイド関数を使ったものよりばらつきが少ない  
活性化関数が変わると結果が変わることに注意

確認テスト 公式を実装コードに当てはめる

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

```
delta2 = functions.d_sigmoid_with_loss(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)
```

## 深層学習（前編2）

### 「Section1：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、勾配がゆるやかになっていく。

下位層のパラメータはほとんど変わらなくなり、訓練は最適値に収束しなくなる。

シグモイド関数は出力の変化が微小なので勾配消失問題を引き起こす。

解決法：

- 活性化関数の選択
- 重みの初期値設定
- バッチ正規化

- 確認テスト シグモイド関数を微分した時、入力値が0のときに最大値をとる。

値として正しいものは？

$$f'(u) = (1 - \text{sigmoid}(u)) \times \text{sigmoid}(u) = (1 - 0.5) \times 0.5 = 0.25$$

- 活性化関数

ReLU関数(今最も使用されている関数)を使う

def relu(x):

```
return np.maximum(0,x)
```

$$f(x) = x \ (x > 0) \text{ or } 0 \ (x \leq 0)$$

- 重みの初期値設定

Xavier～前の層のノード数の平方根で除算した値( $1/\sqrt{n}$ )。

シグモイド関数と組み合わせて使用。

He～ 前の層のノード数の平方根で除算した値に対し $\sqrt{2}$ を掛け合せた値( $\sqrt{2}/n$ )。

ReLU関数と組み合わせて使用する。

- 確認テスト 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？

答：全ての値が同じ値で伝わるためにパラメータのチューニングが

行われなくなる

- 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。

活性化関数の前にバッチ正規化の処理を含んだ層を加える。

- 確認テスト バッチ正規化の効果をあげよ

- 計算の高速化
- 勾配消失が起きづらくなる

- バッチ正規化の数学的記述

- ミニバッチ全体の平均をとる
- ミニバッチ全体の標準偏差を取得

### 3. ミニバッチのインデックス値とスケーリングの積にシフトを加算

- ・確認 シグモイド関数を使用。勾配消失が起きやすい。

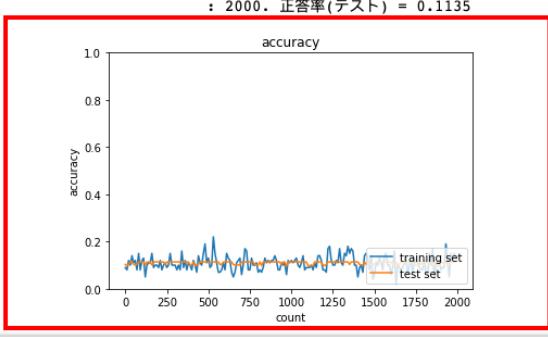
jupyter 2\_2\_1\_vanishing\_gradient Last Checkpoint: 11/24/2018 (autosaved)

File Edit View Insert Cell Kernel Widgets Help

Not Trusted Python 3

```
plt.title("accuracy")
plt.xlabel("count")
plt.ylabel("accuracy")
plt.ylim(0, 1.0)
# グラフの表示
plt.show()

Generation: 2000. 正答率(トレーニング) = 0.13
: 2000. 正答率(テスト) = 0.1135
```



### ReLU - gauss

```
In [3]: 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 data.mnist import load_mnist
from PIL import Image
import pickle
```

- ・確認 ReLU関数 勾配消失が起きづらい

jupyter 2\_2\_1\_vanishing\_gradient Last Checkpoint: 11/24/2018 (autosaved)

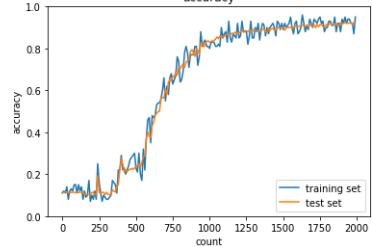
File Edit View Insert Cell Kernel Widgets Help

Not Trusted Python 3

```
for key in ('W1', 'W2', 'W3', 'b1', 'b2', 'b3'):
    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("accuracy")
plt.xlabel("count")
plt.ylabel("accuracy")
plt.ylim(0, 1.0)
# グラフの表示
plt.show()

Generation: 2000. 正答率(トレーニング) = 0.95
: 2000. 正答率(テスト) = 0.9248
```



- ・確認 Sigmoid \* Xavier 勾配消失していない

```
# パラメータに勾配適用
for key in ('W1', 'W2', 'W3', 'b1', 'b2', 'b3'):
    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("accuracy")
plt.xlabel("count")
plt.ylabel("accuracy")
plt.ylim(0, 1.0)
# グラフの表示
plt.show()

Generation: 2000. 正答率(トレーニング) = 0.83
: 2000. 正答率(テスト) = 0.8176
```

- ・確認 ReLU \* He ReLU単独よりも学習が進んでいる

```
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()

Generation: 2000. 正答率(トレーニング) = 0.98
: 2000. 正答率(テスト) = 0.953
```

・確認 Sigmoid \* He (このパターンで上手く学習が進むケースもある)

Jupyter 2\_2\_1\_vanishing\_gradient Last Checkpoint: 11/24/2018 (autosaved)

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3 Logout

```
plt.title("accuracy")
plt.xlabel("count")
plt.ylabel("accuracy")
plt.ylim(0, 1.0)
# グラフの表示
plt.show()
```

Generation: 2000. 正答率(トレーニング) = 0.84  
: 2000. 正答率(テスト) = 0.8112

accuracy

training set

test set

ReLU - He

In [5]:

```
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 data.mnist import load_mnist
from PIL import Image
import pickle
```

・確認 ReLU \* Xavier (このパターンで上手く学習が進むケースもある)

Jupyter 2\_2\_1\_vanishing\_gradient Last Checkpoint: 11/24/2018 (unsaved changes)

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3 Logout

```
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()
```

Generation: 2000. 正答率(トレーニング) = 0.97  
: 2000. 正答率(テスト) = 0.9481

accuracy

training set

test set

In [ ]:

- ・多角的に全体を把握

データの取得をどうするか？

プロダクトを作つてデータを集め。

法的に現在日本はルーズなので今のうちにデータを集めておくことはよい。

## 「Section2：学習率最適化手」

- ・パラメータは勾配加工法で最適化するが、学習率が大きいと発散する。

小さいと収束に時間がかかり、大域的局所解に収まらない。

- ・学習率の決め方

初期の学習率は大きく設定し、徐々に小さくしてゆく。

パラメータ毎に学習率を可変させる。

学習率最適化手法を利用

1. Momentum
2. AdaGrad
3. RMSProp
4. Adam

- ・Momentum

誤差をパラメータで微分したものを学習率の積を減算した後、前回の重みを減算した値と慣性の積を加算する。

$$\boxed{\begin{aligned} V_t &= \mu V_{t-1} - \epsilon \nabla E \\ \mathbf{w}^{(t+1)} &= \mathbf{w}^{(t)} + V_t \end{aligned}}$$

メリット：局所解にはならず大域的最適解となる。

谷間についてから最適値に行くまでの時間が早い。

実装：

```
v[key] = np.zeros_like(network.params[key])
v[key] = momentum * v[key] - learning_rate * grad[key]
network.params[key] += v[key]
```

- ・確認テスト それぞれの特徴

Momentum～谷間から最適解が早い

AdaGrad～ゆるやかな斜面でも収束が早い

RMSProp～パラメータの調整が少なくてすむ

- ・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

$$\begin{aligned} h_0 &= \theta \\ h_t &= h_{t-1} + (\nabla E)^2 \\ \mathbf{w}^{(t+1)} &= \mathbf{w}^{(t)} - \epsilon \frac{1}{\sqrt{h_t + \theta}} \nabla E \end{aligned}$$

メリット：勾配の緩やかな斜面に対して最適解につかづくのが早い

課題：次元が大きくなると学習率が徐々に小さくなるので鞍点に陥りやすい

実装：

```
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

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

$$\begin{aligned} h_t &= \alpha h_{t-1} + (1 - \alpha) (\nabla E)^2 \\ \mathbf{w}^{(t+1)} &= \mathbf{w}^{(t)} - \epsilon \frac{1}{\sqrt{h_t + \theta}} \nabla E \end{aligned}$$

メリット：局所解にはならず最適解に収束

ハイパーパラメータの調整が少なくて済む

実装：

```
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

モメンタムの前回の勾配の指数関数的に減衰平均させる

RMSPropの前回の勾配2乗を指数関数的に減衰平均させる

メリット：両方の良い部分を持っている

実装：

```
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)
```

- Jupiter演習

## 1. SGD 学習率を変えて (0.08) 実行

jupyter 2\_4\_optimizer Last Checkpoint: 11/24/2018 (unsaved changes)

File Edit View Insert Cell Kernel Widgets Help

Not Trusted Python 3

```
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()
```

Generation: 1000. 正答率(トレーニング) = 0.96  
: 1000. 正答率(テスト) = 0.8975

## Momentum

## 2. Momentum 実務では0.5 - 0.9で設定することが多い 0.5で実行

jupyter 2\_4\_optimizer Last Checkpoint: 11/24/2018 (unsaved changes)

File Edit View Insert Cell Kernel Widgets Help

Not Trusted Python 3

```
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()
```

Generation: 1000. 正答率(トレーニング) = 0.89  
: 1000. 正答率(テスト) = 0.9042

## MomentumをもとにAdaGradを作ってみよう

### 3. AdaGrad

jupyter 2\_4\_optimizer Last Checkpoint: 11/24/2018 (autosaved)

File Edit View Insert Cell Kernel Widgets Help Not Trusted Python 3 Logout

```
# 変更しよう
# =====
# if i == 0:
#     h[key] = np.zeros_like(network.params[key])
#     h[key] = momentum * h[key] - learning_rate * grad[key]
#     network.params[key] += h[key]
if i == 0:
    h[key] = np.full_like(network.params[key], 1e-4)
else:
    h[key] += np.square(grad[key])
network.params[key] -= learning_rate * grad[key] / (np.sqrt(h[key]))


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

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

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
```

### 4. RMSProp パラメータを変えて実行 (decay\_rate=0.55で実行)

jupyter 2\_4\_optimizer Last Checkpoint: 11/24/2018 (unsaved changes)

File Edit View Insert Cell Kernel Widgets Help Not Trusted Python 3 Logout

```
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()
```

Generation: 1000. 正答率(トレーニング) = 0.96  
: 1000. 正答率(テスト) = 0.9154

accuracy

training set

test set

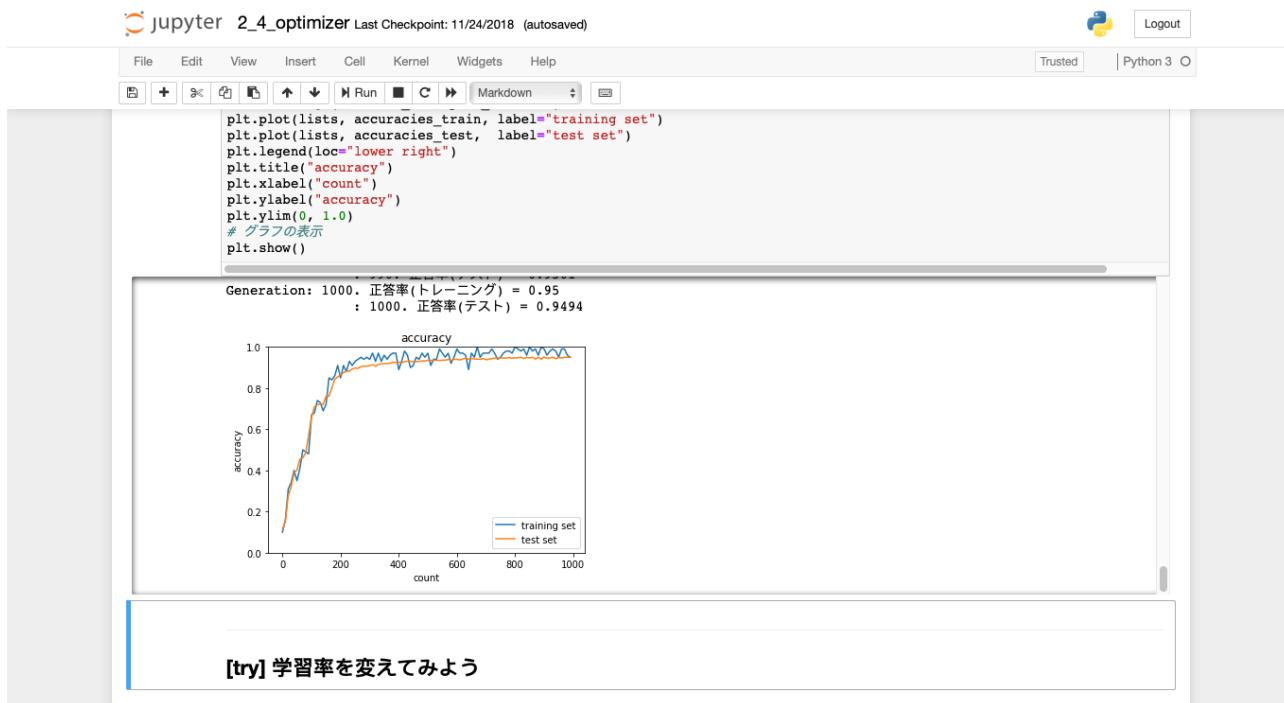
count

## Adam

In [65]: # データの読み込み

```
!tar train.dtrain test.dtest -z -f ./mnist/normalize.tgz -C ./hot_label/
```

## 5. Adam パラメータを変えて実行 (beta1=0.5, beta2=0.555)



### 「Section3:過学習」

- ・テスト誤差と訓練誤差で乖離すること

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...等々  
つまるところ、ネットワークの自由度が高い

- ・正則化～ ネットワークの自由度を制約し過学習を抑制する

L1,L2正則化、ドロップアウトの手法がある

- ・確認テスト リッジ回帰の特徴

全ての重みが限りなく0に近く（完全に0にはならない）

バイアスは正則化されない。誤差関数に掛けて正則化する。

- ・Weight Decay

重みが大きい値をとることで過学習が発生する。

解決策：誤差に対して正則化項を加算することで重みを抑制する。

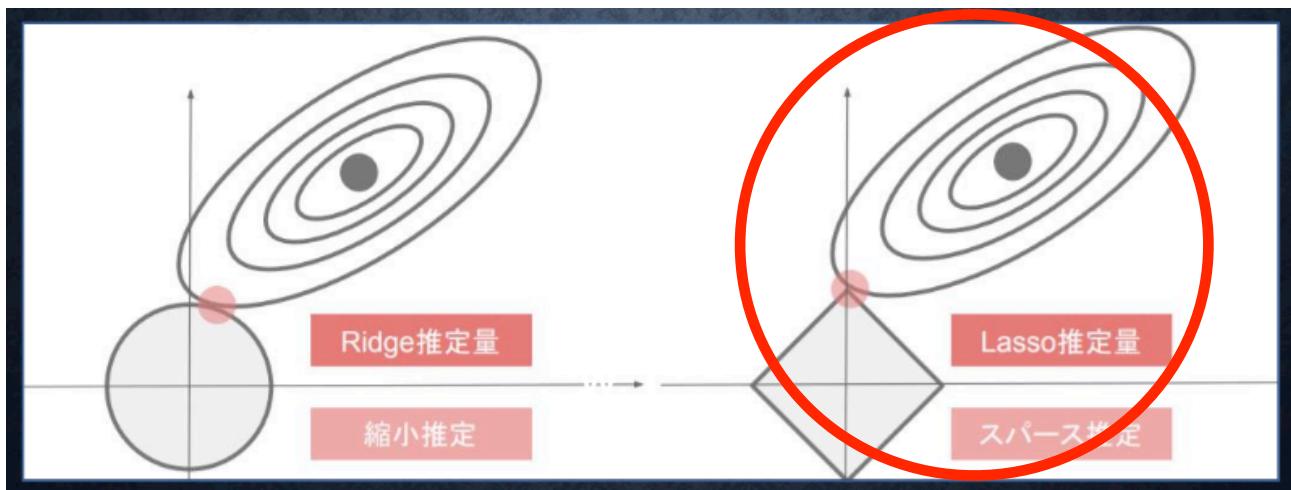
重みをコントロールし、重みの大きさにばらつきを出す。

- ・L1, L2正則化

L1, L2, L3.....Lnと続いてゆく

p=1の場合L1正則化、p=2の場合L2正則化、誤差関数にPノルムを加える。

- ・確認テスト L1正則化を表しているグラフはどちら？



L1

正則化の場合、いくつかのパラメータは0になる。

- ドロップアウト

ノードの数が多いと過学習になりやすい。ランダムにノードを削除して学習させる。

閾値を超えたノードは不活性にする。

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。

- Jupyter演習

### 1. Overfitting

jupyter 2\_5\_overfitting Last Checkpoint: 11/24/2018 (autosaved)

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3 Logout

```

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()

```

Generation: 1000. 正答率(トレーニング) = 1.0  
: 1000. 正答率(テスト) = 0.7477

accuracy

training set test set

## weight decay  
## L2

```

In [5]: 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")

```

## 2. L2 overfittingに比べれば過学習が抑えられている

jupyter 2\_5\_overfitting Last Checkpoint: 11/24/2018 (autosaved)

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3 Logout

```
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()
```

: 980. 正答率(テスト) = 0.725  
Generation: 990. 正答率(トレーニング) = 0.94  
: 990. 正答率(テスト) = 0.7291  
Generation: 1000. 正答率(トレーニング) = 0.9233333333333333  
: 1000. 正答率(テスト) = 0.7312

L1

## 3. L1 overfittingに比べれば過学習が抑制されている

jupyter 2\_5\_overfitting Last Checkpoint: 11/24/2018 (unsaved changes)

File Edit View Insert Cell Kernel Widgets Help Not Trusted Python 3 Logout

```
plt.title("accuracy")
plt.xlabel("count")
plt.ylabel("accuracy")
plt.ylim(0, 1.0)
# グラフの表示
plt.show()
```

Generation: 1000. 正答率(トレーニング) = 0.9566666666666667  
: 1000. 正答率(テスト) = 0.6883

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

Dropout

## 4. L2 weight\_decay\_lambda=0.05

jupyter 2\_5\_overfitting Last Checkpoint: 11/24/2018 (unsaved changes)

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3 Logout

```
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()
```

Generation: 1000. 正答率(トレーニング) = 0.99  
: 1000. 正答率(テスト) = 0.7768

L1

```
In [1]: (x_train, d_train), (x_test, d_test) = load_mnist(normalize=True)
```

## 5. L1 weight\_decay\_lambda=0.01

jupyter 2\_5\_overfitting Last Checkpoint: 11/24/2018 (autosaved)

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3 Logout

```
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()
```

Generation: 990. 正答率(トレーニング) = 0.85  
: 990. 正答率(テスト) = 0.6888  
Generation: 1000. 正答率(トレーニング) = 0.87  
: 1000. 正答率(テスト) = 0.6831

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

weight\_decay\_lambda=1にすると過学習は抑制されるが学習が進まない

## ・Dropout演習

実務はライブラリを使用してドロップアウトする。

1. drop\_out\_ratio=0.5, optimizer=adadmd, dropout=Trueで実行

jupyter 2\_5\_overfitting Last Checkpoint: 11/24/2018 (unsaved changes)

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3 Logout

```
plt.title("accuracy")
plt.xlabel("count")
plt.ylabel("accuracy")
plt.ylim(0, 1.0)
# グラフの表示
plt.show()
```

Generation: 1000. 正答率(トレーニング) = 0.13  
: 1000. 正答率(テスト) = 0.1135

accuracy

accuracy

## [try] dropout\_ratioの値を変更してみよう

## [try] optimizerとdropout\_ratioの値を変更してみよう

2. dropout + L1

jupyter 2\_5\_overfitting Last Checkpoint: 11/24/2018 (unsaved changes)

File Edit View Insert Cell Kernel Widgets Help Not Trusted Python 3 Logout

```
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()
```

Generation: 1000. 正答率(トレーニング) = 0.94  
: 1000. 正答率(テスト) = 0.7511

accuracy

accuracy

実行には時間がかかるので、ロジックを理解し仮説を立てて実行するのが実務では大切。

## 「Section4: 置み込みニューラルネットワークの概念」

- ・ CNN～置み込み層、プーリング層、全結合層の構造を持つ
- ・ LeNet～CNNの大元。
- ・ 置み込み層～画像の場合、縦・横・チャンネルの3次元データである。

3次元データをそのまま学習して次に伝えることができる。

3次元情報の空間情報も学習できる（RGBも学習できる）。

入力値\*フィルター（重み）>出力値+バイアス>活性化関数>出力値

- ・ バイアス～出力画像の前に加算される。
- ・ パディング～0パディング（0以外も可能）
- ・ ストライド～いくつずらして入力値とするか。
- ・ チャンネル～空間情報を分解した数。

チャンネル数分のフィルタ（重み）を用意する必要がある。

全結合の場合一次元データとして処理されてしまうのを防ぐ方法。

- ・ Jupiter演習 im2colの処理

im2col～多次元配列を二次元配列にする。

メリット：行列計算に落とし込みライブラリを使用できる。

1. stride=2で実行。

The screenshot shows a Jupyter Notebook interface with the following details:

- Header:** jupyter 2\_6\_simple\_convolution\_network Last Checkpoint: 11/24/2018 (unsaved changes) Trusted Python 3
- Toolbar:** File Edit View Insert Cell Kernel Widgets Help
- In [3]:** # 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)  
filter\_h = 3  
filter\_w = 3  
stride = 2  
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 =====  
[[[70. 40. 12. 41.]  
 [ 1. 95. 98. 44.]  
 [71. 98. 53. 26.]  
 [60. 77. 8. 9.]]]  
  
===== col =====  
[[70. 40. 12. 1. 95. 98. 71. 98. 53.]  
 [81. 13. 98. 99. 39. 12. 55. 74. 24.]]  
===== =====
- Text Box:** column to image
- In [4]:** # 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):

## 2. col2im 2次元配列を画像データに変換

im2colで変換した画像をcol2imで復元することはできないので注意。

The screenshot shows a Jupyter Notebook interface with the following details:

- Title Bar:** jupyter 2\_6\_simple\_convolution\_network\_after Last Checkpoint: 11/24/2018 (unsaved changes)
- Toolbar:** File, Edit, View, Insert, Cell, Kernel, Widgets, Help, Run, Cell, Run, Kernel, Help, Markdown, Cell, Cell, Run, Kernel, Help, Logout, Not Trusted, Python 3.
- Cell 1:** [try] col2imの処理を確認しよう  
• im2colの確認で出力したcolをimageに変換して確認しよう
- Cell 2 (In [4]):**# ここにcol2imでの処理を書こう  
img = col2im(col, input\_shape=input\_data.shape, filter\_h=filter\_h, filter\_w=filter\_w, stride=stride, pad=pad)  
print(img)  
[[[ 11. 14. 70. 63.]  
[ 90. 136. 72. 8.]  
[ 8. 4. 384. 10.]  
[ 42. 60. 122. 2.]]]  
  
[[[ 74. 122. 4. 34.]  
[110. 344. 80. 0.]  
[ 30. 216. 196. 44.]  
[ 15. 12. 132. 77.]]]]
- Cell 3 (In [ ]):** convolution class  
class Convolution:  
 # W: フィルター, b: バイアス  
 def \_\_init\_\_(self, W, b, stride=1, pad=0):  
 self.W = W  
 self.b = b  
 self.stride = stride  
 self.pad = pad  
  
 # 中間データ (backward時に使用)  
 self.x = None  
 self.col = None

- ・プーリング層～Maxプーリング（入力値の中で最大値を取得し出力）  
アベレージプーリング（入力値の平均値を算出し取得し出力）
- ・確認テスト 6×6の入力画像、2×2のフィルタ、ストライド・パディング=1  
公式： $OH = ((h + 2p - fh) / s) + 1$   
 $OW = ((w + 2p - fw) / s) + 1$   
公式より  $OH=7, OW=7$

### ・ソースコード解説

Maxプーリング実装：

```
# 行ごとに最大値を求める  
arg_max = np.argmax(col, axis=1)  
out = np.max(col, axis=1)
```

## 1. Simple Convolution Network Class実行

jupyter 2\_6\_simple\_convolution\_network Last Checkpoint: 11/24/2018 (autosaved)

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3 Logout

```
plt.ylim(0, 1.0)
# グラフの表示
plt.show()
```

Generation: 1000. 正答率(トレーニング) = 0.9938  
: 1000. 正答率(テスト) = 0.956

accuracy

training set

test set

In [ ]:

In [ ]:

In [ ]:

実行に時間がかかる場合はGPU,CPUを工夫すること。畳み込み層を使用すると手持ちのPCでは時間がかかるってしまう。

## 2. オプティマイザー(Momentum)を変えて実行

jupyter 2\_6\_simple\_convolution\_network Last Checkpoint: 11/24/2018 (unsaved changes)

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3 Logout

```
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()
```

Generation: 1000. 正答率(トレーニング) = 0.973  
: 1000. 正答率(テスト) = 0.937

accuracy

training set

test set

In [ ]:

In [ ]:

## 「Section5:最新のCNN」

- ・AlexNet～画像認識のコンペで2012年に優勝したモデル。DLが注目される原因に。
  - ・5層の畳み込み層とプーリング層、3層の全結合層から構成。
  - ・真似されるのを恐れて、細かい部分は公開されていない。論文から読み解くこと。
  - ・過学習抑制のため全結合層にドロップアウトを使用している。
- ドロップアウトはやはり大切である。