Your repository details have been saved.

syakesaba / deep-learning-from-scratch
forked from oreilly-japan/deep-learning-from-scratch

https://notebooks.azure.com/syakesaba/libraries/deep-learning-from-scratch https://notebooks.azure.com/syakesaba...

Edit

Manage topics

104 commits | 3 branches | 1 release | 7 contributors | MIT

Branch: master | New pull request | Create new file | Upload files | Find file | Clone or download

This branch is 39 commits ahead, 13 commits behind oreilly-japan:master. Pull request Compare

syakesaba a — Latest commit 5a32c4b 5 days ago

ch01	Bactch Normalizationまで	4 months ago
ch02	日本語が入ると駄目っぽい→latex	4 months ago
ch03	日本語が入ると駄目っぽい→latex	4 months ago
ch04	日本語が入ると駄目っぽい→latex	4 months ago
ch05	日本語が入ると駄目っぽい→latex	4 months ago
ch06	ch6 end	4 months ago
ch07	Ch07.End	5 days ago
ch08	ch08.end	5 days ago
common	Add delta(1e-7) for cross_entropy_error	9 months ago
dataset	fixfix	5 months ago
.gitignore	added gitignore	9 months ago
LICENSE.md	updated to DOT graphviz	9 months ago
README.md	a	5 days ago

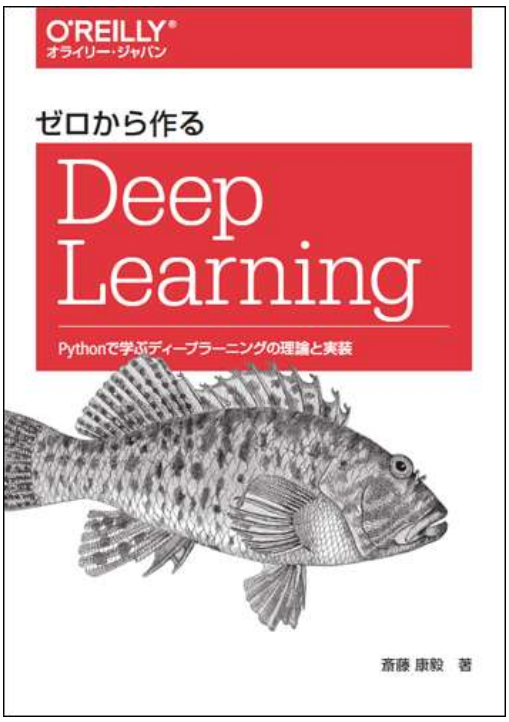
README.md

ゼロから作る Deep Learning

syakesabaの勉強用ノートのリポジトリ。 それぞれのチャプターに合わせてipynbファイルを作成し JupyterNotebookで実際にコードを動かしてグラフを出したりしてます。

以下、オリジナルの方のREADME.md

ゼロから作る Deep Learning



本リポジトリはオライリー・ジャパン発行書籍『ゼロから作る Deep Learning』のサポートサイトです。

ファイル構成

フォルダ名	説明
ch01	1章で使用するソースコード
ch02	2章で使用するソースコード
...	...
ch08	8章で使用するソースコード
common	共通で使用するソースコード
dataset	データセット用のソースコード

ソースコードの解説は本書籍をご覧ください。

必要条件

ソースコードを実行するには、下記のソフトウェアがインストールされている必要があります。

- Python 3.x
- NumPy
- Matplotlib

※Pythonのバージョンは、3系を利用します。

実行方法

各章のフォルダへ移動して、Pythonコマンドを実行します。

```
$ cd ch01
$ python man.py

$ cd ../ch05
$ python train_nueralnet.py
```

ライセンス

本リポジトリのソースコードはMITライセンスです。 商用・非商用問わず、自由にご利用ください。

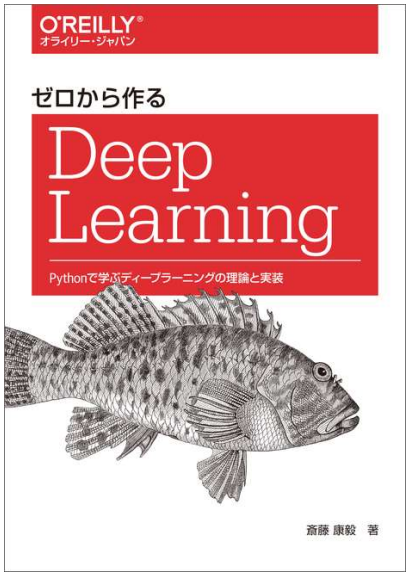
正誤表

本書の正誤情報は以下のページで公開しています。

https://github.com/oreilly-japan/deep-learning-from-scratch/wiki/errata

本ページに掲載されていない誤植など間違いを見つけた方は、japan@oreilly.co.jpまでお知らせください。

SYA-KE Deep Learning Training



Parent Repository URL

https://github.com/oreilly-japan/deep-learning-from-scratch (https://github.com/oreilly-japan/deep-learning-from-scratch)

Pythonに慣れる

In [1]:

```
!python --version
```

Python 3.6.5

In [2]:

```
#もう何年もやってるので省略
```

Numpyに慣れる

参考URL

http://www.sist.ac.jp/~kanakubo/research/hosoku/gyoretu.html
(http://www.sist.ac.jp/~kanakubo/research/hosoku/gyoretu.html)

配列、行列

In [3]:

```
import numpy as np
```

In [4]:

```
np.__version__
```

Out[4]:

```
'1.14.3'
```

In [5]:

```
x = np.array([1.0,2.0,3.0])
```

In [6]:

```
y = np.array([2.0,4.0,6.0])
```

In [7]:

```
x + y
```

Out[7]:

```
array([3., 6., 9.])
```

In [8]:

```
x - y
```

Out[8]:

```
array([-1., -2., -3.])
```

In [9]:

```
x * y
```

Out[9]:

```
array([ 2.,  8., 18.])
```

In [10]:

```
x / y
```

Out[10]:

```
array([0.5, 0.5, 0.5])
```

In [11]:

```
x / 2.0
```

Out[11]:

```
array([0.5, 1. , 1.5])
```

In [12]:

```
A = np.array([[1,2],[3,4]])
```

In [13]:

```
print(A)
```

```
[[1 2]
 [3 4]]
```

In [14]:

```
A.shape
```

Out[14]:

```
(2, 2)
```

In [15]:

```
A.dtype
```

Out[15]:

```
dtype('int32')
```

In [16]:

```
B = np.array([[3,0],[0,6]])
```

In [17]:

```
A + B
```

Out[17]:

```
array([[ 4,  2],
       [ 3, 10]])
```

In [18]:

```
A * B
```

Out[18]:

```
array([[ 3,  0],
       [ 0, 24]])
```

In [19]:

```
A*10
```

Out[19]:

```
array([[10, 20],
       [30, 40]])
```

参考 (http://www.scipy-lectures.org/intro/numpy/operations.html)

In [20]:

```
X = np.array([[51,55],[14,19],[0,4]])
X
```

Out[20]:

```
array([[51, 55],
       [14, 19],
       [ 0,  4]])
```

In [21]:

```
X[0] #0行目
```

Out[21]:

```
array([51, 55])
```

In [22]:

```
X[0][1] #(0, 1) =>(0行1列)
```

Out[22]:

```
55
```

In [23]:

```
for row in X:
    print(row)
```

```
[51 55]
[14 19]
[0 4]
```

In [24]:

```
AX = X.flatten() #行列を配列に
AX
```

Out[24]:

```
array([51, 55, 14, 19,  0,  4])
```

In [25]:

```
AX[AX>15]#配列から条件の値を抽出
```

Out[25]:

```
array([51, 55, 19])
```

In [26]:

```
X>15
```

Out[26]:

```
array([[ True,  True],
       [False,  True],
       [False, False]])
```

In [27]:

```
X[X>15]#行列でもできる。
```

Out[27]:

```
array([51, 55, 19])
```

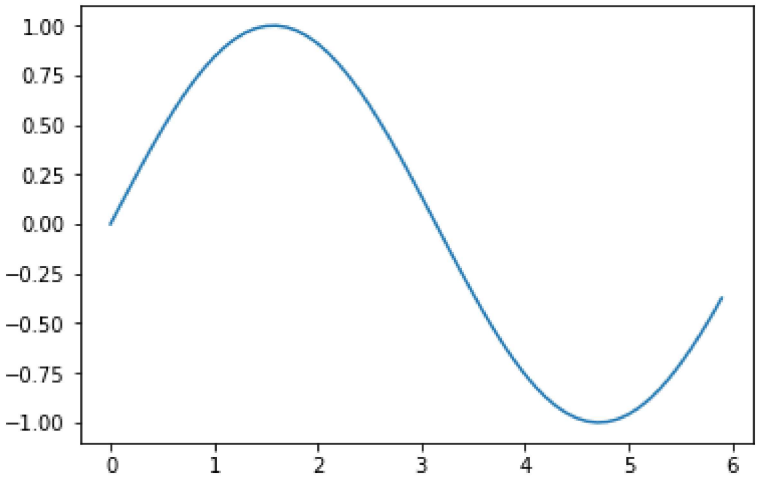
ニューラルネットワークにおいては行列は　行数＝入力数 列数＝重み数

Matplotlibに慣れる

参考URL (http://qiita.com/hik0107/items/de5785f680096df93efa)

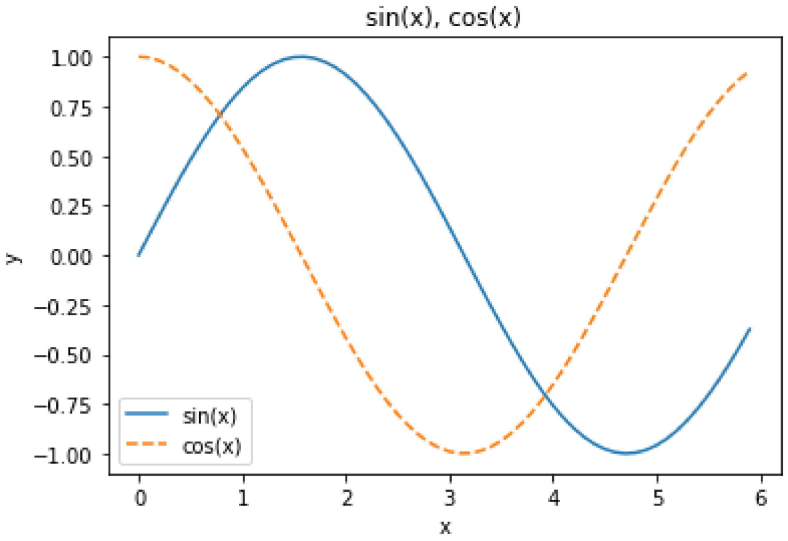
In [28]:

```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np

x = np.arange(0,6,0.1) #0to6 by 0.1 => sin(0to6)
y = np.sin(x)
plt.plot(x,y)
plt.show()
plt.close()
```



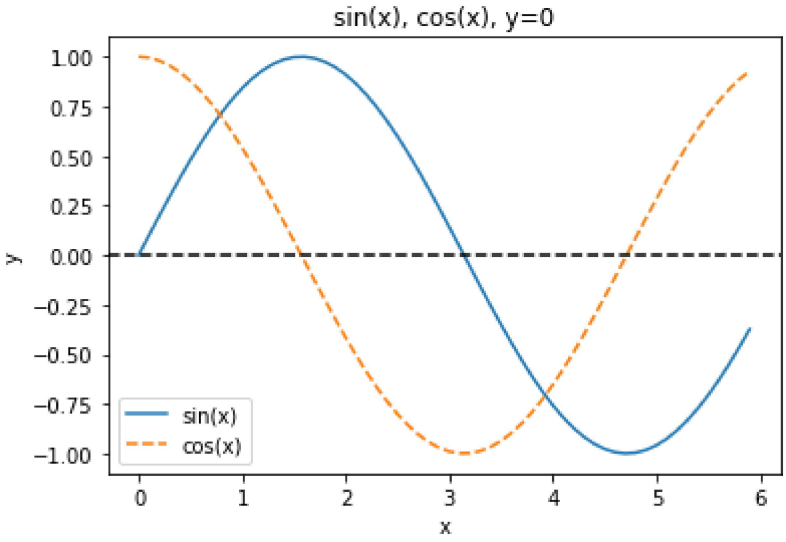
In [29]:

```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np

x = np.arange(0,6,0.1) #0to6 by 0.1 => sin(0to6)
y1 = np.sin(x)
y2 = np.cos(x)
plt.plot(x,y1,label="sin(x)")
plt.plot(x,y2,label="cos(x)", linestyle="--")
plt.xlabel("x")
plt.ylabel("y")
plt.title("sin(x), cos(x)")
plt.legend()#show label box
plt.show()
plt.close()
```



In [30]:

```
%matplotlib inline
#plt.show()をinlineにて表示する
#Esc+Lで行番号を表示する
import matplotlib.pyplot as plt
import numpy as np

x = np.arange(0,6,0.1) #0to6 by 0.1 => sin(0to6)
y1 = np.sin(x)
y2 = np.cos(x)
plt.plot(x,y1,label="sin(x)")
plt.plot(x,y2,label="cos(x)", linestyle="--")
plt.axhline(color='black',linestyle="--")
plt.xlabel("x")
plt.ylabel("y")
plt.title("sin(x), cos(x), y=0")
plt.legend()#show label box
plt.show()
plt.close()
```

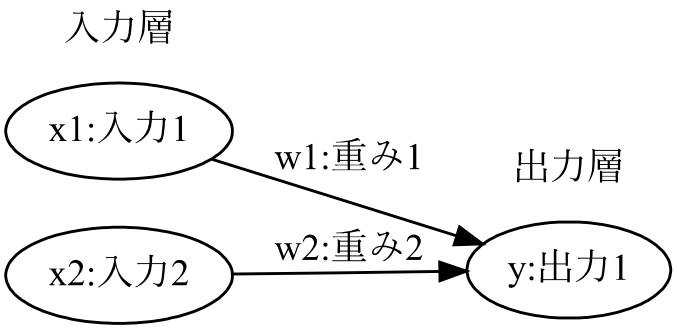


パーセプトロンに慣れる

In [31]:

```
from graphviz import Digraph
dot = Digraph(comment="単純パーセプトロン")
dot.attr(rankdir="LR")
dot.attr(splines="line")
dot.attr(fixedsize="true")
with dot.subgraph(name="cluster_x") as x:
    x.attr(label="入力層")
    x.attr(color="white")
    x.node("x1","x1:入力1")
    x.node("x2","x2:入力2")
with dot.subgraph(name="cluster_y") as y:
    y.attr(label="出力層")
    y.attr(color="white")
    y.node("y","y:出力1")
dot.edge("x1", "y", label="w1:重み1")
dot.edge("x2",  "y", label="w2:重み2")
#print(dot)
dot
```

Out[31]:

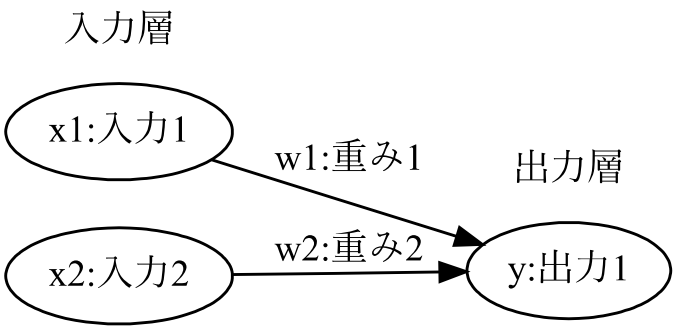


パーセプトロンに慣れる

In [1]:

```
from graphviz import Digraph
dot = Digraph(comment="単純パーセプトロン")
dot.attr(rankdir="LR")
dot.attr(splines="line")
dot.attr(fixedsize="true")
with dot.subgraph(name="cluster_x") as x:
    x.attr(label="入力層")
    x.attr(color="white")
    x.node("x1","x1:入力1")
    x.node("x2","x2:入力2")
with dot.subgraph(name="cluster_y") as y:
    y.attr(label="出力層")
    y.attr(color="white")
    y.node("y","y:出力1")
dot.edge("x1", "y", label="w1:重み1")
dot.edge("x2",  "y", label="w2:重み2")
#print(dot)
dot
```

Out[1]:



今、x1:入力1とx2:入力2を0～1の変数と仮定すると、以下のように発火判定をして任意のCPU処理を実装することが可能になる。
つまり、パーセプトロンは**チューリング完全**であり、電子コンピュータで出来ることがパーセプトロンでも出来ることを示している。

$$AND \quad y = \begin{cases} 0 & (0.5x_1 + 0.5x_2 \leq 0.6) \\ 1 & (0.5x_1 + 0.5x_2 > 0.6) \end{cases}$$

In [2]:

```
def AND(x1, x2):
    w1, w2, theta = 0.5, 0.5, 0.6 #0001
    tmp = x1*w1 + x2*w2
    if tmp <= theta:
        return 0
    else:
        return 1
```

$$OR \quad y = \begin{cases} 0 & (0.5x_1 + 0.5x_2 \leq 0.4) \\ 1 & (0.5x_1 + 0.5x_2 > 0.4) \end{cases}$$

In [3]:

```
def OR(x1, x2):
    w1, w2, theta = 0.5, 0.5, 0.4 #0111
    tmp = x1*w1 + x2*w2
    if tmp <= theta:
        return 0
    else:
        return 1
```

$$NAND \quad y = \begin{cases} 0 & (0.5x_1 + 0.5x_2 > 0.6) \\ 1 & (0.5x_1 + 0.5x_2 \leq 0.6) \end{cases}$$

In [4]:

```
def NAND(x1, x2):
    w1, w2, theta = 0.5, 0.5, 0.6 #1110
    tmp = x1*w1 + x2*w2
    if tmp > theta:
        return 0
    else:
        return 1
```

$$NOR \quad y = \begin{cases} 0 & (0.5x_1 + 0.5x_2 > 0.4) \\ 1 & (0.5x_1 + 0.5x_2 \leq 0.4) \end{cases}$$

In [5]:

```
def NOR(x1, x2):
    w1, w2, theta = 0.5, 0.5, 0.4 #1000
    tmp = x1*w1 + x2*w2
    if tmp > theta:
        return 0
    else:
        return 1
```

$$XOR \quad y = \begin{Bmatrix} 0 & AND(NAND, OR) \\ 1 & AND(NAND, OR) \end{Bmatrix}$$

In [1]:

```
def XOR(x1, x2):
    AND_x1x2 = AND(x1,x2) # 0001
    OR_x1x2 = OR(x1, x2) #0111
    AND_x1x2 = NAND(x1,x2) #1110
    return AND(NAND_x1x2, OR_x1x2) #0110
```

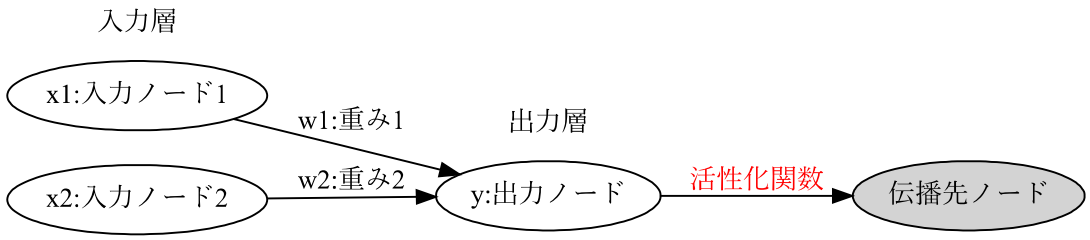
活性化関数

発火するか、しないか or 発火するならどのくらいの発火なのかを定める

In [1]:

```
from graphviz import Digraph
dot = Digraph(comment="単純パーセプトロン")
dot.attr(rankdir="LR")
dot.attr(splines="line")
dot.attr(fixedsize="true")
with dot.subgraph(name="cluster_x") as x:
    x.attr(label="入力層")
    x.attr(color="white")
    x.node("x1","x1:入力ノード1")
    x.node("x2","x2:入力ノード2")
with dot.subgraph(name="cluster_y") as y:
    y.attr(label="出力層")
    y.attr(color="white")
    y.node("y","y:出力ノード")
with dot.subgraph(name="cluster_a") as a:
    a.attr(color="white")
    a.node("a","伝播先ノード",style="filled")

dot.edge("x1", "y", label="w1:重み1")
dot.edge("x2",  "y", label="w2:重み2")
dot.edge("y",  "a", label="活性化関数",fontcolor="red")
#print(dot)
dot
```

Out[1]:



Sigmoid関数 (シグモイド関数)

便利な非線形関数

$h_\theta(x) = \frac{1}{1+e^{-\theta x}}$

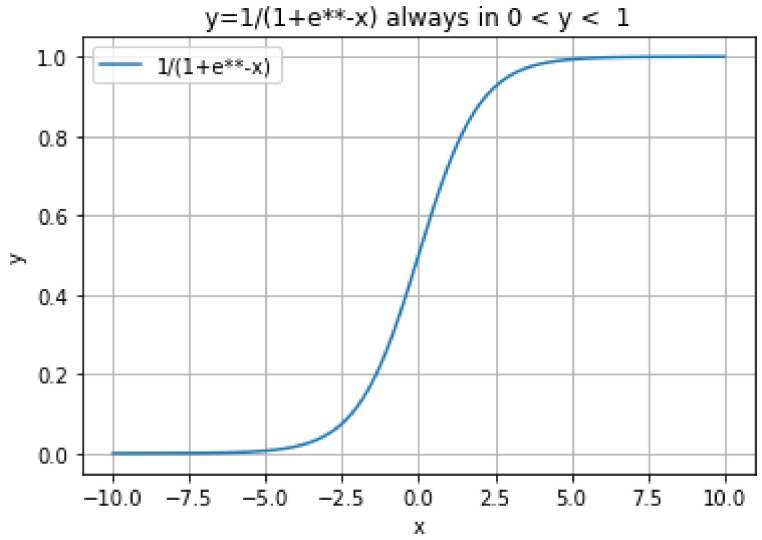
これは$\theta = 1$の時、標準シグモイド関数となる。

$h_1(x) = \frac{1}{1+e^{-x}}$

標準シグモイド関数はxの値がいかなる数値であっても結果は$0 < h_1(x) < 1$となる

In [2]:

```
%matplotlib inline
#plt.show()をinlineにて表示する
#Esc+Lで行番号を表示する1
import matplotlib.pyplot as plt
import numpy as np

x = np.arange(-10,10,0.001) #-0 to 10 by 0.001
y = 1.0/(1.0 + np.e ** (-1.0*x))

plt.plot(x,y,label="1/(1+e**-x)", linestyle="-")
plt.xlabel("x")
plt.ylabel("y")
plt.title("y=1/(1+e**-x) always in 0 < y <  1")
plt.legend()#show label box
plt.grid()
plt.show()
plt.close()
```



ステップ関数

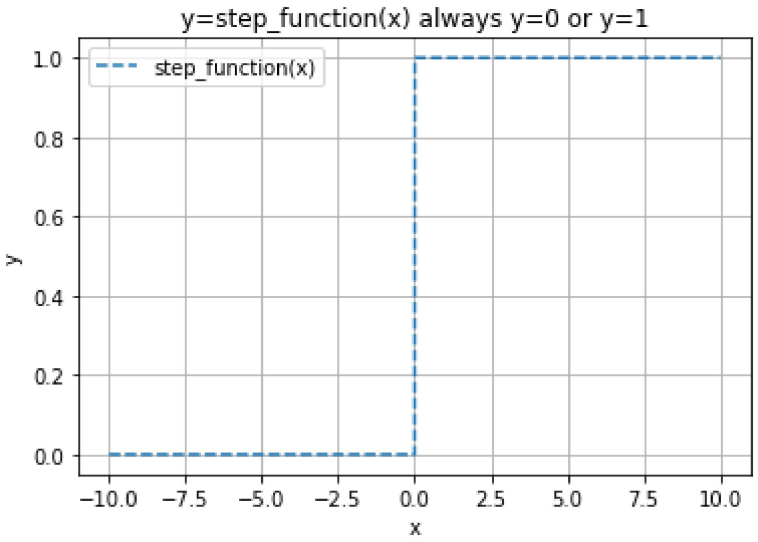
xがどの値であっても0か1を指す非線形関数

In [3]:

```
import numpy as np
def step_function(x):
    """x: numpy.Array"""
    y = x > 0
    #x(int)がboolに変換されるので、またintに直す
    return y.astype(np.int)
```

In [4]:

```
%matplotlib inline
#plt.show()をinlineにて表示する
#Esc+Lで行番号を表示する1
import matplotlib.pyplot as plt
import numpy as np

import numpy as np
def step_function(x):
    y = x > 0
    return y.astype(np.int)

x = np.arange(-10,10,0.001) #-10 to 10 by 0.001
y = step_function(x)
plt.plot(x,y,label="step_function(x)", linestyle="--")
plt.xlabel("x")
plt.ylabel("y")
plt.title("y=step_function(x) always y=0 or y=1")
plt.legend()#show label box
plt.grid()
plt.show()
plt.close()
```

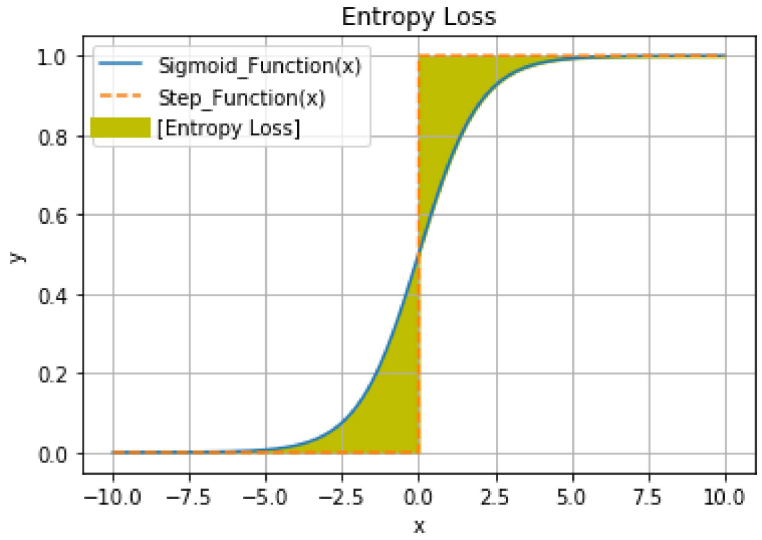


シグモイド関数とステップ関数の関係

ステップ関数では情報量が欠ける

In [5]:

```
%matplotlib inline
#plt.show()をinlineにて表示する
#Esc+Lで行番号を表示する1
import matplotlib.pyplot as plt
import numpy as np

x = np.arange(-10,10,0.001) #-0 to 10 by 0.001
y1 = 1.0/(1.0 + np.e ** (-1.0*x))

def step_function(x):
    y = x > 0
    return y.astype(np.int)

y2 = step_function(x)

plt.plot(x,y1,label="Sigmoid_Function(x)", linestyle="-")
plt.plot(x,y2,label="Step_Function(x)", linestyle="--")
plt.fill_between(x,y1,y2,color="y")
plt.plot([], [], color="y", linewidth=10, label="[Entropy Loss]")
plt.title("""Entropy Loss""")
plt.xlabel("x")
plt.ylabel("y")
plt.legend()#show label box
plt.grid()
plt.show()
plt.close()
```



活性関数は非線形関数でなければならない

活性関数が線形関数の場合は計算結果が予測可能となり計算式も因数分解が可能となる(＝一瞬で演算が収束する)。そうなった場合多層パーセプトロンは組む意味がない。今後は非線形(パーセプトロンやシグモイドのような)になることを意識しなければならない。活性化関数に線形関数を使っても得られる情報は価値（情報量、エントロピー）が少ない。

活性関数の特徴

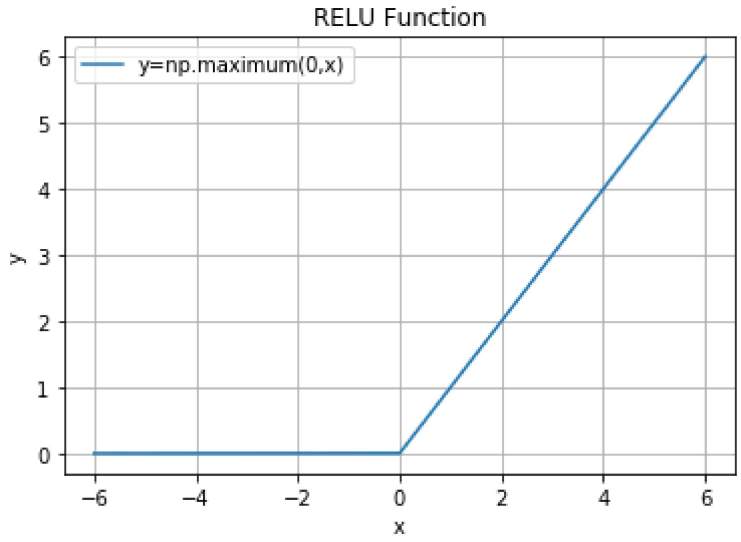
- ステップ関数：劇的な信号の変化を発生させる
- シグモイド関数：緩やかな信号の変化を発生させる
- RELU：発火しないものに関して逆伝搬をさせなくする（ニューロンの無効化）

RELU関数（Rectified Liner Unit）

$$h(x) = \begin{cases} x & (x > 0) \\ 0 & (x \leq 0) \end{cases}$$

In [6]:

```
%matplotlib inline
#plt.show()をinlineにて表示する
#Esc+Lで行番号を表示する1
import matplotlib.pyplot as plt
import numpy as np

x = np.arange(-6,6,0.001) #-6 to 6 by 0.001

def relu(x):
    return np.maximum(0,x)

plt.plot(x,relu(x),label="y=np.maximum(0,x)", linestyle="-")
plt.xlabel("x")
plt.ylabel("y")
plt.title("RELU Function")
plt.xlabel("x")
plt.ylabel("y")
plt.legend()#show label box
plt.grid()
plt.show()
plt.close()
```



多次元配列に慣れる

In [7]:

```
import numpy as np
A = np.array([1,2,3,4])
print(A)
print("何次元配列➡",A.ndim)
print("何行、何列➡",A.shape)
```

```
[1 2 3 4]
何次元配列➡ 1
何行、何列➡ (4,)
```

In [8]:

```
import numpy as np
A = np.array([[1,2,3,4],[5,6,7,8]])
print(A)
print("何次元配列➡",A.ndim)
print("何行、何列➡",A.shape)
```

```
[[1 2 3 4]
 [5 6 7 8]]
何次元配列➡ 2
何行、何列➡ (2, 4)
```

In [9]:

```
import numpy as np
A = np.array([[1,2,3,4],[5,6,7,8],[0,-1,-2,-3]])
print(A)
print("何次元配列➡",A.ndim)
print("何行、何列➡",A.shape)
```

```
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 0 -1 -2 -3]]
何次元配列➡ 2
何行、何列➡ (3, 4)
```

内積(ドット積)

In [10]:

```
import numpy as np
A = np.array([[1,2,3,4],[5,6,7,8],[0,-1,-2,-3]])
B = np.array([[1,2,3],[5,6,7],[0,-1,-2],[4,5,6]])
print("何次元配列➡",A.ndim)
print("何行、何列➡",A.shape)
print(A)
print("x")
print("何次元配列➡",B.ndim)
print("何行、何列➡",B.shape)
print(B)
print("=")
C=np.dot(A,B)
print(C)
print("何次元配列➡",C.ndim)
print("何行、何列➡",C.shape)
```

```
何次元配列➡ 2
何行、何列➡ (3, 4)
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 0 -1 -2 -3]]
x
何次元配列➡ 2
何行、何列➡ (4, 3)
[[ 1  2  3]
 [ 5  6  7]
 [ 0 -1 -2]
 [ 4  5  6]]
=
[[ 27  31  35]
 [ 67  79  91]
 [-17 -19 -21]]
何次元配列➡ 2
何行、何列➡ (3, 3)
```

ニューラルネットワークにおける内積(ドット積)

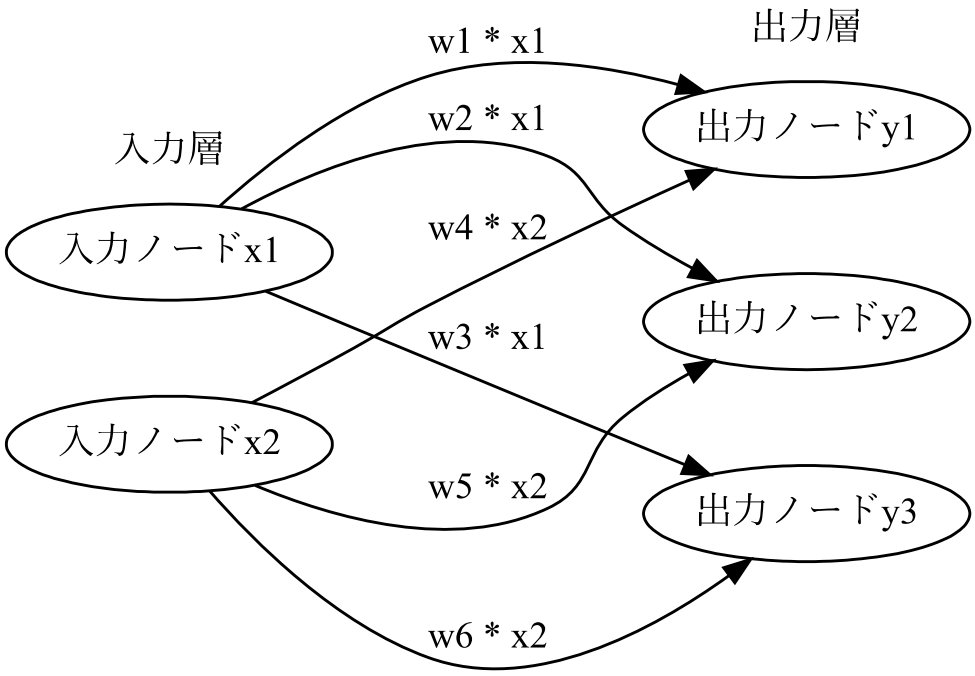
行列に見立てる事により、一度に計算が可能となり、なおかつ行列計算式が使用可能となる

一層ニューラルネットワークの場合（バイヤス無し）

In [11]:

```
from graphviz import Digraph
dot = Digraph(comment="単純パーセプトロン")
dot.attr(rankdir="LR")
dot.attr(ranksep="1.0")
dot.attr(nodesep="0.5")
with dot.subgraph(name="cluster_x") as x:
    x.attr(label="入力層")
    x.attr(color="white")
    x.node("x1","入力ノードx1")
    x.node("x2","入力ノードx2")
with dot.subgraph(name="cluster_y") as y:
    y.attr(label="出力層")
    y.attr(color="white")
    y.node("y1","出力ノードy1")
    y.node("y2","出力ノードy2")
    y.node("y3","出力ノードy3")
dot.edge("x1", "y1", label="w1 * x1")
dot.edge("x1", "y2", label="w2 * x1")
dot.edge("x1", "y3", label="w3 * x1")
dot.edge("x2", "y1", label="w4 * x2")
dot.edge("x2", "y2", label="w5 * x2")
dot.edge("x2", "y3", label="w6 * x2")
dot
```

Out[11]:



In [12]:

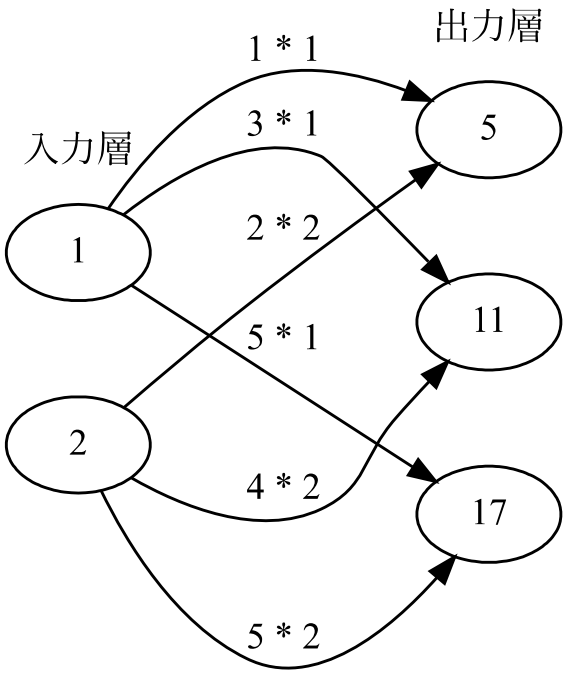
```
import numpy as np
x1 = 1
x2 = 2
X = np.array([x1,x2])
x1_W = [1,3,5]
x2_W = [2,4,6]
W = np.array([x1_W, x2_W])
Y = np.dot(X,W)
print("入力")
print(X)
print("出力重み")
print(W)
print("出力")
y1,y2,y3 = Y
print("y1=",y1)
print("y2=",y2)
print("y3=",y3)
#np.dotの一行で各行列の計算ができた→ニューラルネットワークの伝搬計算がすぐさま可能に
```

```
入力
[1 2]
出力重み
[[1 3 5]
 [2 4 6]]
出力
y1= 5
y2= 11
y3= 17
```

In [13]:

```
import numpy as np
x1 = 1
x2 = 2
X = np.array([x1,x2])
x1_W = [1,3,5]
x2_W = [2,4,6]
W = np.array([x1_W, x2_W])
Y = np.dot(X,W)
y1,y2,y3 = Y

from graphviz import Digraph
dot = Digraph(comment="単純パーセプトロン")
dot.attr(rankdir="LR")
dot.attr(ranksep="1.0")
dot.attr(nodesep="0.5")
with dot.subgraph(name="cluster_x") as x:
    x.attr(label="入力層")
    x.attr(color="white")
    x.node("x1","1")
    x.node("x2","2")
with dot.subgraph(name="cluster_y") as y:
    y.attr(label="出力層")
    y.attr(color="white")
    y.node("y1","5")
    y.node("y2","11")
    y.node("y3","17")
dot.edge("x1", "y1", label="1 * 1")
dot.edge("x1", "y2", label="3 * 1")
dot.edge("x1", "y3", label="5 * 1")
dot.edge("x2", "y1", label="2 * 2")
dot.edge("x2", "y2", label="4 * 2")
dot.edge("x2", "y3", label="5 * 2")
dot
```

Out[13]:

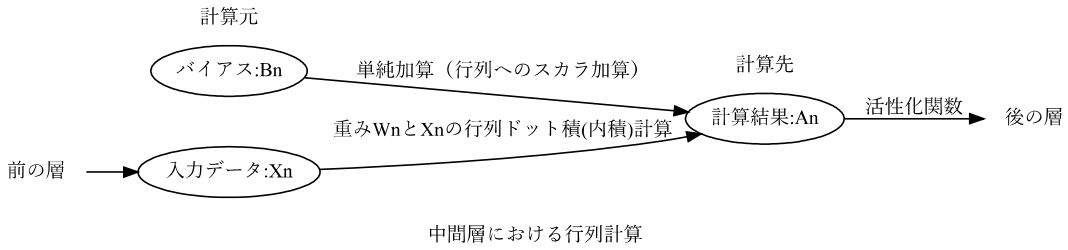


多層ニューラルネットワーク

In [14]:

```
from graphviz import Digraph
dot = Digraph()
dot.attr(label="中間層における行列計算")
dot.attr(rankdir="LR")
dot.attr(ranksep="0.5")
dot.attr(nodesep="0.5")
with dot.subgraph(name="cluster_in") as x:
    x.attr(label="")
    x.attr(color="white")
    x.node("in1","前の層",color="white")
with dot.subgraph(name="cluster_x") as x:
    x.attr(label="計算元")
    x.attr(color="white")
    x.node("b1","バイアス:Bn")
    x.node("x1","入力データ:Xn")
with dot.subgraph(name="cluster_y") as y:
    y.attr(label="計算先")
    y.attr(color="white")
    y.node("y1","計算結果:An")
with dot.subgraph(name="cluster_out") as y:
    y.attr(label="")
    y.attr(color="white")
    y.node("out1","後の層",color="white")

dot.edge("in1","x1", label="")
dot.edge("b1","y1", label="単純加算（行列へのスカラ加算）")
dot.edge("x1", "y1", label="重みWnとXnの行列ドット積(内積)計算")
dot.edge("y1", "out1", label="活性化関数")
dot
```

Out[14]:



Calc $A^n = XW^n + B^n$ Result $A^n = (a^1, a^2, a^3, \ldots, a^n)$ Input $X^n = (x^1, x^2, \ldots, x^n)$ Bias

$$B^n = (b^1, b^2, b^3, \ldots, b^n)\ Weight\ W^n = \left\{ \begin{array}{l} W_{(1,1)}, W_{1,2)}, W_{(1,3)}, \ldots \\ W_{(2,1)}, W_{(2,2)}, W_{(2,3)}, \ldots \\ W_{(3,1)}, W_{(3,2)}, W_{(3,3)}, \ldots \\ \quad \ldots \\ W_{(N,1)}, W_{(N,2)}, W_{N,3)}, \ldots \end{array} \right\}$$

In [15]:

```
X = np.array([1.0, 0.5])
W1 = np.array([[0.1, 0.3, 0.5], [0.2, 0.4, 0.6]])
B1 = np.array([0.1, 0.2, 0.3])
A1 = np.dot(X, W1) + B1
print("入力層の列数",X.shape)
print("中間層の入力層側行数、中間層の出力層側の列数",W1.shape)
print("出力層の行数",B1.shape)
print(A1)
```

```
入力層の列数 (2,)
中間層の入力層側行数、中間層の出力層側の列数 (2, 3)
出力層の行数 (3,)
[0.3 0.7 1.1]
```

層と層の間で要素数が異ならなければ、Xの要素数とW1の行数は合致することになるので行列の計算が可能。 つまり、ただ一度も途中でニューロンの数が変わらない前提となるので入力層の数＝計算結果の数となる。 この場合、入力層を増やさなければ、計算結果も増えないため、入力層の数だけの特徴量しか持つことができない。

出力層の設計

ソフトマックス関数

In [16]:

```
import numpy as np
def softmax(array):
    """一つの行列を0～1に直します。"""
    return np.exp(array) / np.sum(np.exp(array))
```

In [17]:

```
a1,a2,a3 = softmax([1,2,3])
print("a1,a2,a3=",a1,a2,a3)
print("SUM=",a1+a2+a3)
```

```
a1,a2,a3= 0.09003057317038046 0.24472847105479767 0.6652409557748219
SUM= 1.0
```

注意点

In [18]:

```
print(np.exp([1,2,100000])) #eの100000乗は64bit以上になるので、無限大と扱われてしまう。
```

```
[2.71828183 7.3890561         inf]

c:¥users¥4037554¥appdata¥local¥programs¥python¥python36-32¥lib¥site-packages¥ipyke
rnel_launcher.py:1: RuntimeWarning: overflow encountered in exp
  """Entry point for launching an IPython kernel.
```

In [19]:

```
import numpy as np
def softmax(array):
    """一つの行列を0～1に直します。inf簡易対策バージョン。"""
    array = array - np.max(array)
    return np.exp(array) / np.sum(np.exp(array))
```

In [20]:

```
a1,a2,a3 = softmax([1,2,3])
print("a1,a2,a3=",a1,a2,a3)
print("SUM=",a1+a2+a3)
```

```
a1,a2,a3= 0.09003057317038046 0.24472847105479764 0.6652409557748218
SUM= 0.9999999999999999
```

In [21]:

```
a1,a2,a3 = softmax([1,2,1000000])
print("a1,a2,a3=",a1,a2,a3)
print("SUM=",a1+a2+a3)
```

```
a1,a2,a3= 0.0 0.0 1.0
SUM= 1.0
```

ニューラルネットワークでは定数の大きさではなく影響力の大きさで計算するので、このオーバーフロー対策は不要

ニューラルネットワーク層に関する注意点

- 入力層のニューロン数＝テストケースをベクトル化した際の種類の数、ただし異常値やNullを除去したもの
- 中間層のニューロン数＝ハイパーパラメータ
- 出力層のニューロン数＝分類問題であるなら、分類されるべき種類の数、ただし確実にそれ以外の種類が出ないという保証があるもの

MNIST 0-9画像データセット

利用の前にmnist.pyの中身を参照して元々の画像データがどういう形式で、どういう形に変換されたかを確認すること。 例えば、訓練画像と全く同じデータがテスト画像の中に含まれていないかどうか、外れ値が無いかどうか、Nullが無いかどうか、など。（今回は入っていない） データのチェック及び加工処理がほとんどの工数を占める

In [22]:

```
import sys,os
sys.path.append(os.pardir) #mnist.pyを呼び出すための設定
from dataset.mnist import load_mnist

#flattenで行列ではなく配列にする
#normalizeで0～1に正規化する
(x_train, t_train),  (x_test, t_test) = load_mnist(flatten=True,normalize=False)
#1画素 = 0～256

print(x_train.shape) # 784画素 x 6万件の学習データ
print(t_train.shape) # 6万件の学習ラベルデータ
print(x_test.shape) # 784画素 x 1万件のテストデータ
print(t_test.shape) # 1万件のテストラベルデータ
```

```
(60000, 784)
(60000,)
(10000, 784)
(10000,)
```

- 今回は0～9の分類になるため、10個の出力層が必要となる
- 入力層は画素数になるので、784個
- 中間層は不明だが、10個以上でなければならない。

In [23]:

```
%matplotlib inline
#plt.show()をinlineにて表示する
#Esc+Lで行番号を表示する1
import numpy as np
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
from dataset.mnist import load_mnist
from PIL import Image

(x_train, t_train), (x_test, t_test) = load_mnist(flatten=True, normalize=False)
img = x_train[0]
label = t_train[0]
img = img.reshape(28, 28)
print("行、列",img.shape)
print("とりあえず画像の表示＝＞",label)
img_data = Image.fromarray(np.uint8(img))
img_data
```

```
行、列 (28, 28)
とりあえず画像の表示＝＞ 5
```

Out[23]:

In [24]:

```
# coding: utf-8
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
import pickle
from dataset.mnist import load_mnist
from common.functions import sigmoid, softmax

def init_network():
    """学習済の推論モデル(ただの重みとゲタの行列)を読み込む。"""
    with open("sample_weight.pkl", 'rb') as f:
        network = pickle.load(f)
    return network

network = init_network()

W1, W2, W3 = network['W1'], network['W2'], network['W3']
b1, b2, b3 = network['b1'], network['b2'], network['b3']
network = init_network()

def predict(network, x):
    a1 = np.dot(x, W1) + b1
    z1 = sigmoid(a1)
    a2 = np.dot(z1, W2) + b2
    z2 = sigmoid(a2)
    a3 = np.dot(z2, W3) + b3
    y = softmax(a3)
    return y

def get_data():
    """正規化済、配列化済のテストデータを読み込む"""
    (x_train, t_train), (x_test, t_test) = load_mnist(normalize=True, flatten=True, one_hot_labe
l=False)
    return x_test, t_test

x, t = get_data()
accuracy_cnt = 0
for i in range(len(x)):
    y = predict(network, x[i])
    p= np.argmax(y) # 最も確率の高い要素のインデックスを取得
    if p == t[i]:
        accuracy_cnt += 1
print("正解率:" + str(float(accuracy_cnt) / len(x)) + "(" + str(int(accuracy_cnt))+" /"+str(len(
x)) + ")")
```

正解率:0.9352(9352 /10000)

In [25]:

```
W1.shape
```

Out[25]:

(784, 50)

In [26]:

```
W2.shape
```

Out[26]:

(50, 100)

In [27]:

```
W3.shape
```

Out[27]:

(100, 10)

入力層：入力の数と一致している。行数は固定。 中間層：前の行列の列数と後の行列の行数が一致している。行数・列数は固定ではない。50と100の数字は適当に決めることもできるが大きいほど計算も増える。
出力層：出力の数と一致している。列数は固定。

バッチ処理

行列にスカラを掛けても同一の計算式になるため、入力層を100個にして出力層も100個にすることで同時計算することも可能

In [28]:

```
x, t = get_data()
network = init_network()

batch_size = 100
accuracy_cnt = 0
for i in range(0, len(x), batch_size):
    x_batch = x[i:i+batch_size]
    y_batch = predict(network, x_batch)
    p = np.argmax(y_batch, axis=1)
    accuracy_cnt += np.sum(p == t[i:i+batch_size])

print("正確さ：" + str(int(accuracy_cnt) / len(x)))
```

正確さ：0.9352

正確さに変動が無いことを確認すること。

ニューラルネットワークの学習

用語の説明

- データ駆動：人の意識なく集められたデータのみを信用する考え方
- 人駆動：人の意識によって集められた特徴量（ベクトル）、人の意識によって集められた分類法によって得られたデータを使用する考え方
- 教師データ：ニューラルネットワークでいう、学習データ。学習データはディープラーニングにおいてデータ駆動として扱う都合テストデータと一致してはならない。

過学習(overfitting)

ディープラーニングのおいて特定個人・個別のデータの特徴量のみ意識してしまい汎化能力に欠けてしまうこと。 ディープラーニングに置いては 学習データ≠テストデータ≠将来的なデータ でなければならない。

損失関数(loss function)

出力における、テスト結果における正解との誤差のこと。平均二乗誤差(mean squared error)と交差エントロピー誤差（cross entropy error）の二つが代表的にある。この損失関数の値が小さいほど正解に近いということになる。

平均二乗誤差

$E = \frac{1}{2}\Sigma_k((y_k - t_k)^2)$

In [1]:

```
import numpy as np
import pandas as pd
def mean_squared_error(y, t):
    return 0.5 * np.sum((y-t)**2)
a1,a2 = np.asarray([1,2,3]), np.asarray([1,2,4])
print(mean_squared_error(a1,a2))
a1,a2 = np.asarray([1,2,3]), np.asarray([1,2,10])
print( mean_squared_error(a1,a2) )
a1,a2 = np.asarray([1,2,3]), np.asarray([1,2,3])
print( mean_squared_error(a1,a2) )
```

```
0.5
24.5
0.0
```

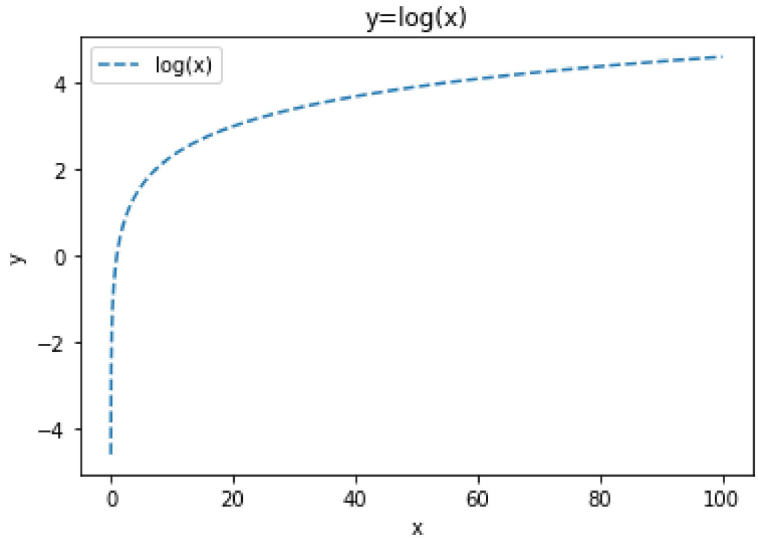
交差エントロピー誤差

$E = -\Sigma_k(p_k \log q_k)$
$(0 \leq p \leq 1, 0 \leq q \leq 1)$

log(x)の関数の特徴

In [2]:

```
%matplotlib inline
#plt.show()をinlineにて表示する
#Esc+Lで行番号を表示する
import matplotlib.pyplot as plt
import numpy as np

import numpy as np
def log(x):
    return np.log(x)
x = np.arange(0.01,100,0.01)
y = log(x)

plt.plot(x,y,label="log(x)", linestyle="--")

plt.xlabel("x")
plt.ylabel("y")

plt.title("y=log(x)")

plt.legend()#show label box

plt.show()
plt.close()
```



交差エントロピー誤差の関数

In [3]:

```
def softmax(array):
    return np.exp(array) / np.sum(np.exp(array))
def cross_entropy_error(y, t):
    delta = 1e-7 #Avoid log(0) => Inf
    return -np.sum(t * np.log(y + delta))
```

In [4]:

```
t = [0, 0, 1, 0, 0, 0, 0, 0, 0, 0]#One-Shot配列（正解＝1、不正解＝0に正規化したもの）
y = [0.1, 0.05, 0.6, 0.0, 0.05, 0.1, 0.0, 0.1, 0.0, 0.0]#SoftMax関数により正規化済
print("SUM t =",np.sum(np.array(t)))
print("SUM y =", np.sum(np.array(y)))
print("交差エントロピー誤差 = ",cross_entropy_error(np.array(y),np.array(t)))
t = [0, 0, 1, 0, 0, 0, 0, 0, 0, 0]#One-Shot配列（正解＝1、不正解＝0に正規化したもの）
y = [0.1, 0.05, 0.1, 0.0, 0.05, 0.1, 0.0, 0.6, 0.0, 0.0]#SoftMax関数により正規化済
print("SUM t =",np.sum(np.array(t)))
print("SUM y =", np.sum(np.array(y)))
print("交差エントロピー誤差 = ",cross_entropy_error(np.array(y),np.array(t)))
```

```
SUM t = 1
SUM y = 1.0
交差エントロピー誤差 =  0.510825457099338
SUM t = 1
SUM y = 1.0
交差エントロピー誤差 =  2.302584092994546
```

ミニバッチ学習（サンプリング）

損失関数を行列横断的に実施するためには、以下の関数を用いる

$$E = -\frac{1}{N}\Sigma_n \Sigma_k (t_{nk} \log y_{nk})$$

関数は難しいが単純に損失関数を各配列ごとにやって最後に足しているだけ。それぞれの配列の内容や要素数はもちろん異なるため、それぞれ正規化されていることが条件。最後に誤差を平均化するため配列数で割る。（これにより、それぞれのバッチのパーセプトロンの数（＝配列数）や次元数（＝配列の長さ）を変動させても互いに損失関数として比較可能とすることが出来る。）　→　膨大なデータの中から任意の数のデータのみをサンプリングして機械学習に流し込むことができるので、計算量を削減できる。しかも、サンプリングパーセンテージはその配列毎に変えることができる。（batch_size）

MNISTデータを使ってミニバッチ学習を実践

In [5]:

```
# MNISTデータのロード
import sys,os
sys.path.append(os.pardir) #mnist.pyを呼び出すための設定
from dataset.mnist import load_mnist

#flattenで行列ではなく配列にする
#normalizeで0～1に正規化する
(x_train, t_train), (x_test, t_test) = load_mnist(normalize=False, one_hot_label=True)
#1画素 = 0～256

print(x_train.shape) # 784画素 x 6万件の学習データ
print(t_train.shape) # 6万件の学習ラベルデータ
print(x_test.shape) # 784画素 x 1万件のテストデータ
print(t_test.shape) # 1万件のテストラベルデータ
```

```
(60000, 784)
(60000, 10)
(10000, 784)
(10000, 10)
```

In [6]:

```
#MNISTデータからランダムサンプリング
batch_size = 10
np.random.choice(x_train.shape[0], batch_size)#0～60000の配列のうち、どの配列を使うか選んで数字でインデックスを返す
```

Out[6]:

```
array([11357, 23726, 38963, 48357, 18302, 45106, 47393, 21141, 52581,
       30185])
```

ミニバッチ対応版（交差エントロピー誤差　損失関数の実装）

In [7]:

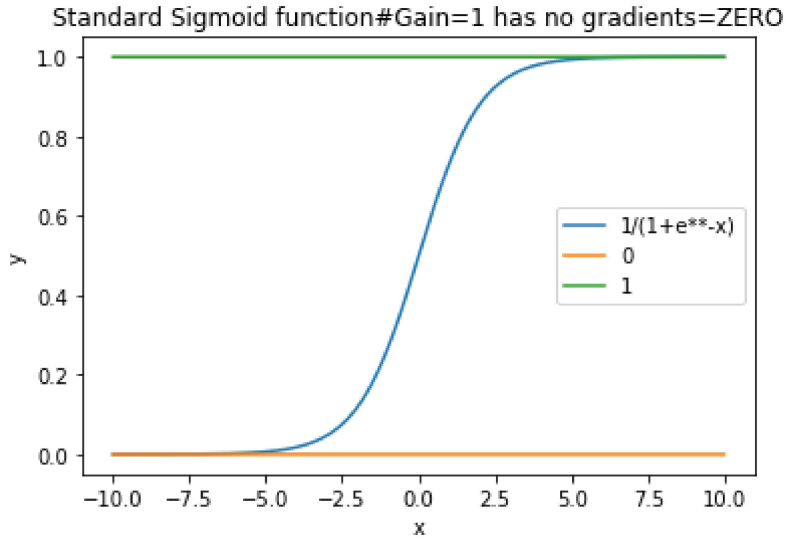
```
def cross_entropy_error(y, t):
    if y.ndim == 1:
        t = t.reshape(1,t.size)
        y = y.reshape(1, y.size)

    batch_size = y.shape[0]
    # One-Hot表現の場合、0 or 1なので、
    return -np.sum(np.log(y[np.arange(batch_size), t])) / batch_size
```

認識精度よりも損失度を認識度の指標とする理由

- 認識精度はどのレベルまで認識できれば良いのか上限が見えず、たまたま100%になる（微分し傾きが0になる場所が多く存在する）ことがありデータ表現として0～1の間で表せられないことが多い。（機械が扱うには少し弱いデータ）
- 誤差精度は滅多に100%になることがなく、ゆるやかに収束する（微分し傾きが0になる場所が滅多に存在しない）ため微細なパラメータ調整の結果得られる誤差移動率（＝微分した際の傾き）が算出可能となり0～1の間で表せられる。（機械が扱いやすい）

In [8]:

```
%matplotlib inline
#plt.show()をinlineにて表示する
#Esc+Lで行番号を表示する1
import matplotlib.pyplot as plt
import numpy as np

x1 = np.arange(-10,10,0.01) #-0 to 10 by 0.01
y1 = 1.0/(1.0 + np.e ** (-1.0*x1))

x2 = x1
y2 = x2*0

x3 = x1
y3 = x3*0+1

plt.plot(x1,y1,label="1/(1+e**-x)", linestyle="-")
plt.plot(x2,y2,label="0", linestyle="-")
plt.plot(x3,y3,label="1", linestyle="-")

plt.xlabel("x")
plt.ylabel("y")

plt.title("Standard Sigmoid function#Gain=1 has no gradients=ZERO")

plt.legend()#show label box

plt.show()
plt.close()
```



誤差の遷移に傾き（微分）を使う際の丸め誤差

In [9]:

```
np.float32(1e-50)
```

Out[9]:

0.0

微分方程式

$$f'(x) = \frac{d}{dx}f(x) = \lim_{\Delta x \to 0}\frac{f(x+\Delta x) - f(x)}{\Delta x}$$

数値微分プログラム

In [10]:

```
def numerical_diff(f, x):
    h = 10e-4
    return (f(x+h) - f(x)) / 2*h
```

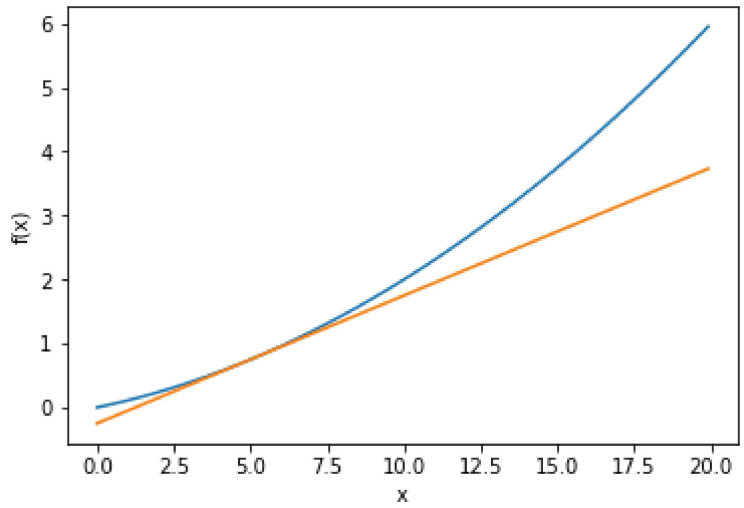
解析的な微分と数値微分

- 数値的微分： 最小の微動値(h)を+-し、結果の計算結果の差分を2で割る
- 解析的微分： 微分方程式を使用し変数を残した状態で算出する

In [11]:

```
# coding: utf-8
import numpy as np
import matplotlib.pylab as plt


def numerical_diff(f, x):
    h = 1e-4 # 0.0001
    return (f(x+h) - f(x-h)) / (2*h)


def function_1(x):
    return 0.01*x**2 + 0.1*x


def tangent_line(f, x):
    d = numerical_diff(f, x)
    print(d)
    y = f(x) - d*x
    return lambda t: d*t + y

x = np.arange(0.0, 20.0, 0.1)
y = function_1(x)
plt.xlabel("x")
plt.ylabel("f(x)")

tf = tangent_line(function_1, 5)
y2 = tf(x)

plt.plot(x, y)
plt.plot(x, y2)
plt.show()
#傾きが0.2になっていない！
```

0.1999999999990898



- 傾きが0.2になっていないが、接線にはなっているように見えることに着目する

偏微分

略（微分が対象軸（＝変数）ごとに実施できることと、複数の微分演算子が微分の順番を変えても最終的には成り立つということが示されていること）

勾配

行列式において、全ての値についての偏微分を求めてベクトル行列化したものを勾配と呼ぶ。

In [12]:

```
def numerical_gradient(f, x):
    """f関数におけるx配列の数値微分をし、それぞれの値の勾配（微分した際の傾きを返します）をxと同じ列数の配列で返します。"""
    h = 1e-4 # 0.0001（微動）
    grad = np.zeros_like(x) # 0 * xと同義

    for idx in range(x.size):

        #一旦退避。
        tmp_val = x[idx]

        #もしプラスの方向に微動した場合の結果は？
        x[idx] = float(tmp_val) + h
        fxh1 = f(x) # f(x+h)

        #もしマイナスの方向に微動した場合の結果は？
        x[idx] = tmp_val - h
        fxh2 = f(x) # f(x-h)

        #配列を数値微分
        grad[idx] = (fxh1 - fxh2) / (2*h)

        #傾きの計算が終わったので、一旦値を戻す
        x[idx] = tmp_val

    return grad
```

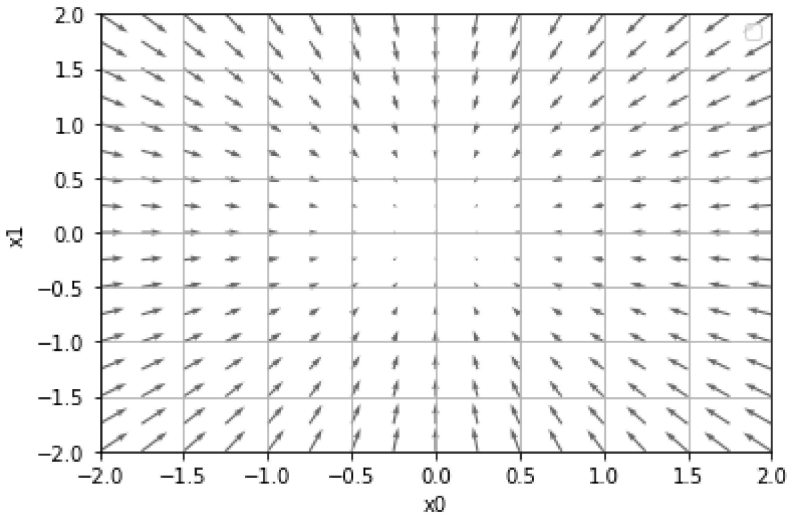
勾配降下のイメージ図（2次元関数の場合）→傾きが0に収束するためのベクトルがプロットされる

In [13]:

```
# coding: utf-8
# cf.http://d.hatena.ne.jp/white_wheels/20100327/p3
import numpy as np
import matplotlib.pylab as plt
from mpl_toolkits.mplot3d import Axes3D


def _numerical_gradient_no_batch(f, x):
    h = 1e-4 # 0.0001
    grad = np.zeros_like(x)

    for idx in range(x.size):
        tmp_val = x[idx]
        x[idx] = float(tmp_val) + h
        fxh1 = f(x) # f(x+h)

        x[idx] = tmp_val - h
        fxh2 = f(x) # f(x-h)
        grad[idx] = (fxh1 - fxh2) / (2*h)

        x[idx] = tmp_val # 値を元に戻す

    return grad


def numerical_gradient(f, X):
    if X.ndim == 1:
        return _numerical_gradient_no_batch(f, X)
    else:
        grad = np.zeros_like(X)

        for idx, x in enumerate(X):
            grad[idx] = _numerical_gradient_no_batch(f, x)

        return grad


def function_2(x):
    if x.ndim == 1:
        return np.sum(x**2)
    else:
        return np.sum(x**2, axis=1)


def tangent_line(f, x):
    d = numerical_gradient(f, x)
    print(d)
    y = f(x) - d*x
    return lambda t: d*t + y

if __name__ == '__main__':
    x0 = np.arange(-2, 2.5, 0.25)
    x1 = np.arange(-2, 2.5, 0.25)
    X, Y = np.meshgrid(x0, x1)

    X = X.flatten()
    Y = Y.flatten()

    grad = numerical_gradient(function_2, np.array([X, Y]) )
```

```
    plt.figure()
    plt.quiver(X, Y, -grad[0], -grad[1],  angles="xy",color="#666666")#,headwidth=10,scale=40,color="#444444")
    plt.xlim([-2, 2])
    plt.ylim([-2, 2])
    plt.xlabel('x0')
    plt.ylabel('x1')
    plt.grid()
    plt.legend()
    plt.draw()
    plt.show()
```

No handles with labels found to put in legend.



関数が収束する方向にベクトルが向いていることに注意。

勾配法

関数の傾きが0となる点を目指して関数の値を徐々に収束させる手法。 今回の場合は損失関数を0に収束させるため、勾配降下法と呼ぶ。

勾配法

$$x_1 = x_0 - \eta\frac{\partial f}{\partial x_0},\ \ x_2 = x_1 - \eta\frac{\partial f}{\partial x_1}, \ldots$$
$$\eta = learning\,rate(e.g.,\ 0.1,\ 0.001, \ldots)$$

勾配法プログラム

In [14]:

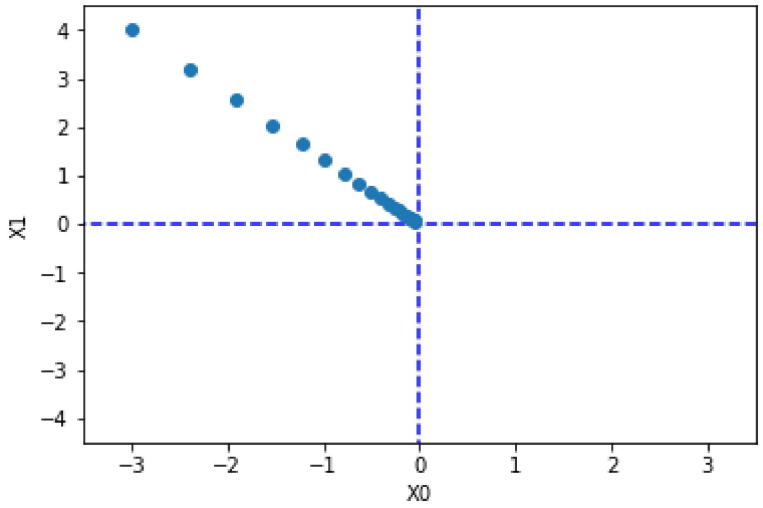
```
import numpy as np
import matplotlib.pylab as plt
from gradient_2d import numerical_gradient

def gradient_descent(f, init_x, lr=0.01, step_num=100):
    """f : 最適化が必要な関数, init_xは初期値, lrは学習率, step_numは勾配降下回数. fの傾きが最小となるような"""
    x = init_x
    x_history = []

    for i in range(step_num):
        x_history.append( x.copy() )

        grad = numerical_gradient(f, x)
        x -= lr * grad

    return x,np.array(x_history)
```

$$f(x_0, x_1) = x_0^2 + x_1^2$$

の最小値を勾配法プログラムによって求める。

In [15]:

```
def function_2(x):
    return x[0]**2 + x[1]**2

init_x = np.array([-3.0, 4.0])

lr = 0.1
step_num = 20
x, x_history = gradient_descent(function_2, init_x, lr=lr, step_num=step_num)

plt.plot( [-5, 5], [0,0], '--b')
plt.plot( [0,0], [-5, 5], '--b')
plt.plot(x_history[:,0], x_history[:,1], 'o')

plt.xlim(-3.5, 3.5)
plt.ylim(-4.5, 4.5)
plt.xlabel("X0")
plt.ylabel("X1")
plt.show()
```



$(kf_x) = kf_x$
$(f \pm g)_x = f_x \pm g_x$
$(f \cdot g)_x = f_x \cdot g + f \cdot g_x$
$\left(\frac{f}{g}\right)_x = \frac{f_x \cdot g - f \cdot g_x}{g^2}$
$\frac{\partial f}{\partial x} = \frac{df}{du}\frac{\partial u}{\partial x}$
$\frac{\partial f}{\partial y} = \frac{df}{du}\frac{\partial u}{\partial y}$

ニューラルネットワーク（行列式）における勾配降下プログラム

In [16]:

```
# coding: utf-8
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 common.functions import softmax, cross_entropy_error
from common.gradient import numerical_gradient


class simpleNet:
    def __init__(self):
        self.W = np.random.randn(2,3)#2行x3列の行列として作成
        #ガウス分布で初期化

    def predict(self, x):
        """内積をとる（自身のW（重み）を積算）"""
        return np.dot(x, self.W)

    def loss(self, x, t):
        """x = 入力データ、 t = テストデータ"""
        z = self.predict(x)
        y = softmax(z)#0～1の値へと汎化
        loss = cross_entropy_error(y, t) #tはone-shot行列とする（正解=1、不正解=0）

        return loss

x = np.array([0.6, 0.9])# 1行2列
t = np.array([0, 0, 1]) #1行3列

net = simpleNet()
f = lambda w: net.loss(x, t)
dW = numerical_gradient(f, net.W)

print("W=",net.W)
print("dW=",dW)
print("WをdWを下げるようにベクトルを向ける必要がある")
```

```
W= [[-0.29987897 -2.12317372  0.47065595]
 [-0.39304798 -0.98756266 -0.75852701]]
dW= [[ 0.25654033  0.05031211 -0.30685244]
 [ 0.38481049  0.07546816 -0.46027865]]
WをdWを下げるようにベクトルを向ける必要がある
```

学習アルゴリズムの実装

1. ミニバッチ（訓練データサンプリング）
    - 訓練データの中からランダムに一部のデータを選びだす。その選ばれたデータをミニバッチと言い、ここでは、そのミニバッチの損失関数の値を減らすことを目的とする。（ミニバッチは無作為サンプリングの手法のため、**確率的勾配降下法(stochastic gradient descent, SGD)**と呼ばれる）
2. 勾配算出
    - ミニバッチの損失関数を減らすために、各重みのパラメータの勾配を求める。勾配は損失関数の値をもっとも減らす方向を示す。
3. パラメータの更新
    - 重みパラメータを勾配方向に微小量だけ更新する。
4. 繰り返し
    - 1 ~ 3 の繰り返し

In [17]:

```
# coding: utf-8
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
from common.functions import *
from common.gradient import numerical_gradient


class TwoLayerNet:
    """
    2層ニューラルネットワークの実装
    """
    def __init__(self, input_size, hidden_size, output_size, weight_init_std=0.01):
        """input_size = 入力の数(MNISTの場合、28 x 28の画素=784), hidden_size = 隠れ層の数(ハイパーパラメータ), output_size = 出力の数(MNISTの場合、0～9の数字=10), weight_init_std = 重みの標準化引数(?)"""
        # 重みの初期化
        self.params = {}
        self.params['W1'] = weight_init_std * np.random.randn(input_size, hidden_size) #ガウス分布による重みパラメータの初期化
        self.params['b1'] = np.zeros(hidden_size) #一様分布によるバイアスパラメータの初期化
        self.params['W2'] = weight_init_std * np.random.randn(hidden_size, output_size) #ガウス分布による重みパラメータの初期化
        self.params['b2'] = np.zeros(output_size) #一様分布によるバイアスパラメータの初期化

    def predict(self, x):
        """入力に対する出力を確率配列として返す"""
        W1, W2 = self.params['W1'], self.params['W2']
        b1, b2 = self.params['b1'], self.params['b2']

        a1 = np.dot(x, W1) + b1
        z1 = sigmoid(a1)
        a2 = np.dot(z1, W2) + b2
        y = softmax(a2)

        return y

    # x:入力データ, t:教師データ
    def loss(self, x, t):
        """誤差を交差エントロピー誤差関数によって返す（y =確率配列 , t =one-hot配列）"""
        y = self.predict(x)
        return cross_entropy_error(y, t)

    def accuracy(self, x, t):
        """正確度を返す。x = 入力データ、t = テストデータ、accuracy = **%"""
        y = self.predict(x)
        y = np.argmax(y, axis=1)
        t = np.argmax(t, axis=1)

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

    # x:入力データ, t:教師データ
    def numerical_gradient(self, x, t):
        loss_W = lambda W: self.loss(x, t)

        grads = {}
        grads['W1'] = numerical_gradient(loss_W, self.params['W1'])
        grads['b1'] = numerical_gradient(loss_W, self.params['b1'])
        grads['W2'] = numerical_gradient(loss_W, self.params['W2'])
```

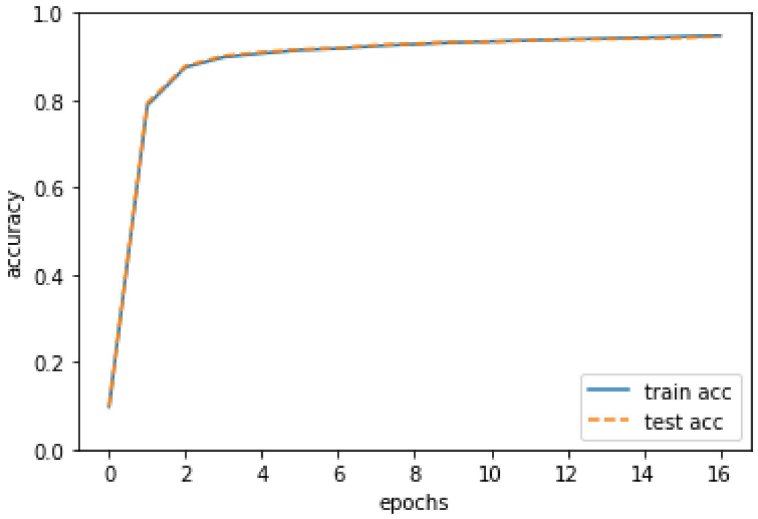
```
        grads['b2'] = numerical_gradient(loss_W, self.params['b2'])

        return grads

    def gradient(self, x, t):
        W1, W2 = self.params['W1'], self.params['W2']
        b1, b2 = self.params['b1'], self.params['b2']
        grads = {}

        batch_num = x.shape[0]

        # forward
        a1 = np.dot(x, W1) + b1
        z1 = sigmoid(a1)
        a2 = np.dot(z1, W2) + b2
        y = softmax(a2)

        # backward
        dy = (y - t) / batch_num
        grads['W2'] = np.dot(z1.T, dy)
        grads['b2'] = np.sum(dy, axis=0)

        da1 = np.dot(dy, W2.T)
        dz1 = sigmoid_grad(a1) * da1
        grads['W1'] = np.dot(x.T, dz1)
        grads['b1'] = np.sum(dz1, axis=0)

        return grads
```

2層ニューラルネットワークを使った学習

In [18]:

```
# coding: utf-8
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
import matplotlib.pyplot as plt
from dataset.mnist import load_mnist
from two_layer_net import TwoLayerNet

# データの読み込み
(x_train, t_train), (x_test, t_test) = load_mnist(normalize=True, one_hot_label=True)

network = TwoLayerNet(input_size=784, hidden_size=50, output_size=10)

iters_num = 10000  # 繰り返しの回数を適宜設定する
train_size = x_train.shape[0]
batch_size = 100
learning_rate = 0.1

train_loss_list = []
train_acc_list = []
test_acc_list = []

iter_per_epoch = max(train_size / batch_size, 1)

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

    # 勾配の計算
    #grad = network.numerical_gradient(x_batch, t_batch)
    grad = network.gradient(x_batch, t_batch)

    # パラメータの更新
    for key in ('W1', 'b1', 'W2', 'b2'):
        network.params[key] -= learning_rate * grad[key]

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

    if i % iter_per_epoch == 0:
        train_acc = network.accuracy(x_train, t_train)
        test_acc = network.accuracy(x_test, t_test)
        train_acc_list.append(train_acc)
        test_acc_list.append(test_acc)
        print("train acc, test acc | " + str(train_acc) + ", " + str(test_acc))

# グラフの描画
markers = {'train': 'o', 'test': 's'}
x = np.arange(len(train_acc_list))
plt.plot(x, train_acc_list, label='train acc')
plt.plot(x, test_acc_list, label='test acc', linestyle='--')
plt.xlabel("epochs")
plt.ylabel("accuracy")
plt.ylim(0, 1.0)
plt.legend(loc='lower right')
plt.show()
```

```
train acc, test acc | 0.09736666666666667, 0.0982
train acc, test acc | 0.7876666666666666, 0.7933
train acc, test acc | 0.8752166666666666, 0.8781
train acc, test acc | 0.89835, 0.9014
train acc, test acc | 0.9069333333333334, 0.9097
train acc, test acc | 0.91465, 0.916
train acc, test acc | 0.91805, 0.919
train acc, test acc | 0.92405, 0.926
train acc, test acc | 0.9279833333333334, 0.9289
train acc, test acc | 0.9319166666666666, 0.9331
train acc, test acc | 0.93435, 0.9334
train acc, test acc | 0.9368833333333333, 0.9367
train acc, test acc | 0.9393833333333333, 0.9385
train acc, test acc | 0.9413, 0.9398
train acc, test acc | 0.9433666666666667, 0.9411
train acc, test acc | 0.9456833333333333, 0.9425
train acc, test acc | 0.9476, 0.9463
```



学習データにおける正確さとテストデータにおける正確さが概ね合致しているので、「過学習」は起きていないと思われる！

誤差逆伝播法

バックプロパゲーション,Backpropagation (https://ja.wikipedia.org/wiki/%E3%83%90%E3%83%83%E3%82%AF%E3%83%97%E3%83%AD%E3%83%9
のこと。ニューロン単位で損失関数計算し「局所誤差」を求め、ニューロンに対する入力の内、より大きな重みで接続された前段のニューロンに対して、局所誤差の責任があると判定する。連鎖律により、層全体のニューロンに対する入力を集積させてその入力を呼び起こしたニューロンを特定し、重みの傾斜を再計算することで責任を逆伝播させる。

連鎖律

連鎖律 (https://ja.wikipedia.org/wiki/%E9%80%A3%E9%8E%96%E5%BE%8B)とは微分法において以下の性質が成り立つ式のこと。 合成関数の導関数の積が元の構成関数の導関数と等しくなる。

$$\frac{dz}{dx} = \frac{dz}{dt}\frac{dt}{dx}$$

計算グラフに置ける誤差逆伝播

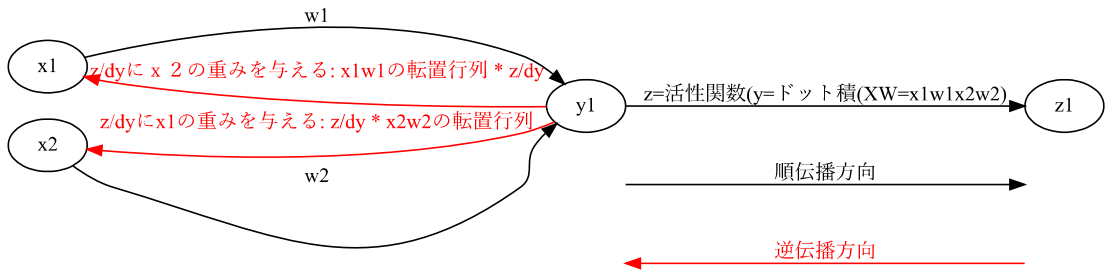
誤差逆伝播法のための計算グラフまとめ (http://marumaru.tonkotsu.jp/%E8%AA%A4%E5%B7%AE%E9%80%86%E4%BC%9D%E6%92%AD%E6%B3
より

ドット積の逆伝播

In [1]:

```
from graphviz import Digraph
dot = Digraph(comment="積の逆伝播")
dot.attr(rankdir="LR")
#dot.attr(splines="line") #line or curved or ortho or polyline;
dot.attr(fixedsize="true")
dot.attr(label="ドット積の逆伝播（単純に転置行列をしかるべき方向から掛け合わせて元の行列に戻すだけ）")
with dot.subgraph(name="main") as main:
    with main.subgraph(name="cluster_x") as x:
        x.attr(label="")
        x.attr(color="white")
        x.node("x1","x1")
        x.node("x2","x2")
    with main.subgraph(name="cluster_y") as y:
        y.attr(label="")
        y.attr(color="white")
        y.node("y0","y0", color="white", fontcolor="white")
        y.node("y1","y1")
        y.node("y2","y2", color="white", fontcolor="white")
    dot.edge("x1", "y1", label="w1")
    dot.edge("x2",  "y1", label="w2")
    with main.subgraph(name="cluster_z") as z:
        z.attr(label="")
        z.attr(color="white")
        z.node("z0","z0", color="white", fontcolor="white")
        z.node("z1","z1")
        z.node("z2","z2", color="white", fontcolor="white")
    main.edge("y0",  "z0", label="順伝播方向")
    main.edge("z2",  "y2", label="逆伝播方向", color="red", fontcolor="red")
    main.edge("y1",  "z1", label="z=活性関数(y=ドット積(XW=x1w1x2w2)")
    main.edge("y1",  "x1", label="z/dyにｘ２の重みを与える: x1w1の転置行列 * z/dy", color="red",
 fontcolor="red")
    main.edge("y1",  "x2", label="z/dyにx1の重みを与える: z/dy * x2w2の転置行列", color="red", f
ontcolor="red")

#print(dot)
dot
```

Out[1]:



ドット積の逆伝播（単純に転置行列をしかるべき方向から掛け合わせて元の行列に戻すだけ）

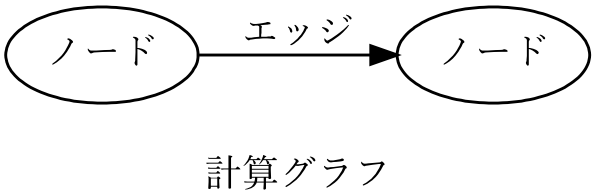
計算グラフの復習

以下のようなグラフで表されたもの

In [2]:

```
from graphviz import Digraph
dot = Digraph(comment="計算グラフ")
dot.attr(rankdir="LR")
#dot.attr(splines="line") #line or curved or ortho or polyline;
dot.attr(fixedsize="true")
dot.attr(label="計算グラフ")
with dot.subgraph(name="main") as main:
    with main.subgraph(name="cluster_x") as x:
        x.attr(label="")
        x.attr(color="white")
        x.node("x1","ノード")
    with main.subgraph(name="cluster_y") as y:
        y.attr(label="")
        y.attr(color="white")
        y.node("y1","ノード")
    dot.edge("x1", "y1", label="エッジ")
#print(dot)
dot
```

Out[2]:



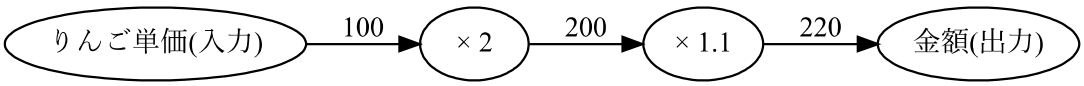
計算グラフ

問1.太郎くんはスーパーで一個１００円のりんごを２個買いました。支払う金額を求めなさい。ただし、消費税が10%適用されるものとします。

In [3]:

```
from graphviz import Digraph
dot = Digraph(comment="計算グラフ")
dot.attr(rankdir="LR")
#dot.attr(splines="line") #line or curved or ortho or polyline;
dot.attr(fixedsize="true")
dot.attr(label="エッジを計算結果、ノードを計算式とした計算グラフ")
with dot.subgraph(name="main") as main:
    with main.subgraph(name="cluster_x") as x:
        x.attr(label="")
        x.attr(color="white")
        x.node("x1","りんご単価(入力)")
    with main.subgraph(name="cluster_y") as y:
        y.attr(label="")
        y.attr(color="white")
        y.node("y1","× 2")
    with main.subgraph(name="cluster_y") as z:
        z.attr(label="")
        z.attr(color="white")
        z.node("z1","× 1.1")
    with main.subgraph(name="cluster_y") as out:
        out.attr(label="")
        out.attr(color="white")
        out.node("out1","金額(出力)")
    dot.edge("x1", "y1", label="100")
    dot.edge("y1", "z1", label="200")
    dot.edge("z1", "out1", label="220")
#print(dot)
dot
```

Out[3]:

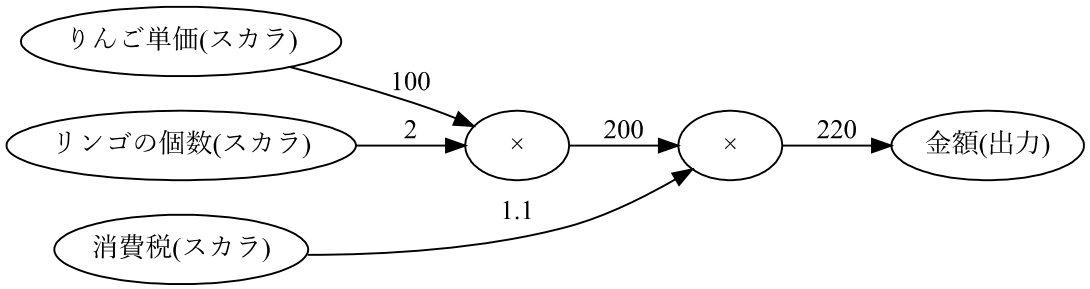


エッジを計算結果、ノードを計算式とした計算グラフ

In [4]:

```
from graphviz import Digraph
dot = Digraph(comment="計算グラフ")
dot.attr(rankdir="LR")
#dot.attr(splines="") #line or curved or ortho or polyline;
dot.attr(fixedsize="true")
dot.attr(label="ノードをfunctionとした計算グラフ")
with dot.subgraph(name="main") as main:
    with main.subgraph(name="cluster_x") as x:
        x.attr(label="")
        x.attr(color="white")
        x.node("x1","りんご単価(スカラ)")
        x.node("x2","リンゴの個数(スカラ)")
        x.node("x3","消費税(スカラ)")
    with main.subgraph(name="cluster_y") as y:
        y.attr(label="")
        y.attr(color="white")
        y.node("y1","×")
    with main.subgraph(name="cluster_y") as z:
        z.attr(label="")
        z.attr(color="white")
        z.node("z1","×")
    with main.subgraph(name="cluster_y") as out:
        out.attr(label="")
        out.attr(color="white")
        out.node("out1","金額(出力)")
    dot.edge("x1", "y1", label="100")
    dot.edge("x2", "y1", label="2")
    dot.edge("x3", "z1", label="1.1")
    dot.edge("y1", "z1", label="200")
    dot.edge("z1", "out1", label="220")
#print(dot)
dot
```

Out[4]:



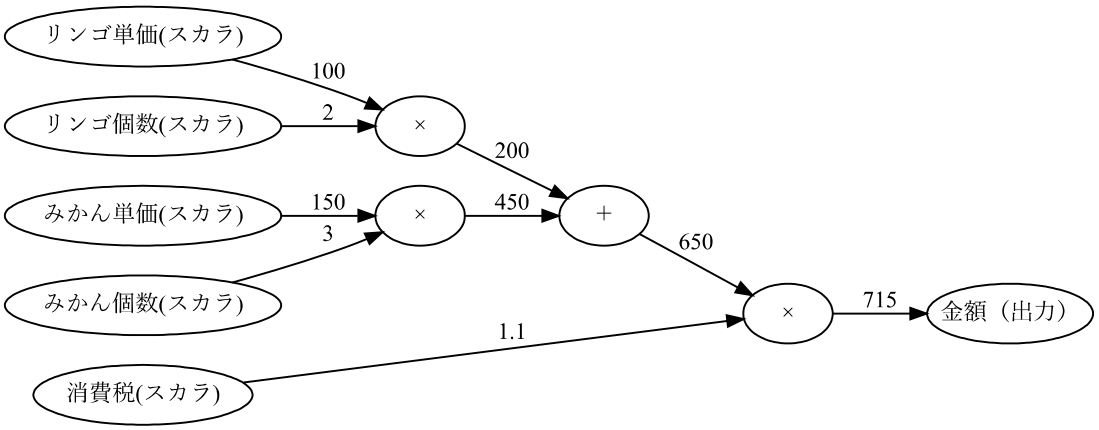
ノードをfunctionとした計算グラフ

問2. 太郎くんはスーパーで一個１００円のりんごを２個、1個150円のみかんを3個買いました。支払う金額を求めなさい。ただし、消費税が10%適用されるものとします。

In [5]:

```
from graphviz import Digraph
dot = Digraph(comment="計算グラフ")
dot.attr(rankdir="LR")
#dot.attr(splines="") #line or curved or ortho or polyline;
dot.attr(fixedsize="true")
dot.attr(label="積算ノードと加算ノードをあわせた計算グラフ")
with dot.subgraph(name="main") as main:
    with main.subgraph(name="cluster_x") as x:
        x.attr(label="")
        x.attr(color="white")
        x.node("x1","リンゴ単価(スカラ)")
        x.node("x2","リンゴ個数(スカラ)")
        x.node("x3","みかん単価(スカラ)")
        x.node("x4","みかん個数(スカラ)")
        x.node("x5","消費税(スカラ)")
    with main.subgraph(name="cluster_y") as y:
        y.attr(label="")
        y.attr(color="white")
        y.node("y1","×")
        y.node("y2","×")
    with main.subgraph(name="cluster_z") as z:
        z.attr(label="")
        z.attr(color="white")
        z.node("z1","＋")
    with main.subgraph(name="cluster_a") as a:
        a.attr(label="")
        a.attr(color="white")
        a.node("a1","×")
    with main.subgraph(name="cluster_out") as out:
        out.attr(label="")
        out.attr(color="white")
        out.node("out1","金額（出力）")
    dot.edge("x1", "y1", label="100")
    dot.edge("x2", "y1", label="2")
    dot.edge("x3", "y2", label="150")
    dot.edge("x4", "y2", label="3")
    dot.edge("y1", "z1", label="200")
    dot.edge("y2", "z1", label="450")
    dot.edge("z1", "a1", label="650")
    dot.edge("x5", "a1", label="1.1")
    dot.edge("a1", "out1", label="715")
#print(dot)
dot
```

Out[5]:



積算ノードと加算ノードをあわせた計算グラフ

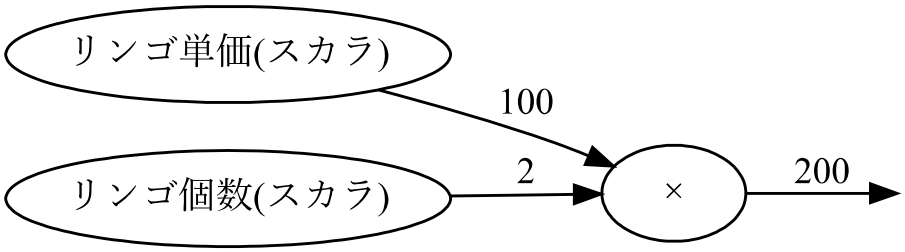
局所的な計算

一部のみを抜粋しても計算が可能であるということが、一つの計算ノードの再利用可能性を示している。

In [6]:

```
from graphviz import Digraph
dot = Digraph(comment="計算グラフ")
dot.attr(rankdir="LR")
#dot.attr(splines="") #line or curved or ortho or polyline;
dot.attr(fixedsize="true")
dot.attr(label="積算ノードと加算ノードをあわせた計算グラフ（局地的な計算）")
with dot.subgraph(name="main") as main:
    with main.subgraph(name="cluster_x") as x:
        x.attr(label="")
        x.attr(color="white")
        x.node("x1","リンゴ単価(スカラ)")
        x.node("x2","リンゴ個数(スカラ)")
    with main.subgraph(name="cluster_y") as y:
        y.attr(label="")
        y.attr(color="white")
        y.node("y1","×")
    with main.subgraph(name="cluster_z") as z:
        z.attr(label="")
        z.attr(color="white")
        z.node("z1","", color="white")
    dot.edge("x1", "y1", label="100")
    dot.edge("x2", "y1", label="2")
    dot.edge("y1", "z1", label="200")
#print(dot)
dot
```

Out[6]:



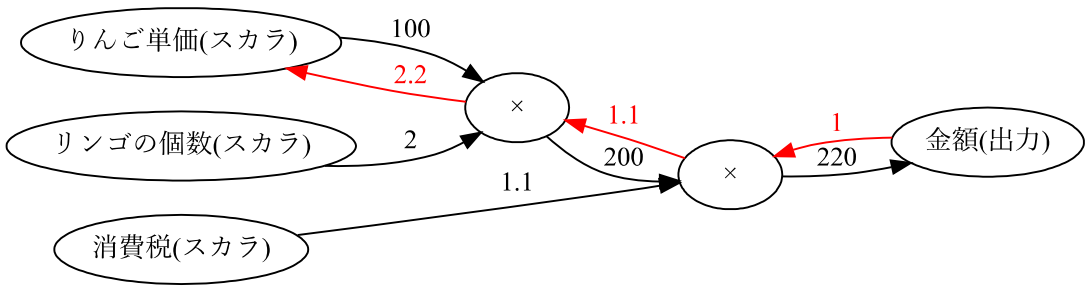
積算ノードと加算ノードをあわせた計算グラフ（局地的な計算）

逆伝搬の例

In [7]:

```
from graphviz import Digraph
dot = Digraph(comment="計算グラフ")
dot.attr(rankdir="LR")
#dot.attr(splines="") #line or curved or ortho or polyline;
dot.attr(fixedsize="true")
dot.attr(label="ノードをfunctionとした計算グラフ")
with dot.subgraph(name="main") as main:
    with main.subgraph(name="cluster_x") as x:
        x.attr(label="")
        x.attr(color="white")
        x.node("x1","りんご単価(スカラ)")
        x.node("x2","リンゴの個数(スカラ)")
        x.node("x3","消費税(スカラ)")
    with main.subgraph(name="cluster_y") as y:
        y.attr(label="")
        y.attr(color="white")
        y.node("y1","×")
    with main.subgraph(name="cluster_y") as z:
        z.attr(label="")
        z.attr(color="white")
        z.node("z1","×")
    with main.subgraph(name="cluster_y") as out:
        out.attr(label="")
        out.attr(color="white")
        out.node("out1","金額(出力)")
    #Forward propagation
    dot.edge("x1", "y1", label="100")
    dot.edge("x2", "y1", label="2")
    dot.edge("x3", "z1", label="1.1")
    dot.edge("y1", "z1", label="200")
    dot.edge("z1", "out1", label="220")

    #Back propagation
    dot.edge("out1", "z1", label="1", color="red", fontcolor="red")
    dot.edge("z1", "y1", label="1.1", color="red", fontcolor="red")
    dot.edge("y1", "x1", label="2.2", color="red", fontcolor="red")

#print(dot)
dot
```

Out[7]:



ノードをfunctionとした計算グラフ

りんごが「１」微動した場合の価格の微動は「2.2」であることが示されている。

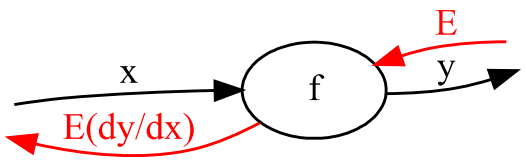
連鎖律

計算グラフの逆伝搬

In [8]:

```
from graphviz import Digraph
dot = Digraph(comment="計算グラフ")
dot.attr(rankdir="LR")
#dot.attr(splines="line") #line or curved or ortho or polyline;
dot.attr(fixedsize="true")
dot.attr(label="計算グラフの逆伝搬")
with dot.subgraph(name="main") as main:
    with main.subgraph(name="cluster_x") as x:
        x.attr(label="")
        x.attr(color="white")
        x.node("x1","", color="white")
    with main.subgraph(name="cluster_y") as y:
        y.attr(label="")
        y.attr(color="white")
        y.node("y1","f")
    with main.subgraph(name="cluster_y") as z:
        z.attr(label="")
        z.attr(color="white")
        z.node("z1","", color="white")
    dot.edge("x1", "y1", label="x")
    dot.edge("y1", "z1", label="y")
    dot.edge("z1", "y1", label="E", color="red", fontcolor="red")
    dot.edge("y1", "x1", label="E(dy/dx)", color="red", fontcolor="red")
#print(dot)
dot
```

Out[8]:



計算グラフの逆伝搬

- x: 下流から得たスカラ値
- y: $f(x)$
- E: 上流から得たスカラ値
- dy/dx: 関数fの導関数

連鎖律とは

ある関数が合成関数で表すことができる場合、その**合成関数の微分は、合成関数を構成するそれぞれの関数の微分の積によって表すことができる。**これを**連鎖律の原理**と呼ぶ。各ノードの微分式が逆伝搬の際に元の式の有効性を保ちつつそのまま伝達できることを示している。

$y = (x + y)^2$ の微分の例

$y = (x + y)^2$ は以下のような変数 z と t によって表せる。

式1. $z = t^2$
式2. $t = x + y$

式1,式2を合成関数と呼ぶ。

$y = (x + y)^2$ の導関数は$\frac{\partial y}{\partial x}$
$z = t^2$ の導関数は$\frac{\partial z}{\partial t}$

であるので、連鎖律によりこの合成関数の導関数は

$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial t}\frac{\partial t}{\partial x}$ となる。

∂tは積算で互いに打ち消し合い、最終的に元の式の導関数に戻ることを確認する。

合成関数の微分の計算

$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial t}\frac{\partial t}{\partial x}$ を計算する。

- $\frac{\partial z}{\partial t} = 2t$ (通常の解析微分)
- $\frac{\partial t}{\partial x} = 1$

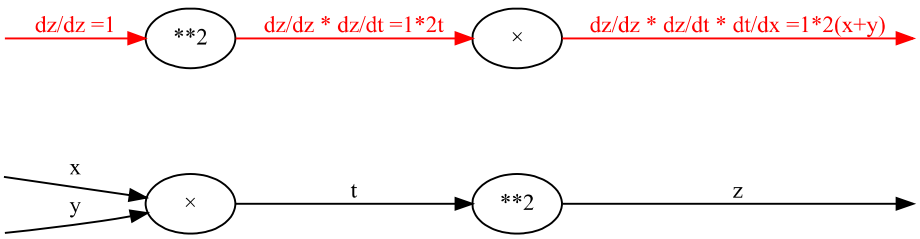
tを展開すると $2(x + y)$。この結果は$y = (x + y)^2$ の導関数: $\frac{\partial y}{\partial x}$ と一致することを確認する。

$y = (x + y)^2$ を計算グラフで示す

In [9]:

```
from graphviz import Digraph
dot = Digraph(comment="計算グラフ")
dot.attr(rankdir="LR")
#dot.attr(ranksep="1.2")
dot.attr(nodesep=".05")
#dot.attr(margin="1")
#dot.attr(splines="polyline") #line or curved or ortho or polyline;
dot.attr(fixedsize="true")
dot.attr(label="y = x^2の計算グラフと逆伝搬計算グラフ（赤字）")
with dot.subgraph(name="main") as main:
    with main.subgraph(name="cluster_L1") as L1:
        L1.attr(label="")
        L1.attr(color="white")
        L1.node("main.L1N1", "", color="white")
        L1.node("main.L1N2", "", color="white")
    with main.subgraph(name="cluster_L2") as L2:
        L2.attr(label="")
        L2.attr(color="white")
        L2.node("main.L2N1","×")
    with main.subgraph(name="cluster_L3") as L3:
        L3.attr(label="")
        L3.attr(color="white")
        L3.node("main.L3N1", "**2")
    with main.subgraph(name="cluster_a") as L4:
        L4.attr(label="")
        L4.attr(color="white")
        L4.node("main.L4N1", "", color="white")
    #Forward Propagation
    dot.edge("main.L1N1", "main.L2N1", label="x")
    dot.edge("main.L1N2", "main.L2N1", label="y")
    dot.edge("main.L2N1", "main.L3N1", label="t")
    dot.edge("main.L3N1", "main.L4N1", label="z")
with dot.subgraph(name="sub") as sub:
    with sub.subgraph(name="cluster_L1") as L1:
        L1.attr(label="")
        L1.attr(color="white")
        L1.node("sub.L1N1", "", color="white")
        L1.node("sub.L1N2", "", color="white")
    with sub.subgraph(name="cluster_L2") as L2:
        L2.attr(label="")
        L2.attr(color="white")
        L2.node("sub.L2N1","×")
    with sub.subgraph(name="cluster_L3") as L3:
        L3.attr(label="")
        L3.attr(color="white")
        L3.node("sub.L3N1", "**2")
    with sub.subgraph(name="cluster_a") as L4:
        L4.attr(label="")
        L4.attr(color="white")
        L4.node("sub.L4N1", "", color="white")
    #Back Propagation
    dot.edge("sub.L4N1", "sub.L3N1", label="dz/dz =1", color="red", fontcolor="red")
    dot.edge("sub.L3N1", "sub.L2N1", label="dz/dz * dz/dt =1*2t", color="red", fontcolor="red")
    dot.edge("sub.L2N1", "sub.L1N1", label="dz/dz * dz/dt * dt/dx =1*2(x+y)", color="red", fontc
olor="red")
    #dot.edge("L2N1", "L1N2", label="", color="red", fontcolor="red")
#print(dot)
dot
```

Out[9]:



y = x^2の計算グラフと逆伝搬計算グラフ（赤字）

あらゆる式は2入力1出力の計算グラフに直すことが出来る

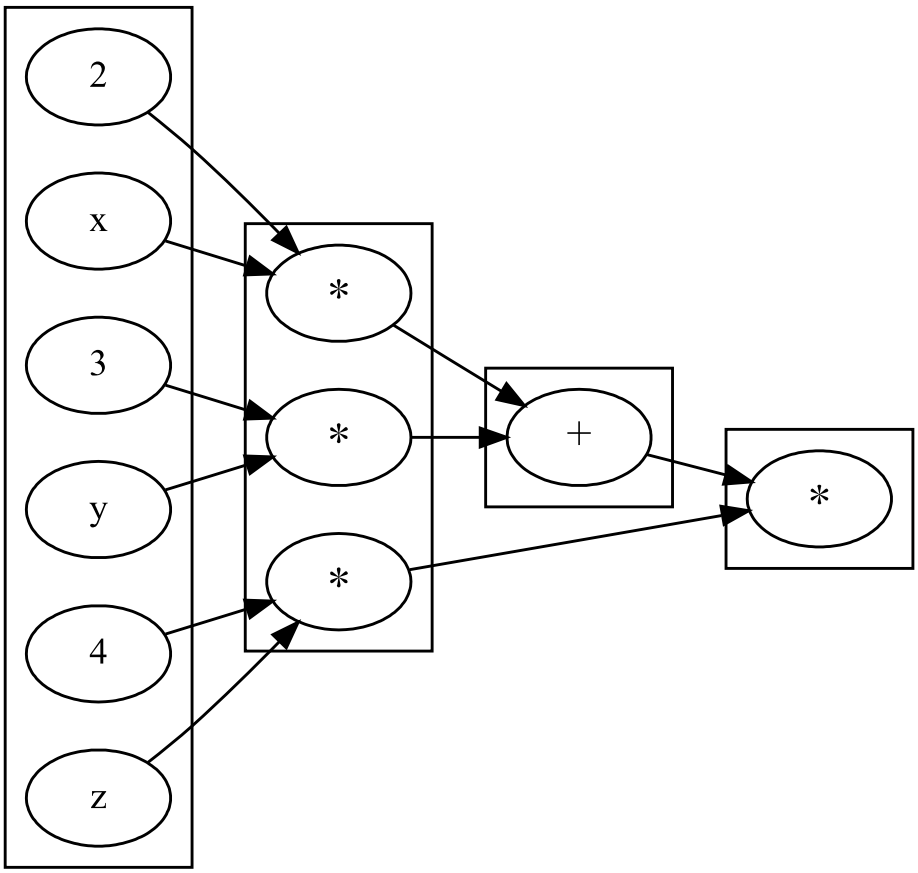
例

$f(x, y, z) = 2x + 3y * 4z$

In [10]:

```
from graphviz import Digraph
dot = Digraph(comment="計算グラフ")
dot.attr(rankdir="LR")
#dot.attr(splines="") #line or curved or ortho or polyline;
dot.attr(fixedsize="true")
dot.attr(label="f(x,y,z) = 2x + 3y * 4zの計算グラフ")
with dot.subgraph(name="main") as main:
    with main.subgraph(name="cluster_L1") as L1:
        L1.attr(label="")
        L1.node("L1N1", "2")
        L1.node("L1N2", "x")
        L1.node("L1N3", "3")
        L1.node("L1N4", "y")
        L1.node("L1N5", "4")
        L1.node("L1N6", "z")
    with main.subgraph(name="cluster_L2") as L2:
        L2.attr(label="")
        L2.node("L2N1","*")
        L2.node("L2N2","*")
        L2.node("L2N3","*")
    with main.subgraph(name="cluster_L3") as L3:
        L3.attr(label="")
        L3.node("L3N1", "+")
    with main.subgraph(name="cluster_L4") as L4:
        L4.attr(label="")
        L4.node("L4N1", "*")
    dot.edge("L1N1", "L2N1", label="")
    dot.edge("L1N2", "L2N1", label="")
    dot.edge("L1N3", "L2N2", label="")
    dot.edge("L1N4", "L2N2", label="")
    dot.edge("L1N5", "L2N3", label="")
    dot.edge("L1N6", "L2N3", label="")

    dot.edge("L2N1", "L3N1", label="")
    dot.edge("L2N2", "L3N1", label="")

    dot.edge("L3N1", "L4N1", label="")
    dot.edge("L2N3", "L4N1", label="")

    #print(dot)
dot
```

Out[10]:



f(x,y,z) = 2x + 3y * 4zの計算グラフ

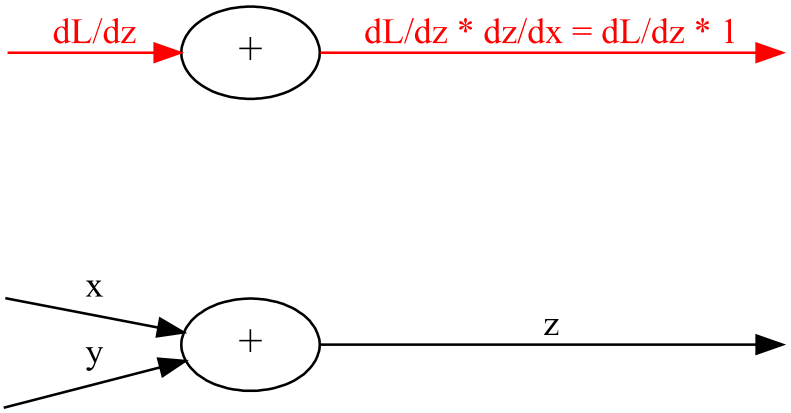
片方の成分での微分を（双方に対する誤差として）逆伝搬させることで、**より誤差を大きく出した片方に責任を負わせることが可能となる**ことが分かる。 これが計算グラフによって示されている。

加算ノードの逆伝搬

In [11]:

```
from graphviz import Digraph
dot = Digraph(comment="計算グラフ")
dot.attr(rankdir="LR")
#dot.attr(splines="") #line or curved or ortho or polyline;
dot.attr(fixedsize="true")
dot.attr(label="加算ノード計算グラフ")
with dot.subgraph(name="main") as main:
    with main.subgraph(name="cluster_L1") as L1:
        L1.attr(label="")
        L1.attr(color="white")
        L1.node("main.L1N1", "", color="white")
        L1.node("main.L1N2", "", color="white")
    with main.subgraph(name="cluster_L2") as L2:
        L2.attr(label="")
        L2.attr(color="white")
        L2.node("main.L2N1","＋")
    with main.subgraph(name="cluster_L3") as L3:
        L3.attr(label="")
        L3.attr(color="white")
        L3.node("main.L3N1", "", color="white")
    #Forward Propagation
    dot.edge("main.L1N1", "main.L2N1", label="x")
    dot.edge("main.L1N2", "main.L2N1", label="y")
    dot.edge("main.L2N1", "main.L3N1", label="z")
with dot.subgraph(name="sub") as sub:
    with sub.subgraph(name="cluster_L1") as L1:
        L1.attr(label="")
        L1.attr(color="white")
        L1.node("sub.L1N1", "", color="white")
        L1.node("sub.L1N2", "", color="white")
    with sub.subgraph(name="cluster_L2") as L2:
        L2.attr(label="")
        L2.attr(color="white")
        L2.node("sub.L2N1","＋")
    with sub.subgraph(name="cluster_L3") as L3:
        L3.attr(label="")
        L3.attr(color="white")
        L3.node("sub.L3N1", "", color="white")
    #Back Propagation
    dot.edge("sub.L3N1", "sub.L2N1", label="dL/dz", color="red", fontcolor="red")
    dot.edge("sub.L2N1", "sub.L1N1", label="dL/dz * dz/dx = dL/dz * 1", color="red", fontcolor=
"red")
#print(dot)
dot
```

Out[11]:



加算ノード計算グラフ

In [12]:

```
class AddLayer:
    """
    加算ノード
    """
    def __init__(self):
        pass

    def forward(self, x, y):
        """
        順伝播
        """
        out = x + y
        return out

    def backward(self, dout):
        """
        逆伝播：微分
        """
        dx = dout * 1
        dy = dout * 1
        return dx, dy
```

- $z = x + y$
  - $\frac{\partial z}{\partial x} = 1$
  - $\frac{\partial z}{\partial y} = 1$

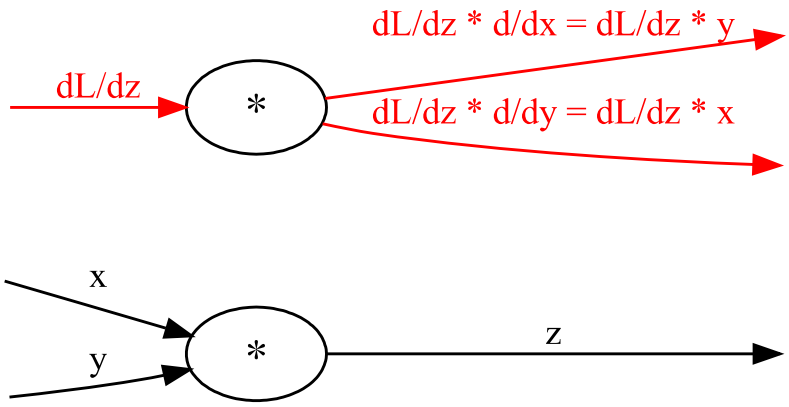
よって、加算ノードの微分逆伝搬は上流からの伝搬Lをそのまま下流に伝達させるのみ。

乗算ノードの逆伝搬

In [13]:

```
from graphviz import Digraph
dot = Digraph(comment="計算グラフ")
dot.attr(rankdir="LR")
#dot.attr(splines="") #line or curved or ortho or polyline;
dot.attr(fixedsize="true")
dot.attr(label="乗算ノード計算グラフ")
with dot.subgraph(name="main") as main:
    with main.subgraph(name="cluster_L1") as L1:
        L1.attr(label="")
        L1.attr(color="white")
        L1.node("main.L1N1", "", color="white")
        L1.node("main.L1N2", "", color="white")
    with main.subgraph(name="cluster_L2") as L2:
        L2.attr(label="")
        L2.attr(color="white")
        L2.node("main.L2N1","*")
    with main.subgraph(name="cluster_L3") as L3:
        L3.attr(label="")
        L3.attr(color="white")
        L3.node("main.L3N1", "", color="white")
    #Forward Propagation
    dot.edge("main.L1N1", "main.L2N1", label="x")
    dot.edge("main.L1N2", "main.L2N1", label="y")
    dot.edge("main.L2N1", "main.L3N1", label="z")
with dot.subgraph(name="sub") as sub:
    with sub.subgraph(name="cluster_L1") as L1:
        L1.attr(label="")
        L1.attr(color="white")
        L1.node("sub.L1N1", "", color="white")
        L1.node("sub.L1N2", "", color="white")
    with sub.subgraph(name="cluster_L2") as L2:
        L2.attr(label="")
        L2.attr(color="white")
        L2.node("sub.L2N1","*")
    with sub.subgraph(name="cluster_L3") as L3:
        L3.attr(label="")
        L3.attr(color="white")
        L3.node("sub.L3N1", "", color="white")
    #Back Propagation
    dot.edge("sub.L3N1", "sub.L2N1", label="dL/dz", color="red", fontcolor="red")
    dot.edge("sub.L2N1", "sub.L1N1", label="dL/dz * d/dx = dL/dz * y", color="red", fontcolor="r
ed")
    dot.edge("sub.L2N1", "sub.L1N2", label="dL/dz * d/dy = dL/dz * x", color="red", fontcolor="r
ed")
#print(dot)
dot
```

Out[13]:



乗算ノード計算グラフ

In [14]:

```
class MulLayer:
    """
    乗算ノード
    """
    def __init__(self):
        self.x = None
        self.y = None

    def forward(self, x, y):
        """
        順伝播
        """
        self.x = x
        self.y = y
        out = x * y
        return out

    def backward(self, dout):
        """
        逆伝播：微分
        """
        dx = dout * self.y
        dy = dout * self.x
        return dx, dy
```

- $z = xy$
  - $\frac{\partial z}{\partial x} = y$
  - $\frac{\partial z}{\partial y} = x$

よって、乗算ノードの微分逆伝搬は上流からの伝搬Lを入力変数で微分して下流に伝達させる。（伝搬方向がひっくり返る）

ReLUノード（ReLUレイヤ）

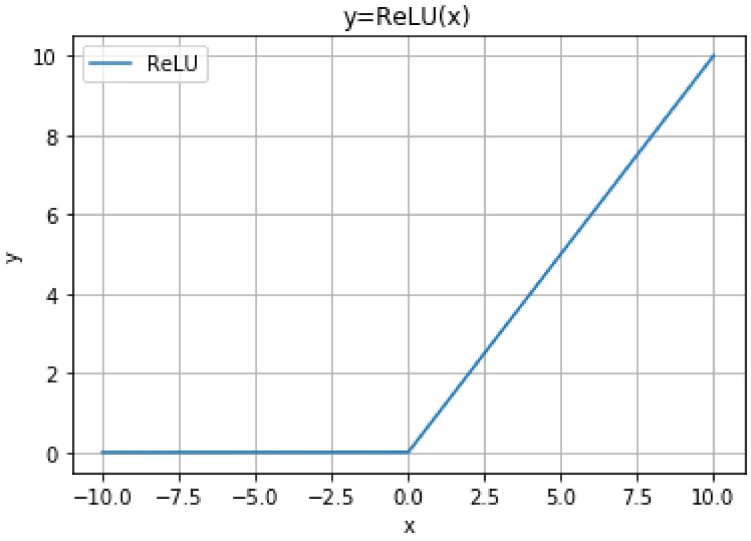
$$Forwarding\ \ f(x) = \begin{cases} x & (x > 0) \\ 0 & (x \leq 0) \end{cases}$$

$$BackProp\ \ f'(x) = \begin{cases} 1 & (x > 0) \\ 0 & (x \leq 0) \end{cases}$$

In [15]:

```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
x = np.arange(-10,10,0.001) #-0 to 10 by 0.001
y = x * (x > 0)
plt.plot(x,y,label="ReLU", linestyle="-")
plt.xlabel("x")
plt.ylabel("y")
plt.title("y=ReLU(x)")
plt.legend()
plt.grid()
plt.show()
plt.close()
```

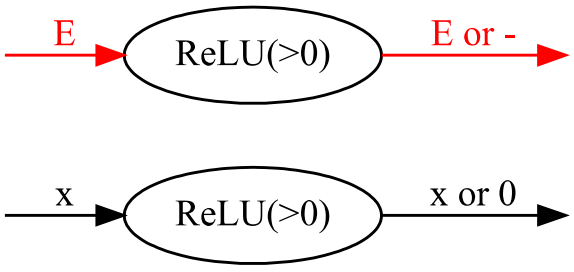
```
/home/nbuser/anaconda3_420/lib/python3.5/site-packages/matplotlib/font_manager.py:
281: UserWarning: Matplotlib is building the font cache using fc-list. This may ta
ke a moment.
  'Matplotlib is building the font cache using fc-list. '
```



In [16]:

```
from graphviz import Digraph
dot = Digraph(comment="計算グラフ")
dot.attr(rankdir="LR")
#dot.attr(splines="") #line or curved or ortho or polyline;
dot.attr(fixedsize="true")
dot.attr(label="ReLUノード計算グラフ")
with dot.subgraph(name="main") as main:
    with main.subgraph(name="cluster_L1") as L1:
        L1.attr(label="")
        L1.attr(color="white")
        L1.node("main.L1N1", "", color="white")
    with main.subgraph(name="cluster_L2") as L2:
        L2.attr(label="")
        L2.attr(color="white")
        L2.node("main.L2N1","ReLU(>0)")
    with main.subgraph(name="cluster_L3") as L3:
        L3.attr(label="")
        L3.attr(color="white")
        L3.node("main.L3N1", "", color="white")
    #Forward Propagation
    dot.edge("main.L1N1", "main.L2N1", label="x")
    dot.edge("main.L2N1", "main.L3N1", label="x or 0")
with dot.subgraph(name="sub") as sub:
    with sub.subgraph(name="cluster_L1") as L1:
        L1.attr(label="")
        L1.attr(color="white")
        L1.node("sub.L1N1", "", color="white")
    with sub.subgraph(name="cluster_L2") as L2:
        L2.attr(label="")
        L2.attr(color="white")
        L2.node("sub.L2N1","ReLU(>0)")
    with sub.subgraph(name="cluster_L3") as L3:
        L3.attr(label="")
        L3.attr(color="white")
        L3.node("sub.L3N1", "", color="white")
    #Back Propagation
    dot.edge("sub.L3N1", "sub.L2N1", label="E", color="red", fontcolor="red")
    dot.edge("sub.L2N1", "sub.L1N1", label="E or -", color="red", fontcolor="red")
dot
```

Out[16]:



ReLUノード計算グラフ

ReLUノードにおける逆伝搬をすることで「発火しなかった細胞はそこで伝搬をストップする」という作用を与えることが可能

In [17]:

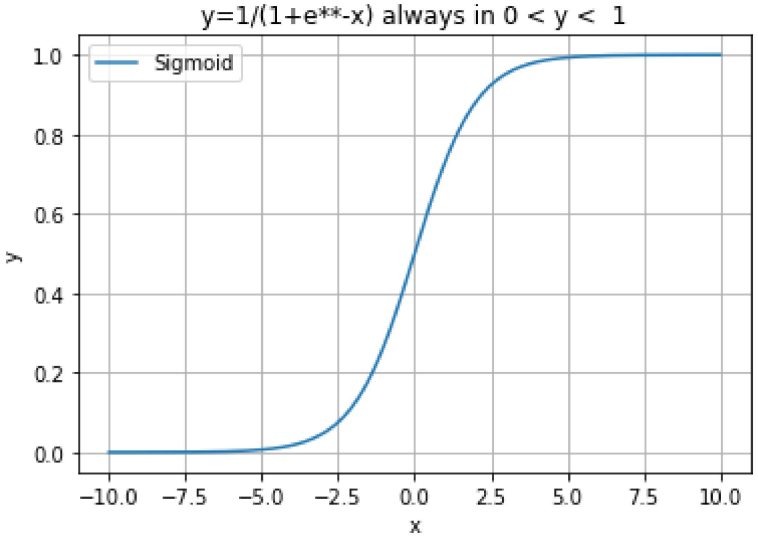
```
class Relu:
    """
    ReLUの実装
    """
    def __init__(self):
        self.mask = None

    def forward(self, x):
        """
        type x: numpy.array
        """
        self.mask = (x <= 0)
        out = x.copy()
        out[self.mask] = 0
        return out

    def backward(self, dout):
        """
        type dout: numpy.array
        """
        dout[self.mask] = 0
        dx = dout
        return dx
```

Sigmoidノード（Sigmoidレイヤ）

順伝播　$f(x) = \frac{1}{1+e^{(-x)}}$　　逆伝播　$f'(x) = f(x)(1 - f(x))$　　→ $1 \geq f(x), f'(x) \geq 0$

In [18]:

```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
x = np.arange(-10,10,0.001) #-0 to 10 by 0.001
y = 1.0/(1.0 + np.e ** (-1.0*x))
plt.plot(x,y,label="Sigmoid", linestyle="-")
plt.xlabel("x")
plt.ylabel("y")
plt.title("y=1/(1+e**-x) always in 0 < y <  1")
plt.legend()#show label box
plt.grid()
plt.show()
plt.close()
```



In [19]:

```
# coding: utf-8
import sys, os
sys.path.append(os.pardir)

from common.functions import sigmoid

class Sigmoid:
    """
    シグモイド関数の実装
    """
    def __init__(self):
        self.out = None

    def forward(self, x):
        out = sigmoid(x)
        self.out = out
        return out

    def backward(self, dout):
        dx = dout * (1.0 - self.out) * self.out

        return dx
```

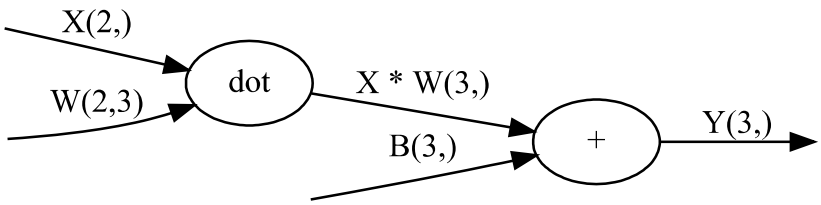
Affineノード（Affineレイヤ）

行列の内積を求めることを幾何学の分野でアフィン変換と呼ぶ。
スカラが複数同時に行列として計算できるということが以下に示される。
ただし、行列の次元数は合わせる必要がある。

In [20]:

```
from graphviz import Digraph
dot = Digraph(comment="計算グラフ")
dot.attr(rankdir="LR")
#dot.attr(splines="") #line or curved or ortho or polyline;
dot.attr(fixedsize="true")
dot.attr(label="Affine変換とバイアス加算の計算グラフ※括弧内は（行数, 列数）")
with dot.subgraph(name="main") as main:
    main.attr(color="white")
    with main.subgraph(name="cluster_L1") as L1:
        L1.attr(label="")
        L1.node("L1N1", "", color="white")
        L1.node("L1N2", "", color="white")
    with main.subgraph(name="cluster_L2") as L2:
        L2.attr(label="")
        L2.node("L2N1", "dot")
        L2.node("L2N2", "", color="white")
    with main.subgraph(name="cluster_L3") as L3:
        L3.attr(label="")
        L3.node("L3N1", "+")
    with main.subgraph(name="cluster_L4") as L4:
        L4.attr(label="")
        L4.node("L4N1", "", color="white")
    dot.edge("L1N1", "L2N1", label="X(2,)")
    dot.edge("L1N2", "L2N1", label="W(2,3)")
    dot.edge("L2N1", "L3N1", label="X * W(3,)")
    dot.edge("L2N2", "L3N1", label="B(3,)")
    dot.edge("L3N1", "L4N1", label="Y(3,)")
dot
```

Out[20]:



Affine変換とバイアス加算の計算グラフ※括弧内は（行数,列数）

Affine変換は単純にこれまでの算術ノードの計算を一つ一つの行列の要素毎に実行しているにすぎない

In [21]:

```
class Affine:
    def __init__(self, W, b):
        self.W =W
        self.b = b

        self.x = None
        self.original_x_shape = None
        # 重み・バイアスパラメータの微分
        self.dW = None
        self.db = None

    def forward(self, x):
        # テンソル対応
        self.original_x_shape = x.shape
        x = x.reshape(x.shape[0], -1)
        self.x = x

        out = np.dot(self.x, self.W) + self.b

        return out

    def backward(self, dout):
        dx = np.dot(dout, self.W.T)
        self.dW = np.dot(self.x.T, dout)
        self.db = np.sum(dout, axis=0)

        dx = dx.reshape(*self.original_x_shape)  # 入力データの形状に戻す（テンソル対応）
        return dx
```

SoftMax-WithLossノード（Softmaxレイヤ）

複数のスコアが適用された出力を正規化し、後のレイヤにおいてそのデータを正規的に使用する際に必要なノード。損失関数として、クロスエントロピー誤差計算関数を使っているので-WithLossというサフィックスが付いている。正規化が必要ない（スコアのたった一つの最頻値や最大値・最小値・平均値を使うような機械学習）の場合はこのノードは必要なし。ディープラーニングの際はスコアをそのまま入力として使うことをせず、一度確率的な数値(1>$p_n\left(\sum p_n == 1\right)$>0)に直してから使用するためこのノードが必要となるケースがある。**出力層で使われる際、もし分類問題の場合は入力データの数==出力データの数==分類数となっている必要がある。**

In [22]:

```
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))

def cross_entropy_error(y, t):
    if y.ndim == 1:
        t = t.reshape(1, t.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 t.size == y.size:
        t = t.argmax(axis=1)

    batch_size = y.shape[0]
    return -np.sum(np.log(y[np.arange(batch_size), t] + 1e-7)) / batch_size

def softmax_loss(X, t):
    y = softmax(X)
    return cross_entropy_error(y, t)
```

In [23]:

```
softmax(np.array([[0,1],[2,3]]))
```

Out[23]:

```
array([[0.26894142, 0.73105858],
       [0.26894142, 0.73105858]])
```

In [24]:

```
class SoftmaxWithLoss:
    def __init__(self):
        self.loss = None
        self.y = None # softmaxの出力
        self.t = None # 教師データ

    def forward(self, x, t):
        self.t = t
        self.y = softmax(x)
        self.loss = cross_entropy_error(self.y, self.t)

        return self.loss

    def backward(self, dout=1):
        batch_size = self.t.shape[0]
        if self.t.size == self.y.size: # 教師データがone-hot-vectorの場合
            dx = (self.y - self.t) / batch_size
        else:
            dx = self.y.copy()
            dx[np.arange(batch_size), self.t] -= 1
            dx = dx / batch_size

        return dx
```

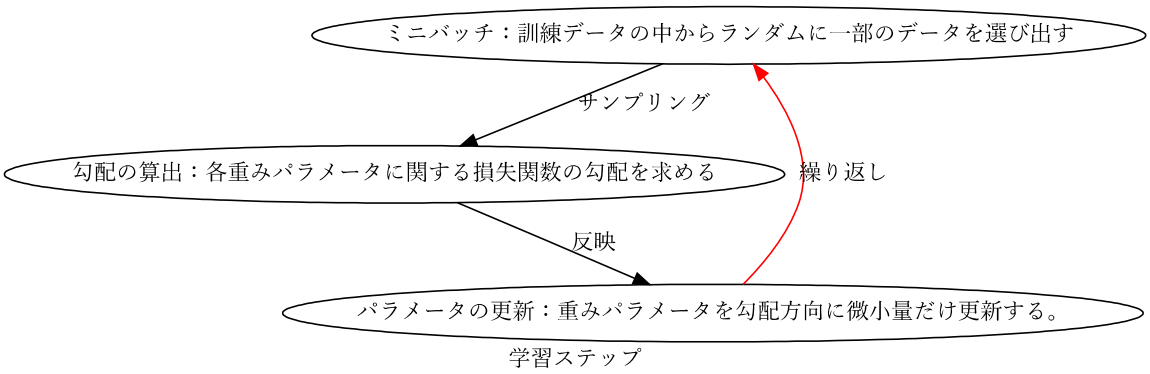
誤差逆伝搬法

ニューラルネットワークの学習ステップ

In [25]:

```
from graphviz import Digraph
dot = Digraph(comment="学習ステップ")
dot.attr(rankdir="UD")
#dot.attr(splines="") #line or curved or ortho or polyline;
dot.attr(fixedsize="true")
dot.attr(label="学習ステップ")
with dot.subgraph(name="main") as main:
    dot.node("MiniBatch","ミニバッチ：訓練データの中からランダムに一部のデータを選び出す")
    dot.node("CalcGrad","勾配の算出：各重みパラメータに関する損失関数の勾配を求める")
    dot.node("ReParam","パラメータの更新：重みパラメータを勾配方向に微小量だけ更新する。")
    dot.edge("MiniBatch", "CalcGrad", label="サンプリング")
    dot.edge("CalcGrad", "ReParam", label="反映")
    dot.edge("ReParam", "MiniBatch", label="繰り返し",color="red")
dot
```

Out[25]:



TwoLayerNet （誤差逆伝搬法（非：数値微分法）を追加バージョン）の実装

In [26]:

```
# coding: utf-8
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 common.layers import *
from common.gradient import numerical_gradient
from collections import OrderedDict


class TwoLayerNet:
    """
    このクラス自身は特に処理をしません。
    """
    def __init__(self, input_size, hidden_size, output_size, weight_init_std = 0.01):
        """
        各種レイヤを作成し、レイヤのハイパーパラメータを初期化します。
        ガウス分布などの分布の適合線に関する情報: https://support.minitab.com/ja-jp/minitab/18/help-and-how-to/graphs/supporting-topics/exploring-data-and-revising-graphs/distributions-for-fitted-lines/
        分布のスケールパラメータに関する情報： https://en.wikipedia.org/wiki/Scale_parameter
        Args:
            input_size (int): 入力層のニューロン数（入力データ要素数）
            hidden_size (int): 隠れ層のニューロン数
            output_size (int): 出力層のニューロン数（出力データ要素数）
            weight_init_std (int): 重み初期化時のガウス分布（正規分布）のスケールパラメタ＝分布の幅
        Returns:
            None
        """
        self.params = {}
        #randnで"正規分布"でランダムな行列（input_size列、hidden_size行）を生成し
        #それをweight_init_stdのスケールに狭めて重みデータの初期値を決める
        #バイアスはhidden_size行数だけ必要になるのでゼロで初期化
        self.params['W1'] = weight_init_std * np.random.randn(input_size, hidden_size)
        self.params['b1'] = np.zeros(hidden_size)
        #隠れ層についても同様に実施
        self.params['W2'] = weight_init_std * np.random.randn(hidden_size, output_size)
        self.params['b2'] = np.zeros(output_size)
        #それらの重み・バイアスの変数を持った各々の各レイヤを作成。
        self.layers = OrderedDict()
        #Affine1→Relu1→Affine2
        self.layers['Affine1'] = Affine(self.params['W1'], self.params['b1'])
        self.layers['Relu1'] = Relu()
        self.layers['Affine2'] = Affine(self.params['W2'], self.params['b2'])

        #出力層（誤差計算層）を初期化
        self.lastLayer = SoftmaxWithLoss()

    def predict(self, x):
        """
        OrderedDictを順番に取り出し順伝搬させる
        各レイヤの持っている重みとバイアスが使われる。
        Args:
            x (int): 計算対象の行列（入力行列）
        Returns:
            x (int): 計算結果（出力行列）
        """
        for layer in self.layers.values():
            x = layer.forward(x)
```

```
        return x

    # x:入力データ, t:教師データ
    def loss(self, x, t):
        """
        OrderedDictを順番に取り出し順伝搬させた後
        学習データと教師データを
        Args:
            x (int): 入力データ
            t (int): 教師データ
        Returns:
            x (int): 出力データ
        """
        y = self.predict(x)
        return self.lastLayer.forward(y, t)

    def accuracy(self, x, t):
        """
        Args:
            x (int): 入力データ
            t (int): 教師データ
        Returns:
            accuracy (float): 認識精度
        """
        y = self.predict(x)
        y = np.argmax(y, axis=1)
        if t.ndim != 1 : t = np.argmax(t, axis=1)

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

    # x:入力データ, t:教師データ
    def numerical_gradient(self, x, t):
        """
        Args:
            x (int): 入力データ
            t (int): 教師データ
        Returns:
            grads (dict): 数理的微分法を用いて計算された傾き及びバイアス
        """
        loss_W = lambda W: self.loss(x, t)

        grads = {}
        grads['W1'] = numerical_gradient(loss_W, self.params['W1'])
        grads['b1'] = numerical_gradient(loss_W, self.params['b1'])
        grads['W2'] = numerical_gradient(loss_W, self.params['W2'])
        grads['b2'] = numerical_gradient(loss_W, self.params['b2'])

        return grads

    def gradient(self, x, t):
        """
        Args:
            x (int): 入力データ
            t (int): 教師データ
        Returns:
            grads (dict): 解析的微分法を用いて計算された傾き及びバイアス
        """
        # forward
        self.loss(x, t)
```

```
        # backward
        dout = 1
        dout = self.lastLayer.backward(dout)

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

        # 設定
        grads = {}
        grads['W1'], grads['b1'] = self.layers['Affine1'].dW, self.layers['Affine1'].db
        grads['W2'], grads['b2'] = self.layers['Affine2'].dW, self.layers['Affine2'].db

        return grads
```

誤差逆伝搬法の微分方式の内、解析的微分法の勾配が正しく計算されていることを確認するには数値的微分法の計算結果と照らし合わせると良い。

誤差逆伝播法の学習

In [27]:

```
# coding: utf-8
import sys, os
sys.path.append(os.pardir)

import numpy as np
from dataset.mnist import load_mnist
from two_layer_net import TwoLayerNet

# データの読み込み ★学習データとテストデータは同じものを使ってはならない
(x_train, t_train), (x_test, t_test) = load_mnist(normalize=True, one_hot_label=True)
#784個の入力データ：縦28画素*横28画素
#50個の想定される構成因子数
#10個の出力データ：0～9の数字
network = TwoLayerNet(input_size=784, hidden_size=50, output_size=10)

iters_num = 10000 #学習回数
train_size = x_train.shape[0] #学習データ数
batch_size = 100 #バッチサイズ（100個サンプリングして学習）
learning_rate = 0.1 #学習率

train_loss_list = [] #途中経過保存用
train_acc_list = [] #途中経過保存用
test_acc_list = [] #途中経過保存用

iter_per_epoch = max(train_size / batch_size, 1) #バッチ回数

print("学習回数：", iters_num, "学習データ数：", train_size, "バッチサイズ：", batch_size, "学習率：", learning_rate)

for i in range(iters_num):
    #学習データからバッチ用のデータをランダムに抽出（一様分布）
    batch_mask = np.random.choice(train_size, batch_size)
    x_batch = x_train[batch_mask]
    t_batch = t_train[batch_mask]
    # 勾配
    #grad = network.numerical_gradient(x_batch, t_batch)
    grad = network.gradient(x_batch, t_batch)
    # 更新
    for key in ('W1', 'b1', 'W2', 'b2'):
        network.params[key] -= learning_rate * grad[key]
        # 差分 * 学習率 を減算し、ニューラルネットワークの重みとバイアスの値を更新
    loss = network.loss(x_batch, t_batch)
    train_loss_list.append(loss)

    #以下、経過出力用コード
    if i % iter_per_epoch == 0:
        train_acc = network.accuracy(x_train, t_train)
        test_acc = network.accuracy(x_test, t_test)
        train_acc_list.append(train_acc)
        test_acc_list.append(test_acc)
        print("★%d個サンプル学習データを%d回学習★" % (batch_size, i), "学習率：", train_acc, "認識率：", test_acc, "勾配差分：", loss)
```

学習回数： 10000 学習データ数： 60000 バッチサイズ： 100 学習率： 0.1
★100個サンプル学習データを0回学習★ 学習率： 0.059866666666666665 認識率： 0.0604 勾配差分： 2.3029724580090503
★100個サンプル学習データを600回学習★ 学習率： 0.9049333333333334 認識率： 0.9079 勾配差分： 0.25196564925931403
★100個サンプル学習データを1200回学習★ 学習率： 0.9239333333333334 認識率： 0.9243 勾配差分： 0.33200771267981016
★100個サンプル学習データを1800回学習★ 学習率： 0.93385 認識率： 0.9338 勾配差分： 0.2631430208924852
★100個サンプル学習データを2400回学習★ 学習率： 0.9451666666666667 認識率： 0.9437 勾配差分： 0.27544850688906797
★100個サンプル学習データを3000回学習★ 学習率： 0.9511833333333334 認識率： 0.9468 勾配差分： 0.20889939210592628
★100個サンプル学習データを3600回学習★ 学習率： 0.9557 認識率： 0.9522 勾配差分： 0.12284927627356054
★100個サンプル学習データを4200回学習★ 学習率： 0.9613 認識率： 0.9563 勾配差分： 0.0860223050467403
★100個サンプル学習データを4800回学習★ 学習率： 0.9644166666666667 認識率： 0.9606 勾配差分： 0.18300004043458046
★100個サンプル学習データを5400回学習★ 学習率： 0.96675 認識率： 0.9634 勾配差分： 0.05001404369577504
★100個サンプル学習データを6000回学習★ 学習率： 0.96875 認識率： 0.9627 勾配差分： 0.08052315605516638
★100個サンプル学習データを6600回学習★ 学習率： 0.9701833333333333 認識率： 0.9655 勾配差分： 0.07549477934736186
★100個サンプル学習データを7200回学習★ 学習率： 0.9729 認識率： 0.9664 勾配差分： 0.05086949475875577
★100個サンプル学習データを7800回学習★ 学習率： 0.9742333333333333 認識率： 0.9676 勾配差分： 0.05080182316512686
★100個サンプル学習データを8400回学習★ 学習率： 0.9753333333333334 認識率： 0.9689 勾配差分： 0.08594969223927429
★100個サンプル学習データを9000回学習★ 学習率： 0.9769666666666666 認識率： 0.9704 勾配差分： 0.03641650867545596
★100個サンプル学習データを9600回学習★ 学習率： 0.9769666666666666 認識率： 0.9682 勾配差分： 0.04800738457481304

学習に関するテクニック

最適化（optimization） (http://www.orsj.or.jp/archive2/or60-4/or60_4_191.pdf)はニューラルネットワークの課題。損失関数（loss function）の値を小さくするようにパラメータを調整すること。過学習・勾配消失・爆発（発散）を防ぐ手法を学ぶことが学習に関するテクニックである。一般的には以下のよな手法が知られている。

- パラメータの更新方法を変更する
- 活性化関数を変更する
- ネットワークの重みの初期値を調整する
- 学習係数を下げる
- ネットワークの自由度を制約する（Dropoutなど）

しかし、近年においては

- Batch Normalizationを挿入する
- ハイパーパラメータを探索する

という手法が主流になっている。

勾配更新アルゴリズムの選定による最適化

基本的には、勾配消失・爆発を防ぐための手法であり、これまでは

活性化関数を変更する（ReLUなど） ネットワークの重みの初期値を事前学習する 学習係数を下げる ネットワークの自由度を制約する（Dropoutなど）

勾配降下法

微分の傾きを収束させることを目指す方法。

バッチ勾配降下法

ミニバッチ処理により、全てのデータ／テストを一括で順番に計算し勾配を求める。確実だが、リソースの問題がある。

確率的勾配降下法

全てのデータを一括で処理せず、データをある確率分布に基づいて無作為全抽出し勾配を求める。 優良なデータを先に処理することが出来る可能性がある。

ミニバッチ勾配降下法

全てのデータを一括で処理せず、データをある確率分布に基づいて無作為標本抽出し勾配を求める。 サンプリングレート（バッチサイズ）とサンプリング分布（基本的にランダムサンプリングとなる）により 優良な標本データを先に処理し目的関数に近づける事ができる。 現在の機械学習においては基本的にこの手法が推奨される。

勾配更新アルゴリズムによる収束までの違い

http://ruder.io/optimizing-gradient-descent/ (http://ruder.io/optimizing-gradient-descent/)
https://keras.io/optimizers/ (https://keras.io/optimizers/)

様々な勾配降下法の最適化アルゴリズム

SGD(Stochastic Gradient Descent, 確率的勾配降下法)

$W \leftarrow W - \eta \frac{\partial L}{\partial W}$
$0 < \eta < 1$
$W :: Weight \ \eta :: Learning\ Rate \ \ L :: Loss\ Function \ \partial :: Partial\ Differentiation$

単純な勾配降下法全てのベース(Vanilla)となる。全集合（学習データ集合、テストデータ集合）の中の一部(batch_size分)の集合をepoch回だけ**無作為サンプリング（サンプリング分布に注意）**し、勾配降下法の計算に反映する手法。大規模なデータを処理する際、少ない計算量で実装可能であり、非常にシンプルなのでMLアルゴリズムのデバッグが容易。**目的関数と一致しない集合で学習してしまうと急激に勾配が上下し目的関数もそれにつられて大きく変動する可能性がある。**

In [1]:

```
class SGD:
    """確率的勾配降下法（Stochastic Gradient Descent）"""
    def __init__(self, lr=0.01):
        self.lr = lr
    def update(self, params, grads):
        for key in params.keys():
            params[key] -= self.lr * grads[key]
```

Momentam (Momentam SGD)

$v \leftarrow \alpha v - \eta \frac{\partial L}{\partial W}$
$W \leftarrow W + v$
$0 < \alpha < 1$
$v :: velocity \ \ \alpha :: Air\ Resistance\ (aka.\ Friction)$

SGDにおいて勾配更新の際に勾配を直接計算に反映するのではなく「直前の勾配の加速度」変数を掛け算し続けることでSGDの勾配更新を滑らかにする方法。複雑な関数でなければSGDよりも早く収束する可能性がある。勾配と正方向の加速度が「勢い」であり、逆方向の加速度が物理学でいう「摩擦」や「空気抵抗」に近い。**加速度に定数αを毎回掛け算しているのは、静止摩擦を付加し収束を期待する為である。**

In [8]:

```
class Momentum:
    """慣性SGD (Momentam SGD)"""
    def __init__(self, lr=0.01, momentum=0.9):
        self.lr = lr
        self.momentum = momentum
        self.v = None
    def update(self, params, grads):
        if self.v is None:
            self.v = {}
            for key, val in params.items():
                self.v[key] = np.zeros_like(val)
        for key in params.keys():
            self.v[key] = self.momentum*self.v[key] - self.lr*grads[key]
            params[key] += self.v[key]
```

NAG (Nesterov's Accelerated Gradient)

Momentamでは加速度を計算する際に勾配計算前の値を使用していたが、NAGでは勾配計算後の値を使用することで、勾配の変化に関してMomentamよりも一歩先で気づくことが出来るようになる。それ以外は特に変わらない。同盟で複数のアルゴリズムが存在する。

Adagrad

$h \leftarrow h + \frac{\partial L}{\partial W} \odot \frac{\partial L}{\partial W}$
$W \leftarrow W - \eta \frac{1}{\sqrt{h}} \frac{\partial L}{\partial W}$
$h :: Rate\ of\ Learning\ Rate$ $\odot :: Hadamard\ Product$ (アダマール積::行列の要素ごとの単純乗算。内積でない。)

勾配更新の際に「学習率」自体を勾配で自乗したもので割る。これにより学習率そのものの指数関数的な減衰をもたらす。最初の勾配が大きいものが強く学習され、収束に従って緩やかに学習されるようになるので最初に大きく学習を進めることが出来る。最初に学習したデータが良好なデータであればあるほど効果的に働くが学習が進むに連れて学習率が0に近づく為、**オンライン学習（継続的学習）の場合は新鮮なデータを受け入れるのが難しくなる欠点がある。**アダマール積を使用しているのは、各種パラメータ（行・列）毎のデータ変動を一個ずつ対応して反映できるようにするためである。※hは最初は非常に小さな数値に設定し、ゼロ除算を回避すること。

In [4]:

```
class AdaGrad:
    """AdaGrad (Adaptive subgradient methods for online learning and stochastic optimization)"""
    def __init__(self, lr=0.01):
        self.lr = lr
        self.h = None
    def update(self, params, grads):
        if self.h is None:
            self.h = {}
            for key, val in params.items():
                self.h[key] = np.zeros_like(val)
        for key in params.keys():
            self.h[key] += grads[key] * grads[key]
            params[key] -= self.lr * grads[key] / (np.sqrt(self.h[key]) + 1e-7)
            #1e-7はゼロ除算を防ぐため
```

RMSprop

$h \leftarrow \alpha h + (1-\alpha)\frac{\partial L}{\partial W} \odot \frac{\partial L}{\partial W}$
$W \leftarrow W - \eta \frac{1}{\sqrt{h}} \frac{\partial L}{\partial W}$
$0 \leq \alpha \leq 1$
α :: $Rate\ of\ exponential\ moving\ average; EMA$; 指数移動平均

Adagradにおいて学習が進むにつれて新鮮なデータを受入し難くなる欠点を克服するため、**一つ前の学習率に定数αを掛けて減衰させ、受け入れる新鮮なデータの学習率の影響を強める**手法。hに関して指数移動平均を適用したものである。過去のデータの影響力の加重が指数関数的に減衰するので、αの数値が低すぎた場合は過去のデータの影響力が直ちに低下してしまい、新鮮すぎる結果が発生し、数値が高すぎた場合は腐った結果が発生するリスクがある。

In [5]:

```
class RMSprop:
    """RMSprop"""
    def __init__(self, lr=0.01, decay_rate = 0.99):
        self.lr = lr
        self.decay_rate = decay_rate
        self.h = None
    def update(self, params, grads):
        if self.h is None:
            self.h = {}
            for key, val in params.items():
                self.h[key] = np.zeros_like(val)
        for key in params.keys():
            self.h[key] *= self.decay_rate
            self.h[key] += (1 - self.decay_rate) * grads[key] * grads[key]
            params[key] -= self.lr * grads[key] / (np.sqrt(self.h[key]) + 1e-7)
```

AdaDelta

AdagradとRMSpropをあわせたもの。学習率が存在せず移動平均がそのまま学習率に反映される。

Adam

AdaGrad、RMSprop、AdaDeltaを合わせスケールを追加したもの

In [6]:

```
class Adam:
    """Adam (Adaptive moment estimation)"""
    def __init__(self, lr=0.001, beta1=0.9, beta2=0.999):
        self.lr = lr
        self.beta1 = beta1
        self.beta2 = beta2
        self.iter = 0
        self.m = None
        self.v = None
    def update(self, params, grads):
        if self.m is None:
            self.m, self.v = {}, {}
            for key, val in params.items():
                self.m[key] = np.zeros_like(val)
                self.v[key] = np.zeros_like(val)
        self.iter += 1
        lr_t  = self.lr * np.sqrt(1.0 - self.beta2**self.iter) / (1.0 - self.beta1**self.iter)

        for key in params.keys():
            #self.m[key] = self.beta1*self.m[key] + (1-self.beta1)*grads[key]
            #self.v[key] = self.beta2*self.v[key] + (1-self.beta2)*(grads[key]**2)
            self.m[key] += (1 - self.beta1) * (grads[key] - self.m[key])
            self.v[key] += (1 - self.beta2) * (grads[key]**2 - self.v[key])

            params[key] -= lr_t * self.m[key] / (np.sqrt(self.v[key]) + 1e-7)

            #unbias_m += (1 - self.beta1) * (grads[key] - self.m[key]) # correct bias
            #unbisa_b += (1 - self.beta2) * (grads[key]*grads[key] - self.v[key]) # correct bias
            #params[key] += self.lr * unbias_m / (np.sqrt(unbisa_b) + 1e-7)
```

MNISTデータセットによる更新手法の比較

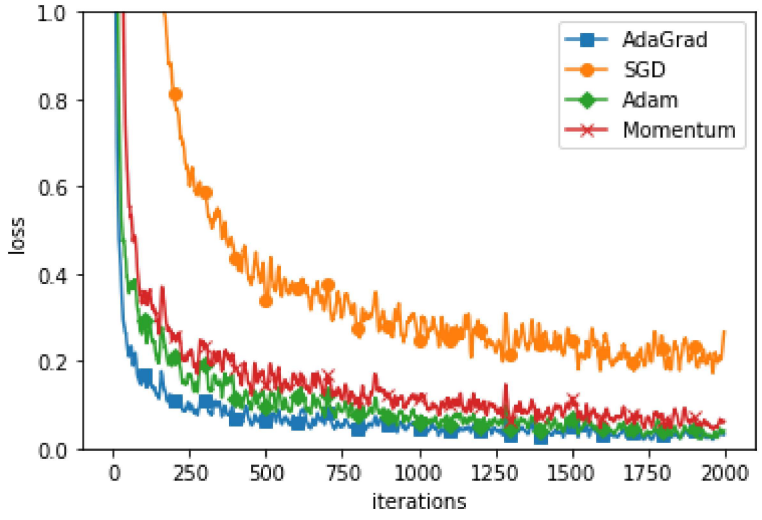
In [2]:

```
# coding: utf-8
import os
import sys
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 matplotlib.pyplot as plt
from dataset.mnist import load_mnist
from common.util import smooth_curve
from common.multi_layer_net import MultiLayerNet
from common.optimizer import *


# 0:MNISTデータの読み込み==========
(x_train, t_train), (x_test, t_test) = load_mnist(normalize=True)

train_size = x_train.shape[0]
batch_size = 128
max_iterations = 2000


# 1:実験の設定==========
optimizers = {}
optimizers['SGD'] = SGD()
optimizers['Momentum'] = Momentum()
optimizers['AdaGrad'] = AdaGrad()
optimizers['Adam'] = Adam()
#optimizers['RMSprop'] = RMSprop()

networks = {}
train_loss = {}
for key in optimizers.keys():
    networks[key] = MultiLayerNet(
        input_size=784, hidden_size_list=[100, 100, 100, 100],
        output_size=10)
    train_loss[key] = []


# 2:訓練の開始==========
for i in range(max_iterations):
    batch_mask = np.random.choice(train_size, batch_size)
    x_batch = x_train[batch_mask]
    t_batch = t_train[batch_mask]

    for key in optimizers.keys():
        grads = networks[key].gradient(x_batch, t_batch)
        optimizers[key].update(networks[key].params, grads)

        loss = networks[key].loss(x_batch, t_batch)
        train_loss[key].append(loss)

    if i % 100 == 0:
        print( "===========" + "iteration:" + str(i) + "===========")
        for key in optimizers.keys():
            loss = networks[key].loss(x_batch, t_batch)
            print(key + ":" + str(loss))


# 3.グラフの描画==========
markers = {"SGD": "o", "Momentum": "x", "AdaGrad": "s", "Adam": "D"}
x = np.arange(max_iterations)
```

```
for key in optimizers.keys():
    plt.plot(x, smooth_curve(train_loss[key]), marker=markers[key], markevery=100, label=key)
plt.xlabel("iterations")
plt.ylabel("loss")
plt.ylim(0, 1)
plt.legend()
plt.show()
```

==========iteration:0==========
AdaGrad:2.073306458903855
SGD:2.523300165662966
Adam:2.121038352570406
Momentum:2.38792091140304
==========iteration:100==========
AdaGrad:0.2131088105273944
SGD:1.6044179080128664
Adam:0.3808571137285117
Momentum:0.3969831247395848
==========iteration:200==========
AdaGrad:0.13124025086807267
SGD:0.7830985515944628
Adam:0.22064014047211133
Momentum:0.3216731484452754
==========iteration:300==========
AdaGrad:0.06808923098744606
SGD:0.6325237467606476
Adam:0.16051907326110734
Momentum:0.23042036201574118
==========iteration:400==========
AdaGrad:0.06313508623562003
SGD:0.45209547106182657
Adam:0.13090639195898196
Momentum:0.16881274795022827
==========iteration:500==========
AdaGrad:0.0746353204854379
SGD:0.33067102745064325
Adam:0.12430283308476585
Momentum:0.17795406996349522
==========iteration:600==========
AdaGrad:0.041437727939144256
SGD:0.36918525431898697
Adam:0.127014809423026
Momentum:0.16947254478379217
==========iteration:700==========
AdaGrad:0.07854171706682012
SGD:0.3318232246950431
Adam:0.1209986514495654
Momentum:0.14913166839228031
==========iteration:800==========
AdaGrad:0.05195006133127446
SGD:0.3025152503875316
Adam:0.10859296970221899
Momentum:0.1732654572600429
==========iteration:900==========
AdaGrad:0.07781836733265007
SGD:0.26347814560455185
Adam:0.08727690727995363
Momentum:0.15707644843134425
==========iteration:1000==========
AdaGrad:0.03065622516498518
SGD:0.20389467073413406
Adam:0.027024578641755104
Momentum:0.07996497290843516
==========iteration:1100==========
AdaGrad:0.09872289321572919
SGD:0.23547226998879367
Adam:0.0934946021846558
Momentum:0.14254957758981768
==========iteration:1200==========

AdaGrad:0.055191379811268274
SGD:0.3109738353784294
Adam:0.06514229213904917
Momentum:0.1379175719410351
==========iteration:1300==========
AdaGrad:0.04387204579920525
SGD:0.26196650572611313
Adam:0.0625853256957209
Momentum:0.09036903564636829
==========iteration:1400==========
AdaGrad:0.02219086430991235
SGD:0.1877208319403718
Adam:0.029426675946386555
Momentum:0.0845638241078313
==========iteration:1500==========
AdaGrad:0.11608670588608397
SGD:0.33773967606904787
Adam:0.08622036049471533
Momentum:0.21650175829174115
==========iteration:1600==========
AdaGrad:0.046115608649423
SGD:0.25291991636150213
Adam:0.09449451108675003
Momentum:0.09031445958452815
==========iteration:1700==========
AdaGrad:0.03207546608854157
SGD:0.16174878427926614
Adam:0.019466190695012753
Momentum:0.03876243380325855
==========iteration:1800==========
AdaGrad:0.014310622225061862
SGD:0.17121837372399037
Adam:0.0194596575829714
Momentum:0.0393406117694879
==========iteration:1900==========
AdaGrad:0.020184199816534614
SGD:0.20354824799564297
Adam:0.021922311598918012
Momentum:0.040479245423072255



留意

- SGDは最初から最後まで学習に行き詰まっている
- Momentumは振れ幅が大きい
- Adamは最初の収束が早く振れ幅が少ない
- AdaGradが最初の方で効率よく学習しそのままキープできている

重みの初期値の調整による最適化

重みの初期値を0にするのはNG

これまではガウス分布（正規分布,np.random.randn）により生成される平均0標準偏差1のランダム分布から得られる重みを0.01倍して重みに割り当てていた。 それを例えばW = np.zeros_like(W)してしまうというのはバッドプラクティスである。
ニューラルネットワークにおいて重みの初期値を各要素毎に均一に設定してしまうと誤差逆伝搬方で各層が均一の値で更新されてしまい、層と層の間の計算に意味が無くなってしまうためである。

隠れ層のアクティベーション分布から重みの初期値の最適な標準偏差を探る

隠れ層のアクティベーション（活性化関数適用後の出力データ）の分布を観察することで、ニューラルネットワークのパラメータの調整に関して有効な手立てを考えることができる。重みの初期値を調整しアクティベーション分布に偏りなくさせてニューラルネットワークの「表現の制限」と「勾配消失（Gradient Vanishing）」を避けることが目標。

重みの初期値（Sigmoid関数をアクティベーション関数にした場合）

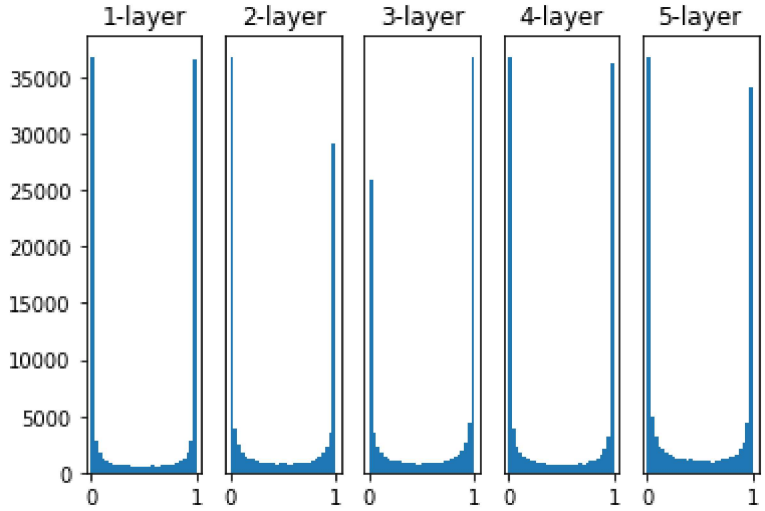
In [14]:

```
# coding: utf-8
import numpy as np
import matplotlib.pyplot as plt


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


def ReLU(x):
    return np.maximum(0, x)


def tanh(x):
    return np.tanh(x)

input_data = np.random.randn(1000, 100)  # 1000個のデータ
node_num = 100  # 各隠れ層のノード（ニューロン）の数
hidden_layer_size = 5  # 隠れ層が5層
activations = {}  # ここにアクティベーションの結果を格納する

x = input_data
print("標準偏差1の正規分布： w = np.random.randn(node_num, node_num) * 1")
for i in range(hidden_layer_size):
    if i != 0:
        x = activations[i-1]

    # 初期値の値をいろいろ変えて実験しよう！
    w = np.random.randn(node_num, node_num) * 1
    # w = np.random.randn(node_num, node_num) * 0.01
    # w = np.random.randn(node_num, node_num) * np.sqrt(1.0 / node_num)
    # w = np.random.randn(node_num, node_num) * np.sqrt(2.0 / node_num)


    a = np.dot(x, w)


    # 活性化関数の種類も変えて実験しよう！
    z = sigmoid(a)
    # z = ReLU(a)
    # z = tanh(a)

    activations[i] = z

# ヒストグラムを描画
for i, a in activations.items():
    plt.subplot(1, len(activations), i+1)
    plt.title(str(i+1) + "-layer")
    if i != 0: plt.yticks([], [])
    # plt.xlim(0.1, 1)
    # plt.ylim(0, 7000)
    plt.hist(a.flatten(), 30, range=(0,1))
plt.show()

x = input_data

print("標準偏差0.01の正規分布： w = np.random.randn(node_num, node_num) * 0.01")
for i in range(hidden_layer_size):
    if i != 0:
```

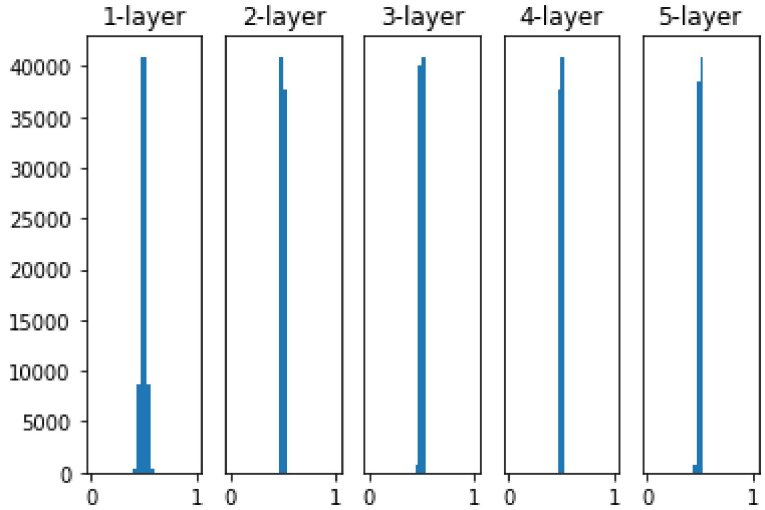
```
        x = activations[i-1]

    # 初期値の値をいろいろ変えて実験しよう！
    #w = np.random.randn(node_num, node_num) * 1
    w = np.random.randn(node_num, node_num) * 0.01
    # w = np.random.randn(node_num, node_num) * np.sqrt(1.0 / node_num)
    # w = np.random.randn(node_num, node_num) * np.sqrt(2.0 / node_num)


    a = np.dot(x, w)


    # 活性化関数の種類も変えて実験しよう！
    z = sigmoid(a)
    # z = ReLU(a)
    # z = tanh(a)

    activations[i] = z

# ヒストグラムを描画
for i, a in activations.items():
    plt.subplot(1, len(activations), i+1)
    plt.title(str(i+1) + "-layer")
    if i != 0: plt.yticks([], [])
    # plt.xlim(0.1, 1)
    # plt.ylim(0, 7000)
    plt.hist(a.flatten(), 30, range=(0,1))
plt.show()

x = input_data

print("Xavierの初期値： w = np.random.randn(node_num, node_num) * np.sqrt(1.0 / node_num)")
for i in range(hidden_layer_size):
    if i != 0:
        x = activations[i-1]

    # 初期値の値をいろいろ変えて実験しよう！
    #w = np.random.randn(node_num, node_num) * 1
    #w = np.random.randn(node_num, node_num) * 0.01
    w = np.random.randn(node_num, node_num) * np.sqrt(1.0 / node_num)
    # w = np.random.randn(node_num, node_num) * np.sqrt(2.0 / node_num)


    a = np.dot(x, w)


    # 活性化関数の種類も変えて実験しよう！
    z = sigmoid(a)
    # z = ReLU(a)
    # z = tanh(a)

    activations[i] = z

# ヒストグラムを描画
for i, a in activations.items():
    plt.subplot(1, len(activations), i+1)
    plt.title(str(i+1) + "-layer")
    if i != 0: plt.yticks([], [])
    # plt.xlim(0.1, 1)
    # plt.ylim(0, 7000)
    plt.hist(a.flatten(), 30, range=(0,1))
```

```
plt.show()

x = input_data

print("Heの初期値： w = np.random.randn(node_num, node_num) * np.sqrt(2.0 / node_num)")
for i in range(hidden_layer_size):
    if i != 0:
        x = activations[i-1]

    # 初期値の値をいろいろ変えて実験しよう！
    #w = np.random.randn(node_num, node_num) * 1
    #w = np.random.randn(node_num, node_num) * 0.01
    #w = np.random.randn(node_num, node_num) * np.sqrt(1.0 / node_num)
    w = np.random.randn(node_num, node_num) * np.sqrt(2.0 / node_num)


    a = np.dot(x, w)


    # 活性化関数の種類も変えて実験しよう！
    z = sigmoid(a)
    # z = ReLU(a)
    # z = tanh(a)

    activations[i] = z

# ヒストグラムを描画
for i, a in activations.items():
    plt.subplot(1, len(activations), i+1)
    plt.title(str(i+1) + "-layer")
    if i != 0: plt.yticks([], [])
    # plt.xlim(0.1, 1)
    # plt.ylim(0, 7000)
    plt.hist(a.flatten(), 30, range=(0,1))
plt.show()
```

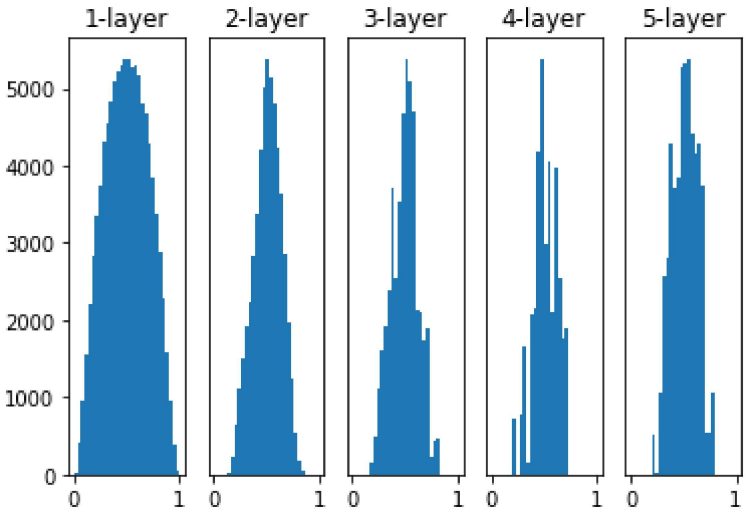
標準偏差1の正規分布： w = np.random.randn(node_num, node_num) * 1



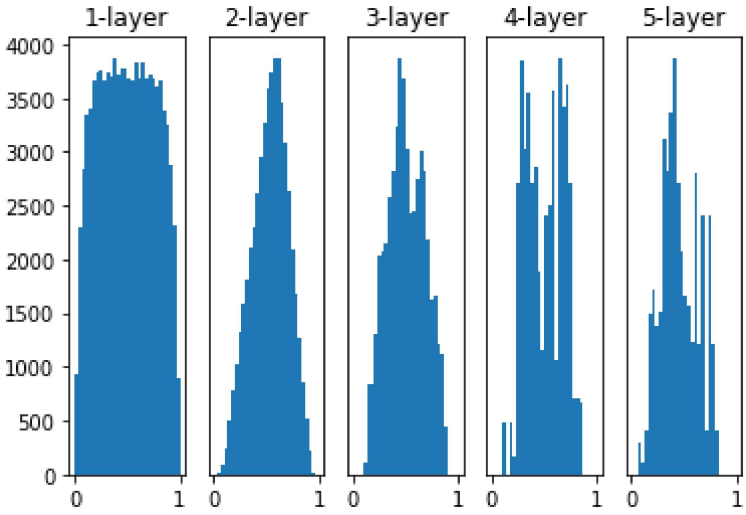
標準偏差0.01の正規分布： w = np.random.randn(node_num, node_num) * 0.01



Xavierの初期値： w = np.random.randn(node_num, node_num) * np.sqrt(1.0 / node_num)



Heの初期値： `w = np.random.randn(node_num, node_num) * np.sqrt(2.0 / node_num)`



重みの初期値（ReLU関数をアクティベーション関数にした場合）

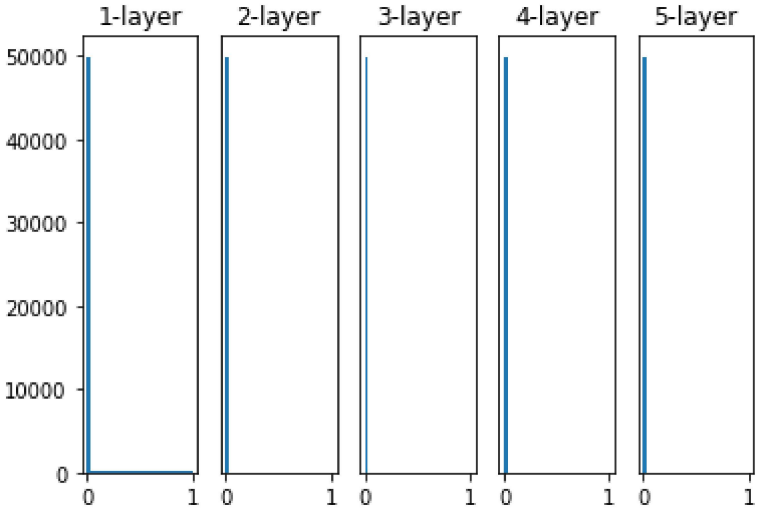
In [15]:

```
# coding: utf-8
import numpy as np
import matplotlib.pyplot as plt


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


def ReLU(x):
    return np.maximum(0, x)


def tanh(x):
    return np.tanh(x)

input_data = np.random.randn(1000, 100)  # 1000個のデータ
node_num = 100  # 各隠れ層のノード（ニューロン）の数
hidden_layer_size = 5  # 隠れ層が5層
activations = {}  # ここにアクティベーションの結果を格納する

x = input_data
print("標準偏差1の正規分布： w = np.random.randn(node_num, node_num) * 1")
for i in range(hidden_layer_size):
    if i != 0:
        x = activations[i-1]

    # 初期値の値をいろいろ変えて実験しよう！
    w = np.random.randn(node_num, node_num) * 1
    # w = np.random.randn(node_num, node_num) * 0.01
    # w = np.random.randn(node_num, node_num) * np.sqrt(1.0 / node_num)
    # w = np.random.randn(node_num, node_num) * np.sqrt(2.0 / node_num)


    a = np.dot(x, w)


    # 活性化関数の種類も変えて実験しよう！
    #z = sigmoid(a)
    z = ReLU(a)
    # z = tanh(a)

    activations[i] = z

# ヒストグラムを描画
for i, a in activations.items():
    plt.subplot(1, len(activations), i+1)
    plt.title(str(i+1) + "-layer")
    if i != 0: plt.yticks([], [])
    # plt.xlim(0.1, 1)
    # plt.ylim(0, 7000)
    plt.hist(a.flatten(), 30, range=(0,1))
plt.show()

x = input_data

print("標準偏差0.01の正規分布： w = np.random.randn(node_num, node_num) * 0.01")
for i in range(hidden_layer_size):
    if i != 0:
```

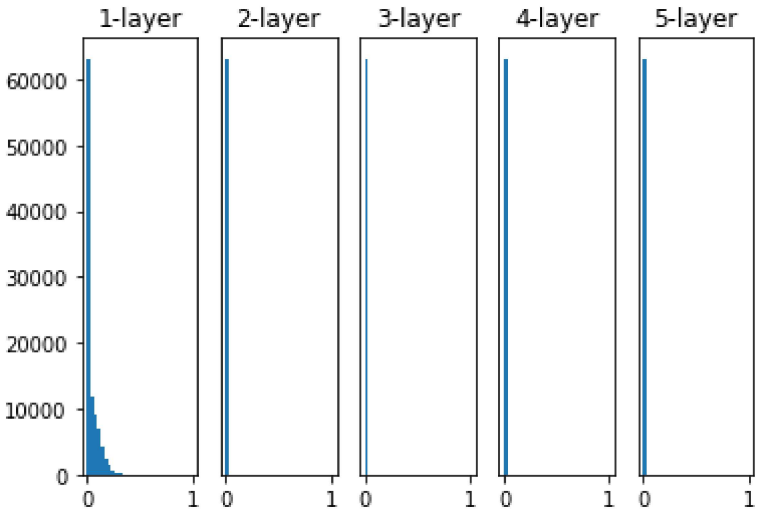
```
        x = activations[i-1]

    # 初期値の値をいろいろ変えて実験しよう！
    #w = np.random.randn(node_num, node_num) * 1
    w = np.random.randn(node_num, node_num) * 0.01
    # w = np.random.randn(node_num, node_num) * np.sqrt(1.0 / node_num)
    # w = np.random.randn(node_num, node_num) * np.sqrt(2.0 / node_num)


    a = np.dot(x, w)


    # 活性化関数の種類も変えて実験しよう！
    #z = sigmoid(a)
    z = ReLU(a)
    # z = tanh(a)

    activations[i] = z

# ヒストグラムを描画
for i, a in activations.items():
    plt.subplot(1, len(activations), i+1)
    plt.title(str(i+1) + "-layer")
    if i != 0: plt.yticks([], [])
    # plt.xlim(0.1, 1)
    # plt.ylim(0, 7000)
    plt.hist(a.flatten(), 30, range=(0,1))
plt.show()

x = input_data

print("Xavierの初期値： w = np.random.randn(node_num, node_num) * np.sqrt(1.0 / node_num)")
for i in range(hidden_layer_size):
    if i != 0:
        x = activations[i-1]

    # 初期値の値をいろいろ変えて実験しよう！
    #w = np.random.randn(node_num, node_num) * 1
    #w = np.random.randn(node_num, node_num) * 0.01
    w = np.random.randn(node_num, node_num) * np.sqrt(1.0 / node_num)
    # w = np.random.randn(node_num, node_num) * np.sqrt(2.0 / node_num)


    a = np.dot(x, w)


    # 活性化関数の種類も変えて実験しよう！
    #z = sigmoid(a)
    z = ReLU(a)
    # z = tanh(a)

    activations[i] = z

# ヒストグラムを描画
for i, a in activations.items():
    plt.subplot(1, len(activations), i+1)
    plt.title(str(i+1) + "-layer")
    if i != 0: plt.yticks([], [])
    # plt.xlim(0.1, 1)
    # plt.ylim(0, 7000)
    plt.hist(a.flatten(), 30, range=(0,1))
```

```
plt.show()

x = input_data

print("Heの初期値： w = np.random.randn(node_num, node_num) * np.sqrt(2.0 / node_num)")
for i in range(hidden_layer_size):
    if i != 0:
        x = activations[i-1]

    # 初期値の値をいろいろ変えて実験しよう！
    #w = np.random.randn(node_num, node_num) * 1
    #w = np.random.randn(node_num, node_num) * 0.01
    #w = np.random.randn(node_num, node_num) * np.sqrt(1.0 / node_num)
    w = np.random.randn(node_num, node_num) * np.sqrt(2.0 / node_num)


    a = np.dot(x, w)


    # 活性化関数の種類も変えて実験しよう！
    #z = sigmoid(a)
    z = ReLU(a)
    # z = tanh(a)

    activations[i] = z

# ヒストグラムを描画
for i, a in activations.items():
    plt.subplot(1, len(activations), i+1)
    plt.title(str(i+1) + "-layer")
    if i != 0: plt.yticks([], [])
    # plt.xlim(0.1, 1)
    # plt.ylim(0, 7000)
    plt.hist(a.flatten(), 30, range=(0,1))
plt.show()
```

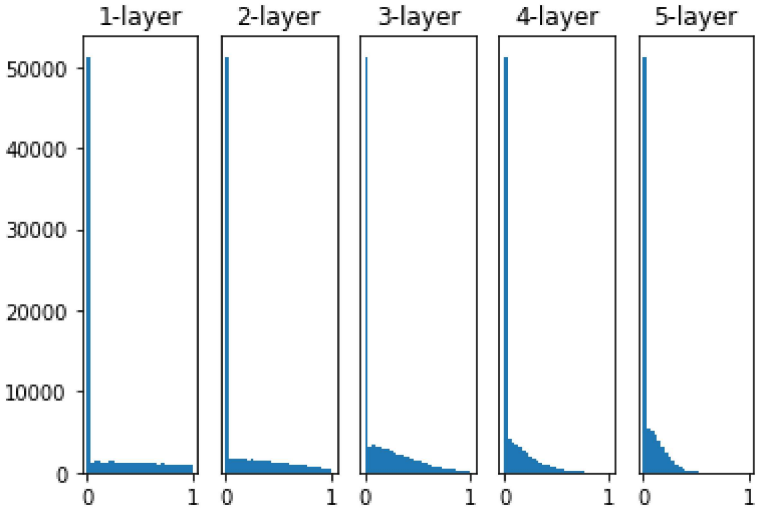
標準偏差1の正規分布： w = np.random.randn(node_num, node_num) * 1



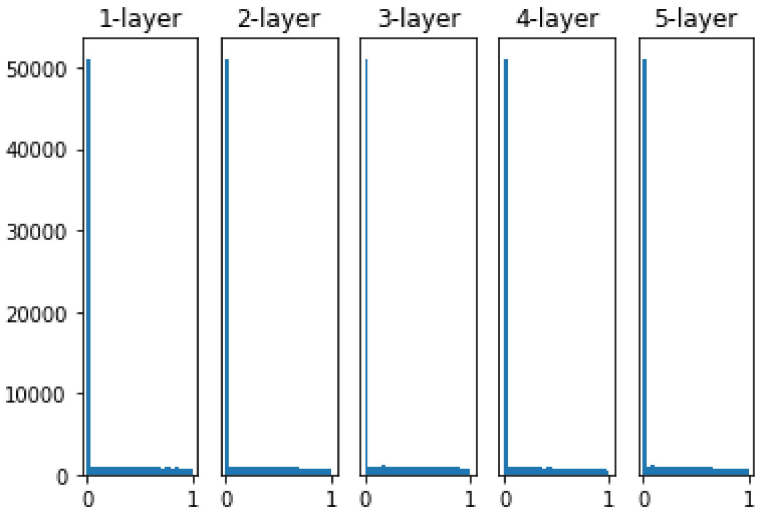
標準偏差0.01の正規分布： w = np.random.randn(node_num, node_num) * 0.01



Xavierの初期値： w = np.random.randn(node_num, node_num) * np.sqrt(1.0 / node_num)



Heの初期値： w = np.random.randn(node_num, node_num) * np.sqrt(2.0 / node_num)



重みの初期値（tanh関数をアクティベーション関数にした場合）

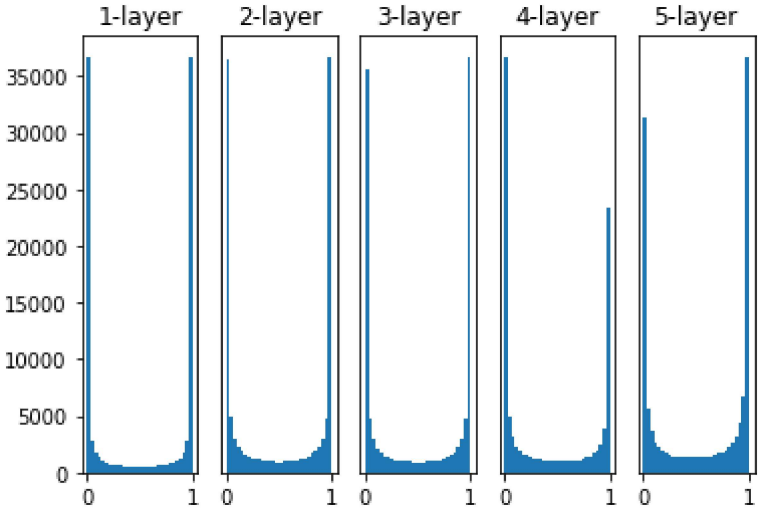
In [16]:

```
# coding: utf-8
import numpy as np
import matplotlib.pyplot as plt


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


def ReLU(x):
    return np.maximum(0, x)


def tanh(x):
    return np.tanh(x)

input_data = np.random.randn(1000, 100)  # 1000個のデータ
node_num = 100  # 各隠れ層のノード（ニューロン）の数
hidden_layer_size = 5  # 隠れ層が5層
activations = {}  # ここにアクティベーションの結果を格納する

x = input_data
print("標準偏差1の正規分布： w = np.random.randn(node_num, node_num) * 1")
for i in range(hidden_layer_size):
    if i != 0:
        x = activations[i-1]

    # 初期値の値をいろいろ変えて実験しよう！
    w = np.random.randn(node_num, node_num) * 1
    # w = np.random.randn(node_num, node_num) * 0.01
    # w = np.random.randn(node_num, node_num) * np.sqrt(1.0 / node_num)
    # w = np.random.randn(node_num, node_num) * np.sqrt(2.0 / node_num)


    a = np.dot(x, w)


    # 活性化関数の種類も変えて実験しよう！
    z = sigmoid(a)
    #z = ReLU(a)
    # z = tanh(a)

    activations[i] = z

# ヒストグラムを描画
for i, a in activations.items():
    plt.subplot(1, len(activations), i+1)
    plt.title(str(i+1) + "-layer")
    if i != 0: plt.yticks([], [])
    # plt.xlim(0.1, 1)
    # plt.ylim(0, 7000)
    plt.hist(a.flatten(), 30, range=(0,1))
plt.show()

x = input_data

print("標準偏差0.01の正規分布： w = np.random.randn(node_num, node_num) * 0.01")
for i in range(hidden_layer_size):
    if i != 0:
```

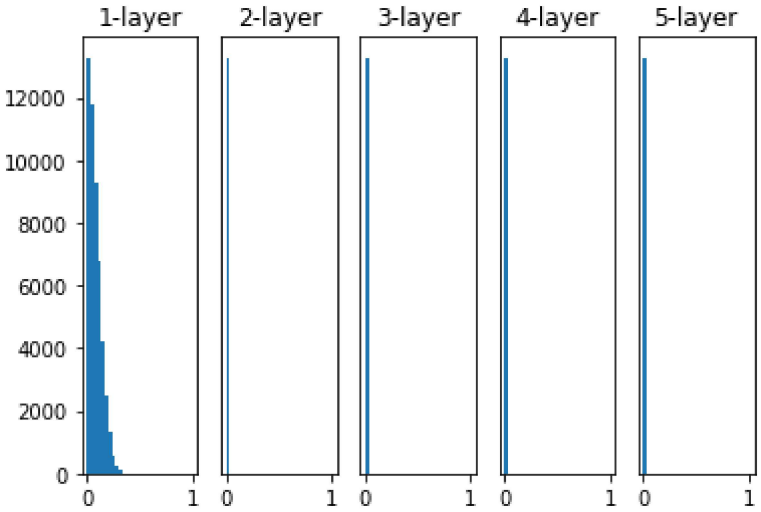
```
        x = activations[i-1]

    # 初期値の値をいろいろ変えて実験しよう！
    #w = np.random.randn(node_num, node_num) * 1
    w = np.random.randn(node_num, node_num) * 0.01
    # w = np.random.randn(node_num, node_num) * np.sqrt(1.0 / node_num)
    # w = np.random.randn(node_num, node_num) * np.sqrt(2.0 / node_num)


    a = np.dot(x, w)


    # 活性化関数の種類も変えて実験しよう！
    #z = sigmoid(a)
    #z = ReLU(a)
    z = tanh(a)

    activations[i] = z

# ヒストグラムを描画
for i, a in activations.items():
    plt.subplot(1, len(activations), i+1)
    plt.title(str(i+1) + "-layer")
    if i != 0: plt.yticks([], [])
    # plt.xlim(0.1, 1)
    # plt.ylim(0, 7000)
    plt.hist(a.flatten(), 30, range=(0,1))
plt.show()

x = input_data

print("Xavierの初期値： w = np.random.randn(node_num, node_num) * np.sqrt(1.0 / node_num)")
for i in range(hidden_layer_size):
    if i != 0:
        x = activations[i-1]

    # 初期値の値をいろいろ変えて実験しよう！
    #w = np.random.randn(node_num, node_num) * 1
    #w = np.random.randn(node_num, node_num) * 0.01
    w = np.random.randn(node_num, node_num) * np.sqrt(1.0 / node_num)
    # w = np.random.randn(node_num, node_num) * np.sqrt(2.0 / node_num)


    a = np.dot(x, w)


    # 活性化関数の種類も変えて実験しよう！
    #z = sigmoid(a)
    #z = ReLU(a)
    z = tanh(a)

    activations[i] = z

# ヒストグラムを描画
for i, a in activations.items():
    plt.subplot(1, len(activations), i+1)
    plt.title(str(i+1) + "-layer")
    if i != 0: plt.yticks([], [])
    # plt.xlim(0.1, 1)
    # plt.ylim(0, 7000)
    plt.hist(a.flatten(), 30, range=(0,1))
```

```
plt.show()

x = input_data

print("Heの初期値： w = np.random.randn(node_num, node_num) * np.sqrt(2.0 / node_num)")
for i in range(hidden_layer_size):
    if i != 0:
        x = activations[i-1]

    # 初期値の値をいろいろ変えて実験しよう！
    #w = np.random.randn(node_num, node_num) * 1
    #w = np.random.randn(node_num, node_num) * 0.01
    #w = np.random.randn(node_num, node_num) * np.sqrt(1.0 / node_num)
    w = np.random.randn(node_num, node_num) * np.sqrt(2.0 / node_num)


    a = np.dot(x, w)


    # 活性化関数の種類も変えて実験しよう！
    #z = sigmoid(a)
    #z = ReLU(a)
    z = tanh(a)

    activations[i] = z

# ヒストグラムを描画
for i, a in activations.items():
    plt.subplot(1, len(activations), i+1)
    plt.title(str(i+1) + "-layer")
    if i != 0: plt.yticks([], [])
    # plt.xlim(0.1, 1)
    # plt.ylim(0, 7000)
    plt.hist(a.flatten(), 30, range=(0,1))
plt.show()
```

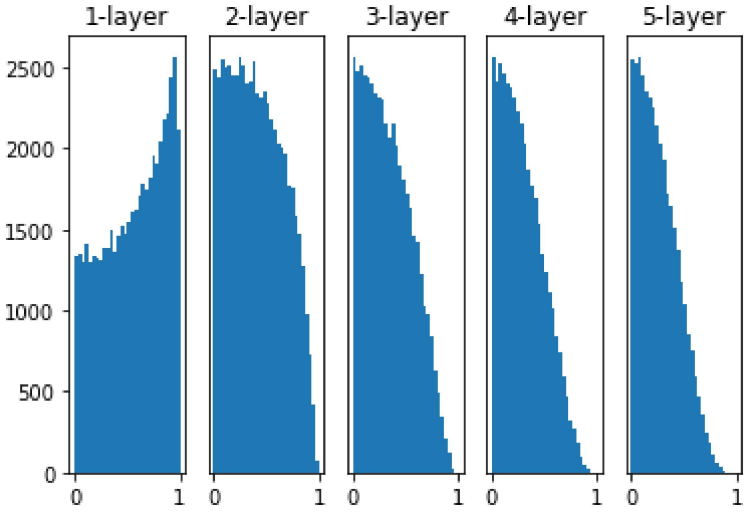
標準偏差1の正規分布： w = np.random.randn(node_num, node_num) * 1



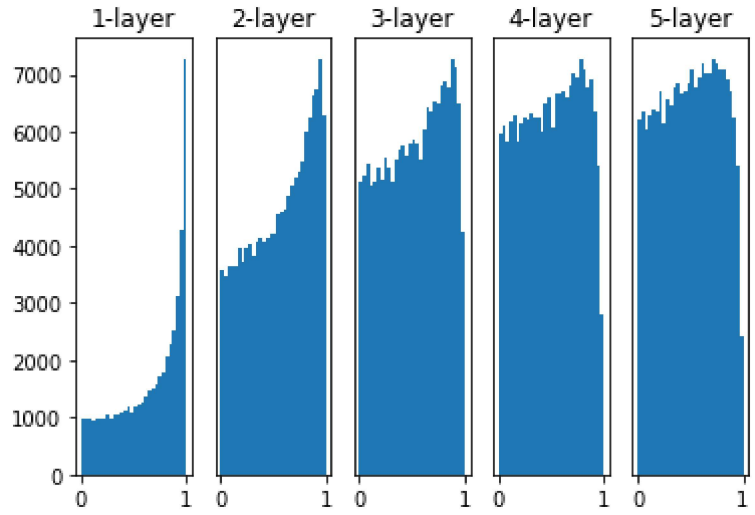
標準偏差0.01の正規分布： w = np.random.randn(node_num, node_num) * 0.01



Xavierの初期値： w = np.random.randn(node_num, node_num) * np.sqrt(1.0 / node_num)



Heの初期値： w = np.random.randn(node_num, node_num) * np.sqrt(2.0 / node_num)



ベストプラクティス：アクティベーション関数がReLU関数ならHeの初期値を、アクティベーション関数がSigmoid関数ならXavierの初期値を、アクティベーション関数がtanh関数なら・・・？

ReLU関数をアクティベーション関数として持ったMNISTニューラルネットワークに対するの重み初期値分布による誤差比較

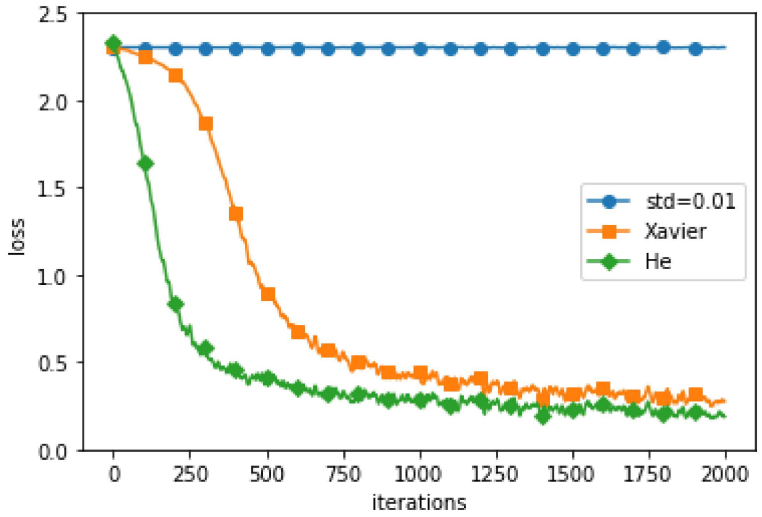
In [28]:

```
# coding: utf-8
import os
import sys

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
import matplotlib.pyplot as plt
from dataset.mnist import load_mnist
from common.util import smooth_curve
from common.multi_layer_net import MultiLayerNet
from common.optimizer import SGD


# 0:MNISTデータの読み込み==========
(x_train, t_train), (x_test, t_test) = load_mnist(normalize=True)

train_size = x_train.shape[0]
batch_size = 128
max_iterations = 2000


# 1:実験の設定==========
weight_init_types = {'std=0.01': 0.01, 'Xavier': 'sigmoid', 'He': 'relu'}
optimizer = SGD(lr=0.01)

networks = {}
train_loss = {}
for key, weight_type in weight_init_types.items():
    networks[key] = MultiLayerNet(input_size=784, hidden_size_list=[100, 100, 100, 100],
                                  output_size=10, weight_init_std=weight_type)
    train_loss[key] = []


# 2:訓練の開始==========
for i in range(max_iterations):
    batch_mask = np.random.choice(train_size, batch_size)
    x_batch = x_train[batch_mask]
    t_batch = t_train[batch_mask]

    for key in weight_init_types.keys():
        grads = networks[key].gradient(x_batch, t_batch)
        optimizer.update(networks[key].params, grads)

        loss = networks[key].loss(x_batch, t_batch)
        train_loss[key].append(loss)

    if i % 100 == 0:
        print("===========" + "iteration:" + str(i) + "===========")
        for key in weight_init_types.keys():
            loss = networks[key].loss(x_batch, t_batch)
            print(key + ":" + str(loss))


# 3.グラフの描画==========
markers = {'std=0.01': 'o', 'Xavier': 's', 'He': 'D'}
x = np.arange(max_iterations)
for key in weight_init_types.keys():
    plt.plot(x, smooth_curve(train_loss[key]), marker=markers[key], markevery=100, label=key)
plt.xlabel("iterations")
```

```
plt.ylabel("loss")
plt.ylim(0, 2.5)
plt.legend()
plt.show()
```

==========iteration:0==========
std=0.01:2.3025047128640304
Xavier:2.313383374896556
He:2.352801464350353
==========iteration:100==========
std=0.01:2.30188197530518
Xavier:2.2583643765094026
He:1.6330238543552074
==========iteration:200==========
std=0.01:2.30225644165031
Xavier:2.127229207938062
He:0.8634702094812619
==========iteration:300==========
std=0.01:2.301582259289641
Xavier:1.8551789364382827
He:0.5666139899888614
==========iteration:400==========
std=0.01:2.301262224991351
Xavier:1.448088413431125
He:0.5453957233261428
==========iteration:500==========
std=0.01:2.304384023598031
Xavier:0.9737266777861089
He:0.45937795981259955
==========iteration:600==========
std=0.01:2.301924097038047
Xavier:0.6974414715127122
He:0.40279157119477155
==========iteration:700==========
std=0.01:2.3049056291970613
Xavier:0.6206533838797386
He:0.3511152901133047
==========iteration:800==========
std=0.01:2.3021688467700567
Xavier:0.5277811176619274
He:0.31619016329750155
==========iteration:900==========
std=0.01:2.305490581934902
Xavier:0.5316058519119236
He:0.3863825629734081
==========iteration:1000==========
std=0.01:2.3019716689938097
Xavier:0.33345604396308015
He:0.19290576757400066
==========iteration:1100==========
std=0.01:2.3047431769893496
Xavier:0.32166373878701876
He:0.19660967913562705
==========iteration:1200==========
std=0.01:2.3065100112564387
Xavier:0.3919677636547596
He:0.2811031060728417
==========iteration:1300==========
std=0.01:2.298382991120612
Xavier:0.3256385951040112
He:0.23813610990007453
==========iteration:1400==========
std=0.01:2.3002620821553146
Xavier:0.19148429374938503
He:0.1170751590197682
==========iteration:1500==========

```
std=0.01:2.3102575567578034
Xavier:0.2768294280683701
He:0.17432819226626506
==========iteration:1600==========
std=0.01:2.296022719092897
Xavier:0.3269595160743929
He:0.2578675253280397
==========iteration:1700==========
std=0.01:2.2974744452575573
Xavier:0.2851566039208631
He:0.21482860725066988
==========iteration:1800==========
std=0.01:2.305259722415504
Xavier:0.3747880669248833
He:0.28387139627777247
==========iteration:1900==========
std=0.01:2.2960603606317145
Xavier:0.30538080507020426
He:0.21328616728326982
```



先述の通り、Heの初期値で学習が上手くいっている（誤差が減衰していっている）ことを確認する。これは

```
networks[key] = MultiLayerNet(input_size=784, hidden_size_list=[100, 100, 100, 100],
                              output_size=10, weight_init_std=weight_type)
```

がReLU関数をアクティベーション関数としてもっているからである。今回は5層 x100個のニューロンのニューラルネットワークでMNISTの手書き文字を学習する際のケースである。

アクティバーション関数によって発火されるニューロンの分布が偏った場合は学習以前の問題であり、もし偏りにより発火しなかった／発火の度合いが極端に低いニューロンにより引き起こされる学習が滞ってしまう。（学習の結果ニューロンの発火が起こらないのではなく、学習パラメータの設定が偏った結果発火が起こらないというのは正当なアクティベーションとは呼ばない）

Batch Normalization

先述では各層での「発火分布の正規化」をすることで学習の偏りを無くすということをしたがBatch Normalizationでは「各層の中の、ミニバッチ処理の中での発火分布の正規化」をすることで学習の方よりを無くす。以下のメリットがある。

- 学習を速く進行させることが出来る（学習係数を大きくすることが出来る）
- 初期値にそれほど依存しない
- 過学習を抑制する（Dropoutなどの必要性を減らす）

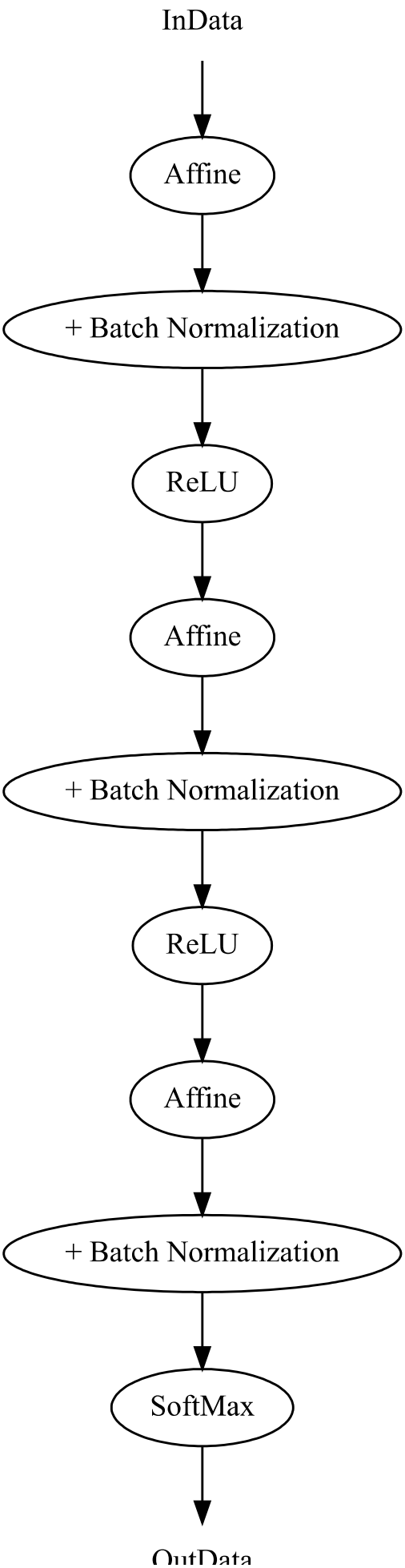
Batch Normalization とは

In [25]:

```
from graphviz import Digraph
dot = Digraph(comment="Batch Normalization: バッチ毎に正規化処理を行う")
dot.attr(rankdir="UD")
#dot.attr(ranksep="1.2")
dot.attr(nodesep=".05")
#dot.attr(margin="1")
#dot.attr(splines="polyline") #line or curved or ortho or polyline;
dot.attr(fixedsize="true")
dot.attr(label="Batch Normalization: バッチ毎に正規化処理を行う")
with dot.subgraph(name="main") as main:
    with main.subgraph(name="cluster_L1") as L1:
        L1.attr(label="")
        L1.attr(color="white")
        L1.node("main.L1N1", "InData", color="white")
    with main.subgraph(name="cluster_L2") as L2:
        L2.attr(label="")
        L2.attr(color="white")
        L2.node("main.L2N1","Affine")
    with main.subgraph(name="cluster_L3") as L3:
        L3.attr(label="")
        L3.attr(color="white")
        L3.node("main.L3N1", "+ Batch Normalization")
    with main.subgraph(name="cluster_L4") as L4:
        L4.attr(label="")
        L4.attr(color="white")
        L4.node("main.L4N1", "ReLU")
    with main.subgraph(name="cluster_L5") as L5:
        L5.attr(label="")
        L5.attr(color="white")
        L5.node("main.L5N1","Affine")
    with main.subgraph(name="cluster_L6") as L6:
        L6.attr(label="")
        L6.attr(color="white")
        L6.node("main.L6N1", "+ Batch Normalization")
    with main.subgraph(name="cluster_L7") as L7:
        L7.attr(label="")
        L7.attr(color="white")
        L7.node("main.L7N1", "ReLU")
    with main.subgraph(name="cluster_L8") as L8:
        L8.attr(label="")
        L8.attr(color="white")
        L8.node("main.L8N1","Affine")
    with main.subgraph(name="cluster_L9") as L9:
        L9.attr(label="")
        L9.attr(color="white")
        L9.node("main.L9N1", "+ Batch Normalization")
    with main.subgraph(name="cluster_L10") as L10:
        L10.attr(label="")
        L10.attr(color="white")
        L10.node("main.L10N1", "SoftMax")
    with main.subgraph(name="cluster_L11") as L11:
        L11.attr(label="")
        L11.attr(color="white")
        L11.node("main.L11N1", "OutData", color="white")
    dot.edge("main.L1N1", "main.L2N1", label="")
    dot.edge("main.L2N1", "main.L3N1", label="")
    dot.edge("main.L3N1", "main.L4N1", label="")
    dot.edge("main.L4N1", "main.L5N1", label="")
    dot.edge("main.L5N1", "main.L6N1", label="")
```

```
    dot.edge("main.L6N1", "main.L7N1", label="")
    dot.edge("main.L7N1", "main.L8N1", label="")
    dot.edge("main.L8N1", "main.L9N1", label="")
    dot.edge("main.L9N1", "main.L10N1", label="")
    dot.edge("main.L10N1", "main.L11N1", label="")
dot
```

Out[25]:


InData
Affine
+ Batch Normalization
ReLU
Affine
+ Batch Normalization
ReLU
Affine
+ Batch Normalization
SoftMax
OutData

OutData

Batch Normalization: バッチ毎に正規化処理を行う

Batch Normalizationはその名の通り学習の際のミニバッチを基本単位としてミニバッチ毎に正規化を行います。具体的には、データの平均が0で分散が1になるように正規化をその都度行う。正規化を行うタイミングは活性化関数の前（または後）にすることで、データ分布の偏りを減らすことが出来る。

BatchNormalizationによる正規化処理

https://qiita.com/t-tkd3a/items/14950dbf55f7a3095600 (https://qiita.com/t-tkd3a/items/14950dbf55f7a3095600)

ミニバッチ配列を平均0 分散1に計算し直す処理（一般的にな正規化処理）

- ミニバッチ平均: $\mu_x \leftarrow \frac{1}{N}\sum_{i=1}^{N} x_i$
- ミニバッチ分散: $\sigma_x^2 \leftarrow \frac{1}{N}\sum_{i=1}^{N}(x_i - \mu_x)^2$
- ミニバッチの正規化計算式: $\hat{x_i} \leftarrow \frac{x_i - \mu_x}{\sqrt{\sigma_x^2 + \epsilon}}$

x_i : ミニバッチ配列 $(x_1, x_2, x_3, x_4, \ldots, x_i)$
ϵ : 非常に小さな値 (0.0000001等) ※0除算回避の為
$\hat{x_i}$: 正規化後ミニバッチ配列 $(\hat{x}_1, \hat{x}_2, \hat{x}_3, \hat{x}_4, \ldots, \hat{x}_i)$

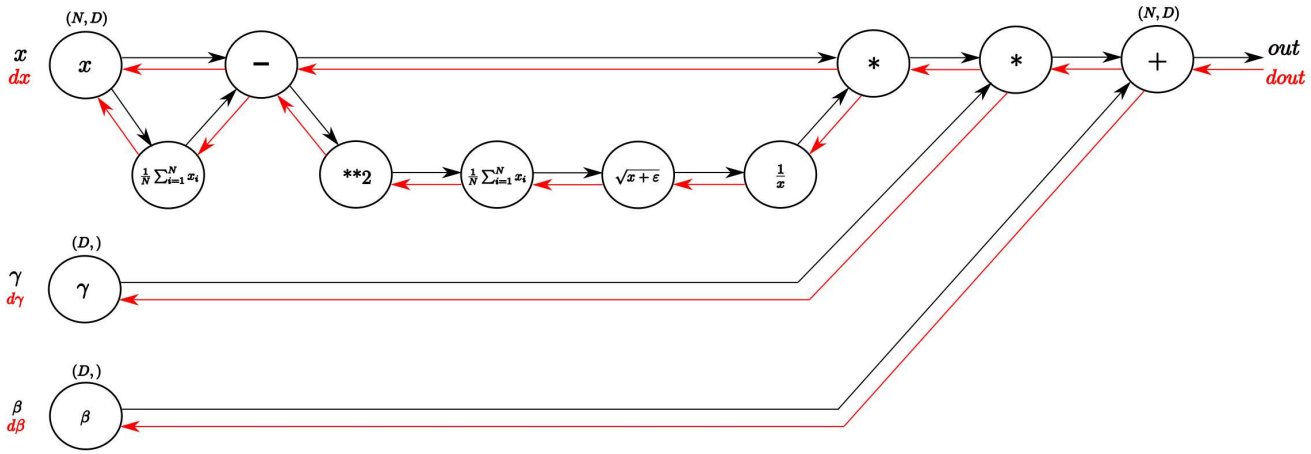
正規化したデータに対するスケールとシフト変換が可能になる

- ミニバッチ配列のスケール・シフト: $y_i \leftarrow \gamma\hat{x} + \beta$

γ : 変倍調整パラメタ。$\gamma = 1$を初期値とする。
β : 移動調整パラメタ。$\beta = 0$を初期値とする。

Batch Normalizationの計算グラフ

https://kratzert.github.io/2016/02/12/understanding-the-gradient-flow-through-the-batch-normalization-layer.html (https://kratzert.github.io/2016/02/12/understanding-the-gradient-flow-through-the-batch-normalization-layer.html)



Btach Normalizationの評価

In [27]:

```
# coding: utf-8
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
import matplotlib.pyplot as plt
from dataset.mnist import load_mnist
from common.multi_layer_net_extend import MultiLayerNetExtend
from common.optimizer import SGD, Adam

(x_train, t_train), (x_test, t_test) = load_mnist(normalize=True)

# 学習データを削減
x_train = x_train[:1000]
t_train = t_train[:1000]

max_epochs = 20
train_size = x_train.shape[0]
batch_size = 100
learning_rate = 0.01


def __train(weight_init_std):
    bn_network = MultiLayerNetExtend(input_size=784, hidden_size_list=[100, 100, 100, 100, 100],
 output_size=10,
                                     weight_init_std=weight_init_std, use_batchnorm=True)
    network = MultiLayerNetExtend(input_size=784, hidden_size_list=[100, 100, 100, 100, 100], ou
tput_size=10,
                                  weight_init_std=weight_init_std)
    optimizer = SGD(lr=learning_rate)

    train_acc_list = []
    bn_train_acc_list = []

    iter_per_epoch = max(train_size / batch_size, 1)
    epoch_cnt = 0

    for i in range(1000000000):
        batch_mask = np.random.choice(train_size, batch_size)
        x_batch = x_train[batch_mask]
        t_batch = t_train[batch_mask]

        for _network in (bn_network, network):
            grads = _network.gradient(x_batch, t_batch)
            optimizer.update(_network.params, grads)

        if i % iter_per_epoch == 0:
            train_acc = network.accuracy(x_train, t_train)
            bn_train_acc = bn_network.accuracy(x_train, t_train)
            train_acc_list.append(train_acc)
            bn_train_acc_list.append(bn_train_acc)

            print("epoch:" + str(epoch_cnt) + " | " + str(train_acc) + " - " + str(bn_train_acc
))

            epoch_cnt += 1
            if epoch_cnt >= max_epochs:
                break

    return train_acc_list, bn_train_acc_list
```

```
# 3.グラフの描画==========
weight_scale_list = np.logspace(0, -4, num=16)
x = np.arange(max_epochs)

for i, w in enumerate(weight_scale_list):
    print( "============== " + str(i+1) + "/16" + " ==============")
    train_acc_list, bn_train_acc_list = __train(w)

    plt.subplot(4,4,i+1)
    plt.title("W:" + str(w))
    if i == 15:
        plt.plot(x, bn_train_acc_list, label='Batch Normalization', markevery=2)
        plt.plot(x, train_acc_list, linestyle = "--", label='Normal(without BatchNorm)', markeve
ry=2)
    else:
        plt.plot(x, bn_train_acc_list, markevery=2)
        plt.plot(x, train_acc_list, linestyle="--", markevery=2)

    plt.ylim(0, 1.0)
    if i % 4:
        plt.yticks([])
    else:
        plt.ylabel("accuracy")
    if i < 12:
        plt.xticks([])
    else:
        plt.xlabel("epochs")
    plt.legend(loc='lower right')

plt.show()
```

```
============== 1/16 ==============
epoch:0 | 0.093 - 0.113
epoch:1 | 0.117 - 0.128
epoch:2 | 0.116 - 0.134
epoch:3 | 0.116 - 0.15
epoch:4 | 0.116 - 0.172
epoch:5 | 0.117 - 0.188
epoch:6 | 0.116 - 0.217
epoch:7 | 0.116 - 0.23
epoch:8 | 0.116 - 0.25
epoch:9 | 0.116 - 0.266
epoch:10 | 0.116 - 0.286
epoch:11 | 0.116 - 0.304
epoch:12 | 0.116 - 0.325
epoch:13 | 0.116 - 0.331
epoch:14 | 0.116 - 0.339
epoch:15 | 0.116 - 0.354
epoch:16 | 0.116 - 0.371
epoch:17 | 0.116 - 0.386
epoch:18 | 0.116 - 0.407
```

No handles with labels found to put in legend.

```
epoch:19 | 0.116 - 0.416
============== 2/16 ==============
epoch:0 | 0.097 - 0.068
```

```
/home/nbuser/library/common/multi_layer_net_extend.py:101: RuntimeWarning: overflo
w encountered in square
  weight_decay += 0.5 * self.weight_decay_lambda * np.sum(W**2)
/home/nbuser/library/common/multi_layer_net_extend.py:101: RuntimeWarning: invalid
value encountered in double_scalars
  weight_decay += 0.5 * self.weight_decay_lambda * np.sum(W**2)
/home/nbuser/anaconda3_420/lib/python3.5/site-packages/numpy/core/_methods.py:26:
RuntimeWarning: invalid value encountered in reduce
  return umr_maximum(a, axis, None, out, keepdims)
/home/nbuser/library/common/layers.py:12: RuntimeWarning: invalid value encountere
d in less_equal
  self.mask = (x <= 0)
```

```
epoch:1 | 0.097 - 0.079
epoch:2 | 0.097 - 0.104
epoch:3 | 0.097 - 0.113
epoch:4 | 0.097 - 0.133
epoch:5 | 0.097 - 0.165
epoch:6 | 0.097 - 0.185
epoch:7 | 0.097 - 0.203
epoch:8 | 0.097 - 0.245
epoch:9 | 0.097 - 0.284
epoch:10 | 0.097 - 0.295
epoch:11 | 0.097 - 0.332
epoch:12 | 0.097 - 0.347
epoch:13 | 0.097 - 0.373
epoch:14 | 0.097 - 0.385
epoch:15 | 0.097 - 0.407
epoch:16 | 0.097 - 0.42
epoch:17 | 0.097 - 0.441
epoch:18 | 0.097 - 0.454
```

No handles with labels found to put in legend.

```
epoch:19 | 0.097 - 0.464
============== 3/16 ==============
epoch:0 | 0.087 - 0.072
epoch:1 | 0.268 - 0.087
epoch:2 | 0.373 - 0.109
epoch:3 | 0.502 - 0.13
epoch:4 | 0.58 - 0.179
epoch:5 | 0.636 - 0.202
epoch:6 | 0.694 - 0.241
epoch:7 | 0.715 - 0.279
epoch:8 | 0.765 - 0.318
epoch:9 | 0.794 - 0.353
epoch:10 | 0.809 - 0.373
epoch:11 | 0.84 - 0.413
epoch:12 | 0.864 - 0.446
epoch:13 | 0.885 - 0.478
epoch:14 | 0.894 - 0.507
epoch:15 | 0.916 - 0.529
epoch:16 | 0.925 - 0.541
epoch:17 | 0.93 - 0.57
epoch:18 | 0.944 - 0.599

No handles with labels found to put in legend.
epoch:19 | 0.95 - 0.62
============== 4/16 ==============
epoch:0 | 0.11 - 0.079
epoch:1 | 0.202 - 0.104
epoch:2 | 0.33 - 0.16
epoch:3 | 0.455 - 0.225
epoch:4 | 0.517 - 0.291
epoch:5 | 0.602 - 0.362
epoch:6 | 0.643 - 0.421
epoch:7 | 0.67 - 0.482
epoch:8 | 0.695 - 0.518
epoch:9 | 0.714 - 0.559
epoch:10 | 0.74 - 0.594
epoch:11 | 0.752 - 0.616
epoch:12 | 0.77 - 0.646
epoch:13 | 0.78 - 0.668
epoch:14 | 0.789 - 0.682
epoch:15 | 0.801 - 0.698
epoch:16 | 0.815 - 0.722
epoch:17 | 0.825 - 0.737
epoch:18 | 0.831 - 0.755

No handles with labels found to put in legend.
```

```
epoch:19 | 0.839 - 0.764
============== 5/16 ==============
epoch:0 | 0.095 - 0.105
epoch:1 | 0.099 - 0.164
epoch:2 | 0.107 - 0.222
epoch:3 | 0.115 - 0.333
epoch:4 | 0.131 - 0.421
epoch:5 | 0.156 - 0.516
epoch:6 | 0.169 - 0.585
epoch:7 | 0.199 - 0.646
epoch:8 | 0.227 - 0.694
epoch:9 | 0.239 - 0.719
epoch:10 | 0.257 - 0.742
epoch:11 | 0.247 - 0.76
epoch:12 | 0.247 - 0.78
epoch:13 | 0.261 - 0.807
epoch:14 | 0.264 - 0.82
epoch:15 | 0.275 - 0.833
epoch:16 | 0.26 - 0.846
epoch:17 | 0.27 - 0.852
epoch:18 | 0.264 - 0.861


No handles with labels found to put in legend.

epoch:19 | 0.279 - 0.872
============== 6/16 ==============
epoch:0 | 0.087 - 0.092
epoch:1 | 0.125 - 0.268
epoch:2 | 0.137 - 0.511
epoch:3 | 0.129 - 0.637
epoch:4 | 0.132 - 0.709
epoch:5 | 0.113 - 0.749
epoch:6 | 0.113 - 0.788
epoch:7 | 0.121 - 0.824
epoch:8 | 0.139 - 0.848
epoch:9 | 0.113 - 0.863
epoch:10 | 0.143 - 0.878
epoch:11 | 0.122 - 0.897
epoch:12 | 0.123 - 0.908
epoch:13 | 0.116 - 0.916
epoch:14 | 0.116 - 0.932
epoch:15 | 0.116 - 0.938
epoch:16 | 0.116 - 0.947
epoch:17 | 0.129 - 0.955
epoch:18 | 0.157 - 0.958

No handles with labels found to put in legend.
```

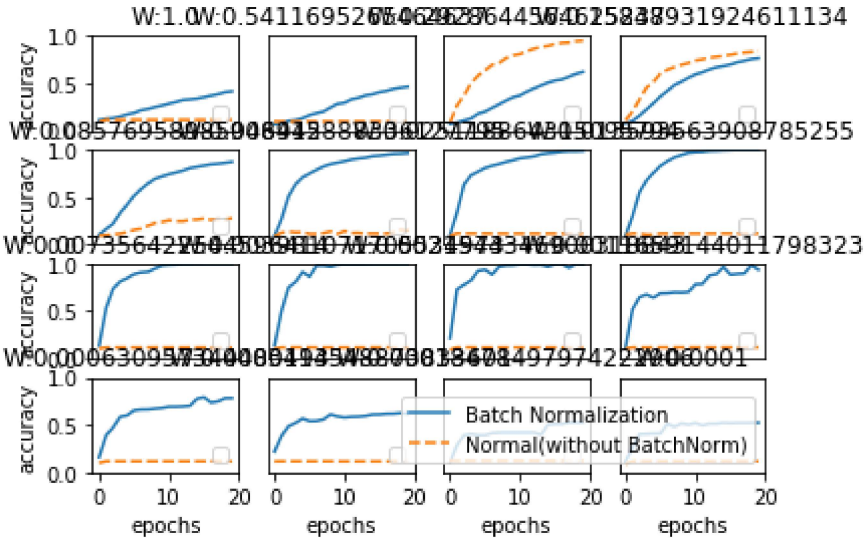
```
epoch:19 | 0.146 - 0.963
============== 7/16 ==============
epoch:0 | 0.094 - 0.102
epoch:1 | 0.118 - 0.34
epoch:2 | 0.117 - 0.635
epoch:3 | 0.117 - 0.731
epoch:4 | 0.117 - 0.768
epoch:5 | 0.117 - 0.808
epoch:6 | 0.116 - 0.831
epoch:7 | 0.116 - 0.849
epoch:8 | 0.116 - 0.871
epoch:9 | 0.116 - 0.889
epoch:10 | 0.116 - 0.91
epoch:11 | 0.116 - 0.919
epoch:12 | 0.116 - 0.93
epoch:13 | 0.116 - 0.952
epoch:14 | 0.116 - 0.962
epoch:15 | 0.116 - 0.969
epoch:16 | 0.116 - 0.98
epoch:17 | 0.116 - 0.985
epoch:18 | 0.116 - 0.985

No handles with labels found to put in legend.

epoch:19 | 0.116 - 0.988
============== 8/16 ==============
epoch:0 | 0.105 - 0.092
epoch:1 | 0.117 - 0.299
epoch:2 | 0.117 - 0.553
epoch:3 | 0.105 - 0.677
epoch:4 | 0.105 - 0.757
epoch:5 | 0.116 - 0.83
epoch:6 | 0.116 - 0.879
epoch:7 | 0.117 - 0.919
epoch:8 | 0.116 - 0.944
epoch:9 | 0.116 - 0.967
epoch:10 | 0.116 - 0.975
epoch:11 | 0.116 - 0.978
epoch:12 | 0.116 - 0.987
epoch:13 | 0.116 - 0.989
epoch:14 | 0.117 - 0.993
epoch:15 | 0.117 - 0.995
epoch:16 | 0.117 - 0.998
epoch:17 | 0.116 - 0.997
epoch:18 | 0.117 - 0.997

No handles with labels found to put in legend.
```

```
epoch:19 | 0.116 - 0.997
============== 9/16 ==============
epoch:0 | 0.105 - 0.134
epoch:1 | 0.116 - 0.532
epoch:2 | 0.116 - 0.736
epoch:3 | 0.116 - 0.811
epoch:4 | 0.116 - 0.846
epoch:5 | 0.116 - 0.889
epoch:6 | 0.116 - 0.907
epoch:7 | 0.116 - 0.913
epoch:8 | 0.117 - 0.948
epoch:9 | 0.116 - 0.98
epoch:10 | 0.116 - 0.986
epoch:11 | 0.116 - 0.989
epoch:12 | 0.116 - 0.996
epoch:13 | 0.116 - 0.996
epoch:14 | 0.116 - 0.999
epoch:15 | 0.116 - 0.999
epoch:16 | 0.116 - 1.0
epoch:17 | 0.116 - 1.0
epoch:18 | 0.116 - 1.0


No handles with labels found to put in legend.

epoch:19 | 0.116 - 1.0
============== 10/16 ==============
epoch:0 | 0.105 - 0.137
epoch:1 | 0.116 - 0.499
epoch:2 | 0.117 - 0.75
epoch:3 | 0.117 - 0.807
epoch:4 | 0.117 - 0.912
epoch:5 | 0.117 - 0.862
epoch:6 | 0.116 - 0.979
epoch:7 | 0.116 - 0.979
epoch:8 | 0.116 - 0.968
epoch:9 | 0.116 - 0.987
epoch:10 | 0.116 - 0.993
epoch:11 | 0.116 - 0.992
epoch:12 | 0.116 - 0.992
epoch:13 | 0.116 - 0.994
epoch:14 | 0.117 - 0.996
epoch:15 | 0.117 - 0.997
epoch:16 | 0.117 - 0.997
epoch:17 | 0.117 - 0.997
epoch:18 | 0.117 - 0.998

No handles with labels found to put in legend.
```

```
epoch:19 | 0.117 - 0.998
============== 11/16 ==============
epoch:0 | 0.092 - 0.212
epoch:1 | 0.117 - 0.73
epoch:2 | 0.117 - 0.777
epoch:3 | 0.117 - 0.823
epoch:4 | 0.116 - 0.923
epoch:5 | 0.116 - 0.936
epoch:6 | 0.116 - 0.884
epoch:7 | 0.116 - 0.968
epoch:8 | 0.116 - 0.98
epoch:9 | 0.116 - 0.982
epoch:10 | 0.116 - 0.991
epoch:11 | 0.116 - 0.991
epoch:12 | 0.116 - 0.985
epoch:13 | 0.116 - 0.987
epoch:14 | 0.116 - 0.968
epoch:15 | 0.116 - 0.992
epoch:16 | 0.116 - 0.995
epoch:17 | 0.116 - 0.955
epoch:18 | 0.116 - 0.995

No handles with labels found to put in legend.

epoch:19 | 0.116 - 0.992
============== 12/16 ==============
epoch:0 | 0.105 - 0.112
epoch:1 | 0.116 - 0.517
epoch:2 | 0.117 - 0.647
epoch:3 | 0.116 - 0.674
epoch:4 | 0.117 - 0.642
epoch:5 | 0.117 - 0.687
epoch:6 | 0.117 - 0.689
epoch:7 | 0.117 - 0.698
epoch:8 | 0.117 - 0.696
epoch:9 | 0.117 - 0.701
epoch:10 | 0.117 - 0.782
epoch:11 | 0.117 - 0.786
epoch:12 | 0.117 - 0.875
epoch:13 | 0.116 - 0.89
epoch:14 | 0.117 - 0.963
epoch:15 | 0.117 - 0.884
epoch:16 | 0.117 - 0.891
epoch:17 | 0.117 - 0.896
epoch:18 | 0.117 - 0.985

No handles with labels found to put in legend.
```

```
epoch:19 | 0.117 - 0.935
============== 13/16 ==============
epoch:0 | 0.099 - 0.16
epoch:1 | 0.117 - 0.39
epoch:2 | 0.117 - 0.472
epoch:3 | 0.117 - 0.585
epoch:4 | 0.117 - 0.6
epoch:5 | 0.117 - 0.653
epoch:6 | 0.117 - 0.662
epoch:7 | 0.117 - 0.663
epoch:8 | 0.117 - 0.671
epoch:9 | 0.117 - 0.678
epoch:10 | 0.117 - 0.692
epoch:11 | 0.117 - 0.692
epoch:12 | 0.117 - 0.695
epoch:13 | 0.117 - 0.702
epoch:14 | 0.117 - 0.776
epoch:15 | 0.117 - 0.791
epoch:16 | 0.117 - 0.736
epoch:17 | 0.117 - 0.748
epoch:18 | 0.116 - 0.781


No handles with labels found to put in legend.

epoch:19 | 0.116 - 0.782
============== 14/16 ==============
epoch:0 | 0.116 - 0.22
epoch:1 | 0.117 - 0.383
epoch:2 | 0.117 - 0.486
epoch:3 | 0.117 - 0.52
epoch:4 | 0.117 - 0.567
epoch:5 | 0.117 - 0.539
epoch:6 | 0.117 - 0.541
epoch:7 | 0.117 - 0.558
epoch:8 | 0.117 - 0.61
epoch:9 | 0.116 - 0.589
epoch:10 | 0.116 - 0.578
epoch:11 | 0.116 - 0.585
epoch:12 | 0.116 - 0.587
epoch:13 | 0.116 - 0.594
epoch:14 | 0.116 - 0.609
epoch:15 | 0.116 - 0.61
epoch:16 | 0.116 - 0.617
epoch:17 | 0.116 - 0.618
epoch:18 | 0.116 - 0.625

No handles with labels found to put in legend.
```

```
epoch:19 | 0.116 - 0.618
============== 15/16 ==============
epoch:0 | 0.117 - 0.103
epoch:1 | 0.117 - 0.289
epoch:2 | 0.117 - 0.39
epoch:3 | 0.117 - 0.405
epoch:4 | 0.117 - 0.393
epoch:5 | 0.117 - 0.391
epoch:6 | 0.117 - 0.413
epoch:7 | 0.117 - 0.418
epoch:8 | 0.117 - 0.419
epoch:9 | 0.117 - 0.419
epoch:10 | 0.117 - 0.418
epoch:11 | 0.117 - 0.419
epoch:12 | 0.117 - 0.419
epoch:13 | 0.117 - 0.396
epoch:14 | 0.117 - 0.503
epoch:15 | 0.117 - 0.508
epoch:16 | 0.117 - 0.508
epoch:17 | 0.117 - 0.526
epoch:18 | 0.117 - 0.526

No handles with labels found to put in legend.

epoch:19 | 0.117 - 0.525
============== 16/16 ==============
epoch:0 | 0.087 - 0.116
epoch:1 | 0.117 - 0.258
epoch:2 | 0.116 - 0.402
epoch:3 | 0.116 - 0.404
epoch:4 | 0.117 - 0.409
epoch:5 | 0.117 - 0.409
epoch:6 | 0.117 - 0.512
epoch:7 | 0.117 - 0.479
epoch:8 | 0.117 - 0.515
epoch:9 | 0.117 - 0.502
epoch:10 | 0.117 - 0.52
epoch:11 | 0.117 - 0.494
epoch:12 | 0.117 - 0.513
epoch:13 | 0.117 - 0.512
epoch:14 | 0.117 - 0.522
epoch:15 | 0.117 - 0.518
epoch:16 | 0.117 - 0.521
epoch:17 | 0.117 - 0.521
epoch:18 | 0.117 - 0.522
epoch:19 | 0.117 - 0.523
```



正則化

正則化とは特定のデータに囚われない学習モデルを作ること。

過学習

過学習とは、訓練データだけに適応しすぎてしまい、訓練データに含まれない他のデータにはうまく対応できない状態を言う。
過学習が起きる原因としては以下のような原因が挙げられる。

- パラメータを大量に持ち、表現力の高いモデルであること
- 訓練データが少ないこと

わざと過学習させる

In [29]:

```
# coding: utf-8
import os
import sys

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
import matplotlib.pyplot as plt
from dataset.mnist import load_mnist
from common.multi_layer_net import MultiLayerNet
from common.optimizer import SGD

(x_train, t_train), (x_test, t_test) = load_mnist(normalize=True)

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

# weight decay（荷重減衰）の設定 =======================
#weight_decay_lambda = 0 # weight decayを使用しない場合
weight_decay_lambda = 0.1
# ====================================================

network = MultiLayerNet(input_size=784, hidden_size_list=[100, 100, 100, 100, 100, 100], output_
size=10,
                        weight_decay_lambda=weight_decay_lambda)
optimizer = SGD(lr=0.01)

max_epochs = 201
train_size = x_train.shape[0]
batch_size = 100

train_loss_list = []
train_acc_list = []
test_acc_list = []

iter_per_epoch = max(train_size / batch_size, 1)
epoch_cnt = 0

for i in range(1000000000):
    batch_mask = np.random.choice(train_size, batch_size)
    x_batch = x_train[batch_mask]
    t_batch = t_train[batch_mask]

    grads = network.gradient(x_batch, t_batch)
    optimizer.update(network.params, grads)

    if i % iter_per_epoch == 0:
        train_acc = network.accuracy(x_train, t_train)
        test_acc = network.accuracy(x_test, t_test)
        train_acc_list.append(train_acc)
        test_acc_list.append(test_acc)

        print("epoch:" + str(epoch_cnt) + ", train acc:" + str(train_acc) + ", test acc:" + str(
test_acc))

        epoch_cnt += 1
        if epoch_cnt >= max_epochs:
            break
```

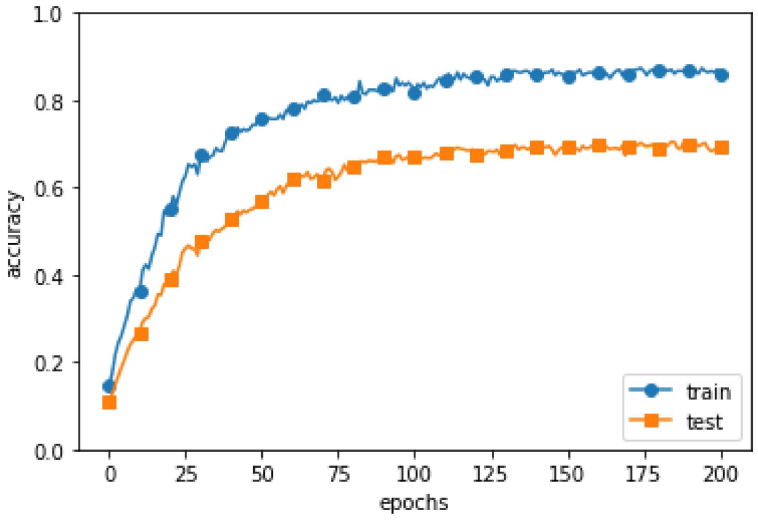
```
# 3.グラフの描画==========
markers = {'train': 'o', 'test': 's'}
x = np.arange(max_epochs)
plt.plot(x, train_acc_list, marker='o', label='train', markevery=10)
plt.plot(x, test_acc_list, marker='s', label='test', markevery=10)
plt.xlabel("epochs")
plt.ylabel("accuracy")
plt.ylim(0, 1.0)
plt.legend(loc='lower right')
plt.show()
```

epoch:0, train acc:0.14666666666666667, test acc:0.1087
epoch:1, train acc:0.17333333333333334, test acc:0.127
epoch:2, train acc:0.21666666666666667, test acc:0.1475
epoch:3, train acc:0.24333333333333335, test acc:0.1669
epoch:4, train acc:0.26, test acc:0.1875
epoch:5, train acc:0.2833333333333333, test acc:0.2045
epoch:6, train acc:0.30666666666666664, test acc:0.2249
epoch:7, train acc:0.34, test acc:0.2411
epoch:8, train acc:0.3466666666666667, test acc:0.2496
epoch:9, train acc:0.36666666666666664, test acc:0.2624
epoch:10, train acc:0.36333333333333334, test acc:0.2681
epoch:11, train acc:0.41, test acc:0.2916
epoch:12, train acc:0.42333333333333334, test acc:0.3005
epoch:13, train acc:0.4133333333333333, test acc:0.3031
epoch:14, train acc:0.44333333333333336, test acc:0.3216
epoch:15, train acc:0.4633333333333333, test acc:0.3298
epoch:16, train acc:0.49333333333333335, test acc:0.3542
epoch:17, train acc:0.49, test acc:0.3539
epoch:18, train acc:0.5466666666666666, test acc:0.3776
epoch:19, train acc:0.5366666666666666, test acc:0.3813
epoch:20, train acc:0.55, test acc:0.392
epoch:21, train acc:0.58, test acc:0.409
epoch:22, train acc:0.56, test acc:0.4
epoch:23, train acc:0.5866666666666667, test acc:0.4197
epoch:24, train acc:0.6133333333333333, test acc:0.4506
epoch:25, train acc:0.6266666666666667, test acc:0.4604
epoch:26, train acc:0.6533333333333333, test acc:0.4668
epoch:27, train acc:0.6466666666666666, test acc:0.4606
epoch:28, train acc:0.6533333333333333, test acc:0.4579
epoch:29, train acc:0.63, test acc:0.4444
epoch:30, train acc:0.6733333333333333, test acc:0.4795
epoch:31, train acc:0.67, test acc:0.4755
epoch:32, train acc:0.66, test acc:0.4763
epoch:33, train acc:0.67, test acc:0.4813
epoch:34, train acc:0.6733333333333333, test acc:0.4952
epoch:35, train acc:0.69, test acc:0.5024
epoch:36, train acc:0.6833333333333333, test acc:0.4992
epoch:37, train acc:0.6833333333333333, test acc:0.5056
epoch:38, train acc:0.7033333333333334, test acc:0.5105
epoch:39, train acc:0.72, test acc:0.5236
epoch:40, train acc:0.7266666666666667, test acc:0.5294
epoch:41, train acc:0.7133333333333334, test acc:0.5265
epoch:42, train acc:0.7366666666666667, test acc:0.5455
epoch:43, train acc:0.7266666666666667, test acc:0.5378
epoch:44, train acc:0.7333333333333333, test acc:0.547
epoch:45, train acc:0.73, test acc:0.5428
epoch:46, train acc:0.7333333333333333, test acc:0.549
epoch:47, train acc:0.7433333333333333, test acc:0.5504
epoch:48, train acc:0.7433333333333333, test acc:0.5557
epoch:49, train acc:0.76, test acc:0.5782
epoch:50, train acc:0.7566666666666667, test acc:0.5692
epoch:51, train acc:0.75, test acc:0.5744
epoch:52, train acc:0.76, test acc:0.5832
epoch:53, train acc:0.7566666666666667, test acc:0.5899
epoch:54, train acc:0.7566666666666667, test acc:0.582
epoch:55, train acc:0.7566666666666667, test acc:0.595
epoch:56, train acc:0.7666666666666667, test acc:0.6031
epoch:57, train acc:0.7566666666666667, test acc:0.5873
epoch:58, train acc:0.7766666666666666, test acc:0.6036
epoch:59, train acc:0.7733333333333333, test acc:0.6112
epoch:60, train acc:0.78, test acc:0.6178

epoch:61, train acc:0.7866666666666666, test acc:0.6215
epoch:62, train acc:0.7833333333333333, test acc:0.6201
epoch:63, train acc:0.79, test acc:0.6258
epoch:64, train acc:0.78, test acc:0.6258
epoch:65, train acc:0.7966666666666666, test acc:0.6354
epoch:66, train acc:0.7933333333333333, test acc:0.6198
epoch:67, train acc:0.8, test acc:0.6219
epoch:68, train acc:0.7966666666666666, test acc:0.6297
epoch:69, train acc:0.8, test acc:0.625
epoch:70, train acc:0.8133333333333334, test acc:0.6176
epoch:71, train acc:0.8, test acc:0.6394
epoch:72, train acc:0.8033333333333333, test acc:0.6428
epoch:73, train acc:0.8, test acc:0.6353
epoch:74, train acc:0.8066666666666666, test acc:0.6169
epoch:75, train acc:0.7933333333333333, test acc:0.6248
epoch:76, train acc:0.81, test acc:0.6346
epoch:77, train acc:0.8, test acc:0.6519
epoch:78, train acc:0.8066666666666666, test acc:0.6388
epoch:79, train acc:0.8066666666666666, test acc:0.6458
epoch:80, train acc:0.81, test acc:0.6473
epoch:81, train acc:0.8, test acc:0.6401
epoch:82, train acc:0.8433333333333334, test acc:0.6587
epoch:83, train acc:0.8166666666666667, test acc:0.659
epoch:84, train acc:0.8133333333333334, test acc:0.6574
epoch:85, train acc:0.8233333333333334, test acc:0.6614
epoch:86, train acc:0.8233333333333334, test acc:0.6604
epoch:87, train acc:0.8233333333333334, test acc:0.6593
epoch:88, train acc:0.82, test acc:0.6623
epoch:89, train acc:0.8333333333333334, test acc:0.66
epoch:90, train acc:0.8266666666666667, test acc:0.6722
epoch:91, train acc:0.8333333333333334, test acc:0.6602
epoch:92, train acc:0.83, test acc:0.668
epoch:93, train acc:0.82, test acc:0.6559
epoch:94, train acc:0.85, test acc:0.6719
epoch:95, train acc:0.8333333333333334, test acc:0.6745
epoch:96, train acc:0.84, test acc:0.6625
epoch:97, train acc:0.8333333333333334, test acc:0.6657
epoch:98, train acc:0.84, test acc:0.6708
epoch:99, train acc:0.8333333333333334, test acc:0.656
epoch:100, train acc:0.8166666666666667, test acc:0.6687
epoch:101, train acc:0.8366666666666667, test acc:0.6751
epoch:102, train acc:0.83, test acc:0.6647
epoch:103, train acc:0.84, test acc:0.6682
epoch:104, train acc:0.8266666666666667, test acc:0.6682
epoch:105, train acc:0.8333333333333334, test acc:0.6666
epoch:106, train acc:0.83, test acc:0.671
epoch:107, train acc:0.84, test acc:0.6772
epoch:108, train acc:0.85, test acc:0.6735
epoch:109, train acc:0.8466666666666667, test acc:0.6762
epoch:110, train acc:0.8433333333333334, test acc:0.6779
epoch:111, train acc:0.86, test acc:0.6819
epoch:112, train acc:0.8533333333333334, test acc:0.6779
epoch:113, train acc:0.8466666666666667, test acc:0.6881
epoch:114, train acc:0.8633333333333333, test acc:0.6912
epoch:115, train acc:0.8466666666666667, test acc:0.6884
epoch:116, train acc:0.86, test acc:0.6872
epoch:117, train acc:0.8533333333333334, test acc:0.6851
epoch:118, train acc:0.85, test acc:0.6842
epoch:119, train acc:0.8533333333333334, test acc:0.6871
epoch:120, train acc:0.8566666666666667, test acc:0.6741
epoch:121, train acc:0.8533333333333334, test acc:0.6797

epoch:122, train acc:0.85, test acc:0.6805
epoch:123, train acc:0.8466666666666667, test acc:0.6762
epoch:124, train acc:0.8333333333333334, test acc:0.683
epoch:125, train acc:0.8533333333333334, test acc:0.6827
epoch:126, train acc:0.8533333333333334, test acc:0.6809
epoch:127, train acc:0.8433333333333334, test acc:0.6866
epoch:128, train acc:0.85, test acc:0.6748
epoch:129, train acc:0.8466666666666667, test acc:0.6789
epoch:130, train acc:0.86, test acc:0.6842
epoch:131, train acc:0.8533333333333334, test acc:0.691
epoch:132, train acc:0.8666666666666667, test acc:0.6924
epoch:133, train acc:0.8666666666666667, test acc:0.6966
epoch:134, train acc:0.8633333333333333, test acc:0.6866
epoch:135, train acc:0.8633333333333333, test acc:0.6918
epoch:136, train acc:0.8666666666666667, test acc:0.6916
epoch:137, train acc:0.8666666666666667, test acc:0.6869
epoch:138, train acc:0.86, test acc:0.6909
epoch:139, train acc:0.8533333333333334, test acc:0.6889
epoch:140, train acc:0.86, test acc:0.6936
epoch:141, train acc:0.86, test acc:0.6987
epoch:142, train acc:0.86, test acc:0.6918
epoch:143, train acc:0.8633333333333333, test acc:0.6921
epoch:144, train acc:0.8633333333333333, test acc:0.6945
epoch:145, train acc:0.87, test acc:0.6864
epoch:146, train acc:0.8566666666666667, test acc:0.6747
epoch:147, train acc:0.8633333333333333, test acc:0.6838
epoch:148, train acc:0.86, test acc:0.6914
epoch:149, train acc:0.8566666666666667, test acc:0.6888
epoch:150, train acc:0.8533333333333334, test acc:0.6922
epoch:151, train acc:0.8566666666666667, test acc:0.6854
epoch:152, train acc:0.8633333333333333, test acc:0.6877
epoch:153, train acc:0.8633333333333333, test acc:0.6891
epoch:154, train acc:0.87, test acc:0.6831
epoch:155, train acc:0.8566666666666667, test acc:0.695
epoch:156, train acc:0.86, test acc:0.6894
epoch:157, train acc:0.8633333333333333, test acc:0.6901
epoch:158, train acc:0.8633333333333333, test acc:0.6948
epoch:159, train acc:0.8566666666666667, test acc:0.6952
epoch:160, train acc:0.8633333333333333, test acc:0.6997
epoch:161, train acc:0.8633333333333333, test acc:0.6964
epoch:162, train acc:0.86, test acc:0.6969
epoch:163, train acc:0.86, test acc:0.6923
epoch:164, train acc:0.85, test acc:0.6878
epoch:165, train acc:0.86, test acc:0.6896
epoch:166, train acc:0.8666666666666667, test acc:0.6905
epoch:167, train acc:0.87, test acc:0.686
epoch:168, train acc:0.8566666666666667, test acc:0.6954
epoch:169, train acc:0.8666666666666667, test acc:0.6733
epoch:170, train acc:0.86, test acc:0.6944
epoch:171, train acc:0.8633333333333333, test acc:0.7021
epoch:172, train acc:0.87, test acc:0.6985
epoch:173, train acc:0.87, test acc:0.6966
epoch:174, train acc:0.8733333333333333, test acc:0.7019
epoch:175, train acc:0.8666666666666667, test acc:0.6967
epoch:176, train acc:0.8566666666666667, test acc:0.6941
epoch:177, train acc:0.8666666666666667, test acc:0.6958
epoch:178, train acc:0.87, test acc:0.6978
epoch:179, train acc:0.87, test acc:0.6945
epoch:180, train acc:0.8666666666666667, test acc:0.6881
epoch:181, train acc:0.8666666666666667, test acc:0.6954
epoch:182, train acc:0.87, test acc:0.6921

```
epoch:183, train acc:0.8666666666666667, test acc:0.6971
epoch:184, train acc:0.87, test acc:0.7043
epoch:185, train acc:0.86, test acc:0.7043
epoch:186, train acc:0.8666666666666667, test acc:0.6904
epoch:187, train acc:0.8666666666666667, test acc:0.6886
epoch:188, train acc:0.8666666666666667, test acc:0.7035
epoch:189, train acc:0.87, test acc:0.6978
epoch:190, train acc:0.87, test acc:0.6995
epoch:191, train acc:0.8633333333333333, test acc:0.6968
epoch:192, train acc:0.8666666666666667, test acc:0.6996
epoch:193, train acc:0.86, test acc:0.6943
epoch:194, train acc:0.8733333333333333, test acc:0.6986
epoch:195, train acc:0.8666666666666667, test acc:0.7018
epoch:196, train acc:0.8633333333333333, test acc:0.6899
epoch:197, train acc:0.8666666666666667, test acc:0.6829
epoch:198, train acc:0.8633333333333333, test acc:0.6856
epoch:199, train acc:0.87, test acc:0.6921
epoch:200, train acc:0.86, test acc:0.693
```



訓練データに関する認識精度は100%なのに、テストデータに関する認識精度は60%～80%で高止まりしている→過学習状態

Weight Decay （荷重減衰）

学習途中で大きな重みを持ったニューロンに対してペナルティを付与（損失関数への加算）すること。
過学習により大きな重みを持ってしまったニューロンの発火を抑制できる。

- $w \leftarrow w - \eta\frac{\partial C(w)}{\partial w} - WeightDecay$
  - $WeightDecay \leftarrow \frac{1}{2}\lambda W^2$
    - $W(L2realm) \leftarrow \sqrt{w_1^2 + w_2^2 + \cdots + w_n^2}$
    - $W(L1realm) \leftarrow |w_1^2| + |w_2^2| + \cdots + |w_n^2|$

※Weight Decayのイメージ的には$-(\frac{\partial C(w)}{\partial w} + WeightDecay)$であり、Cost関数+WeightDecayで重みの大きさに比例してと言った感じになる

Weight Decay対応版NN

In [2]:

```
# coding: utf-8
import os
import sys

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
import matplotlib.pyplot as plt
from dataset.mnist import load_mnist
from common.multi_layer_net import MultiLayerNet
from common.optimizer import SGD

(x_train, t_train), (x_test, t_test) = load_mnist(normalize=True)

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

# weight decay（荷重減衰）の設定 =======================
#weight_decay_lambda = 0 # weight decayを使用しない場合
weight_decay_lambda = 0.1
# ====================================================

network = MultiLayerNet(input_size=784, hidden_size_list=[100, 100, 100, 100, 100, 100], output_
size=10,
                        weight_decay_lambda=weight_decay_lambda)
optimizer = SGD(lr=0.01)

max_epochs = 201
train_size = x_train.shape[0]
batch_size = 100

train_loss_list = []
train_acc_list = []
test_acc_list = []

iter_per_epoch = max(train_size / batch_size, 1)
epoch_cnt = 0

for i in range(1000000000):
    batch_mask = np.random.choice(train_size, batch_size)
    x_batch = x_train[batch_mask]
    t_batch = t_train[batch_mask]

    grads = network.gradient(x_batch, t_batch)
    optimizer.update(network.params, grads)

    if i % iter_per_epoch == 0:
        train_acc = network.accuracy(x_train, t_train)
        test_acc = network.accuracy(x_test, t_test)
        train_acc_list.append(train_acc)
        test_acc_list.append(test_acc)

        print("epoch:" + str(epoch_cnt) + ", train acc:" + str(train_acc) + ", test acc:" + str(
test_acc))

        epoch_cnt += 1
        if epoch_cnt >= max_epochs:
            break
```

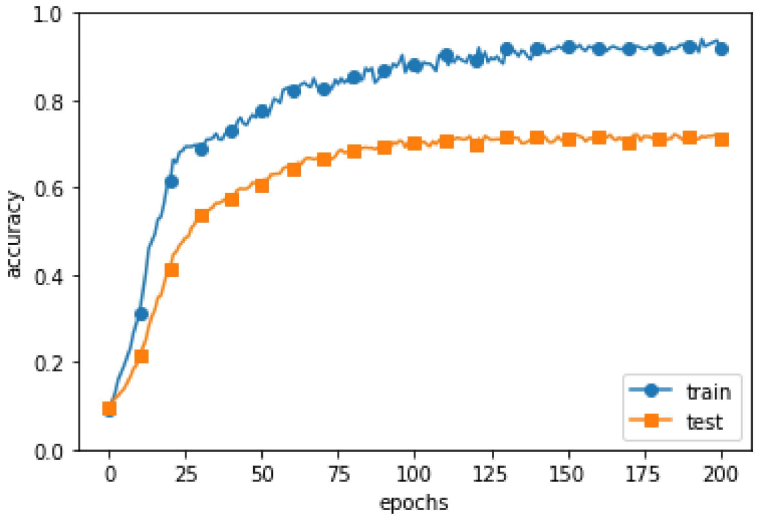
```
# 3.グラフの描画==========
markers = {'train': 'o', 'test': 's'}
x = np.arange(max_epochs)
plt.plot(x, train_acc_list, marker='o', label='train', markevery=10)
plt.plot(x, test_acc_list, marker='s', label='test', markevery=10)
plt.xlabel("epochs")
plt.ylabel("accuracy")
plt.ylim(0, 1.0)
plt.legend(loc='lower right')
plt.show()
```

epoch:0, train acc:0.09333333333333334, test acc:0.0978
epoch:1, train acc:0.11, test acc:0.1078
epoch:2, train acc:0.13, test acc:0.1171
epoch:3, train acc:0.16, test acc:0.1228
epoch:4, train acc:0.17666666666666667, test acc:0.1323
epoch:5, train acc:0.19333333333333333, test acc:0.1394
epoch:6, train acc:0.21333333333333335, test acc:0.1529
epoch:7, train acc:0.23333333333333334, test acc:0.1672
epoch:8, train acc:0.26666666666666666, test acc:0.1844
epoch:9, train acc:0.29, test acc:0.1938
epoch:10, train acc:0.31333333333333335, test acc:0.2147
epoch:11, train acc:0.36, test acc:0.2329
epoch:12, train acc:0.4033333333333333, test acc:0.2501
epoch:13, train acc:0.46, test acc:0.2811
epoch:14, train acc:0.4766666666666667, test acc:0.3041
epoch:15, train acc:0.49333333333333335, test acc:0.3196
epoch:16, train acc:0.5266666666666666, test acc:0.3472
epoch:17, train acc:0.5333333333333333, test acc:0.3533
epoch:18, train acc:0.5633333333333334, test acc:0.3796
epoch:19, train acc:0.6033333333333334, test acc:0.4042
epoch:20, train acc:0.6133333333333333, test acc:0.4144
epoch:21, train acc:0.6633333333333333, test acc:0.4455
epoch:22, train acc:0.6566666666666666, test acc:0.4527
epoch:23, train acc:0.68, test acc:0.4668
epoch:24, train acc:0.6833333333333333, test acc:0.4745
epoch:25, train acc:0.6933333333333334, test acc:0.4847
epoch:26, train acc:0.6933333333333334, test acc:0.4874
epoch:27, train acc:0.6933333333333334, test acc:0.5076
epoch:28, train acc:0.6966666666666667, test acc:0.5159
epoch:29, train acc:0.6966666666666667, test acc:0.5329
epoch:30, train acc:0.69, test acc:0.5351
epoch:31, train acc:0.6966666666666667, test acc:0.5316
epoch:32, train acc:0.71, test acc:0.5443
epoch:33, train acc:0.71, test acc:0.5501
epoch:34, train acc:0.7133333333333334, test acc:0.555
epoch:35, train acc:0.7233333333333334, test acc:0.5645
epoch:36, train acc:0.71, test acc:0.5637
epoch:37, train acc:0.71, test acc:0.5683
epoch:38, train acc:0.7233333333333334, test acc:0.5725
epoch:39, train acc:0.7233333333333334, test acc:0.5785
epoch:40, train acc:0.73, test acc:0.5759
epoch:41, train acc:0.7266666666666667, test acc:0.582
epoch:42, train acc:0.75, test acc:0.5937
epoch:43, train acc:0.76, test acc:0.5974
epoch:44, train acc:0.7466666666666667, test acc:0.5962
epoch:45, train acc:0.7433333333333333, test acc:0.5969
epoch:46, train acc:0.7566666666666667, test acc:0.6018
epoch:47, train acc:0.7666666666666667, test acc:0.6109
epoch:48, train acc:0.76, test acc:0.6097
epoch:49, train acc:0.7733333333333333, test acc:0.6061
epoch:50, train acc:0.7766666666666666, test acc:0.6075
epoch:51, train acc:0.7866666666666666, test acc:0.6187
epoch:52, train acc:0.7633333333333333, test acc:0.618
epoch:53, train acc:0.7833333333333333, test acc:0.6295
epoch:54, train acc:0.8033333333333333, test acc:0.6302
epoch:55, train acc:0.8, test acc:0.6311
epoch:56, train acc:0.7933333333333333, test acc:0.6321
epoch:57, train acc:0.8233333333333334, test acc:0.6435
epoch:58, train acc:0.8266666666666667, test acc:0.6464
epoch:59, train acc:0.83, test acc:0.6491
epoch:60, train acc:0.8233333333333334, test acc:0.6407

epoch:61, train acc:0.8266666666666667, test acc:0.6528
epoch:62, train acc:0.83, test acc:0.6561
epoch:63, train acc:0.8366666666666667, test acc:0.6535
epoch:64, train acc:0.84, test acc:0.6641
epoch:65, train acc:0.8333333333333334, test acc:0.6673
epoch:66, train acc:0.83, test acc:0.6664
epoch:67, train acc:0.85, test acc:0.6641
epoch:68, train acc:0.8333333333333334, test acc:0.6661
epoch:69, train acc:0.82, test acc:0.6627
epoch:70, train acc:0.8266666666666667, test acc:0.6657
epoch:71, train acc:0.83, test acc:0.6624
epoch:72, train acc:0.83, test acc:0.6688
epoch:73, train acc:0.8266666666666667, test acc:0.665
epoch:74, train acc:0.8466666666666667, test acc:0.6707
epoch:75, train acc:0.8366666666666667, test acc:0.6816
epoch:76, train acc:0.85, test acc:0.6861
epoch:77, train acc:0.84, test acc:0.6792
epoch:78, train acc:0.8466666666666667, test acc:0.6811
epoch:79, train acc:0.84, test acc:0.6822
epoch:80, train acc:0.8566666666666667, test acc:0.6849
epoch:81, train acc:0.86, test acc:0.6881
epoch:82, train acc:0.8466666666666667, test acc:0.6873
epoch:83, train acc:0.8466666666666667, test acc:0.6904
epoch:84, train acc:0.8733333333333333, test acc:0.6917
epoch:85, train acc:0.8666666666666667, test acc:0.6902
epoch:86, train acc:0.8733333333333333, test acc:0.6906
epoch:87, train acc:0.84, test acc:0.6871
epoch:88, train acc:0.8466666666666667, test acc:0.6911
epoch:89, train acc:0.8566666666666667, test acc:0.6964
epoch:90, train acc:0.8666666666666667, test acc:0.6943
epoch:91, train acc:0.8733333333333333, test acc:0.6938
epoch:92, train acc:0.8666666666666667, test acc:0.6913
epoch:93, train acc:0.8833333333333333, test acc:0.7031
epoch:94, train acc:0.8766666666666667, test acc:0.6998
epoch:95, train acc:0.8866666666666667, test acc:0.707
epoch:96, train acc:0.9033333333333333, test acc:0.7051
epoch:97, train acc:0.8633333333333333, test acc:0.6918
epoch:98, train acc:0.8833333333333333, test acc:0.7044
epoch:99, train acc:0.87, test acc:0.6986
epoch:100, train acc:0.88, test acc:0.7018
epoch:101, train acc:0.8633333333333333, test acc:0.6954
epoch:102, train acc:0.87, test acc:0.7016
epoch:103, train acc:0.8866666666666667, test acc:0.705
epoch:104, train acc:0.8833333333333333, test acc:0.6991
epoch:105, train acc:0.8833333333333333, test acc:0.6942
epoch:106, train acc:0.8766666666666667, test acc:0.7057
epoch:107, train acc:0.8666666666666667, test acc:0.699
epoch:108, train acc:0.9033333333333333, test acc:0.7076
epoch:109, train acc:0.9, test acc:0.7124
epoch:110, train acc:0.9033333333333333, test acc:0.7089
epoch:111, train acc:0.88, test acc:0.6996
epoch:112, train acc:0.8933333333333333, test acc:0.7129
epoch:113, train acc:0.8866666666666667, test acc:0.7104
epoch:114, train acc:0.9, test acc:0.7134
epoch:115, train acc:0.8933333333333333, test acc:0.7115
epoch:116, train acc:0.89, test acc:0.7099
epoch:117, train acc:0.8866666666666667, test acc:0.7113
epoch:118, train acc:0.9033333333333333, test acc:0.7166
epoch:119, train acc:0.8966666666666666, test acc:0.7059
epoch:120, train acc:0.8933333333333333, test acc:0.6997
epoch:121, train acc:0.92, test acc:0.712

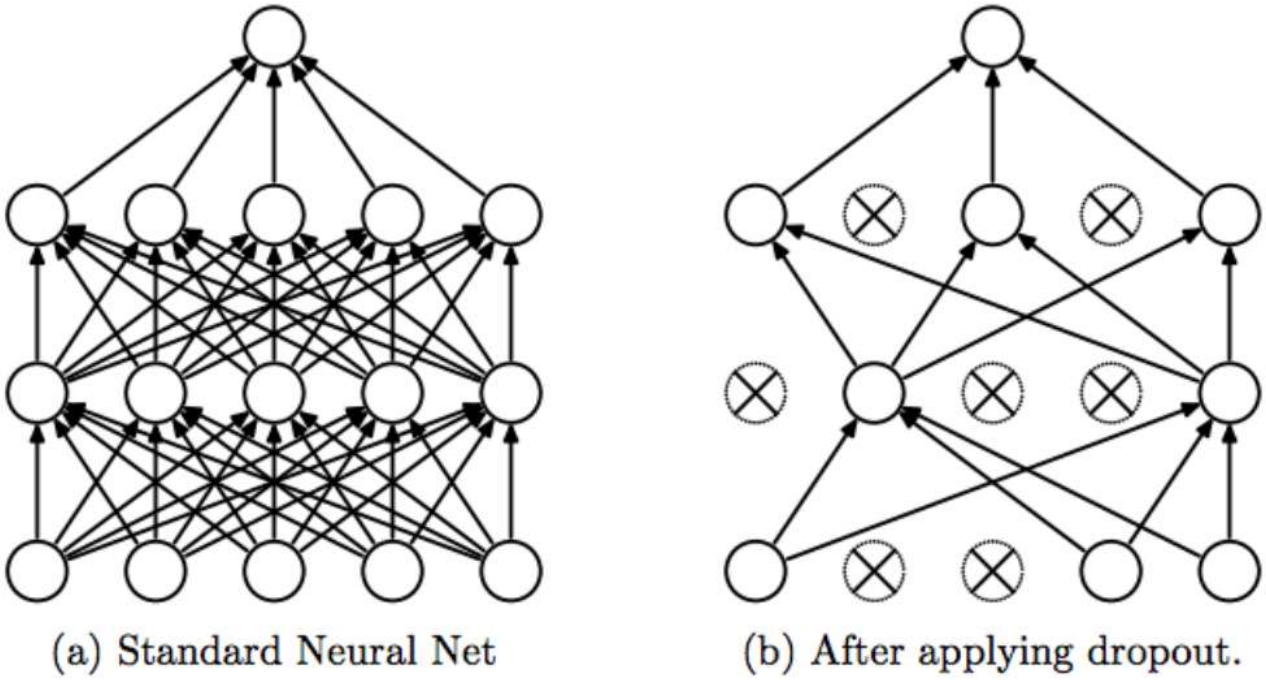
epoch:122, train acc:0.8933333333333333, test acc:0.7
epoch:123, train acc:0.91, test acc:0.7175
epoch:124, train acc:0.8866666666666667, test acc:0.7131
epoch:125, train acc:0.8966666666666666, test acc:0.7092
epoch:126, train acc:0.8933333333333333, test acc:0.7099
epoch:127, train acc:0.9, test acc:0.7104
epoch:128, train acc:0.89, test acc:0.7107
epoch:129, train acc:0.8833333333333333, test acc:0.7036
epoch:130, train acc:0.92, test acc:0.7185
epoch:131, train acc:0.9133333333333333, test acc:0.7157
epoch:132, train acc:0.9166666666666666, test acc:0.7161
epoch:133, train acc:0.91, test acc:0.7129
epoch:134, train acc:0.9, test acc:0.7127
epoch:135, train acc:0.9133333333333333, test acc:0.7014
epoch:136, train acc:0.91, test acc:0.7011
epoch:137, train acc:0.9, test acc:0.7051
epoch:138, train acc:0.8933333333333333, test acc:0.7085
epoch:139, train acc:0.91, test acc:0.7144
epoch:140, train acc:0.9166666666666666, test acc:0.7168
epoch:141, train acc:0.9166666666666666, test acc:0.7163
epoch:142, train acc:0.9266666666666666, test acc:0.7193
epoch:143, train acc:0.91, test acc:0.713
epoch:144, train acc:0.91, test acc:0.7039
epoch:145, train acc:0.9166666666666666, test acc:0.7028
epoch:146, train acc:0.9166666666666666, test acc:0.7088
epoch:147, train acc:0.9233333333333333, test acc:0.7165
epoch:148, train acc:0.92, test acc:0.7081
epoch:149, train acc:0.9133333333333333, test acc:0.7121
epoch:150, train acc:0.9233333333333333, test acc:0.7113
epoch:151, train acc:0.9166666666666666, test acc:0.7211
epoch:152, train acc:0.92, test acc:0.7175
epoch:153, train acc:0.92, test acc:0.7201
epoch:154, train acc:0.9266666666666666, test acc:0.7168
epoch:155, train acc:0.9166666666666666, test acc:0.7146
epoch:156, train acc:0.9133333333333333, test acc:0.7056
epoch:157, train acc:0.9233333333333333, test acc:0.7123
epoch:158, train acc:0.92, test acc:0.7121
epoch:159, train acc:0.9233333333333333, test acc:0.7161
epoch:160, train acc:0.9166666666666666, test acc:0.7169
epoch:161, train acc:0.92, test acc:0.7155
epoch:162, train acc:0.92, test acc:0.7148
epoch:163, train acc:0.9133333333333333, test acc:0.715
epoch:164, train acc:0.92, test acc:0.7201
epoch:165, train acc:0.9233333333333333, test acc:0.7117
epoch:166, train acc:0.92, test acc:0.7084
epoch:167, train acc:0.9166666666666666, test acc:0.714
epoch:168, train acc:0.9166666666666666, test acc:0.7098
epoch:169, train acc:0.92, test acc:0.7088
epoch:170, train acc:0.92, test acc:0.7044
epoch:171, train acc:0.9233333333333333, test acc:0.7139
epoch:172, train acc:0.9233333333333333, test acc:0.7207
epoch:173, train acc:0.9233333333333333, test acc:0.7202
epoch:174, train acc:0.92, test acc:0.7069
epoch:175, train acc:0.9166666666666666, test acc:0.7169
epoch:176, train acc:0.9166666666666666, test acc:0.7166
epoch:177, train acc:0.9233333333333333, test acc:0.7161
epoch:178, train acc:0.92, test acc:0.712
epoch:179, train acc:0.9166666666666666, test acc:0.7081
epoch:180, train acc:0.9166666666666666, test acc:0.713
epoch:181, train acc:0.92, test acc:0.7135
epoch:182, train acc:0.9133333333333333, test acc:0.706

```
epoch:183, train acc:0.91, test acc:0.7115
epoch:184, train acc:0.92, test acc:0.7119
epoch:185, train acc:0.9166666666666666, test acc:0.7214
epoch:186, train acc:0.92, test acc:0.7187
epoch:187, train acc:0.9266666666666666, test acc:0.7135
epoch:188, train acc:0.9266666666666666, test acc:0.7074
epoch:189, train acc:0.92, test acc:0.7173
epoch:190, train acc:0.9233333333333333, test acc:0.7184
epoch:191, train acc:0.9266666666666666, test acc:0.7122
epoch:192, train acc:0.9266666666666666, test acc:0.711
epoch:193, train acc:0.91, test acc:0.7109
epoch:194, train acc:0.94, test acc:0.7173
epoch:195, train acc:0.9233333333333333, test acc:0.7149
epoch:196, train acc:0.9266666666666666, test acc:0.7172
epoch:197, train acc:0.93, test acc:0.7172
epoch:198, train acc:0.9333333333333333, test acc:0.7196
epoch:199, train acc:0.9366666666666666, test acc:0.7162
epoch:200, train acc:0.92, test acc:0.7141
```



Dropout （忘却）

Dropout (https://medium.com/@amarbudhiraja/https-medium-com-amarbudhiraja-learning-less-to-learn-better-dropout-in-deep-machine-learning-74334da4bfc5)は一つの学習モデルの中でニューロンをランダムに選び出し、データが流れるたびに消去する手法。
「1 - 消去する割合」を最後の伝達の際に乗算することにより消去分の平均化を実施し 一つの学習モデルの中で多種のニューロンが発火するように強制させることができる。
複数の学習モデルによる出力を平均したものをアンサンブル学習と呼ぶが、それに近いことを一つの学習モデルで実施することができる。
DropoutされたニューロンはReLUによるスイッチオフ作用と同様、逆伝搬の対象とならない。



(a) Standard Neural Net　(b) After applying dropout.

In [3]:

```
class Dropout:
    """
    http://arxiv.org/abs/1207.0580
    """
    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 [1]:

```
# coding: utf-8
import os
import sys
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
import matplotlib.pyplot as plt
from dataset.mnist import load_mnist
from common.multi_layer_net_extend import MultiLayerNetExtend
from common.trainer import Trainer

(x_train, t_train), (x_test, t_test) = load_mnist(normalize=True)

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

# Dropuoutの有無、割り合いの設定 ========================
use_dropout = True  # Dropoutなしのときの場合はFalseに
dropout_ratio = 0.2
# ====================================================

network = MultiLayerNetExtend(input_size=784, hidden_size_list=[100, 100, 100, 100, 100, 100],
                              output_size=10, use_dropout=use_dropout, dropout_ration=dropout_ra
tio)
trainer = Trainer(network, x_train, t_train, x_test, t_test,
                  epochs=301, mini_batch_size=100,
                  optimizer='sgd', optimizer_param={'lr': 0.01}, verbose=True)
trainer.train()

train_acc_list, test_acc_list = trainer.train_acc_list, trainer.test_acc_list

# グラフの描画==========
markers = {'train': 'o', 'test': 's'}
x = np.arange(len(train_acc_list))
plt.plot(x, train_acc_list, marker='o', label='train', markevery=10)
plt.plot(x, test_acc_list, marker='s', label='test', markevery=10)
plt.xlabel("epochs")
plt.ylabel("accuracy")
plt.ylim(0, 1.0)
plt.legend(loc='lower right')
plt.show()
```

```
train loss:2.29832839634101
=== epoch:1, train acc:0.11333333333333333, test acc:0.0791 ===
train loss:2.297517209178537
train loss:2.2955034213216354
train loss:2.2935054357429094
=== epoch:2, train acc:0.13, test acc:0.0836 ===
train loss:2.2915445943865134
train loss:2.2932091073647025
train loss:2.2944382333283424
=== epoch:3, train acc:0.13, test acc:0.0884 ===
train loss:2.2996963623609634
train loss:2.2962057345642095
train loss:2.2918471366115747
=== epoch:4, train acc:0.13666666666666666, test acc:0.0943 ===
train loss:2.2985209683004317
train loss:2.2938675797102275
train loss:2.2881600803373585
=== epoch:5, train acc:0.14, test acc:0.0969 ===
train loss:2.286785688408503
train loss:2.290505963341967
train loss:2.288732557618638
=== epoch:6, train acc:0.15, test acc:0.1036 ===
train loss:2.2949984009511666
train loss:2.3009488920719052
train loss:2.2973663366129697
=== epoch:7, train acc:0.15666666666666668, test acc:0.1083 ===
train loss:2.2903975188256704
train loss:2.280853564123864
train loss:2.2913004521441804
=== epoch:8, train acc:0.17, test acc:0.1129 ===
train loss:2.2952920620965487
train loss:2.2960547856730296
train loss:2.2800716735007676
=== epoch:9, train acc:0.17333333333333334, test acc:0.1171 ===
train loss:2.294858623863755
train loss:2.2792948736124417
train loss:2.2903521925650683
=== epoch:10, train acc:0.17333333333333334, test acc:0.1226 ===
train loss:2.2645030804239754
train loss:2.2889437386336073
train loss:2.2927251088557297
=== epoch:11, train acc:0.17, test acc:0.1247 ===
train loss:2.2782813904556933
train loss:2.282887423615841
train loss:2.2805177549452167
=== epoch:12, train acc:0.17, test acc:0.1281 ===
train loss:2.2848701947712713
train loss:2.2819798640644207
train loss:2.286856187291836
=== epoch:13, train acc:0.16, test acc:0.1279 ===
train loss:2.285458398812042
train loss:2.2770987651166217
train loss:2.2772608904251035
=== epoch:14, train acc:0.16, test acc:0.1331 ===
train loss:2.2935405977943186
train loss:2.2791208944756076
train loss:2.2778281293311857
=== epoch:15, train acc:0.16333333333333333, test acc:0.1349 ===
train loss:2.285538189129508
train loss:2.2754843954083115
train loss:2.2701752627414917
```

```
=== epoch:16, train acc:0.16333333333333333, test acc:0.1386 ===
train loss:2.2852088432172555
train loss:2.2754783810332886
train loss:2.2730095540020803
=== epoch:17, train acc:0.17, test acc:0.1413 ===
train loss:2.2816786124792845
train loss:2.2824435716401066
train loss:2.2791824233244147
=== epoch:18, train acc:0.17666666666666667, test acc:0.1447 ===
train loss:2.2720210030438444
train loss:2.2822256442993925
train loss:2.2853501795346336
=== epoch:19, train acc:0.18, test acc:0.1464 ===
train loss:2.273037078676334
train loss:2.2849280809849803
train loss:2.2748931142712046
=== epoch:20, train acc:0.17666666666666667, test acc:0.1477 ===
train loss:2.263865080045291
train loss:2.2809122986443597
train loss:2.2719997473502436
=== epoch:21, train acc:0.17333333333333334, test acc:0.1498 ===
train loss:2.281826646616146
train loss:2.2660933022643768
train loss:2.2789564214731346
=== epoch:22, train acc:0.16666666666666666, test acc:0.1514 ===
train loss:2.2664560433149394
train loss:2.2754734824715137
train loss:2.277203339121532
=== epoch:23, train acc:0.18, test acc:0.1553 ===
train loss:2.2745572712969695
train loss:2.2762141676849468
train loss:2.279957844025212
=== epoch:24, train acc:0.18, test acc:0.1562 ===
train loss:2.2711526413276
train loss:2.274253252730152
train loss:2.2728695900838893
=== epoch:25, train acc:0.19, test acc:0.1587 ===
train loss:2.2678753501058897
train loss:2.2747538936780183
train loss:2.2671982243905777
=== epoch:26, train acc:0.19666666666666666, test acc:0.1628 ===
train loss:2.2649169297714966
train loss:2.268527663570704
train loss:2.2733035468560443
=== epoch:27, train acc:0.19666666666666666, test acc:0.1648 ===
train loss:2.2716226906046946
train loss:2.272909988214367
train loss:2.2709899108399307
=== epoch:28, train acc:0.19666666666666666, test acc:0.1669 ===
train loss:2.252917218814546
train loss:2.2671874530759535
train loss:2.266114683658497
=== epoch:29, train acc:0.2, test acc:0.1679 ===
train loss:2.26612574419879
train loss:2.272834810620966
train loss:2.255908925152327
=== epoch:30, train acc:0.21666666666666667, test acc:0.1721 ===
train loss:2.259900863748509
train loss:2.2662396817914527
train loss:2.255211388382625
=== epoch:31, train acc:0.21666666666666667, test acc:0.1736 ===
```

```
train loss:2.2716823833007114
train loss:2.2551021577626686
train loss:2.25756456628241
=== epoch:32, train acc:0.22666666666666666, test acc:0.1756 ===
train loss:2.2615404238871077
train loss:2.2629584035190997
train loss:2.260159467130955
=== epoch:33, train acc:0.22333333333333333, test acc:0.1774 ===
train loss:2.2588688962947128
train loss:2.2497292944907725
train loss:2.2642899289856375
=== epoch:34, train acc:0.23333333333333334, test acc:0.1784 ===
train loss:2.2574746148859335
train loss:2.283248975395685
train loss:2.2529112240595106
=== epoch:35, train acc:0.23666666666666666, test acc:0.1837 ===
train loss:2.2622975417300513
train loss:2.2624054267357185
train loss:2.249895615444449
=== epoch:36, train acc:0.24, test acc:0.1849 ===
train loss:2.2670328726038598
train loss:2.2621533687290696
train loss:2.2588617512240075
=== epoch:37, train acc:0.24333333333333335, test acc:0.1867 ===
train loss:2.259179399059643
train loss:2.253242469348756
train loss:2.247486517750717
=== epoch:38, train acc:0.25, test acc:0.1879 ===
train loss:2.2533742735867084
train loss:2.254941007058005
train loss:2.2407765820346395
=== epoch:39, train acc:0.25333333333333335, test acc:0.1916 ===
train loss:2.258613534954242
train loss:2.247623559283962
train loss:2.268959664046296
=== epoch:40, train acc:0.26, test acc:0.1933 ===
train loss:2.249891330378538
train loss:2.256890796810187
train loss:2.247667066989199
=== epoch:41, train acc:0.26, test acc:0.1953 ===
train loss:2.2704824088637743
train loss:2.2488045143502355
train loss:2.2437003127282664
=== epoch:42, train acc:0.26666666666666666, test acc:0.1973 ===
train loss:2.2523807950539116
train loss:2.2397836718422135
train loss:2.24990683008903
=== epoch:43, train acc:0.27, test acc:0.2005 ===
train loss:2.2578519128744285
train loss:2.245957259608089
train loss:2.2455337022932746
=== epoch:44, train acc:0.2833333333333333, test acc:0.2022 ===
train loss:2.2368133046152505
train loss:2.262788080324759
train loss:2.2458886315206934
=== epoch:45, train acc:0.2733333333333333, test acc:0.2061 ===
train loss:2.2591430991965438
train loss:2.234241043993578
train loss:2.2586311029223025
=== epoch:46, train acc:0.28, test acc:0.2053 ===
train loss:2.222956956758696
```

```
train loss:2.228936803838681
train loss:2.249675717408843
=== epoch:47, train acc:0.2833333333333333, test acc:0.2077 ===
train loss:2.2629887629670034
train loss:2.245758133203707
train loss:2.2467057453249604
=== epoch:48, train acc:0.29, test acc:0.2129 ===
train loss:2.2543647819232944
train loss:2.2453637517095584
train loss:2.240164806992922
=== epoch:49, train acc:0.2866666666666667, test acc:0.2129 ===
train loss:2.254486378612578
train loss:2.2201123098316953
train loss:2.2358391483161792
=== epoch:50, train acc:0.29, test acc:0.217 ===
train loss:2.2306948543954297
train loss:2.2387434363935004
train loss:2.2473820654757386
=== epoch:51, train acc:0.3, test acc:0.2189 ===
train loss:2.236813776616849
train loss:2.2327346416822356
train loss:2.2313009134611725
=== epoch:52, train acc:0.3, test acc:0.2214 ===
train loss:2.2327784127447683
train loss:2.2466330516219633
train loss:2.2349343107036757
=== epoch:53, train acc:0.30666666666666664, test acc:0.22 ===
train loss:2.2120717657043683
train loss:2.214069398395574
train loss:2.2449798731115522
=== epoch:54, train acc:0.31, test acc:0.2226 ===
train loss:2.231670010271403
train loss:2.208174297828759
train loss:2.2310706224590025
=== epoch:55, train acc:0.30666666666666664, test acc:0.2245 ===
train loss:2.248968844804475
train loss:2.2379922462270883
train loss:2.2311563585547827
=== epoch:56, train acc:0.31, test acc:0.2277 ===
train loss:2.2365131425041938
train loss:2.2320453124203565
train loss:2.2284694205679587
=== epoch:57, train acc:0.31333333333333335, test acc:0.2303 ===
train loss:2.240106986259587
train loss:2.2072662986245453
train loss:2.240476220852277
=== epoch:58, train acc:0.32, test acc:0.2331 ===
train loss:2.2022581701275707
train loss:2.2305626728040977
train loss:2.232550576319791
=== epoch:59, train acc:0.31333333333333335, test acc:0.2355 ===
train loss:2.2164690108743788
train loss:2.2384948349091247
train loss:2.2174577187363536
=== epoch:60, train acc:0.31666666666666665, test acc:0.2377 ===
train loss:2.203436636070427
train loss:2.2404318985893332
train loss:2.231574927495129
=== epoch:61, train acc:0.31666666666666665, test acc:0.2385 ===
train loss:2.247779013772617
train loss:2.2123116139375227
```

```
train loss:2.2117629419841047
=== epoch:62, train acc:0.32, test acc:0.2404 ===
train loss:2.231018472985949
train loss:2.2134659140320823
train loss:2.2011576165551014
=== epoch:63, train acc:0.31666666666666665, test acc:0.2398 ===
train loss:2.2251669704074115
train loss:2.2128283050378745
train loss:2.228705887045967
=== epoch:64, train acc:0.3233333333333333, test acc:0.2403 ===
train loss:2.2266405273524708
train loss:2.2283060698461337
train loss:2.220345354242907
=== epoch:65, train acc:0.32666666666666666, test acc:0.241 ===
train loss:2.237107954598329
train loss:2.22430519050207
train loss:2.22021881929206
=== epoch:66, train acc:0.3333333333333333, test acc:0.244 ===
train loss:2.2156404558751897
train loss:2.208203764009914
train loss:2.231787058600594
=== epoch:67, train acc:0.33666666666666667, test acc:0.2443 ===
train loss:2.1945454470826498
train loss:2.2150530940962994
train loss:2.225977097337029
=== epoch:68, train acc:0.34, test acc:0.2475 ===
train loss:2.1945139231452306
train loss:2.2097811970283523
train loss:2.23011405625922
=== epoch:69, train acc:0.34, test acc:0.2485 ===
train loss:2.189245972540314
train loss:2.2321913926930774
train loss:2.2128769501911303
=== epoch:70, train acc:0.34, test acc:0.2505 ===
train loss:2.209266847309963
train loss:2.176589871637686
train loss:2.229902944427294
=== epoch:71, train acc:0.34, test acc:0.2521 ===
train loss:2.1982242913214107
train loss:2.1840348027463548
train loss:2.214237344426824
=== epoch:72, train acc:0.3466666666666667, test acc:0.2539 ===
train loss:2.191702721981008
train loss:2.2241854594834023
train loss:2.2197084818423574
=== epoch:73, train acc:0.35, test acc:0.254 ===
train loss:2.181630354147048
train loss:2.2046671667545423
train loss:2.201394397955632
=== epoch:74, train acc:0.35, test acc:0.2575 ===
train loss:2.178275874617148
train loss:2.2006772537541943
train loss:2.2064814714724084
=== epoch:75, train acc:0.3566666666666667, test acc:0.2595 ===
train loss:2.2012985610647666
train loss:2.193641728229958
train loss:2.216273029760619
=== epoch:76, train acc:0.35, test acc:0.2603 ===
train loss:2.2092634712797166
train loss:2.2058755765889906
train loss:2.197699747531764
```

=== epoch:77, train acc:0.35333333333333333, test acc:0.2615 ===
train loss:2.180238675074255
train loss:2.1814282623429757
train loss:2.205994926510995
=== epoch:78, train acc:0.36, test acc:0.2652 ===
train loss:2.167356653829178
train loss:2.187012304756953
train loss:2.1963325293936076
=== epoch:79, train acc:0.36333333333333334, test acc:0.2672 ===
train loss:2.1863307494982736
train loss:2.1746856847859486
train loss:2.213334870217395
=== epoch:80, train acc:0.36, test acc:0.2689 ===
train loss:2.165857739306088
train loss:2.20828948841151
train loss:2.1832698268212463
=== epoch:81, train acc:0.37, test acc:0.2684 ===
train loss:2.133199185867035
train loss:2.2134738164785737
train loss:2.185963233861319
=== epoch:82, train acc:0.36333333333333334, test acc:0.2672 ===
train loss:2.1610250867454335
train loss:2.184446294636514
train loss:2.1626425763014865
=== epoch:83, train acc:0.36666666666666664, test acc:0.2665 ===
train loss:2.2136047062933715
train loss:2.191795617877527
train loss:2.1528031667063643
=== epoch:84, train acc:0.36666666666666664, test acc:0.2703 ===
train loss:2.173837084499662
train loss:2.170769218481603
train loss:2.1540714331212274
=== epoch:85, train acc:0.36666666666666664, test acc:0.2706 ===
train loss:2.1780299712930584
train loss:2.1609687650636777
train loss:2.1486965443000785
=== epoch:86, train acc:0.37333333333333335, test acc:0.2709 ===
train loss:2.135846031839616
train loss:2.1395748147344746
train loss:2.1635213850066024
=== epoch:87, train acc:0.36333333333333334, test acc:0.2699 ===
train loss:2.2219853369711395
train loss:2.155097212159508
train loss:2.1438370538022844
=== epoch:88, train acc:0.36333333333333334, test acc:0.27 ===
train loss:2.1513360814796774
train loss:2.169221200481271
train loss:2.1676650716010823
=== epoch:89, train acc:0.36, test acc:0.2701 ===
train loss:2.1210169377858117
train loss:2.136203346860075
train loss:2.13054361059659
=== epoch:90, train acc:0.36333333333333334, test acc:0.2706 ===
train loss:2.1293628025675355
train loss:2.1323041786814345
train loss:2.186175816479871
=== epoch:91, train acc:0.36, test acc:0.2707 ===
train loss:2.1869058183399033
train loss:2.1414327108739117
train loss:2.1403571933673127
=== epoch:92, train acc:0.36, test acc:0.2686 ===

```
train loss:2.1896068773217405
train loss:2.1565571282789757
train loss:2.1745130760527744
=== epoch:93, train acc:0.37333333333333335, test acc:0.2726 ===
train loss:2.1266097907382098
train loss:2.118985398484502
train loss:2.1429953747649866
=== epoch:94, train acc:0.36666666666666664, test acc:0.2709 ===
train loss:2.1162481521891277
train loss:2.152306315688107
train loss:2.1388200459142026
=== epoch:95, train acc:0.37, test acc:0.2708 ===
train loss:2.1571580145616904
train loss:2.1357840464698232
train loss:2.1495652115848416
=== epoch:96, train acc:0.37, test acc:0.272 ===
train loss:2.1843436786685766
train loss:2.1716691123020815
train loss:2.1373181165144626
=== epoch:97, train acc:0.37333333333333335, test acc:0.2738 ===
train loss:2.136421507622148
train loss:2.107752786400467
train loss:2.1437163283578053
=== epoch:98, train acc:0.37333333333333335, test acc:0.2736 ===
train loss:2.145714789986651
train loss:2.167947929353382
train loss:2.115169835949725
=== epoch:99, train acc:0.37666666666666665, test acc:0.2773 ===
train loss:2.1780990883264986
train loss:2.141827623419385
train loss:2.1459647961940895
=== epoch:100, train acc:0.37666666666666665, test acc:0.278 ===
train loss:2.10069193433242
train loss:2.10529606890214
train loss:2.145674583179772
=== epoch:101, train acc:0.37666666666666665, test acc:0.2764 ===
train loss:2.124901235300126
train loss:2.1137755339491924
train loss:2.139267445137226
=== epoch:102, train acc:0.37666666666666665, test acc:0.2775 ===
train loss:2.123183688794261
train loss:2.1779798602167104
train loss:2.104651215866207
=== epoch:103, train acc:0.37666666666666665, test acc:0.2791 ===
train loss:2.134380708915295
train loss:2.1272637413473707
train loss:2.1287629504726184
=== epoch:104, train acc:0.37666666666666665, test acc:0.2792 ===
train loss:2.1030514390000783
train loss:2.118164627057271
train loss:2.124158595675108
=== epoch:105, train acc:0.37666666666666665, test acc:0.2805 ===
train loss:2.127945843099574
train loss:2.116919386437551
train loss:2.15211923327773
=== epoch:106, train acc:0.37666666666666665, test acc:0.2816 ===
train loss:2.1228520004854214
train loss:2.1531099601497408
train loss:2.088836316377165
=== epoch:107, train acc:0.37666666666666665, test acc:0.2827 ===
train loss:2.0739639840115016
```

```
train loss:2.0809786881902865
train loss:2.1089833193726673
=== epoch:108, train acc:0.37666666666666665, test acc:0.2818 ===
train loss:2.100121154569872
train loss:2.08434224585347
train loss:2.076249154592536
=== epoch:109, train acc:0.37666666666666665, test acc:0.2846 ===
train loss:2.138235022532617
train loss:2.082355661666623
train loss:2.074983908103446
=== epoch:110, train acc:0.37666666666666665, test acc:0.2876 ===
train loss:2.140514280938427
train loss:2.121031602410614
train loss:2.07049448551188
=== epoch:111, train acc:0.37666666666666665, test acc:0.2883 ===
train loss:2.1330693282454107
train loss:2.103974284301976
train loss:2.094920490977927
=== epoch:112, train acc:0.37666666666666665, test acc:0.2909 ===
train loss:2.085128522963577
train loss:2.084155008658898
train loss:2.0782318962152893
=== epoch:113, train acc:0.37666666666666665, test acc:0.2913 ===
train loss:2.0905232147244113
train loss:2.105097377926383
train loss:2.132134530449913
=== epoch:114, train acc:0.37666666666666665, test acc:0.2912 ===
train loss:2.0893305480304396
train loss:2.033906427703011
train loss:2.132713222368998
=== epoch:115, train acc:0.38666666666666666, test acc:0.2915 ===
train loss:2.0307767395986485
train loss:2.055616347132545
train loss:2.0932661508024997
=== epoch:116, train acc:0.38, test acc:0.2905 ===
train loss:2.083538747634126
train loss:2.0871128866573034
train loss:2.061687430765731
=== epoch:117, train acc:0.38333333333333336, test acc:0.2907 ===
train loss:2.017599563151018
train loss:2.1271040201485767
train loss:2.0730842231670046
=== epoch:118, train acc:0.38333333333333336, test acc:0.2921 ===
train loss:2.049924315176204
train loss:2.084346847922379
train loss:2.093901064020835
=== epoch:119, train acc:0.38333333333333336, test acc:0.2918 ===
train loss:2.1075371846210027
train loss:2.0153969150476057
train loss:2.0694832123918414
=== epoch:120, train acc:0.38666666666666666, test acc:0.2932 ===
train loss:2.0497482057127763
train loss:2.0253302107827755
train loss:2.171133021647678
=== epoch:121, train acc:0.38666666666666666, test acc:0.2946 ===
train loss:2.0408664269443157
train loss:2.0458376541495227
train loss:2.0666756494108065
=== epoch:122, train acc:0.4, test acc:0.296 ===
train loss:2.0388988286452703
train loss:2.0195411534736447
```

```
train loss:2.0870410633720438
=== epoch:123, train acc:0.39666666666666667, test acc:0.297 ===
train loss:2.1065908292712763
train loss:2.0643881965477306
train loss:2.0390656631892603
=== epoch:124, train acc:0.39666666666666667, test acc:0.2973 ===
train loss:2.0182127632009874
train loss:2.0594335311104355
train loss:2.023359138726855
=== epoch:125, train acc:0.39666666666666667, test acc:0.297 ===
train loss:1.9826138918041791
train loss:2.024198173161602
train loss:2.05024663599046
=== epoch:126, train acc:0.4, test acc:0.2972 ===
train loss:2.076031377941603
train loss:2.044183425208949
train loss:2.101608317062335
=== epoch:127, train acc:0.41, test acc:0.2982 ===
train loss:1.9969917179597758
train loss:2.0089532894489794
train loss:2.0945686001829666
=== epoch:128, train acc:0.41, test acc:0.2988 ===
train loss:2.0915268886694216
train loss:2.072352134576801
train loss:2.003426824021395
=== epoch:129, train acc:0.4166666666666667, test acc:0.3059 ===
train loss:2.0860832349315923
train loss:2.0211463704620636
train loss:1.9820403094489263
=== epoch:130, train acc:0.42, test acc:0.3091 ===
train loss:1.9682435400603964
train loss:1.994428807291368
train loss:2.017806930743705
=== epoch:131, train acc:0.41333333333333333, test acc:0.3063 ===
train loss:2.080331827784879
train loss:2.0010969828290186
train loss:2.022092749768271
=== epoch:132, train acc:0.42333333333333334, test acc:0.312 ===
train loss:2.1052754756390804
train loss:1.9965313104307567
train loss:2.0040488177322864
=== epoch:133, train acc:0.4266666666666667, test acc:0.3136 ===
train loss:1.9224892914674312
train loss:2.0029194506628865
train loss:2.0332973609258724
=== epoch:134, train acc:0.43, test acc:0.3149 ===
train loss:1.9895067359495997
train loss:1.9722918128832712
train loss:2.0218221744310663
=== epoch:135, train acc:0.43333333333333335, test acc:0.3168 ===
train loss:2.0375256793239553
train loss:1.9624881024770664
train loss:1.9970082526912902
=== epoch:136, train acc:0.43, test acc:0.3161 ===
train loss:1.9071749533118196
train loss:1.9495676773131563
train loss:1.9427318714183024
=== epoch:137, train acc:0.4266666666666667, test acc:0.3145 ===
train loss:2.0485491676073573
train loss:2.0190723516321505
train loss:2.0503322083770446
```

```
=== epoch:138, train acc:0.4266666666666667, test acc:0.3183 ===
train loss:1.9068774972522449
train loss:2.043070759262321
train loss:2.030819737086666
=== epoch:139, train acc:0.43, test acc:0.3213 ===
train loss:2.0082380224687655
train loss:1.9048421564007365
train loss:1.9868865322809819
=== epoch:140, train acc:0.43, test acc:0.3228 ===
train loss:1.9606151841970219
train loss:1.9788873153994126
train loss:1.9312944817914774
=== epoch:141, train acc:0.43, test acc:0.3238 ===
train loss:1.970039851077012
train loss:1.9395630459199487
train loss:1.919286404122869
=== epoch:142, train acc:0.4266666666666667, test acc:0.3209 ===
train loss:1.9953231799925815
train loss:1.9718620451478757
train loss:1.9412270293649376
=== epoch:143, train acc:0.4266666666666667, test acc:0.3235 ===
train loss:1.8853875304722099
train loss:1.9565685151596304
train loss:1.964097177434063
=== epoch:144, train acc:0.4266666666666667, test acc:0.3235 ===
train loss:1.9461219277274047
train loss:1.8821691461400427
train loss:1.944036506499269
=== epoch:145, train acc:0.4266666666666667, test acc:0.3241 ===
train loss:1.8971360697462227
train loss:1.951830379679794
train loss:1.9410498156636375
=== epoch:146, train acc:0.43, test acc:0.3244 ===
train loss:1.9956665230819508
train loss:1.8957443558418903
train loss:1.9156289575768006
=== epoch:147, train acc:0.43333333333333335, test acc:0.3245 ===
train loss:1.938402577170696
train loss:1.923679607694646
train loss:1.9509386453225905
=== epoch:148, train acc:0.44333333333333336, test acc:0.3276 ===
train loss:1.8648072358117036
train loss:1.9385239006006083
train loss:1.829007137174299
=== epoch:149, train acc:0.44, test acc:0.3306 ===
train loss:1.9731857988750088
train loss:1.8787589139139103
train loss:1.9085524346504612
=== epoch:150, train acc:0.43666666666666665, test acc:0.329 ===
train loss:1.9149361944903753
train loss:1.8555522782054275
train loss:1.9670544389159899
=== epoch:151, train acc:0.43333333333333335, test acc:0.3302 ===
train loss:1.8991626522408578
train loss:1.9671371426890878
train loss:1.8746604677484127
=== epoch:152, train acc:0.43333333333333335, test acc:0.3316 ===
train loss:1.8848841537745509
train loss:1.9855028399121337
train loss:1.86251804223826
=== epoch:153, train acc:0.44, test acc:0.3364 ===
```

```
train loss:1.8612438011658048
train loss:1.900514756634074
train loss:1.9497803755275087
=== epoch:154, train acc:0.44, test acc:0.3372 ===
train loss:1.874318510006663
train loss:1.8487472555081157
train loss:1.845850742639374
=== epoch:155, train acc:0.44333333333333336, test acc:0.3426 ===
train loss:1.9110121298349783
train loss:1.91263465196669
train loss:1.8779332626107177
=== epoch:156, train acc:0.44333333333333336, test acc:0.3414 ===
train loss:1.8577232727713189
train loss:1.8792238124361003
train loss:1.9033427951892596
=== epoch:157, train acc:0.44333333333333336, test acc:0.3443 ===
train loss:1.8495938400176988
train loss:1.8928944255262952
train loss:1.8219448713943533
=== epoch:158, train acc:0.44333333333333336, test acc:0.3446 ===
train loss:1.943519381447387
train loss:1.9239985898629937
train loss:1.8780054171818992
=== epoch:159, train acc:0.44666666666666666, test acc:0.3528 ===
train loss:1.7509194833599897
train loss:1.9589477275353504
train loss:1.8267850775070427
=== epoch:160, train acc:0.44333333333333336, test acc:0.3535 ===
train loss:1.8161705320654375
train loss:1.8077639298505188
train loss:1.8321940452913277
=== epoch:161, train acc:0.44333333333333336, test acc:0.3569 ===
train loss:1.7990097556738938
train loss:1.863737606074631
train loss:1.8678210851851935
=== epoch:162, train acc:0.44, test acc:0.3578 ===
train loss:1.834643614935043
train loss:1.7643975943197094
train loss:1.909453328725977
=== epoch:163, train acc:0.43666666666666665, test acc:0.3602 ===
train loss:1.8336880743171877
train loss:1.8258596219442842
train loss:1.7631824680290018
=== epoch:164, train acc:0.43666666666666665, test acc:0.3595 ===
train loss:1.7519686727794674
train loss:1.781297722754459
train loss:1.8058772550522648
=== epoch:165, train acc:0.43666666666666665, test acc:0.3593 ===
train loss:1.8647998704501247
train loss:1.793448828088624
train loss:1.9186107178532381
=== epoch:166, train acc:0.46, test acc:0.3655 ===
train loss:1.7449722008676714
train loss:1.8143708673840642
train loss:1.8262517277483812
=== epoch:167, train acc:0.4666666666666667, test acc:0.3685 ===
train loss:1.8673199752894405
train loss:1.7164652124555246
train loss:1.771392244628406
=== epoch:168, train acc:0.47333333333333333, test acc:0.3693 ===
train loss:1.8239060602134114
```

```
train loss:1.8360233175430178
train loss:1.7571416293211461
=== epoch:169, train acc:0.48, test acc:0.3712 ===
train loss:1.8692933149925448
train loss:1.8353859184089691
train loss:1.731120863526684
=== epoch:170, train acc:0.48, test acc:0.3758 ===
train loss:1.686311558843458
train loss:1.8247101090737474
train loss:1.663157837890984
=== epoch:171, train acc:0.47333333333333333, test acc:0.3738 ===
train loss:1.7968381127197637
train loss:1.7830320348760953
train loss:1.826336163556094
=== epoch:172, train acc:0.4766666666666667, test acc:0.3785 ===
train loss:1.722506314653719
train loss:1.8032776022220742
train loss:1.801556646146871
=== epoch:173, train acc:0.48333333333333334, test acc:0.3828 ===
train loss:1.7675268241339992
train loss:1.75140722564387
train loss:1.851109060838458
=== epoch:174, train acc:0.49, test acc:0.3886 ===
train loss:1.651648527361461
train loss:1.7654359574530505
train loss:1.7976802452518859
=== epoch:175, train acc:0.48333333333333334, test acc:0.3848 ===
train loss:1.7381087519080578
train loss:1.8074650752076007
train loss:1.8354123492170251
=== epoch:176, train acc:0.48333333333333334, test acc:0.3875 ===
train loss:1.6816927869415803
train loss:1.837948925265058
train loss:1.8197879679307127
=== epoch:177, train acc:0.49, test acc:0.3889 ===
train loss:1.7782287870758864
train loss:1.7288460980407194
train loss:1.7850543229044087
=== epoch:178, train acc:0.5, test acc:0.3917 ===
train loss:1.7012075538122622
train loss:1.5073768183658487
train loss:1.8349949568985655
=== epoch:179, train acc:0.5033333333333333, test acc:0.3939 ===
train loss:1.6878720387221633
train loss:1.7798215761972835
train loss:1.7164386645425032
=== epoch:180, train acc:0.5, test acc:0.3921 ===
train loss:1.7041270358153202
train loss:1.7435848443107984
train loss:1.7295246508551927
=== epoch:181, train acc:0.5033333333333333, test acc:0.3958 ===
train loss:1.8613056970774684
train loss:1.7881524370579793
train loss:1.720722457707567
=== epoch:182, train acc:0.5066666666666667, test acc:0.399 ===
train loss:1.6778067015990303
train loss:1.7829789161424916
train loss:1.670981518907206
=== epoch:183, train acc:0.5033333333333333, test acc:0.4017 ===
train loss:1.8029589435022608
train loss:1.639963239755873
```

```
train loss:1.7716996574614574
=== epoch:184, train acc:0.51, test acc:0.4061 ===
train loss:1.6827613393514034
train loss:1.6462675926409096
train loss:1.7897547738801836
=== epoch:185, train acc:0.51, test acc:0.4054 ===
train loss:1.7244268988466305
train loss:1.6794946556009864
train loss:1.6786125330311241
=== epoch:186, train acc:0.51, test acc:0.4047 ===
train loss:1.709813926916945
train loss:1.6749336540043331
train loss:1.6444440493716395
=== epoch:187, train acc:0.51, test acc:0.4095 ===
train loss:1.6901316489177902
train loss:1.6258524558864627
train loss:1.6650110792321604
=== epoch:188, train acc:0.5133333333333333, test acc:0.4104 ===
train loss:1.7723278387054133
train loss:1.7831826974900822
train loss:1.736583402227726
=== epoch:189, train acc:0.52, test acc:0.4153 ===
train loss:1.7243797052821792
train loss:1.8635439556015725
train loss:1.5796871948070126
=== epoch:190, train acc:0.5333333333333333, test acc:0.418 ===
train loss:1.6623420039244914
train loss:1.7016542422708738
train loss:1.7356511566519728
=== epoch:191, train acc:0.5266666666666666, test acc:0.422 ===
train loss:1.7062698202666835
train loss:1.7086935022678282
train loss:1.7291357208964806
=== epoch:192, train acc:0.5266666666666666, test acc:0.4254 ===
train loss:1.6711756315186972
train loss:1.6327873968630104
train loss:1.6959206067651207
=== epoch:193, train acc:0.5266666666666666, test acc:0.4244 ===
train loss:1.6778762980027775
train loss:1.6157384178400802
train loss:1.7085513106943815
=== epoch:194, train acc:0.5333333333333333, test acc:0.4237 ===
train loss:1.7482967343734137
train loss:1.5849192855503094
train loss:1.6936581362689833
=== epoch:195, train acc:0.5366666666666666, test acc:0.4274 ===
train loss:1.7152920076647091
train loss:1.632362777120693
train loss:1.7358648375317285
=== epoch:196, train acc:0.54, test acc:0.4283 ===
train loss:1.6243867995951407
train loss:1.6442338912547658
train loss:1.7240110002874953
=== epoch:197, train acc:0.5333333333333333, test acc:0.4267 ===
train loss:1.5428374199380996
train loss:1.6229867062375798
train loss:1.6876764398303175
=== epoch:198, train acc:0.5366666666666666, test acc:0.4259 ===
train loss:1.6755693108077745
train loss:1.595730123876183
train loss:1.5046518303420018
```

```
=== epoch:199, train acc:0.5333333333333333, test acc:0.4242 ===
train loss:1.5918511749789364
train loss:1.6262421887689382
train loss:1.6485144067161983
=== epoch:200, train acc:0.5466666666666666, test acc:0.428 ===
train loss:1.4631485979498877
train loss:1.6164756557992563
train loss:1.6090414535523672
=== epoch:201, train acc:0.5466666666666666, test acc:0.4316 ===
train loss:1.657305072285242
train loss:1.7097605787473968
train loss:1.657547343599202
=== epoch:202, train acc:0.5466666666666666, test acc:0.4346 ===
train loss:1.6319034680268916
train loss:1.555679045560186
train loss:1.6494659826349138
=== epoch:203, train acc:0.5533333333333333, test acc:0.4378 ===
train loss:1.6735395811806495
train loss:1.694376670203009
train loss:1.5707392516307224
=== epoch:204, train acc:0.5566666666666666, test acc:0.4419 ===
train loss:1.5844265380416158
train loss:1.5096792868405107
train loss:1.511955675525893
=== epoch:205, train acc:0.5533333333333333, test acc:0.442 ===
train loss:1.52480319412095
train loss:1.6696999601967875
train loss:1.5078975029309973
=== epoch:206, train acc:0.55, test acc:0.4416 ===
train loss:1.6611845668964893
train loss:1.5781575892832607
train loss:1.523912235210865
=== epoch:207, train acc:0.5566666666666666, test acc:0.4451 ===
train loss:1.6679421386258966
train loss:1.5103637389578644
train loss:1.6755019970965461
=== epoch:208, train acc:0.5533333333333333, test acc:0.4495 ===
train loss:1.5014021658940664
train loss:1.607813314928856
train loss:1.5198533105333234
=== epoch:209, train acc:0.56, test acc:0.4519 ===
train loss:1.4581950031945268
train loss:1.6363156742218559
train loss:1.6303501907431077
=== epoch:210, train acc:0.5666666666666667, test acc:0.4541 ===
train loss:1.6162799641572403
train loss:1.528322036811921
train loss:1.5645230114314734
=== epoch:211, train acc:0.57, test acc:0.4568 ===
train loss:1.7125762356035483
train loss:1.627258160405607
train loss:1.6805479655254294
=== epoch:212, train acc:0.5766666666666667, test acc:0.461 ===
train loss:1.5614200679860089
train loss:1.5296029921545875
train loss:1.4217465198513872
=== epoch:213, train acc:0.58, test acc:0.4614 ===
train loss:1.4584725067734732
train loss:1.6451437385362297
train loss:1.6127904239492634
=== epoch:214, train acc:0.58, test acc:0.4612 ===
```

```
train loss:1.6029260321451482
train loss:1.456196597131088
train loss:1.4649867329843869
=== epoch:215, train acc:0.5733333333333334, test acc:0.4604 ===
train loss:1.4121363841886676
train loss:1.6190467931317907
train loss:1.635359114473035
=== epoch:216, train acc:0.5733333333333334, test acc:0.4606 ===
train loss:1.4881906514793826
train loss:1.6429552288502278
train loss:1.4965347403190339
=== epoch:217, train acc:0.5766666666666667, test acc:0.4663 ===
train loss:1.5356503816877036
train loss:1.5878253144653645
train loss:1.616513269394687
=== epoch:218, train acc:0.5833333333333334, test acc:0.4669 ===
train loss:1.4094760811294063
train loss:1.5315387913965524
train loss:1.481291292498078
=== epoch:219, train acc:0.5833333333333334, test acc:0.4646 ===
train loss:1.486132431102844
train loss:1.4674464909984388
train loss:1.6648192319535362
=== epoch:220, train acc:0.5866666666666667, test acc:0.4667 ===
train loss:1.5917942811035775
train loss:1.623724956044348
train loss:1.5543725346573984
=== epoch:221, train acc:0.5833333333333334, test acc:0.4718 ===
train loss:1.4376415627728811
train loss:1.5345723234131932
train loss:1.5733983683798918
=== epoch:222, train acc:0.58, test acc:0.4733 ===
train loss:1.5420796925223448
train loss:1.5055808609756978
train loss:1.5681149346639112
=== epoch:223, train acc:0.58, test acc:0.4744 ===
train loss:1.479872747579446
train loss:1.499778138714185
train loss:1.5580150920162652
=== epoch:224, train acc:0.5833333333333334, test acc:0.478 ===
train loss:1.409883549923255
train loss:1.6102658543388984
train loss:1.4423528169845437
=== epoch:225, train acc:0.5833333333333334, test acc:0.4779 ===
train loss:1.5432657342490623
train loss:1.4188541842418911
train loss:1.6059647502549075
=== epoch:226, train acc:0.5833333333333334, test acc:0.477 ===
train loss:1.3912771628841318
train loss:1.4300122878777788
train loss:1.4978587086395252
=== epoch:227, train acc:0.5833333333333334, test acc:0.4786 ===
train loss:1.563694544653839
train loss:1.4388622479232598
train loss:1.4868839848961728
=== epoch:228, train acc:0.5933333333333334, test acc:0.4804 ===
train loss:1.5698933091175302
train loss:1.4362801209692089
train loss:1.3117389931019547
=== epoch:229, train acc:0.59, test acc:0.4798 ===
train loss:1.627591315774281
```

```
train loss:1.529241359834925
train loss:1.4640083750668547
=== epoch:230, train acc:0.5966666666666667, test acc:0.4833 ===
train loss:1.5760679420887795
train loss:1.4324450471545518
train loss:1.592123079945782
=== epoch:231, train acc:0.5966666666666667, test acc:0.4853 ===
train loss:1.4755663746793073
train loss:1.4294715958895123
train loss:1.5980396752970858
=== epoch:232, train acc:0.5966666666666667, test acc:0.4878 ===
train loss:1.5161016155632114
train loss:1.4606502110460784
train loss:1.4397633695346976
=== epoch:233, train acc:0.5966666666666667, test acc:0.4902 ===
train loss:1.5385670642162006
train loss:1.4751708550607012
train loss:1.3954419195585437
=== epoch:234, train acc:0.6066666666666667, test acc:0.4914 ===
train loss:1.5096913203268238
train loss:1.3558539418999072
train loss:1.4157901308278897
=== epoch:235, train acc:0.6033333333333334, test acc:0.496 ===
train loss:1.3590444496603125
train loss:1.2694058142685294
train loss:1.6003143439721208
=== epoch:236, train acc:0.61, test acc:0.4925 ===
train loss:1.477281168593999
train loss:1.4341444792812883
train loss:1.5669903065580955
=== epoch:237, train acc:0.6133333333333333, test acc:0.4941 ===
train loss:1.4357021126740548
train loss:1.4381279642245344
train loss:1.5137010668470254
=== epoch:238, train acc:0.6166666666666667, test acc:0.4959 ===
train loss:1.4628133361381066
train loss:1.4625908812206754
train loss:1.4410826582255643
=== epoch:239, train acc:0.6166666666666667, test acc:0.4939 ===
train loss:1.493846985506153
train loss:1.4885242497837783
train loss:1.4600581548211284
=== epoch:240, train acc:0.6166666666666667, test acc:0.4928 ===
train loss:1.4621277980082996
train loss:1.3474370600416592
train loss:1.5271540077345243
=== epoch:241, train acc:0.62, test acc:0.4932 ===
train loss:1.6200439948547858
train loss:1.5101871465151329
train loss:1.4043282330559372
=== epoch:242, train acc:0.6133333333333333, test acc:0.4956 ===
train loss:1.370642650589147
train loss:1.3904241836040918
train loss:1.3181735986293972
=== epoch:243, train acc:0.6133333333333333, test acc:0.4933 ===
train loss:1.4427755412084915
train loss:1.4176316211977416
train loss:1.3596819475474171
=== epoch:244, train acc:0.62, test acc:0.4941 ===
train loss:1.3856265601968079
train loss:1.390152966041148
```

```
train loss:1.4587440527933424
=== epoch:245, train acc:0.6233333333333333, test acc:0.4991 ===
train loss:1.58952961586632
train loss:1.4707709186179592
train loss:1.4488052268931242
=== epoch:246, train acc:0.62, test acc:0.499 ===
train loss:1.4043825775051635
train loss:1.4650628100392589
train loss:1.3832041973843763
=== epoch:247, train acc:0.6166666666666667, test acc:0.502 ===
train loss:1.5534555306264237
train loss:1.4786836695376708
train loss:1.4781212997784823
=== epoch:248, train acc:0.6233333333333333, test acc:0.5049 ===
train loss:1.268035375025152
train loss:1.4293120223174522
train loss:1.3508572257983749
=== epoch:249, train acc:0.63, test acc:0.507 ===
train loss:1.4879923739950147
train loss:1.3887185191889233
train loss:1.5549914013002846
=== epoch:250, train acc:0.63, test acc:0.5083 ===
train loss:1.341191354517398
train loss:1.1524690971788218
train loss:1.341045043807285
=== epoch:251, train acc:0.6233333333333333, test acc:0.5061 ===
train loss:1.2709083535883685
train loss:1.4276478180579886
train loss:1.2357029133212898
=== epoch:252, train acc:0.62, test acc:0.5089 ===
train loss:1.3160007522305066
train loss:1.1444118067356492
train loss:1.5016176162485693
=== epoch:253, train acc:0.62, test acc:0.5109 ===
train loss:1.2805204306723152
train loss:1.4594550438385319
train loss:1.2760159384759342
=== epoch:254, train acc:0.6233333333333333, test acc:0.513 ===
train loss:1.2263614055350784
train loss:1.315888889294026
train loss:1.389463637558941
=== epoch:255, train acc:0.6266666666666667, test acc:0.5118 ===
train loss:1.4997616335511907
train loss:1.4775674959075735
train loss:1.394098991800057
=== epoch:256, train acc:0.6233333333333333, test acc:0.5104 ===
train loss:1.3744833120984907
train loss:1.3118103222000892
train loss:1.2380510276210484
=== epoch:257, train acc:0.6266666666666667, test acc:0.5118 ===
train loss:1.35066450456144
train loss:1.4651622281039405
train loss:1.4143904040087485
=== epoch:258, train acc:0.6266666666666667, test acc:0.5131 ===
train loss:1.3194570745500178
train loss:1.1685561402886828
train loss:1.4337578889872196
=== epoch:259, train acc:0.63, test acc:0.514 ===
train loss:1.3125739060889794
train loss:1.3090392661851609
train loss:1.4616375222484417
```

```
=== epoch:260, train acc:0.6333333333333333, test acc:0.5142 ===
train loss:1.2792808150764539
train loss:1.4973884826240549
train loss:1.1570986656013584
=== epoch:261, train acc:0.6333333333333333, test acc:0.5159 ===
train loss:1.2360936718833548
train loss:1.2673324028665487
train loss:1.3983416701983344
=== epoch:262, train acc:0.6333333333333333, test acc:0.5161 ===
train loss:1.3568766983189124
train loss:1.51953208055084
train loss:1.3407403988367819
=== epoch:263, train acc:0.6333333333333333, test acc:0.5209 ===
train loss:1.2892582469480223
train loss:1.1590885232308545
train loss:1.2105767417791444
=== epoch:264, train acc:0.6333333333333333, test acc:0.52 ===
train loss:1.4112004281047839
train loss:1.249660674985193
train loss:1.3378797401348712
=== epoch:265, train acc:0.6333333333333333, test acc:0.5193 ===
train loss:1.2434904702648655
train loss:1.1982376565698565
train loss:1.4409099394128027
=== epoch:266, train acc:0.6366666666666667, test acc:0.5197 ===
train loss:1.1913454839312718
train loss:1.3330182874223282
train loss:1.2565606898504746
=== epoch:267, train acc:0.66, test acc:0.5197 ===
train loss:1.281361344721476
train loss:1.147865640366727
train loss:1.163981259452073
=== epoch:268, train acc:0.6566666666666666, test acc:0.5197 ===
train loss:1.3050407778158462
train loss:1.1619954966943815
train loss:1.3516389669760789
=== epoch:269, train acc:0.6533333333333333, test acc:0.5199 ===
train loss:1.2279778768939327
train loss:1.2045699235228553
train loss:1.1854383839076568
=== epoch:270, train acc:0.6566666666666666, test acc:0.521 ===
train loss:1.341665105501327
train loss:1.2692589962812761
train loss:1.2917721548176386
=== epoch:271, train acc:0.6533333333333333, test acc:0.5222 ===
train loss:1.1781752965940457
train loss:1.298736468088618
train loss:1.390469090783325
=== epoch:272, train acc:0.65, test acc:0.5245 ===
train loss:1.419372286320349
train loss:1.2626271986670716
train loss:1.4427941600920926
=== epoch:273, train acc:0.66, test acc:0.5246 ===
train loss:1.2803669994551752
train loss:1.2157899788244215
train loss:1.254755257346305
=== epoch:274, train acc:0.66, test acc:0.5261 ===
train loss:1.2735615889856557
train loss:1.2444637807621133
train loss:1.1768399185723328
=== epoch:275, train acc:0.66, test acc:0.5264 ===
```

```
train loss:1.2619033611399286
train loss:1.1686176424593873
train loss:1.3758780429783155
=== epoch:276, train acc:0.6566666666666666, test acc:0.5303 ===
train loss:1.297787282179929
train loss:1.4100004998611348
train loss:1.2754717405791272
=== epoch:277, train acc:0.66, test acc:0.5331 ===
train loss:1.2135289260376156
train loss:1.292731160987047
train loss:1.2385992098865357
=== epoch:278, train acc:0.66, test acc:0.5343 ===
train loss:1.3544960491638227
train loss:1.2167287901148724
train loss:1.0984436108384212
=== epoch:279, train acc:0.6666666666666666, test acc:0.535 ===
train loss:1.2817858857115982
train loss:1.10463088923796
train loss:1.22116759022126
=== epoch:280, train acc:0.6633333333333333, test acc:0.5361 ===
train loss:1.1159045173352622
train loss:1.2087038199340692
train loss:1.2346311030028962
=== epoch:281, train acc:0.6633333333333333, test acc:0.5371 ===
train loss:1.3048945658090743
train loss:1.1524022575034114
train loss:1.208285521561405
=== epoch:282, train acc:0.66, test acc:0.5369 ===
train loss:1.291017320996441
train loss:1.1606354294634658
train loss:1.2174675890227875
=== epoch:283, train acc:0.66, test acc:0.5366 ===
train loss:1.2615897425843376
train loss:1.363437385593665
train loss:1.1208878630901955
=== epoch:284, train acc:0.6666666666666666, test acc:0.5398 ===
train loss:1.123443615871755
train loss:1.326286450720884
train loss:1.317411938096556
=== epoch:285, train acc:0.66, test acc:0.5395 ===
train loss:1.159788402428123
train loss:1.00345292331444
train loss:1.0294548312977447
=== epoch:286, train acc:0.6633333333333333, test acc:0.5397 ===
train loss:1.1947424231617358
train loss:1.1519514665515964
train loss:1.1564194134903514
=== epoch:287, train acc:0.6733333333333333, test acc:0.5418 ===
train loss:1.0998195639409252
train loss:1.1124627273402778
train loss:1.1093295600536204
=== epoch:288, train acc:0.6833333333333333, test acc:0.5438 ===
train loss:1.2438564422312728
train loss:1.0843781050699308
train loss:1.2365612664081147
=== epoch:289, train acc:0.6833333333333333, test acc:0.5436 ===
train loss:1.154932343200356
train loss:1.2437153423373553
train loss:1.2286979500859923
=== epoch:290, train acc:0.6933333333333334, test acc:0.5453 ===
train loss:1.2582088674212202
```

```
train loss:1.2486933815430965
train loss:1.1661344088445074
=== epoch:291, train acc:0.69, test acc:0.5468 ===
train loss:1.0637956121933334
train loss:1.1052771914603132
train loss:1.103153227178867
=== epoch:292, train acc:0.6933333333333334, test acc:0.5455 ===
train loss:1.15177289063316
train loss:1.1265229393875222
train loss:1.0612983161066902
=== epoch:293, train acc:0.6866666666666666, test acc:0.5475 ===
train loss:1.092782418198133
train loss:1.028531826528122
train loss:1.114657865537807
=== epoch:294, train acc:0.7, test acc:0.5481 ===
train loss:1.048447756272369
train loss:1.1085499654859714
train loss:1.1358765118660985
=== epoch:295, train acc:0.6933333333333334, test acc:0.5481 ===
train loss:1.1207715587884104
train loss:1.042577608945435
train loss:1.2942974538660619
=== epoch:296, train acc:0.6966666666666667, test acc:0.5496 ===
train loss:1.075272148596963
train loss:1.0969113854657062
train loss:1.2738765853464067
=== epoch:297, train acc:0.7033333333333334, test acc:0.5533 ===
train loss:1.1599976587630951
train loss:1.0945595856221981
train loss:1.0816429976751474
=== epoch:298, train acc:0.7033333333333334, test acc:0.5498 ===
train loss:1.1731712515631025
train loss:1.0808633721308063
train loss:1.1718141165503868
=== epoch:299, train acc:0.7066666666666667, test acc:0.5503 ===
train loss:1.1696123351320127
train loss:1.1458717185104812
train loss:1.0710112641225613
=== epoch:300, train acc:0.7, test acc:0.55 ===
train loss:1.0482047864167503
train loss:1.1389192988301808
train loss:1.1782273781359587
=== epoch:301, train acc:0.7033333333333334, test acc:0.5538 ===
train loss:1.0283817248874554
train loss:1.093119824364711
=============== Final Test Accuracy ===============
test acc:0.5542
```

```
<matplotlib.figure.Figure at 0x7f75ef73da20>
```

ハイパーパラメータの検証

ハイパーパラメータとは、各層のニューロン数やバッチサイズ、パラメータ更新の際の学習係数など、今まであてずっぽで設定していた数値。
ハイパーパラメータは適切な値に設定しなければ性能の悪い学習モデルになってしまう。

検証データ (Validation Data)

- 検証データ: 学習モデルのハイパーパラメータを検証する際に使うデータ
- テストデータ（評価データ）: 学習モデルを検証する際に使うデータ
- 学習データ（訓練データ）: 学習データ

ハイパーパラメータがテストデータ／学習データにのみ特化し汎化性能が失われる場合がある為
テストデータ／学習データを検証データに使ってはならない。
データの分け方としては以下のようなケースがある。

In [9]:

```
# coding: utf-8
import os
import sys
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
import matplotlib.pyplot as plt
from dataset.mnist import load_mnist
from common.util import shuffle_dataset

(x_train, t_train), (x_test, t_test) = load_mnist()
#訓練データをシャッフル
x_train, t_train = shuffle_dataset(x_train, t_train)

#検証データの分割
validation_rate = 0.20
validation_num = int(x_train.shape[0] * validation_rate)

x_val = x_train[:validation_num]#検証データ
t_val = t_train[:validation_num]#検証データ
x_train = x_train[validation_num:]#学習データ
t_train = t_train[validation_num:]#テストデータ
```

ーからデータを分けるなら
生データ → (検証データ → 学習用データ + 評価用データ) + (学習データ → 学習用データ + 評価用データ)
となる。
け方についてはー様分布によるランダムサンプリングが好ましい。
サンプルコードではー度データをシャッフルしてからー定の割合を抽出している。

ハイパーパラメータの最適化

ハイパーパラメータの最適化

- おおまかな範囲（10の階乗単位の対数スケール。例：10^{-3} ～ 10^{3}）を適当に決める。
- おおまかな範囲からランダムにハイパーパラメータを選出する。
- 評価を行い学習モデルの認識精度を観察してから範囲を狭めていく。

ハイパーパラメータはおおまかにランダムにザクッと決めることが効果的(http://www.jmlr.org/papers/volume13/bergstra12a/bergstra12a.pdf)であるがベイズ最適化を駆使(https://papers.nips.cc/paper/4522-practical-bayesian-optimization-of-machine-learning-algorithms.pdf)して厳密に計算することも可能。
ただし、ハイパーパラメータは評価に至るまで時間がかかるため、時間削減のためepochを小さくして 筋の悪そうなハイパーパラメータは早めに見切りをつけることが大切。

ハイパーパラメータの最適化手順

1. ハイパーパラメータの範囲を設定する
2. 設定されたハイパーパラメータの範囲からランダムにサンプリングする
3. ステップ2でサンプリングされたハイパーパラメータの値を使用して学習を行い検証データでの認識精度を評価する（※epochは小さく設定する）
4. ステップ2とステップ3をある回数（100回など）繰り返しそれらの認識精度の結果から、ハイパーパラメータの範囲を狭める。
5. 1.から4.を気が済むまで繰り返す。

ハイパーパラメータ最適化の実装

Pythonでは、10 ** np.random.uniform(-3, 3)と書くことで$10^{-3} \sim 10^{3}$の範囲でランダムサンプリング可能。

In [21]:

```
import numpy as np
print(10 ** np.random.uniform(-3, 3))
print(10 ** np.random.uniform(-3, 3))
print(10 ** np.random.uniform(-3, 3))
print(10 ** np.random.uniform(-3, 3))
```

```
2.6031755993127823
0.6363440506605467
429.65687301578265
349.5372690293394
```

In [24]:

```
weight_decay = 10 ** np.random.uniform(-8, -4)
lr = 10 ** np.random.uniform(-6, -2)
print("weight_decay:", weight_decay)
print("learning rate:", lr)
```

```
weight_decay: 4.289791449003003e-08
learning rate: 5.510074094447752e-06
```

In [17]:

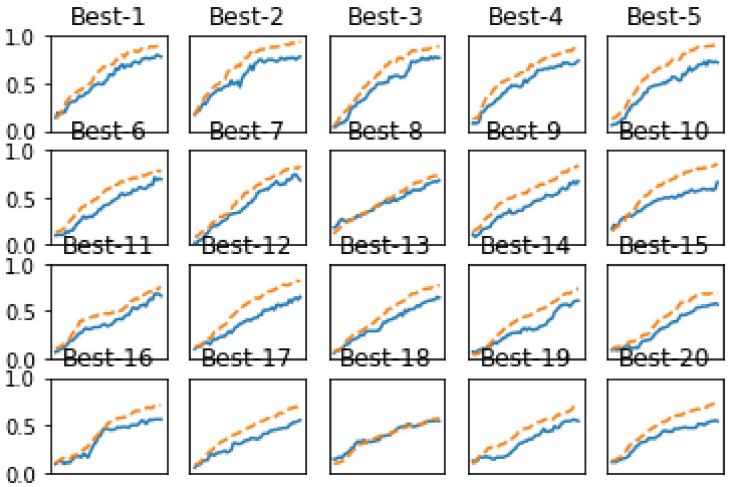
```
# coding: utf-8
import sys, os
sys.path.append(os.pardir)  # 親ディレクトリのファイルをインポートするための設定
import numpy as np
import matplotlib.pyplot as plt
from dataset.mnist import load_mnist
from common.multi_layer_net import MultiLayerNet
from common.util import shuffle_dataset
from common.trainer import Trainer

(x_train, t_train), (x_test, t_test) = load_mnist(normalize=True)

# 高速化のため訓練データの削減
x_train = x_train[:500]
t_train = t_train[:500]

# 検証データの分離
validation_rate = 0.20
validation_num = int(x_train.shape[0] * validation_rate)
x_train, t_train = shuffle_dataset(x_train, t_train)
x_val = x_train[:validation_num]
t_val = t_train[:validation_num]
x_train = x_train[validation_num:]
t_train = t_train[validation_num:]


def __train(lr, weight_decay, epocs=50):
    network = MultiLayerNet(input_size=784, hidden_size_list=[100, 100, 100, 100, 100, 100],
                            output_size=10, weight_decay_lambda=weight_decay)
    trainer = Trainer(network, x_train, t_train, x_val, t_val,
                      epochs=epocs, mini_batch_size=100,
                      optimizer='sgd', optimizer_param={'lr': lr}, verbose=False)
    trainer.train()

    return trainer.test_acc_list, trainer.train_acc_list


# ハイパーパラメータのランダム探索======================================
optimization_trial = 100
results_val = {}
results_train = {}
for _ in range(optimization_trial):
    # 探索したハイパーパラメータの範囲を指定===============
    weight_decay = 10 ** np.random.uniform(-8, -4)
    lr = 10 ** np.random.uniform(-6, -2)
    # ================================================

    val_acc_list, train_acc_list = __train(lr, weight_decay)
    print("val acc:" + str(val_acc_list[-1]) + " | lr:" + str(lr) + ", weight decay:" + str(weig
ht_decay))
    key = "lr:" + str(lr) + ", weight decay:" + str(weight_decay)
    results_val[key] = val_acc_list
    results_train[key] = train_acc_list

# グラフの描画==========================================================
print("=========== Hyper-Parameter Optimization Result ===========")
graph_draw_num = 20
col_num = 5
row_num = int(np.ceil(graph_draw_num / col_num))
```

```
i = 0

for key, val_acc_list in sorted(results_val.items(), key=lambda x:x[1][-1], reverse=True):
    print("Best-" + str(i+1) + "(val acc:" + str(val_acc_list[-1]) + ") | " + key)

    plt.subplot(row_num, col_num, i+1)
    plt.title("Best-" + str(i+1))
    plt.ylim(0.0, 1.0)
    if i % 5: plt.yticks([])
    plt.xticks([])
    x = np.arange(len(val_acc_list))
    plt.plot(x, val_acc_list)
    plt.plot(x, results_train[key], "--")
    i += 1

    if i >= graph_draw_num:
        break

plt.show()
```

val acc:0.56 | lr:0.004566696120232825, weight decay:1.5969743292639288e-07
val acc:0.67 | lr:0.006525974176274847, weight decay:1.2753302851950627e-07
val acc:0.08 | lr:1.1516746032106855e-05, weight decay:3.7189747861876775e-05
val acc:0.11 | lr:0.000260264068700295, weight decay:6.889341753981655e-07
val acc:0.57 | lr:0.0040205140430813685, weight decay:7.366089464868839e-07
val acc:0.11 | lr:5.0993701157740635e-06, weight decay:1.781310577020698e-08
val acc:0.11 | lr:1.105994750024369e-05, weight decay:1.5707152987989059e-06
val acc:0.12 | lr:6.0042844139806414e-05, weight decay:4.580200944918364e-06
val acc:0.12 | lr:0.0005471894574985656, weight decay:1.2617970079911852e-07
val acc:0.17 | lr:2.603729891157405e-05, weight decay:3.200377992803121e-06
val acc:0.08 | lr:5.23834874861114e-05, weight decay:1.0251318508398392e-06
val acc:0.1 | lr:1.094529307828141e-05, weight decay:9.453825835413832e-05
val acc:0.08 | lr:3.879487392778149e-06, weight decay:1.0569174608096627e-08
val acc:0.16 | lr:3.451373020909633e-06, weight decay:4.480359344197798e-07
val acc:0.18 | lr:0.0010596414980648709, weight decay:9.283996502980033e-08
val acc:0.13 | lr:3.0031329872430472e-06, weight decay:1.974049066745719e-08
val acc:0.78 | lr:0.009779048474070934, weight decay:1.351682025547439e-06
val acc:0.29 | lr:0.002309019930593434, weight decay:8.561708735450507e-06
val acc:0.54 | lr:0.0038101215997068446, weight decay:1.7574178600691726e-05
val acc:0.13 | lr:3.1107203795863696e-06, weight decay:3.323793465545872e-05
val acc:0.13 | lr:0.0003450855004374945, weight decay:1.7646710648000065e-05
val acc:0.12 | lr:0.00033261550485782257, weight decay:3.6912792427116194e-07
val acc:0.08 | lr:4.843001704656871e-05, weight decay:2.557289613877472e-08
val acc:0.15 | lr:1.4164040092490789e-06, weight decay:1.8824196286487735e-06
val acc:0.29 | lr:0.0006761083432232099, weight decay:9.577261646331784e-05
val acc:0.15 | lr:0.000360492404095738, weight decay:1.7750976479277526e-08
val acc:0.21 | lr:0.0007978964376835612, weight decay:1.4428453765119217e-08
val acc:0.13 | lr:6.848749908356776e-06, weight decay:7.989549207841927e-07
val acc:0.69 | lr:0.004445971857514631, weight decay:5.8606897926086976e-08
val acc:0.72 | lr:0.008220792113215686, weight decay:2.5325368104168772e-08
val acc:0.17 | lr:0.000324726614354249, weight decay:2.9578190702854325e-06
val acc:0.54 | lr:0.004029755255736933, weight decay:5.47933121162538e-07
val acc:0.19 | lr:0.0006164121963136637, weight decay:1.4148327137096943e-05
val acc:0.55 | lr:0.004359193185972747, weight decay:4.546379947243963e-07
val acc:0.05 | lr:4.803128649485642e-05, weight decay:1.485951669384858e-06
val acc:0.22 | lr:3.8581706337123985e-05, weight decay:5.579346796522656e-08
val acc:0.08 | lr:2.0824451920771285e-05, weight decay:2.466493803362757e-05
val acc:0.14 | lr:1.8943628853261206e-06, weight decay:3.851744658671523e-05
val acc:0.78 | lr:0.008374908719779505, weight decay:8.349159115994573e-06
val acc:0.09 | lr:1.4657645570309318e-06, weight decay:2.1448507872079716e-07
val acc:0.13 | lr:0.0004472210326178419, weight decay:2.6097529292406664e-07
val acc:0.07 | lr:1.679031562515611e-06, weight decay:5.358053813737613e-07
val acc:0.09 | lr:7.047966422809452e-06, weight decay:9.2455192370086e-08
val acc:0.74 | lr:0.006475159571652492, weight decay:7.2050817257212294e-06
val acc:0.27 | lr:0.0018683199114762454, weight decay:2.879708798899426e-07
val acc:0.16 | lr:1.033703624967391e-05, weight decay:1.4036926978716486e-08
val acc:0.08 | lr:1.2119080600658149e-06, weight decay:1.1190364058808228e-06
val acc:0.08 | lr:0.0010163825027416696, weight decay:9.138084636153747e-06
val acc:0.46 | lr:0.003568102655626634, weight decay:3.159881562696843 5e-05
val acc:0.65 | lr:0.006789710392273347, weight decay:1.7478414439085595e-07
val acc:0.15 | lr:3.494799891412799e-05, weight decay:4.38662124084519e-06
val acc:0.77 | lr:0.009323085074401091, weight decay:4.625089765870598e-07
val acc:0.33 | lr:0.0023201963637831044, weight decay:8.599198952080675e-05
val acc:0.13 | lr:0.00022775845061572797, weight decay:7.011227793740556e-06
val acc:0.24 | lr:0.0013307899272796172, weight decay:1.1559487537211119e-05
val acc:0.68 | lr:0.004362195756987248, weight decay:2.8434760553946477e-08
val acc:0.1 | lr:1.869330592751728e-05, weight decay:3.630681532090649e-06
val acc:0.66 | lr:0.004391866605439157, weight decay:6.690754210451607e-08
val acc:0.11 | lr:0.00012157840787319765, weight decay:3.019207683588879e-06
val acc:0.07 | lr:5.73197619547558e-06, weight decay:2.1248114247065703e-08
val acc:0.05 | lr:6.625590516575907e-05, weight decay:4.1668550057624566e-05

```
val acc:0.12 | lr:5.660096880946212e-06, weight decay:4.047970759554877e-06
val acc:0.47 | lr:0.004094660684753804, weight decay:2.139026281739255e-05
val acc:0.07 | lr:0.0001437853850230587, weight decay:3.501513239480806e-05
val acc:0.07 | lr:2.7678348082938608e-06, weight decay:3.605875969270973e-05
val acc:0.61 | lr:0.00501244747823184, weight decay:6.6061630974215995e-06
val acc:0.08 | lr:1.4459656560560125e-06, weight decay:6.921478885518981e-08
val acc:0.06 | lr:1.877902881369377e-06, weight decay:8.861234175020568e-06
val acc:0.16 | lr:0.00027992860374476103, weight decay:8.136908635997955e-06
val acc:0.64 | lr:0.005584996179822284, weight decay:2.090197431586645e-05
val acc:0.09 | lr:3.7044851999906146e-05, weight decay:4.041497491549873e-06
val acc:0.53 | lr:0.004741102906808209, weight decay:3.502481235924246e-07
val acc:0.06 | lr:0.0001500538282592443, weight decay:8.710511069186092e-05
val acc:0.05 | lr:0.0007201893628910685, weight decay:1.4141463050091068e-07
val acc:0.1 | lr:0.0002495235789467229, weight decay:1.0247541807452934e-06
val acc:0.33 | lr:0.0033522635404612876, weight decay:5.607510911261655e-06
val acc:0.54 | lr:0.00420288585593756, weight decay:2.9751099377293143e-08
val acc:0.1 | lr:3.139872005608798e-05, weight decay:2.115085732690944e-08
val acc:0.08 | lr:0.0001734550858119241 7, weight decay:1.572664068064797e-05
val acc:0.09 | lr:4.8541455667862805e-05, weight decay:2.497774082582095e-06
val acc:0.07 | lr:7.845278502551505e-05, weight decay:4.45901951288139e-07
val acc:0.08 | lr:9.645301964391163e-05, weight decay:4.5264854697146816e-08
val acc:0.1 | lr:0.00010109347952662167, weight decay:1.4972460411885978e-08
val acc:0.2 | lr:8.355921001909981e-06, weight decay:2.7235869781658287e-08
val acc:0.24 | lr:0.0014810412114846852, weight decay:5.607416368730088e-06
val acc:0.16 | lr:6.507964953119368e-06, weight decay:1.3242922217095415e-05
val acc:0.36 | lr:0.0025833134792508723, weight decay:2.6120485931760436e-06
val acc:0.13 | lr:1.527643704818154e-06, weight decay:2.570591487346121e-07
val acc:0.05 | lr:7.307140440620332e-06, weight decay:9.590537582632035e-05
val acc:0.08 | lr:1.1744292411860575e-05, weight decay:3.821521013064728e-07
val acc:0.08 | lr:1.4362586820432618e-05, weight decay:8.85781365016604e-08
val acc:0.14 | lr:2.0988274915780365e-06, weight decay:1.9002015863970334e-06
val acc:0.68 | lr:0.005762442868939066, weight decay:4.455212914034312e-05
val acc:0.1 | lr:0.0004516820555058821, weight decay:8.336087557854645e-06
val acc:0.42 | lr:0.003253784601778332, weight decay:1.9826434940886548e-07
val acc:0.66 | lr:0.006332897041544693 6, weight decay:2.3015817865279555e-05
val acc:0.35 | lr:0.0020977436884009595, weight decay:1.8457007243634437e-05
val acc:0.15 | lr:0.0004325436447926334, weight decay:1.5319094146776284e-07
val acc:0.44 | lr:0.0023420582425195203, weight decay:6.1012606383627386e-06
val acc:0.09 | lr:4.264418219877188e-06, weight decay:1.7105643403207038e-05
=========== Hyper-Parameter Optimization Result ===========
Best-1(val acc:0.78) | lr:0.008374908719779505, weight decay:8.349159115994573e-06
Best-2(val acc:0.78) | lr:0.009779048474070934, weight decay:1.351682025547439e-06
Best-3(val acc:0.77) | lr:0.009323085074401091, weight decay:4.625089765870598e-07
Best-4(val acc:0.74) | lr:0.006475159571652492, weight decay:7.2050817257212294e-0
6
Best-5(val acc:0.72) | lr:0.008220792113215686, weight decay:2.5325368104168772e-0
8
Best-6(val acc:0.69) | lr:0.004445971857514631, weight decay:5.8606897926086976e-0
8
Best-7(val acc:0.68) | lr:0.005762442868939066, weight decay:4.455212914034312e-05
Best-8(val acc:0.68) | lr:0.004362195756987248, weight decay:2.8434760553946477e-0
8
Best-9(val acc:0.67) | lr:0.006525974176274847, weight decay:1.2753302851950627e-0
7
Best-10(val acc:0.66) | lr:0.004391866605439157, weight decay:6.690754210451607e-0
8
Best-11(val acc:0.66) | lr:0.0063328970415446936, weight decay:2.3015817865279555e
-05
Best-12(val acc:0.65) | lr:0.006789710392273347, weight decay:1.7478414439085595e-
07
Best-13(val acc:0.64) | lr:0.005584996179822284, weight decay:2.090197431586645e-0
```

```
5
Best-14(val acc:0.61) | lr:0.00501244747823184, weight decay:6.6061630974215995e-0
6
Best-15(val acc:0.57) | lr:0.0040205140430813685, weight decay:7.366089464868839e-
07
Best-16(val acc:0.56) | lr:0.004566696120232825, weight decay:1.5969743292639288e-
07
Best-17(val acc:0.55) | lr:0.004359193185972747, weight decay:4.546379947243963e-0
7
Best-18(val acc:0.54) | lr:0.004029755255736933, weight decay:5.47933121162538e-07
Best-19(val acc:0.54) | lr:0.00420288585593756, weight decay:2.9751099377293143e-0
8
Best-20(val acc:0.54) | lr:0.0038101215997068446, weight decay:1.7574178600691726e
-05
```



認識精度が高い上位20のグラフを見る限りでは

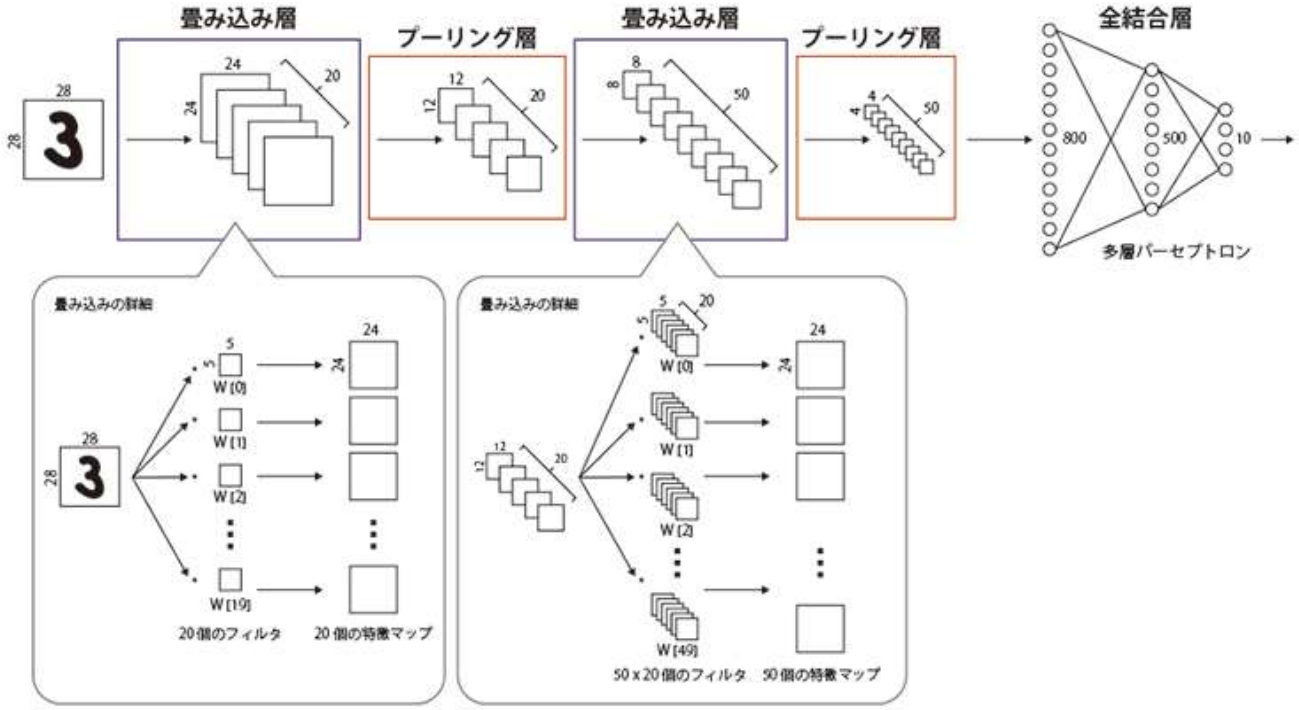
- lr(学習係数)が0.001～0.01
- weight_decayが10^{-8}～10^{-6} くらいが良さそうである。

畳み込みニューラルネットワーク

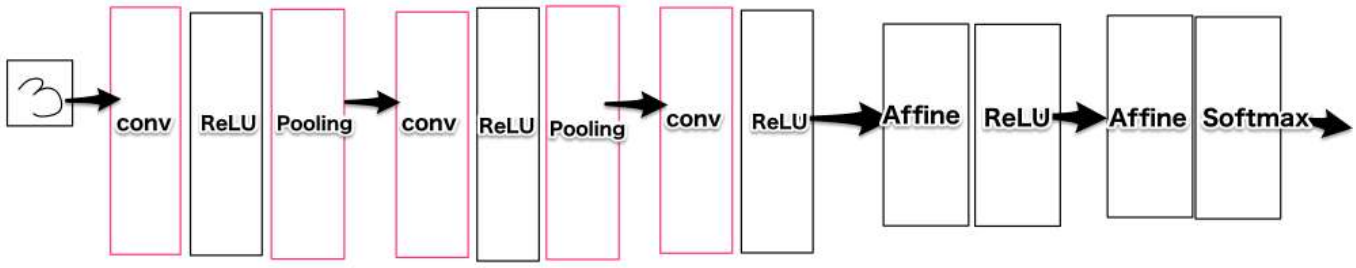
畳み込み層

- 畳み込み層: CNN（Convolutional Neural Network）で新たに取り入れられる層の一つ。
- 特徴マップ(feature map): 畳み込み層における入出力データのこと
- 入力特徴マップ(input feature map): 畳み込み層における入力データのこと
- 出力特徴マップ(output feature map): 畳み込み層における出力データのこと

http://www.hpc.co.jp/AboutDeepLearning.html (http://www.hpc.co.jp/AboutDeepLearning.html)



http://pythonskywalker.hatenablog.com/entry/2016/12/26/164545
(http://pythonskywalker.hatenablog.com/entry/2016/12/26/164545)



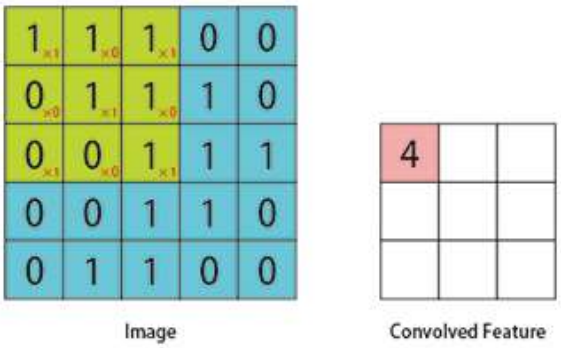
全結合層の問題点

これまでのニューラルネットワークは隣接する層のすべてのニューロンとニューロンの間で結合がある（fully-connected; 全結合;）状態だった。
我々はこれまでそれぞれの結合部をAffineレイヤ（行列の内積計算）で実装しReLUにて発火させた。これは何の考慮もせずに入力値をそのまま行列に変換してニューラルネットワークを構築したことになる。つまり今回のMNISTの手書き文字の場合、元々あった縦・横などの「データの形状情報」やRGB（ピクセルの色合いの順序情報;R要素・G要素・B要素の区別;）が失われてしまうことになる。
データをそのまま全結合層の入力（行列）として扱った場合3次元以上の情報は失われてしまう！
例えば、画像を行列として扱った場合に失われる情報としては以下が考えられる

- ピクセルのR,G,B,Aのそれぞれの濃度
- 縦・横のピクセル幅
- 上下左右で連続したピクセル間の値の推移

畳み込み演算

畳み込み演算は画像処理で言う「フィルター演算」にあたる。
http://www.hpc.co.jp/AboutDeepLearning.html (http://www.hpc.co.jp/AboutDeepLearning.html)



「フィルター」はカーネルとも呼ばれ、入力データに対してあるウインドウサイズの区画で積和演算を行う。全結合のニューラルネットワークでは行列が重みパラメータとなっていたが、CNNの場合はこのフィルターの乗算内容が「重み」に対応する。
積和演算の結果は各ウィンドウの位置ごとに取得され、結果は新たな出力データとなる。
その出力データの行列にスカラをスカラ加算（行列の要素全てに等しく加算）することでバイアスも実現する。
バイアスはスカラなので、1 x 1の行列となる。

パディング

入力データの大外にダミーの枠を用意することで、積和演算結果のサイズを調整できる。
畳み込み演算を繰り返すと段々出力サイズが小さくなるので、最終的な出力サイズが1になることを防ぐことができる。
畳み込み演算回数（隠れ層の数）によっては不要の場合がある。

ストライド

フィルターの適用する位置の間隔のこと。積和演算が一回終わった時点でウィンドウがスライドする幅。
パディングとは逆に、ストライドは大きくすると積和演算結果の出力サイズは小さくなる。

ストライドと出力サイズの関係

- 出力サイズ ($OH = \frac{H+2P-FH}{S} + 1, OW = \frac{W+2P-FW}{S} + 1$)
  - 入力サイズ (H, W)
  - フィルターサイズ (FH, FW)
  - パディング P
  - ストライド S

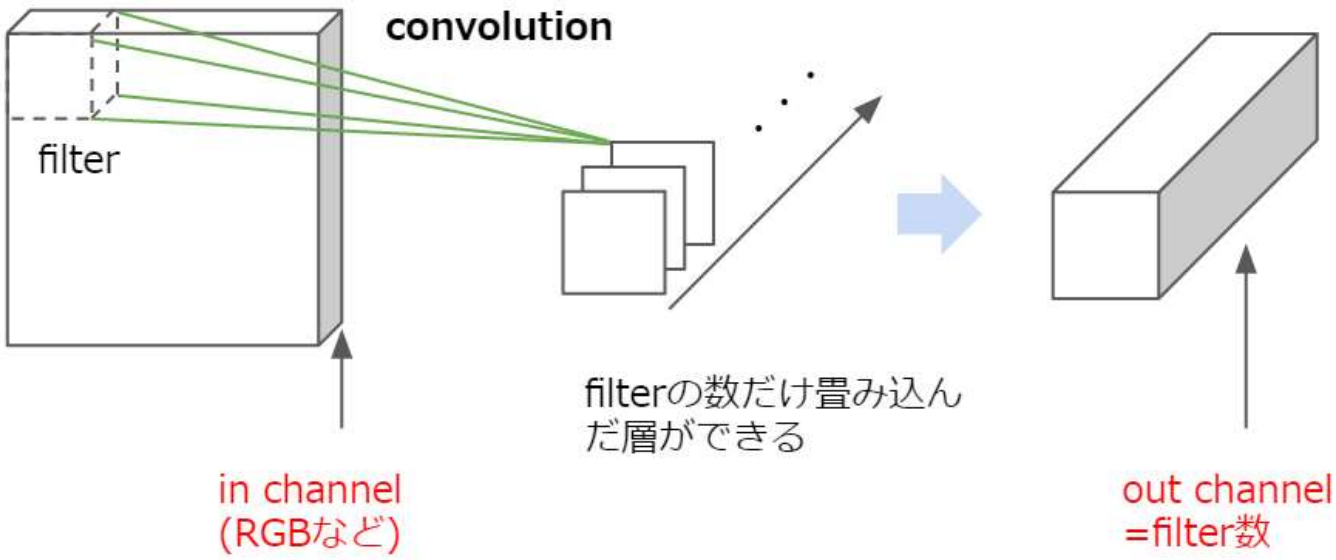
ただし、ストライドで割り切れない値の場合は長方形の形にならない（行列において欠損データが発生する）為 割り切れない値を丸めるかダミー値を挿入すること。

3次元データの畳み込み演算

通常の畳込み演算にチャンネル（奥行き）を加えたもの。

- 適用フィルターの枚数（=重み） x 入力データ = 出力チャンネル数となる。
- バイアスは（フィルター数, 1, 1）という形でブロードキャスト演算により適用される。

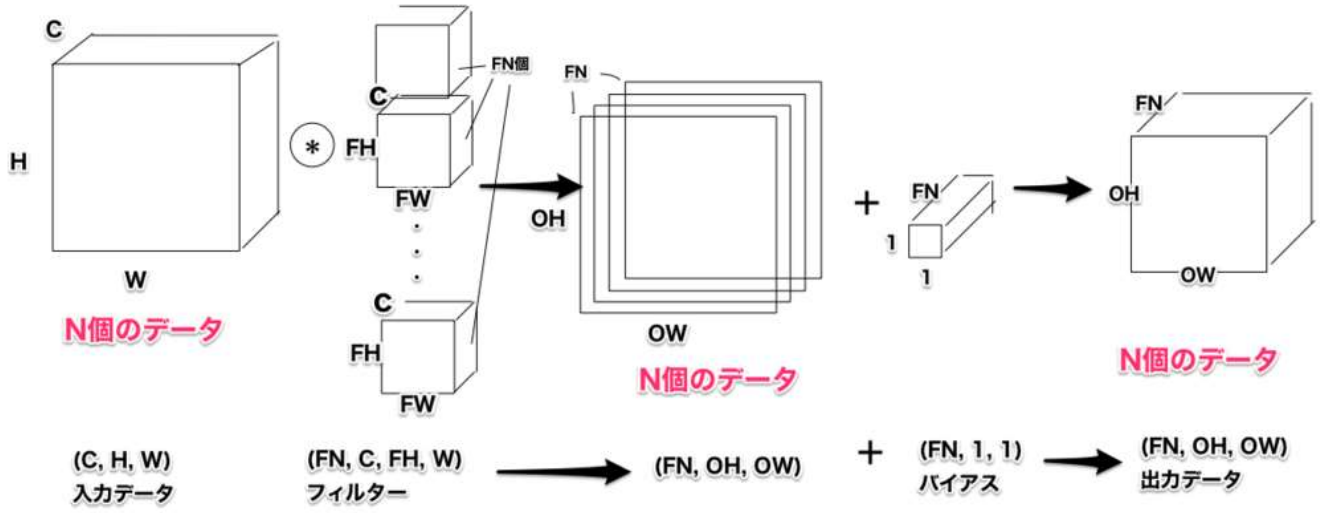
http://pythonskywalker.hatenablog.com/entry/2016/12/26/164545
(http://pythonskywalker.hatenablog.com/entry/2016/12/26/164545)



畳み込み演算のバッチ演算

単にN個のデータを重ねただけ。（4次元データの演算）

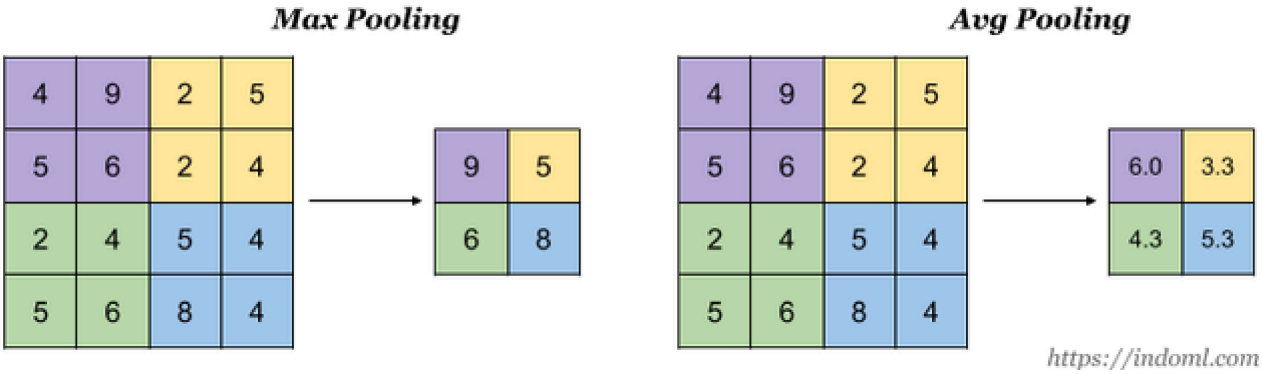
http://pythonskywalker.hatenablog.com/entry/2016/12/26/164545
(http://pythonskywalker.hatenablog.com/entry/2016/12/26/164545)



プーリング層

プーリング層(Pooling)は引数フィルターなどを持たずに、関数のみで入力を縦・横の空間を狭くして出力する演算のこと。
一般的にはウィンドウサイズ＝ストライドとなるように設定する。
MAXプーリングは以下のような形で演算がなされる。他にも、Averageプーリングなどがある。

https://indoml.com/2018/03/07/student-notes-convolutional-neural-networks-cnn-introduction/
(https://indoml.com/2018/03/07/student-notes-convolutional-neural-networks-cnn-introduction/)



プーリング層の特徴

学習するパラメータが無い

プーリング層は畳み込みそうと違い学習パラメータを持たない。（層自体が成長しない）

チャンネル層は変化しない

プーリング層を通過しても入力データと出力データのチャンネル数は変化しない。

微小な位置変化に対してロバスト（頑健）

入力データの小さなズレに対してもプーリング層は同じような結果を返す。
入力データが多少ズレていてもウィンドウサイズに収まっている程度のズレであれば吸収できる。

Convolution／Poolingレイヤの実装

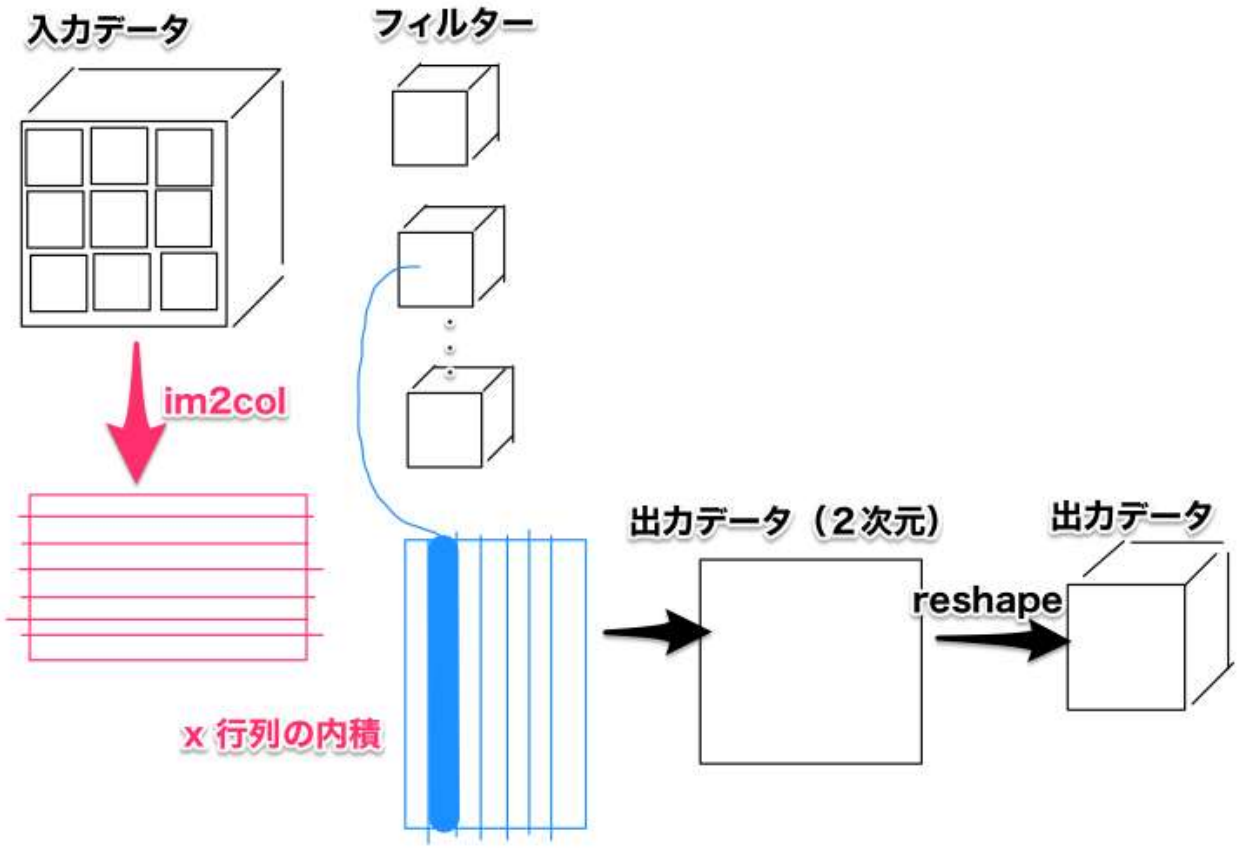
4次元配列

CNNは高さx横幅xチャンネルxデータ数の四次元セットと言える。

im2colによる展開

畳み込み演算はim2colを使用して入力データを単純な行列の計算に落とし込む方法を使うと線形代数ライブラリで高速処理できるようになる。

http://narusawa-a-hiyoko.hatenablog.com/entry/2016/04/12/042039 (http://narusawa-a-hiyoko.hatenablog.com/entry/2016/04/12/042039)



Convolutionレイヤの実装

im2colの実装

In [14]:

```
def im2col(input_data, filter_h, filter_w, stride=1, pad=0):
    """

    Parameters
    ----------
    input_data : (データ数, チャンネル, 高さ, 幅)の4次元配列からなる入力データ
    filter_h : フィルターの高さ
    filter_w : フィルターの幅
    stride : ストライド
    pad : パディング

    Returns
    -------
    col : 2次元配列
    """
    N, C, H, W = input_data.shape #データ数、チャンネル(奥行き)、高さ、幅
    out_h = (H + 2*pad - filter_h)//stride + 1#出力高さを計算
    out_w = (W + 2*pad - filter_w)//stride + 1#出力幅を計算

    img = np.pad(input_data, [(0,0), (0,0), (pad, pad), (pad, pad)], 'constant')#0埋めでパディング
    #no.pad(array, pad_width, mode=None, **kwargs)
    col = np.zeros((N, C, filter_h, filter_w, out_h, out_w))#出力用0埋め二次元データのハコを作成
    #np.zeros(shape, dtype=float, order='C')

    for y in range(filter_h):#ストライドを反映させる
        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]

    col = col.transpose(0, 4, 5, 1, 2, 3).reshape(N*out_h*out_w, -1)
    #XXX: ここが分からん！！！
    return col
```

In [17]:

```
import sys, os
import numpy as np
sys.path.append(os.pardir)
from common.util import im2col

x1 = np.random.rand(1,3,7,7)
print(x1.shape)
#    1: データ数(バッチ数)
#    3: チャンネル(奥行き)
#    7: 高さ
#    7: 幅
#4次元配列からなる入力データ

col1 = im2col(x1, 5, 5, stride=1, pad=0)
#    x1 : (データ数(バッチ数), チャンネル(奥行き), 高さ, 幅)の4次元配列からなる入力データ
#    5 : フィルターの高さ
#    5 : フィルターの幅
#    1 : ストライド
#    0 : パディング
print(col1.shape)
# 9 = #XXX: ここが分からん！
# 75 = (チャンネル(奥行き) x フィルターの高さ x フィルターの高さ)

x2 = np.random.rand(10,3,7,7)
print(x2.shape)

col2 = im2col(x2, 5, 5, stride=1, pad=0)
print(col2.shape)
# 90 = #XXX: ここが分からん！
# 75 = (チャンネル(奥行き) x フィルターの高さ x フィルターの高さ)
```

```
(1, 3, 7, 7)
(9, 75)
(10, 3, 7, 7)
(90, 75)
```

畳み込み層をConvolutionという名前のクラスで実装する

In [19]:

```
class Convolution:
    def __init__(self, W, b, stride=1, pad=0):
        self.W = W #フィルターの「重み」
        #FN, C, FH, FW = self.W.shape
        self.b = b #フィルターの「バイアス」
        self.stride = stride #フィルターのストライド
        self.pad = pad #フィルター適用前のパディング

        # 中間データ（backward時に使用）
        self.x = None
        self.col = None
        self.col_W = None

        # 重み・バイアスパラメータの勾配
        self.dW = None
        self.db = None

    def forward(self, x):
        FN, C, FH, FW = self.W.shape
        #FN=Filterの数, C=チャンネル（奥行）, FH=フィルターの高さ, FW=フィルターの幅
        N, C, H, W = x.shape
        out_h = 1 + int((H + 2*self.pad - FH) / self.stride)
        out_w = 1 + int((W + 2*self.pad - FW) / self.stride)

        col = im2col(x, FH, FW, self.stride, self.pad) #入力データを行列式（二次元データ）に変換
        col_W = self.W.reshape(FN, -1).T #フィルターデータを行列式（二次元データ）に変換

        out = np.dot(col, col_W) + self.b# 二次元データとなり扱えるようになった。Affineレイヤと同じく、内積を取る
        out = out.reshape(N, out_h, out_w, -1).transpose(0, 3, 1, 2)# 内積を取った後に元の出力データの形に戻す

        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)
        dx = col2im(dcol, self.x.shape, FH, FW, self.stride, self.pad)

        return dx
```

Convolutionレイヤの順伝搬

- 入力データをim2colで２次元配列に展開
- フィルターもreshapeで二次元配列に展開
- Affine変換（内積の取得）

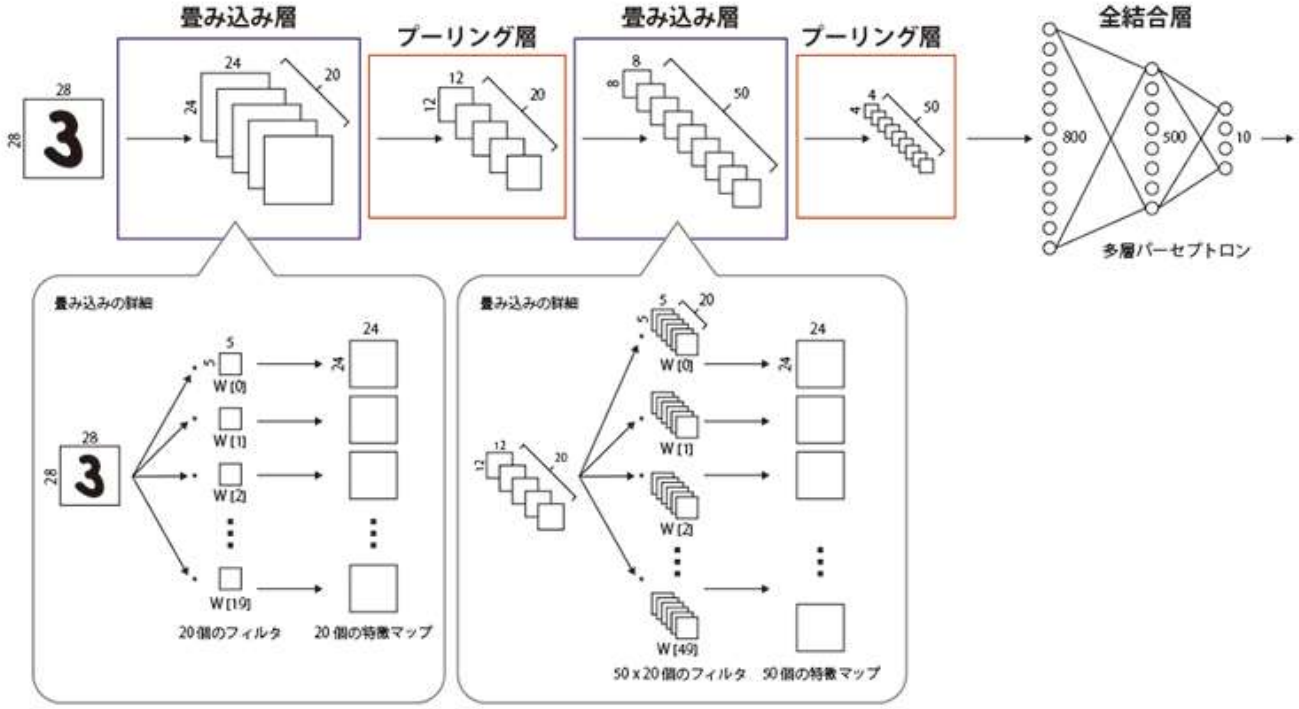
Convolutionレイヤの逆伝搬

- col2imを使う

Poolingレイヤの実装

PoolingレイヤもConvolutionレイヤと同様、im2colを使って入力データを展開する。但し、プーリングの適用領域チャンネル方向によって変わらない。展開した行列から最大値を求め、適切な計上に整形する。（MAX-Pooling）

http://www.hpc.co.jp/AboutDeepLearning.html (http://www.hpc.co.jp/AboutDeepLearning.html)



In [1]:

```
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)

        col = im2col(x, self.pool_h, self.pool_w, self.stride, self.pad)
        col = col.reshape(-1, self.pool_h*self.pool_w)

        arg_max = np.argmax(col, axis=1)
        out = np.max(col, axis=1)
        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.pad)

        return dx
```

CNNの実装

SimpleConvNetクラスの実装

In [4]:

```
from graphviz import Digraph
dot = Digraph(comment="計算グラフ")
dot.attr(rankdir="LR")
#dot.attr(splines="") #line or curved or ortho or polyline;
dot.attr(fixedsize="true")
dot.attr(label="単純なCNNのネットワーク構成")
with dot.subgraph(name="main") as main:
    main.node("L1","Conv")
    main.node("L2","ReLU")
    main.node("L3","Pooling")
    main.node("L4","Affine")
    main.node("L5","ReLU")
    main.node("L6","Affine")
    main.node("L7","Softmax")
    main.edge("L1", "L2",label="")
    main.edge("L2", "L3",label="")
    main.edge("L3", "L4",label="")
    main.edge("L4", "L5",label="")
    main.edge("L5", "L6",label="")
    main.edge("L6", "L7",label="")
    #print(dot)
dot
```

Out[4]:

単純なCNNのネットワーク構成

In [5]:

```
# coding: utf-8
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 pickle
import numpy as np
from collections import OrderedDict
from common.layers import *
from common.gradient import numerical_gradient


class SimpleConvNet:
    """単純なConvNet

    conv - relu - pool - affine - relu - affine - softmax

    Parameters
    ----------
    input_size : 入力サイズ（MNISTの場合は784）
    hidden_size_list : 隠れ層のニューロンの数のリスト（e.g. [100, 100, 100]）
    output_size : 出力サイズ（MNISTの場合は10）
    activation : 'relu' or 'sigmoid'
    weight_init_std : 重みの標準偏差を指定（e.g. 0.01）
        'relu'または'he'を指定した場合は「Heの初期値」を設定
        'sigmoid'または'xavier'を指定した場合は「Xavierの初期値」を設定
    """
    def __init__(self, input_dim=(1, 28, 28),
                 conv_param={'filter_num':30, 'filter_size':5, 'pad':0, 'stride':1},
                 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_stride + 1
        pool_output_size = int(filter_num * (conv_output_size/2) * (conv_output_size/2))

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

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

```
        self.layers['Affine2'] = Affine(self.params['W3'], self.params['b3'])

        self.last_layer = SoftmaxWithLoss()

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

        return x

    def loss(self, x, t):
        """損失関数を求める
        引数のxは入力データ、tは教師ラベル
        """
        y = self.predict(x)
        return self.last_layer.forward(y, t)

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

        acc = 0.0

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

        return acc / x.shape[0]

    def numerical_gradient(self, x, t):
        """勾配を求める（数値微分）

        Parameters
        ----------
        x : 入力データ
        t : 教師ラベル

        Returns
        -------
        各層の勾配を持ったディクショナリ変数
            grads['W1']、grads['W2']、...は各層の重み
            grads['b1']、grads['b2']、...は各層のバイアス
        """
        loss_w = lambda w: self.loss(x, t)

        grads = {}
        for idx in (1, 2, 3):
            grads['W' + str(idx)] = numerical_gradient(loss_w, self.params['W' + str(idx)])
            grads['b' + str(idx)] = numerical_gradient(loss_w, self.params['b' + str(idx)])

        return grads

    def gradient(self, x, t):
        """勾配を求める（誤差逆伝搬法）

        Parameters
        ----------
        x : 入力データ
        t : 教師ラベル
```

```
        Returns
        -------
        各層の勾配を持ったディクショナリ変数
            grads['W1']、grads['W2']、...は各層の重み
            grads['b1']、grads['b2']、...は各層のバイアス
        """
        # forward
        self.loss(x, t)

        # 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
        grads = {}
        grads['W1'], grads['b1'] = self.layers['Conv1'].dW, self.layers['Conv1'].db
        grads['W2'], grads['b2'] = self.layers['Affine1'].dW, self.layers['Affine1'].db
        grads['W3'], grads['b3'] = self.layers['Affine2'].dW, self.layers['Affine2'].db

        return grads

    def save_params(self, file_name="params.pkl"):
        params = {}
        for key, val in self.params.items():
            params[key] = val
        with open(file_name, 'wb') as f:
            pickle.dump(params, f)

    def load_params(self, file_name="params.pkl"):
        with open(file_name, 'rb') as f:
            params = pickle.load(f)
        for key, val in params.items():
            self.params[key] = val

        for i, key in enumerate(['Conv1', 'Affine1', 'Affine2']):
            self.layers[key].W = self.params['W' + str(i+1)]
            self.layers[key].b = self.params['b' + str(i+1)]
```

SimpleConvNetの引数

- 初期化の際の重みの標準偏差
- 入力データ（C, H, W）の次元
- Convolution層
  - フィルターの数
  - フィルターのサイズ
  - フィルターのストライド
  - フィルターのパディング
- 隠れ層
  - 全結合ニューロン数
- 出力層
  - 全結合ニューロン数

SimpleConvNetにおけるMNIST

In [ ]:

```
# coding: utf-8
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
import matplotlib.pyplot as plt
from dataset.mnist import load_mnist
from simple_convnet import SimpleConvNet
from common.trainer import Trainer

# データの読み込み
(x_train, t_train), (x_test, t_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, t_train = x_train[:5000], t_train[:5000]
#x_test, t_test = x_test[:1000], t_test[:1000]

max_epochs = 20

network = SimpleConvNet(input_dim=(1,28,28),
                        conv_param = {'filter_num': 30, 'filter_size': 5, 'pad': 0, 'stride': 1
},
                        hidden_size=100, output_size=10, weight_init_std=0.01)

trainer = Trainer(network, x_train, t_train, x_test, t_test,
                  epochs=max_epochs, mini_batch_size=100,
                  optimizer='Adam', optimizer_param={'lr': 0.001},
                  evaluate_sample_num_per_epoch=1000)
trainer.train()

# パラメータの保存
network.save_params("params.pkl")
print("Saved Network Parameters!")

# グラフの描画
markers = {'train': 'o', 'test': 's'}
x = np.arange(max_epochs)
plt.plot(x, trainer.train_acc_list, marker='o', label='train', markevery=2)
plt.plot(x, trainer.test_acc_list, marker='s', label='test', markevery=2)
plt.xlabel("epochs")
plt.ylabel("accuracy")
plt.ylim(0, 1.0)
plt.legend(loc='lower right')
plt.show()
```

CNNの可視化

一層目の重みの可視化

一層目は畳み込みフィルタである

In [2]:

```
from graphviz import Digraph
dot = Digraph(comment="計算グラフ")
dot.attr(rankdir="LR")
#dot.attr(splines="") #line or curved or ortho or polyline;
dot.attr(fixedsize="true")
dot.attr(label="単純なCNNのネットワーク構成")
with dot.subgraph(name="main") as main:
    main.node("L1","Conv", color="red")
    main.node("L2","ReLU")
    main.node("L3","Pooling")
    main.node("L4","Affine")
    main.node("L5","ReLU")
    main.node("L6","Affine")
    main.node("L7","Softmax")
    main.edge("L1", "L2",label="")
    main.edge("L2", "L3",label="")
    main.edge("L3", "L4",label="")
    main.edge("L4", "L5",label="")
    main.edge("L5", "L6",label="")
    main.edge("L6", "L7",label="")
    #print(dot)
dot
```

Out[2]:

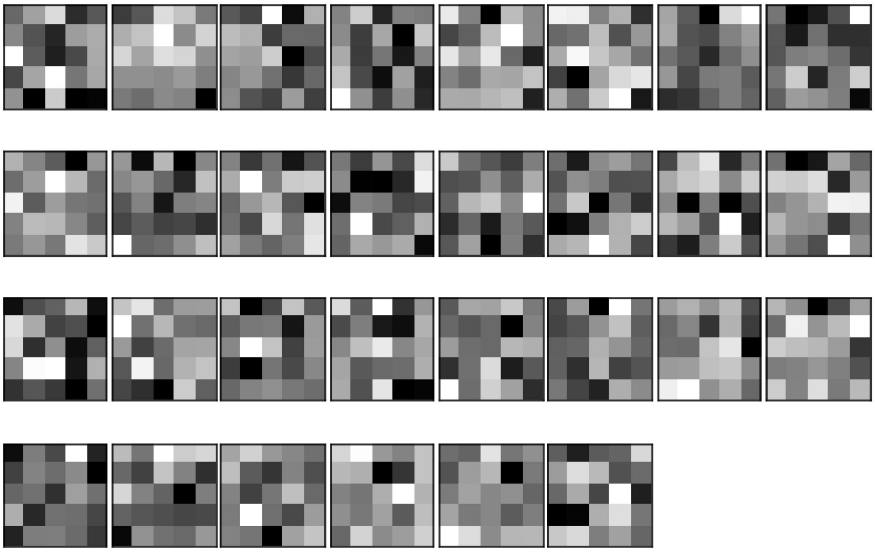
単純なCNNのネットワーク構成

- 畳み込み層の重みとなるフィルタはグレースケールの画像を表現する
  - サイズ（縦・横）: 5 x 5
  - チャンネル(色彩): 1
  - フィルター（数）: 30

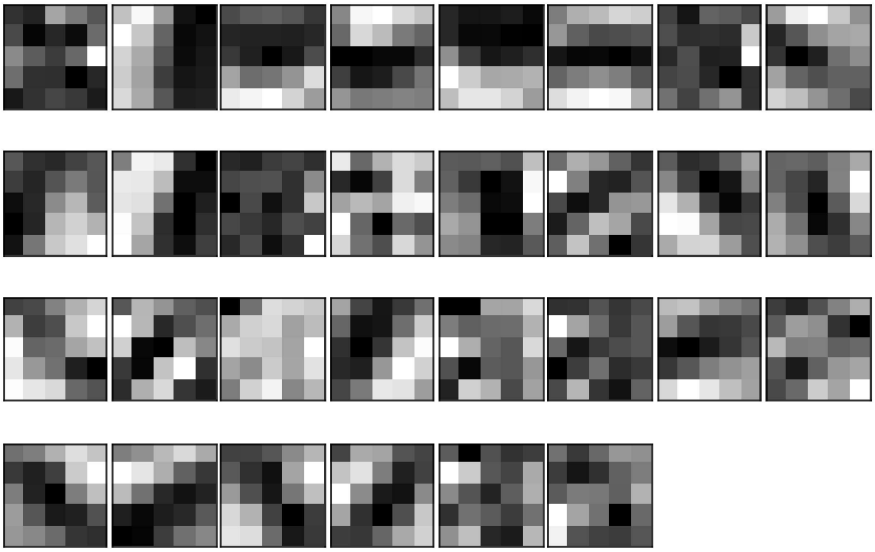
In [4]:

```
# coding: utf-8
import numpy as np
import matplotlib.pyplot as plt
from simple_convnet import SimpleConvNet

def filter_show(filters, nx=8, margin=3, scale=10):
    """
    c.f. https://gist.github.com/aidiary/07d530d5e08011832b12#file-draw_weight-py
    """
    FN, C, FH, FW = filters.shape
    ny = int(np.ceil(FN / nx))

    fig = plt.figure()
    fig.subplots_adjust(left=0, right=1, bottom=0, top=1, hspace=0.05, wspace=0.05)

    for i in range(FN):
        ax = fig.add_subplot(ny, nx, i+1, xticks=[], yticks=[])
        ax.imshow(filters[i, 0], cmap=plt.cm.gray_r, interpolation='nearest')
    plt.show()


network = SimpleConvNet()
# ランダム初期化後の重み
print("学習前")
filter_show(network.params['W1'])
print("学習後")
# 学習後の重み
network.load_params("params.pkl")
filter_show(network.params['W1'])
```

学習前



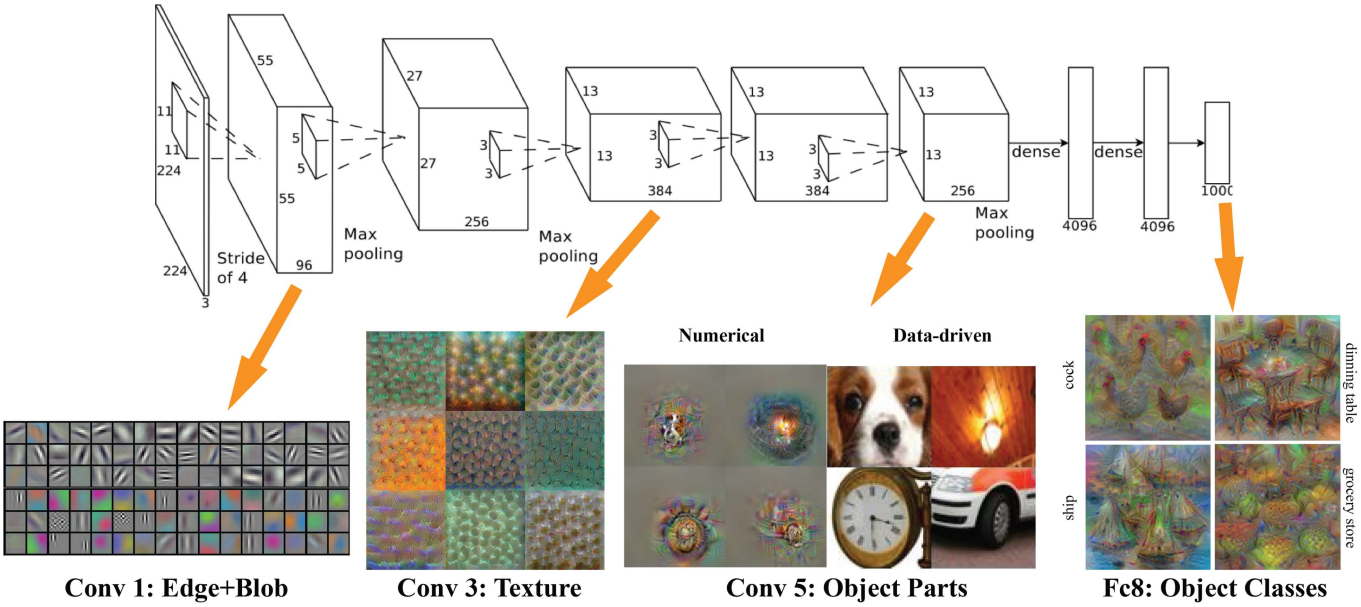
学習後



一様分布で生成されたフィルタが重みの学習後にある程度の偏りを見せていることが確認できる。
これは画像におけるプリミティブな部分（色が大きく変わる部分や固まって同じ色が続く部分）をフィルタが抽出していることを意味する。

階層構造による情報抽出

https://www.kisspng.com/png-deep-learning-convolutional-neural-network-alexnet-3064759/
(https://www.kisspng.com/png-deep-learning-convolutional-neural-network-alexnet-3064759/)



上記はAlexNetによる例

- 1層目はEdge・Blob
- 3層目はTexture
- 5層目はObject Parts
- 全結合層はObject Class

代表的なCNN

LeNet

CNNの元祖

AlexNet

- ReLUを使用
- LRN(Local Response Normalization)を使う※正規化の走り
- Dropoutを使用

アルゴリズムは既に存在していたが、ディープラーニング(ビッグデータ)の流行りは以下に支えられている

- 大量のデータを個人でも保存できるディスク容量単価の底下げ
- 大量のデータを個人でもダウンロードできるネットワーク帯域容量単価の底下げ
- 大量のデータを個人でも並列処理できるGPU計算容量単価の底下げ

ディープラーニング

ディープな手書き数字認識

- 3x3の小さなフィルターによる畳込み層
- 活性化関数はReLU
- 全結合層の後にDropoutレイヤを使用
- Adamによる最適化
- 重みの初期値として「Heの初期値」を使用

In [1]:

```
# coding: utf-8
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 pickle
import numpy as np
from collections import OrderedDict
from common.layers import *


class DeepConvNet:
    """認識率99%以上の高精度なConvNet

    ネットワーク構成は下記の通り
        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（TODO:自動で計算する）
        pre_node_nums = np.array([1*3*3, 16*3*3, 16*3*3, 32*3*3, 32*3*3, 64*3*3, 64*4*4, hidden_size])
        wight_init_scales = np.sqrt(2.0 / pre_node_nums)  # ReLUを使う場合に推奨される初期値

        self.params = {}
        pre_channel_num = input_dim[0]
        for idx, conv_param in enumerate([conv_param_1, conv_param_2, conv_param_3, conv_param_4, conv_param_5, conv_param_6]):
            self.params['W' + str(idx+1)] = wight_init_scales[idx] * np.random.randn(conv_param['filter_num'], pre_channel_num, conv_param['filter_size'], conv_param['filter_size'])
            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(64*4*4, hidden_size)
        self.params['b7'] = np.zeros(hidden_size)
        self.params['W8'] = wight_init_scales[7] * np.random.randn(hidden_size, output_size)
        self.params['b8'] = np.zeros(output_size)

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

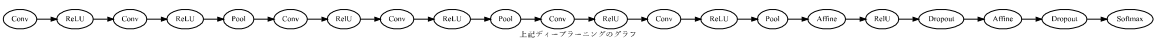
```
                        conv_param_4['stride'], conv_param_4['pad']))
        self.layers.append(Relu())
        self.layers.append(Pooling(pool_h=2, pool_w=2, stride=2))
        self.layers.append(Convolution(self.params['W5'], self.params['b5'],
                        conv_param_5['stride'], conv_param_5['pad']))
        self.layers.append(Relu())
        self.layers.append(Convolution(self.params['W6'], self.params['b6'],
                        conv_param_6['stride'], conv_param_6['pad']))
        self.layers.append(Relu())
        self.layers.append(Pooling(pool_h=2, pool_w=2, stride=2))
        self.layers.append(Affine(self.params['W7'], self.params['b7']))
        self.layers.append(Relu())
        self.layers.append(Dropout(0.5))
        self.layers.append(Affine(self.params['W8'], self.params['b8']))
        self.layers.append(Dropout(0.5))

        self.last_layer = SoftmaxWithLoss()

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

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

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

        acc = 0.0

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

        return acc / x.shape[0]

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

        # 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

    def save_params(self, file_name="params.pkl"):
        params = {}
        for key, val in self.params.items():
            params[key] = val
        with open(file_name, 'wb') as f:
            pickle.dump(params, f)

    def load_params(self, file_name="params.pkl"):
        with open(file_name, 'rb') as f:
            params = pickle.load(f)
        for key, val in params.items():
            self.params[key] = val

        for i, layer_idx in enumerate((0, 2, 5, 7, 10, 12, 15, 18)):
            self.layers[layer_idx].W = self.params['W' + str(i+1)]
            self.layers[layer_idx].b = self.params['b' + str(i+1)]
```

In [5]:

```
from graphviz import Digraph
dot = Digraph(comment="上記ディープラーニングのグラフ")
dot.attr(rankdir="LR")
#dot.attr(splines="") #line or curved or ortho or polyline;
dot.attr(fixedsize="true")
dot.attr(label="上記ディープラーニングのグラフ")
with dot.subgraph(name="main") as main:
    main.node("L1","Conv")
    main.node("L2","ReLU")
    main.node("L3","Conv")
    main.node("L4","ReLU")
    main.node("L5","Pool")
    main.node("L6","Conv")
    main.node("L7","ReLU")
    main.node("L8","Conv")
    main.node("L9","ReLU")
    main.node("L10","Pool")
    main.node("L11","Conv")
    main.node("L12","ReLU")
    main.node("L13","Conv")
    main.node("L14","ReLU")
    main.node("L15","Pool")
    main.node("L16","Affine")
    main.node("L17","ReLU")
    main.node("L18","Dropout")
    main.node("L19","Affine")
    main.node("L20","Dropout")
    main.node("L21","Softmax")
    main.edge("L1", "L2",label="")
    main.edge("L2", "L3",label="")
    main.edge("L3", "L4",label="")
    main.edge("L4", "L5",label="")
    main.edge("L5", "L6",label="")
    main.edge("L6", "L7",label="")
    main.edge("L7", "L8",label="")
    main.edge("L8", "L9",label="")
    main.edge("L9", "L10",label="")
    main.edge("L10", "L11",label="")
    main.edge("L11", "L12",label="")
    main.edge("L12", "L13",label="")
    main.edge("L13", "L14",label="")
    main.edge("L14", "L15",label="")
    main.edge("L15", "L16",label="")
    main.edge("L16", "L17",label="")
    main.edge("L17", "L18",label="")
    main.edge("L18", "L19",label="")
    main.edge("L19", "L20",label="")
    main.edge("L20", "L21",label="")
    #print(dot)
dot
```

Out[5]:



このCNNの認識精度は概ね99%を超す

さらに認識精度を高めるには

- アンサンブル学習(ensemble learning)
- 学習系率の減衰(learning rate decay)
- データ拡張(data aughmentation)

Data Augmentation

入力画像を人工的なアルゴリズムによって微小変化させる

- Crop処理: データを切り出す
- flip処理: データを回線させる
- scale処理: データを拡大縮小させる

層を深くすることのモチベーション

層をいたずらに深くする事の重要性は証明されていないが、昨今の傾向としては層を深くする（認識性能の向上を見込んでいる）方向に向かっている。 層を深くすることの利点

- 層を深くしない場合に比べより少ないパラメータで同レベル以上の表現をすることが出来る
- 特に小さなフィルターを重ねてネットワークを深くすることにより受容野（receptive field）を広くすることが出来る。
- 層を重ねることでReLU等の活性化関数が畳み込み層の間に挟まれることになり、非線形の力を強める事ができる。
- 各層が別の特徴を捉えることでCNN全体が一つの特徴以外に囚われなくなる
    - 各層が学習すべき課題がよりシンプルな問題へと分解される

ディープラーニングの少歴史

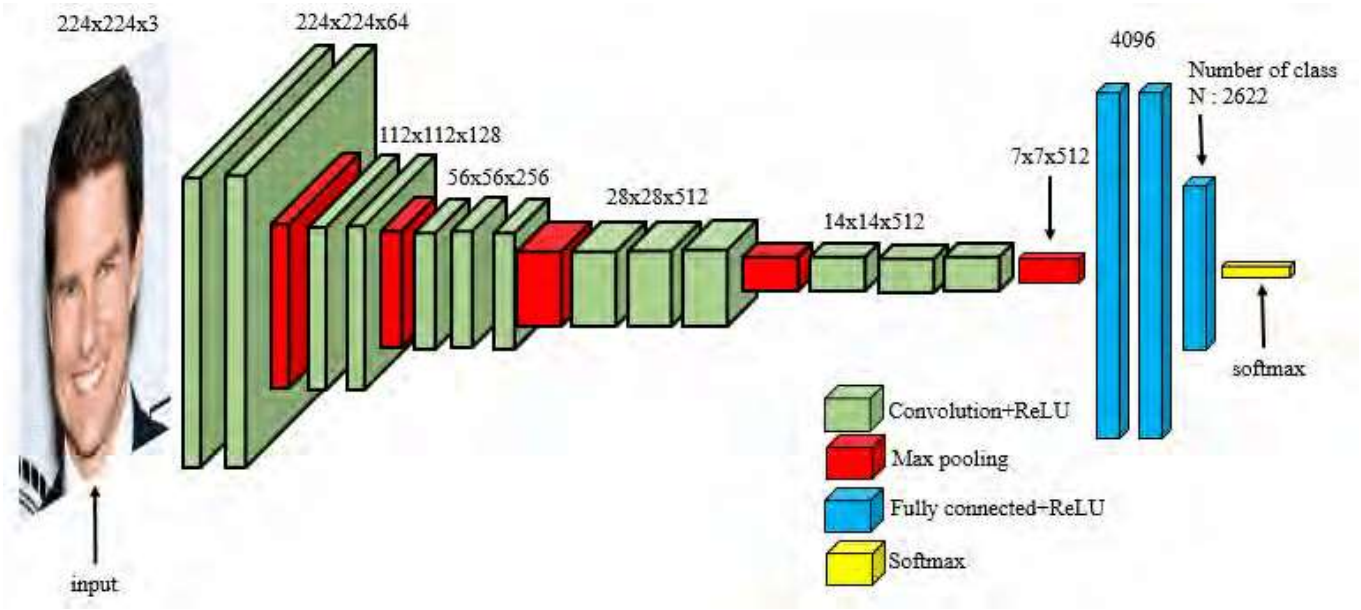
ILSVRC(ImageNet Large Scale Visual Recognition Challenge)でAlexNetが圧倒的成績で優勝したことがブームの始まり

ImageNet

ILSVRC(ImageNet Large Scale Visual Recognition Challenge) (http://www.image-net.org/challenges/LSVRC/)は画像認識のコンペティション。
その中でもVGG, GoogleLeNet, RasNetが有名。

VGG

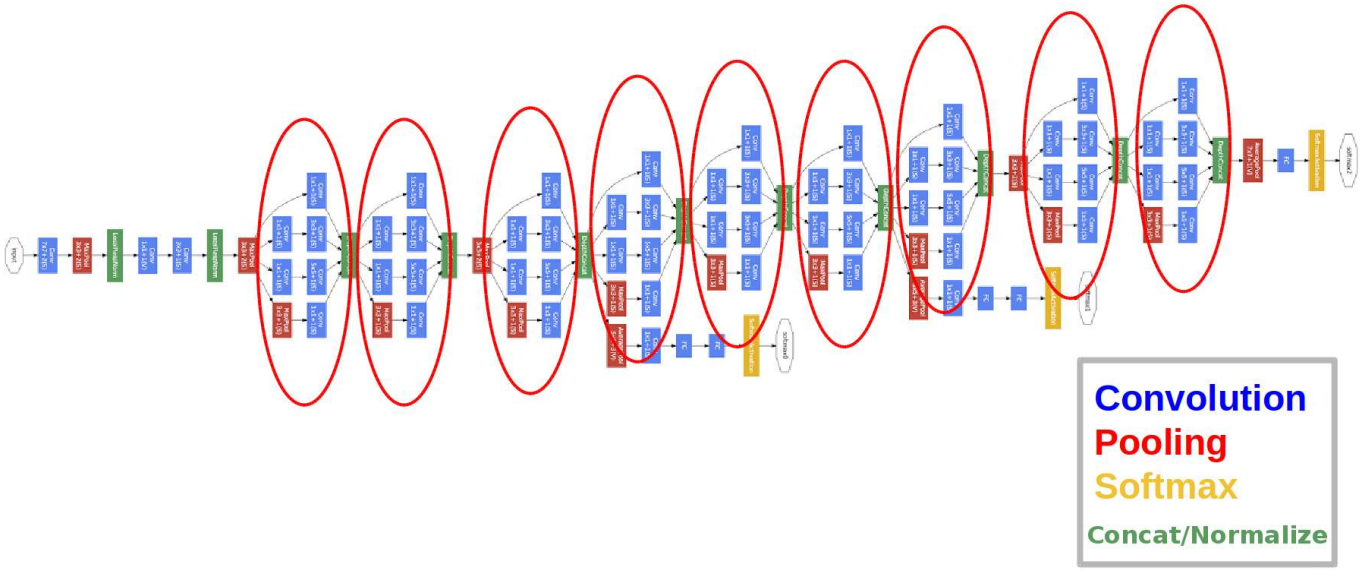
https://www.researchgate.net/figure/A-visualization-of-the-VGG-architecture_fig2_318701491 (https://www.researchgate.net/figure/A-visualization-of-the-VGG-architecture_fig2_318701491)



VGGは非常にシンプル。3x3フィルタの畳み込み層を連続させ、プーリング層でサイズを半分にする。最後に全結合層を経由して出力。

GoogLeNet

https://leonardoaraujosantos.gitbooks.io/artificial-inteligence/content/googlenet.html (https://leonardoaraujosantos.gitbooks.io/artificial-inteligence/content/googlenet.html)

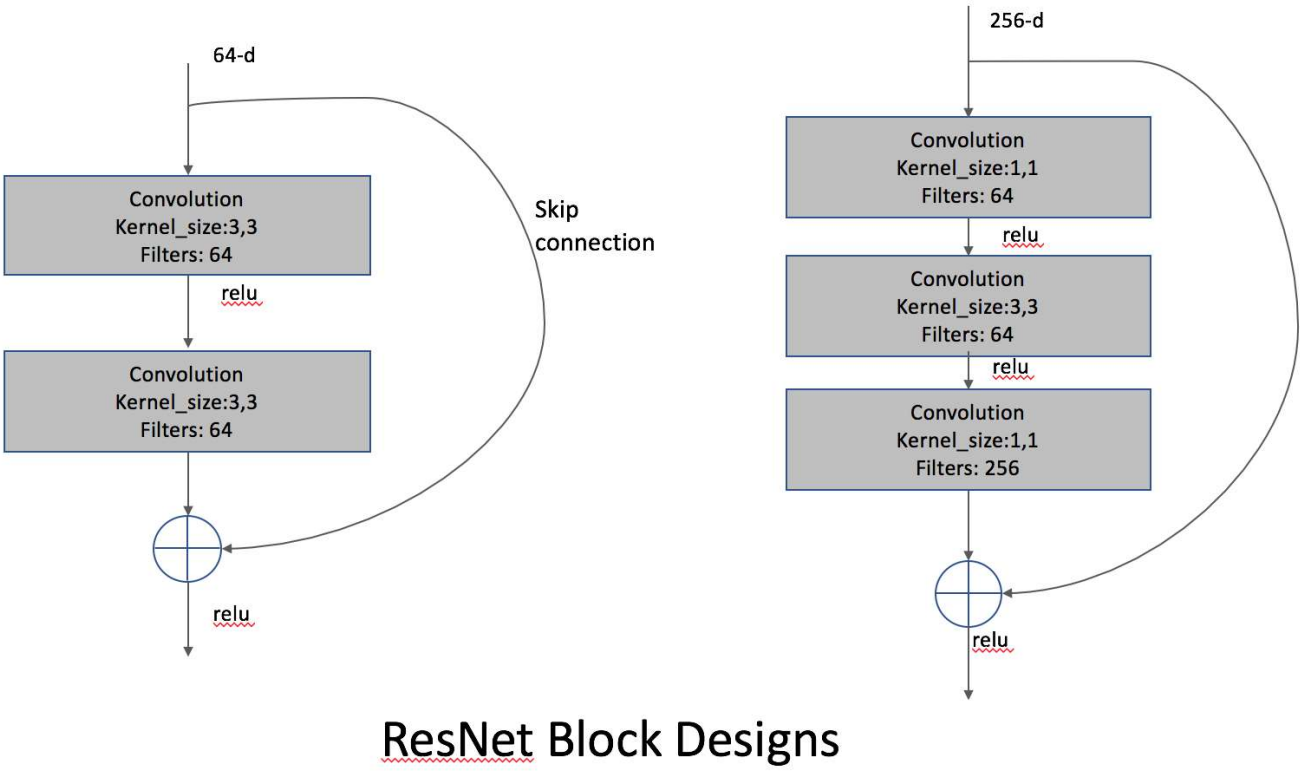


インセプション構造を1x1のフィルタによる畳み込み層で挟むことでパラメータの削減・処理の高速化をしている

ResNet

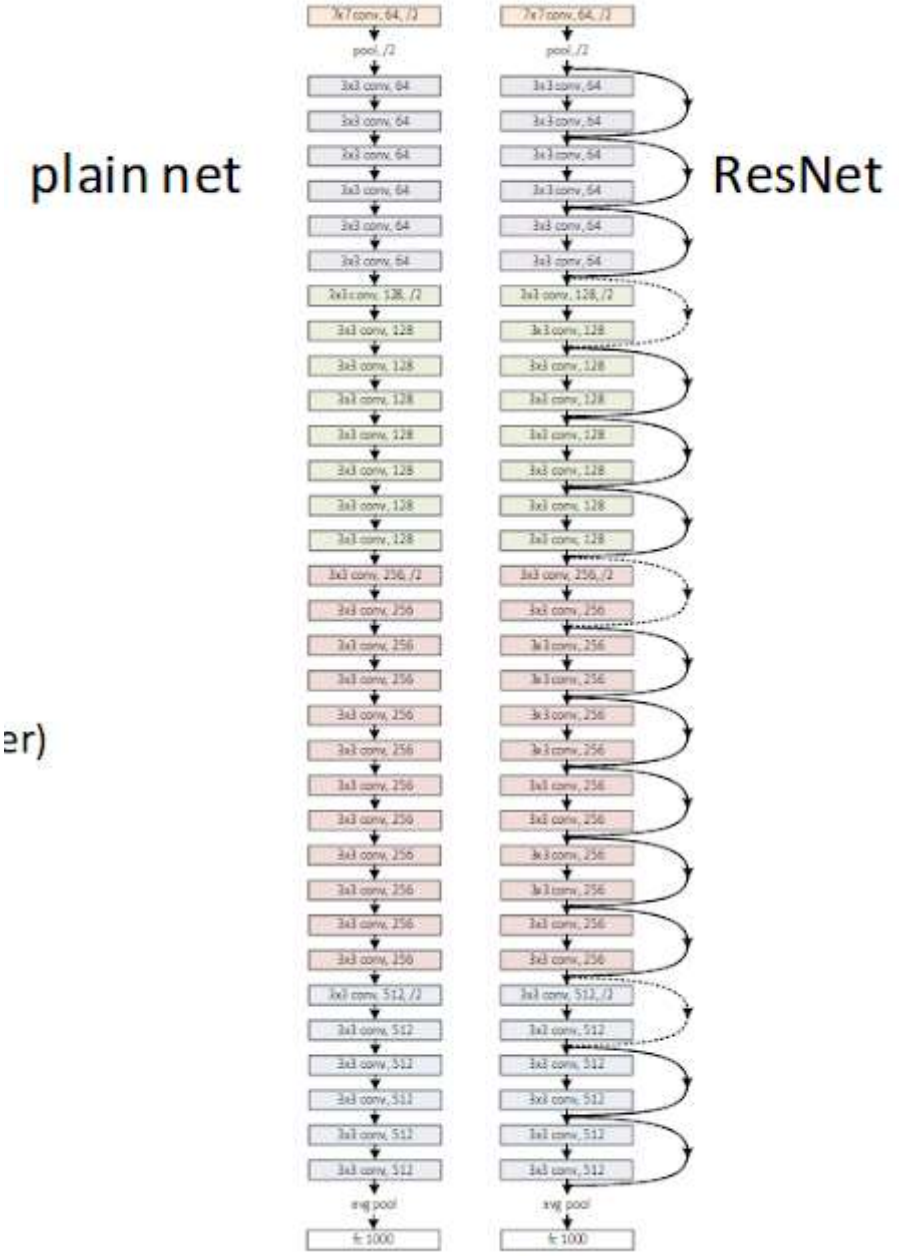
層を深くしすぎると勾配減衰が発生し、最終的な性能が劣るという問題をクリアするために、「スキップ構造」を使っている。 スキップ構造であ伝搬時も逆伝搬時も入力データをそのまま流すので、勾配の変化がゆるやかになる。

https://www.safaribooksonline.com/library/view/practical-computer-vision/9781788297684/12a933e6-e3c6-47d5-9064-512c8a0b4667.xhtml (https://www.safaribooksonline.com/library/view/practical-computer-vision/9781788297684/12a933e6-e3c6-47d5-9064-512c8a0b4667.xhtml)



ResNet Block Designs

ResNetはVGGにスキップ構造を足したもの。
http://alimurreza.blogspot.com/2017/04/deep-residual-network-resnet.html
(http://alimurreza.blogspot.com/2017/04/deep-residual-network-resnet.html)



転移学習

VGGなど既存のネットワーク構成をそのまま引き継ぐことで学習済みの重みを初期値として新しいデータセットを対象に再学習を行うことを「転移学習」と呼ぶ。手元にあるデータがスクアに場合において転移学習は有効な手法である。

ディープラーニングの高速化

ディープラーニングフレームワークの多くはGPU処理をサポートしているが、最近では複数のGPUや複数の筐体での分散学習にも対応してきている。

取り組むべき課題

https://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-93.pdf
(https://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-93.pdf)
畳み込み層の処理が最も計算容量が大きい為、学習時・推論時の両方の観点でいかに畳み込み層の計算量を減らすかがカギとなる。 ここで言う畳み込み層の処理とは、元をたどれば大量の積和演算であり、最終的には積和演算をいかに拘束に効率的に処理するかということに行き着く。

GPUによる高速化

NVIDIAが提供するCUDA開発環境を使ってGPUコンピューティングすることができる。特にim2colのように大きな塊を一気に計算できる形式に変換できる関数と相性が良い。

分散学習による高速化

TensorFlowのような分散学習の機能が入ったライブラリを使用すること

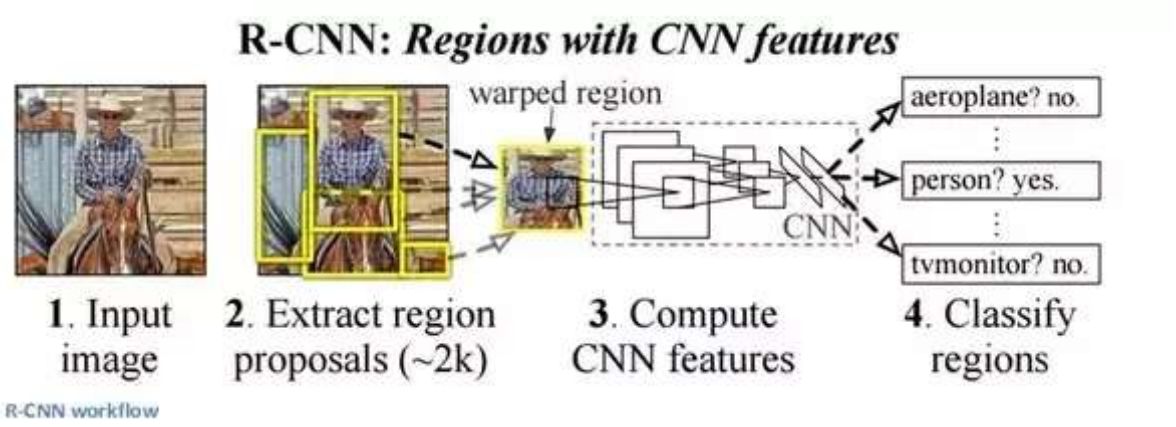
演算精度のビット削減

演算精度のビットを削減しても、そこまで演算結果に影響が無いことが分かっている。一般的に16ビットの半精度浮動小数点数でディープラーニングを行うことができることが分かっている。

ディープラーニングの実用例

物体検出

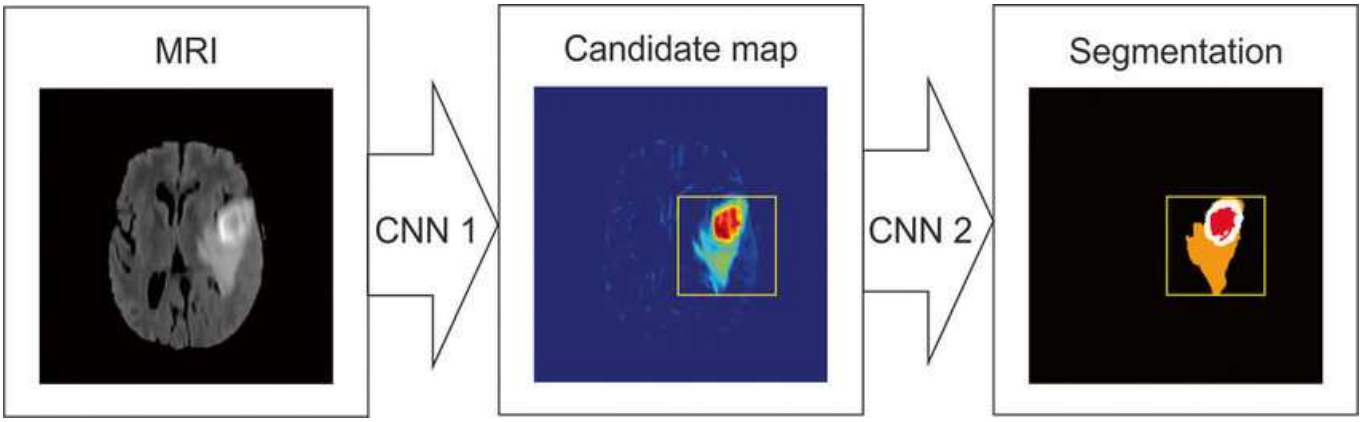
https://www.quora.com/Where-can-I-find-a-nice-tutorial-for-Regional-based-Convolution-neural-Network (https://www.quora.com/Where-can-I-find-a-nice-tutorial-for-Regional-based-Convolution-neural-Network)



物体検出では2.で物体らしき領域を抜き出し、3.で物体の内訳を処理するというフローとなる。
2.についてはR-CNNではSelective Searchと呼ばれる手法で実装されているが、最近ではこれもCNNで行う手法が提案されている。

セグメンテーション

https://www.researchgate.net/figure/Schematic-illustration-of-a-cascaded-CNN-architecture-for-brain-tumor-segmentation-task_fig2_317341396 (https://www.researchgate.net/figure/Schematic-illustration-of-a-cascaded-CNN-architecture-for-brain-tumor-segmentation-task_fig2_317341396)



画像のあるエリアを矩形抽出し、推論処理をすると縦x横のピクセル分も計算が必要なため、無駄である。
そこで、計算する前に一度畳み込みによりエッジ抽出をし、全結合層でもCNNを使い逆畳み込みを実施しデータを抽象化する手法。

画像キャプション生成

https://heuristic.exblog.jp/26280459/ (https://heuristic.exblog.jp/26280459/)

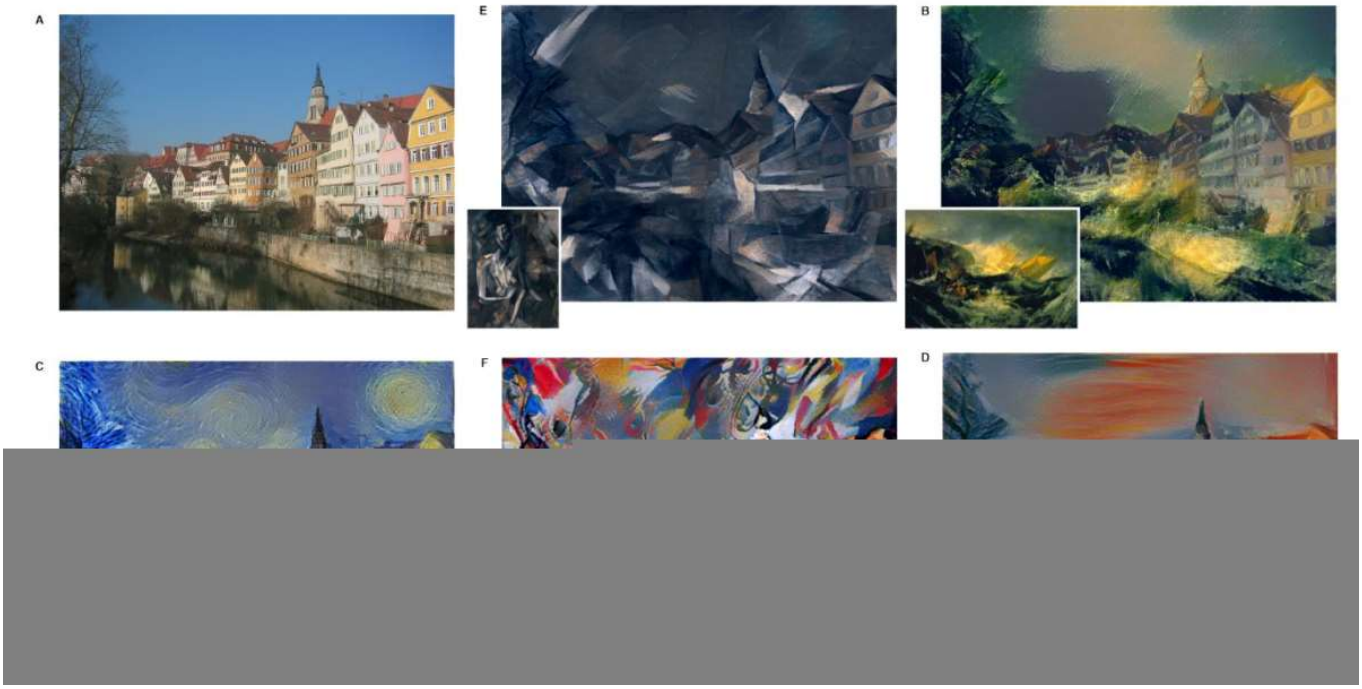
Figure 5. A selection of evaluation results, grouped by human rating.

CNNと自然言語処理を合わせたもの

ディープラーニングの未来

画像スタイル変換

https://japanese.engadget.com/2015/08/31/dnn/ (https://japanese.engadget.com/2015/08/31/dnn/)

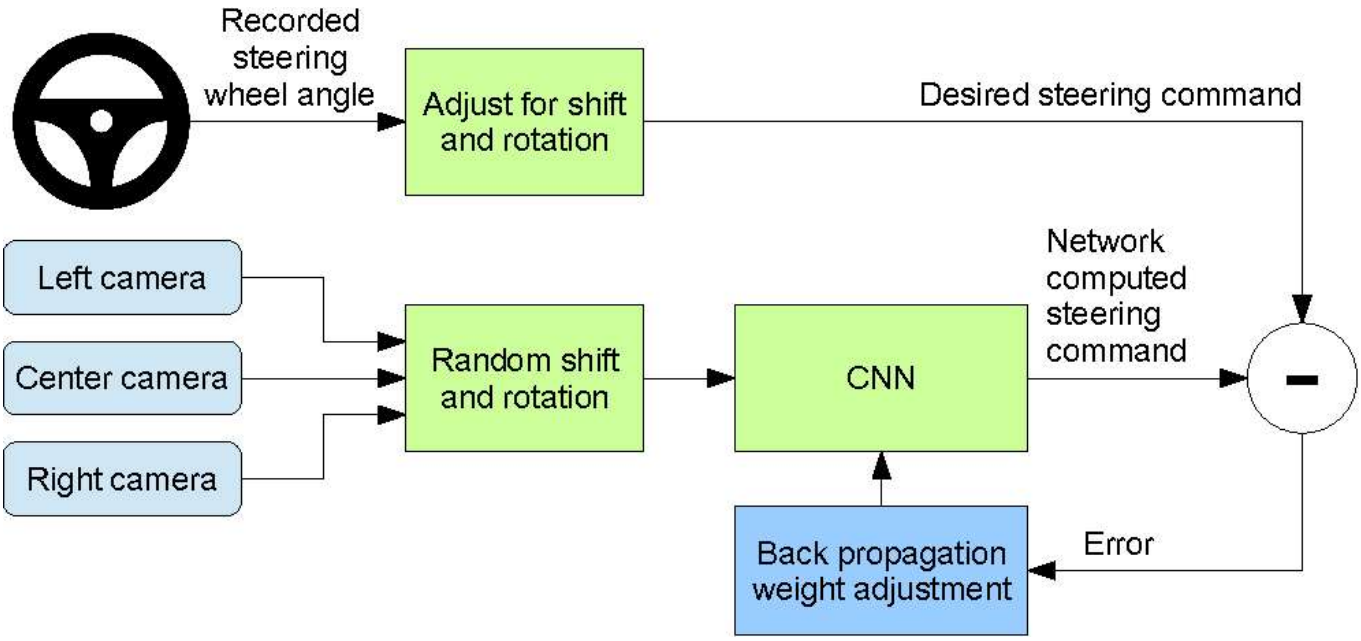


画像生成

https://qiita.com/sergeant-wizard/items/0a57485bc90a35efcf26 (https://qiita.com/sergeant-wizard/items/0a57485bc90a35efcf26)

自動運転

https://www.semanticscholar.org/paper/End-to-End-Learning-for-Self-Driving-Cars-Bojarski-Testa/0e3cc46583217ec81e87045a4f9ae3478a008227 (https://www.semanticscholar.org/paper/End-to-End-Learning-for-Self-Driving-Cars-Bojarski-Testa/0e3cc46583217ec81e87045a4f9ae3478a008227)



Deep Q-Network (強化学習)

https://www.toshiba-sol.co.jp/tech/sat/case/1804_1.htm (https://www.toshiba-sol.co.jp/tech/sat/case/1804_1.htm)

AlphaGoなど