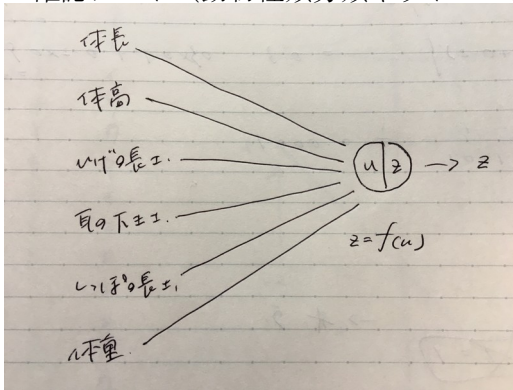


深層学習（前編）

Section 1) 入力層～中間層

- ・確認テスト（動物種類分類ネットワーク）



- ・確認テスト（数式とコード）
 $u = \text{np.dot}(W1, x) + b1$
※内積は行列の次元を合わせる。合わない場合エラーとなる
- ・深層学習 1 - 6 ～演習問題

```
In [13]: #順伝播 (単層・単ユニット)

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

## 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=(3))

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, 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 4 4]

*** バイアス ***
-4.066864334617727

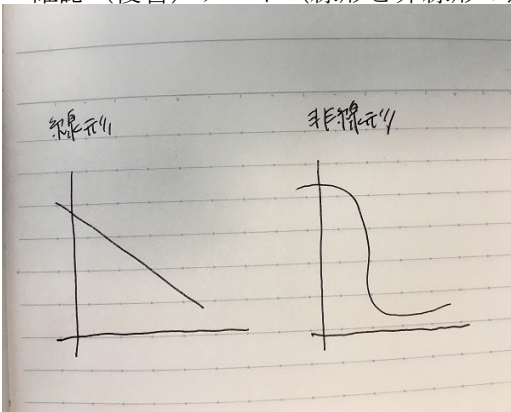
*** 入力 ***
[2 3 3]

*** 総入力 ***
19.933135665382274

*** 中間層出力 ***
19.933135665382274
```

Section 2) 活性化関数

- 活性化関数
 - ニューラルネットワークにおいて次の層への出力の強弱を決める非線形の関数
- 確認 (復習) テスト (線形と非線形の違い)



- 中間層用
 - ステップ関数 (現在ディープラーニングでは使われていない)
 - 閾値越えで発火
 - 出力は 0 or 1 → 課題は 0 ~ 1 の間を表現できないこと
 - シグモイド (ロジスティック) 関数
 - 0 ~ 1 の間を緩やかに変化 → 信号の強弱を表現できるように
 - 大きな値では出力の変化が微小 → 勾配消失問題
 - ReLU 関数
 - 今もっと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
- ※シグモイドかReLUか → 問題の構造で異なる
- 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
- 深層学習 1 - 8 ~ 実装演習

```
In [15]: # 順伝播 (単層・複数ユニット)

# 重み
#W = np.array([
#    [0.1, 0.2, 0.3],
#    [0.2, 0.3, 0.4],
#    [0.3, 0.4, 0.5],
#    [0.4, 0.5, 0.6]
#])

## 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(x, W) + b
print_vec("総入力", u)
```

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

```
*** 重み ***
[[1 1 1]
 [1 4 2]
 [0 1 1]
 [2 0 4]]

*** バイアス ***
[0.1 0.2 0.3]

*** 入力 ***
[1. 5. 2. -1.]

*** 総入力 ***
[4.1 23.2 9.3]

*** 中間層出力 ***
[4.1 23.2 9.3]
```

Section 3) 出力層

3-1 誤差関数

- ・事前に用意するデータ
 - 入力データ
 - 訓練データ（正解値）
- ・誤差関数
 - 出力と訓練データの差を求める

3-2 出力層の活性化関数

- ・中間層との違い→活性化関数が異なる
 - 値の強弱：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）をそのまま変換
- 確率出力：分類問題の場合、出力層の出力は0～1の範囲とし、総和を1とする必要がある
- ・出力層用の活性化関数
 - ーソフトマックス関数
 - ー恒等写像
 - ーシグモイド（ロジスティック）関数
- ・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	

※活性化関数と誤差関数の組み合わせは計算の相性が良いため上記となる

Section 4) 勾配降下法

- ・勾配降下法
 - 深層学習の目的
 - 学習を通して誤差を最小にするネットワークを作成すること
 - 誤差E(w)を最小にするパラメータを発見すること
 - 学習率の値によって学習の効率が大きく異なる
 - 大きくしすぎた場合：最小値にいつまでもたどり着かず発散してしまう
 - 小さくしすぎた場合：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
 - 大域的極小解にとらわれてしまう
 - 学習率の決定、収束率向上のためのアルゴリズム
 - ーMomentum
 - ーAdaGrad
 - ーAdadelta
 - ーAdam
- ・確率的勾配降下法（SGD）
 - ランダムに抽出したサンプルの誤差⇔勾配降下法：全サンプルの平均誤差
 - メリット
 - ーデータが冗長な場合の計算コストの軽減
 - ー望まない局所極小解に収束するリスクの低減
 - ーオンライン学習が可能（誤差を1訓練データ毎に更新できる）

- 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） D_t に属するサンプルの平均誤差
→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活用できる
CPUを利用したスレッド並列化、GPUを利用したSIMD並列化
- 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疑似的に微分を計算する一般的な手法
$$\frac{\partial E}{\partial w} \approx \frac{E(W_m + h) - E(W_m - h)}{2h}$$

デメリット：計算負荷が大きい→誤差逆伝播法を利用する

Section 5) 誤差逆伝播法

- 誤差逆伝播法
算出された誤差を出力層側から順に微分し前の層へ伝播。
必要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。
- 誤差勾配の計算

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

$y = u^{(L)}$: 出力層の活性化関数 (恒等写像)

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

$$\frac{\partial E}{\partial w_{ji}^{(2)}} = \frac{\partial E}{\partial y} \frac{\partial y}{\partial u} \frac{\partial u}{\partial w_{ji}^{(2)}}$$

$$\frac{\partial E}{\partial y} = \frac{\partial}{\partial y} \frac{1}{2} \|y - d\|^2 = y - d$$

$$\frac{\partial y}{\partial u} = \frac{\partial u}{\partial u} = 1$$

$$\frac{\partial u(w)}{\partial w_{ji}} = \frac{\partial}{\partial w_{ji}} (w^{(1)} z^{(1-1)} + b^{(1)})$$

$$= \frac{\partial}{\partial w_{ji}} \begin{bmatrix} w_{11} z_1 + \dots + w_{1i} z_i + \dots + w_{1I} z_I & b_1 & 0 \\ \vdots & \vdots & \vdots \\ w_{j1} z_1 + \dots + w_{ji} z_i + \dots + w_{jI} z_I & b_j & 0 \\ \vdots & \vdots & \vdots \\ w_{J1} z_1 + \dots + w_{Ji} z_i + \dots + w_{JI} z_I & b_I & 0 \end{bmatrix} = \begin{bmatrix} z_1 \\ \vdots \\ z_i \\ \vdots \\ z_I \end{bmatrix}$$

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

• 深層学習 1 - 15 ~ 実装演習

```
In [15]: 1 # サンプルとする関数
2 # xの値を予想するAI
3
4 def f(x):
5     y = 3 * x[0] + 2 * x[1]
6     return y
7
8 # 初期設定
9 def init_network():
10     # print("##### ネットワークの初期化 #####")
11     network = {}
12     nodesNum = 10
13     network['W1'] = np.random.randn(2, nodesNum)
14     network['W2'] = np.random.randn(nodesNum)
15     network['b1'] = np.random.randn(nodesNum)
16     network['b2'] = np.random.randn()
17
18     # print_vec("重み1", network['W1'])
19     # print_vec("重み2", network['W2'])
20     # print_vec("バイアス1", network['b1'])
21     # print_vec("バイアス2", network['b2'])
22
23     return network
24
25 # 順伝播
26 def forward(network, x):
27     # print("##### 順伝播開始 #####")
28
29     W1, W2 = network['W1'], network['W2']
30     b1, b2 = network['b1'], network['b2']
31     u1 = np.dot(x, W1) + b1
32     # z1 = functions.relu(u1)
33
34     ## 試してみよう
35     z1 = functions.sigmoid(u1)
```

```

37 u2 = np.dot(z1, W2) + b2
38 y = u2
39
40 # print_vec("総入力1", u1)
41 # print_vec("中間層出力1", z1)
42 # print_vec("総入力2", u2)
43 # print_vec("出力1", y)
44 # print("出力合計: " + str(np.sum(y)))
45
46 return z1, y
47
48 # 誤差逆伝播
49 def backward(x, d, z1, y):
50     # print("\n##### 誤差逆伝播開始 #####")
51
52     grad = {}
53
54     W1, W2 = network["W1"], network["W2"]
55     b1, b2 = network["b1"], network["b2"]
56
57     # 出力層でのデルタ
58     delta2 = functions.d_mean_squared_error(d, y)
59     # b2の勾配
60     grad["b2"] = np.sum(delta2, axis=0)
61     # W2の勾配
62     grad["W2"] = np.dot(z1.T, delta2)
63     # 中間層でのデルタ
64     # delta1 = np.dot(delta2, W2.T) * functions.d_relu(z1)
65
66     ## 試してみよう
67     delta1 = np.dot(delta2, W2.T) * functions.d_sigmoid(z1)
68
69     delta1 = delta1[np.newaxis, :]
70     # b1の勾配
71     grad["b1"] = np.sum(delta1, axis=0)
72     x = x[np.newaxis, :]
73     # W1の勾配
74     grad["W1"] = np.dot(x.T, delta1)

```

```

75
76 # print_vec("偏微分_重み1", grad["W1"])
77 # print_vec("偏微分_重み2", grad["W2"])
78 # print_vec("偏微分_バイアス1", grad["b1"])
79 # print_vec("偏微分_バイアス2", grad["b2"])
80
81 return grad
82
83 # サンプルデータを作成
84 data_sets_size = 100000
85 data_sets = [0 for i in range(data_sets_size)]
86
87 for i in range(data_sets_size):
88     data_sets[i] = {}
89     # ランダムな値を設定
90     data_sets[i]["x"] = np.random.rand(2)
91
92     ## 試してみよう_入力値の設定
93     data_sets[i]["x"] = np.random.rand(2) * 10 - 5 # -5~5のランダム数値
94
95     # 目標出力を設定
96     data_sets[i]["d"] = f(data_sets[i]["x"])
97
98 losses = []
99 # 学習率
100 learning_rate = 0.07
101
102 # 抽出数
103 epoch = 1000
104
105 # パラメータの初期化
106 network = init_network()
107 # データのランダム抽出
108 random_datasets = np.random.choice(data_sets, epoch)
109

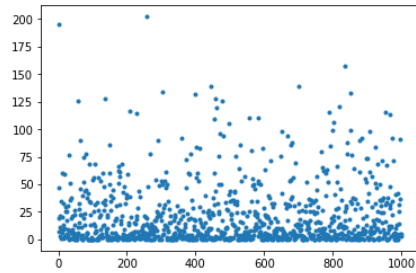
```

```

110 # 勾配降下の繰り返し
111 for dataset in random_datasets:
112     x, d = dataset['x'], dataset['d']
113     z1, y = forward(network, x)
114     grad = backward(x, d, z1, y)
115     # パラメータに勾配適用
116     for key in ('w1', 'w2', 'b1', 'b2'):
117         network[key] -= learning_rate * grad[key]
118
119 # 誤差
120 loss = functions.mean_squared_error(d, y)
121 losses.append(loss)
122
123 print("#### 結果表示 ####")
124 lists = range(epoch)
125
126 plt.plot(lists, losses, '.')
127 # グラフの表示
128 plt.show()

```

結果表示



• 修了課題について

Q 1. 課題の目的とは？どのような工夫ができそうか

i r i s データを使って回帰または分類を行う。

以下の事項について、工夫する余地がある。

ー入力層のノード数

ー中間層の層数・ノード数

ー回帰または分類に応じた出力層の活性化関数・誤差関数

Q 2. 課題を分類タスクで解く場合の意味は何か

該当のデータがどのクラスに属するかを解くことが、課題の意味・目的となる。

Q 3. i r i s データとは何か2行で述べよ

アヤメの花の特徴量を記録したデータ。品種としてセトナ、ヴァージカラー、ヴァージニカの3品種、特徴量としてがく片と花びらそれぞれの長さと幅が記録されている。

深層学習（後編）

Section 1) 勾配消失問題

1-1 活性化関数

- 勾配消失問題

誤差逆伝播法を使用していると、下位層に進んでいくにつれて勾配が緩やかになっていく
→勾配降下法による更新で誤差項の値が0に近づいて行ってしまう
シグモイド関数は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ことが多い
→ReLU関数を使用することで回避する

1-2 初期値の設定方法

- 重みの初期値設定 (Xavier)

ReLU関数、シグモイド関数、双曲線正接関数と共に使用する
重みの要素を前の層のノード数の平方根で除算

- 重みの初期値設定 (He)

ReLU関数と共に使用する (勾配消失問題を回避するためにReLU関数に特化)
重みの要素を前の層のノード数の平方根で除算後に $\sqrt{2}$ を乗算

- 重みの初期値に0を設定する

誤差逆伝播法ですべての重みの値が同じように更新されてしまう
→パラメータのチューニングが行われなくなってしまう
たくさんの重みをもつ意味がなくなってしまうためディープラーニングの手法上ナンセンス
ランダムな初期値が必要

1-3 バッチ正規化

- バッチ正規化とは？

ミニバッチ単位で入力値のデータの偏りを抑制

- バッチ正規化の使いどころとは？

活性化関数に値を渡す前後にバッチ正規化の処理を孕んだ層を加える

- 一般的に考えられるバッチ正規化の効果

→正規化によってデータのばらつきが抑えられることにより計算の高速化が期待できる
→勾配消失問題が起きづらくなる

- 深層学習 2-5 ~実装演習

Sigmoid-He

```
In [3]: 1 import sys, os
2 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
3 import numpy as np
4 from data.mnist import load_mnist
5 from PIL import Image
6 import pickle
7 from common import functions
8 import matplotlib.pyplot as plt
9
10 # mnistをロード
11 (x_train, y_train), (x_test, y_test) = load_mnist(normalize=True, one_hot_label=True)
12 train_size = len(x_train)
13
14 print("データ読み込み完了")
15
16 #入力層サイズ
17 input_layer_size = 784
18 #中間層サイズ
19 hidden_layer_1_size = 40
20 hidden_layer_2_size = 20
21 #出力層サイズ
22 output_layer_size = 10
23 #繰り返し数
24 iters_num = 2000
25 #ミニバッチサイズ
26 batch_size = 100
27 #学習率
28 learning_rate = 0.1
29 #描写頻度
30 plot_interval = 10
```

```

31
32 # 初期設定
33 def init_network():
34     network = {}
35
36     ##### 変更箇所 #####
37
38     # Xavierの初期値
39     network['W1'] = np.random.randn(input_layer_size, hidden_layer_1_size) / (np.sqrt(input_layer_size))
40     network['W2'] = np.random.randn(hidden_layer_1_size, hidden_layer_2_size) / (np.sqrt(hidden_layer_1_size))
41     network['W3'] = np.random.randn(hidden_layer_2_size, output_layer_size) / (np.sqrt(hidden_layer_2_size))
42     # Heの初期値
43     network['W1'] = np.random.randn(input_layer_size, hidden_layer_1_size) / np.sqrt(input_layer_size) * np.sqrt(2)
44     network['W2'] = np.random.randn(hidden_layer_1_size, hidden_layer_2_size) / np.sqrt(hidden_layer_1_size) * np.sqrt(2)
45     network['W3'] = np.random.randn(hidden_layer_2_size, output_layer_size) / np.sqrt(hidden_layer_2_size) * np.sqrt(2)
46
47     #####
48
49     network['b1'] = np.zeros(hidden_layer_1_size)
50     network['b2'] = np.zeros(hidden_layer_2_size)
51     network['b3'] = np.zeros(output_layer_size)
52
53     return network
54
55 # 順伝播
56 def forward(network, x):
57     W1, W2, W3 = network['W1'], network['W2'], network['W3']
58     b1, b2, b3 = network['b1'], network['b2'], network['b3']
59     hidden_f = functions.sigmoid
60
61     u1 = np.dot(x, W1) + b1
62     z1 = hidden_f(u1)

```

```

63     u2 = np.dot(z1, W2) + b2
64     z2 = hidden_f(u2)
65     u3 = np.dot(z2, W3) + b3
66     y = functions.softmax(u3)
67
68     return z1, z2, y
69
70 # 誤差逆伝播
71 def backward(x, d, z1, z2, y):
72     grad = {}
73
74     W1, W2, W3 = network['W1'], network['W2'], network['W3']
75     b1, b2, b3 = network['b1'], network['b2'], network['b3']
76     hidden_d_f = functions.d_sigmoid
77
78     # 出力層でのデルタ
79     delta3 = functions.d_softmax_with_loss(d, y)
80     # b3の勾配
81     grad['b3'] = np.sum(delta3, axis=0)
82     # W3の勾配
83     grad['W3'] = np.dot(z2.T, delta3)
84     # 2層でのデルタ
85     delta2 = np.dot(delta3, W3.T) * hidden_d_f(z2)
86     # b2の勾配
87     grad['b2'] = np.sum(delta2, axis=0)
88     # W2の勾配
89     grad['W2'] = np.dot(z1.T, delta2)
90     # 1層でのデルタ
91     delta1 = np.dot(delta2, W2.T) * hidden_d_f(z1)
92     # b1の勾配
93     grad['b1'] = np.sum(delta1, axis=0)

```

```

94     # W1の勾配
95     grad['W1'] = np.dot(x.T, delta1)
96
97     return grad
98
99 # パラメータの初期化
100 network = init_network()
101
102 accuracies_train = []
103 accuracies_test = []
104
105 # 正答率
106 def accuracy(x, d):
107     z1, z2, y = forward(network, x)
108     y = np.argmax(y, axis=1)
109     if d.ndim != 1 : d = np.argmax(d, axis=1)
110     accuracy = np.sum(y == d) / float(x.shape[0])
111     return accuracy
112
113 for i in range(iters_num):
114     # ランダムにバッチを取得
115     batch_mask = np.random.choice(train_size, batch_size)
116     # ミニバッチに対応する教師訓練画像データを取得
117     x_batch = x_train[batch_mask]
118     # 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
119     d_batch = d_train[batch_mask]
120
121
122

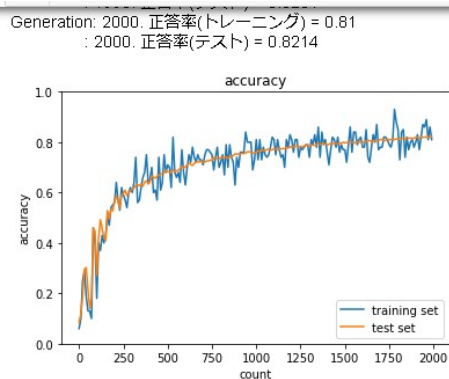
```



```

123 z1, z2, y = forward(network, x_batch)
124 grad = backward(x_batch, d_batch, z1, z2, y)
125
126 if (i+1)%plot_interval==0:
127     accr_test = accuracy(x_test, d_test)
128     accuracies_test.append(accr_test)
129
130     accr_train = accuracy(x_batch, d_batch)
131     accuracies_train.append(accr_train)
132
133     print('Generation: ' + str(i+1) + '. 正答率(トレーニング) = ' + str(accr_train))
134     print('          : ' + str(i+1) + '. 正答率(テスト) = ' + str(accr_test))
135
136     # パラメータに勾配適用
137     for key in ('W1', 'W2', 'W3', 'b1', 'b2', 'b3'):
138         network[key] -= learning_rate * grad[key]
139
140
141 lists = range(0, iters_num, plot_interval)
142 plt.plot(lists, accuracies_train, label="training set")
143 plt.plot(lists, accuracies_test, label="test set")
144 plt.legend(loc="lower right")
145 plt.title("accuracy")
146 plt.xlabel("count")
147 plt.ylabel("accuracy")
148 plt.ylim(0, 1.0)
149 # グラフの表示
150 plt.show()

```



ReLU-Xavier

```

In [4]: 1 import sys, os
2 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
3 import numpy as np
4 from data.mnist import load_mnist
5 from PIL import Image
6 import pickle
7 from common import functions
8 import matplotlib.pyplot as plt
9
10 # mnistをロード
11 (x_train, d_train), (x_test, d_test) = load_mnist(normalize=True, one_hot_label=True)
12 train_size = len(x_train)
13
14 print("データ読み込み完了")
15
16 # 重み初期値補正係数
17 weight_init = 0.01
18 # 入力層サイズ
19 input_layer_size = 784
20 # 中間層サイズ
21 hidden_layer_1_size = 40
22 hidden_layer_2_size = 20
23
24 # 出力層サイズ
25 output_layer_size = 10
26 # 繰り返し数
27 iters_num = 2000
28 # ミニバッチサイズ
29 batch_size = 100

```

```

30 #学習率
31 learning_rate = 0.1
32 #描写頻度
33 plot_interval=10
34
35 #初期設定
36 def init_network():
37     network = {}
38
39     ##### 変更箇所 #####
40
41     # Xavierの初期値
42     network['W1'] = np.random.randn(input_layer_size, hidden_layer_1_size) / (np.sqrt(input_layer_size))
43     network['W2'] = np.random.randn(hidden_layer_1_size, hidden_layer_2_size) / (np.sqrt(hidden_layer_1_size))
44     network['W3'] = np.random.randn(hidden_layer_2_size, output_layer_size) / (np.sqrt(hidden_layer_2_size))
45     # Heの初期値
46     #network['W1'] = np.random.randn(input_layer_size, hidden_layer_1_size) / np.sqrt(input_layer_size) * np.sqrt(2)
47     #network['W2'] = np.random.randn(hidden_layer_1_size, hidden_layer_2_size) / np.sqrt(hidden_layer_1_size) * np.sqrt(2)
48     #network['W3'] = np.random.randn(hidden_layer_2_size, output_layer_size) / np.sqrt(hidden_layer_2_size) * np.sqrt(2)
49
50     #####
51
52     network['b1'] = np.zeros(hidden_layer_1_size)
53     network['b2'] = np.zeros(hidden_layer_2_size)
54     network['b3'] = np.zeros(output_layer_size)
55
56     return network
57

```

```

58 #順伝播
59 def forward(network, x):
60     W1, W2, W3 = network['W1'], network['W2'], network['W3']
61     b1, b2, b3 = network['b1'], network['b2'], network['b3']
62
63     ##### 変更箇所 #####
64
65     hidden_f = functions.relu
66
67     #####
68
69     u1 = np.dot(x, W1) + b1
70     z1 = hidden_f(u1)
71     u2 = np.dot(z1, W2) + b2
72     z2 = hidden_f(u2)
73     u3 = np.dot(z2, W3) + b3
74     y = functions.softmax(u3)
75
76     return z1, z2, y
77
78 #誤差逆伝播
79 def backward(x, d, z1, z2, y):
80     grad = {}
81
82     W1, W2, W3 = network['W1'], network['W2'], network['W3']
83     b1, b2, b3 = network['b1'], network['b2'], network['b3']
84
85     ##### 変更箇所 #####
86
87     hidden_d_f = functions.d_relu
88
89     #####

```

```

90
91     #出力層でのデルタ
92     delta3 = functions.d_softmax_with_loss(d, y)
93     # b3の勾配
94     grad['b3'] = np.sum(delta3, axis=0)
95     # W3の勾配
96     grad['W3'] = np.dot(z2.T, delta3)
97     # 2層でのデルタ
98     delta2 = np.dot(delta3, W3.T) * hidden_d_f(z2)
99     # b2の勾配
100    grad['b2'] = np.sum(delta2, axis=0)
101    # W2の勾配
102    grad['W2'] = np.dot(z1.T, delta2)
103    # 1層でのデルタ
104    delta1 = np.dot(delta2, W2.T) * hidden_d_f(z1)
105    # b1の勾配
106    grad['b1'] = np.sum(delta1, axis=0)
107    # W1の勾配
108    grad['W1'] = np.dot(x.T, delta1)
109
110    return grad
111
112 #パラメータの初期化
113 network = init_network()
114
115 accuracies_train = []
116 accuracies_test = []
117
118 #正答率
119 def accuracy(x, d):

```

```

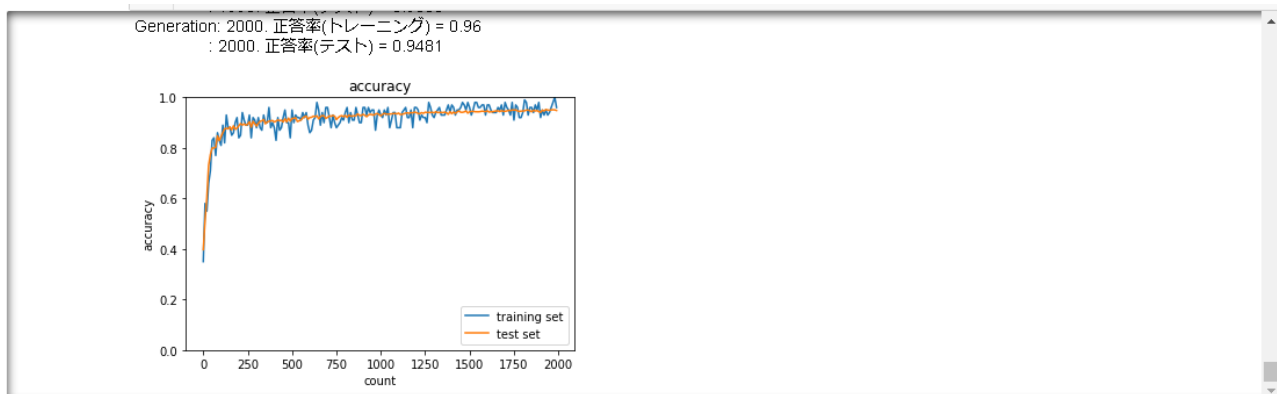
120 z1, z2, y = forward(network, x)
121 y = np.argmax(y, axis=1)
122 if d.ndim != 1 : d = np.argmax(d, axis=1)
123 accuracy = np.sum(y == d) / float(x.shape[0])
124 return accuracy
125
126 for i in range(iters_num):
127     #ランダムにバッチを取得
128     batch_mask = np.random.choice(train_size, batch_size)
129     #ミニバッチに対応する教師訓練画像データを取得
130     x_batch = x_train[batch_mask]
131     #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
132     d_batch = d_train[batch_mask]
133
134
135
136 z1, z2, y = forward(network, x_batch)
137 grad = backward(x_batch, d_batch, z1, z2, y)
138
139 if (i+1)%plot_interval==0:
140     accr_test = accuracy(x_test, d_test)
141     accuracies_test.append(accr_test)
142
143     accr_train = accuracy(x_batch, d_batch)
144     accuracies_train.append(accr_train)
145
146     print('Generation: ' + str(i+1) + '. 正答率(トレーニング) = ' + str(accr_train))
147     print('          : ' + str(i+1) + '. 正答率(テスト) = ' + str(accr_test))
148
149     #パラメータに勾配適用
150     for key in ('W1', 'W2', 'W3', 'b1', 'b2', 'b3'):

```

```

151         network[key] -= learning_rate * grad[key]
152
153
154 lists = range(0, iters_num, plot_interval)
155 plt.plot(lists, accuracies_train, label="training set")
156 plt.plot(lists, accuracies_test, label="test set")
157 plt.legend(loc="lower right")
158 plt.title("accuracy")
159 plt.xlabel("count")
160 plt.ylabel("accuracy")
161 plt.ylim(0, 1.0)
162 #グラフの表示
163 plt.show()

```



※実務上では、Sigmoid-HeよりもSigmoid-He、ReLU-HeよりもReLU-Xavierが効果を発揮することがある

Section 2) 学習率最適化手法

2-1 モメンタム

・モメンタムの特徴

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。慣性はハイパーパラメータ。0.5~0.9の間で設定

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

$$W^{t+1} = W^t + V_t$$

μ : 慣性

・モメンタムのメリット

大域的局所解を得やすい

谷間についてから最も低い位置（最適解）に行くまでの時間が早い

2-2 AdaGrad

- AdaGradの特徴

学習が進むにつれて学習係数を小さくする。最初は大きく学習し、次第に小さく学習する
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

$$h_0 = \theta$$

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

$$W^{t+1} = W^t - \epsilon \frac{1}{\sqrt{h_t + \theta}} \nabla E$$

- AdaGradのメリット

緩やかな斜面に入った場合にも最適解に行くまでの時間が早い

- AdaGradの課題

学習率が徐々に小さくなるため鞍点問題を引き起こす

(次元が大きくなった場合に、極小値と思っていた点が別の視点から見ると極大値だった)

2-3 RMSProp

- RMSPropの特徴

過去の勾配を徐々に忘れて新しい勾配の情報が大きく反映されるように加算

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

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

$$W^{t+1} = W^t - \epsilon \frac{1}{\sqrt{h_t + \theta}} \nabla E$$

- RMSPropのメリット

大域的局所解を得やすい

ハイパーパラメータの調整が必要な場合が少ない

2-4 Adam

- Adamの特徴

以下の2つを孕んだ最適化アルゴリズム

ーモメンタムの、過去の勾配の指数関数的減衰平均

ーRMSPropの、過去の勾配の2乗の指数関数的減衰平均

- 深層学習 2-8 ~実装演習

```
In [6]: 1 # AdaGradを作ってみよう
2 # データの読み込み
3 (x_train, d_train), (x_test, d_test) = load_mnist(normalize=True, one_hot_label=True)
4
5 print("データ読み込み完了")
6
7 # batch_normalizationの設定 =====
8 # use_batchnorm = True
9 use_batchnorm = False
10 # =====
11
12 network = MultiLayerNet(input_size=784, hidden_size_list=[40, 20], output_size=10, activation='sigmoid', weight_init_std=0.01,
13                          use_batchnorm=use_batchnorm)
14
15 iters_num = 1000
16 # iters_num = 500 # 処理を短縮
17
18 train_size = x_train.shape[0]
19 batch_size = 100
20 learning_rate = 0.01
21
22 # AdaGradでは不必要
23 # =====
24
25 # momentum = 0.9
26
27 # =====
28
29 train_loss_list = []
30 accuracies_train = []
31 accuracies_test = []
32
33 plot_interval=10
34
35 for i in range(iters_num):
36     batch_mask = np.random.choice(train_size, batch_size)
37     x_batch = x_train[batch_mask]
38     d_batch = d_train[batch_mask]
```

```

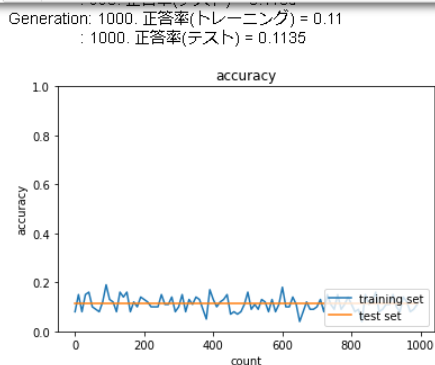
39
40 # 勾配
41 grad = network.gradient(x_batch, d_batch)
42 if i == 0:
43     h = {}
44     for key in ('W1', 'W2', 'W3', 'b1', 'b2', 'b3'):
45
46         # 変更しよう
47         # =====
48         # if i == 0:
49         #     h[key] = np.zeros_like(network.params[key])
50         # h[key] = momentum * h[key] - learning_rate * grad[key]
51         # network.params[key] += h[key]
52         if i == 0:
53             h[key] = np.full_like(network.params[key], 1e-4)
54         else:
55             h[key] += np.square(grad[key])
56             network.params[key] -= learning_rate * grad[key] / (np.sqrt(h[key]))
57         # =====
58
59     loss = network.loss(x_batch, d_batch)
60     train_loss_list.append(loss)
61
62     if (i + 1) % plot_interval == 0:
63         accr_test = network.accuracy(x_test, d_test)
64         accuracies_test.append(accr_test)
65         accr_train = network.accuracy(x_batch, d_batch)
66         accuracies_train.append(accr_train)
67
68         print("Generation: " + str(i+1) + ", 正答率(トレーニング) = " + str(accr_train))
69         print("          : " + str(i+1) + ", 正答率(テスト) = " + str(accr_test))
70
71
72 lists = range(0, iters_num, plot_interval)
73 plt.plot(lists, accuracies_train, label="training set")
74 plt.plot(lists, accuracies_test, label="test set")
75 plt.legend(loc="lower right")

```

```

76 plt.title("accuracy")
77 plt.xlabel("count")
78 plt.ylabel("accuracy")
79 plt.ylim(0, 1.0)
80 # グラフの表示
81 plt.show()

```



Section 3) 過学習について

3-1 L1正則化、L2正則化

- 正則化とは
ネットワークの自由度（層数、ノード数、パラメータの値など）を制約すること
- 正則化手法
 - ー L1正則化、L2正則化
 - ー ドロップアウト
- Weight decay（荷重減衰）
 - ー 過学習の原因
重みが大きい値を取ることで過学習が発生することがある
 - ー 過学習の解決策
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
過学習が起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ばらつきを出す必要がある
- L1、L2正則化
 - $E_n(w) + \frac{1}{p} \lambda \|x\|_p$ … 誤差関数に p ノルムを加える
 - $\|x\|_p = (|x_1|^p + \dots + |x_n|^p)^{1/p}$ … p ノルムの計算
 - $p = 1$: L1正則化、 $p = 2$: L2正則化
 - Lasso推定: L1正則化
 - Ridge推定: L2正則化 … 計算リソースを食うが精度向上に寄与する

3-2 ドロップアウト

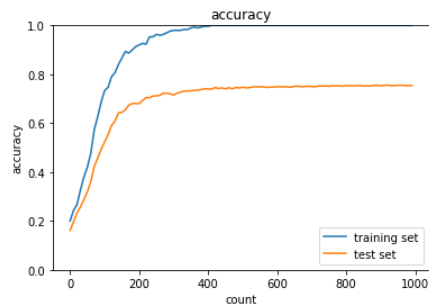
- 過学習の課題
ノード数が多い
→ランダムにノードを削除して学習させる（ドロップアウト）
- ドロップアウトのメリット
データ量を変化させずに異なるモデルを学習させていると解釈できる
- 深層学習 2-1 3～実装演習
weight_decay_lambda の値
→大きすぎても学習が進まない。小さすぎても過学習を抑制できない。
どれくらいの値とするのかはデータセットによっても異なる。

overfitting

```
In [2]: 1 import sys, os
2 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
3 import numpy as np
4 from collections import OrderedDict
5 from common import layers
6 from data.mnist import load_mnist
7 import matplotlib.pyplot as plt
8 from multi_layer_net import MultiLayerNet
9 from common import optimizer
10
11
12 (x_train, d_train), (x_test, d_test) = load_mnist(normalize=True)
13
14 print("データ読み込み完了")
15
16 # 過学習を再現するために、学習データを削減
17 x_train = x_train[:300]
18 d_train = d_train[:300]
19
20 network = MultiLayerNet(input_size=784, hidden_size_list=[100, 100, 100, 100, 100, 100], output_size=10)
21 optimizer = optimizer.SGD(learning_rate=0.01)
22
23 iters_num = 1000
24 train_size = x_train.shape[0]
25 batch_size = 100
26
27 train_loss_list = []
28 accuracies_train = []
29 accuracies_test = []
30
31 plot_interval=10
```

```
32
33
34 for i in range(iters_num):
35     batch_mask = np.random.choice(train_size, batch_size)
36     x_batch = x_train[batch_mask]
37     d_batch = d_train[batch_mask]
38
39     grad = network.gradient(x_batch, d_batch)
40     optimizer.update(network.params, grad)
41
42     loss = network.loss(x_batch, d_batch)
43     train_loss_list.append(loss)
44
45     if (i+1) % plot_interval == 0:
46         accr_train = network.accuracy(x_train, d_train)
47         accr_test = network.accuracy(x_test, d_test)
48         accuracies_train.append(accr_train)
49         accuracies_test.append(accr_test)
50
51     print("Generation: " + str(i+1) + ". 正答率(トレーニング) = " + str(accr_train))
52     print("          : " + str(i+1) + ". 正答率(テスト) = " + str(accr_test))
53
54 lists = range(0, iters_num, plot_interval)
55 plt.plot(lists, accuracies_train, label="training set")
56 plt.plot(lists, accuracies_test, label="test set")
57 plt.legend(loc="lower right")
58 plt.title("accuracy")
59 plt.xlabel("count")
60 plt.ylabel("accuracy")
61 plt.ylim(0, 1.0)
62 # グラフの表示
63 plt.show()
```

Generation: 1000. 正答率(トレーニング) = 1.0
: 1000. 正答率(テスト) = 0.7541

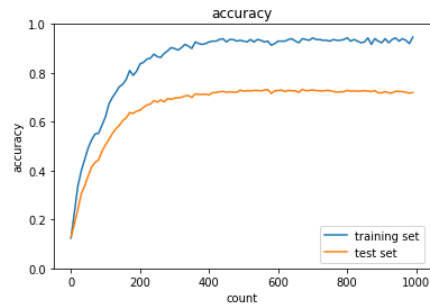


weight decay

L2

```
In [3]: 1 from common import optimizer
2
3 (x_train, d_train), (x_test, d_test) = load_mnist(normalize=True)
4
5 print("データ読み込み完了")
6
7 # 過学習を再現するために、学習データを削減
8 x_train = x_train[:300]
9 d_train = d_train[:300]
10
11
12 network = MultiLayerNet(input_size=784, hidden_size_list=[100, 100, 100, 100, 100, 100], output_size=10)
13
14
15 iters_num = 1000
16 train_size = x_train.shape[0]
17 batch_size = 100
18 learning_rate=0.01
19
20 train_loss_list = []
21 accuracies_train = []
22 accuracies_test = []
23
24 plot_interval=10
25 hidden_layer_num = network.hidden_layer_num
26
27 # 正則化強度設定 =====
28 weight_decay_lambda = 0.1
29 # =====
30
31 for i in range(iters_num):
32     batch_mask = np.random.choice(train_size, batch_size)
33
34     x_batch = x_train[batch_mask]
35     d_batch = d_train[batch_mask]
36
37     grad = network.gradient(x_batch, d_batch)
38     weight_decay = 0
39
40     for idx in range(1, hidden_layer_num+1):
41         grad['W' + str(idx)] = network.layers['Affine' + str(idx)].dw + weight_decay_lambda * network.params['W' + str(idx)]
42         grad['b' + str(idx)] = network.layers['Affine' + str(idx)].db
43         network.params['W' + str(idx)] -= learning_rate * grad['W' + str(idx)]
44         network.params['b' + str(idx)] -= learning_rate * grad['b' + str(idx)]
45         weight_decay += 0.5 * weight_decay_lambda * np.sqrt(np.sum(network.params['W' + str(idx)] ** 2))
46
47     loss = network.loss(x_batch, d_batch) + weight_decay
48     train_loss_list.append(loss)
49
50     if (i+1) % plot_interval == 0:
51         accr_train = network.accuracy(x_train, d_train)
52         accr_test = network.accuracy(x_test, d_test)
53         accuracies_train.append(accr_train)
54         accuracies_test.append(accr_test)
55
56         print('Generation: ' + str(i+1) + '. 正答率(トレーニング) = ' + str(accr_train))
57         print('          : ' + str(i+1) + '. 正答率(テスト) = ' + str(accr_test))
58
59 lists = range(0, iters_num, plot_interval)
60 plt.plot(lists, accuracies_train, label="training set")
61 plt.plot(lists, accuracies_test, label="test set")
62 plt.legend(loc="lower right")
63 plt.title("accuracy")
64 plt.xlabel("count")
65 plt.ylabel("accuracy")
66 plt.ylim(0, 1.0)
67 # グラフの表示
68 plt.show()
```

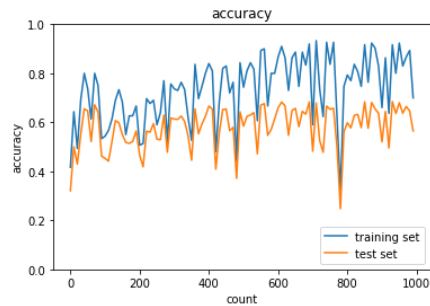
Generation: 1000. 正答率(トレーニング) = 0.9466666666666667
: 1000. 正答率(テスト) = 0.7199



L1

```
In [7]: 1 (x_train, d_train), (x_test, d_test) = load_mnist(normalize=True)
2
3 print("データ読み込み完了")
4
5 # 過学習を再現するために、学習データを削減
6 x_train = x_train[:300]
7 d_train = d_train[:300]
8
9 network = MultiLayerNet(input_size=784, hidden_size_list=[100, 100, 100, 100, 100, 100], output_size=10)
10
11
12 iters_num = 1000
13 train_size = x_train.shape[0]
14 batch_size = 100
15 learning_rate=0.1
16
17 train_loss_list = []
18 accuracies_train = []
19 accuracies_test = []
20
21 plot_interval=10
22 hidden_layer_num = network.hidden_layer_num
23
24 # 正則化強度設定 =====
25 #weight_decay_lambda = 0.005
26 weight_decay_lambda = 0.008
27 # =====
28
29 for i in range(iters_num):
30     batch_mask = np.random.choice(train_size, batch_size)
31     x_batch = x_train[batch_mask]
32     d_batch = d_train[batch_mask]
33
34     grad = network.gradient(x_batch, d_batch)
35     weight_decay = 0
36
37     for idx in range(1, hidden_layer_num+1):
38         grad['W' + str(idx)] = network.layers['Affine' + str(idx)].dW + weight_decay_lambda * np.sign(network.params['W' + str(idx)])
39         grad['b' + str(idx)] = network.layers['Affine' + str(idx)].db
40         network.params['W' + str(idx)] -= learning_rate * grad['W' + str(idx)]
41         network.params['b' + str(idx)] -= learning_rate * grad['b' + str(idx)]
42         weight_decay += weight_decay_lambda * np.sum(np.abs(network.params['W' + str(idx)]))
43
44     loss = network.loss(x_batch, d_batch) + weight_decay
45     train_loss_list.append(loss)
46
47     if (i+1) % plot_interval == 0:
48         accr_train = network.accuracy(x_train, d_train)
49         accr_test = network.accuracy(x_test, d_test)
50         accuracies_train.append(accr_train)
51         accuracies_test.append(accr_test)
52
53     print('Generation: ' + str(i+1) + '. 正答率(トレーニング) = ' + str(accr_train))
54     print('      : ' + str(i+1) + '. 正答率(テスト) = ' + str(accr_test))
55
56 lists = range(0, iters_num, plot_interval)
57 plt.plot(lists, accuracies_train, label="training set")
58 plt.plot(lists, accuracies_test, label="test set")
59 plt.legend(loc="lower right")
60 plt.title("accuracy")
61 plt.xlabel("count")
62 plt.ylabel("accuracy")
63 plt.ylim(0, 1.0)
64 # グラフの表示
65 plt.show()
```


Generation: 1000. 正答率(トレーニング) = 0.7
: 1000. 正答率(テスト) = 0.5648



[try] `weight_decay_lambda`の値を変更して正則化の強さを確認しよう

・深層学習 2 - 1 4 ~ 実装演習

実務においてはスピードが求められるので理論をしっかり理解し仮説を立てて検証することが必要
Dropout

```
In [8]: 1 class Dropout:
2         def __init__(self, dropout_ratio=0.5):
3             self.dropout_ratio = dropout_ratio
4             self.mask = None
5
6         def forward(self, x, train_flg=True):
7             if train_flg:
8                 self.mask = np.random.rand(*x.shape) > self.dropout_ratio
9                 return x * self.mask
10            else:
11                return x * (1.0 - self.dropout_ratio)
12
13        def backward(self, dout):
14            return dout * self.mask

In [10]: 1 from common import optimizer
2         (x_train, d_train), (x_test, d_test) = load_mnist(normalize=True)
3
4         print("データ読み込み完了")
5
6         # 過学習を再現するために、学習データを削減
7         x_train = x_train[:300]
8         d_train = d_train[:300]
9
10        # ドロップアウト設定 =====
11        use_dropout = True
12        # dropout_ratio = 0.15
13        dropout_ratio = 0.3
14        # =====
15
16        network = MultiLayerNet(input_size=784, hidden_size_list=[100, 100, 100, 100, 100, 100], output_size=10,
17                                weight_decay_lambda=weight_decay_lambda, use_dropout=use_dropout, dropout_ratio=dropout_ratio)
18        # optimizer = optimizer.SGD(learning_rate=0.01)
19        optimizer = optimizer.Momentum(learning_rate=0.01, momentum=0.9)
```

```

20 # optimizer = optimizer.AdaGrad(learning_rate=0.01)
21 # optimizer = optimizer.Adam()
22
23 iters_num = 1000
24 train_size = x_train.shape[0]
25 batch_size = 100
26
27 train_loss_list = []
28 accuracies_train = []
29 accuracies_test = []
30
31 plot_interval=10
32
33
34 for i in range(iters_num):
35     batch_mask = np.random.choice(train_size, batch_size)
36     x_batch = x_train[batch_mask]
37     d_batch = d_train[batch_mask]
38
39     grad = network.gradient(x_batch, d_batch)
40     optimizer.update(network.params, grad)
41
42     loss = network.loss(x_batch, d_batch)
43     train_loss_list.append(loss)
44
45     if (i+1) % plot_interval == 0:
46         accr_train = network.accuracy(x_train, d_train)
47         accr_test = network.accuracy(x_test, d_test)
48         accuracies_train.append(accr_train)
49         accuracies_test.append(accr_test)
50
51     print('Generation: ' + str(i+1) + '. 正答率(トレーニング) = ' + str(accr_train))
52     print('          : ' + str(i+1) + '. 正答率(テスト) = ' + str(accr_test))
53
54 lists = range(0, iters_num, plot_interval)
55 plt.plot(lists, accuracies_train, label="training set")

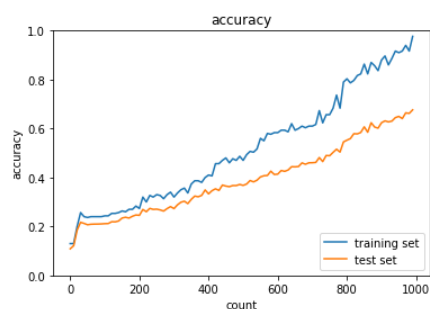
```

```

56 plt.plot(lists, accuracies_test, label="test set")
57 plt.legend(loc="lower right")
58 plt.title("accuracy")
59 plt.xlabel("count")
60 plt.ylabel("accuracy")
61 plt.ylim(0, 1.0)
62 # グラフの表示
63 plt.show()

```

Generation: 1000. 正答率(トレーニング) = 0.9766666666666667
: 1000. 正答率(テスト) = 0.6764



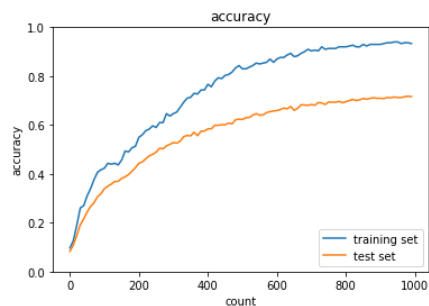
[try] dropout_ratioの値を変更してみよう

[try] optimizerとdropout_ratioの値を変更してみよう

Dropout + L1

```
In [11]: 1 from common import optimizer
2 (x_train, d_train), (x_test, d_test) = load_mnist(normalize=True)
3
4 print("データ読み込み完了")
5
6 # 過学習を再現するために、学習データを削減
7 x_train = x_train[:300]
8 d_train = d_train[:300]
9
10 # ドロップアウト設定 =====
11 use_dropout = True
12 dropout_ratio = 0.08
13 # =====
14
15 network = MultiLayerNet(input_size=784, hidden_size_list=[100, 100, 100, 100, 100, 100], output_size=10,
16 use_dropout = use_dropout, dropout_ratio = dropout_ratio)
17
18 iters_num = 1000
19 train_size = x_train.shape[0]
20 batch_size = 100
21 learning_rate=0.01
22
23 train_loss_list = []
24 accuracies_train = []
25 accuracies_test = []
26 hidden_layer_num = network.hidden_layer_num
27
28 plot_interval=10
29
30 # 正則化強度設定 =====
31 weight_decay_lambda=0.004
32 # =====
33
34 for i in range(iters_num):
35     batch_mask = np.random.choice(train_size, batch_size)
36     x_batch = x_train[batch_mask]
37     d_batch = d_train[batch_mask]
38
39     grad = network.gradient(x_batch, d_batch)
40     weight_decay = 0
41
42     for idx in range(1, hidden_layer_num+1):
43         grad['W' + str(idx)] = network.layers['Affine' + str(idx)].dW + weight_decay_lambda * np.sign(network.params['W' + str(idx)])
44         grad['b' + str(idx)] = network.layers['Affine' + str(idx)].db
45         network.params['W' + str(idx)] -= learning_rate * grad['W' + str(idx)]
46         network.params['b' + str(idx)] -= learning_rate * grad['b' + str(idx)]
47         weight_decay += weight_decay_lambda * np.sum(np.abs(network.params['W' + str(idx)]))
48
49     loss = network.loss(x_batch, d_batch) + weight_decay
50     train_loss_list.append(loss)
51
52     if (i+1) % plot_interval == 0:
53         accr_train = network.accuracy(x_train, d_train)
54         accr_test = network.accuracy(x_test, d_test)
55         accuracies_train.append(accr_train)
56         accuracies_test.append(accr_test)
57
58         print('Generation: ' + str(i+1) + ', 正答率(トレーニング) = ' + str(accr_train))
59         print('          : ' + str(i+1) + ', 正答率(テスト) = ' + str(accr_test))
60
61     lists = range(0, iters_num, plot_interval)
62     plt.plot(lists, accuracies_train, label="training set")
63     plt.plot(lists, accuracies_test, label="test set")
64     plt.legend(loc="lower right")
65     plt.title("accuracy")
66     plt.xlabel("count")
67     plt.ylabel("accuracy")
68     plt.ylim(0, 1.0)
69     # グラフの表示
70     plt.show()
```

Generation: 1000, 正答率(トレーニング) = 0.9333333333333333
: 1000, 正答率(テスト) = 0.716



Section 4) 畳み込みニューラルネットワークの概念

4-1 畳み込み層

- ・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
- ・画像より小さいサイズのフィルタを用意し、画像とフィルタの値を掛け合わせたものの総和を取る

4-1-1 バイアス

- ・総和を取った後、ニューラルネットワークの学習同様にバイアスを加算する

4-1-2 パディング

- ・入力画像と出力画像の大きさを合わせるため、入力画像の周囲にデータを付け足す
ゼロを付け足すゼロパディングが一般的（ゼロである必要性はない）

4-1-3 ストライド

- ・畳み込み演算の際にフィルタを移動させる刻み

4-1-4 チャンネル

- ・画像の奥行き
- ・チャンネル数分だけフィルタが用意される
- ・全結合で画像を学習した場合の課題
画像（縦、横、チャンネルの3次元）が1次元のデータとして処理される
→RGBの各チャンネルの関連性が学習に反映されない
→畳み込み層の登場
- ・深層学習 2-1 7～実装演習

simple convolution network

image to column

```
In [45]: 1 import sys, os
2 sys.path.append(os.pardir)
3 import pickle
4 import numpy as np
5 from collections import OrderedDict
6 from common import layers
7 from common import optimizer
8 from data.mnist import load_mnist
9 import matplotlib.pyplot as plt
10
11 # 画像データを2次元配列に変換
12 """
13 input_data: 入力値
14 filter_h: フィルタの高さ
15 filter_w: フィルタの横幅
16 stride: ストライド
17 pad: パディング
18 """
19 def im2col(input_data, filter_h, filter_w, stride=1, pad=0):
20     # N: number, C: channel, H: height, W: width
21     N, C, H, W = input_data.shape
22     out_h = (H + 2 * pad - filter_h) // stride + 1
23     out_w = (W + 2 * pad - filter_w) // stride + 1
24
25     img = np.pad(input_data, [(0,0), (0,0), (pad, pad), (pad, pad)], 'constant')
26     col = np.zeros((N, C, filter_h, filter_w, out_h, out_w))
27
28     for y in range(filter_h):
29         y_max = y + stride * out_h
30         for x in range(filter_w):
31             x_max = x + stride * out_w
32             col[:, :, y, x, :, :] = img[:, :, y:y_max:stride, x:x_max:stride]
33
34     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)
35
36     col = col.reshape(N * out_h * out_w, -1)
37     return col
```

[try] im2colの処理を確認しよう

- ・関数内でtransposeの処理をしている行をコメントアウトして下のコードを実行してみよう
- ・input_dataの各次元のサイズやフィルタサイズ・ストライド・パディングを変えてみよう

```
In [53]: 1 # im2colの処理確認
2 input_data = np.random.rand(3, 2, 4, 4) * 100 // 1 # number, channel, height, widthを表す
3 print('===== input_data =====\n', input_data)
4 print('=====')
5 filter_h = 3
6 filter_w = 3
7 stride = 1
8 pad = 0
9 col = im2col(input_data, filter_h=filter_h, filter_w=filter_w, stride=stride, pad=pad)
10 print('===== col =====\n', col)
11 print('=====')
```

```

===== input_data =====
[[[23. 88. 10. 61.]
 [70. 86. 54. 46.]
 [ 1. 66. 34. 78.]
 [70. 66. 81. 39.]]

 [[23. 51. 22. 38.]
 [26. 93. 25. 91.]
 [23. 31. 54. 82.]
 [72. 44. 38. 71.]]]

 [[43. 73. 52. 21.]
 [50. 55.  0. 48.]
 [ 7. 99. 84. 91.]
 [57. 12. 39. 94.]]

 [[27. 27. 22. 77.]
 [17. 71. 37. 74.]
 [47. 72. 43. 28.]
 [34. 84. 23. 60.]]]

 [[93. 25. 91. 41.]
 [41. 64. 67.  5.]
 [69.  7. 91. 92.]
 [94. 86. 50.  9.]]

 [[93. 15. 77. 54.]
 [ 1. 21. 44. 96.]
 [97. 58. 97. 63.]
 [80. 13. 89. 43.]]]]
=====

```

```

===== col =====
[[23. 88. 70. 86. 88. 10. 86. 54. 10. 61. 54. 46. 70. 86.  1. 66. 86. 54.]
 [66. 34. 54. 46. 34. 78.  1. 66. 70. 66. 66. 34. 66. 81. 34. 78. 81. 39.]
 [23. 51. 26. 93. 51. 22. 93. 25. 22. 38. 25. 91. 26. 93. 23. 31. 93. 25.]
 [31. 54. 25. 91. 54. 82. 23. 31. 72. 44. 31. 54. 44. 38. 54. 82. 38. 71.]
 [43. 73. 50. 55. 73. 52. 55.  0. 52. 21.  0. 48. 50. 55.  7. 99. 55.  0.]
 [99. 84.  0. 48. 84. 91.  7. 99. 57. 12. 99. 84. 12. 39. 84. 91. 39. 94.]
 [27. 27. 17. 71. 27. 22. 71. 37. 22. 77. 37. 74. 17. 71. 47. 72. 71. 37.]
 [72. 43. 37. 74. 43. 28. 47. 72. 34. 84. 72. 43. 84. 23. 43. 28. 23. 60.]
 [93. 25. 41. 64. 25. 91. 64. 67. 91. 41. 67.  5. 41. 64. 69.  7. 64. 67.]
 [ 7. 91. 67.  5. 91. 92. 69.  7. 94. 86.  7. 91. 86. 50. 91. 92. 50.  9.]
 [93. 15.  1. 21. 15. 77. 21. 44. 77. 54. 44. 96.  1. 21. 97. 58. 21. 44.]
 [58. 97. 44. 96. 97. 63. 97. 58. 80. 13. 58. 97. 13. 89. 97. 63. 89. 43.]]
=====

```

column to image

```

In [54]: 1 # 2次元配列を画像データに変換
          2 def col2im(col, input_shape, filter_h, filter_w, stride=1, pad=0):
          3     # N: number, C: channel, H: height, W: width
          4     N, C, H, W = input_shape
          5     # 切り捨て除算
          6     out_h = (H + 2 * pad - filter_h) // stride + 1
          7     out_w = (W + 2 * pad - filter_w) // stride + 1
          8     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)
          9
         10     img = np.zeros((N, C, H + 2 * pad + stride - 1, W + 2 * pad + stride - 1))
         11     for y in range(filter_h):
         12         y_max = y + stride * out_h
         13         for x in range(filter_w):
         14             x_max = x + stride * out_w
         15             img[:, :, y:y_max:stride, x:x_max:stride] += col[:, :, y, x, :, :]
         16
         17     return img[:, :, pad:H + pad, pad:W + pad]

```

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

```

In [55]: 1 img = col2im(col, input_shape=input_data.shape, filter_h=filter_h, filter_w=filter_w, stride=stride, pad=pad)
          2 print(img)

[[[ [23. 154. 104.  54.]
    [109. 216. 124. 103.]
    [179. 197. 152. 152.]
    [ 93.  48.  53.  72.]]

 [[ 61. 120. 112.  34.]
 [108. 221. 204.  88.]
 [ 92. 301. 196.  93.]
 [ 31. 175.  63.  71.]]]

```

```

[[[ 43. 172. 134.  0.]
  [ 82. 220. 196. 128.]
  [126. 108. 216.  85.]
  [ 71.  84.  94.  34.]]

[[ 21.  12. 147.  84.]
 [127. 188. 192. 127.]
 [116. 301. 109. 137.]
 [ 72.  99.  60.  60.]]

[[[ 93.  32. 132.  67.]
  [157. 103. 280. 136.]
  [ 85. 247. 272. 157.]
  [ 21. 141. 135.  80.]]

[[ 41. 153.  12.  91.]
 [ 95. 207. 273. 188.]
 [  8. 190. 303. 106.]
 [ 58.  84. 133.  43.]]]]

```

4-2 プーリング層

- ・MAXプーリング、AVGプーリングの2種類がある
 - ・対象領域のMAX値または平均値を取得
 - ・深層学習2-19～実装演習
- 実行速度の改善はGPUで実行／CPUをグレードアップなど検討すること

convolution class

```

In [10]: 1 class Convolution:
2         # W: フィルター, b: バイアス
3         def __init__(self, W, b, stride=1, pad=0):
4             self.W = W
5             self.b = b
6             self.stride = stride
7             self.pad = pad
8
9         # 中間データ (backward時に使用)
10        self.x = None
11        self.col = None
12        self.col_W = None
13
14        # フィルター・バイアスパラメータの勾配
15        self.dW = None
16        self.db = None
17
18        def forward(self, x):
19            # FN: filter_number, C: channel, FH: filter_height, FW: filter_width
20            FN, C, FH, FW = self.W.shape
21            N, C, H, W = x.shape
22            # 出力値のheight, width
23            out_h = 1 + int((H + 2 * self.pad - FH) / self.stride)
24            out_w = 1 + int((W + 2 * self.pad - FW) / self.stride)
25
26            # xを行列に変換
27            col = im2col(x, FH, FW, self.stride, self.pad)
28            # フィルターをxに合わせた行列に変換
29            col_W = self.W.reshape(FN, -1).T
30
31            out = np.dot(col, col_W) + self.b
32            # 計算のために変えた形式を戻す
33            out = out.reshape(N, out_h, out_w, -1).transpose(0, 3, 1, 2)
34

```

```

35        self.x = x
36        self.col = col
37        self.col_W = col_W
38
39        return out
40
41        def backward(self, dout):
42            FN, C, FH, FW = self.W.shape
43            dout = dout.transpose(0, 2, 3, 1).reshape(-1, FN)
44
45            self.db = np.sum(dout, axis=0)
46            self.dW = np.dot(self.col.T, dout)
47            self.dW = self.dW.transpose(1, 0).reshape(FN, C, FH, FW)
48
49            dcol = np.dot(dout, self.col_W.T)
50            # dcolを画像データに変換
51            dx = col2im(dcol, self.x.shape, FH, FW, self.stride, self.pad)
52
53            return dx
54

```

pooling class

```
In [11]: 1 class Pooling:
2     def __init__(self, pool_h, pool_w, stride=1, pad=0):
3         self.pool_h = pool_h
4         self.pool_w = pool_w
5         self.stride = stride
6         self.pad = pad
7
8         self.x = None
9         self.arg_max = None
10
11     def forward(self, x):
12         N, C, H, W = x.shape
13         out_h = int(1 + (H - self.pool_h) / self.stride)
14         out_w = int(1 + (W - self.pool_w) / self.stride)
15
16         # xを行列に変換
17         col = im2col(x, self.pool_h, self.pool_w, self.stride, self.pad)
18         # プーリングのサイズに合わせてリサイズ
19         col = col.reshape(-1, self.pool_h*self.pool_w)
20
21         # 行ごとに最大値を求める
22         arg_max = np.argmax(col, axis=1)
23         out = np.max(col, axis=1)
24         # 整形
25         out = out.reshape(N, out_h, out_w, C).transpose(0, 3, 1, 2)
26
27         self.x = x
28         self.arg_max = arg_max
29
30         return out
31
32     def backward(self, dout):
33         dout = dout.transpose(0, 2, 3, 1)
34
35         pool_size = self.pool_h * self.pool_w
36         dmax = np.zeros((dout.size, pool_size))
37         dmax[np.arange(self.arg_max.size), self.arg_max.flatten()] = dout.flatten()
38         dmax = dmax.reshape(dout.shape + (pool_size,))
39
40         dcol = dmax.reshape(dmax.shape[0] * dmax.shape[1] * dmax.shape[2], -1)
41         dx = col2im(dcol, self.x.shape, self.pool_h, self.pool_w, self.stride, self.pad)
42
43         return dx
44
```

simple convolution network class

```
In [12]: 1 class SimpleConvNet:
2     # conv - relu - pool - affine - relu - affine - softmax
3     def __init__(self, input_dim=(1, 28, 28), conv_param={'filter_num':30, 'filter_size':5, 'pad':0, 'stride':1},
4         hidden_size=100, output_size=10, weight_init_std=0.01):
5         filter_num = conv_param['filter_num']
6         filter_size = conv_param['filter_size']
7         filter_pad = conv_param['pad']
8         filter_stride = conv_param['stride']
9         input_size = input_dim[1]
10        conv_output_size = (input_size - filter_size + 2 * filter_pad) / filter_stride + 1
11        pool_output_size = int(filter_num * (conv_output_size / 2) * (conv_output_size / 2))
12
13        # 重みの初期化
14        self.params = {}
15        self.params['W1'] = weight_init_std * np.random.randn(filter_num, input_dim[0], filter_size, filter_size)
16        self.params['b1'] = np.zeros(filter_num)
17        self.params['W2'] = weight_init_std * np.random.randn(pool_output_size, hidden_size)
18        self.params['b2'] = np.zeros(hidden_size)
19        self.params['W3'] = weight_init_std * np.random.randn(hidden_size, output_size)
20        self.params['b3'] = np.zeros(output_size)
21
22        # レイヤの生成
23        self.layers = OrderedDict()
24        self.layers['Conv1'] = layers.Convolution(self.params['W1'], self.params['b1'], conv_param['stride'], conv_param['pad'])
25        self.layers['Relu1'] = layers.ReLU()
26        self.layers['Pool1'] = layers.Pooling(pool_h=2, pool_w=2, stride=2)
27        self.layers['Affine1'] = layers.Affine(self.params['W2'], self.params['b2'])
28        self.layers['Relu2'] = layers.ReLU()
29        self.layers['Affine2'] = layers.Affine(self.params['W3'], self.params['b3'])
30
31        self.last_layer = layers.SoftmaxWithLoss()
32
33    def predict(self, x):
34        for key in self.layers.keys():
35            x = self.layers[key].forward(x)
36        return x
37
38    def loss(self, x, d):
39        y = self.predict(x)
40        return self.last_layer.forward(y, d)
41
```

```

42 def accuracy(self, x, d, batch_size=100):
43     if d.ndim != 1: d = np.argmax(d, axis=1)
44
45     acc = 0.0
46
47     for i in range(int(x.shape[0] / batch_size)):
48         tx = x[(i*batch_size):(i+1)*batch_size]
49         td = d[(i*batch_size):(i+1)*batch_size]
50         y = self.predict(tx)
51         y = np.argmax(y, axis=1)
52         acc += np.sum(y == td)
53
54     return acc / x.shape[0]
55
56 def gradient(self, x, d):
57     # forward
58     self.loss(x, d)
59
60     # backward
61     dout = 1
62     dout = self.last_layer.backward(dout)
63     layers = list(self.layers.values())
64
65     layers.reverse()
66     for layer in layers:
67         dout = layer.backward(dout)
68
69     # 設定
70     grad = {}
71     grad['W1'], grad['b1'] = self.layers['Conv1'].dW, self.layers['Conv1'].db
72     grad['W2'], grad['b2'] = self.layers['Affine1'].dW, self.layers['Affine1'].db
73     grad['W3'], grad['b3'] = self.layers['Affine2'].dW, self.layers['Affine2'].db
74
75     return grad

```

```

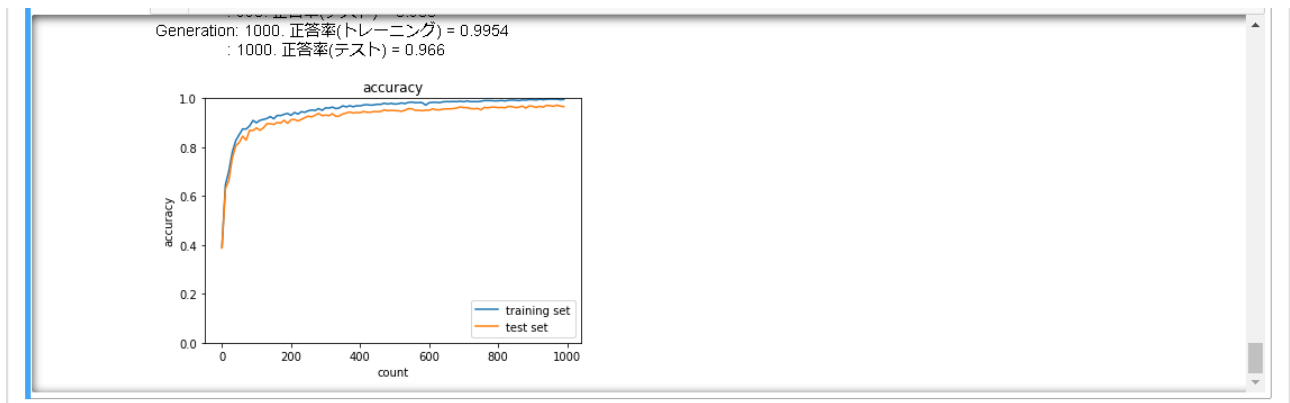
In [13]: 1 from common import optimizer
2
3 # データの読み込み
4 (x_train, d_train), (x_test, d_test) = load_mnist(flatten=False)
5
6 print("データ読み込み完了")
7
8 # 処理に時間のかかる場合はデータを削減
9 x_train, d_train = x_train[:5000], d_train[:5000]
10 x_test, d_test = x_test[:1000], d_test[:1000]
11
12
13 network = SimpleConvNet(input_dim=(1,28,28), conv_param = {'filter_num': 30, 'filter_size': 5, 'pad': 0, 'stride': 1},
14                        hidden_size=100, output_size=10, weight_init_std=0.01)
15
16 optimizer = optimizer.Adam()
17
18 iters_num = 1000
19 train_size = x_train.shape[0]
20 batch_size = 100
21
22 train_loss_list = []
23 accuracies_train = []
24 accuracies_test = []
25
26 plot_interval=10
27
28
29
30 for i in range(iters_num):
31     batch_mask = np.random.choice(train_size, batch_size)
32     x_batch = x_train[batch_mask]
33     d_batch = d_train[batch_mask]
34
35     grad = network.gradient(x_batch, d_batch)
36     optimizer.update(network.params, grad)
37

```

```

38 loss = network.loss(x_batch, d_batch)
39 train_loss_list.append(loss)
40
41 if (i+1) % plot_interval == 0:
42     accr_train = network.accuracy(x_train, d_train)
43     accr_test = network.accuracy(x_test, d_test)
44     accuracies_train.append(accr_train)
45     accuracies_test.append(accr_test)
46
47     print('Generation: ' + str(i+1) + '. 正答率(トレーニング) = ' + str(accr_train))
48     print('              : ' + str(i+1) + '. 正答率(テスト) = ' + str(accr_test))
49
50 lists = range(0, iters_num, plot_interval)
51 plt.plot(lists, accuracies_train, label="training set")
52 plt.plot(lists, accuracies_test, label="test set")
53 plt.legend(loc="lower right")
54 plt.title("accuracy")
55 plt.xlabel("count")
56 plt.ylabel("accuracy")
57 plt.ylim(0, 1.0)
58 # グラフの表示
59 plt.show()

```

Section 5) 最新のCNN

5-1 AlexNet

- 2012年開催の画像認識コンペティションで2位に大差をつけて優勝しディープラーニングが大きく注目を集めるきっかけとなったモデル
- 5層の畳み込み層・プーリング層およびそれに続く3層の全結合層から構成される
- 過学習を抑制する施策としてドロップアウトを使用している