深層学習day1-2　レポート

深層学習day1　確認テスト

1.
入力と出力の間に中間層を設けて、それぞれのパーセプトロンの重みに特徴を学習させることで、モデルを作成しようとしている。

重み　バイアス

2.
（省）

3.
入力層の一つ一つが動物の特徴になる。

4.
u = np.dot(W,x) + b

5.
```
# 2層の総出力
z2 = functions.relu(u2)
```

6.
入力と出力が比例関係にない方が、非線形。

7.
```
functions.relu(u1)
```

8.
二乗することで、誤差の正負を排して評価できる。
1/2は微分した際に形を単純にするための係数である。

9.
```
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))
```

10.
```
def cross_entropy_error(d, y):
    ify.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
```

11.
```
network[key]  -= learning_rate * grad[key]
```

12.
逐次来るデータに対して学習し、モデルを更新していく手法。

13.
重み-誤差関数のグラフに対して、最小値を目指すため、現在地点の重みに対する偏微分の値から関数の傾きを導き、誤差の小さい重みの方へ、学習率の分だけ重みを更新している。

14.
```
delta1 = np.dot(delta2, W2.T) * functions.d_sigmoid(z1)
```

15.
```
delta2 = functions.d_mean_squared_error(d, y)


grad['W2'] = np.dot(z1.T, delta2)
```

深層学習day1　要点まとめ

・Section 1　入力層～中間層
　入力層から中間層への変換は特に変わったことはなく、特徴量に対して重みとバイアスを用いて線形に変換を行う。一般にはそれによって得られた出力を活性化関数によって特徴を分かりやすくする処理がおこなわれる。中間層が多いほど説明力が増す。

・Section 2　活性化関数
　活性化関数にも種類があり、得意不得意がある。ロジスティック回帰にはよくシグモイド関数が用いられていたが、NNのように重みの数が多くなってくると処理が重くなったり、学習の効率が悪くなったりすることから、代わりにReLU関数などが使われる。

・Section 3　出力層
　出力層で出た値は、誤差の評価を含め、推論結果は人間が扱う。中間層とは違って、人間が見て正確に理解しやすいデータとするために、ソフトマックス関数のような、確率として返す活性化関数が用いられる場合がある。

・Section 4　勾配降下法
　重みやバイアスの値を更新するために勾配降下法を用いる。大きく分けると、偏微分をして傾きを出すステップと、学習率分値を更新するステップがあるが、それぞれ学習するデータの特性に合わせて工夫される。たとえば、学習の数が多い場合、偏微分をまとめて行わず、バッチごとに行う。また、誤差の最小値にうまくたどり着かない場合は、学習率や、学習の仕方のアルゴリズムに手を加える。

・Section 5　誤差逆伝播法
　NNのように層を重ねるようになると、出力から遠い層の重みを更新する際に工夫が必要になる。具体的には勾配を逆伝播して再帰的な処理を行わずに重みを更新する。この際に、層が厚くなればなるほど、勾配の値が小さくなっていく、勾配消失問題が発生するため、特に活性化関数を工夫する場合が大い。

深層学習day2　確認テスト

1.
dz/dx = 2

2.
(2)

3.
順伝播の際にその重みに関する特徴が無視され、結果的に学習されなくなる。

4.
特徴量の大きさを揃えられるので、学習の際に特徴量による有利不利が出ない。
勾配学習の際に、極端な爆発や消失をある程度抑えられる。

5.
モメンタム：勾配学習に加速度の概念を取り入れていて、前のステップの学習の方向に進みやすい。

AdaGrad：徐々に更新の幅が小さくなっていくので、緩やかな傾きの場合最小値にたどり着きやすい。一方、鞍点問題も起こりやすい。

RMSprop：AdaGradを改良したもので、急速な学習率の低下を抑えられる。

6.
右

7.
７×７

深層学習day2　要点まとめ

・Section 1　勾配消失問題
　誤差逆伝播をして重みを学習していく際、層が厚くなるほど、誤差が伝播しにくくなる。そのため、誤差関数の偏微分の値が小さくなりすぎないように、ロジスティック回帰などのようにシグモイド関数を使わず、ReLU関数がよく使われる。また、重みがそもそも小さいと、逆伝播する際にそれより前のノードの学習がうまくいかないこともあるので、重みの初期値を工夫する必要がある。

・Section 2　学習率最適化手法
　学習率は一般的には誤差関数の傾きにかけて重みを更新する時に利用し、常に一定であることが多いが、実際に適用すると、学習がうまく収束しないことや、極小値に収束してしまうことがある。それに対し、モメンタムでは、学習率に加速度の概念を導入し、収束を早めたり、極小値を超えられるように工夫されている。AdaGradやRMSpropは最小値付近での収束をよくし、Adamではその両方の利点を生かせるようなアルゴリズムである。

・Section 3　過学習
　過学習とは、準備した訓練用データに特化しすぎたモデルを作ってしまうことによって、汎化性能がおちてしまうことのことである。過学習であることを判定する明確なラインはないが、パラメーターの更新に対して、テスト誤差と訓練誤差の差が開き出したらその可能性がある。これを避けるために、重みによるモデルの表現力を抑えられるように正則化項を誤差関数に導入したり、ノード数をランダムに落としながら学習させる方法がある。

・Section 4　畳み込みニューラルネットワークの概念
　CNNというと、画像の学習に用いられるイメージだったが、実際には、１次元データから３次元データまで、様々な活用が考えられる。層の構成を大きく分けると、コンボリューション層とプーリング層に分かれており、最終的に出力層にて、得たい推論を得る。画像であれば、距離が近いピクセルのデータとの関係が特徴に影響することは明らかであり、そのような特徴をすくいだせるような工夫がなされているアルゴリズムである。

・Section 5　最新のCNN
　AlexNetの説明があった。ILSVRCにて飛躍的な成績を残した学習手法である。3つのコンボリューション層と２つのプーリング層からなる。画像の水増し技術や、ドロップアウトも取り入れられていたそうだ。しかし、翌年には成績は抜かれて、今ではあまり使われていないという。

Section 1 入力層～中間層

Section 2 活性化関数

Section 3 出力層

In [ ]:

```
```

In [ ]:

```
import numpy as np
from common import functions
```

In [20]:

```
def print_vec(text, vec):
    print("*** " + text + " ***")
    print(vec)
    print("shape: " + str(vec.shape))
    print("")
```

In [25]:

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

## 試してみよう_配列の初期化
# W = np.zeros(2)
#W = np.ones(2)
#W = np.random.rand(2)
W = np.random.randint(5, size=(2))

print_vec("重み", W)
```

```
*** 重み ***
[2 3]
shape: (2,)
```

In [22]:

```
# 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)
```

```
*** バイアス ***
0.5
shape: ()
```

In [32]:

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

# 中間層出力:sigmoid
```

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

```
*** 入力 ***
[ 2 -3]
shape: (2,)

*** 総入力 ***
-4.5
shape: ()

*** 中間層出力 ***
0.0
shape: ()

*** 中間層出力 ***
0.01098694263059318
shape: ()
```

In [30]:

```
# 多クラス分類
def init_network():
    print("##### ネットワークの初期化 #####")

    network = {}

    input_layer_size = 3
    hidden_layer_size = 5
    output_layer_size = 6

    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

def forward(network, x):

    print("##### 順伝播開始 #####")
    W1, W2 = network['W1'], network['W2']
    b1, b2 = network['b1'], network['b2']

    # 1層の総入力
    u1 = np.dot(x, W1) + b1

    # 1層の総出力
    z1 = functions.relu(u1)

    # 2層の総入力
    u2 = np.dot(z1, W2) + b2

    # 出力値
#     y = functions.softmax(u2)
    y = 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
```

In [31]:

```
## 事前データ
# 入力値
x = np.array([1., 2., 3.])

# 目標出力
d = np.array([0, 0, 0, 1, 0, 0])

# ネットワークの初期化
network =  init_network()

# 出力
y, z1 = forward(network, x)

# 誤差
loss = functions.cross_entropy_error(d, y)

## 表示
print("¥n##### 結果表示 #####")
print_vec("出力", y)
print_vec("訓練データ", d)
print_vec("交差エントロピー誤差",  loss)
```

```
##### ネットワークの初期化 #####
*** 重み1 ***
[[0.92815961 0.13142438 0.33857123 0.20504201 0.68471141]
 [0.69614507 0.24308604 0.78286638 0.40417049 0.16572272]
 [0.93904484 0.09609288 0.90775857 0.05073835 0.362631  ]]
shape: (3, 5)

*** 重み2 ***
[[0.608667   0.87846321 0.80330869 0.16546431 0.23737385 0.78854372]
 [0.0982093  0.61263932 0.90076926 0.95694635 0.00491237 0.18284198]
 [0.99535831 0.95086403 0.46396797 0.18442008 0.53046818 0.13520954]
 [0.24123805 0.96859557 0.58769259 0.14958729 0.34256739 0.4508791 ]
 [0.31395047 0.85910494 0.15362558 0.42611879 0.50723875 0.81097681]]
shape: (5, 6)

*** バイアス1 ***
[0.00772914 0.38419433 0.01764879 0.22818145 0.80761383]
shape: (5,)

*** バイアス2 ***
[0.47170142 0.02003014 0.1743918  0.33042609 0.19827644 0.6504457 ]
shape: (6,)

##### 順伝播開始 #####
*** 総入力1 ***
[5.14531342 1.29006942 4.6452285  1.39377949 2.91166368]
shape: (5,)

*** 中間層出力1 ***
[5.14531342 1.29006942 4.6452285  1.39377949 2.91166368]
shape: (5,)

*** 総入力2 ***
[ 9.60419831 13.59875991  8.89137879  4.72219877  5.84449454  8.56142609]
shape: (6,)

*** 出力1 ***
[ 9.60419831 13.59875991  8.89137879  4.72219877  5.84449454  8.56142609]
shape: (6,)

出力合計: 51.22245640673955

##### 結果表示 #####
*** 出力 ***
[ 9.60419831 13.59875991  8.89137879  4.72219877  5.84449454  8.56142609]
shape: (6,)
```

```
*** 訓練データ ***
[0 0 0 1 0 0]
shape: (6,)

*** 交差エントロピー誤差 ***
-1.552274552592688
shape: ()
```

様々なパターンの順伝播を試した。中間層の層の数でかなり表現力を増すことが確認できた。
人間に近い理解の仕方をするなら、入力の特徴量を中間層でさらに細かい特徴に分解して、得たい出力に向けて出力層で再整理するといった感じだろうか。
また、活性化関数によって変わる出力値の特徴も把握できた。

Section 4 勾配降下法

Section 5 誤差逆伝播法

In [ ]:

In [2]:
```
import numpy as np
from common import functions
import matplotlib.pyplot as plt
```

In [158]:
```
def print_vec(text, vec):
    print("*** " + text + " ***")
    print(vec)
    #print("shape: " + str(x.shape))
    print("")

def init_network(firstNode=2,nodesNum=3,lastNode=2):
    print("##### ネットワークの初期化 #####")

    network = {}
#     network['W1'] = np.array([
#         [0.1, 0.3, 0.5],
#         [0.2, 0.4, 0.6]
#     ])

#     network['W2'] = np.array([
#         [0.1, 0.4],
#         [0.2, 0.5],
#         [0.3, 0.6]
#     ])

#     network['b1'] = np.array([0.1, 0.2, 0.3])
#     network['b2'] = np.array([0.1, 0.2])


    network['W1'] = np.random.randn(firstNode, nodesNum)
    network['W2'] = np.random.randn(nodesNum,lastNode)
    network['b1'] = np.random.randn(nodesNum)
    network['b2'] = np.random.randn(lastNode)

    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

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)
    y = 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
```

In [159]:

```
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)
    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)
#     print_vec("偏微分_dE/du2", delta2)
    #  中間層でのデルタ
    delta1 = np.dot(delta2, W2.T) * functions.d_relu(z1)
#     print_vec("偏微分_dE/du2", delta1)
    # 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_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
```

In [156]:

```
# 訓練データ
x = np.array([[1.0, 5.0]])
# 目標出力
d = np.array([[0, 1]])
#  学習率
learning_rate = 0.01
network =  init_network()
y, z1 = forward(network, x)

# 誤差
loss1 = functions.cross_entropy_error(d, y)
print_vec("loss1",loss1)

grad = backward(x, d, z1, y)
for key in ('W1', 'W2', 'b1', 'b2'):
    network[key]  -= learning_rate * grad[key]

print("##### 結果表示 #####")

y2, z12 = forward(network, x)
loss2 = functions.cross_entropy_error(d, y2)
print_vec("loss2",loss2)

print("##### 更新後パラメータ #####")
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'])
```

```
##### ネットワークの初期化 #####
*** 重み1 ***
[[ 0.19187865  0.77879186  0.26868155]
 [ 0.03103215 -0.91146877  0.94964822]]

*** 重み2 ***
[[-0.35626312  1.31509972]
 [ 0.50553098  0.66737735]
 [-0.80296317  1.47771501]]

*** バイアス1 ***
[-0.1007849  -0.34950915  1.47603096]

*** バイアス2 ***
[ 0.58582217 -1.12216349]

##### 順伝播開始 #####
*** loss1 ***
-2.1743449096365244


##### 誤差逆伝播開始 #####
*** 偏微分_dE/du2 ***
[[-4.71551183  7.79642068]]

*** 偏微分_dE/du2 ***
[[11.93303361  0.          15.30727018]]

##### 結果表示 #####
##### 順伝播開始 #####
*** loss2 ***
-0.08855679170247827

##### 更新後パラメータ #####
*** 重み1 ***
[[ 0.07254832  0.77879186  0.11560885]
 [-0.56561953 -0.91146877  0.18428471]]

*** 重み2 ***
[[-0.34465096  1.29590068]
 [ 0.50553098  0.66737735]
 [-0.49678718  0.97149703]]

*** バイアス1 ***
[-0.22011524 -0.34950915  1.32295826]

*** バイアス2 ***
[ 0.63297728 -1.20012769]
```

In [166]:

```
# サンプルデータを作成
data_sets_size = 100000
data_sets = [0 for i in range(data_sets_size)]

def f(x):
    y = 3 * x[0]**2 + 2 * x[1]
    return y

for i in range(data_sets_size):
    data_sets[i] = {}
    # ランダムな値を設定
    data_sets[i]['x'] = np.random.rand(2)

    ## 試してみよう_入力値の設定
    # data_sets[i]['x'] = np.random.rand(2) * 10 -5 # -5〜5のランダム数値

    # 目標出力を設定
    data_sets[i]['d'] = f(data_sets[i]['x'])

losses = []
```

```python
# 学習率
learning_rate = 0.06

# 抽出数
epoch = 2000

# パラメータの初期化
network = init_network(2,30,1)
# データのランダム抽出
random_datasets = np.random.choice(data_sets, epoch)

# 勾配降下の繰り返し
for dataset in random_datasets:
    x_, d = dataset['x'], dataset['d']
    x = x_[np.newaxis,:]
    y, z1 = 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]

    # 誤差
    loss = functions.mean_squared_error(d, y)
    losses.append(loss)

print("##### 結果表示 #####")
lists = range(epoch)
```

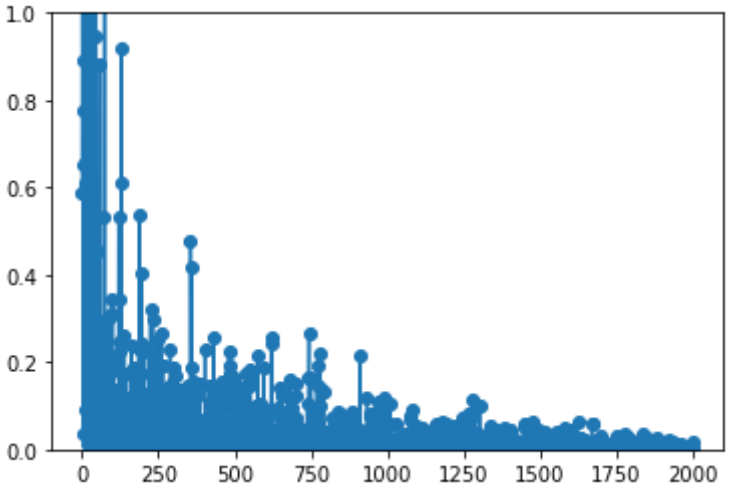
```
##### ネットワークの初期化 #####
*** 重み1 ***
[[-0.29225796  0.98508164 -0.66924177  0.0797783  -0.29798726  0.41714759
   0.37856158 -0.68891242 -1.21566482 -0.06857517 -1.7039425  -0.84314487
   0.31922425 -0.27871193  1.33532816  0.46667898  1.00766694 -1.44217745
  -0.42818187  2.0148607   1.77243793  0.10978657  0.22997063  0.75892031
  -1.57062634 -0.34456892 -1.21070746 -1.35272473 -1.87981953 -0.3581625 ]
 [-0.54937048  1.8824244  -0.7609003  -0.24438728 -0.42452626 -0.07177102
   0.52240615 -0.77084818  1.99244244  1.19102427  0.03536658  1.58262634
  -0.86703823 -0.67720919 -1.00270158 -0.08858491 -1.04532776 -0.54855378
  -0.97735225 -0.75634613  0.74379505  0.26416403  0.9289189  -0.55394556
   1.70708708 -3.35481451 -0.24121956  1.30286261  0.27380861 -0.83096446]]

*** 重み2 ***
[[ 1.75963189]
 [-0.4405648 ]
 [-0.30688098]
 [-0.80620621]
 [-0.21492496]
 [ 0.68255321]
 [ 0.58052748]
 [-0.55477861]
 [ 0.47929342]
 [ 0.32963686]
 [ 1.14140947]
 [-0.10420234]
 [-0.77242259]
 [ 0.99804377]
 [ 2.97693014]
 [ 0.76991671]
 [ 0.46399628]
 [-0.01369808]
 [ 0.24046419]
 [-1.08007195]
 [ 0.93576496]
 [ 1.15050472]
 [-0.76344467]
 [-0.1618647 ]
 [ 1.25038375]
 [-0.26680341]
 [ 0.5728745 ]
 [-1.12976932]
 [-0.48968021]
 [-0.73372867]]
```

```
*** バイアス1 ***
[-0.40284152 -0.30447391 -1.88348317 -0.39818607  0.79061978  1.54590798
  1.51136967 -0.30174109  0.20518255 -0.48310866 -0.18386156 -0.27389653
  0.59473159 -0.98322315 -0.01445675  0.92911145 -0.19781549 -1.61001579
 -1.35253134 -1.24285198  0.962801   -1.70197853  0.26481178 -0.06497598
  0.93479946  0.66936422 -2.99846437  1.81815816  0.52769831  1.10339841]

*** バイアス2 ***
[-0.74543463]

##### 結果表示 #####
```

In [167]:

```
plt.plot(lists, losses, 'o-')
plt.ylim([0,1])
```

Out[167]: (0, 1)

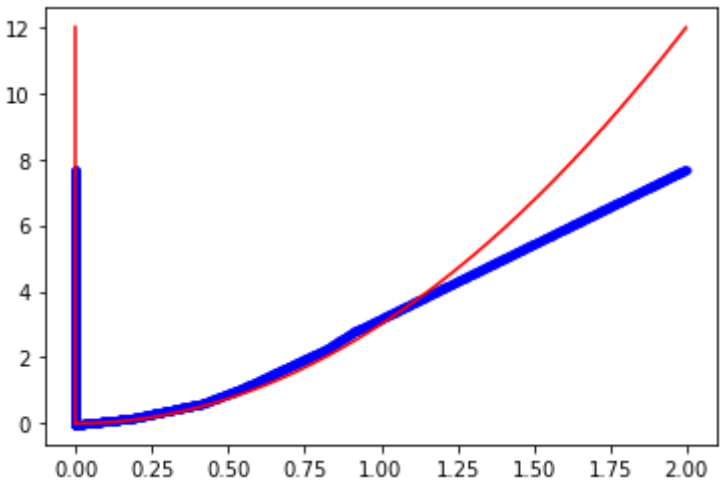


In [168]:

```
test = [[i/500,0] for i in range(1000)]
test_y = [3*(i/500)**2  for i in range(1000)]
y = np.zeros(1000)
z1 = np.zeros(1000)
for j in range(1000):
    y_,z1_ = forward(network,test[j])
    y[j] = y_
plt.plot(test,y,'b.')
plt.plot(test,test_y,'r-')
```

Out[168]:

```
[<matplotlib.lines.Line2D at 0x1a14f1bd6c8>,
 <matplotlib.lines.Line2D at 0x1a14f1a9488>]
```



勾配降下法を用いて、誤差を逆伝播させる学習を行った。また、NNのパラメーターを変えながらの考察も行った。
最後に行ったのは、関数が二次の形でも推測できるかどうか検証した。結果的にわかったの

は、中間層を増やすと、二次特融の曲線の形も徐々に表現できるようになるということと、学習のデータ範囲に依存するということだ。
あらかじめ関数の次数がわかっているなどの知見がある場合は、わざわざNNを使う必要はない
一方、かなりいろいろなデータに追従することがわかった。

In [ ]:

Section 1 勾配消失問題

In [ ]:

In [2]:

```
import numpy as np
from common import layers
from collections import OrderedDict
from common import functions
from data.mnist import load_mnist
import matplotlib.pyplot as plt


class MultiLayerNet:
    '''
    input_size: 入力層のノード数
    hidden_size_list: 隠れ層のノード数のリスト
    output_size: 出力層のノード数
    activation: 活性化関数
    weight_init_std: 重みの初期化方法
    '''
    def __init__(self, input_size, hidden_size_list, output_size, activation='relu',
        self.input_size = input_size
        self.output_size = output_size
        self.hidden_size_list = hidden_size_list
        self.hidden_layer_num = len(hidden_size_list)
        self.params = {}

        # 重みの初期化
        self.__init_weight(weight_init_std)

        # レイヤの生成, sigmoidとreluのみ扱う
        activation_layer = {'sigmoid': layers.Sigmoid, 'relu': layers.Relu}
        self.layers = OrderedDict() # 追加した順番に格納
        for idx in range(1, self.hidden_layer_num+1):
            self.layers['Affine' + str(idx)] = layers.Affine(self.params['W' + str(id
            self.layers['Activation_function' + str(idx)] = activation_layer[activati

        idx = self.hidden_layer_num + 1
        self.layers['Affine' + str(idx)] = layers.Affine(self.params['W' + str(idx)],

        self.last_layer = layers.SoftmaxWithLoss()

    def __init_weight(self, weight_init_std):
        all_size_list = [self.input_size] + self.hidden_size_list + [self.output_size
        for idx in range(1, len(all_size_list)):
            scale = weight_init_std
            if str(weight_init_std).lower() in ('relu', 'he'):
                scale = np.sqrt(2.0 / all_size_list[idx - 1])
            elif str(weight_init_std).lower() in ('sigmoid', 'xavier'):
                scale = np.sqrt(1.0 / all_size_list[idx - 1])

            self.params['W' + str(idx)] = scale * np.random.randn(all_size_list[idx-1
            self.params['b' + str(idx)] = np.zeros(all_size_list[idx])

    def predict(self, x):
        for layer in self.layers.values():
            x = layer.forward(x)

        return x

    def loss(self, x, d):
        y = self.predict(x)
```

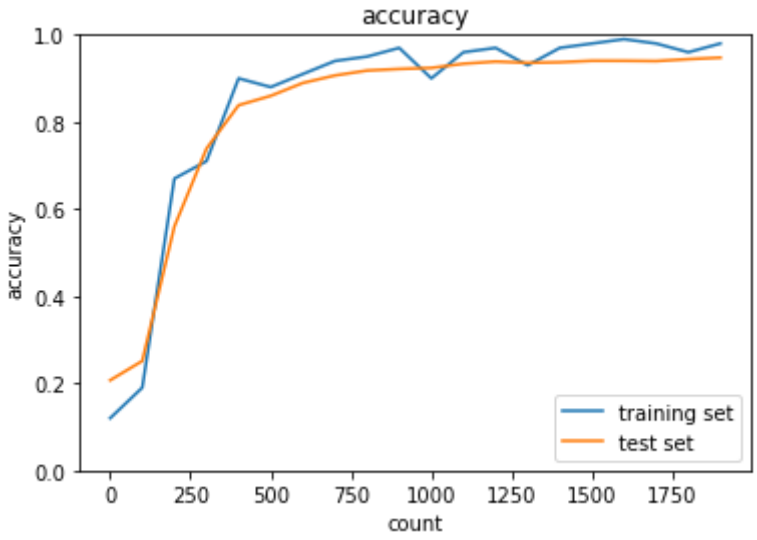
```
        weight_decay = 0
        for idx in range(1, self.hidden_layer_num + 2):
            W = self.params['W' + str(idx)]

        return self.last_layer.forward(y, d) + weight_decay

    def accuracy(self, x, d):
        y = self.predict(x)
        y = np.argmax(y, axis=1)
        if d.ndim != 1 : d = np.argmax(d, axis=1)

        accuracy = np.sum(y == d) / float(x.shape[0])
        return accuracy

    def gradient(self, x, d):
        # forward
        self.loss(x, d)

        # backward
        dout = 1
        dout = self.last_layer.backward(dout)

        layers = list(self.layers.values())
        layers.reverse()
        for layer in layers:
            dout = layer.backward(dout)

        # 設定
        grad = {}
        for idx in range(1, self.hidden_layer_num+2):
            grad['W' + str(idx)] = self.layers['Affine' + str(idx)].dW
            grad['b' + str(idx)] = self.layers['Affine' + str(idx)].db

        return grad
```

In [19]:

```
# データの読み込み
(x_train, d_train), (x_test, d_test) = load_mnist(normalize=True, one_hot_label=True

print("データ読み込み完了")

network = MultiLayerNet(input_size=784, hidden_size_list=[40, 30, 30, 20],
                        output_size=10, activation='relu', weight_init_std=0.1)

iters_num = 2000
train_size = x_train.shape[0]
batch_size = 100
learning_rate = 0.1

train_loss_list = []
accuracies_train = []
accuracies_test = []

plot_interval=100

for i in range(iters_num):
    batch_mask = np.random.choice(train_size, batch_size)
    x_batch = x_train[batch_mask]
    d_batch = d_train[batch_mask]

    # 勾配
    grad = network.gradient(x_batch, d_batch)

    for key in network.params.keys():
```

```
        network.params[key] -= learning_rate * grad[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_trai
        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()
```

```
データ読み込み完了
Generation: 100. 正答率(トレーニング) = 0.12
                : 100. 正答率(テスト) = 0.207
Generation: 200. 正答率(トレーニング) = 0.19
                : 200. 正答率(テスト) = 0.2513
Generation: 300. 正答率(トレーニング) = 0.67
                : 300. 正答率(テスト) = 0.56
Generation: 400. 正答率(トレーニング) = 0.71
                : 400. 正答率(テスト) = 0.7388
Generation: 500. 正答率(トレーニング) = 0.9
                : 500. 正答率(テスト) = 0.8384
Generation: 600. 正答率(トレーニング) = 0.88
                : 600. 正答率(テスト) = 0.8599
Generation: 700. 正答率(トレーニング) = 0.91
                : 700. 正答率(テスト) = 0.8895
Generation: 800. 正答率(トレーニング) = 0.94
                : 800. 正答率(テスト) = 0.9067
Generation: 900. 正答率(トレーニング) = 0.95
                : 900. 正答率(テスト) = 0.918
Generation: 1000. 正答率(トレーニング) = 0.97
                : 1000. 正答率(テスト) = 0.9216
Generation: 1100. 正答率(トレーニング) = 0.9
                : 1100. 正答率(テスト) = 0.924
Generation: 1200. 正答率(トレーニング) = 0.96
                : 1200. 正答率(テスト) = 0.9336
Generation: 1300. 正答率(トレーニング) = 0.97
                : 1300. 正答率(テスト) = 0.9386
Generation: 1400. 正答率(トレーニング) = 0.93
                : 1400. 正答率(テスト) = 0.9357
Generation: 1500. 正答率(トレーニング) = 0.97
                : 1500. 正答率(テスト) = 0.937
Generation: 1600. 正答率(トレーニング) = 0.98
                : 1600. 正答率(テスト) = 0.9404
Generation: 1700. 正答率(トレーニング) = 0.99
                : 1700. 正答率(テスト) = 0.9403
Generation: 1800. 正答率(トレーニング) = 0.98
                : 1800. 正答率(テスト) = 0.9397
Generation: 1900. 正答率(トレーニング) = 0.96
                : 1900. 正答率(テスト) = 0.9441
Generation: 2000. 正答率(トレーニング) = 0.98
                : 2000. 正答率(テスト) = 0.9475
```



In [14]:
```
network.params.keys()
```

Out[14]: dict_keys(['W1', 'b1', 'W2', 'b2', 'W3', 'b3', 'W4', 'b4', 'W5', 'b5'])

活性化関数の定義と、重みの初期化方法を変えると、学習にどのような変化が起こるのかを確認した。
活性化関数の違いは、中間層の数が少ないうちはそれほど大きな影響は見られなかったが、層が厚くなるにつれて、reluの方が正しく学習が行われた。
一方で、重みの初期化方法は少ない層でも影響があった。xavierとheの違いはなかなかつかめなかったが、ガウス分布（0.01）で散らした時は全体的に学習がうまくいかない傾向にあった。一方で分散の大きさを大きくすると学習が進む場合があり、これは、重みの初期値が小さすぎると学習ができなくなることを裏付けている。

Section 2 学習率最適化手法

In [ ]:

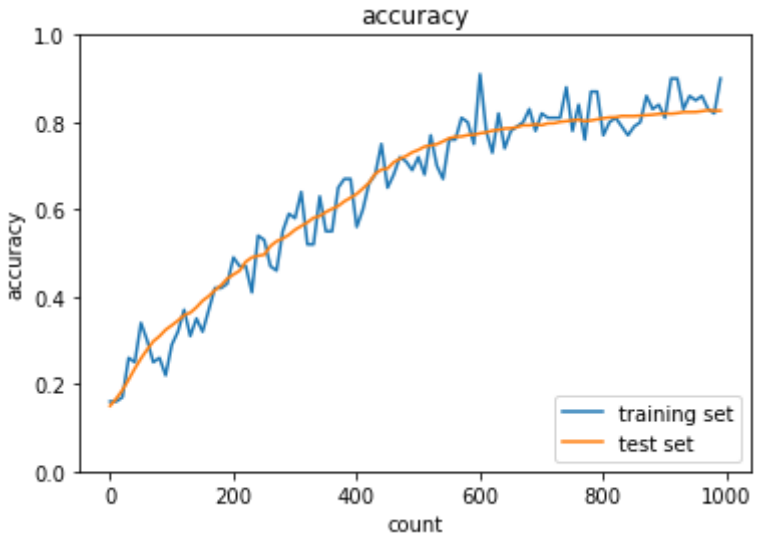
In [3]:
```
import sys, os
sys.path.append(os.pardir)  # 親ディレクトリのファイルをインポートするための設定
import numpy as np
from collections import OrderedDict
from common import layers
from data.mnist import load_mnist
import matplotlib.pyplot as plt
from lesson_2.multi_layer_net import MultiLayerNet
```

SGD

In [4]:
```
# データの読み込み
(x_train, d_train), (x_test, d_test) = load_mnist(normalize=True, one_hot_label=True

print("データ読み込み完了")

# batch_normalizationの設定 ================================
# use_batchnorm = True
use_batchnorm = False
# ====================================================


network = MultiLayerNet(input_size=784, hidden_size_list=[40, 20], output_size=10, ac
                        use_batchnorm=use_batchnorm)

iters_num = 1000
train_size = x_train.shape[0]
batch_size = 100
learning_rate = 0.005

train_loss_list = []
accuracies_train = []
accuracies_test = []

plot_interval=10

for i in range(iters_num):
    batch_mask = np.random.choice(train_size, batch_size)
    x_batch = x_train[batch_mask]
    d_batch = d_train[batch_mask]

    # 勾配
    grad = network.gradient(x_batch, d_batch)

    for key in ('W1', 'W2', 'W3', 'b1', 'b2', 'b3'):
        network.params[key] -= learning_rate * grad[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_tra
#         print('                : ' + str(i+1) + '. 正答率(テスト) = ' + str(accr_tes


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()
```

データ読み込み完了



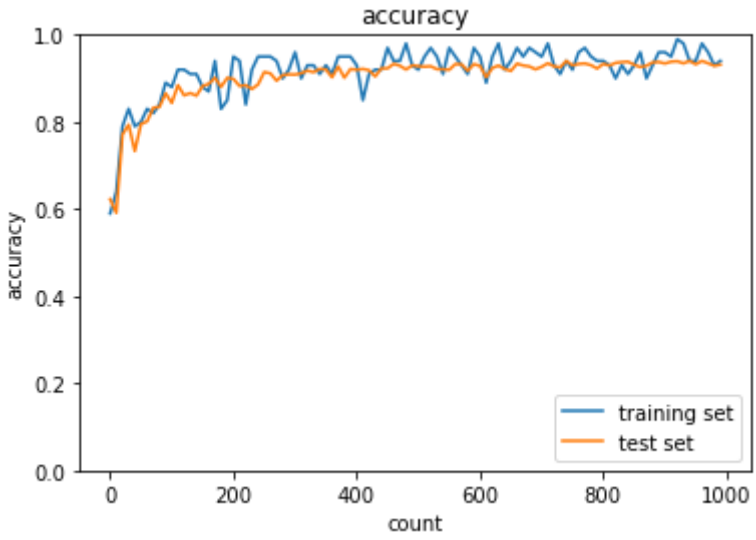
Momentum

In [16]:

```
# データの読み込み
(x_train, d_train), (x_test, d_test) = load_mnist(normalize=True, one_hot_label=True

print("データ読み込み完了")

# batch_normalizationの設定 ================================
# use_batchnorm = True
use_batchnorm = False
# ====================================================

network = MultiLayerNet(input_size=784, hidden_size_list=[40, 20], output_size=10, ac
                        use_batchnorm=use_batchnorm)

iters_num = 1000
train_size = x_train.shape[0]
batch_size = 100
learning_rate = 0.2
# 慣性
momentum = 0.9

train_loss_list = []
accuracies_train = []
accuracies_test = []

plot_interval=10

for i in range(iters_num):
    batch_mask = np.random.choice(train_size, batch_size)
```

```
    x_batch = x_train[batch_mask]
    d_batch = d_train[batch_mask]

    # 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]

        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_tra
#         print('                : ' + str(i+1) + '. 正答率(テスト) = ' + str(accr_tes


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()
```

```
データ読み込み完了
```



AdaGrad

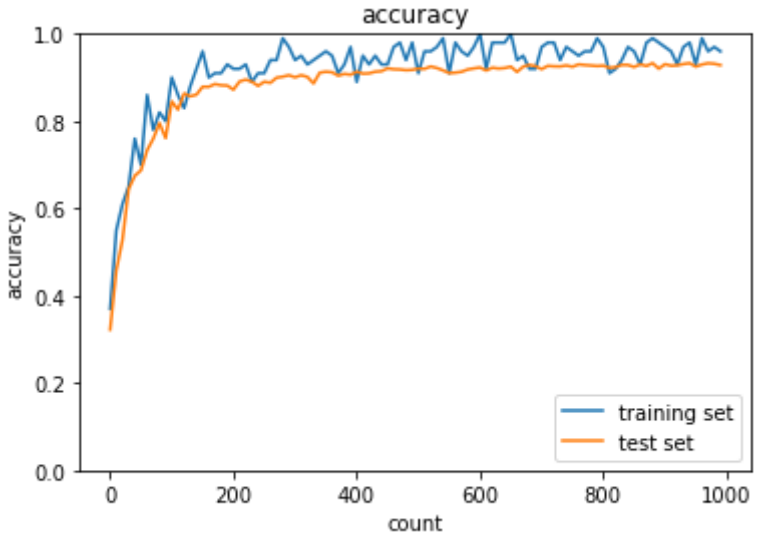
In [14]:

```
# データの読み込み
(x_train, d_train), (x_test, d_test) = load_mnist(normalize=True, one_hot_label=True

print("データ読み込み完了")

# batch_normalizationの設定 ================================
```

```
# use_batchnorm = True
use_batchnorm = False
# ====================================================

network = MultiLayerNet(input_size=784, hidden_size_list=[40, 20], output_size=10, ac
                        use_batchnorm=use_batchnorm)

iters_num = 1000
# iters_num = 500 # 処理を短縮

train_size = x_train.shape[0]
batch_size = 100
learning_rate = 0.2

# AdaGradでは不必要
# ============================

theta = 1e-4

# ============================

train_loss_list = []
accuracies_train = []
accuracies_test = []

plot_interval=10

for i in range(iters_num):
    batch_mask = np.random.choice(train_size, batch_size)
    x_batch = x_train[batch_mask]
    d_batch = d_train[batch_mask]

    # 勾配
    grad = network.gradient(x_batch, d_batch)
    if i == 0:
        h = {}
    for key in ('W1', 'W2', 'W3', 'b1', 'b2', 'b3'):

        # 変更しよう
        # ========================================
        if i == 0:
            h[key] = np.ones_like(network.params[key])*theta
        h[key] = h[key] + grad[key]**2
        network.params[key] -= learning_rate * grad[key]/(theta+h[key]**0.5)

        # ========================================

        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_tra
#         print('                : ' + str(i+1) + '. 正答率(テスト) = ' + str(accr_tes


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()
```

データ読み込み完了



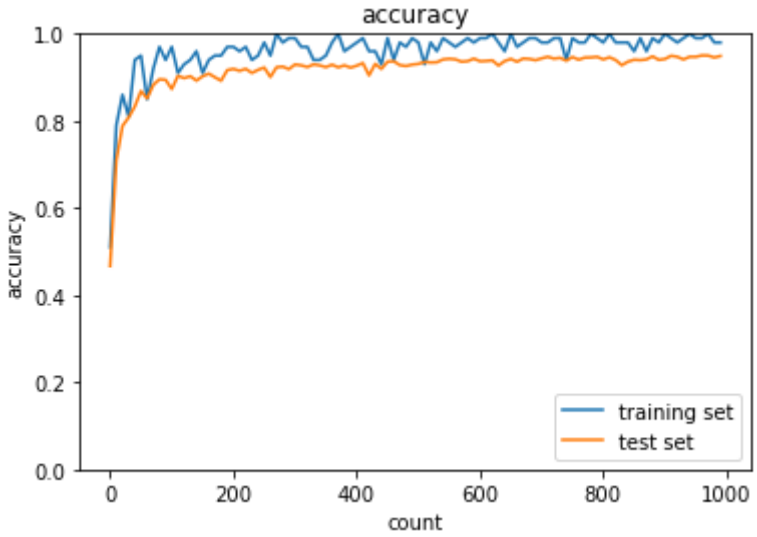
RMSprop

In [7]:

```
# データの読み込み
(x_train, d_train), (x_test, d_test) = load_mnist(normalize=True, one_hot_label=True

print("データ読み込み完了")

# batch_normalizationの設定 ================================
# use_batchnorm = True
use_batchnorm = False
# ====================================================

network = MultiLayerNet(input_size=784, hidden_size_list=[40, 20], output_size=10, ac
                        use_batchnorm=use_batchnorm)

iters_num = 1000
train_size = x_train.shape[0]
batch_size = 100
learning_rate = 0.005
decay_rate = 0.99

train_loss_list = []
accuracies_train = []
accuracies_test = []

plot_interval=10

for i in range(iters_num):
    batch_mask = np.random.choice(train_size, batch_size)
    x_batch = x_train[batch_mask]
    d_batch = d_train[batch_mask]

    # 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)

        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_tra
#         print('                : ' + str(i+1) + '. 正答率(テスト) = ' + str(accr_tes


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()
```

```
データ読み込み完了
```



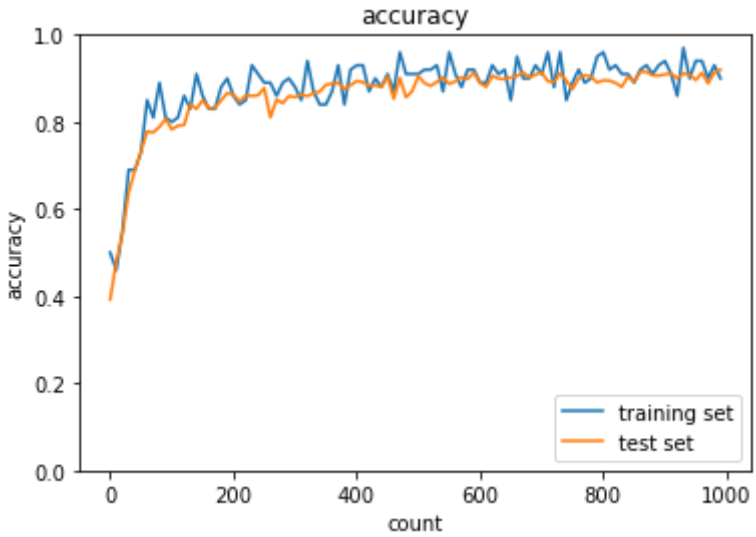
Adam

In [19]:

```
# データの読み込み
(x_train, d_train), (x_test, d_test) = load_mnist(normalize=True, one_hot_label=True

print("データ読み込み完了")

# batch_normalizationの設定 ================================
# use_batchnorm = True
use_batchnorm = False
# ====================================================

network = MultiLayerNet(input_size=784, hidden_size_list=[40, 20], output_size=10, ac
                        use_batchnorm=use_batchnorm)

iters_num = 1000
train_size = x_train.shape[0]
```

```
batch_size = 100
learning_rate = 0.05
beta1 = 0.9
beta2 = 0.999

train_loss_list = []
accuracies_train = []
accuracies_test = []

plot_interval=10

for i in range(iters_num):
    batch_mask = np.random.choice(train_size, batch_size)
    x_batch = x_train[batch_mask]
    d_batch = d_train[batch_mask]

    # 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 - beta
    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)


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

#         print('Generation: ' + str(i+1) + '. 正答率(トレーニング) = ' + str(accr_tra
#         print('                : ' + str(i+1) + '. 正答率(テスト) = ' + str(accr_tes


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()
```

データ読み込み完了



様々な最適化手法の比較を行った。残念ながら、今回の試行だけで最適化手法のとくちょうについて授業通りの確認をできたとは言いずらい部分がある。
少なくともいえるのは、学習率一定の場合より、どの手法も短い時間で収束に向かう傾向が見られた。また、AdaGrad系は、収束に近づくにつれてやや落ち着いてくるように見えなくもない。
いずれにせよ、初期の重みなどもランダムで振ってしまっているので、しっかり比較する場合は面倒だが、初期値もそろえる必要があると思った。

Section 3 過学習

In [ ]:

```
```

In [21]:

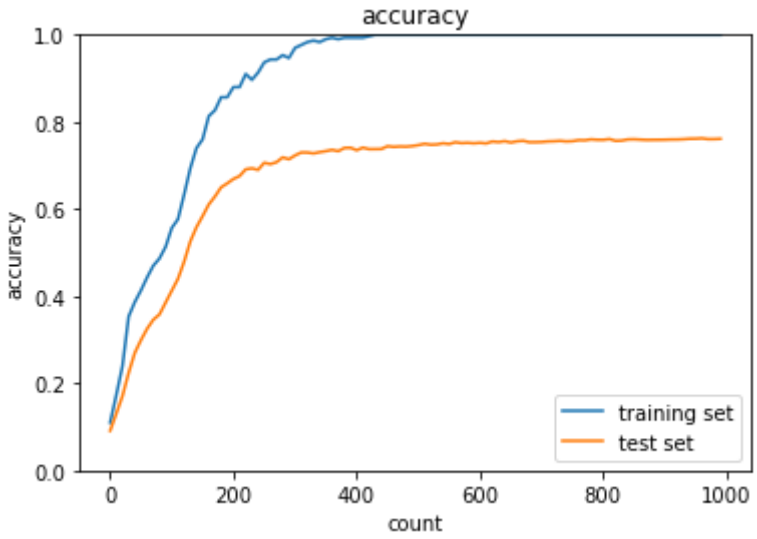
```
import numpy as np
from collections import OrderedDict
from common import layers
from data.mnist import load_mnist
import matplotlib.pyplot as plt
from lesson_2.multi_layer_net import MultiLayerNet
from common import optimizer
```

In [9]:

```
(x_train, d_train), (x_test, d_test) = load_mnist(normalize=True)

print("データ読み込み完了")

# 過学習を再現するために、学習データを削減
x_train = x_train[:300]
d_train = d_train[:300]

network = MultiLayerNet(input_size=784, hidden_size_list=[100, 100, 100, 100, 100, 10
optimizer = optimizer.SGD(learning_rate=0.01)

iters_num = 1000
train_size = x_train.shape[0]
batch_size = 100

train_loss_list = []
accuracies_train = []
accuracies_test = []

plot_interval=10


for i in range(iters_num):
    batch_mask = np.random.choice(train_size, batch_size)
    x_batch = x_train[batch_mask]
    d_batch = d_train[batch_mask]

    grad = network.gradient(x_batch, d_batch)
    optimizer.update(network.params, grad)

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

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

#         print('Generation: ' + str(i+1) + '. 正答率(トレーニング) = ' + str(accr_tra
#         print('                : ' + str(i+1) + '. 正答率(テスト) = ' + str(accr_tes

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()
```

データ読み込み完了



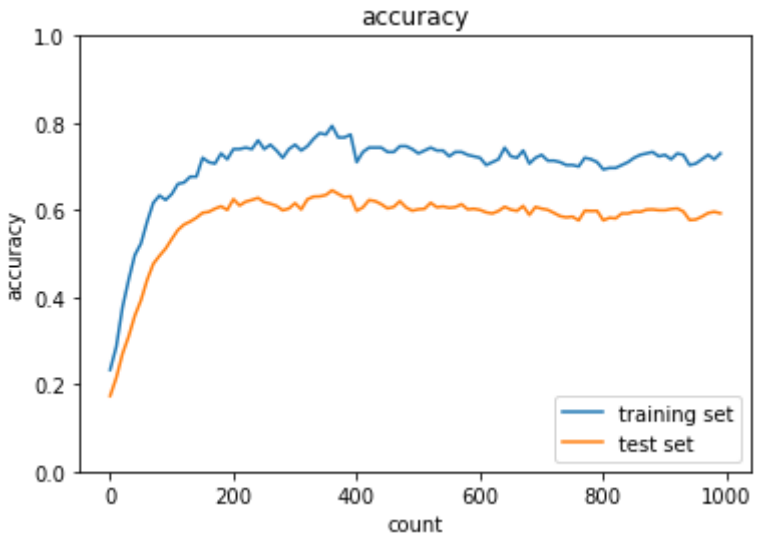
L2

In [29]:

```
(x_train, d_train), (x_test, d_test) = load_mnist(normalize=True)

print("データ読み込み完了")

# 過学習を再現するために、学習データを削減
x_train = x_train[:300]
d_train = d_train[:300]


network = MultiLayerNet(input_size=784, hidden_size_list=[100, 100, 100, 100, 100, 10


iters_num = 1000
train_size = x_train.shape[0]
batch_size = 100
learning_rate=0.01

train_loss_list = []
accuracies_train = []
accuracies_test = []

plot_interval=10
hidden_layer_num = network.hidden_layer_num

# 正則化強度設定 ======================================
weight_decay_lambda = 0.15
# =================================================

for i in range(iters_num):
    batch_mask = np.random.choice(train_size, batch_size)
    x_batch = x_train[batch_mask]
    d_batch = d_train[batch_mask]

    grad = network.gradient(x_batch, d_batch)
    weight_decay = 0

    for idx in range(1, hidden_layer_num+1):
        grad['W' + str(idx)] = network.layers['Affine' + str(idx)].dW + weight_decay_
        grad['b' + str(idx)] = network.layers['Affine' + str(idx)].db
```

```
        network.params['W' + str(idx)] -= learning_rate * grad['W' + str(idx)]
        network.params['b' + str(idx)] -= learning_rate * grad['b' + str(idx)]
        weight_decay += 0.5 * weight_decay_lambda * np.sqrt(np.sum(network.params['W'

    loss = network.loss(x_batch, d_batch) + weight_decay
    train_loss_list.append(loss)

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

#         print('Generation: ' + str(i+1) + '. 正答率(トレーニング) = ' + str(accr_tra
#         print('                : ' + str(i+1) + '. 正答率(テスト) = ' + str(accr_tes


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()
```

```
データ読み込み完了
```



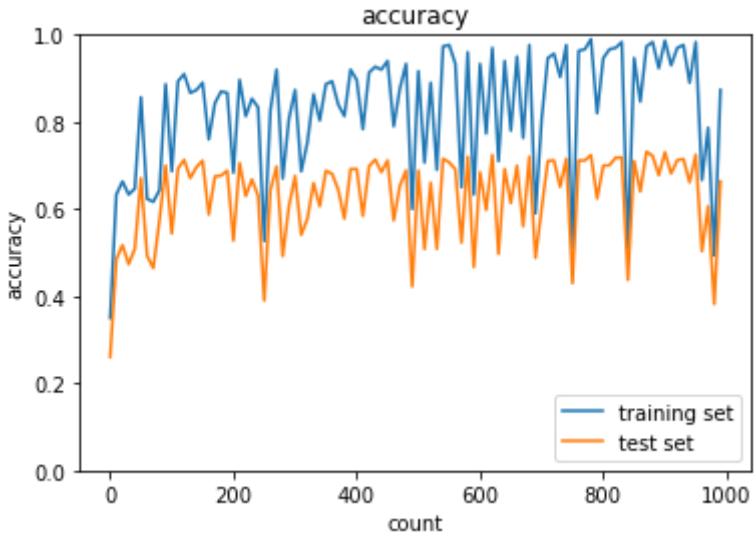
L1

In [18]:

```
(x_train, d_train), (x_test, d_test) = load_mnist(normalize=True)

print("データ読み込み完了")

# 過学習を再現するために、学習データを削減
x_train = x_train[:300]
d_train = d_train[:300]

network = MultiLayerNet(input_size=784, hidden_size_list=[100, 100, 100, 100, 100, 10

iters_num = 1000
train_size = x_train.shape[0]
batch_size = 100
learning_rate=0.1
```

```
train_loss_list = []
accuracies_train = []
accuracies_test = []

plot_interval=10
hidden_layer_num = network.hidden_layer_num

# 正則化強度設定 ======================================
weight_decay_lambda = 0.006
# =====================================================

for i in range(iters_num):
    batch_mask = np.random.choice(train_size, batch_size)
    x_batch = x_train[batch_mask]
    d_batch = d_train[batch_mask]

    grad = network.gradient(x_batch, d_batch)
    weight_decay = 0

    for idx in range(1, hidden_layer_num+1):
        grad['W' + str(idx)] = network.layers['Affine' + str(idx)].dW + weight_decay_
        grad['b' + str(idx)] = network.layers['Affine' + str(idx)].db
        network.params['W' + str(idx)] -= learning_rate * grad['W' + str(idx)]
        network.params['b' + str(idx)] -= learning_rate * grad['b' + str(idx)]
        weight_decay += weight_decay_lambda * np.sum(np.abs(network.params['W' + str(

    loss = network.loss(x_batch, d_batch) + weight_decay
    train_loss_list.append(loss)

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

#         print('Generation: ' + str(i+1) + '. 正答率(トレーニング) = ' + str(accr_tra
#         print('                : ' + str(i+1) + '. 正答率(テスト) = ' + str(accr_tes

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()
```

データ読み込み完了



Dropout

In [19]:

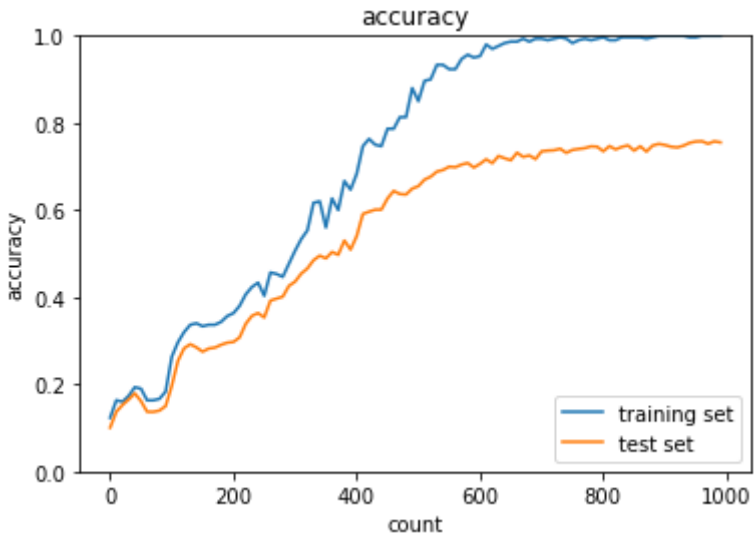
```
class Dropout:
    def __init__(self, dropout_ratio=0.5):
        self.dropout_ratio = dropout_ratio
        self.mask = None

    def forward(self, x, train_flg=True):
        if train_flg:
            self.mask = np.random.rand(*x.shape) > self.dropout_ratio
            return x * self.mask
        else:
            return x * (1.0 - self.dropout_ratio)

    def backward(self, dout):
        return dout * self.mask
```

In [28]:

```
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")

# 過学習を再現するために、学習データを削減
x_train = x_train[:300]
d_train = d_train[:300]

# ドロップアウト設定 ======================================
use_dropout = True
dropout_ratio = 0.3
# ====================================================

network = MultiLayerNet(input_size=784, hidden_size_list=[100, 100, 100, 100, 100, 10
                        weight_decay_lambda=weight_decay_lambda, use_dropout = use_dr
# optimizer = optimizer.SGD(learning_rate=0.01)
optimizer = optimizer.Momentum(learning_rate=0.01, momentum=0.9)
# optimizer = optimizer.AdaGrad(learning_rate=0.01)
# optimizer = optimizer.Adam()

iters_num = 1000
train_size = x_train.shape[0]
batch_size = 100

train_loss_list = []
accuracies_train = []
accuracies_test = []
```

```
plot_interval=10


for i in range(iters_num):
    batch_mask = np.random.choice(train_size, batch_size)
    x_batch = x_train[batch_mask]
    d_batch = d_train[batch_mask]

    grad = network.gradient(x_batch, d_batch)
    optimizer.update(network.params, grad)

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

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

#         print('Generation: ' + str(i+1) + '. 正答率(トレーニング) = ' + str(accr_tra
#         print('                : ' + str(i+1) + '. 正答率(テスト) = ' + str(accr_tes

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()
```

```
データ読み込み完了
```



何を以て過学習を防げているか評価することは難しいが、L2正則化は比較的うまくいったように思える。学習データを最高に上手に学習できた場合で、テストデータがどの程度正答させられるかは事前にわからないので、パラメトリックスタディの中で、モデルを表現するのに適切な重さを決定しなくてはならない。

Section 4 畳み込みニューラルネットワークの概念

In [ ]:

```
```

In [20]:

```
import pickle
import numpy as np
from collections import OrderedDict
from common import layers
from common import optimizer
from data.mnist import load_mnist
import matplotlib.pyplot as plt
import time
```

In [11]:

```
# 画像データを２次元配列に変換
'''
input_data: 入力値
filter_h: フィルターの高さ
filter_w: フィルターの横幅
stride: ストライド
pad: パディング
'''
def im2col(input_data, filter_h, filter_w, stride=1, pad=0):
    # N: number, C: channel, H: height, W: width
    N, C, H, W = input_data.shape
    out_h = (H + 2 * pad - filter_h)//stride + 1
    out_w = (W + 2 * pad - filter_w)//stride + 1

    img = np.pad(input_data, [(0,0), (0,0), (pad, pad), (pad, pad)], 'constant')
    col = np.zeros((N, C, filter_h, filter_w, out_h, out_w))

    for y in range(filter_h):
        y_max = y + stride * out_h
        for x in range(filter_w):
            x_max = x + stride * out_w
            col[:, :, y, x, :, :] = img[:, :, y:y_max:stride, x:x_max:stride]
    print(col)
    col = col.transpose(0, 4, 5, 1, 2, 3) # (N, C, filter_h, filter_w, out_h, out_w) -

    col = col.reshape(N * out_h * out_w, -1)
    return col
```

In [13]:

```
# 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):
    # N: number, C: channel, H: height, W: width
    N, C, H, W = input_shape
    # 切り捨て除算
    out_h = (H + 2 * pad - filter_h)//stride + 1
    out_w = (W + 2 * pad - filter_w)//stride + 1
    col = col.reshape(N, out_h, out_w, C, filter_h, filter_w).transpose(0, 3, 4, 5, 1

    img = np.zeros((N, C, H + 2 * pad + stride - 1, W + 2 * pad + stride - 1))
    for y in range(filter_h):
        y_max = y + stride * out_h
        for x in range(filter_w):
            x_max = x + stride * out_w
            img[:, :, y:y_max:stride, x:x_max:stride] += col[:, :, y, x, :, :]

    return img[:, :, pad:H + pad, pad:W + pad]
```

In [15]:

```
# 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('==============================')
col2im(col ,input_data.shape, filter_h=filter_h, filter_w=filter_w, stride=stride, pa
```

```
========== input_data ==========
 [[[[13. 76. 63. 39.]
   [71. 46.  3. 13.]
   [73. 62. 40. 30.]
   [98. 27.  4. 40.]]]


 [[[46. 95. 17. 22.]
   [41.  9.  3. 87.]
   [19. 53. 33.  0.]
   [69. 99. 13. 19.]]]]
==============================
[[[[[[13. 76.]
     [71. 46.]]

    [[76. 63.]
     [46.  3.]]

    [[63. 39.]
     [ 3. 13.]]]


   [[[71. 46.]
     [73. 62.]]

    [[46.  3.]
     [62. 40.]]

    [[ 3. 13.]
     [40. 30.]]]


   [[[73. 62.]
     [98. 27.]]

    [[62. 40.]
     [27.  4.]]

    [[40. 30.]
     [ 4. 40.]]]]]



 [[[[[46. 95.]
     [41.  9.]]

    [[95. 17.]
     [ 9.  3.]]

    [[17. 22.]
     [ 3. 87.]]]


   [[[41.  9.]
     [19. 53.]]

    [[ 9.  3.]
     [53. 33.]]

    [[ 3. 87.]
     [33.  0.]]]
```

```
   [[[19. 53.]
     [69. 99.]]

    [[53. 33.]
     [99. 13.]]

    [[33.  0.]
     [13. 19.]]]]]]
============= col ==============
 [[13. 76. 63. 71. 46.  3. 73. 62. 40.]
 [76. 63. 39. 46.  3. 13. 62. 40. 30.]
 [71. 46.  3. 73. 62. 40. 98. 27.  4.]
 [46.  3. 13. 62. 40. 30. 27.  4. 40.]
 [46. 95. 17. 41.  9.  3. 19. 53. 33.]
 [95. 17. 22.  9.  3. 87. 53. 33.  0.]
 [41.  9.  3. 19. 53. 33. 69. 99. 13.]
 [ 9.  3. 87. 53. 33.  0. 99. 13. 19.]]
==============================
```

Out[15]:

```
array([[[[ 13., 152., 126.,  39.],
         [142., 184.,  12.,  26.],
         [146., 248., 160.,  60.],
         [ 98.,  54.,   8.,  40.]]],


       [[[ 46., 190.,  34.,  22.],
         [ 82.,  36.,  12., 174.],
         [ 38., 212., 132.,   0.],
         [ 69., 198.,  26.,  19.]]]])
```

In [16]:

```
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
        self.col_W = None

        # フィルター・バイアスパラメータの勾配
        self.dW = None
        self.db = None

    def forward(self, x):
        # FN: filter_number, C: channel, FH: filter_height, FW: filter_width
        FN, C, FH, FW = self.W.shape
        N, C, H, W = x.shape
        # 出力値のheight, width
        out_h = 1 + int((H + 2 * self.pad - FH) / self.stride)
        out_w = 1 + int((W + 2 * self.pad - FW) / self.stride)

        # 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
```

```
    def backward(self, dout):
        FN, C, FH, FW = self.W.shape
        dout = dout.transpose(0, 2, 3, 1).reshape(-1, FN)

        self.db = np.sum(dout, axis=0)
        self.dW = np.dot(self.col.T, dout)
        self.dW = self.dW.transpose(1, 0).reshape(FN, C, FH, FW)

        dcol = np.dot(dout, self.col_W.T)
        # dcolを画像データに変換
        dx = col2im(dcol, self.x.shape, FH, FW, self.stride, self.pad)

        return dx

class Pooling:
    def __init__(self, pool_h, pool_w, stride=1, pad=0):
        self.pool_h = pool_h
        self.pool_w = pool_w
        self.stride = stride
        self.pad = pad

        self.x = None
        self.arg_max = None

    def forward(self, x):
        N, C, H, W = x.shape
        out_h = int(1 + (H - self.pool_h) / self.stride)
        out_w = int(1 + (W - self.pool_w) / self.stride)

        # xを行列に変換
        col = im2col(x, self.pool_h, self.pool_w, self.stride, self.pad)
        # プーリングのサイズに合わせてリサイズ
        col = col.reshape(-1, self.pool_h*self.pool_w)

        # 行ごとに最大値を求める
        arg_max = np.argmax(col, axis=1)
        out = np.max(col, axis=1)
        # 整形
        out = out.reshape(N, out_h, out_w, C).transpose(0, 3, 1, 2)

        self.x = x
        self.arg_max = arg_max

        return out

    def backward(self, dout):
        dout = dout.transpose(0, 2, 3, 1)

        pool_size = self.pool_h * self.pool_w
        dmax = np.zeros((dout.size, pool_size))
        dmax[np.arange(self.arg_max.size), self.arg_max.flatten()] = dout.flatten()
        dmax = dmax.reshape(dout.shape + (pool_size,))

        dcol = dmax.reshape(dmax.shape[0] * dmax.shape[1] * dmax.shape[2], -1)
        dx = col2im(dcol, self.x.shape, self.pool_h, self.pool_w, self.stride, self.p

        return dx

class SimpleConvNet:
    # conv - relu - pool - affine - relu - affine - softmax
    def __init__(self, input_dim=(1, 28, 28), conv_param={'filter_num':30, 'filter_si
                 hidden_size=100, output_size=10, weight_init_std=0.01):
        filter_num = conv_param['filter_num']
        filter_size = conv_param['filter_size']
        filter_pad = conv_param['pad']
```

```
        filter_stride = conv_param['stride']
        input_size = input_dim[1]
        conv_output_size = (input_size - filter_size + 2 * filter_pad) / filter_strid
        pool_output_size = int(filter_num * (conv_output_size / 2) * (conv_output_siz

        # 重みの初期化
        self.params = {}
        self.params['W1'] = weight_init_std * np.random.randn(filter_num, input_dim[0
        self.params['b1'] = np.zeros(filter_num)
        self.params['W2'] = weight_init_std * np.random.randn(pool_output_size, hidde
        self.params['b2'] = np.zeros(hidden_size)
        self.params['W3'] = weight_init_std * np.random.randn(hidden_size, output_siz
        self.params['b3'] = np.zeros(output_size)

        # レイヤの生成
        self.layers = OrderedDict()
        self.layers['Conv1'] = layers.Convolution(self.params['W1'], self.params['b1'
        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()

    def predict(self, x):
        for key in self.layers.keys():
            x = self.layers[key].forward(x)
        return x

    def loss(self, x, d):
        y = self.predict(x)
        return self.last_layer.forward(y, d)

    def accuracy(self, x, d, batch_size=100):
        if d.ndim != 1 : d = np.argmax(d, axis=1)

        acc = 0.0

        for i in range(int(x.shape[0] / batch_size)):
            tx = x[i*batch_size:(i+1)*batch_size]
            td = d[i*batch_size:(i+1)*batch_size]
            y = self.predict(tx)
            y = np.argmax(y, axis=1)
            acc += np.sum(y == td)

        return acc / x.shape[0]

    def gradient(self, x, d):
        # forward
        self.loss(x, d)

        # backward
        dout = 1
        dout = self.last_layer.backward(dout)
        layers = list(self.layers.values())

        layers.reverse()
        for layer in layers:
            dout = layer.backward(dout)

        # 設定
        grad = {}
        grad['W1'], grad['b1'] = self.layers['Conv1'].dW, self.layers['Conv1'].db
        grad['W2'], grad['b2'] = self.layers['Affine1'].dW, self.layers['Affine1'].db
```

```
        grad['W3'], grad['b3'] = self.layers['Affine2'].dW, self.layers['Affine2'].db

        return grad
```

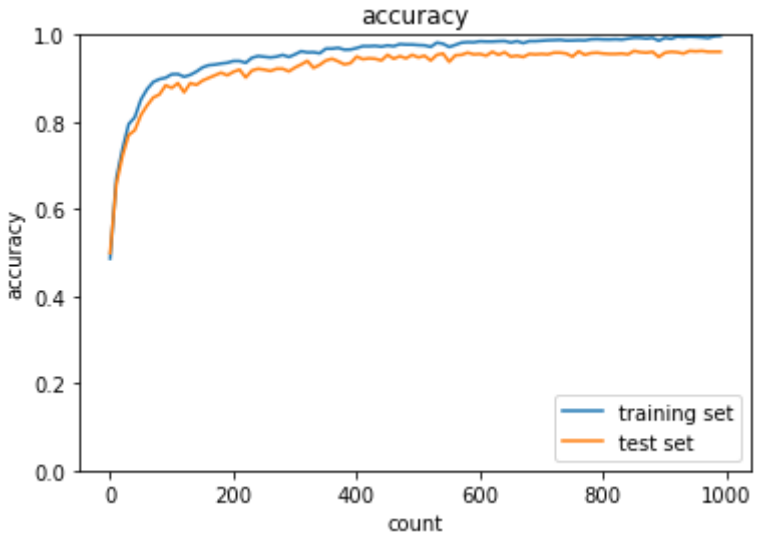
In [17]:

```
from common import optimizer

# データの読み込み
(x_train, d_train), (x_test, d_test) = load_mnist(flatten=False)

print("データ読み込み完了")

# 処理に時間のかかる場合はデータを削減
x_train, d_train = x_train[:5000], d_train[:5000]
x_test, d_test = x_test[:1000], d_test[:1000]


network = SimpleConvNet(input_dim=(1,28,28), conv_param = {'filter_num': 30, 'filter_
                        hidden_size=100, output_size=10, weight_init_std=0.01)

optimizer = optimizer.Adam()

iters_num = 1000
train_size = x_train.shape[0]
batch_size = 100

train_loss_list = []
accuracies_train = []
accuracies_test = []

plot_interval=10



for i in range(iters_num):
    batch_mask = np.random.choice(train_size, batch_size)
    x_batch = x_train[batch_mask]
    d_batch = d_train[batch_mask]

    grad = network.gradient(x_batch, d_batch)
    optimizer.update(network.params, grad)

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

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

        print('Generation: ' + str(i+1) + '. 正答率(トレーニング) = ' + str(accr_trai
        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()
```

```
データ読み込み完了
Generation: 10. 正答率(トレーニング) = 0.4864
                : 10. 正答率(テスト) = 0.499
Generation: 20. 正答率(トレーニング) = 0.67
                : 20. 正答率(テスト) = 0.657
Generation: 30. 正答率(トレーニング) = 0.738
                : 30. 正答率(テスト) = 0.722
Generation: 40. 正答率(トレーニング) = 0.7948
                : 40. 正答率(テスト) = 0.77
Generation: 50. 正答率(トレーニング) = 0.8108
                : 50. 正答率(テスト) = 0.781
Generation: 60. 正答率(トレーニング) = 0.8516
                : 60. 正答率(テスト) = 0.816
Generation: 70. 正答率(トレーニング) = 0.8754
                : 70. 正答率(テスト) = 0.838
Generation: 80. 正答率(トレーニング) = 0.8918
                : 80. 正答率(テスト) = 0.856
Generation: 90. 正答率(トレーニング) = 0.8982
                : 90. 正答率(テスト) = 0.863
Generation: 100. 正答率(トレーニング) = 0.9014
                : 100. 正答率(テスト) = 0.884
Generation: 110. 正答率(トレーニング) = 0.9094
                : 110. 正答率(テスト) = 0.878
Generation: 120. 正答率(トレーニング) = 0.9098
                : 120. 正答率(テスト) = 0.889
Generation: 130. 正答率(トレーニング) = 0.9036
                : 130. 正答率(テスト) = 0.868
Generation: 140. 正答率(トレーニング) = 0.908
                : 140. 正答率(テスト) = 0.889
Generation: 150. 正答率(トレーニング) = 0.9156
                : 150. 正答率(テスト) = 0.885
Generation: 160. 正答率(トレーニング) = 0.9252
                : 160. 正答率(テスト) = 0.895
Generation: 170. 正答率(トレーニング) = 0.9302
                : 170. 正答率(テスト) = 0.901
Generation: 180. 正答率(トレーニング) = 0.9316
                : 180. 正答率(テスト) = 0.907
Generation: 190. 正答率(トレーニング) = 0.934
                : 190. 正答率(テスト) = 0.913
Generation: 200. 正答率(トレーニング) = 0.9358
                : 200. 正答率(テスト) = 0.907
Generation: 210. 正答率(トレーニング) = 0.9398
                : 210. 正答率(テスト) = 0.915
Generation: 220. 正答率(トレーニング) = 0.9398
                : 220. 正答率(テスト) = 0.92
Generation: 230. 正答率(トレーニング) = 0.9356
                : 230. 正答率(テスト) = 0.902
Generation: 240. 正答率(トレーニング) = 0.9472
                : 240. 正答率(テスト) = 0.918
Generation: 250. 正答率(トレーニング) = 0.9512
                : 250. 正答率(テスト) = 0.922
Generation: 260. 正答率(トレーニング) = 0.9496
                : 260. 正答率(テスト) = 0.92
Generation: 270. 正答率(トレーニング) = 0.948
                : 270. 正答率(テスト) = 0.917
Generation: 280. 正答率(トレーニング) = 0.9498
                : 280. 正答率(テスト) = 0.922
Generation: 290. 正答率(トレーニング) = 0.9538
                : 290. 正答率(テスト) = 0.922
Generation: 300. 正答率(トレーニング) = 0.949
                : 300. 正答率(テスト) = 0.916
Generation: 310. 正答率(トレーニング) = 0.9552
                : 310. 正答率(テスト) = 0.925
Generation: 320. 正答率(トレーニング) = 0.9622
                : 320. 正答率(テスト) = 0.932
Generation: 330. 正答率(トレーニング) = 0.9596
                : 330. 正答率(テスト) = 0.94
Generation: 340. 正答率(トレーニング) = 0.96
                : 340. 正答率(テスト) = 0.924
Generation: 350. 正答率(トレーニング) = 0.9576
                : 350. 正答率(テスト) = 0.931
Generation: 360. 正答率(トレーニング) = 0.9682
                : 360. 正答率(テスト) = 0.941
Generation: 370. 正答率(トレーニング) = 0.9682
                : 370. 正答率(テスト) = 0.945
Generation: 380. 正答率(トレーニング) = 0.97
                : 380. 正答率(テスト) = 0.939
Generation: 390. 正答率(トレーニング) = 0.9658
                : 390. 正答率(テスト) = 0.932
Generation: 400. 正答率(トレーニング) = 0.9662
```

```
                : 400. 正答率(テスト) = 0.935
Generation: 410. 正答率(トレーニング) = 0.9696
                : 410. 正答率(テスト) = 0.95
Generation: 420. 正答率(トレーニング) = 0.974
                : 420. 正答率(テスト) = 0.944
Generation: 430. 正答率(トレーニング) = 0.9738
                : 430. 正答率(テスト) = 0.946
Generation: 440. 正答率(トレーニング) = 0.9746
                : 440. 正答率(テスト) = 0.945
Generation: 450. 正答率(トレーニング) = 0.9728
                : 450. 正答率(テスト) = 0.941
Generation: 460. 正答率(トレーニング) = 0.9756
                : 460. 正答率(テスト) = 0.954
Generation: 470. 正答率(トレーニング) = 0.9738
                : 470. 正答率(テスト) = 0.945
Generation: 480. 正答率(トレーニング) = 0.9786
                : 480. 正答率(テスト) = 0.951
Generation: 490. 正答率(トレーニング) = 0.9776
                : 490. 正答率(テスト) = 0.946
Generation: 500. 正答率(トレーニング) = 0.9776
                : 500. 正答率(テスト) = 0.954
Generation: 510. 正答率(トレーニング) = 0.9764
                : 510. 正答率(テスト) = 0.948
Generation: 520. 正答率(トレーニング) = 0.9758
                : 520. 正答率(テスト) = 0.952
Generation: 530. 正答率(トレーニング) = 0.9722
                : 530. 正答率(テスト) = 0.941
Generation: 540. 正答率(トレーニング) = 0.982
                : 540. 正答率(テスト) = 0.954
Generation: 550. 正答率(トレーニング) = 0.9794
                : 550. 正答率(テスト) = 0.957
Generation: 560. 正答率(トレーニング) = 0.972
                : 560. 正答率(テスト) = 0.938
Generation: 570. 正答率(トレーニング) = 0.9768
                : 570. 正答率(テスト) = 0.953
Generation: 580. 正答率(トレーニング) = 0.982
                : 580. 正答率(テスト) = 0.954
Generation: 590. 正答率(トレーニング) = 0.9832
                : 590. 正答率(テスト) = 0.959
Generation: 600. 正答率(トレーニング) = 0.9828
                : 600. 正答率(テスト) = 0.955
Generation: 610. 正答率(トレーニング) = 0.985
                : 610. 正答率(テスト) = 0.956
Generation: 620. 正答率(トレーニング) = 0.9844
                : 620. 正答率(テスト) = 0.952
Generation: 630. 正答率(トレーニング) = 0.9844
                : 630. 正答率(テスト) = 0.961
Generation: 640. 正答率(トレーニング) = 0.9852
                : 640. 正答率(テスト) = 0.954
Generation: 650. 正答率(トレーニング) = 0.9856
                : 650. 正答率(テスト) = 0.96
Generation: 660. 正答率(トレーニング) = 0.9822
                : 660. 正答率(テスト) = 0.95
Generation: 670. 正答率(トレーニング) = 0.9854
                : 670. 正答率(テスト) = 0.952
Generation: 680. 正答率(トレーニング) = 0.981
                : 680. 正答率(テスト) = 0.949
Generation: 690. 正答率(トレーニング) = 0.9854
                : 690. 正答率(テスト) = 0.956
Generation: 700. 正答率(トレーニング) = 0.9854
                : 700. 正答率(テスト) = 0.955
Generation: 710. 正答率(トレーニング) = 0.9864
                : 710. 正答率(テスト) = 0.956
Generation: 720. 正答率(トレーニング) = 0.9874
                : 720. 正答率(テスト) = 0.955
Generation: 730. 正答率(トレーニング) = 0.9878
                : 730. 正答率(テスト) = 0.959
Generation: 740. 正答率(トレーニング) = 0.9884
                : 740. 正答率(テスト) = 0.959
Generation: 750. 正答率(トレーニング) = 0.9868
                : 750. 正答率(テスト) = 0.957
Generation: 760. 正答率(トレーニング) = 0.987
                : 760. 正答率(テスト) = 0.95
Generation: 770. 正答率(トレーニング) = 0.9876
                : 770. 正答率(テスト) = 0.962
Generation: 780. 正答率(トレーニング) = 0.9872
                : 780. 正答率(テスト) = 0.954
Generation: 790. 正答率(トレーニング) = 0.9896
                : 790. 正答率(テスト) = 0.958
Generation: 800. 正答率(トレーニング) = 0.9902
```

```
                : 800. 正答率(テスト) = 0.959
Generation: 810. 正答率(トレーニング) = 0.9894
                : 810. 正答率(テスト) = 0.957
Generation: 820. 正答率(トレーニング) = 0.9892
                : 820. 正答率(テスト) = 0.956
Generation: 830. 正答率(トレーニング) = 0.9898
                : 830. 正答率(テスト) = 0.956
Generation: 840. 正答率(トレーニング) = 0.989
                : 840. 正答率(テスト) = 0.957
Generation: 850. 正答率(トレーニング) = 0.9912
                : 850. 正答率(テスト) = 0.955
Generation: 860. 正答率(トレーニング) = 0.9932
                : 860. 正答率(テスト) = 0.963
Generation: 870. 正答率(トレーニング) = 0.993
                : 870. 正答率(テスト) = 0.96
Generation: 880. 正答率(トレーニング) = 0.9914
                : 880. 正答率(テスト) = 0.959
Generation: 890. 正答率(トレーニング) = 0.993
                : 890. 正答率(テスト) = 0.961
Generation: 900. 正答率(トレーニング) = 0.9864
                : 900. 正答率(テスト) = 0.949
Generation: 910. 正答率(トレーニング) = 0.9932
                : 910. 正答率(テスト) = 0.959
Generation: 920. 正答率(トレーニング) = 0.9908
                : 920. 正答率(テスト) = 0.961
Generation: 930. 正答率(トレーニング) = 0.996
                : 930. 正答率(テスト) = 0.96
Generation: 940. 正答率(トレーニング) = 0.9948
                : 940. 正答率(テスト) = 0.957
Generation: 950. 正答率(トレーニング) = 0.9958
                : 950. 正答率(テスト) = 0.963
Generation: 960. 正答率(トレーニング) = 0.9952
                : 960. 正答率(テスト) = 0.962
Generation: 970. 正答率(トレーニング) = 0.9942
                : 970. 正答率(テスト) = 0.963
Generation: 980. 正答率(トレーニング) = 0.9926
                : 980. 正答率(テスト) = 0.961
Generation: 990. 正答率(トレーニング) = 0.9964
                : 990. 正答率(テスト) = 0.961
Generation: 1000. 正答率(トレーニング) = 0.9976
                : 1000. 正答率(テスト) = 0.961
```



In [40]:

```
class DoubleConvNet:
    # conv - relu - pool - conv - relu - pool - affine - relu - affine - softmax
    def __init__(self, input_dim=(1, 28, 28),
                 conv_param_1={'filter_num':10, 'filter_size':7, 'pad':1, 'stride':1},
                 conv_param_2={'filter_num':20, 'filter_size':3, 'pad':1, 'stride':1},
                 hidden_size=100, output_size=10, weight_init_std=0.01):
        conv_output_size_1 = (input_dim[1] - conv_param_1['filter_size'] + 2 * conv_p
#         conv_output_size_2 = (conv_output_size_1 - conv_param_2['filter_size'] + 2 *
        conv_output_size_2 = (conv_output_size_1 / 2 - conv_param_2['filter_size'] +
        pool_output_size = int(conv_param_2['filter_num'] * (conv_output_size_2 / 2)
        # 重みの初期化
        self.params = {}
```

```
        self.params['W1'] = weight_init_std * np.random.randn(conv_param_1['filter_nu
        self.params['b1'] = np.zeros(conv_param_1['filter_num'])
        self.params['W2'] = weight_init_std * np.random.randn(conv_param_2['filter_nu
        self.params['b2'] = np.zeros(conv_param_2['filter_num'])
        self.params['W3'] = weight_init_std * np.random.randn(pool_output_size, hidde
        self.params['b3'] = np.zeros(hidden_size)
        self.params['W4'] = weight_init_std * np.random.randn(hidden_size, output_siz
        self.params['b4'] = np.zeros(output_size)
        # レイヤの生成
        self.layers = OrderedDict()
        self.layers['Conv1'] = layers.Convolution(self.params['W1'], self.params['b1'
        self.layers['Relu1'] = layers.Relu()

        self.layers['Pool1'] = layers.Pooling(pool_h=2, pool_w=2, stride=2)

        self.layers['Conv2'] = layers.Convolution(self.params['W2'], self.params['b2'
        self.layers['Relu2'] = layers.Relu()
        self.layers['Pool2'] = layers.Pooling(pool_h=2, pool_w=2, stride=2)
        self.layers['Affine1'] = layers.Affine(self.params['W3'], self.params['b3'])
        self.layers['Relu3'] = layers.Relu()
        self.layers['Affine2'] = layers.Affine(self.params['W4'], self.params['b4'])
        self.last_layer = layers.SoftmaxWithLoss()

    def predict(self, x):
        for key in self.layers.keys():
            x = self.layers[key].forward(x)
#             print(key)
        return x

    def loss(self, x, d):
        y = self.predict(x)
        return self.last_layer.forward(y, d)

    def accuracy(self, x, d, batch_size=100):
        if d.ndim != 1 : d = np.argmax(d, axis=1)

        acc = 0.0

        for i in range(int(x.shape[0] / batch_size)):
            tx = x[i*batch_size:(i+1)*batch_size]
            td = d[i*batch_size:(i+1)*batch_size]
            y = self.predict(tx)
            y = np.argmax(y, axis=1)
            acc += np.sum(y == td)

        return acc / x.shape[0]

    def gradient(self, x, d):
        # forward
        self.loss(x, d)

        # backward
        dout = 1
        dout = self.last_layer.backward(dout)
        layers = list(self.layers.values())

        layers.reverse()
        for layer in layers:
            dout = layer.backward(dout)

        # 設定
        grad = {}
        grad['W1'], grad['b1'] = self.layers['Conv1'].dW, self.layers['Conv1'].db
        grad['W2'], grad['b2'] = self.layers['Conv2'].dW, self.layers['Conv2'].db
        grad['W3'], grad['b3'] = self.layers['Affine1'].dW, self.layers['Affine1'].db
```

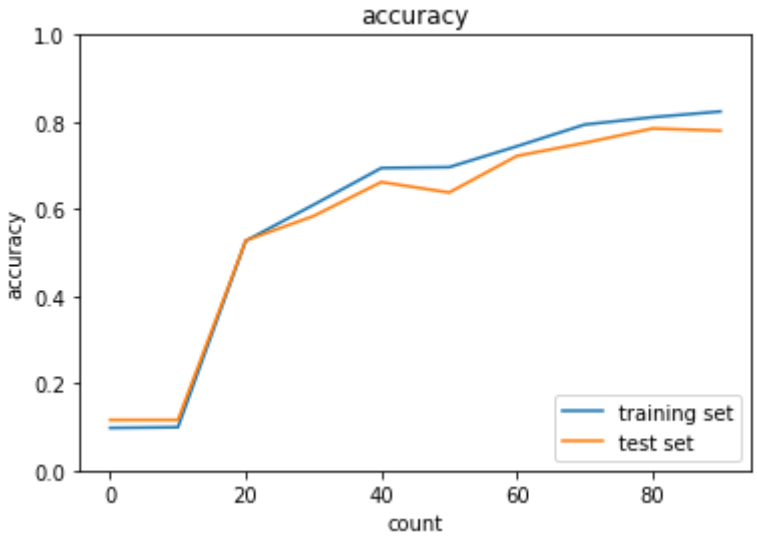
```
        grad['W4'], grad['b4'] = self.layers['Affine2'].dW, self.layers['Affine2'].db

        return grad
```

In [41]:

```
from common import optimizer

# データの読み込み
(x_train, d_train), (x_test, d_test) = load_mnist(flatten=False)

print("データ読み込み完了")
start = time.time()
# 処理に時間のかかる場合はデータを削減
x_train, d_train = x_train[:5000], d_train[:5000]
x_test, d_test = x_test[:1000], d_test[:1000]


network = DoubleConvNet(input_dim=(1,28,28),
                        conv_param_1={'filter_num':10, 'filter_size':7, 'pad':1, 'st
                        conv_param_2={'filter_num':20, 'filter_size':3, 'pad':1, 'st
                        hidden_size=100, output_size=10, weight_init_std=0.01)

optimizer = optimizer.Adam()

# 時間がかかるため100に設定
iters_num = 100
# iters_num = 1000
train_size = x_train.shape[0]
batch_size = 100

train_loss_list = []
accuracies_train = []
accuracies_test = []

plot_interval=10



for i in range(iters_num):
    batch_mask = np.random.choice(train_size, batch_size)
    x_batch = x_train[batch_mask]
    d_batch = d_train[batch_mask]

    grad = network.gradient(x_batch, d_batch)
    optimizer.update(network.params, grad)
    loss = network.loss(x_batch, d_batch)
    train_loss_list.append(loss)

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

        process_time = time.time() - start
        print(process_time)
        print('Generation: ' + str(i+1) + '. 正答率(トレーニング) = ' + str(accr_trai
        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()
```

```
データ読み込み完了
8.877216815948486
Generation: 10. 正答率(トレーニング) = 0.0976
                : 10. 正答率(テスト) = 0.116
17.647892236709595
Generation: 20. 正答率(トレーニング) = 0.0992
                : 20. 正答率(テスト) = 0.116
27.017614364624023
Generation: 30. 正答率(トレーニング) = 0.527
                : 30. 正答率(テスト) = 0.528
34.85488724708557
Generation: 40. 正答率(トレーニング) = 0.6098
                : 40. 正答率(テスト) = 0.584
42.806095123291016
Generation: 50. 正答率(トレーニング) = 0.6942
                : 50. 正答率(テスト) = 0.662
50.64944839477539
Generation: 60. 正答率(トレーニング) = 0.6962
                : 60. 正答率(テスト) = 0.638
58.88264727592468
Generation: 70. 正答率(トレーニング) = 0.7442
                : 70. 正答率(テスト) = 0.722
67.79273748397827
Generation: 80. 正答率(トレーニング) = 0.794
                : 80. 正答率(テスト) = 0.752
76.91342759132385
Generation: 90. 正答率(トレーニング) = 0.8108
                : 90. 正答率(テスト) = 0.785
86.37917971611023
Generation: 100. 正答率(トレーニング) = 0.824
                 : 100. 正答率(テスト) = 0.78
```



CNNの実装を試みた。層の仕組みを勉強するとともに、２層畳み込み層があるモデルにおいて、プーリング層を一つ消した場合の比較を行った。
実際には重みの初期値が違うということも考慮にいれなくてはいけないかもしれないが、プーリング層抜きだと154秒で正答率（テスト）が0.85に対し、プーリング層ありだと86秒で正答率（テスト）が0.78であった。
今回は同じイテレーション回数で比較したが、同じ秒数で比較した場合、どちらに優劣がつくかわからない。少なくとも、プーリング層があるせいで決定的に精度が落ちることはなく、解析時間をかなり短縮できることを確認できた。

Section 5 最近のCNN

In [ ]:

In [5]:
```
import pickle
import numpy as np
from collections import OrderedDict
from common import layers
from data.mnist import load_mnist
import matplotlib.pyplot as plt
from common import optimizer
import time
```

In [3]:
```
class DeepConvNet:
    '''
    認識率99%以上の高精度なConvNet

    conv - relu - conv- relu - pool -
    conv - relu - conv- relu - pool -
    conv - relu - conv- relu - pool -
    affine - relu - dropout - affine - dropout - softmax
    '''
    def __init__(self, input_dim=(1, 28, 28),
                 conv_param_1 = {'filter_num':16, 'filter_size':3, 'pad':1, 'stride':1
                 conv_param_2 = {'filter_num':16, 'filter_size':3, 'pad':1, 'stride':1
                 conv_param_3 = {'filter_num':32, 'filter_size':3, 'pad':1, 'stride':1
                 conv_param_4 = {'filter_num':32, 'filter_size':3, 'pad':2, 'stride':1
                 conv_param_5 = {'filter_num':64, 'filter_size':3, 'pad':1, 'stride':1
                 conv_param_6 = {'filter_num':64, 'filter_size':3, 'pad':1, 'stride':1
                 hidden_size=50, output_size=10):
        # 重みの初期化===========
        # 各層のニューロンひとつあたりが、前層のニューロンといくつのつながりがあるか
        pre_node_nums = np.array([1*3*3, 16*3*3, 16*3*3, 32*3*3, 32*3*3, 64*3*3, 64*
        wight_init_scales = np.sqrt(2.0 / pre_node_nums)  # Heの初期値

        self.params = {}
        pre_channel_num = input_dim[0]
        for idx, conv_param in enumerate([conv_param_1, conv_param_2, conv_param_3, c
            self.params['W' + str(idx+1)] = wight_init_scales[idx] * np.random.randn(
            self.params['b' + str(idx+1)] = np.zeros(conv_param['filter_num'])
            pre_channel_num = conv_param['filter_num']
        self.params['W7'] = wight_init_scales[6] * np.random.randn(pre_node_nums[6],
        print(self.params['W7'].shape)
        self.params['b7'] = np.zeros(hidden_size)
        self.params['W8'] = wight_init_scales[7] * np.random.randn(pre_node_nums[7],
        self.params['b8'] = np.zeros(output_size)

        # レイヤの生成===========
        self.layers = []
        self.layers.append(layers.Convolution(self.params['W1'], self.params['b1'],
                           conv_param_1['stride'], conv_param_1['pad']))
        self.layers.append(layers.Relu())
        self.layers.append(layers.Convolution(self.params['W2'], self.params['b2'],
                           conv_param_2['stride'], conv_param_2['pad']))
        self.layers.append(layers.Relu())
        self.layers.append(layers.Pooling(pool_h=2, pool_w=2, stride=2))
        self.layers.append(layers.Convolution(self.params['W3'], self.params['b3'],
                           conv_param_3['stride'], conv_param_3['pad']))
        self.layers.append(layers.Relu())
        self.layers.append(layers.Convolution(self.params['W4'], self.params['b4'],
                           conv_param_4['stride'], conv_param_4['pad']))
```

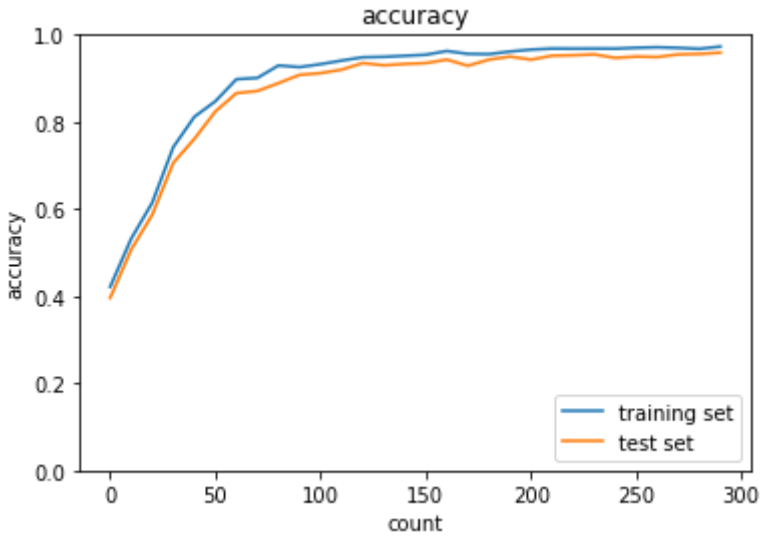
```
        self.layers.append(layers.Relu())
        self.layers.append(layers.Pooling(pool_h=2, pool_w=2, stride=2))
        self.layers.append(layers.Convolution(self.params['W5'], self.params['b5'],
                           conv_param_5['stride'], conv_param_5['pad']))
        self.layers.append(layers.Relu())
        self.layers.append(layers.Convolution(self.params['W6'], self.params['b6'],
                           conv_param_6['stride'], conv_param_6['pad']))
        self.layers.append(layers.Relu())
        self.layers.append(layers.Pooling(pool_h=2, pool_w=2, stride=2))
        self.layers.append(layers.Affine(self.params['W7'], self.params['b7']))
        self.layers.append(layers.Relu())
        self.layers.append(layers.Dropout(0.5))
        self.layers.append(layers.Affine(self.params['W8'], self.params['b8']))
        self.layers.append(layers.Dropout(0.5))

        self.last_layer = layers.SoftmaxWithLoss()

    def predict(self, x, train_flg=False):
        for layer in self.layers:
            if isinstance(layer, layers.Dropout):
                x = layer.forward(x, train_flg)
            else:
                x = layer.forward(x)
        return x

    def loss(self, x, d):
        y = self.predict(x, train_flg=True)
        return self.last_layer.forward(y, d)

    def accuracy(self, x, d, batch_size=100):
        if d.ndim != 1 : d = np.argmax(d, axis=1)

        acc = 0.0

        for i in range(int(x.shape[0] / batch_size)):
            tx = x[i*batch_size:(i+1)*batch_size]
            td = d[i*batch_size:(i+1)*batch_size]
            y = self.predict(tx, train_flg=False)
            y = np.argmax(y, axis=1)
            acc += np.sum(y == td)

        return acc / x.shape[0]

    def gradient(self, x, d):
        # forward
        self.loss(x, d)

        # backward
        dout = 1
        dout = self.last_layer.backward(dout)

        tmp_layers = self.layers.copy()
        tmp_layers.reverse()
        for layer in tmp_layers:
            dout = layer.backward(dout)

        # 設定
        grads = {}
        for i, layer_idx in enumerate((0, 2, 5, 7, 10, 12, 15, 18)):
            grads['W' + str(i+1)] = self.layers[layer_idx].dW
            grads['b' + str(i+1)] = self.layers[layer_idx].db

        return grads
```

In [12]:
```
from common import optimizer

(x_train, d_train), (x_test, d_test) = load_mnist(flatten=False)

# 処理に時間のかかる場合はデータを削減
x_train, d_train = x_train[:5000], d_train[:5000]
x_test, d_test = x_test[:1000], d_test[:1000]

print("データ読み込み完了　解析開始")
start = time.time()

network = DeepConvNet()
optimizer = optimizer.Adam()

iters_num = 300
train_size = x_train.shape[0]
batch_size = 100

train_loss_list = []
accuracies_train = []
accuracies_test = []

plot_interval=10


for i in range(iters_num):
    batch_mask = np.random.choice(train_size, batch_size)
    x_batch = x_train[batch_mask]
    d_batch = d_train[batch_mask]

    grad = network.gradient(x_batch, d_batch)
    optimizer.update(network.params, grad)

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

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

        process_time = time.time() - start
        print(process_time)

        print('Generation: ' + str(i+1) + '. 正答率(トレーニング) = ' + str(accr_trai
        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()
```

```
データ読み込み完了　解析開始
(1024, 50)
71.18267250061035
Generation: 10. 正答率(トレーニング) = 0.4216
                 : 10. 正答率(テスト) = 0.396
137.20938205718994
Generation: 20. 正答率(トレーニング) = 0.532
```

```
                : 20. 正答率(テスト) = 0.507
203.4637713432312
Generation: 30. 正答率(トレーニング) = 0.6148
                : 30. 正答率(テスト) = 0.587
273.37864661216736
Generation: 40. 正答率(トレーニング) = 0.7422
                : 40. 正答率(テスト) = 0.706
340.0337264537811
Generation: 50. 正答率(トレーニング) = 0.8116
                : 50. 正答率(テスト) = 0.761
406.65348076820374
Generation: 60. 正答率(トレーニング) = 0.8474
                : 60. 正答率(テスト) = 0.824
473.4613709449768
Generation: 70. 正答率(トレーニング) = 0.898
                : 70. 正答率(テスト) = 0.866
539.8696601390839
Generation: 80. 正答率(トレーニング) = 0.901
                : 80. 正答率(テスト) = 0.871
606.5378136634827
Generation: 90. 正答率(トレーニング) = 0.9296
                : 90. 正答率(テスト) = 0.889
672.9677352905273
Generation: 100. 正答率(トレーニング) = 0.9258
                : 100. 正答率(テスト) = 0.908
739.1366457939148
Generation: 110. 正答率(トレーニング) = 0.9326
                : 110. 正答率(テスト) = 0.912
805.5427045822144
Generation: 120. 正答率(トレーニング) = 0.9408
                : 120. 正答率(テスト) = 0.92
872.9223501682281
Generation: 130. 正答率(トレーニング) = 0.9484
                : 130. 正答率(テスト) = 0.935
939.0733180046082
Generation: 140. 正答率(トレーニング) = 0.9494
                : 140. 正答率(テスト) = 0.93
1005.5923912525177
Generation: 150. 正答率(トレーニング) = 0.9518
                : 150. 正答率(テスト) = 0.933
1071.9495239257812
Generation: 160. 正答率(トレーニング) = 0.9542
                : 160. 正答率(テスト) = 0.935
1137.995910167694
Generation: 170. 正答率(トレーニング) = 0.9628
                : 170. 正答率(テスト) = 0.943
1204.2031939029694
Generation: 180. 正答率(トレーニング) = 0.9562
                : 180. 正答率(テスト) = 0.929
1270.436176776886
Generation: 190. 正答率(トレーニング) = 0.9556
                : 190. 正答率(テスト) = 0.943
1336.7942032814026
Generation: 200. 正答率(トレーニング) = 0.9616
                : 200. 正答率(テスト) = 0.95
1402.9214317798615
Generation: 210. 正答率(トレーニング) = 0.966
                : 210. 正答率(テスト) = 0.943
1469.4618694782257
Generation: 220. 正答率(トレーニング) = 0.9684
                : 220. 正答率(テスト) = 0.952
1535.1021420955658
Generation: 230. 正答率(トレーニング) = 0.9682
                : 230. 正答率(テスト) = 0.953
1600.81880402565
Generation: 240. 正答率(トレーニング) = 0.9684
                : 240. 正答率(テスト) = 0.955
1668.5313398838043
Generation: 250. 正答率(トレーニング) = 0.9682
                : 250. 正答率(テスト) = 0.947
1739.4128260612488
Generation: 260. 正答率(トレーニング) = 0.9702
                : 260. 正答率(テスト) = 0.95
1805.298320531845
Generation: 270. 正答率(トレーニング) = 0.9714
                : 270. 正答率(テスト) = 0.949
1877.8751652240753
Generation: 280. 正答率(トレーニング) = 0.97
                : 280. 正答率(テスト) = 0.955
1944.0977573394775
```

```
Generation: 290. 正答率(トレーニング) = 0.9678
                : 290. 正答率(テスト) = 0.956
2010.1307580471039
Generation: 300. 正答率(トレーニング) = 0.973
                : 300. 正答率(テスト) = 0.959
```



深層CNNのさわりを試みた。今回のモデルでも、実際現役で使われるようなモデルに対して学習データ数もそんなに多くない上に、層もとても厚いわけではないが、それでもかなりの時間を要した。
NN系の開発に、GPUを使ったり、バッチ学習を用いたりした理由を肌感的に体験できた。