

# パターン認識と学習

## 深層学習(1)

管理工学科

篠沢佳久

# 資料の内容

- 深層学習(1)
  - オートエンコーダ
  - 畳み込みニューラルネットワーク(1)
- 実習①(多層のネットワークの学習)
- 実習②(オートエンコーダ)

# 深層學習 (Deep Learning)

# ニューラルネットワークの大規模化

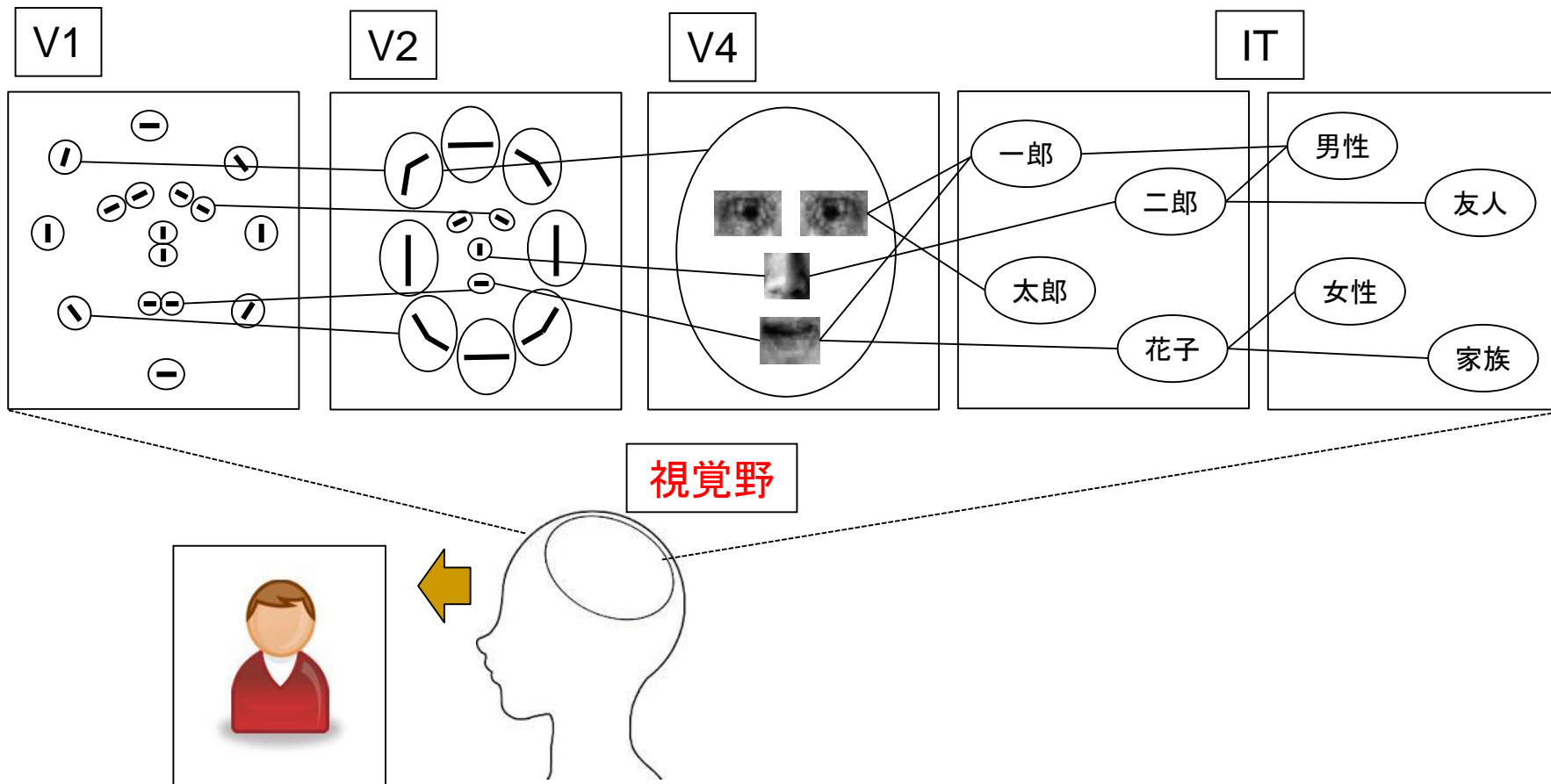
- 大規模データ
  - データ数, 入力情報の次元数が増加
  - 入力層が増加
- 多クラス分類問題
  - 出力層が増加
- ネットワークの大規模化
  - 多層化(深層)にすればよいのか

# 多層化した場合

- 単純に層を増やせばよいわけではない
- 層を増やせば、学習が困難になる  
→実習①(DL.py(8層のネットワーク)を実行してみてください)
- 階層化, 層間の結合方法(ネットワークアーキテクチャ)はどうすればよいか
- 層を増やすと何ができるようになるのか

# 多層化した場合、何ができるのか

## ■ (ヒント) 人間の視覚野のモデル



# 深層学習 (Deep Learning)

- 目的

Deep learning methods aim at learning **feature hierarchies** with features from higher levels of the hierarchy formed by the composition of lower level features.

- 階層的な特徴の学習

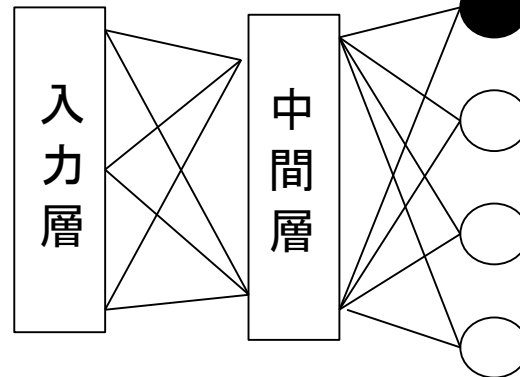
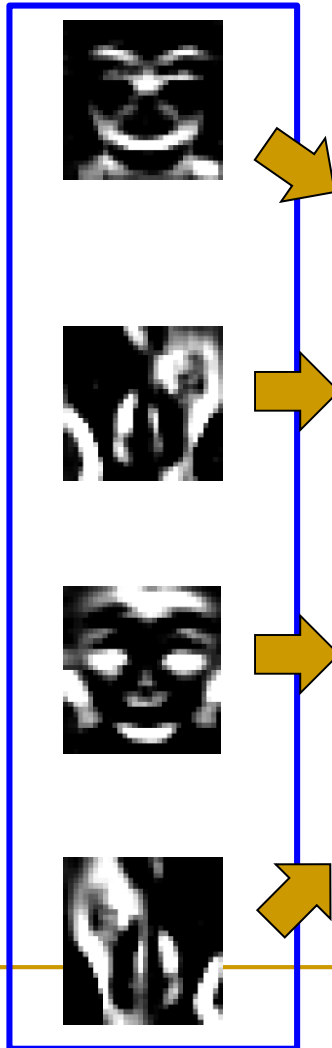
- ネットワークの下層部 局所的な特徴
- ネットワークの上層部 大局的な特徴

# (従来?の)ニューラルネットワークでの認識

特徴抽出処理

人手で設計

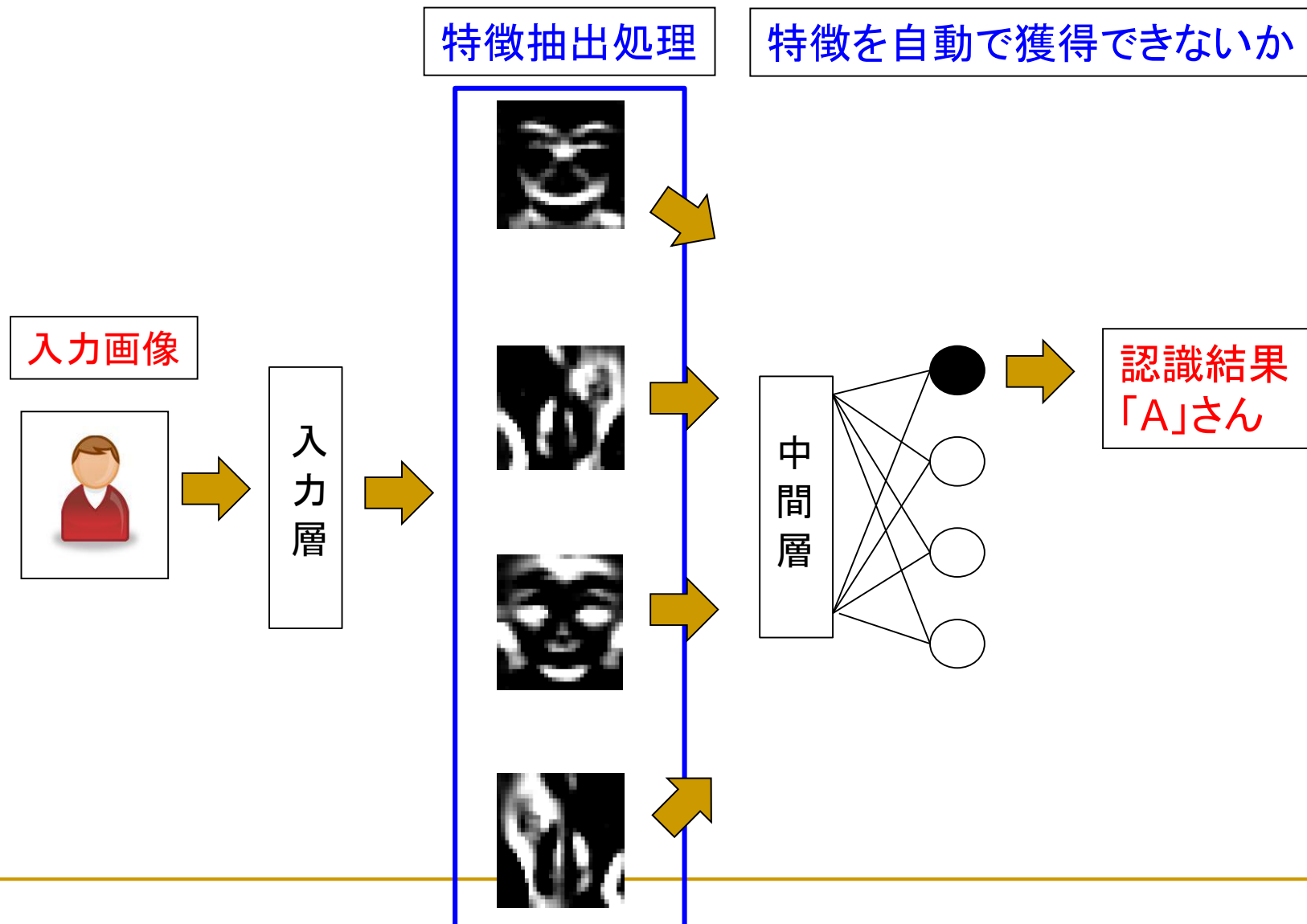
入力画像



認識結果  
「A」さん

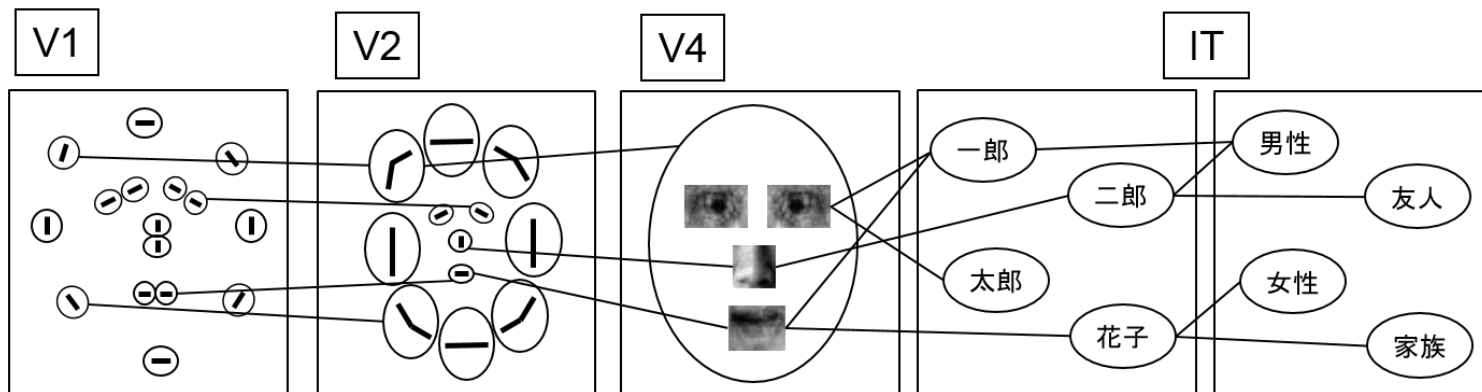


# ニューラルネットワークに期待すること

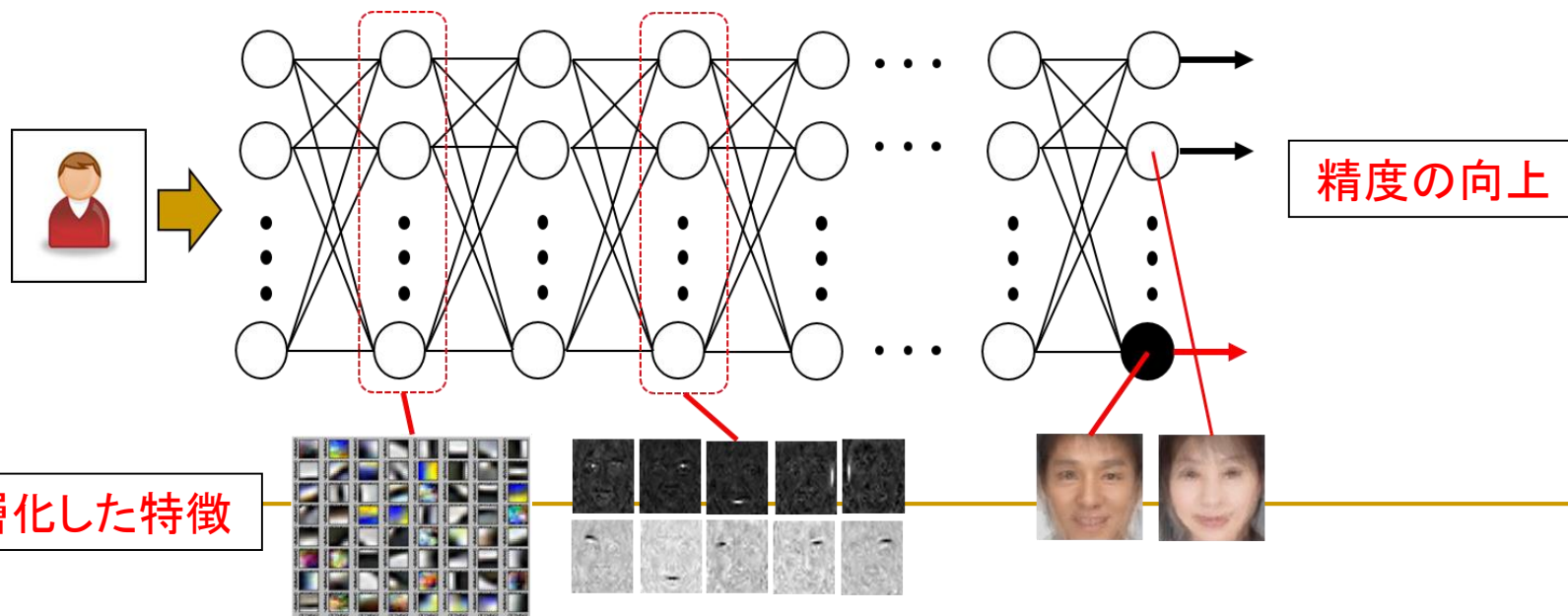


# 深層学習に期待すること

視覚野



深層学習



階層化した特徴

精度の向上

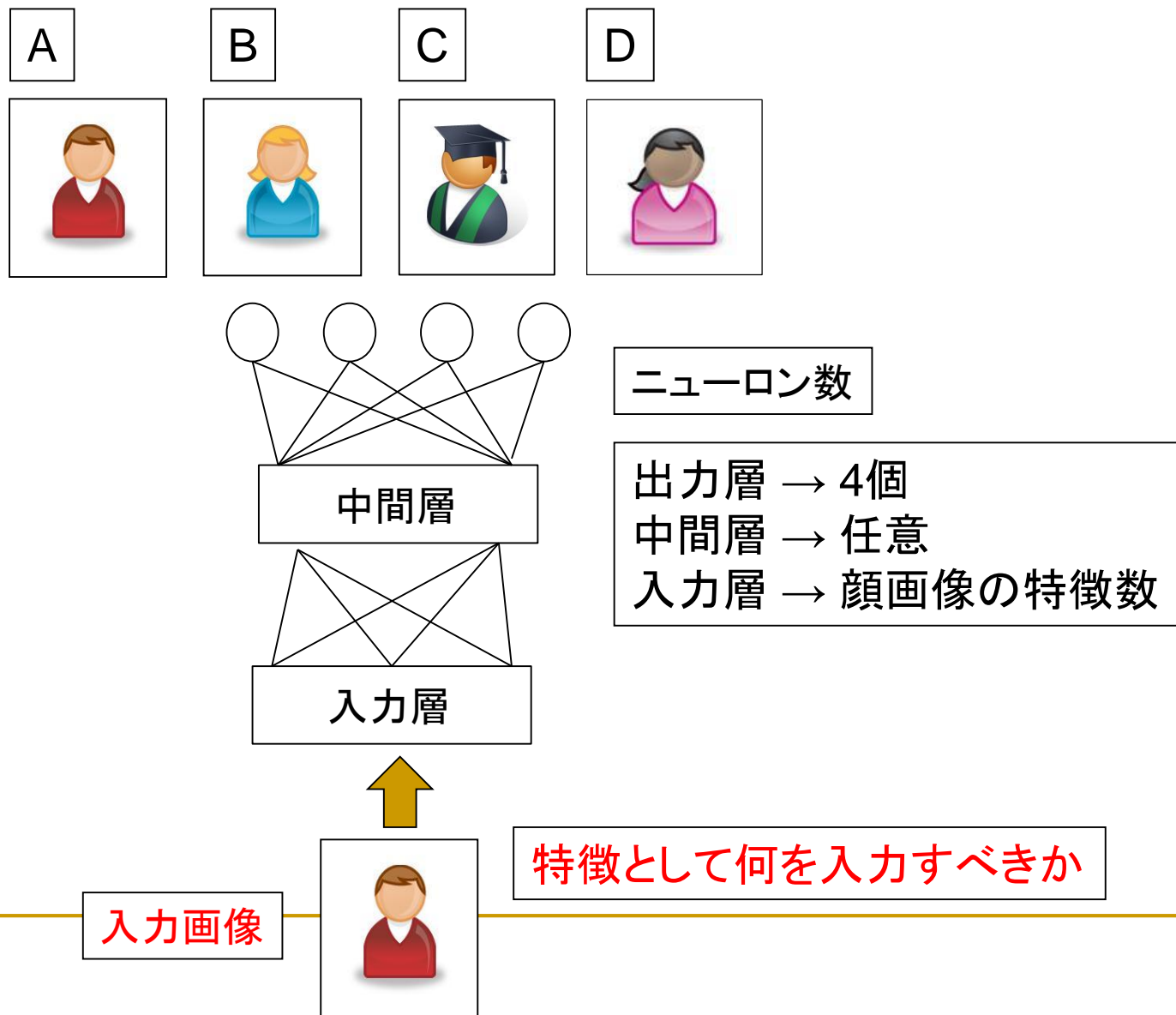
# 深層学習で考えるべきこと

- 単純に層を増やせばよいわけではない
- 階層化した特徴を学習できるようにするため,
  - ネットワークアーキテクチャをどうすべきか
  - 学習をどう行なうべきか
- ネットワークが複雑化, 巨大化した場合,
  - 学習方法をどう改良すべきか
- なぜ上手くいったのか説明できるようにする

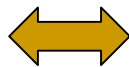
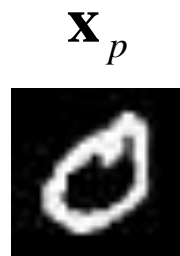
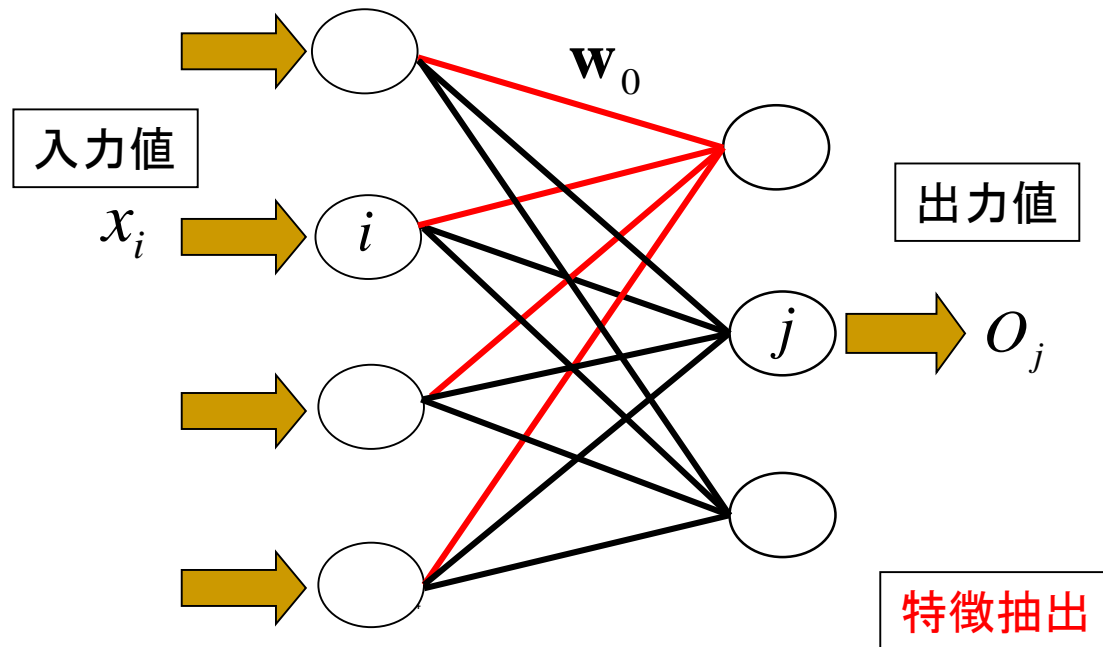
# オートエンコーダ

事前学習 (PreTraining)

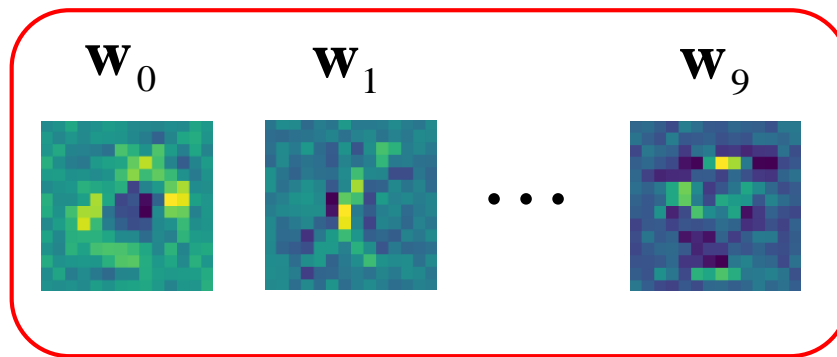
# ニューラルネットワークでの認識



# ニューラルネットワークの動作



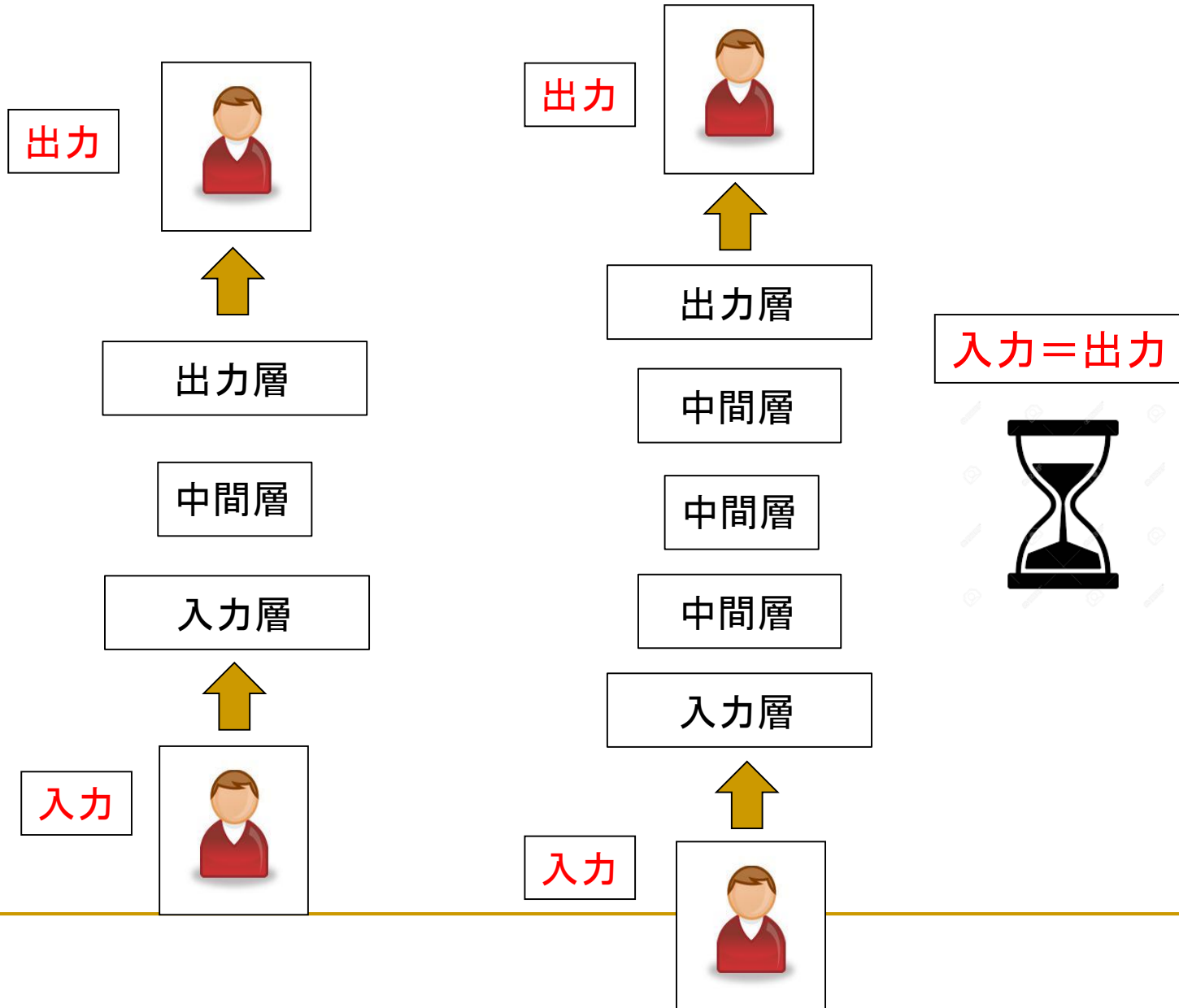
認識



# ニューラルネットワークによる特徴抽出

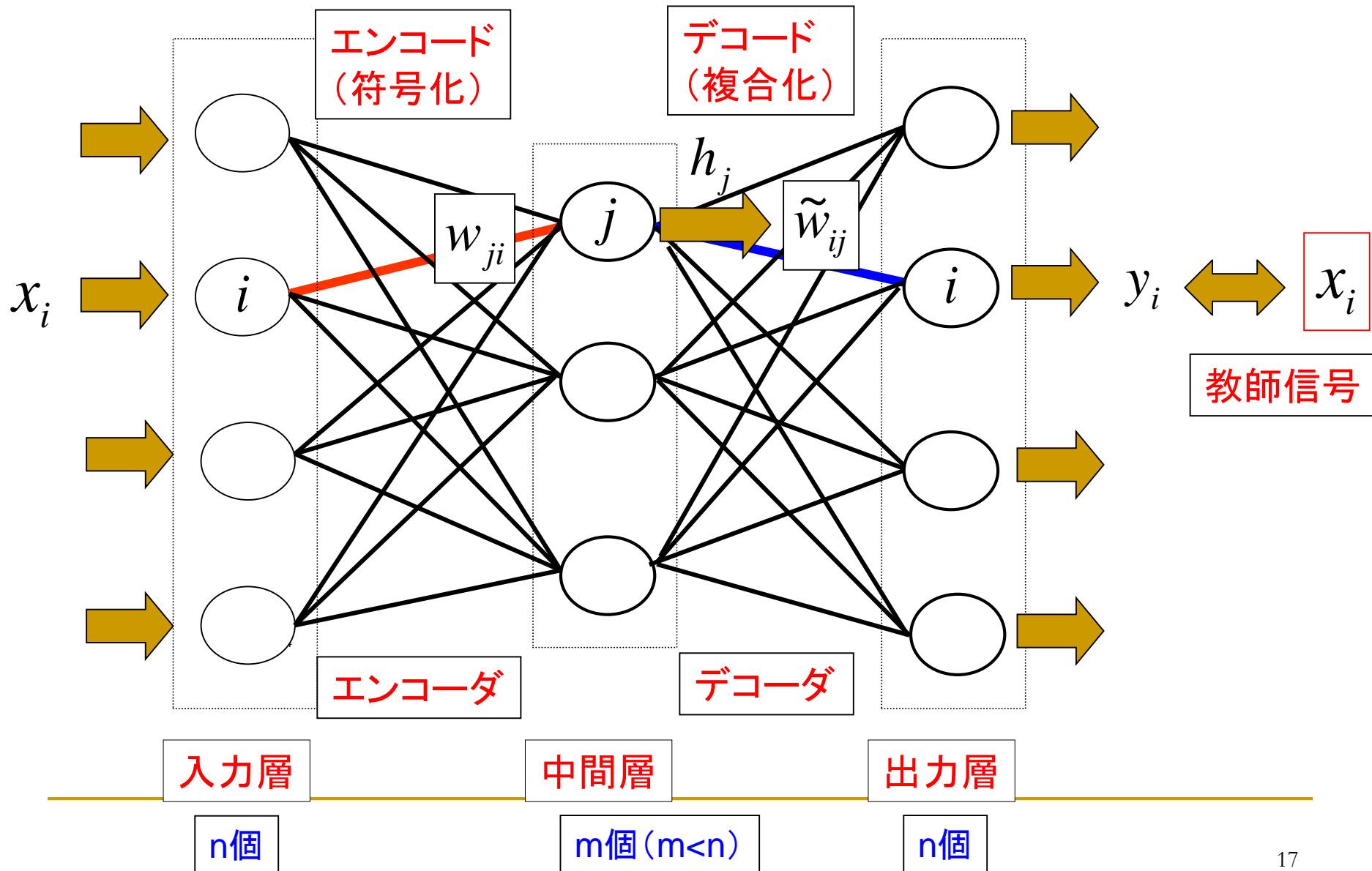
- 特徴とは
  - 入力情報の次元数を削減  
→ 代表的な手法として主成分分析
- ニューラルネットワークによる特徴抽出
  - ヘップの学習則
  - オートエンコーダ (Auto Encoder)

# オートエンコーダ（砂時計型ネットワーク）





# オートエンコーダ①



# オートエンコーダ②

## ■ オートエンコーダの動作

□ 出力値＝入力値       $\mathbf{y} = \mathbf{x}$

中間層

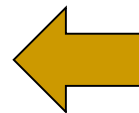
$$h_j = f\left(\sum_{i=1}^n w_{ji} x_i\right)$$

出力層

$$y_i = f\left(\sum_{j=1}^m \tilde{w}_{ij} h_j\right)$$

教師信号  
＝入力値

$x_i$



# オートエンコーダ③

## ■ オートエンコーダの動作

中間層

$$\mathbf{h} = f(W\mathbf{x} + \mathbf{b})$$

入力層と中間層  
結合係数  $W$   
閾値  $\mathbf{b}$

出力層

$$\begin{aligned}\mathbf{y} &= \tilde{f}(\tilde{W}\mathbf{h} + \tilde{\mathbf{b}}) \\ &= \tilde{f}(\tilde{W}f(W\mathbf{x} + \mathbf{b}) + \tilde{\mathbf{b}})\end{aligned}$$

中間層と出力層  
結合係数  $\tilde{W}$   
閾値  $\tilde{\mathbf{b}}$

$$\tilde{W} = W^t$$

重み共有

# オートエンコーダの学習①

- 出力値が実数値の場合 (活性化関数: 恒等関数)

$$E = \sum_{p=1}^P \| \mathbf{y}_p - \mathbf{x}_p \|^2 = \sum_{p=1}^P \| (\tilde{W}f(W\mathbf{x}_p + \mathbf{b}) + \tilde{\mathbf{b}}) - \mathbf{x}_p \|^2$$

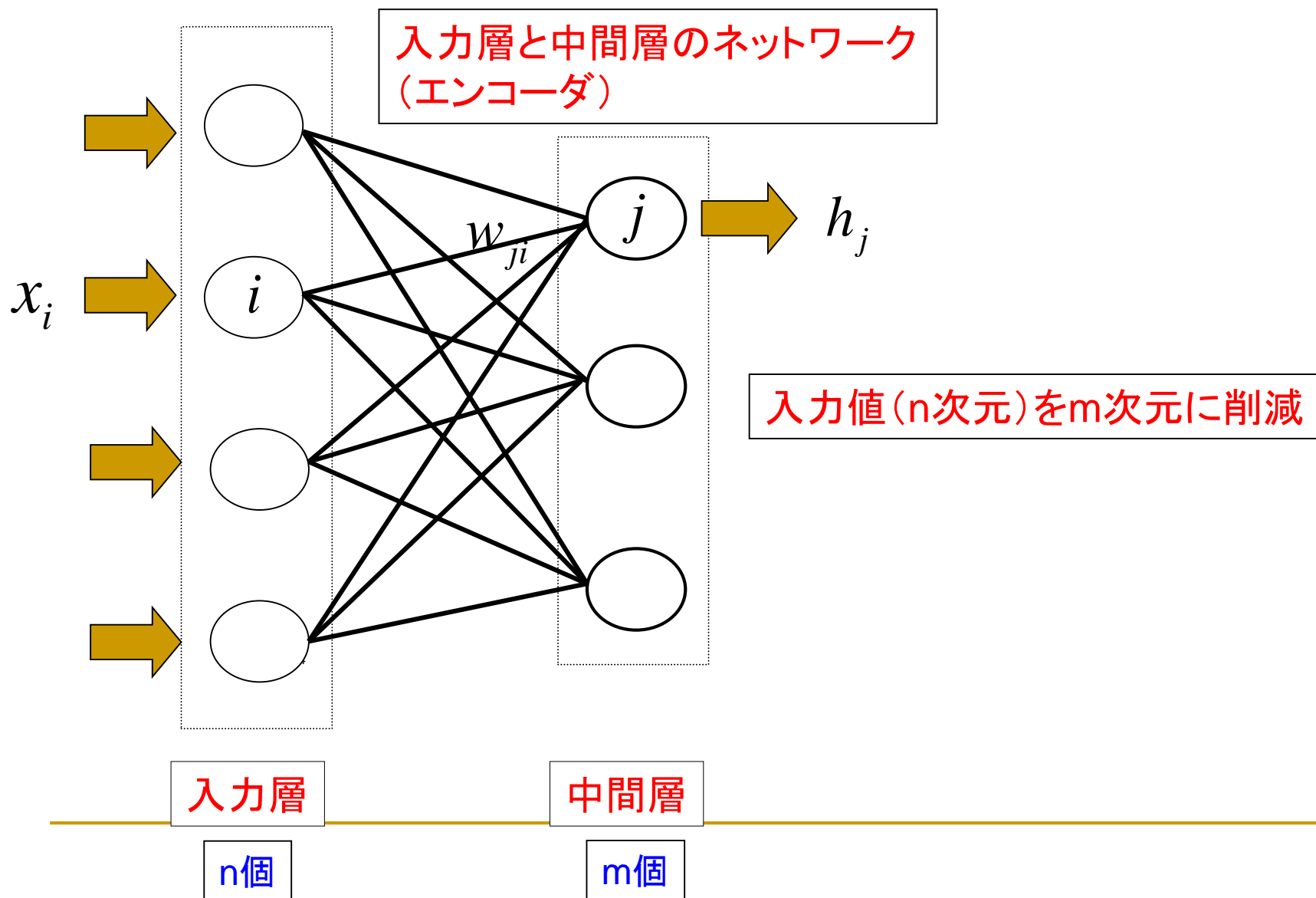
誤差二乗和

- 出力値が二値の場合 (活性化関数: シグモイド関数)

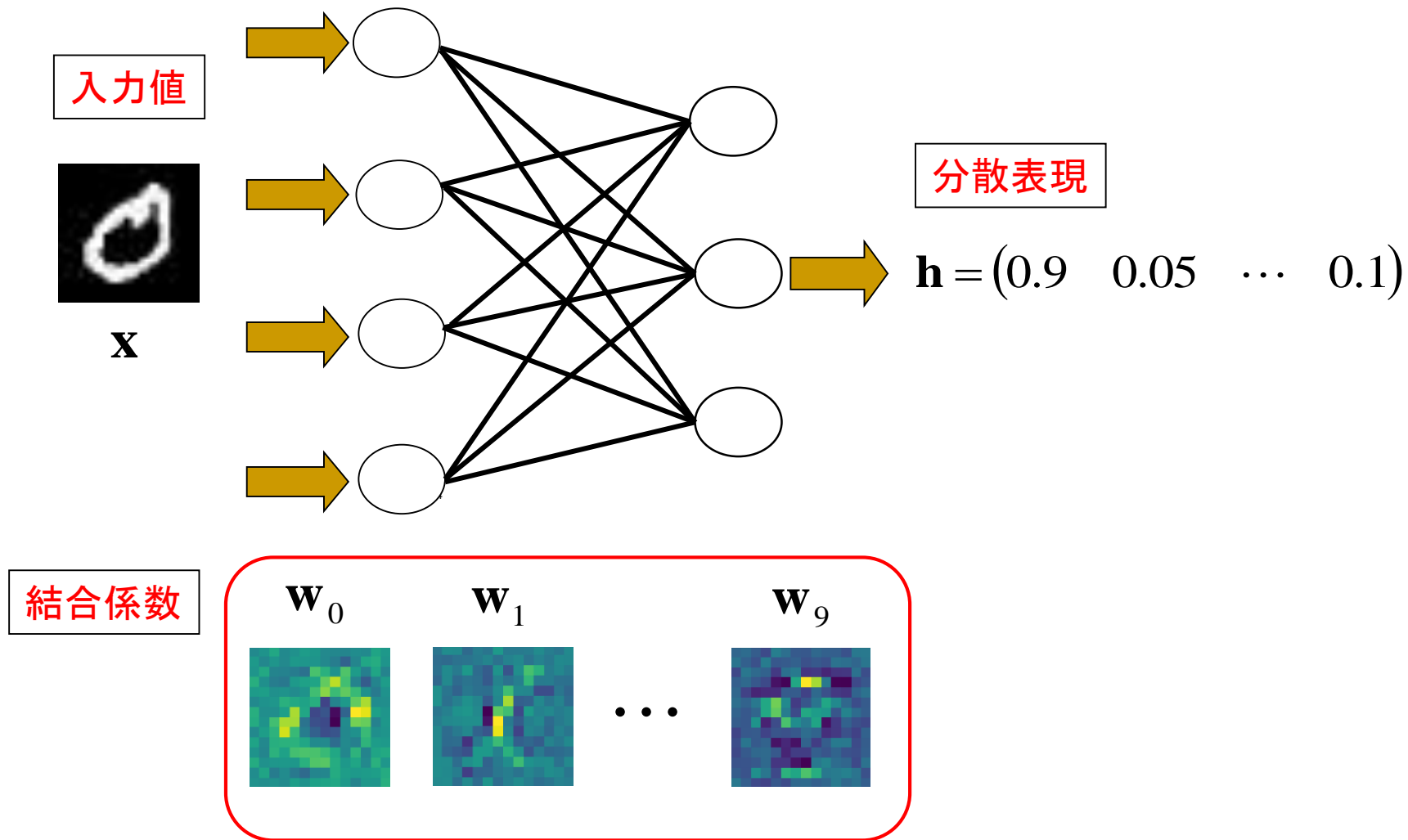
$$E = - \sum_{p=1}^P \sum_{i=1}^n (x_{pi} \log y_{pi} + (1 - x_{pi}) \log(1 - y_{pi}))$$

交差エントロピー

# 特徴抽出(入力情報の次元削減)

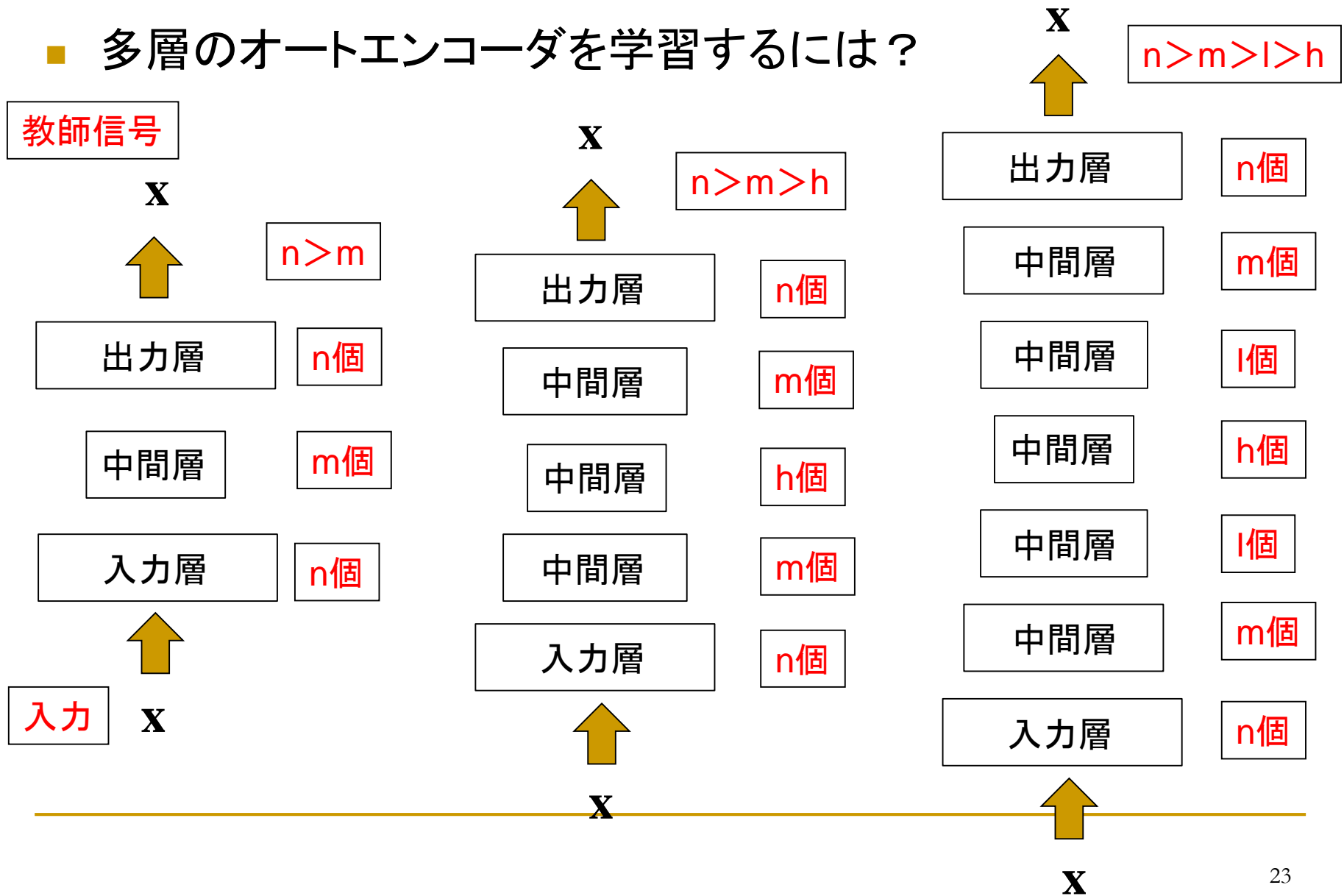


# 分散表現

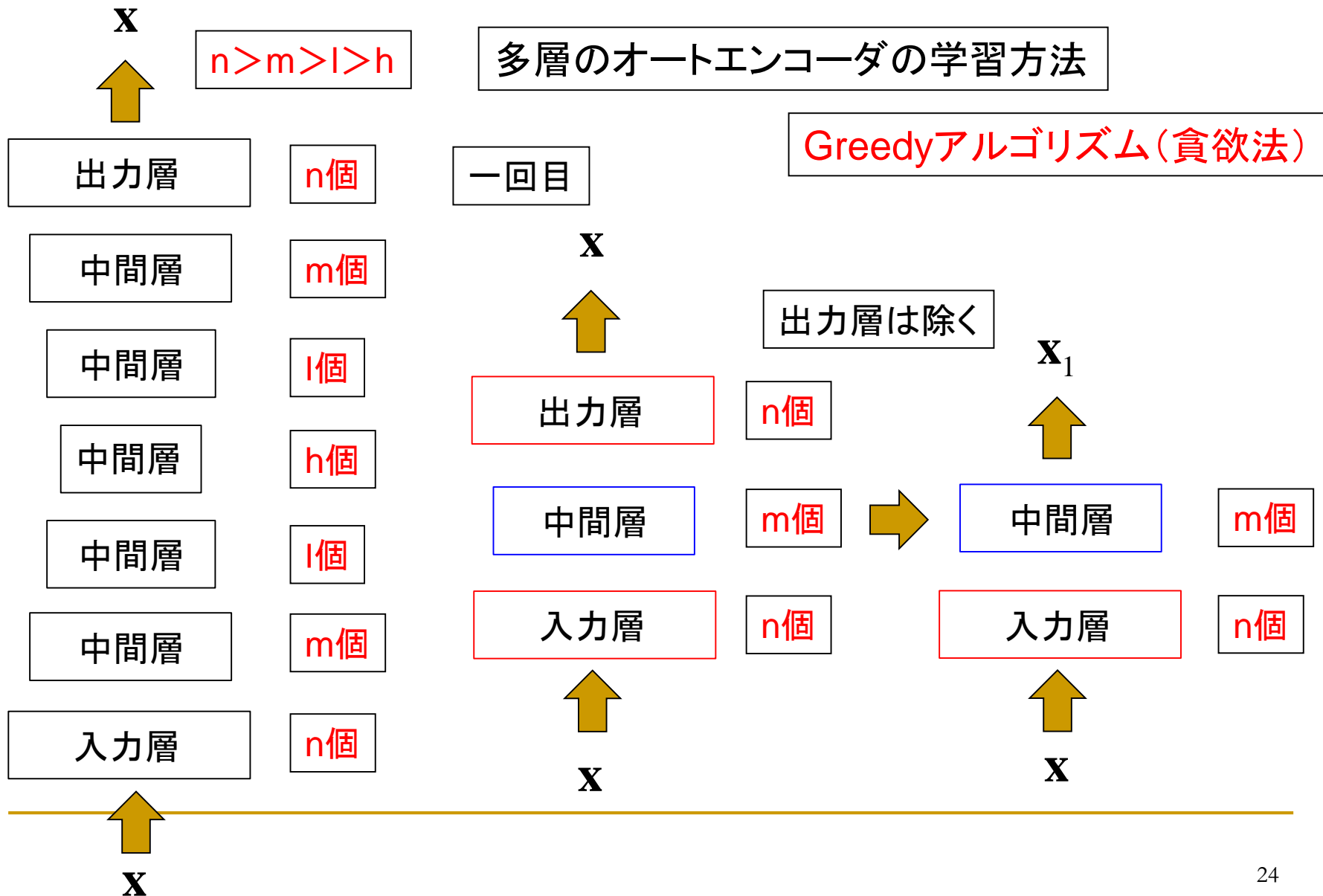


# オートエンコーダの学習②

- 多層のオートエンコーダを学習するには？

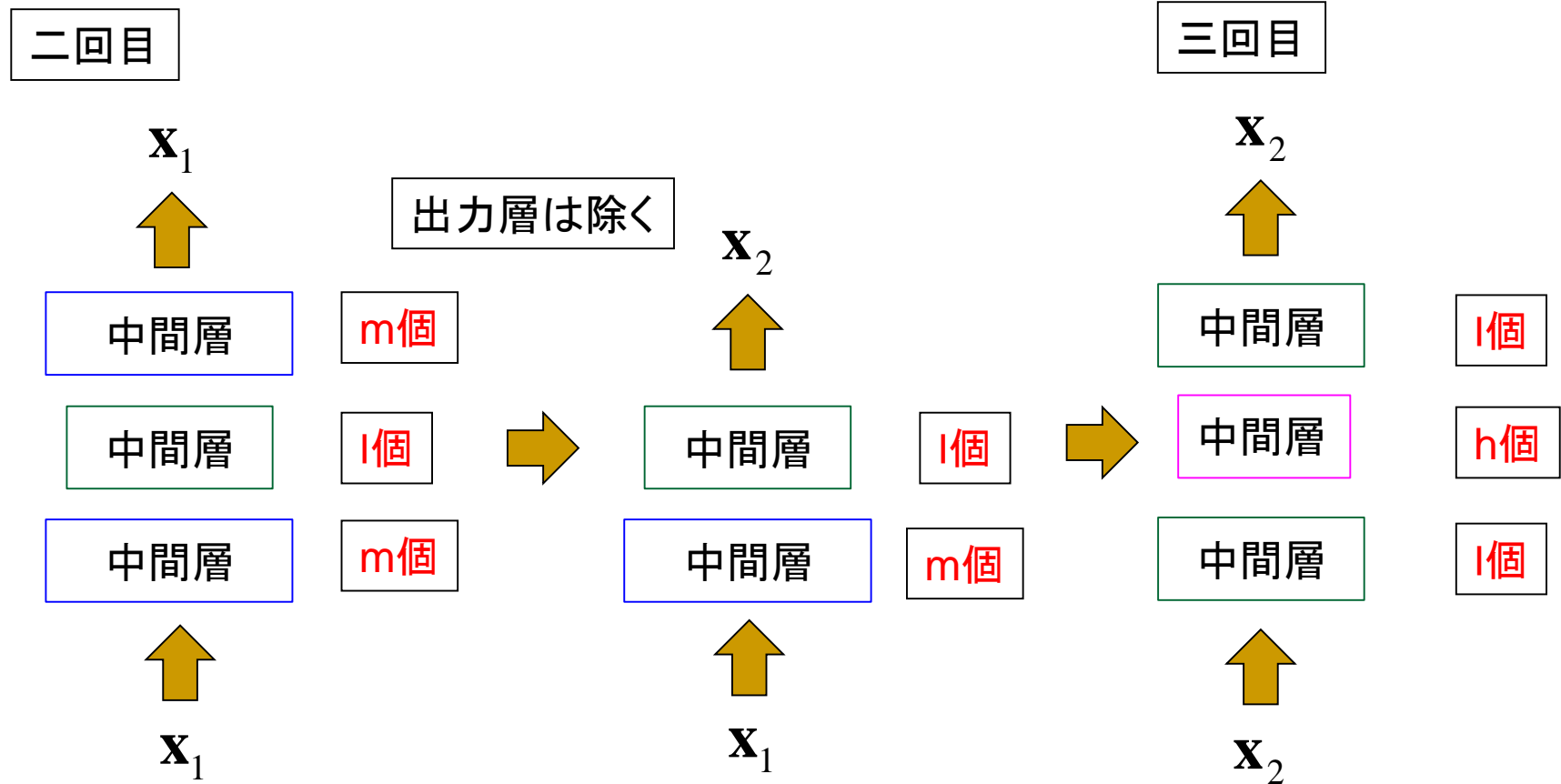


# Stackedオートエンコーダ①

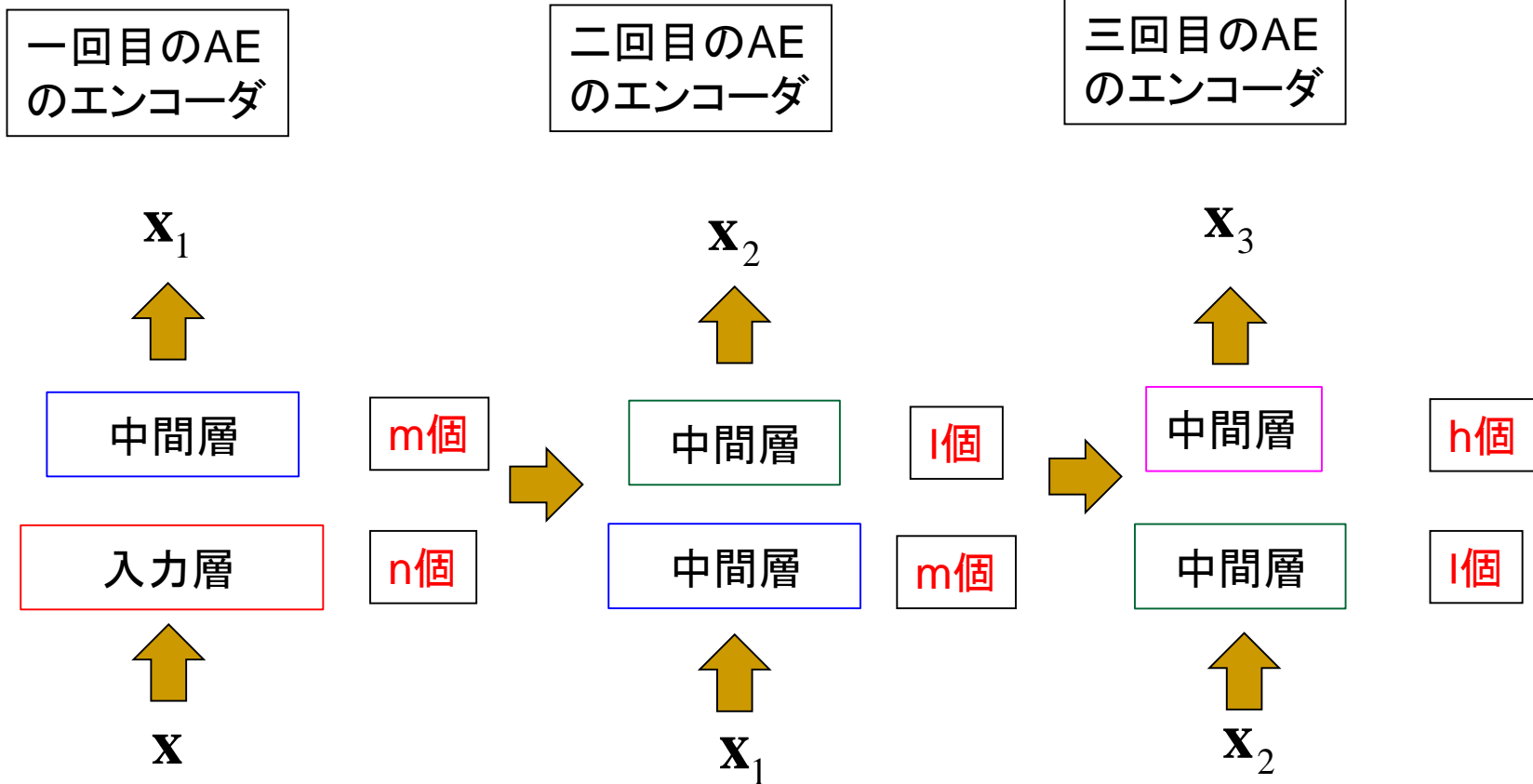




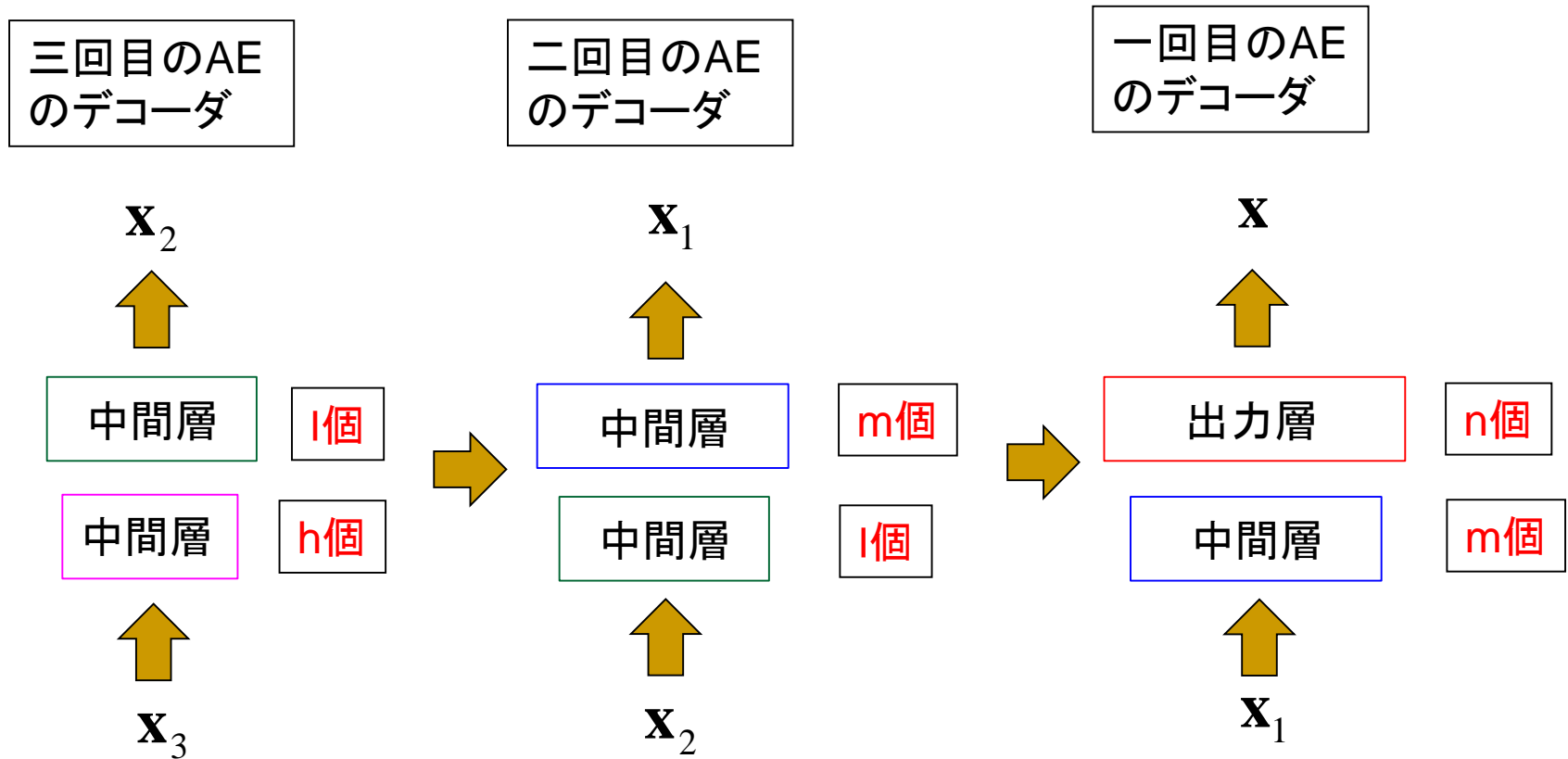
# Stackedオートエンコーダ②



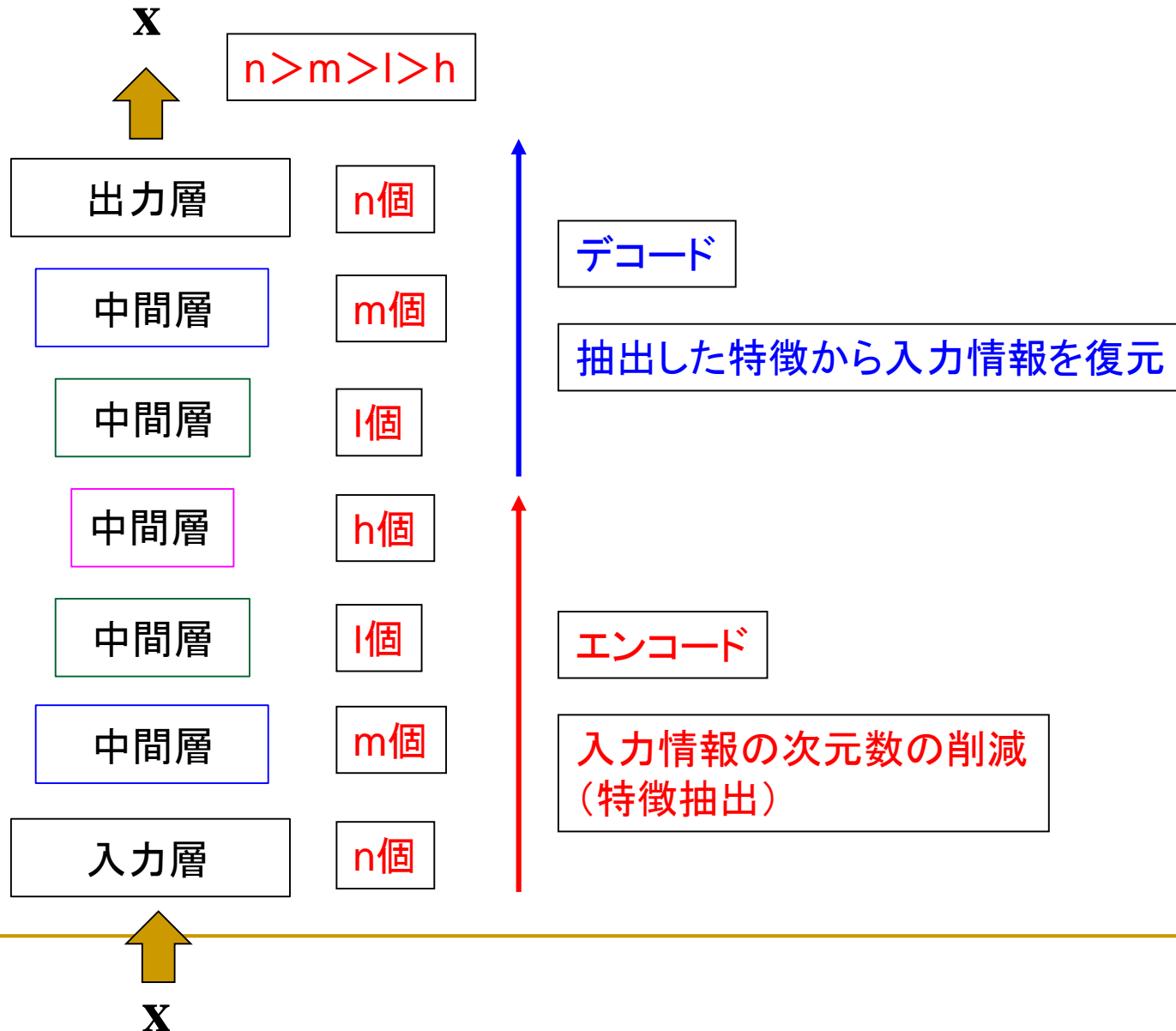
# Stackedオートエンコーダ③



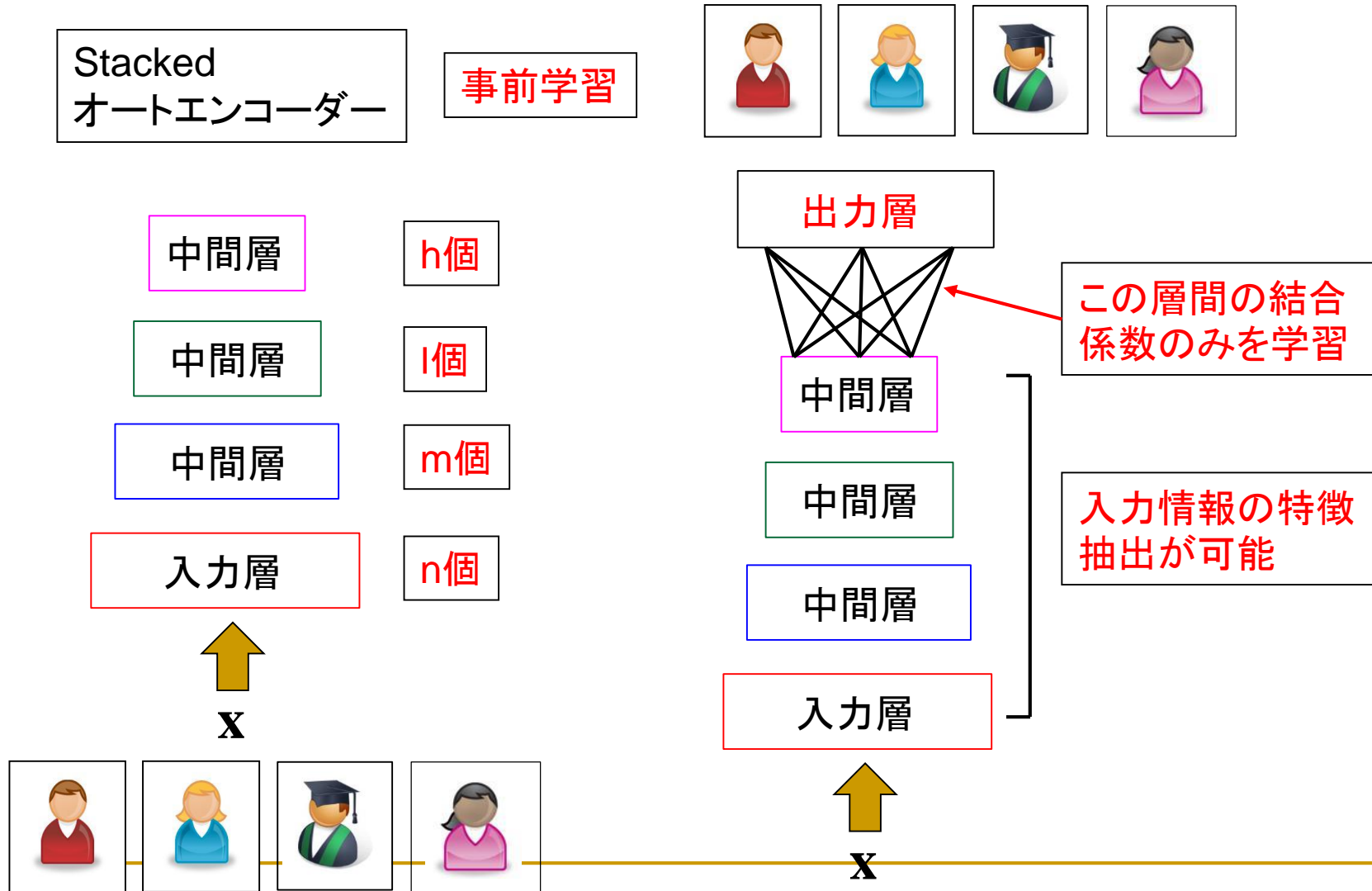
# Stackedオートエンコーダ④



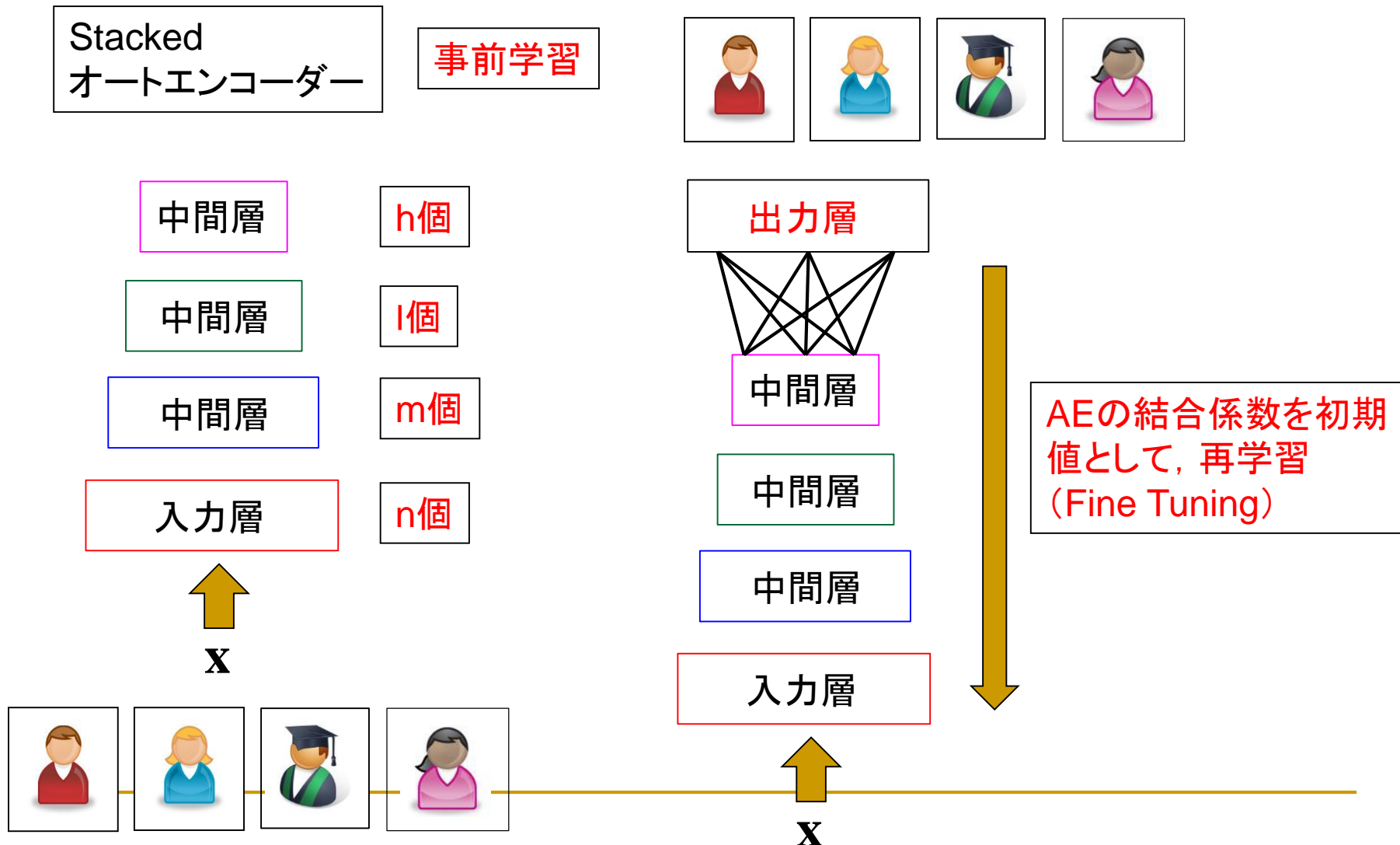
# Stackedオートエンコーダ⑤



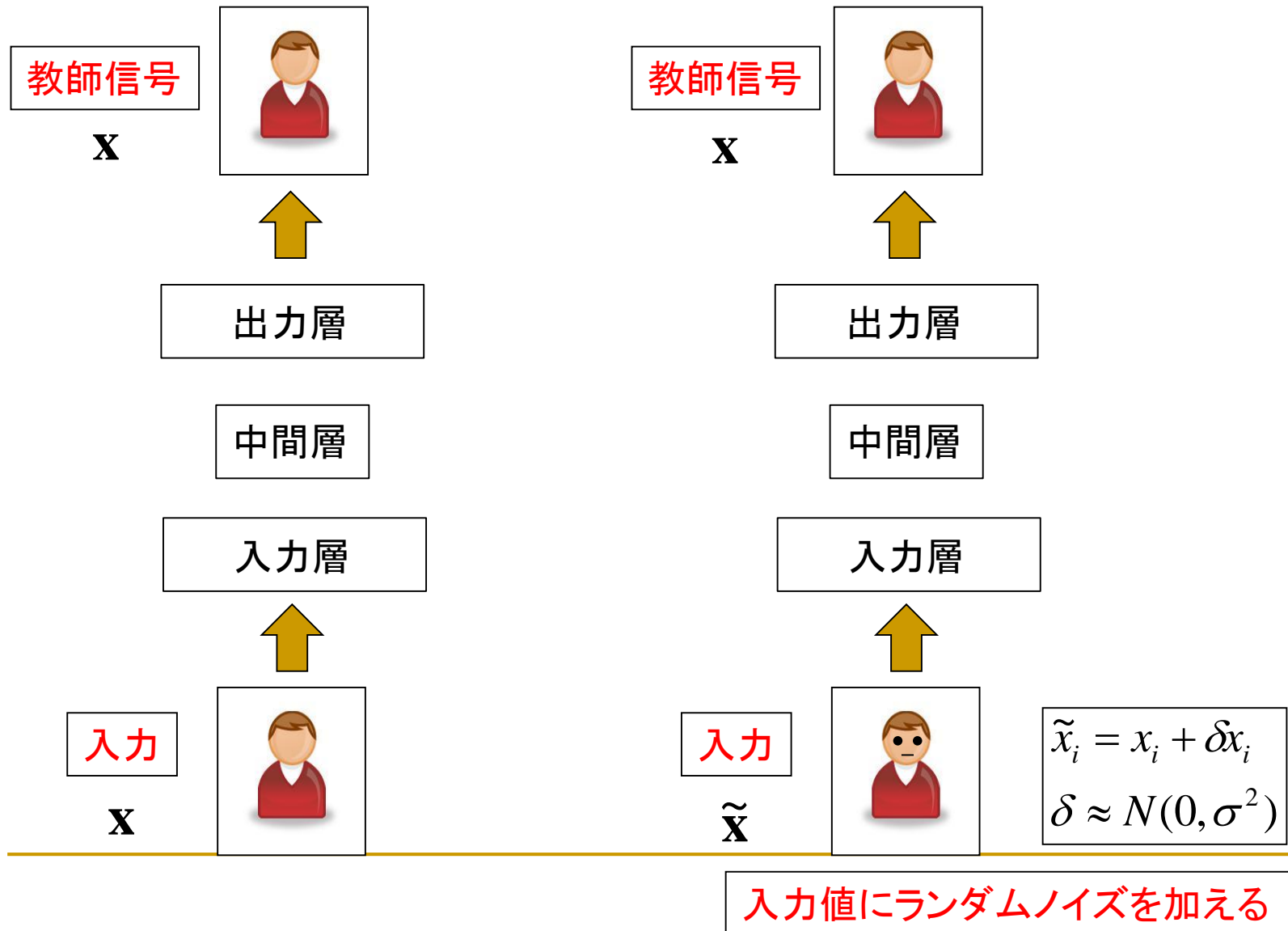
# 事前学習 (PreTraining) ①



# 事前学習 (PreTraining) ②



# Denoisingオートエンコーダ①



# Denoisingオートエンコーダ②

## ■ 学習

$$E = \sum_{p=1}^P \| \mathbf{y}_p - \mathbf{x}_p \|^2$$

教師信号→ノイズの含まれていない入力

## ■ Denoisingオートエンコーダ

- 入力のノイズの除去が可能
- ノイズを含んだ入力からの特徴抽出が可能



# エンコーダ・デコーダの応用

## ■ 画像処理

- 畳み込み(逆畳み込み)ニューラルネットワーク
- セマンティックセグメンテーション
- 画像変換, 画像生成

## ■ 自然言語処理

- Sequence to Sequenceモデル
- 機械翻訳
- 対話文応答
- 文章の自動生成

# 畳み込みニューラルネットワーク (Convolution Neural Network)

空間フィルタリング処理

畳み込み層

プーリング層

# 空間フィルタリング処理(畳み込み処理)①

- 各画素について、その画素周辺の $N \times N$ 画素の小領域と、 $N \times N$ の空間フィルタとの積和を行なう
- 入力画像を $f$ 、空間フィルタを $h$ とした場合、下記の式に基づいて変換後の画素値 $g$ を求める

畳み込み処理

$$g(x, y) = \sum_{k=0}^{N-1} \sum_{l=0}^{N-1} h(k, l) f(x + k, y + l)$$

## 空間フィルタリング処理(畳み込み処理)②

3×3の空間フィルタ h

0	-1	0
-1	5	-1
0	-1	0

変換する画素 (x,y)

40	60	40
80	110	80
100	100	100

$$\begin{aligned} &40 \times 0 + 60 \times (-1) + 40 \times 0 + \\ &80 \times (-1) + 110 \times 5 + 80 \times (-1) + \\ &100 \times 0 + 100 \times (-1) + 100 \times 0 \\ &= 230 \end{aligned}$$

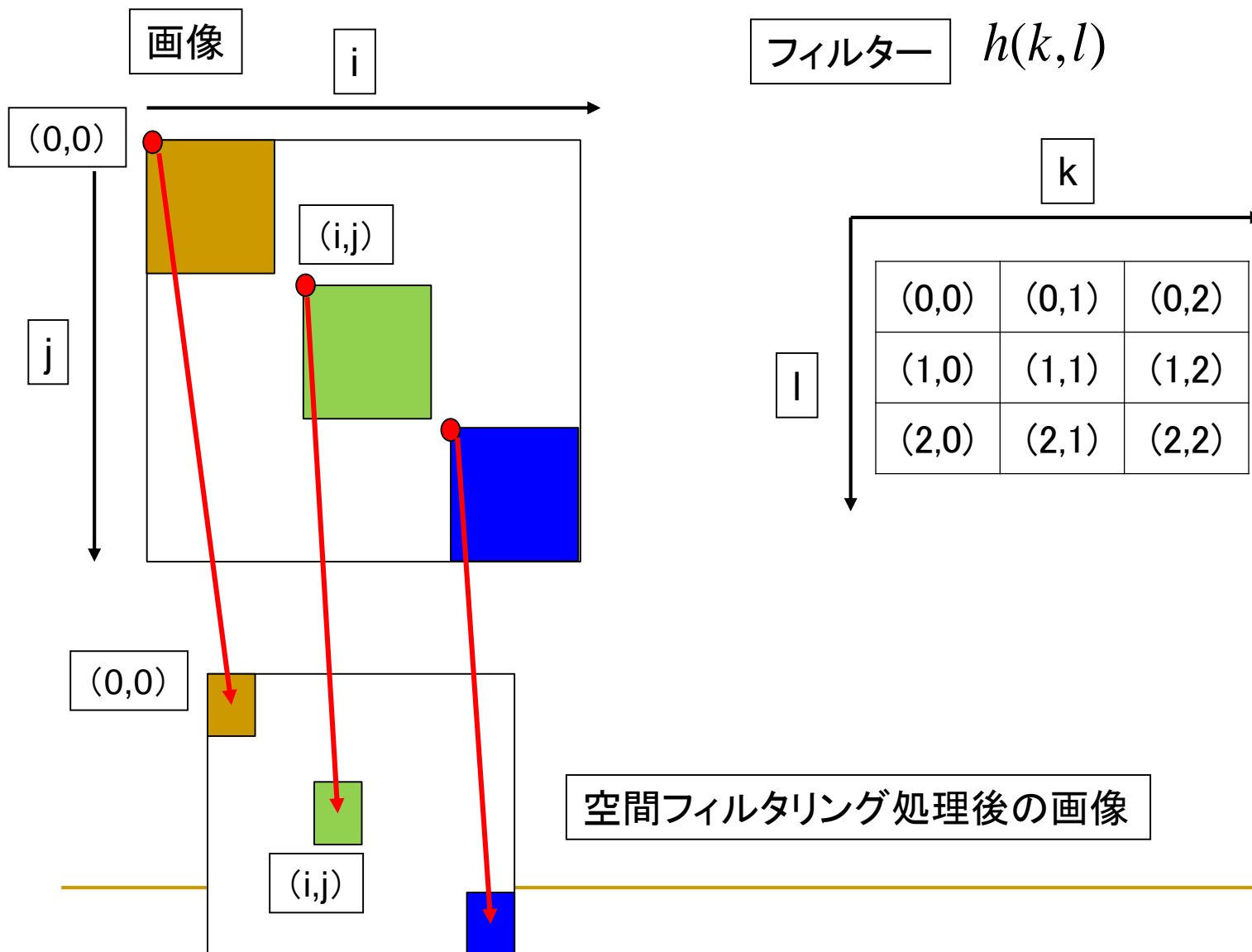
変換後の画素値 g

230

周辺の3×3の小領域 f

以上の処理を全ての画素で行なう

# 空間フィルタリング処理(畳み込み処理)③



# 空間フィルタリング処理(畳み込み処理)④

入力画像  $f(x,y)$

2	4	1	3	5
3	2	6	2	8
1	0	3	4	2
6	2	1	7	5
5	3	2	5	6

平滑化フィルタ  $h$

$$h(k,l) = \frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

出力画像  $g(x,y)$

$(2+4+1+3+2+6+1+0+3)/9$

2.444444	2.777778	3.777778
2.666667	3	4.222222
2.555556	3	3.888889

$(3+4+2+1+7+5+2+5+6)/9$

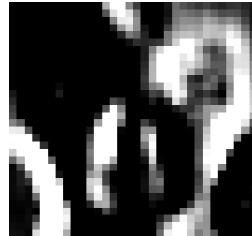
元の画像よりも小さくなってしましますが、同じ大きさにすることも可能です(パディング)

# 空間フィルタリング処理(畳み込み処理)⑤

元画像



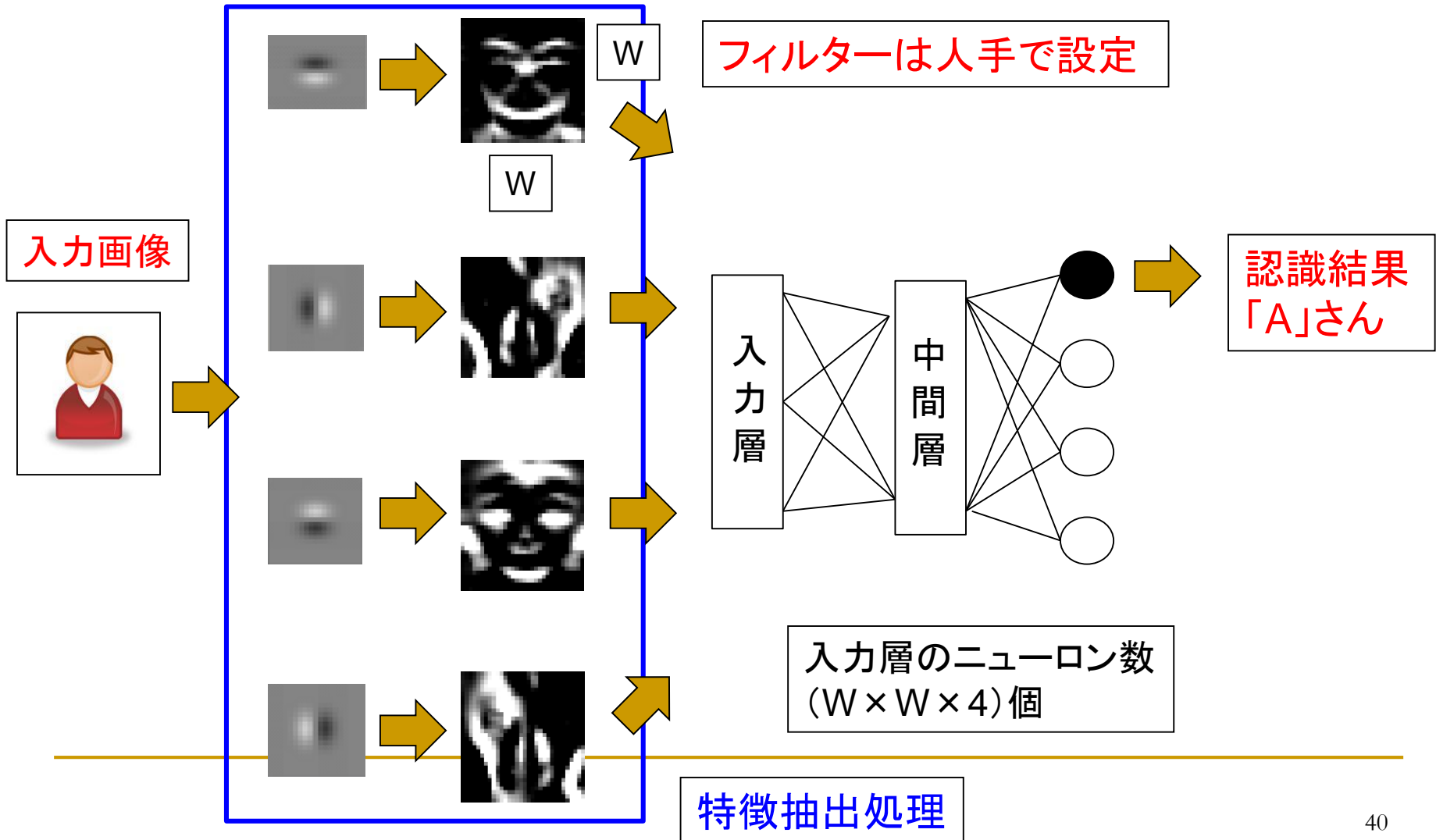
ガボールフィルター



フィルタリング後の画像

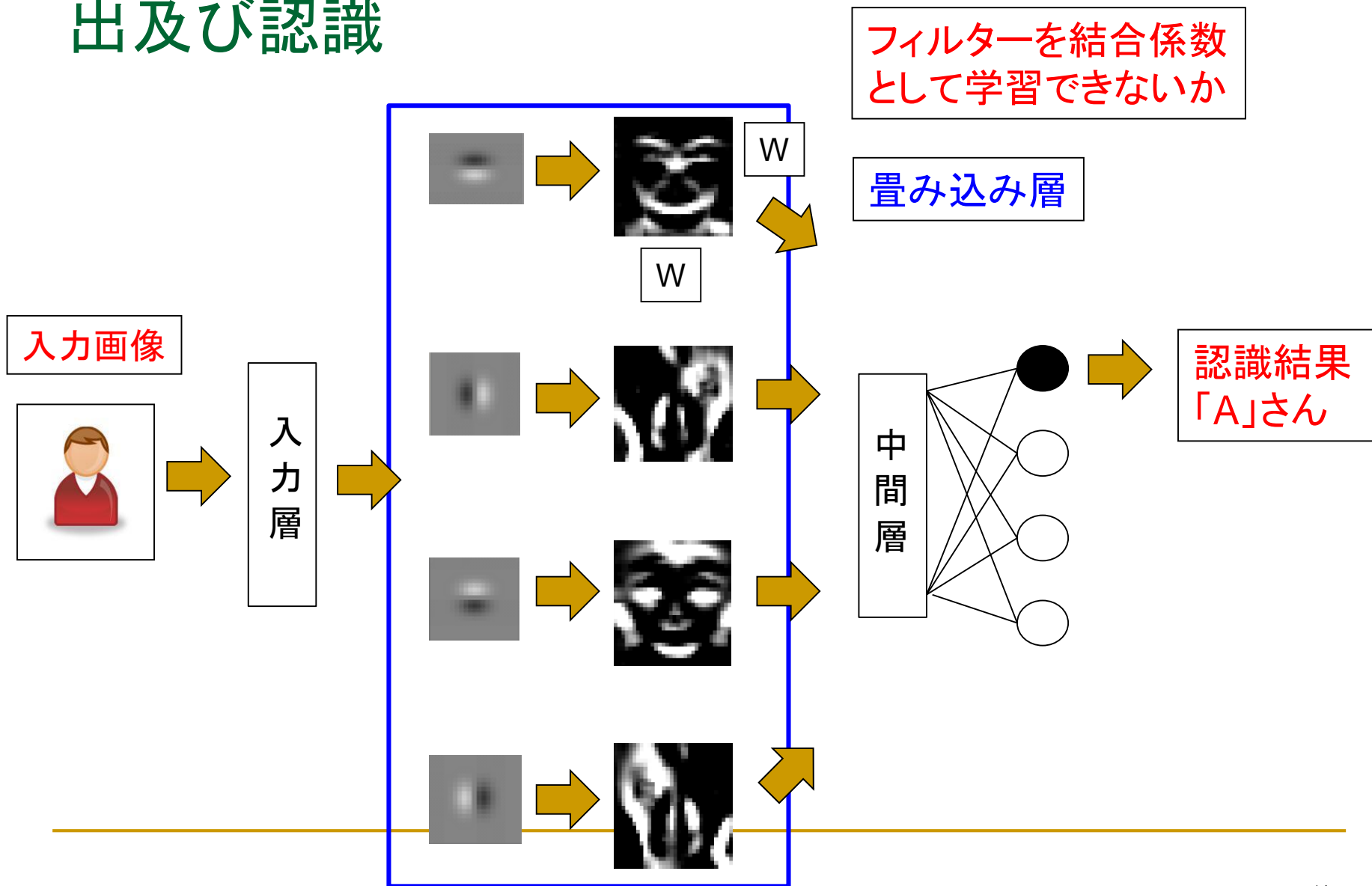
# (従来？の)ニューラルネットワークでの認識

空間フィルタリング処理

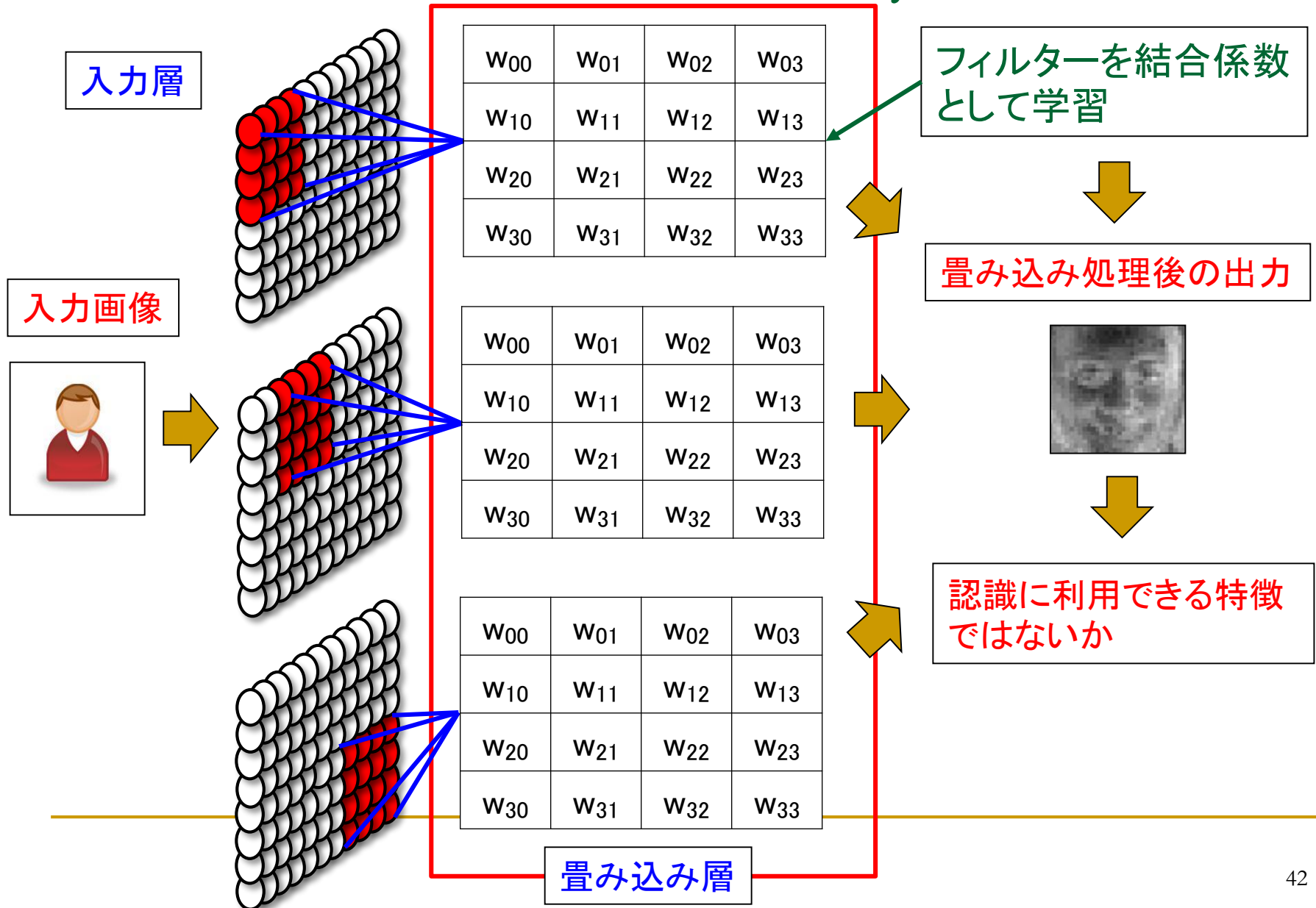




# (近年?の)ニューラルネットワークでの特徴抽出及び認識



# 畳み込み層 (Convolution Layer)



# 畳み込み層 (Convolution Layer)

入力層

入力画像

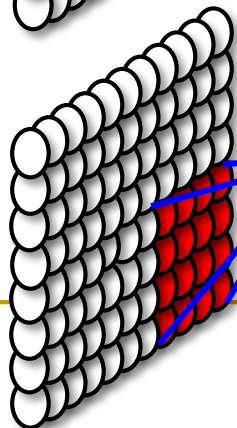
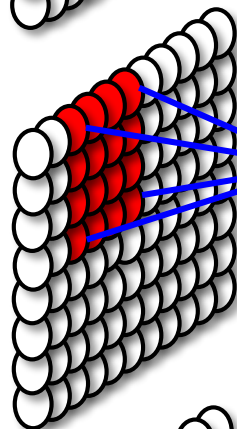
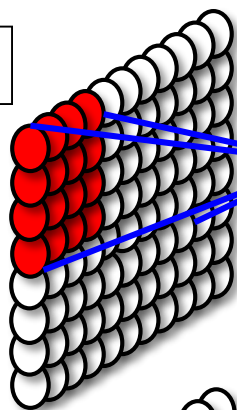
Y

X

$$f(x, y)$$

$$x = 0, 1, \dots, X - 1$$

$$y = 0, 1, \dots, Y - 1$$



$W_{00}$	$W_{01}$	$W_{02}$	$W_{03}$
$W_{10}$	$W_{11}$	$W_{12}$	$W_{13}$
$W_{20}$	$W_{21}$	$W_{22}$	$W_{23}$
$W_{30}$	$W_{31}$	$W_{32}$	$W_{33}$

$W_{00}$	$W_{01}$	$W_{02}$	$W_{03}$
$W_{10}$	$W_{11}$	$W_{12}$	$W_{13}$
$W_{20}$	$W_{21}$	$W_{22}$	$W_{23}$
$W_{30}$	$W_{31}$	$W_{32}$	$W_{33}$

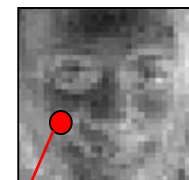
$W_{00}$	$W_{01}$	$W_{02}$	$W_{03}$
$W_{10}$	$W_{11}$	$W_{12}$	$W_{13}$
$W_{20}$	$W_{21}$	$W_{22}$	$W_{23}$
$W_{30}$	$W_{31}$	$W_{32}$	$W_{33}$

フィルター (結合係数)

$$w(p, q)$$

$$p = 0, 1, \dots, N - 1$$

$$q = 0, 1, \dots, N - 1$$



$$g(x, y) = \sum_{p=0}^{N-1} \sum_{q=0}^{N-1} f(x+p, y+q) w(p, q)$$

畳み込み層

# パディング

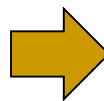
入力画像  $f(x,y)$

2	4	1	3	5
3	2	6	2	8
1	0	3	4	2
6	2	1	7	5
5	3	2	5	6

$$h(k,l) = \frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

出力画像  $g(x,y)$

2.444	2.778	3.778
2.667	3.000	4.222
2.556	3.000	3.889

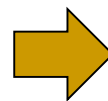


画像の大きさが小さくなる

周囲を0(ゼロパディング)

0	0	0	0	0	0	0
0	2	4	1	3	5	0
0	3	2	6	2	8	0
0	1	0	3	4	2	0
0	6	2	1	7	5	0
0	5	3	2	5	6	0
0	0	0	0	0	0	0

$$(0+0+0+0+2+4+0+3+2)/9$$



1.222	2.000	2.000	2.778	2.000
1.333	2.444	2.778	3.778	2.667
1.556	2.667	3.000	4.222	3.111
1.889	2.556	3.000	3.889	3.222
1.778	2.111	2.222	2.889	2.556
0.889	1.111	1.111	1.444	1.222

画像の大きさは変わらない

# ストライド①

ストライドが1の場合

2	4	1	3	5	2	1
3	2	6	2	8	3	4
1	0	3	4	2	1	2
6	2	1	7	5	2	4
5	3	2	5	6	1	7
3	4	1	6	7	0	3
1	4	0	8	5	2	8

入力画像  $f(x,y)$

$$h(k,l) = \frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$



2.444	2.778	3.778	3.333	3.111
2.667	3.000	4.222	3.778	3.444
2.556	3.000	3.889	3.667	3.333
3.000	3.444	4.444	4.333	3.889
2.556	3.667	4.444	4.444	4.333

出力画像  $g(x,y)$

# ストライド②

ストライドが1の場合

2	4	1	3	5	2	1
3	2	6	2	8	3	4
1	0	3	4	2	1	2
6	2	1	7	5	2	4
5	3	2	5	6	1	7
3	4	1	6	7	0	3
1	4	0	8	5	2	8

入力画像  $f(x,y)$

$$h(k,l) = \frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$



2.444	2.778	3.778	3.333	3.111
2.667	3.000	4.222	3.778	3.444
2.556	3.000	3.889	3.667	3.333
3.000	3.444	4.444	4.333	3.889
2.556	3.667	4.444	4.444	4.333

出力画像  $g(x,y)$

# ストライド③

ストライドが2の場合

2	4	1	3	5	2	1
3	2	6	2	8	3	4
1	0	3	4	2	1	2
6	2	1	7	5	2	4
5	3	2	5	6	1	7
3	4	1	6	7	0	3
1	4	0	8	5	2	8

入力画像  $f(x,y)$

$$h(k,l) = \frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$



2.444	3.778	3.111
2.556	3.889	3.333
2.556	4.444	4.333

出力画像  $g(x,y)$

# ストライド④

ストライドが2の場合

2	4	1	3	5	2	1
3	2	6	2	8	3	4
1	0	3	4	2	1	2
6	2	1	7	5	2	4
5	3	2	5	6	1	7
3	4	1	6	7	0	3
1	4	0	8	5	2	8

入力画像  $f(x,y)$

$$h(k,l) = \frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$



2.444	3.778	3.111
2.556	3.889	3.333
2.556	4.444	4.333

出力画像  $g(x,y)$

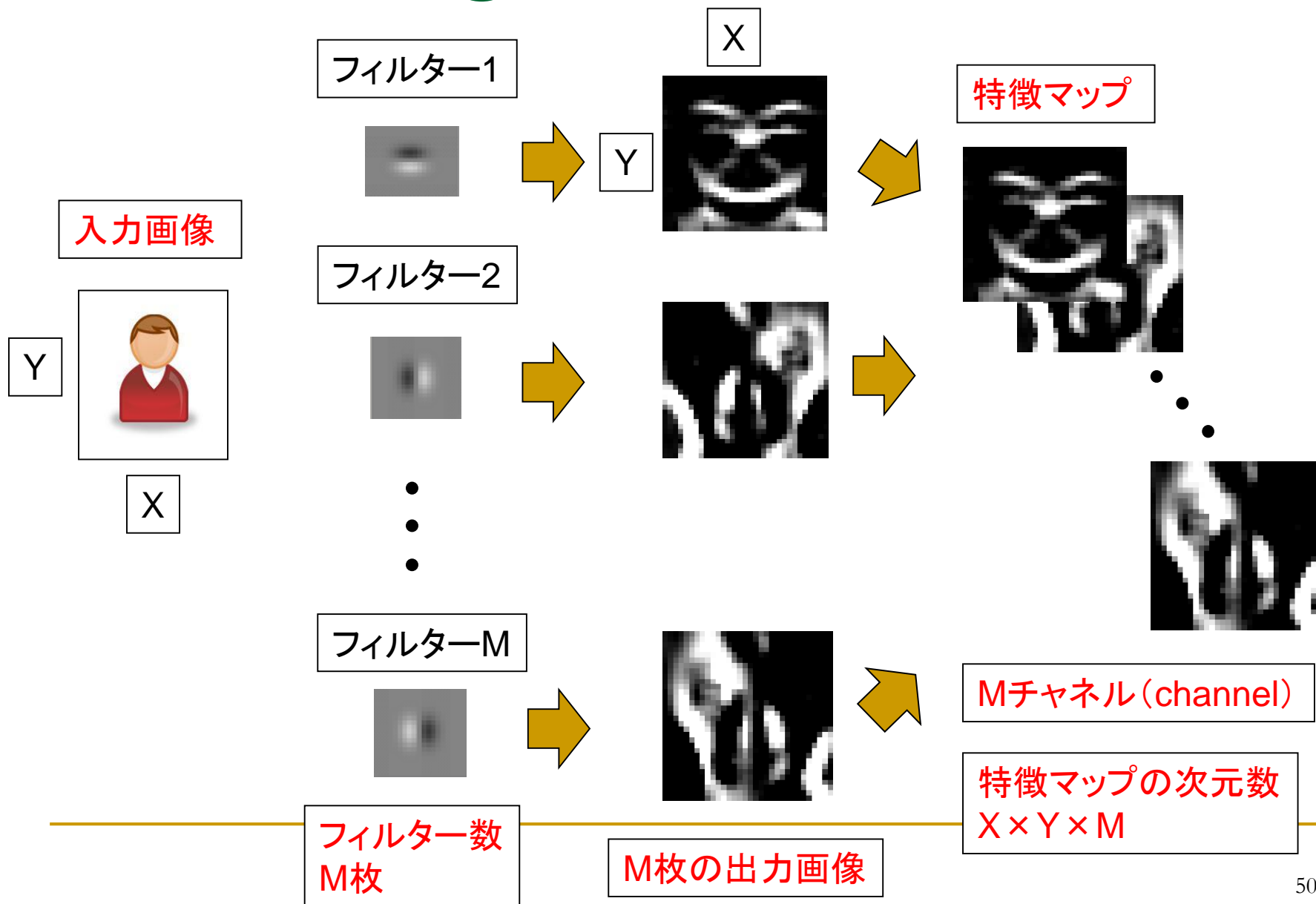


# 畳み処理後の画像\*の大きさ

- スライド, パディングによって畳み処理後の画像(特徴マップ)の大きさを変更可能
- 入力画像の大きさ  $I_W \times I_H$
- フィルターの大きさ  $F_W \times F_H$
- スライド  $s$
- パディング  $p$
- 畳み処理後の特徴マップの大きさ
- $O_W \times O_H = ((I_W - F_W + 2p) / s + 1) \times ((I_H - F_H + 2p) / s + 1)$

\*特徴マップと呼びます

# 特徴マップ①



# 特徴マップ②

## 表記方法

(バッチサイズ, チャンネル数, Y, X)

### ■ 入力画像 (1枚 = バッチサイズ = 1)

- グレースケール画像 (1チャンネル)
- 大きさ  $X \times Y$
- 次元数  $X \times Y \times 1$

(1, 1, Y, X)

チャンネル数は同じ

### ■ フィルター

- 枚数 M枚 (1チャンネル)
- 大きさ  $W \times W$
- 次元数  $W \times W \times M$

## 表記方法

(枚数, チャンネル数, Y, X)  
もしくは (枚数, Y, X)

(M, 1, W, W)

(M, W, W)

### ■ 畳み処理後の特徴マップ

- 枚数 M枚チャンネル
- 大きさ  $X \times Y^*$
- 次元数  $X \times Y \times M$

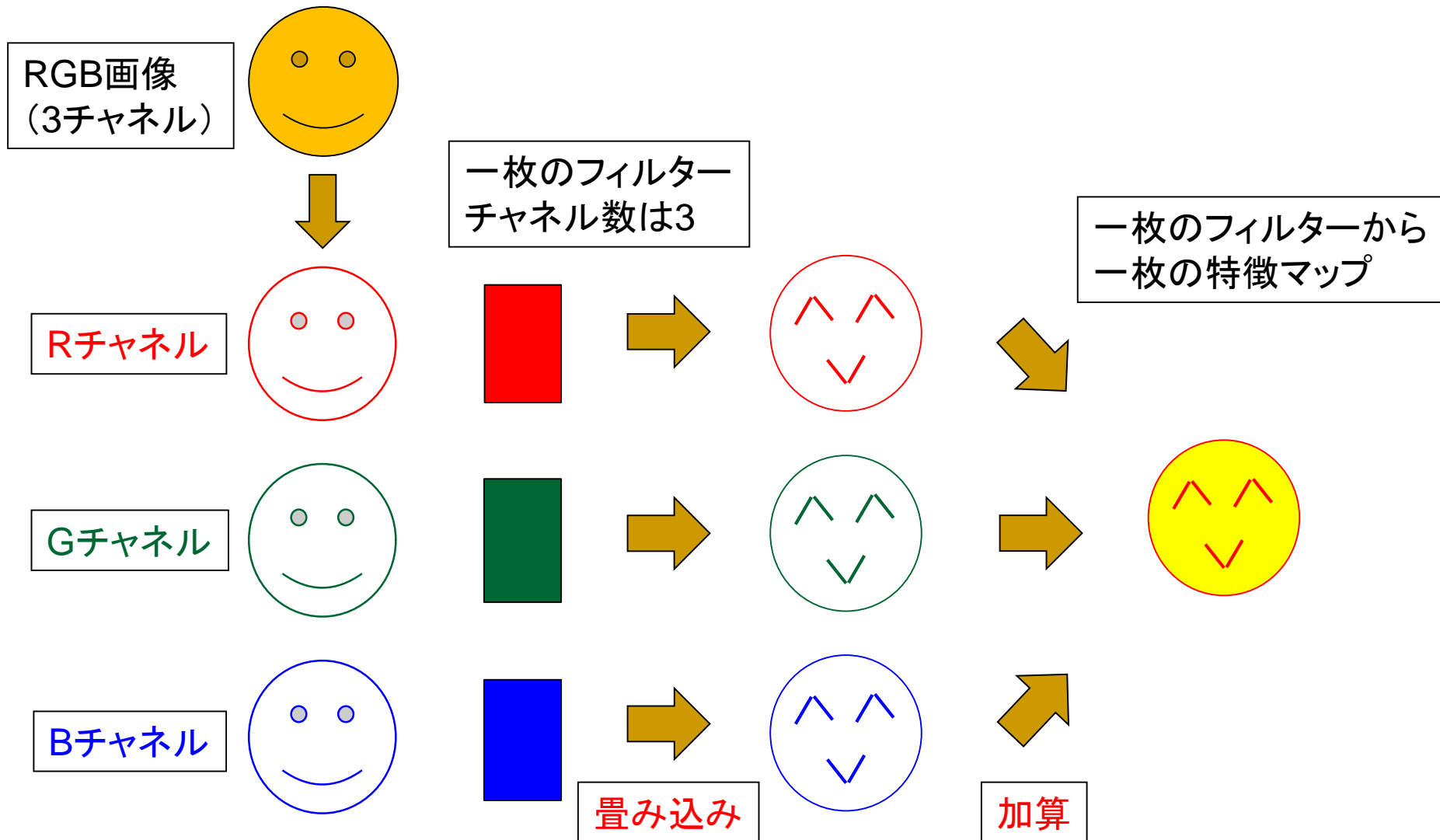
## 表記方法

(バッチサイズ, チャンネル数, Y, X)

(1, M, Y, X)

\*入力画像の大きさと同じになるようにパディング, スライドを調整したという仮定

# 入力値のチャンネル数が複数の場合



# 特徴マップ③

## 表記方法

(バッチサイズ, チャンネル数, Y, X)

### ■ 入力画像 (1枚=バッチサイズ=1)

- RGB画像 (3チャンネル)
- 大きさ  $X \times Y$
- 次元数  $X \times Y \times 3$

(1, 3, Y, X)

チャンネル数は同じ

### ■ フィルター

- 枚数 M枚 (3チャンネル)
- 大きさ  $W \times W$
- 次元数  $W \times W \times 3$

## 表記方法

(枚数, チャンネル数, Y, X)  
もしくは (枚数, Y, X)

(M, 3, W, W)

(M, W, W)

### ■ 畳み処理後の特徴マップ

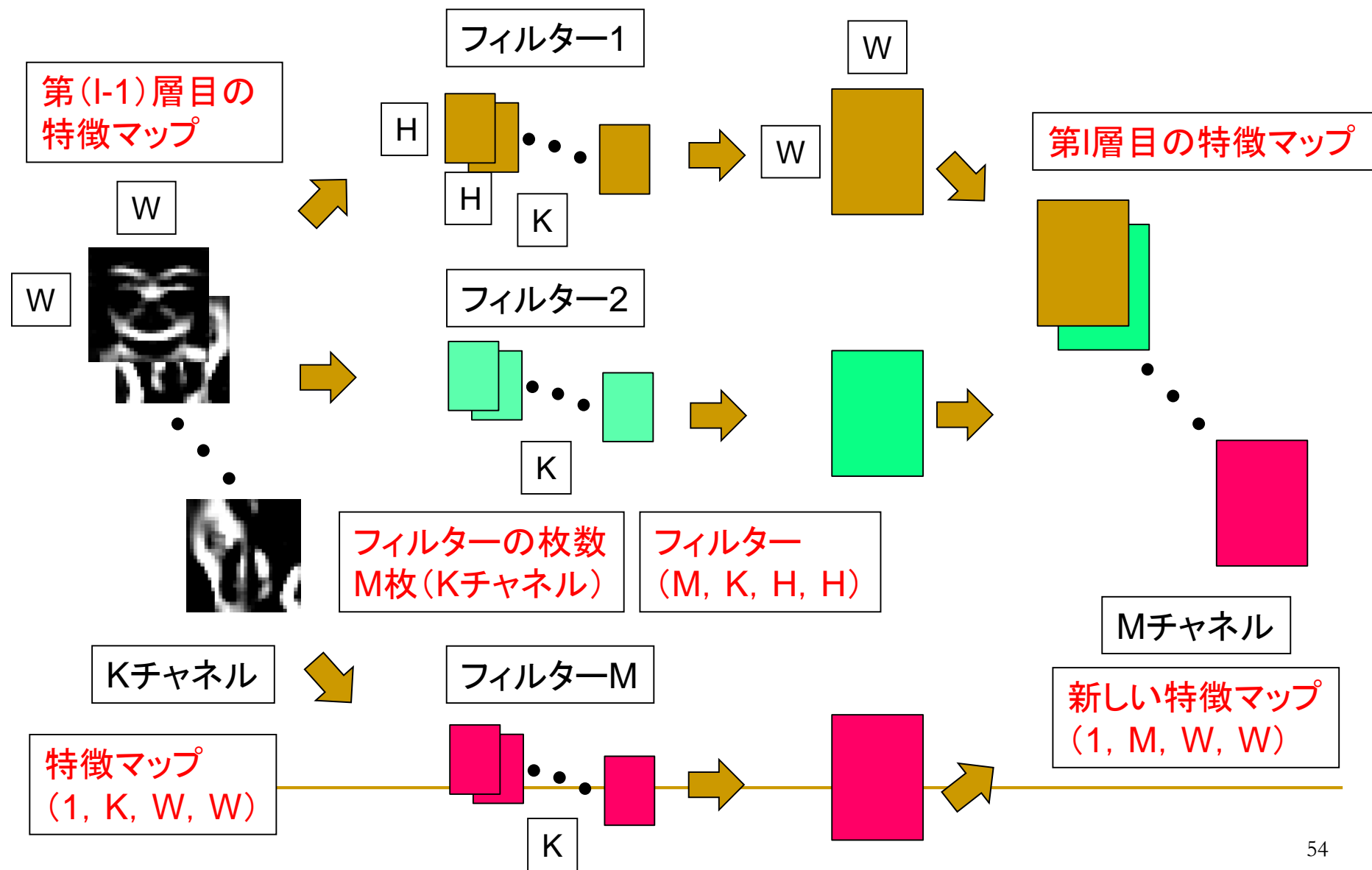
- 枚数 M枚チャンネル
- 大きさ  $X \times Y$
- 次元数  $X \times Y \times M$

## 表記方法

(バッチサイズ, チャンネル数, Y, X)

(1, M, Y, X)

# 特徴マップの畳み込み処理①





# 特徴マップの畳み込み処理②

$$u_{ijm} = \sum_{k=1}^K \sum_{p=0}^{H-1} \sum_{q=0}^{H-1} z_{i+p, j+q, k}^{(l-1)} h_{pqkm}$$



$$u_{ijm} = \sum_{k=1}^K \sum_{p=0}^{H-1} \sum_{q=0}^{H-1} z_{i+p, j+q, k}^{(l-1)} h_{pqkm} + b_{ijm}$$

フィルターごとに閾値(M個)

第1層の特徴マップ

$$z_{ijm}^{(l)} = f(u_{ijm})$$

活性化関数



# LeNet(畳み込みニューラルネットワーク) (Y.LeCun,1989)

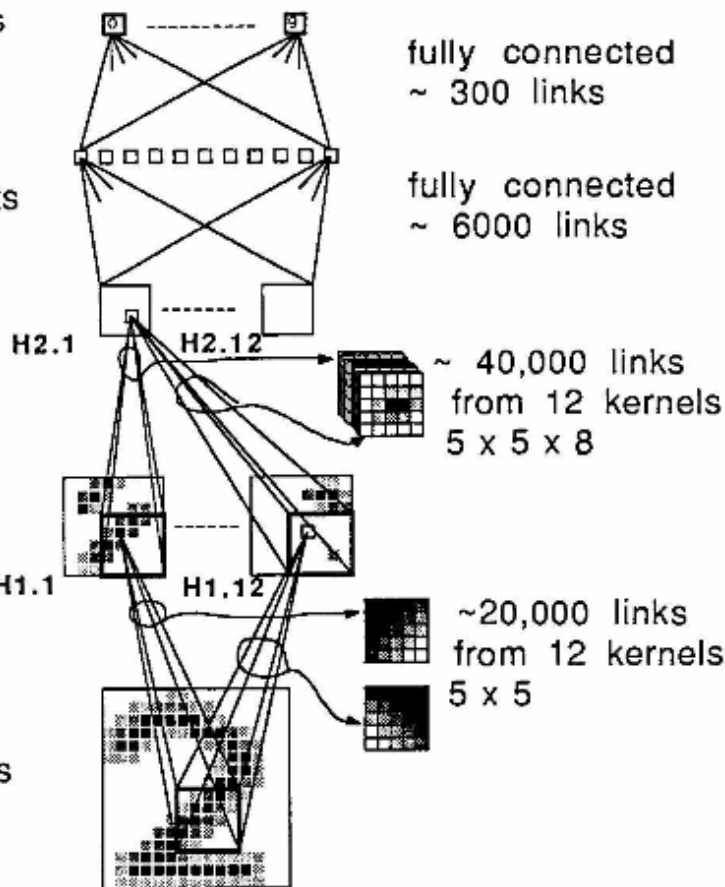
10 output units

layer H3  
30 hidden units

layer H2  
 $12 \times 16 = 192$   
hidden units

layer H1  
 $12 \times 64 = 768$   
hidden units

256 input units



出力層

全結合層

畳み込み層

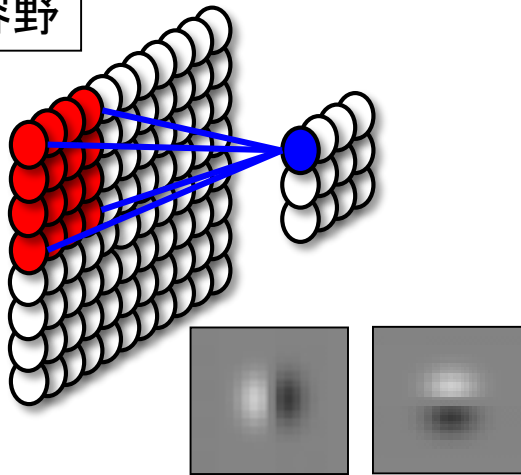
畳み込み層

入力層

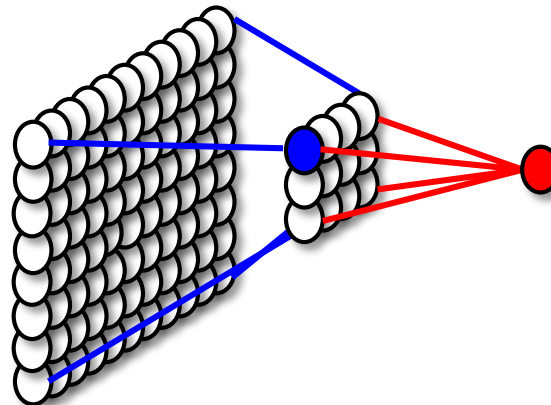
# 単純型細胞，複雑型細胞

単純型細胞

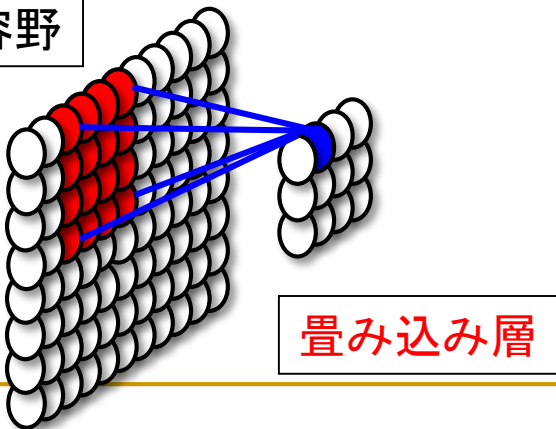
受容野



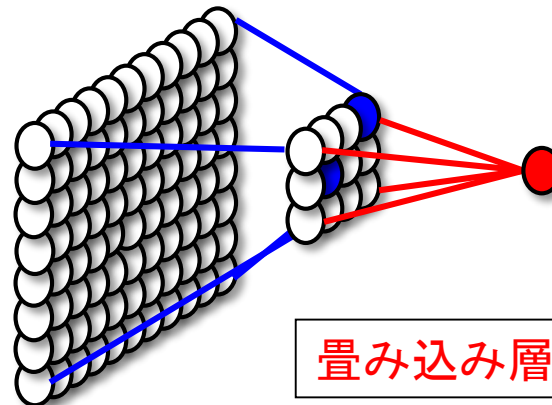
複雑型細胞



受容野



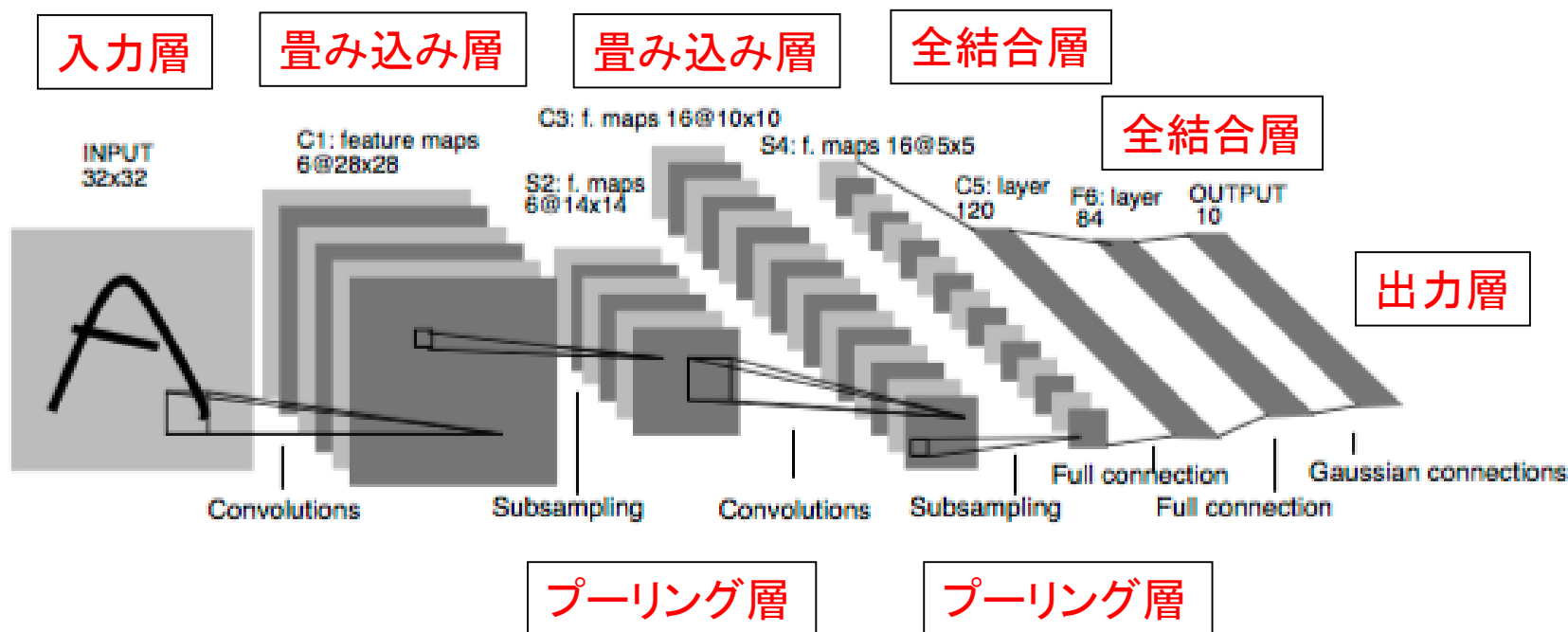
畳み込み層



畳み込み層+プーリング層

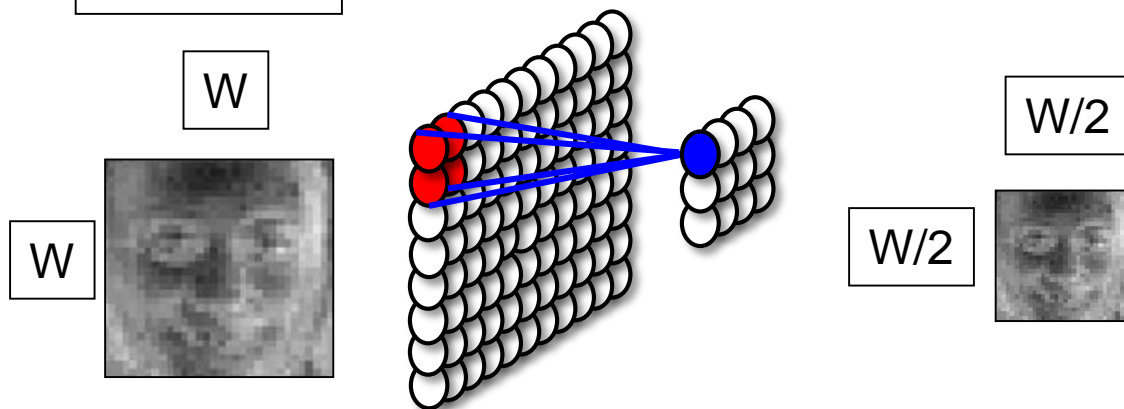
# LeNet-5 (畳み込みニューラルネットワーク) (Y. LeCun, 1998)

- 畳み込み層, プーリング層から構成されるニューラルネットワーク
  - 畳み込み層・・・単純型細胞
  - 畳み込み層, プーリング層・・・複雑型細胞



# プーリング層 (Pooling Layer)

特徴マップ



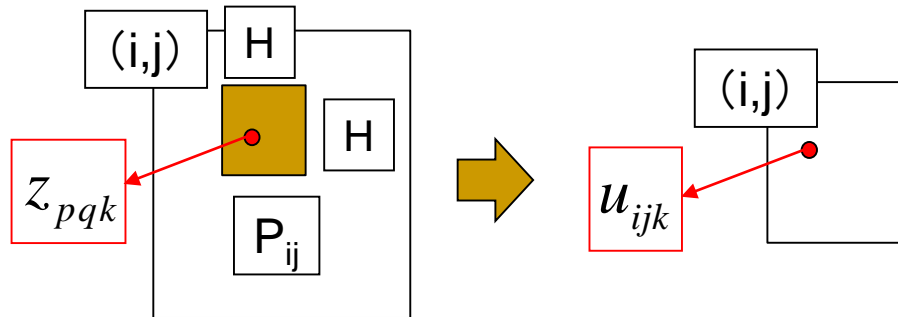
最大プーリング (大きさは $2 \times 2$ , スライドは2)

22	27	26	68	64	39	38	72
16	84	29	4	10	47	25	3
37	66	7	89	49	72	81	24
83	67	61	70	95	88	43	48
54	70	0	49	54	34	29	92
10	97	25	1	67	43	10	67
61	66	59	16	54	85	58	17
29	32	87	63	37	15	8	44



84	68	64	72
83	89	95	81
97	49	67	92
66	87	85	58

# プーリング処理



最大プーリング

$$u_{ijk} = \max_{(p,q) \in P_{ij}} z_{pqk}$$

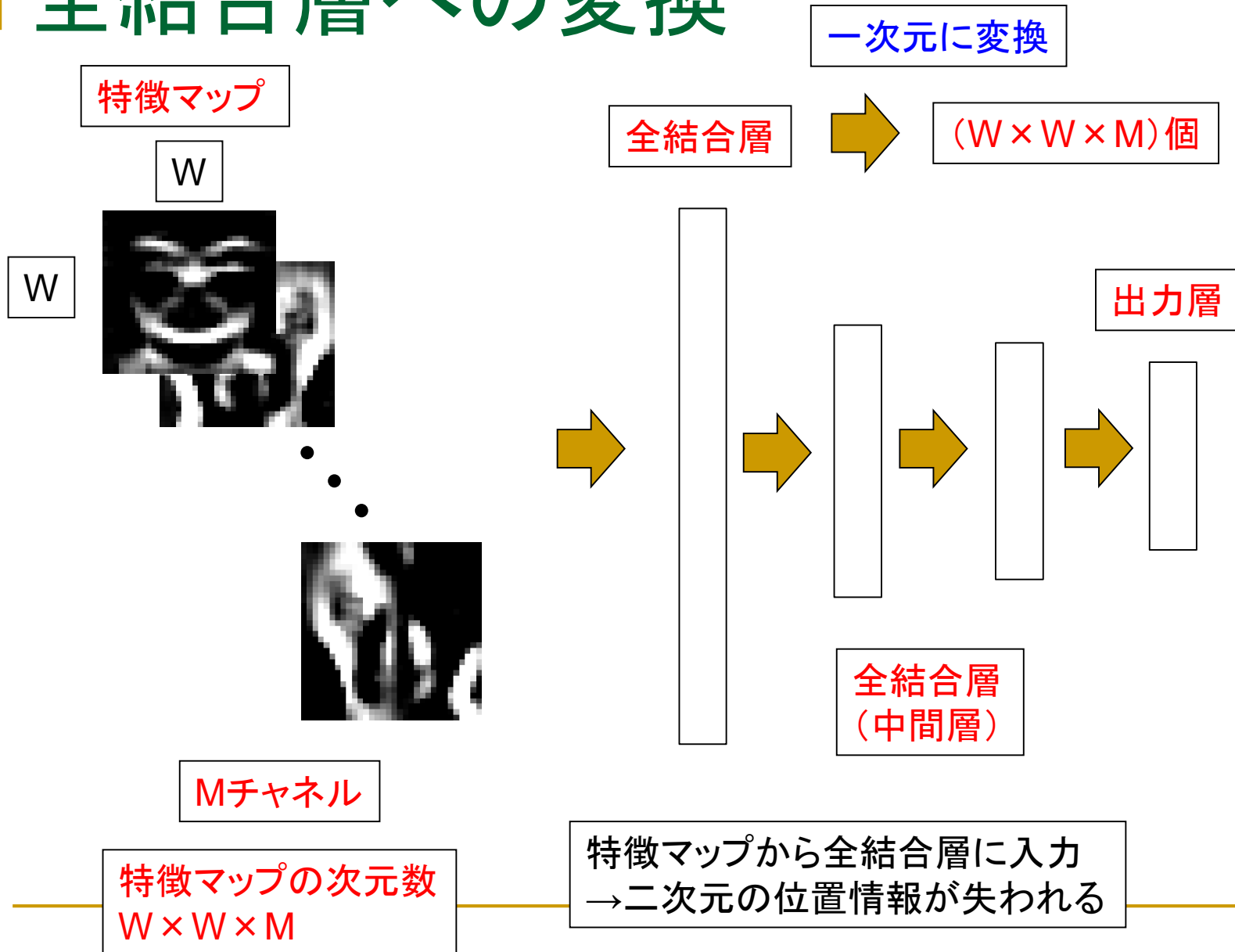
平均プーリング

$$u_{ijk} = \frac{1}{H^2} \sum_{(p,q) \in P_{ij}} z_{pqk}$$

Lpプーリング

$$u_{ijk} = \left( \frac{1}{H^2} \sum_{(p,q) \in P_{ij}} z_{pqk}^P \right)^{\frac{1}{P}}$$

# 全結合層への変換



# 出力層

回帰の場合

全結合層

出力層

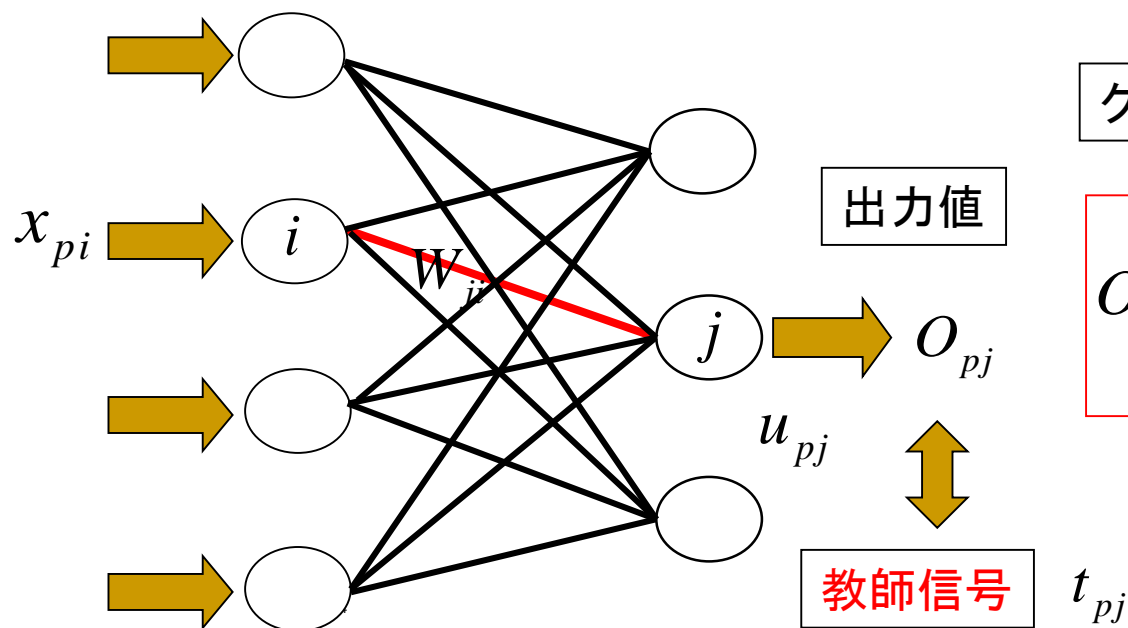
$$O_{pj} = u_{pj}$$

恒等関数

クラス分類の場合

$$O_{pj} = \frac{e^{u_{pj}}}{\sum_{k=1}^K e^{u_{pk}}}$$

ソフトマックス関数



出力値

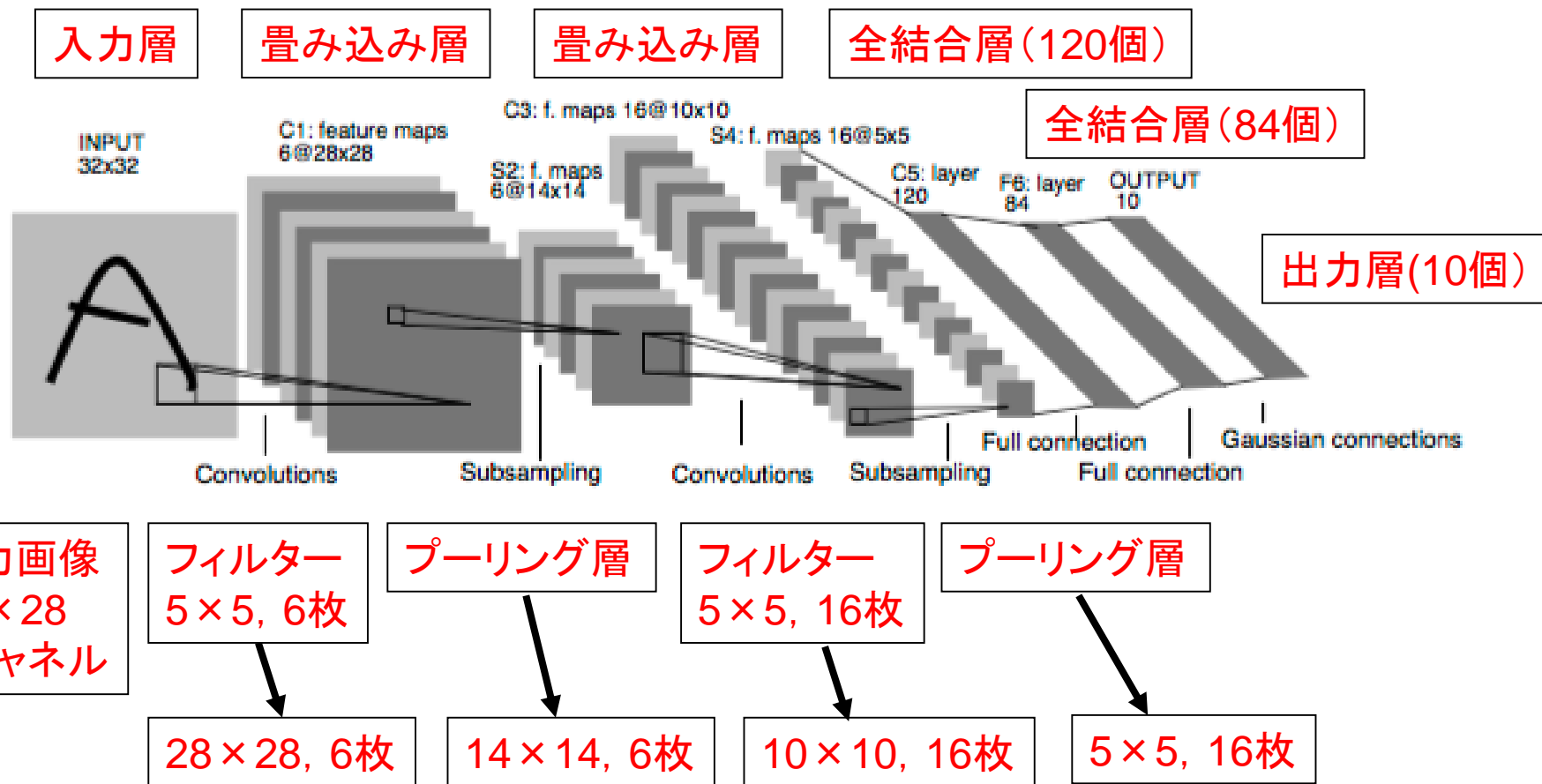
教師信号

$$u_{pj} = \sum_{i=1}^H x_{pi} W_{ji} + b_j$$

$$O_{pj} = f(u_{pj})$$

# 畳み込みニューラルネットワーク(LeNet5)

数字認識





# VGG (Visual Geometry Group) ①

## 表記方法

(バッチサイズB, チャンネル数, Y, X)

(B,3,224,224)

入力層

maxプーリング  
2×2, スライド2

畳み込み処理後の活性化関数  
ReLU

畳み込み層

畳み込み層

プーリング層

畳み込み層

畳み込み層

プーリング層

畳み込み層

畳み込み層

畳み込み層

畳み込み層

プーリング層

畳み込み層

畳み込み層

畳み込み層

畳み込み層

プーリング層

(64,3,3)  
stride:1

(128,3,3)  
stride:1

(256,3,3)  
stride:1

(512,3,3)  
stride:1

(B,64,224,224)

(B,128,112,112)

(B,256,56,56)

(B,512,28,28)

(B,64,112,112)

(B,128,56,56)

(B,256,28,28)

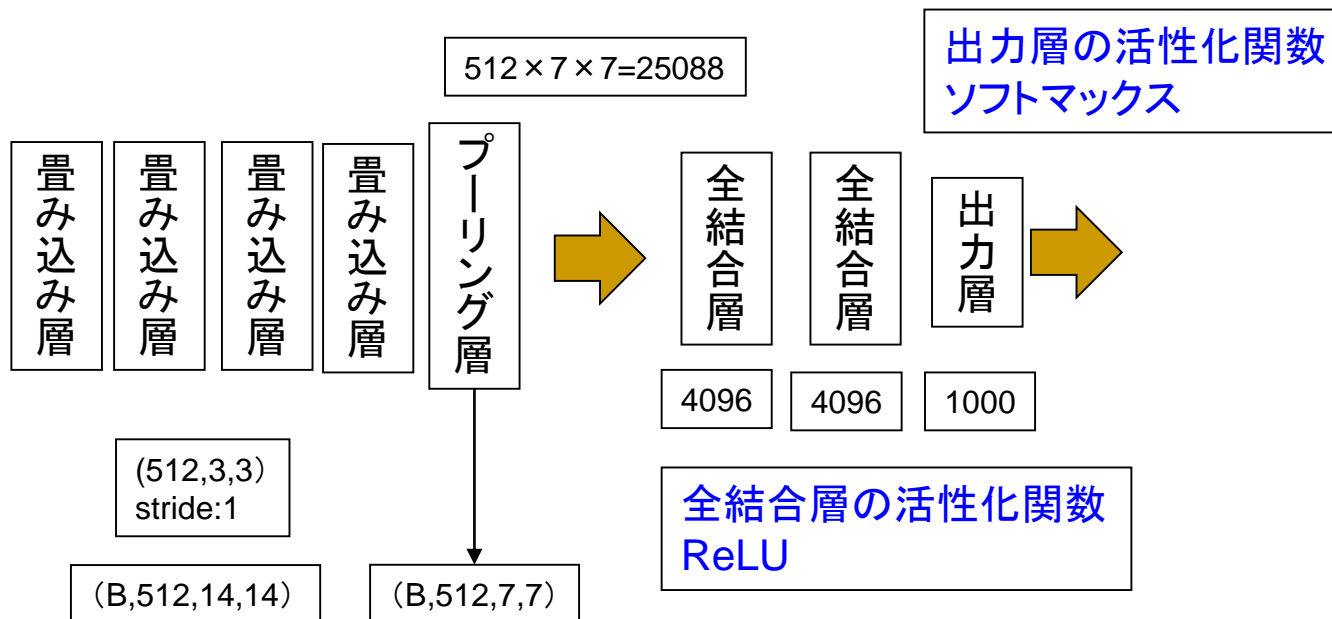
(B,512,14,14)

次頁

# VGG (Visual Geometry Group) ②

## 表記方法

(バッチサイズB, チャンネル数, Y, X)



# 畳み込みニューラルネットワーク の行列計算

# 行列計算による畳み込み処理

入力画像


1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

フィルター

1	2
3	4

畳み込み処理後の画像

44	54	64
84	94	104
124	134	144


$$1 \times 1 + 2 \times 2 + 5 \times 3 + 6 \times 4$$

入力画像

$X$

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16



変換①

$X'$

1	2	3	5	6	7	9	10	11
2	3	4	6	7	8	10	11	12
5	6	7	9	10	11	13	14	15
6	7	8	10	11	12	14	15	16

$$U' = FX'$$

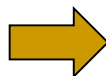
44	54	64	84	94	104	124	134	144
----	----	----	----	----	-----	-----	-----	-----



変換③

$$U \leftarrow U'$$

44	54	64
84	94	104
124	134	144



畳み処理後の  
特徴マップ

$$O = f(U)$$

フィルター

$W^{(l)}$

1	2
3	4



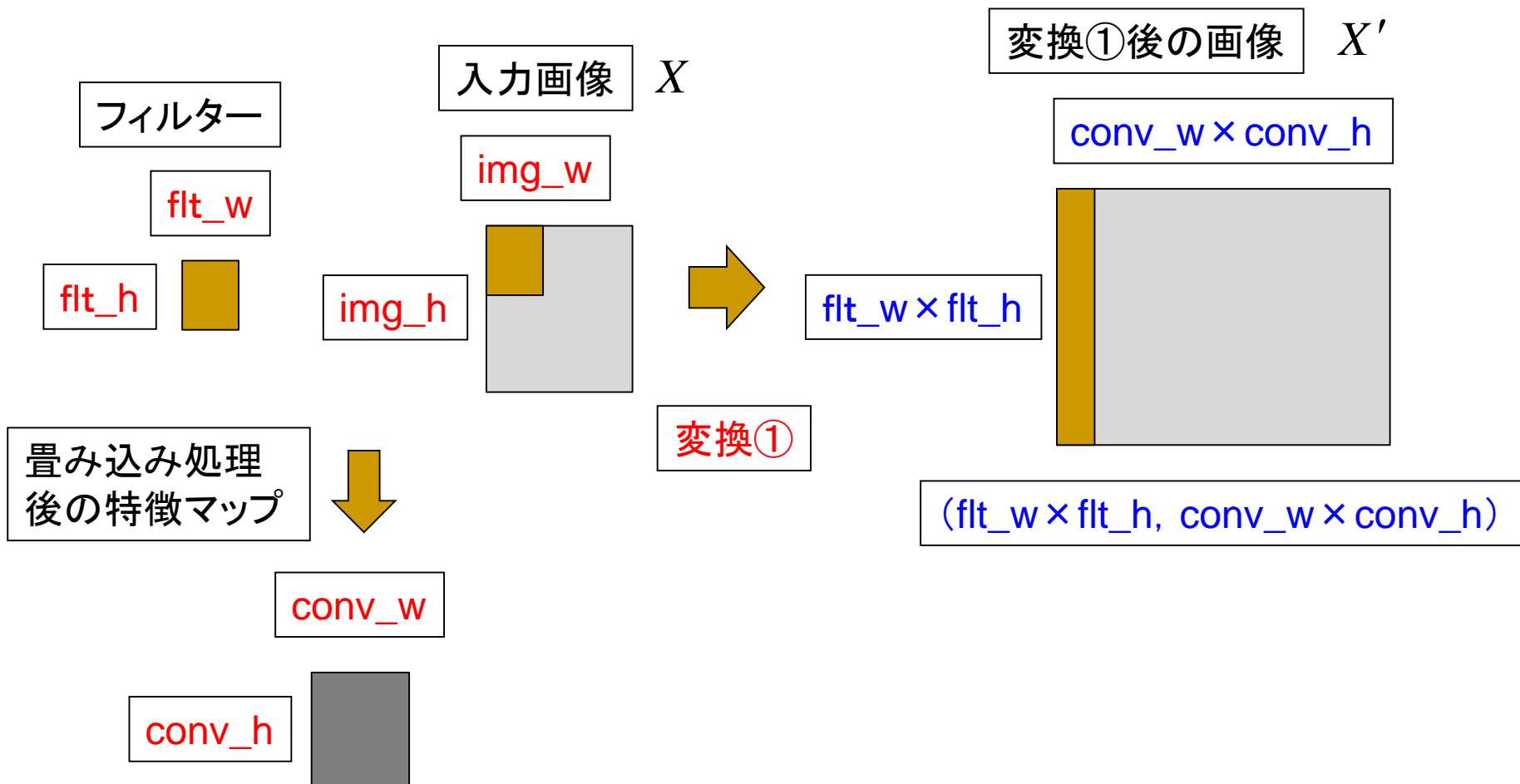
変換②

$F$

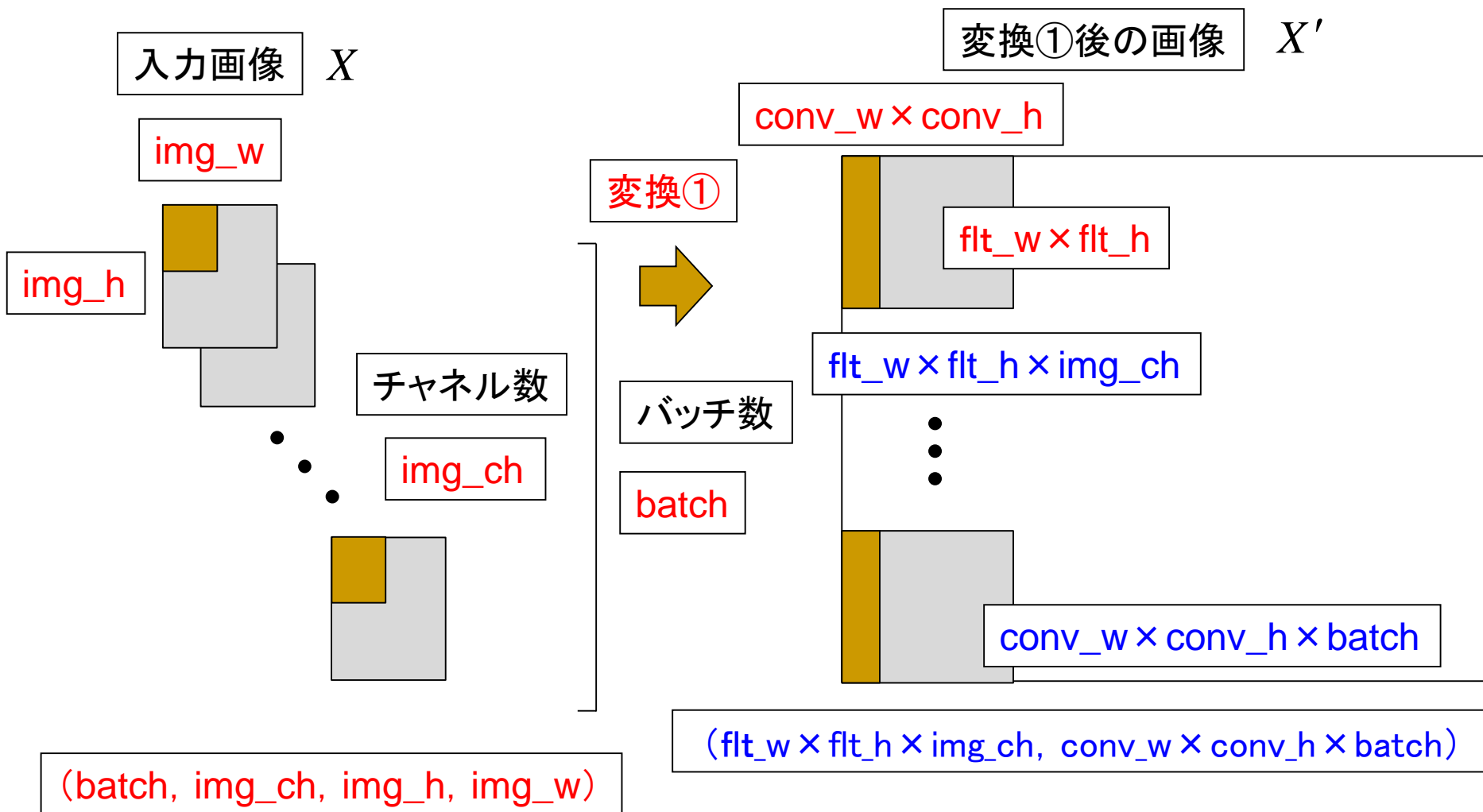
1	2	3	4
---	---	---	---

# 入力画像の変換(変換①)

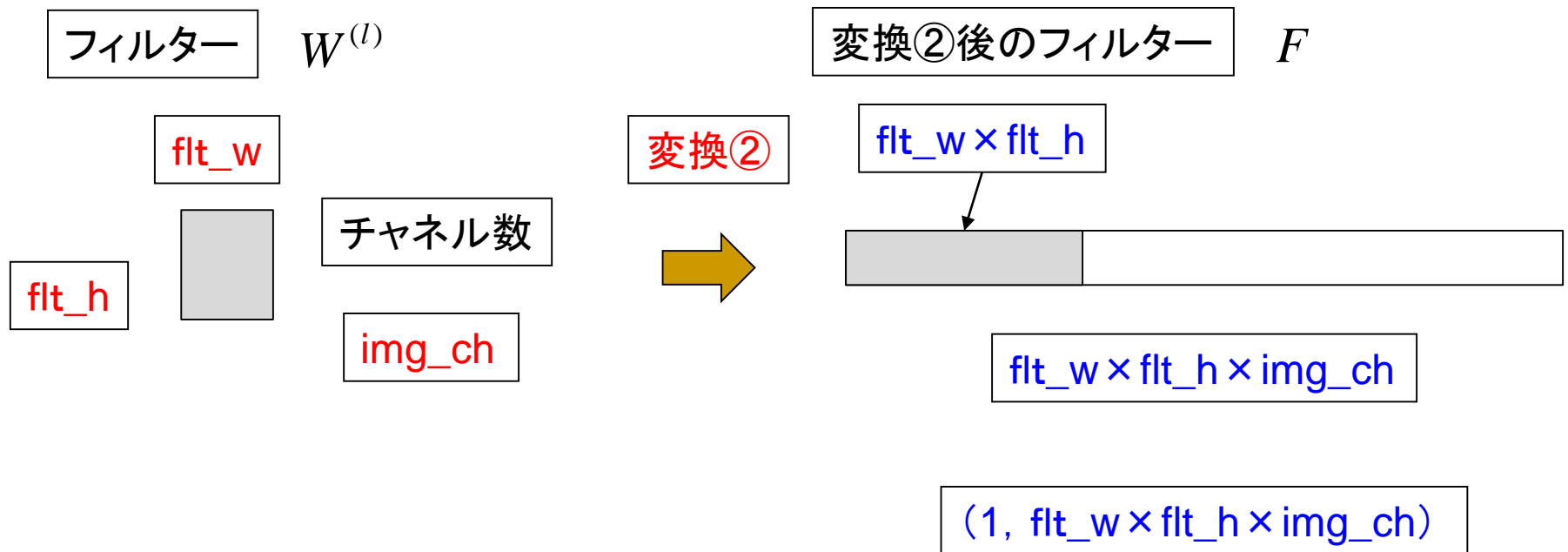
(バッチサイズ1, チャンネル数1, フィルター数1の場合)



# 入力画像の変換(変換①)

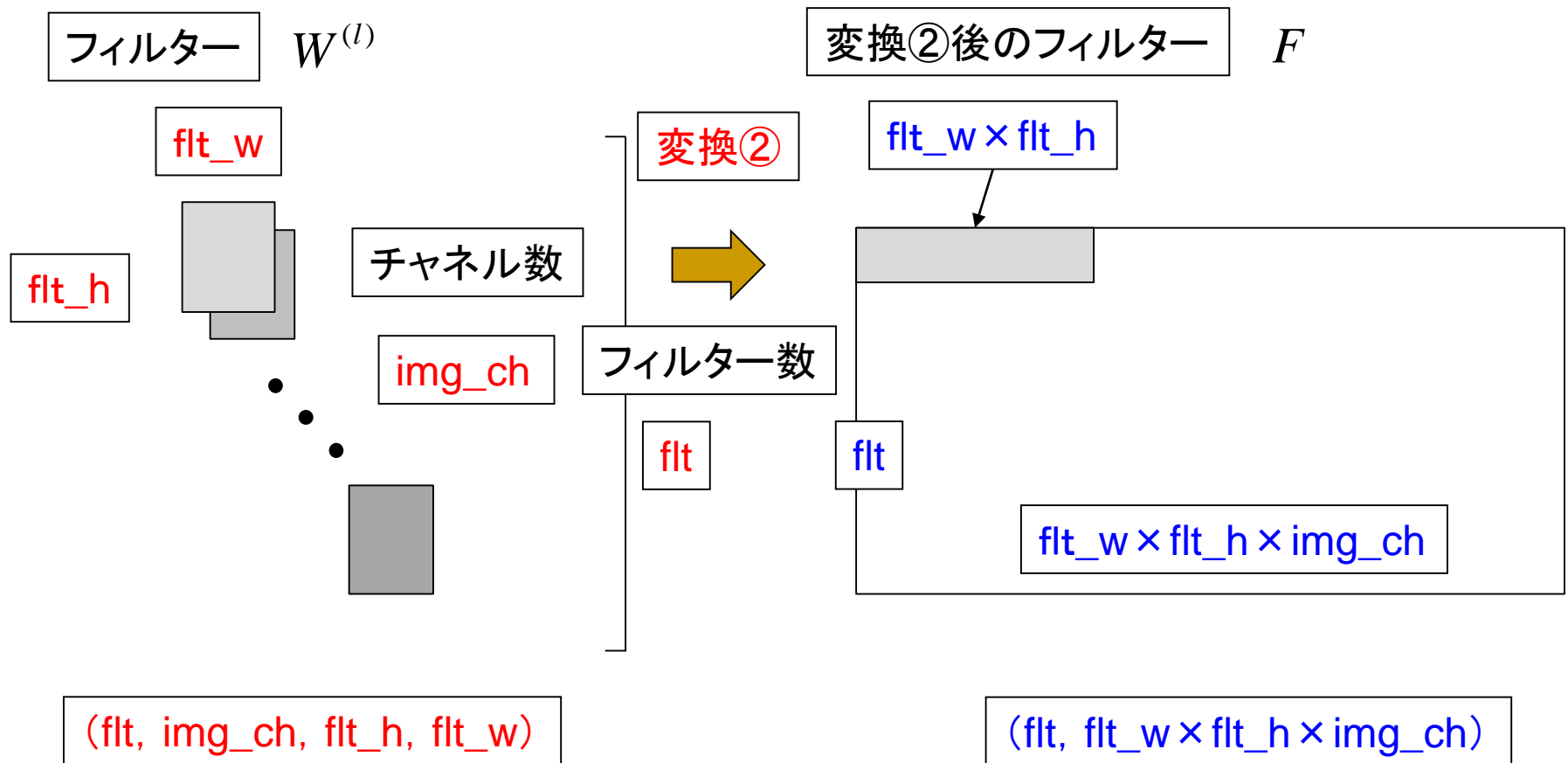


# フィルターの変換(変換②) (フィルター数1の場合)



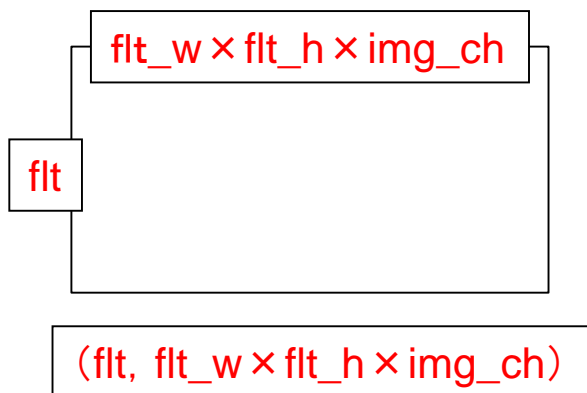


# フィルターの変換(変換②)

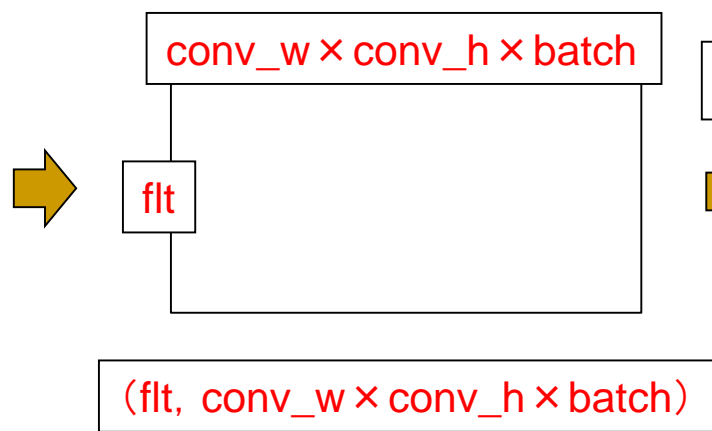


# 行列計算による畳み込み処理

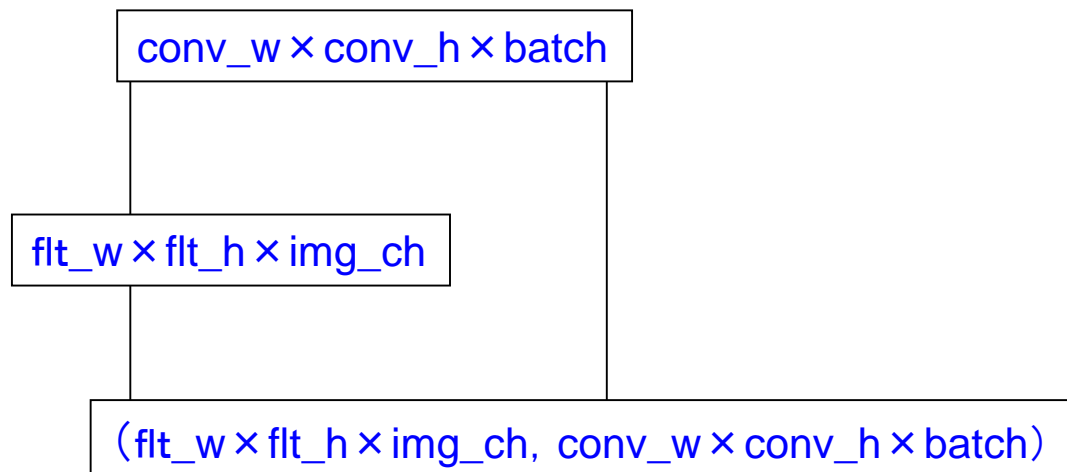
変換②後のフィルター  $F$



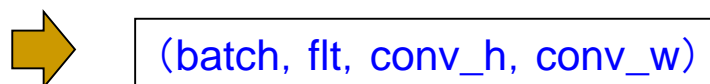
$$U' = WX'$$



変換①後の入力画像  $X'$



変換③  $U$

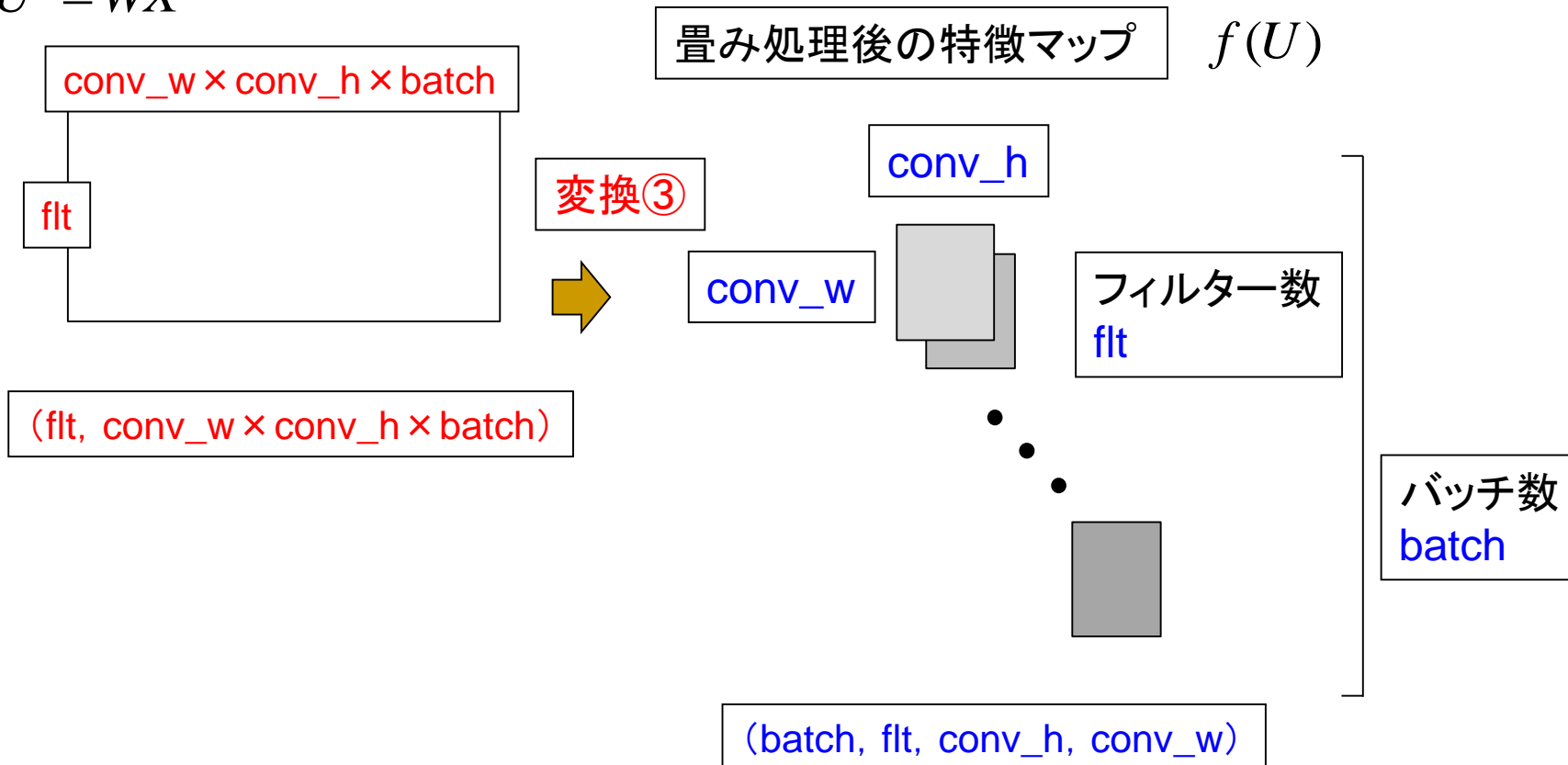


$$O = f(U)$$

畳み込み後の  
特徴マップ

# 特徴マップへの変換(変換③)

$$U' = WX'$$



# 行列計算によるプーリング処理①

入力画像

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16



プーリング処理後  
の画像

6	8
14	16

Maxプーリング  
2×2, スライド2

# 行列計算によるプーリング処理②

入力画像  $X$

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

変換④後の画像

$X'$

変換④

1	2	5	6
3	4	7	18
9	10	13	14
11	12	15	16

行ごとに最大値を求める

変換⑤

$U$

プーリング処理  
後の画像

$P$

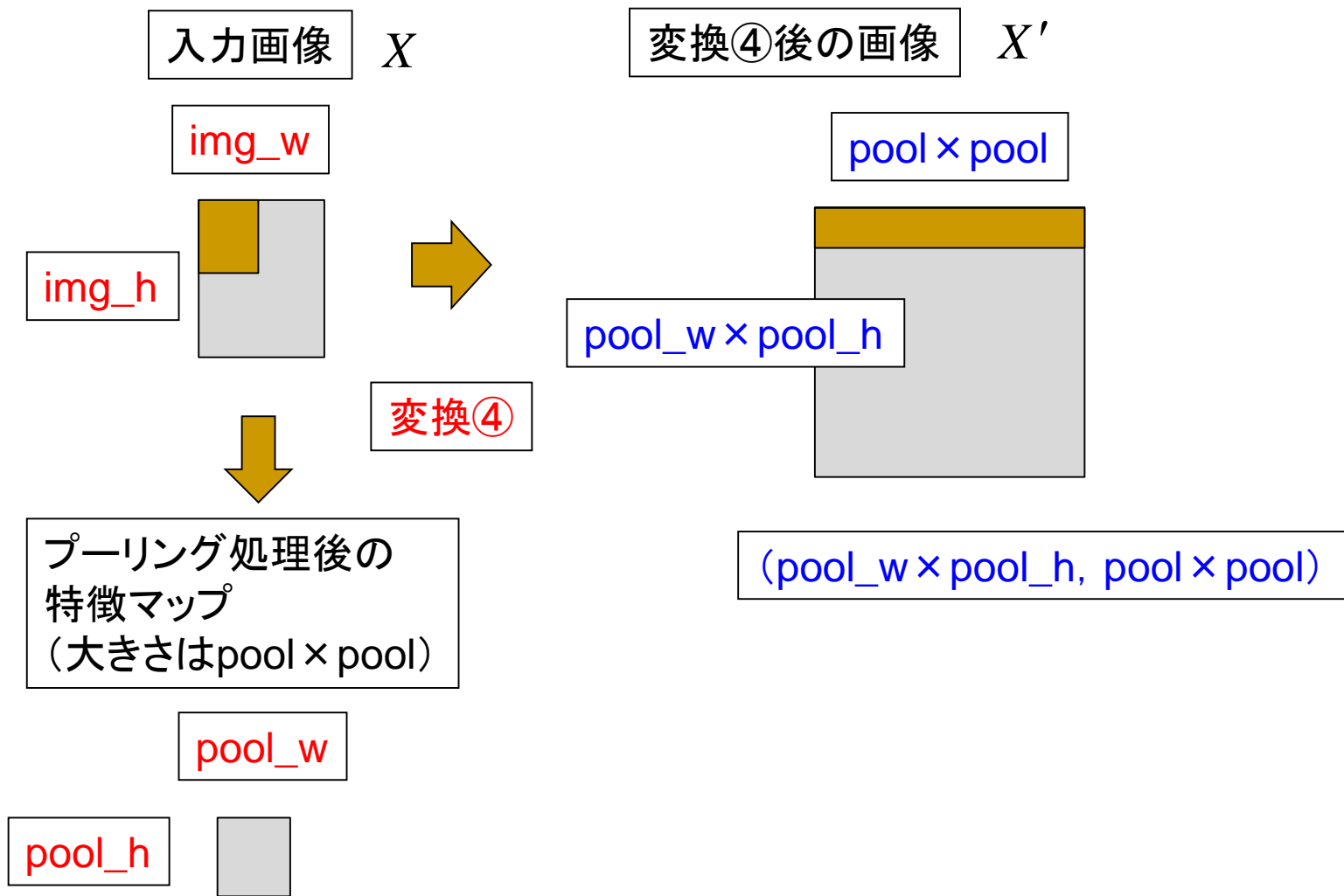
6
8
14
16



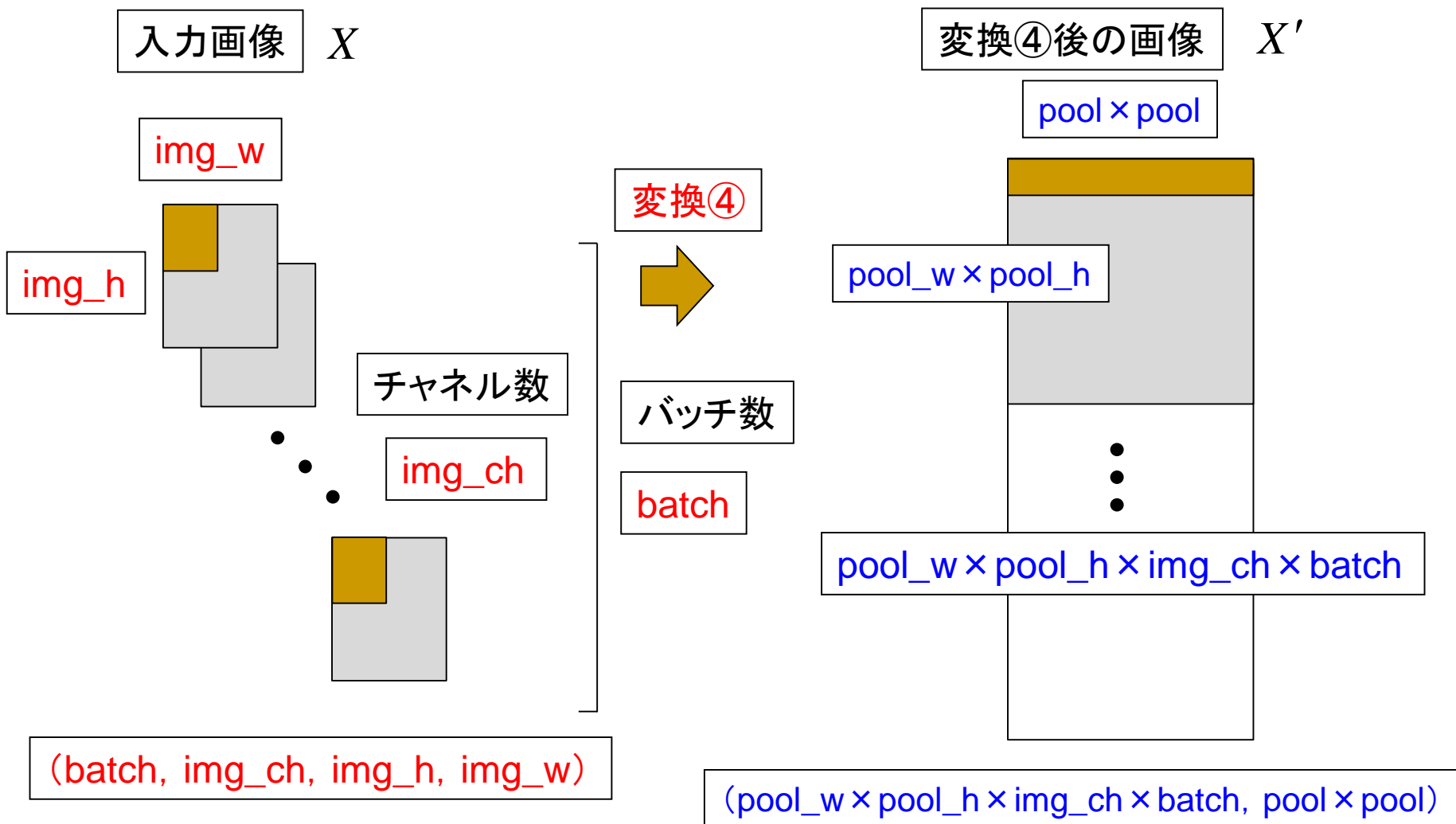
6	8
14	16

プーリング処理  
後の特徴マップ

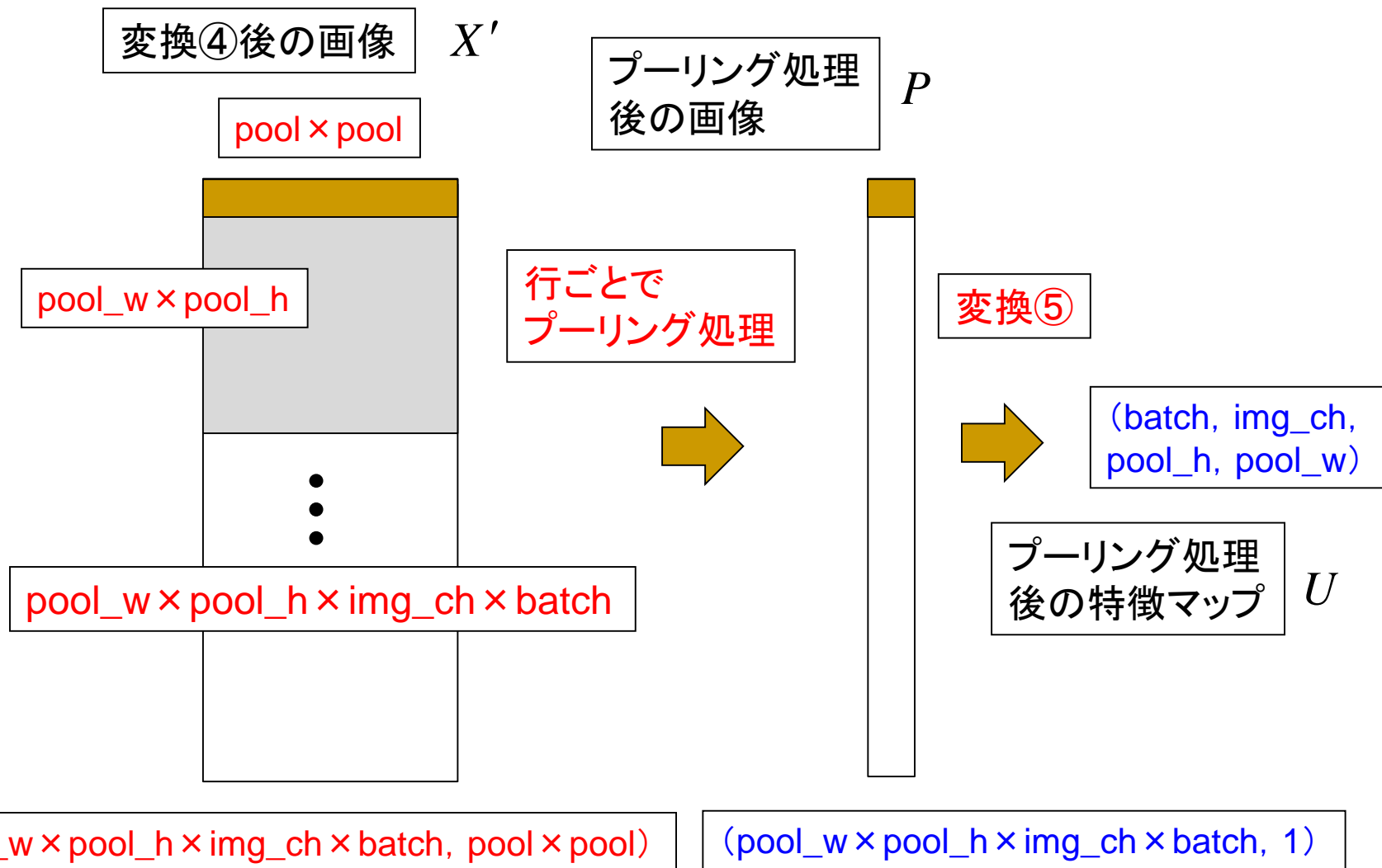
## 変換④(バッチサイズ1, チャンネル数1の場合)



# 入力画像の変換(変換④)



# 行列計算によるプーリング処理





# 特徴マップへの変換(変換⑤)

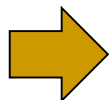
プーリング処理  
後の画像

$P$

プーリング処理  
後の特徴マップ

$U$

変換⑤



pool\_w

pool\_h

チャンネル数  
img\_ch

バッチ数  
batch

(batch, img\_ch, pool\_h, pool\_w)

(pool\_w × pool\_h × img\_ch × batch, 1)

# 畳み込みニューラルネットワーク の学習

誤差逆伝播

# 畳み込みニューラルネットワークの学習

## ① 出力層の学習

出力層と全結合間の学習

## ② 全結合層の学習

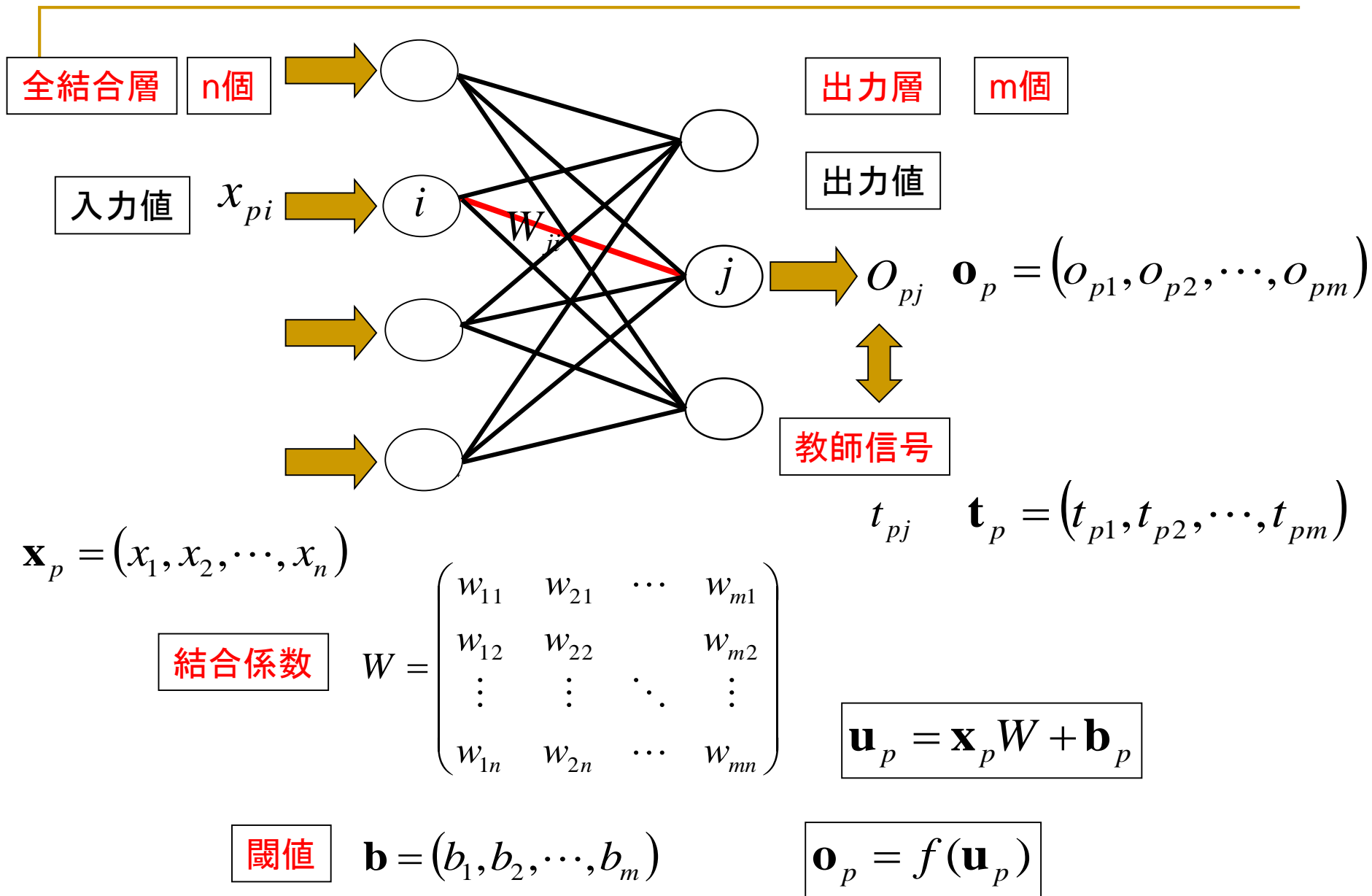
全結合層間の学習

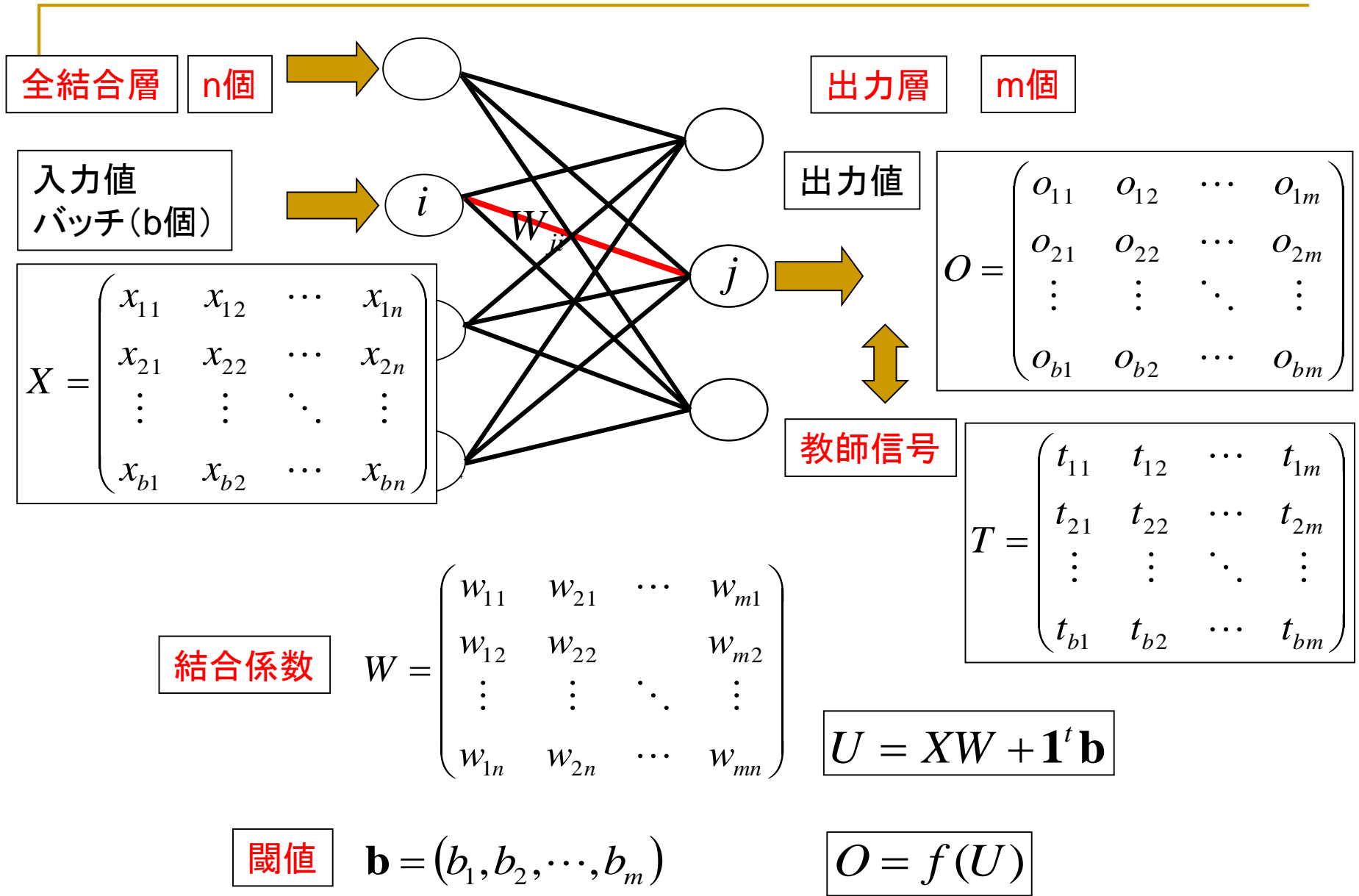
## ③ 畳み込み層の学習

全結合層から逆伝播してきた誤差の変換  
フィルタの学習

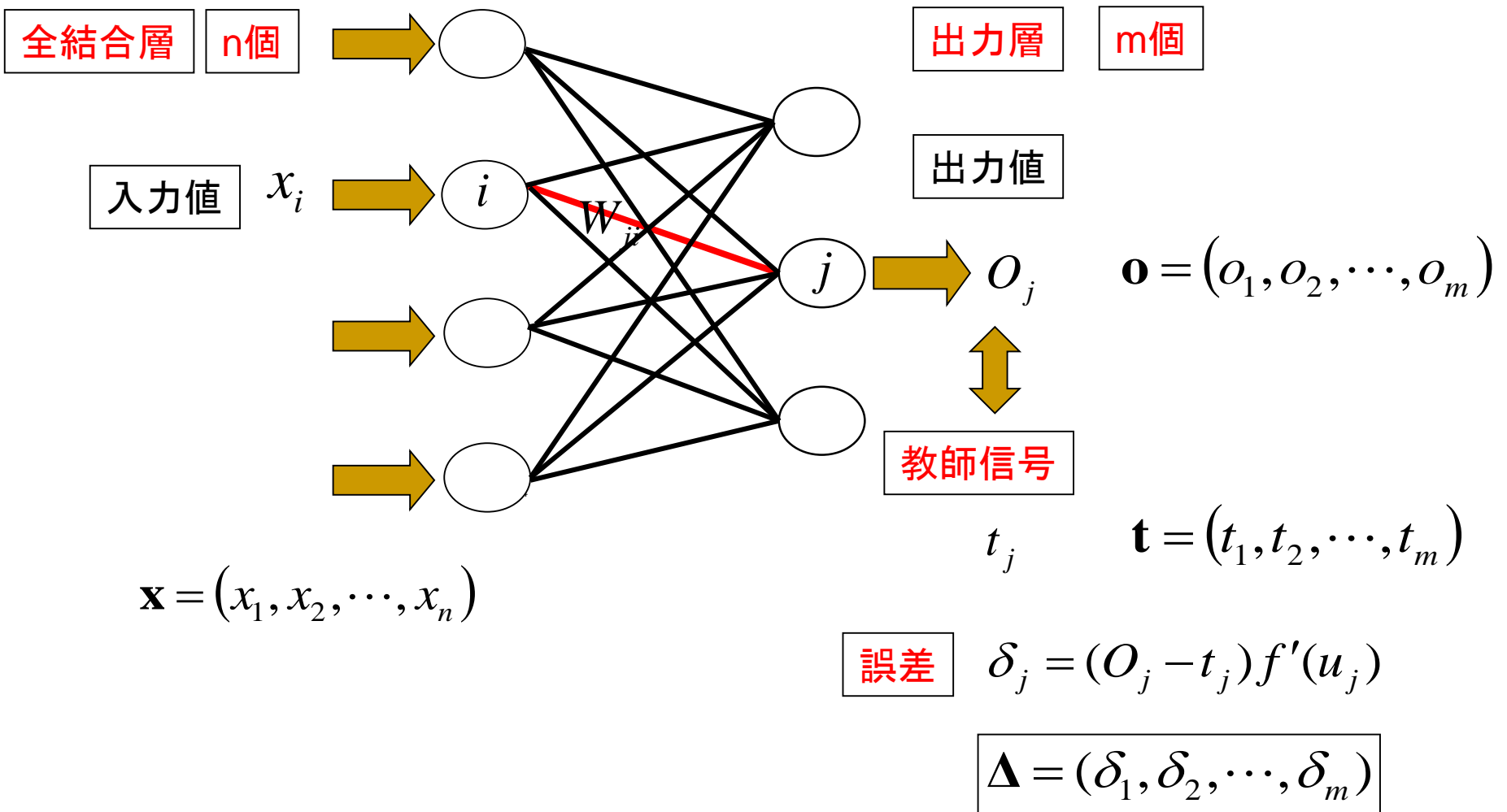
## ④ プーリング層での逆伝播

プーリング層は学習しない, 誤差を逆伝播するのみ





# 出力層の学習(行列計算)①



# 出力層の学習(行列計算)②

$$\begin{aligned}\partial W &= \begin{pmatrix} \partial w_{11} & \partial w_{21} & \cdots & \partial w_{m1} \\ \partial w_{12} & \partial w_{22} & & \partial w_{m2} \\ \vdots & \vdots & \ddots & \vdots \\ \partial w_{1n} & \partial w_{2n} & \cdots & \partial w_{mn} \end{pmatrix} = \begin{pmatrix} \delta_1 x_1 & \delta_2 x_1 & \cdots & \delta_m x_1 \\ \delta_1 x_2 & \delta_2 x_2 & & \delta_m x_2 \\ \vdots & \vdots & \ddots & \vdots \\ \delta_1 x_n & \delta_2 x_n & \cdots & \delta_m x_n \end{pmatrix} \\ &= \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} (\delta_1 \quad \delta_2 \quad \cdots \quad \delta_m) = \mathbf{x}^t \Delta\end{aligned}$$

結合係数

$$W' = W - \alpha \partial W = W - \alpha \mathbf{x}^t \Delta$$

閾値

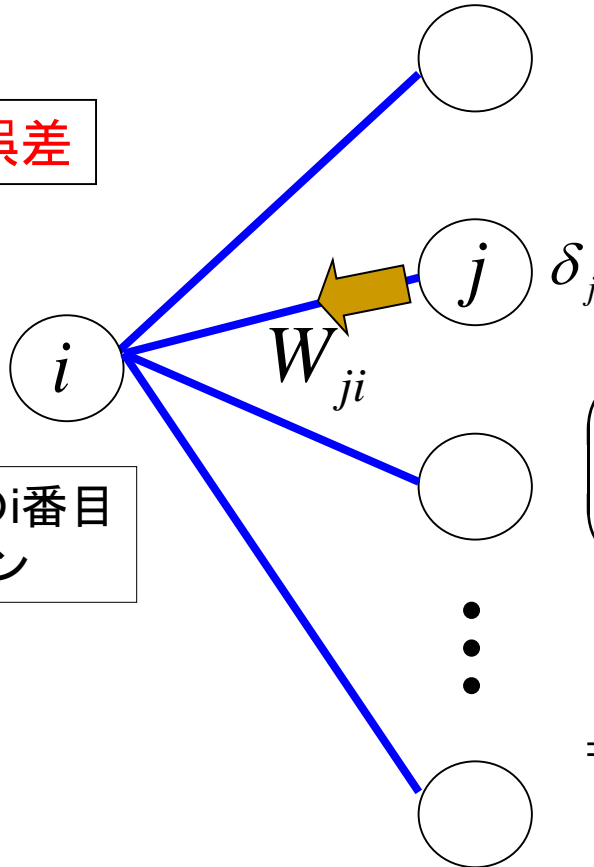
$$\mathbf{b}' = \mathbf{b} - \alpha \partial \mathbf{b} = \mathbf{b} - \alpha \mathbf{1}^t \Delta$$

# 誤差の逆伝播(行列計算)

逆伝播する誤差

$$\sum_{j=1}^m W_{ji} \delta_j$$

全結合層の*i*番目のニューロン



出力層の*j*番目のニューロン

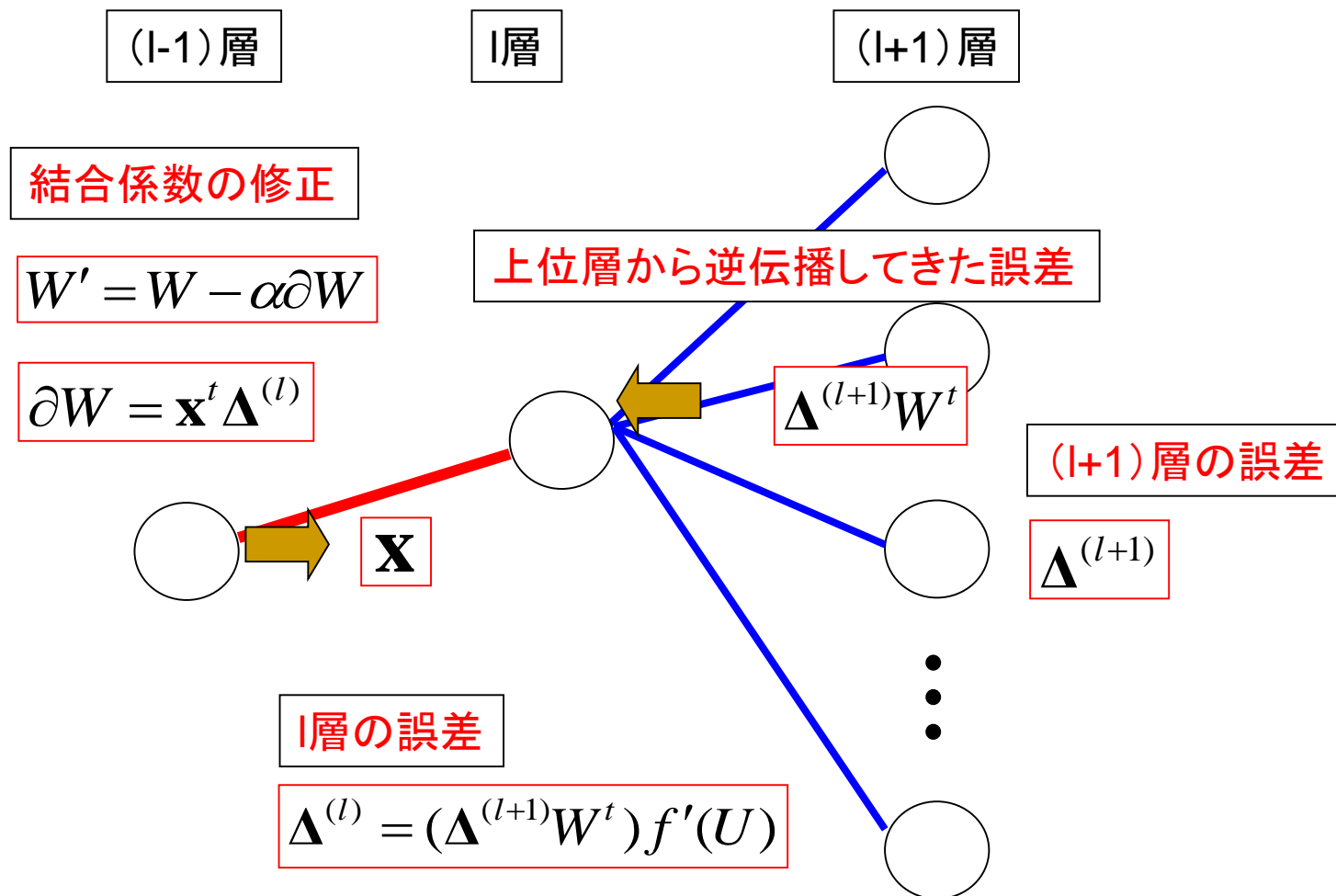
$$\left( \sum_{j=1}^m W_{j1} \delta_j, \sum_{j=1}^m W_{j2} \delta_j, \dots, \sum_{j=1}^m W_{jn} \delta_j \right)$$

$$= (\delta_1, \delta_2, \dots, \delta_m) \begin{pmatrix} w_{11} & w_{21} & \cdots & w_{n1} \\ w_{12} & w_{22} & & w_{n2} \\ \vdots & \vdots & \ddots & \vdots \\ w_{1m} & w_{2m} & \cdots & w_{mn} \end{pmatrix}$$

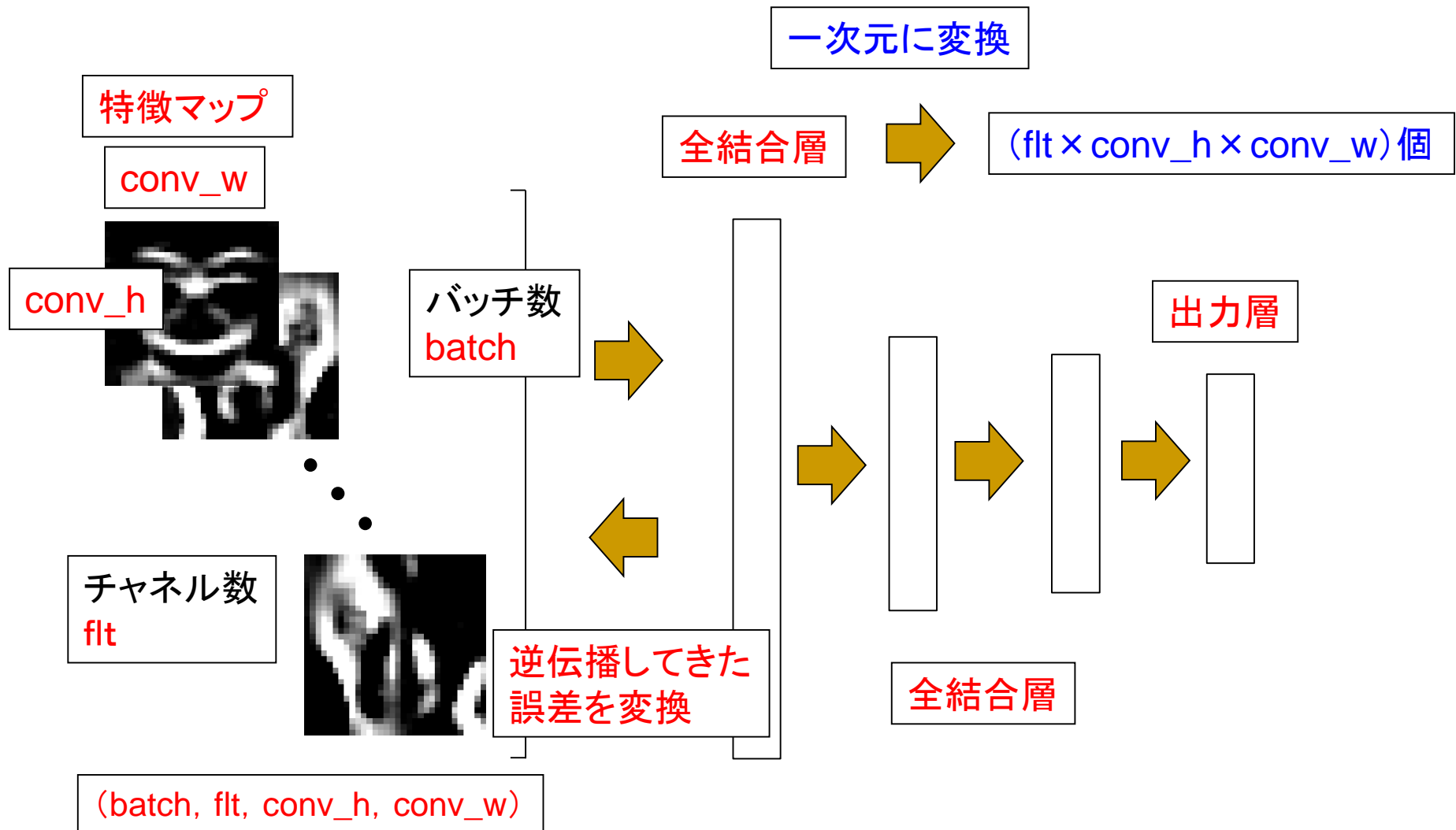
$$= \Delta W^t$$



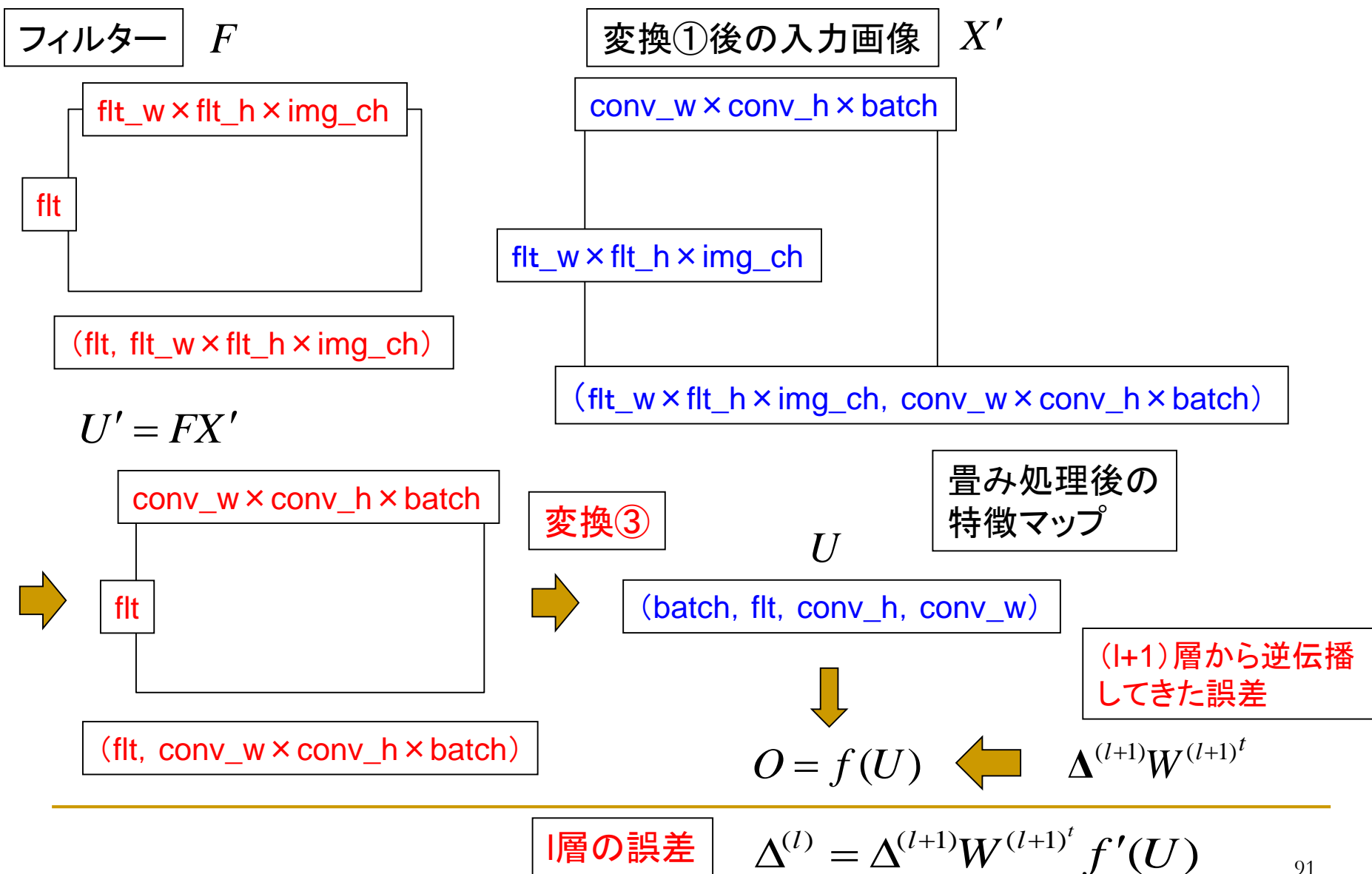
# 全結合層の学習



# 全結合層から逆伝播してきた誤差の変換



# 畳み込み層の学習①



# 畳み込み層の学習②

変換①後の入力画像  $X'$

誤差

$$\Delta^{(l)} = \Delta^{(l+1)} W^{(l+1)'} f'(U)$$

(batch, flt, conv\_h, conv\_w)

変換



flt

conv\_w × conv\_h × batch

(conv\_w × conv\_h × batch, flt)

conv\_w × conv\_h × batch

flt\_w × flt\_h × img\_ch

(flt\_w × flt\_h × img\_ch,  
conv\_w × conv\_h × batch)

フィルターの修正値

転置

$$\partial F = X' \Delta^{(l)}$$

(flt\_w × flt\_h × img\_ch, flt)



(flt, flt\_w × flt\_h × img\_ch)

変換②の逆変換

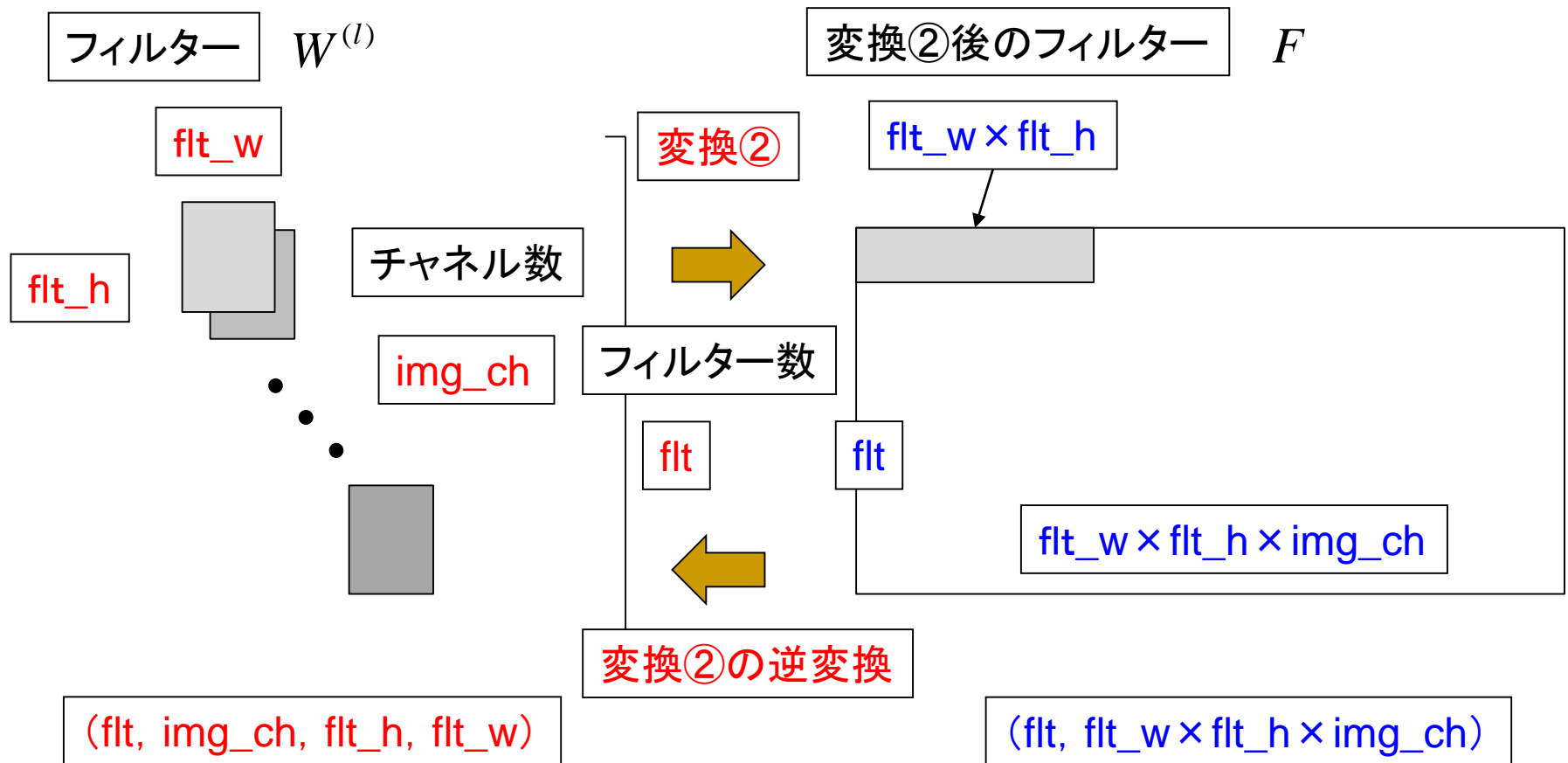


$\partial W^{(l)}$

フィルター  
(4次元)

(flt, img\_ch, flt\_h, flt\_w)

# 畳み込み層の学習③



# 誤差の逆伝播①

誤差  $\Delta^{(l)}$

flt

conv\_w × conv\_h × batch

(conv\_w × conv\_h × batch, flt)

フィルター  $F$

flt\_w × flt\_h × img\_ch

flt

(flt, flt\_w × flt\_h × img\_ch)

逆伝播される誤差

$\Delta^{(l)} F$

conv\_w × conv\_h × batch

flt\_w × flt\_h × img\_ch

conv\_w × conv\_h × batch

(conv\_w × conv\_h × batch,  
flt\_w × flt\_h × img\_ch)

転置

flt\_w × flt\_h × img\_ch

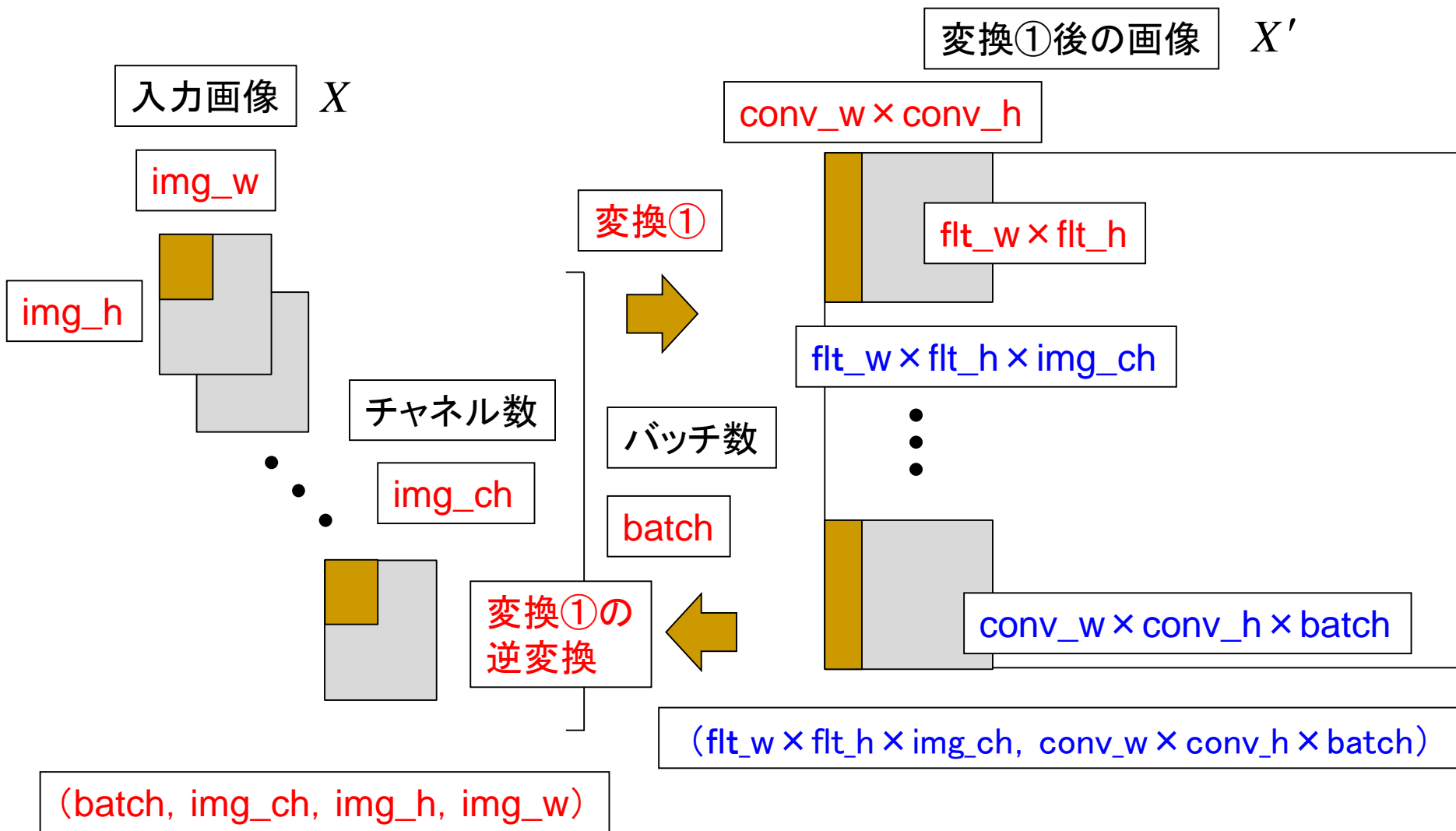
(flt\_w × flt\_h × img\_ch,  
conv\_w × conv\_h × batch)

変換①の  
逆変換

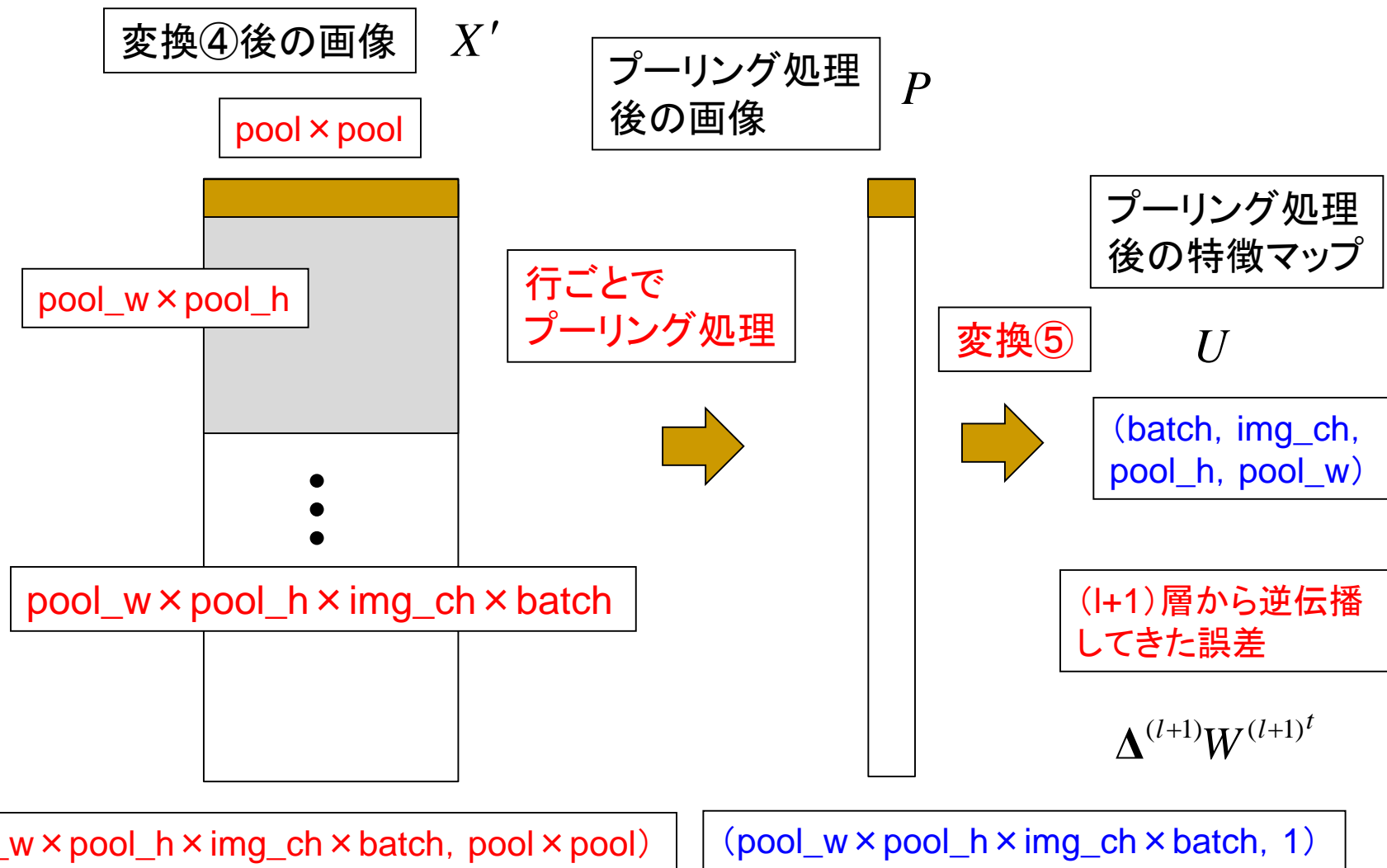
(batch, img\_ch, flt\_h, flt\_w)

$\Delta^{(l)} W^{(l)t}$

# 誤差の逆伝播②



# プーリング層での逆伝播①





# プーリング層での逆伝播②

変換④後の画像  $X'$

pool × pool

pool\_w × pool\_h

⋮

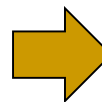
pool\_w × pool\_h × img\_ch × batch

最大値の位置を記憶

プーリング処理  
後の画像

$P$

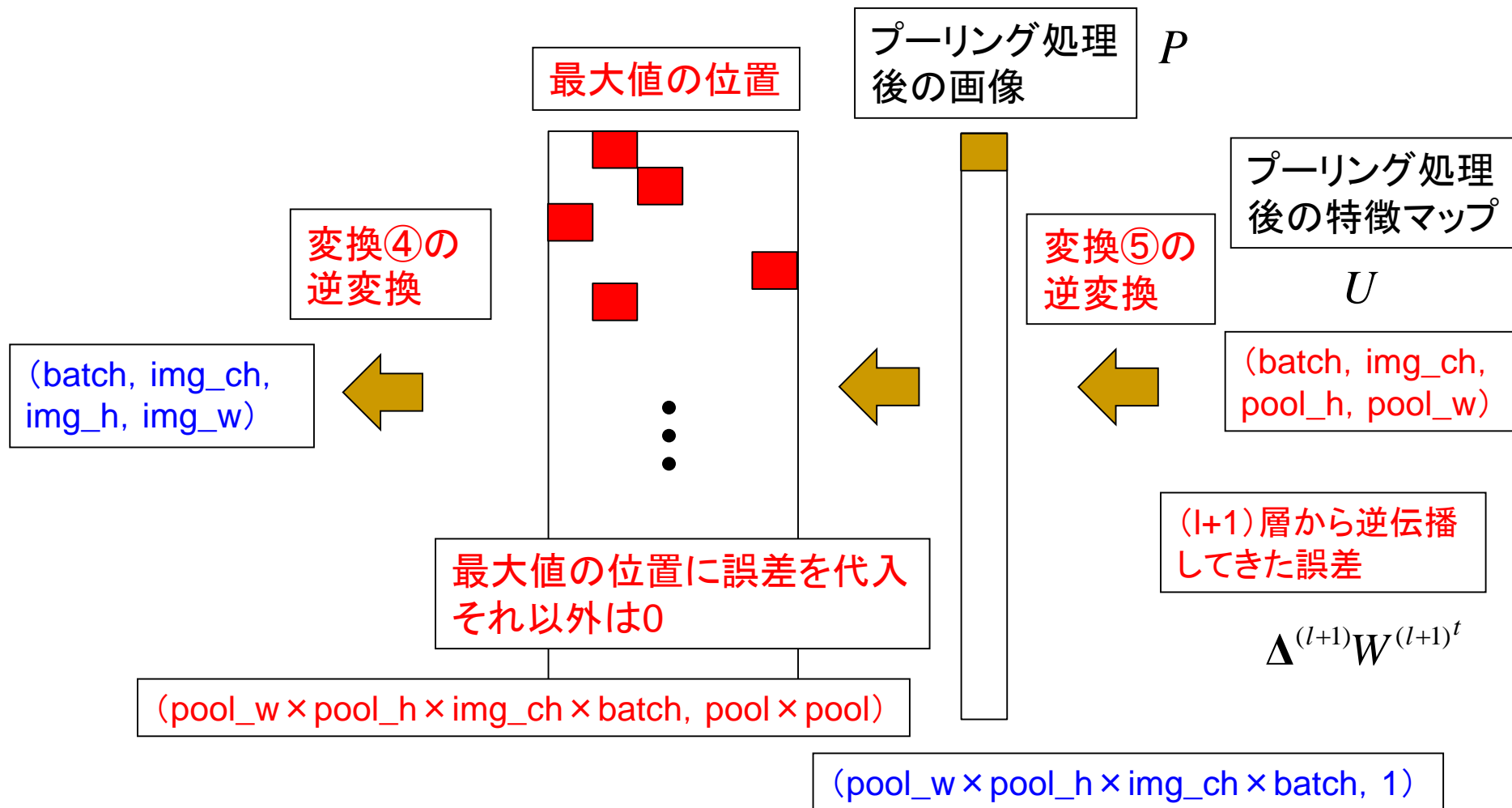
行ごとに  
プーリング処理



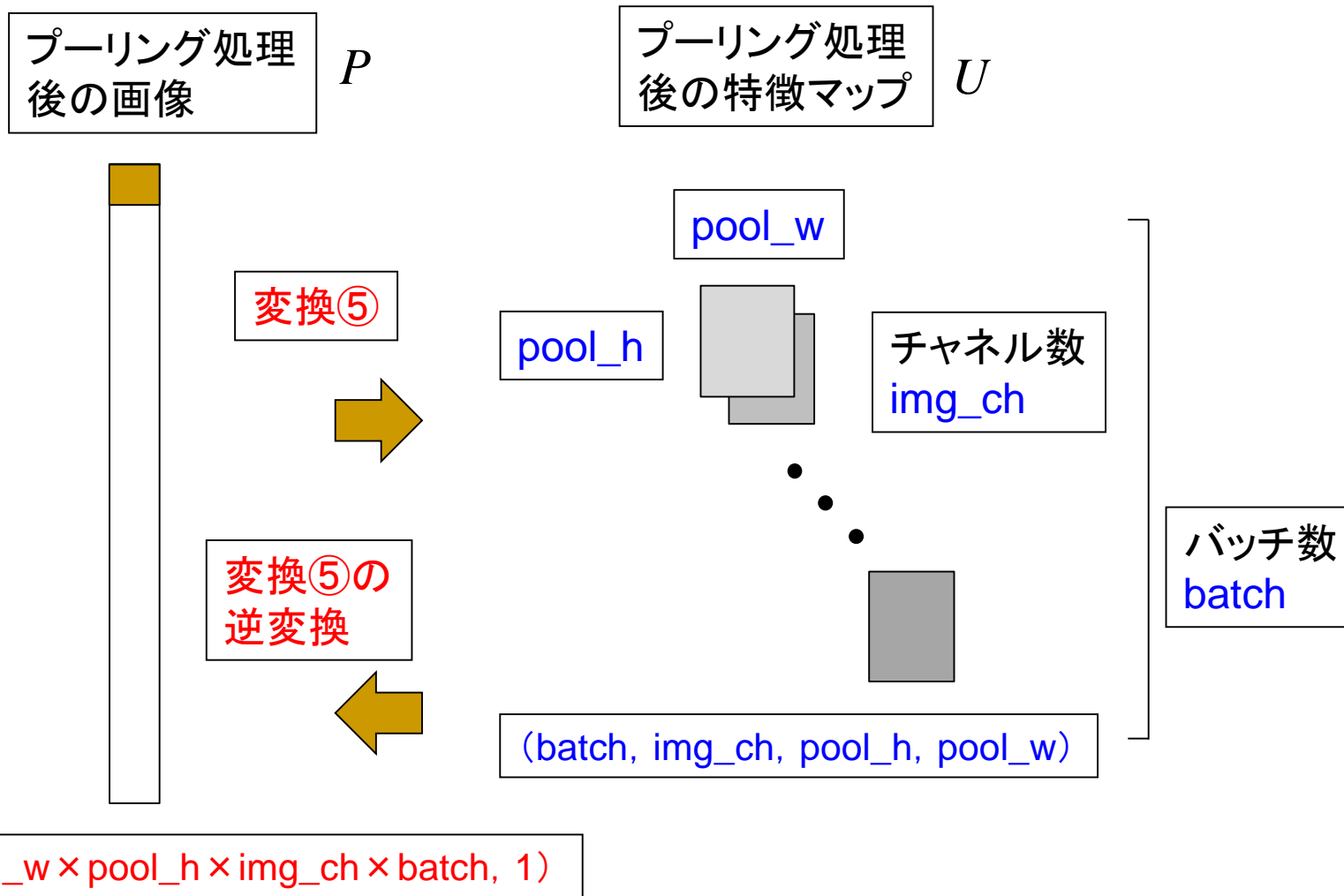
(pool\_w × pool\_h × img\_ch × batch, pool × pool)

(pool\_w × pool\_h × img\_ch × batch, 1)

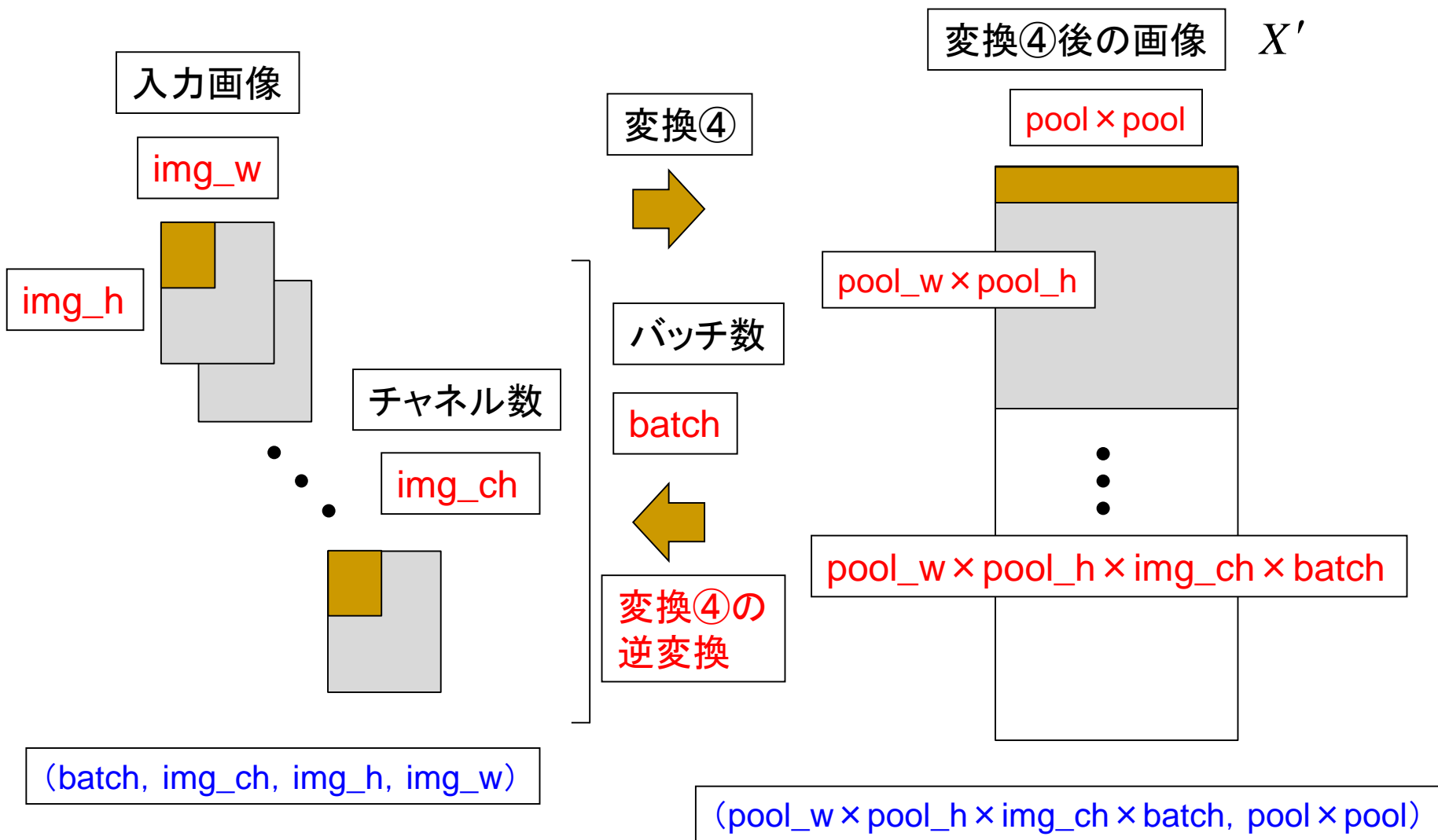
# プーリング層での逆伝播③



# プーリング層での逆伝播④



# プーリング層での逆伝播④



# 実習①

(多層のネットワークの学習)

# 多層のネットワークの学習 (DL.py)

- MNISTの数字画像認識
- MNISTのデータがあるフォルダーにプログラムは置いて下さい
- 「dat」というフォルダーを作成して下さい

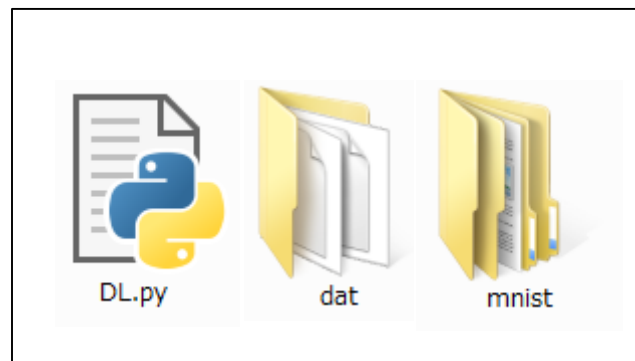
- 実行方法

- 学習

- > python DL.py t

- 認識

- > python DL.py p

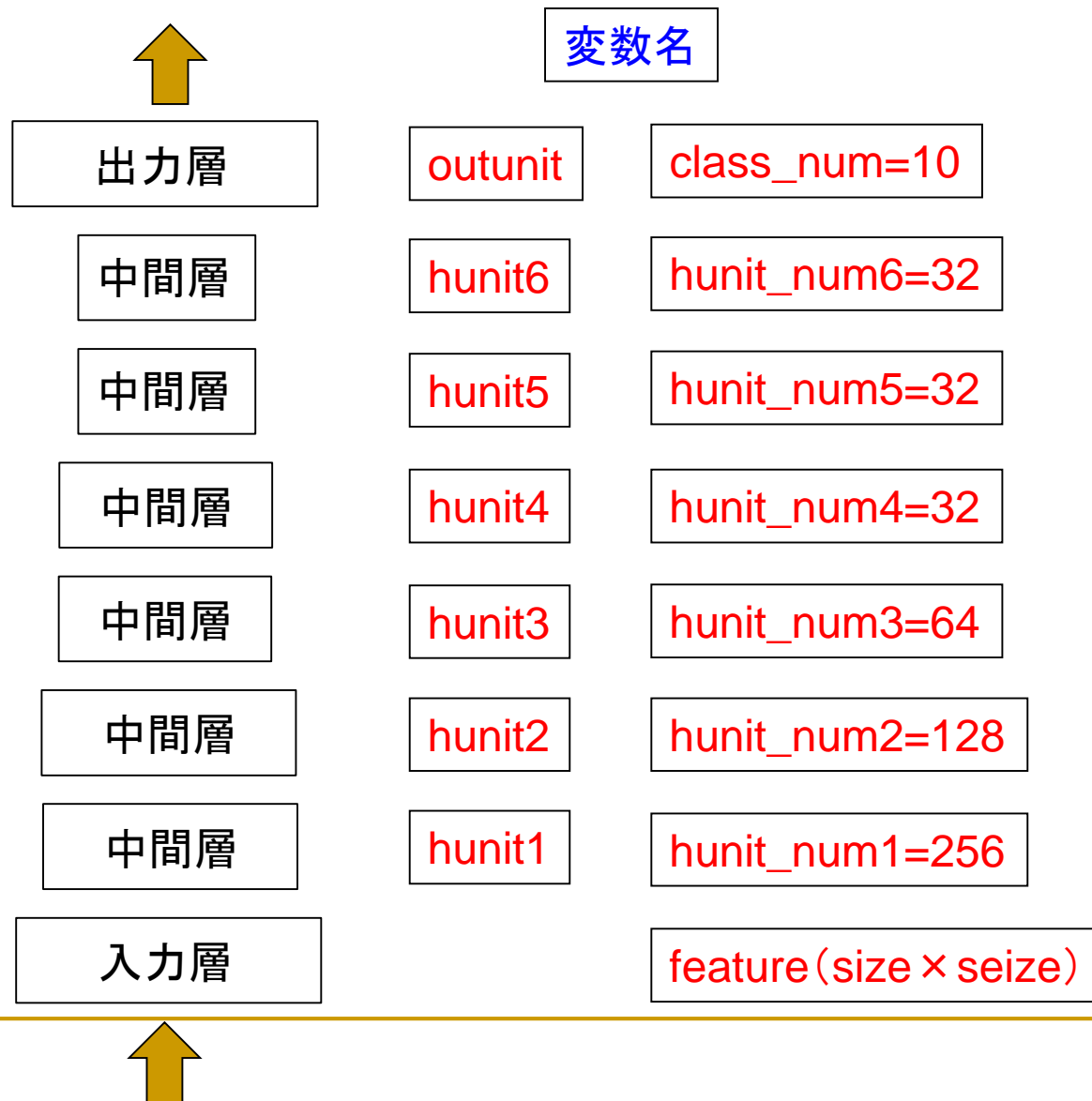


引数をつけて下さい

# ニューラルネットワークの構造①

- 出力層
  - 損失関数: 誤差二乗和
  - 活性化関数: シグモイド関数
  - 個数: クラス数(class\_num)
- 中間層(6層)
  - 活性化関数: ReLU関数
  - 個数: 次頁参照
- 入力層
  - 個数: 特徴数(feature)

# ニューラルネットワークの構造②





# メインメソッド

```
if __name__ == '__main__':
```

```
    # 中間層の個数
```

```
    hunit_num1 = 256
```

```
    hunit_num2 = 128
```

```
    hunit_num3 = 64
```

```
    hunit_num4 = 32
```

```
    hunit_num5 = 32
```

```
    hunit_num6 = 32
```

```
    # 中間層のコンストラクター
```

```
    hunit1 = Hunit( feature , hunit_num1 )
```

```
    hunit2 = Hunit( hunit_num1 , hunit_num2 )
```

```
    hunit3 = Hunit( hunit_num2 , hunit_num3 )
```

```
    hunit4 = Hunit( hunit_num3 , hunit_num4 )
```

```
    hunit5 = Hunit( hunit_num4 , hunit_num5 )
```

```
    hunit6 = Hunit( hunit_num5 , hunit_num6 )
```

プログラムはBP.pyを用いています  
違う箇所のみ説明します

中間層の個数:hunit\_num1  
一つ前の層(入力層)の個数:feature

コンストラクター(\_\_init\_\_)により, 配列を確保  
→初期化  
hunit1.w:feature × hunit\_num  
hunit1.b:hunit\_num

## # 出力層のコンストラクター

```
outunit = Outunit( hunit_num6 , class_num )
```

```
argvs = sys.argv
```

```
# 引数がtの場合
```

```
if argvs[1] == "t":
```

```
    # 学習データの読み込み
```

```
    flag = 0
```

```
    Read_data( flag )
```

```
    # 学習
```

```
    Train()
```

```
# 引数がpの場合
```

```
elif argvs[1] == "p":
```

```
    # テストデータの読み込み
```

```
    flag = 1
```

```
    Read_data( flag )
```

```
    # テストデータの予測
```

```
    Predict()
```

出力層の個数: class\_num

一つ前の層(中間層)の個数: hunit\_num6

コンストラクター( \_\_init\_\_ )により, 配列を確保  
→ 初期化

outunit.w: hunit\_num6 × class\_num

outunit.b: class\_num

flag=0

→ 学習データの読み込み

flag=1

→ テストデータの読み込み

# 学習①

```
def Train():
```

```
    # エポック数
```

```
    epoch = 200
```

学習回数: (epoch × class\_num × train\_num) 回

```
    for e in range( epoch ):
```

```
        error = 0.0
```

```
        for i in range(class_num):
```

```
            for j in range(0,train_num):
```

```
                # 入力データ
```

```
                rnd_c = np.random.randint(class_num)
```

```
                rnd_n = np.random.randint(train_num)
```

```
                input_data = data_vec[rnd_c][rnd_n].reshape(1,feature)
```

入力データ

(1 × feature)に変形

ランダムにクラス(rnd\_c), データ(rnd\_n)を選択し, 入力

# 学習②

① hunit1.Propagationメソッド  
入力値(input\_data)を渡す

② hunit2.Propagationメソッド  
中間層の出力値(hunit1.out)を渡す

③ hunit3.Propagationメソッド  
中間層の出力値(hunit2.out)を渡す

④ hunit4.Propagationメソッド  
中間層の出力値(hunit3.out)を渡す

⑤ hunit5.Propagationメソッド  
中間層の出力値(hunit4.out)を渡す

⑥ hunit6.Propagationメソッド  
中間層の出力値(hunit5.out)を渡す

⑦ outunit.Propagationメソッド  
中間層の出力値(hunit6.out)を渡す

# 伝播

```
hunit1.Propagation( input_data )  
hunit2.Propagation( hunit1.out )  
hunit3.Propagation( hunit2.out )  
hunit4.Propagation( hunit3.out )  
hunit5.Propagation( hunit4.out )  
hunit6.Propagation( hunit5.out )  
outunit.Propagation( hunit6.out )
```

# 教師信号

```
teach = np.zeros( (1,class_num) )  
teach[0][rnd_c] = 1
```

教師信号  
(1 × class\_num)  
→ rnd\_c番目の要素は1  
それ以外は0

# 学習③

# 誤差

```
outunit.Error( teach )  
hunit6.Error( outunit.error )  
hunit5.Error( hunit6.error )  
hunit4.Error( hunit5.error )  
hunit3.Error( hunit4.error )  
hunit2.Error( hunit3.error )  
hunit1.Error( hunit2.error )
```

① outunit.Errorメソッド  
教師信号 (teach) を渡す

② hunit6.Errorメソッド  
教師信号 (outunit.error) を渡す

③ hunit5.Errorメソッド  
教師信号 (hunit6.error) を渡す

④ hunit4.Errorメソッド  
教師信号 (hunit5.error) を渡す

⑤ hunit3.Errorメソッド  
教師信号 (hunit4.error) を渡す

⑥ hunit2.Errorメソッド  
教師信号 (hunit3.error) を渡す

⑦ hunit1.Errorメソッド  
教師信号 (hunit2.error) を渡す

## # 重みの修正

```
outunit.Update_weight()  
hunit6.Update_weight()  
hunit5.Update_weight()  
hunit4.Update_weight()  
hunit3.Update_weight()  
hunit2.Update_weight()  
hunit1.Update_weight()
```

出力層, 中間層での  
重みの修正

誤差二乗和

```
error += np.dot( ( outunit.out - teach ) , ( outunit.out - teach ).T )  
print( e , "->" , error )
```

## # 重みの保存

```
outunit.Save( "dat/BP-out.npz" )  
hunit1.Save( "dat/BP-hunit1.npz" )  
hunit2.Save( "dat/BP-hunit2.npz" )  
hunit3.Save( "dat/BP-hunit3.npz" )  
hunit4.Save( "dat/BP-hunit4.npz" )  
hunit5.Save( "dat/BP-hunit5.npz" )  
hunit6.Save( "dat/BP-hunit6.npz" )
```

outunit.Saveメソッド  
出力層の保存  
保存ファイル名("dat/BP-out.npz")を渡す

hunit1.Saveメソッド  
出力層の保存  
保存ファイル名("dat/BP-hunit1.npz")を渡す

# 予測

```
def Predict():
```

```
    # 重みのロード
```

```
    outunit.Load( "dat/BP-out.npz" )
```

```
    hunit1.Load( "dat/BP-hunit1.npz" )
```

```
    hunit2.Load( "dat/BP-hunit2.npz" )
```

```
    hunit3.Load( "dat/BP-hunit3.npz" )
```

```
    hunit4.Load( "dat/BP-hunit4.npz" )
```

```
    hunit5.Load( "dat/BP-hunit5.npz" )
```

```
    hunit6.Load( "dat/BP-hunit6.npz" )
```

```
    # 混合行列
```

```
    result = np.zeros((class_num,class_num), dtype=np.int32)
```

```
    for i in range(class_num):
```

```
        for j in range(0,train_num):
```

```
            # 入力データ
```

```
            input_data = data_vec[i][j].reshape(1,feature)
```

outunit.Loadメソッド  
出力層のロード  
ロードしたいファイル名  
("dat/BP-out.npz")を渡す

hunit1.Loadメソッド  
中間層のロード  
ロードしたいファイル名  
("dat/BP-hunit1.npz")を渡す

入力データ  
(1 × feature)に変形

## # 伝播

```
hunit1.Propagation( input_data )  
hunit2.Propagation( hunit1.out )  
hunit3.Propagation( hunit2.out )  
hunit4.Propagation( hunit3.out )  
hunit5.Propagation( hunit4.out )  
hunit6.Propagation( hunit5.out )  
outunit.Propagation( hunit6.out )
```

## # 教師信号

```
teach = np.zeros( (1,class_num) )  
teach[0][i] = 1
```

### 教師信号

(1 × class\_num)  
→ i番目の要素は1  
それ以外は0

## # 予測

```
ans = np.argmax( outunit.out[0] )  
result[i][ans] +=1  
print( i , j , "->" , ans )
```

### outunit.out

(1 × class\_num)

### np.argmax(配列)

配列中, 最大値の要素番号を返す



```
print( "¥n [混合行列]" )
```

```
print( result )
```

```
print( "¥n 正解数 ->" , np.trace(result) )
```

混合行列の表示  
正解数の表示

- 確認して下さい
- 学習が停止することは多くありませんか
- 3層のネットワークと比較して精度は向上しましたか



```
C:\Windows\system32\cmd.exe
9 98 -> 9
9 99 -> 5

[混合行列]
[[96  0  0  0  2  1  1  0  0  0]
 [ 0 90  1  0  0  0  0  1  7  1]
 [ 9  0 54  0  0  3  9  3 18  4]
 [ 0  0  0 32  0 53  1  2 11  1]
 [ 0  0  0  0 82  0  7  0  1 10]
 [ 3  0  0  0  3 85  1  2  4  2]
 [ 3  0  1  0  3  2 91  0  0  0]
 [ 0  0  3  0  3  0  0 71  5 18]
 [ 2  0  0  0  6 14  2  1 69  6]
 [ 0  0  0  0 19  3  0  0  1 77]]

正解数 -> 747

C:\home\shino\prml-2018\12-10\program>
```

## 実習②(オートエンコーダ)

# オートエンコーダ (AE.py)

- MNISTの数字画像を対象
- MNISTのデータがあるフォルダーにプログラムは置いて下さい
- 「fig」「dat」というフォルダーを作成して下さい

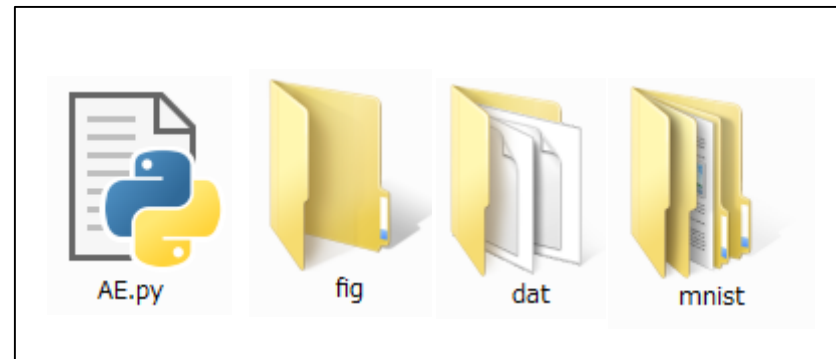
- 実行方法

- 学習

- > python AE.py t

- 認識

- > python AE.py p



引数をつけて下さい

# ニューラルネットワークの構造

- 出力層
  - 損失関数: 誤差二乗和
  - 活性化関数: 恒等関数
  - 個数: 特徴数 ( $\text{feature} = \text{size} \times \text{size}$ )
- 中間層
  - 活性化関数: ソフトマックス関数\*
  - 個数: 32個, 100個
- 入力層
  - 個数: 特徴数 ( $\text{feature} = \text{size} \times \text{size}$ )

\*正確には活性化関数は恒等関数, 中間層からの出力値の合計を1としています

# 変数の宣言

# クラス数

class\_num = 10

プログラムはBP.pyを用いています

# 画像の大きさ

size = 14

feature = size \* size

feature

入力層の個数(特徴数)

# 学習データ数

train\_num = 100

data\_vec

(クラス数, 学習(テスト)データ数, 特徴数)

# データ

data\_vec = np.zeros((class\_num, train\_num, feature), dtype=np.float64)

# 学習係数

alpha = 0.1

### # シグモイド関数

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

### # シグモイド関数の微分

```
def Sigmoid_( x ):
    return ( 1-Sigmoid(x) ) * Sigmoid(x)
```

### # ReLU関数

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

### # ReLU関数の微分

```
def ReLU_( x ):
    return np.where( x > 0, 1, 0 )
```

### # ソフトマックス関数

```
def Softmax( x ):
    return np.exp(x)/np.sum(np.exp(x), axis=1, keepdims=True)
```

# 出力層のクラス①

```
class Outunit:
```

```
    def __init__(self, m, n):
```

```
        # 重み
```

```
        self.w = np.random.uniform(-0.5,0.5,(m,n))
```

```
        # 閾値
```

```
        self.b = np.random.uniform(-0.5,0.5,n)
```

```
    def Propagation(self, x):
```

```
        self.x = x
```

```
        # 内部状態
```

```
        self.u = np.dot(self.x, self.w) + self.b
```

```
        # 出力値(恒等関数)
```

```
        self.out = self.u
```

m: 出力層の個数

n: 一つ下の層(中間層)の個数

重み: (n × m) の行列

→ 0.5から0.5の乱数で初期化

閾値: m次元のベクトル

→ 0.5から0.5の乱数で初期化

$$\mathbf{u}_p = \mathbf{x}_p W + \mathbf{b}_p$$

$$\mathbf{o}_p = \mathbf{u}_p$$

# 出力層のクラス②

```
def Error(self, t):
```

```
    # 誤差
```

```
    f_ = 1
```

$f_$ : 活性化関数の微分

```
    delta = ( self.out - t ) * f_
```

```
    # 重み, 閾値の修正値
```

```
    self.grad_w = np.dot(self.x.T, delta)
```

```
    self.grad_b = np.sum(delta, axis=0)
```

```
    # 前の層に伝播する誤差
```

```
    self.error = np.dot(delta, self.w.T)
```

```
def Update_weight(self):
```

```
    # 重み, 閾値の修正
```

```
    self.w -= alpha * self.grad_w
```

```
    self.b -= alpha * self.grad_b
```

損失関数が誤差二乗和  
活性化関数が恒等関数の場合

$$\frac{\partial E}{\partial V_{kj}} = \frac{\partial E}{\partial S_k} \frac{\partial S_k}{\partial V_{kj}} = (O_k - t_k) H_j$$

誤差

$$\partial W = \mathbf{x}^t \Delta$$

$$\partial \mathbf{b} = \mathbf{1}^t \Delta$$

$$\Delta W^t$$

$$W' = W - \alpha \partial W$$

$$\mathbf{b}' = \mathbf{b} - \alpha \partial \mathbf{b}$$



# 出力層のクラス③

```
def Save(self, filename):
```

```
# 重み, 閾値の保存
```

```
np.savez(filename, w=self.w, b=self.b)
```

```
def Load(self, filename):
```

```
# 重み, 閾値のロード
```

```
work = np.load(filename)
```

```
self.w = work['w']
```

```
self.b = work['b']
```

np.savez

numpy形式のデータの保存(バイナリ)

np.savez(ファイル名, 変数名)

→ ファイル名.npzとして保存

重み→キー「w」

閾値→キー「b」

np.load

numpy形式のデータのロード

np.load(ファイル名)

キー「w」→重み

キー「b」→閾値

# 中間層のクラス①

```
class Hunit:
```

```
def __init__(self, m, n):
```

```
    # 重み
```

```
    self.w = np.random.uniform(-0.5,0.5,(m,n))
```

```
    # 閾値
```

```
    self.b = np.random.uniform(-0.5,0.5,n)
```

```
def Propagation(self, x):
```

```
    self.x = x
```

```
    # 内部状態
```

```
    self.u = np.dot(self.x, self.w) + self.b
```

```
    # 出力値(ソフトマックス関数)
```

```
    self.out = Softmax( self.u )
```

m: 中間層の個数

n: 一つ下の層(入力層)の個数

重み:  $(n \times m)$  の行列  
→ 0.5から0.5の乱数で初期化

閾値: m次元のベクトル  
→ 0.5から0.5の乱数で初期化

x: 入力ベクトル(n次元)

$$\mathbf{u}_p = \mathbf{x}_p W + \mathbf{b}_p$$

$$\mathbf{o}_p = f(\mathbf{u}_p)$$

# 中間層のクラス②

```
def Error(self, p_error):
```

```
    # 誤差
```

```
    f_ = 1
```

```
    delta = p_error * f_
```

```
    # 重み, 閾値の修正値
```

```
    self.grad_w = np.dot(self.x.T, delta)
```

```
    self.grad_b = np.sum(delta, axis=0)
```

```
    # 前の層に伝播する誤差
```

```
    self.error = np.dot(delta, self.w.T)
```

```
def Update_weight(self):
```

```
    # 重み, 閾値の修正
```

```
    self.w -= alpha * self.grad_w
```

```
    self.b -= alpha * self.grad_bZ
```

p\_error:

一つ上の層(出力層)逆伝播してきた誤差(m次元)


$$\delta_j = \left( \sum_k V_{kj} \delta_k \right) f'(S_j)$$

$$\partial W = \mathbf{x}^t \Delta$$

$$\partial \mathbf{b} = \mathbf{1}^t \Delta$$

$$\Delta W^t$$

$$W' = W - \alpha \partial W$$

$$\mathbf{b}' = \mathbf{b} - \alpha \partial \mathbf{b}$$

# 中間層のクラス③

```
def Save(self, filename):
```

```
    # 重み, 閾値の保存
```

```
    np.savez(filename, w=self.w, b=self.b)
```

```
def Load(self, filename):
```

```
    # 重み, 閾値のロード
```

```
    work = np.load(filename)
```

```
    self.w = work['w']
```

```
    self.b = work['b']
```

np.savez

numpy形式のデータの保存(バイナリ)

np.savez(ファイル名, 変数名)

→ ファイル名.npzとして保存

重み→キー「w」

閾値→キー「b」

np.load

numpy形式のデータのロード

np.load(ファイル名)

キー「w」→重み

キー「b」→閾値

# データの読み込み

```
def Read_data( flag ):  
    dir = [ "train" , "test" ]  
    for i in range(class_num):  
        for j in range(1,train_num+1):
```

flagが0の場合→学習データ(「mnist/train/」)  
flagが1の場合→テストデータ(「mnist/test/」)  
からデータを読み込む

```
        # グレースケール画像で読み込み→大きさの変更→numpyに変換, ベクトル化  
        train_file = mnist/" + dir[ flag ] + "/" + str(i) + "/" + str(i) + "_" + str(j) + ".jpg"  
        work_img = Image.open(train_file).convert('L')  
        resize_img = work_img.resize((size, size))  
        data_vec[i][j-1] = np.asarray(resize_img).astype(np.float64).flatten()  
  
        # 入力値の合計を1とする  
        data_vec[i][j-1] = data_vec[i][j-1] / np.sum( data_vec[i][j-1] )
```

# メインメソッド

```
if __name__ == '__main__':
```

```
# 中間層の個数
```

```
hunit_num = 32
```

```
# 中間層のコンストラクター
```

```
hunit = Hunit( feature , hunit_num )
```

```
# 出力層のコンストラクター
```

```
outunit = Outunit( hunit_num , feature )
```

```
argvs = sys.argv
```

中間層の個数: hunit\_num  
一つ前の層(入力層)の個数: feature

コンストラクター(\_\_init\_\_)により, 配列を確保  
→初期化  
hunit.w: feature × hunit\_num  
hunit.b: hunit\_num

出力層の個数: class\_num  
一つ前の層(中間層)の個数: hunit\_num

コンストラクター(\_\_init\_\_)により, 配列を確保  
→初期化  
outunit.w: hunit\_num × feature  
outunit.b: feature

# 引数がtの場合

if args[1] == "t":

# 学習データの読み込み

flag = 0

Read\_data( flag )

flag=0

→学習データの読み込み

# 学習

Train()

# 引数がpの場合

elif args[1] == "p":

os.system( "del fig\*.png" )

# テストデータの読み込み

flag = 1

Read\_data( flag )

flag=1

→テストデータの読み込み

# テストデータの予測

Predict()

# 学習

```
def Train():
```

```
    # エPOCH数
```

```
    epoch = 1000
```

学習回数: (epoch × class\_num × train\_num) 回

```
    for e in range( epoch ):
```

```
        error = 0.0
```

```
        for i in range(class_num):
```

```
            for j in range(0,train_num):
```

```
                # 入力データ
```

```
                rnd_c = np.random.randint(class_num)
```

```
                rnd_n = np.random.randint(train_num)
```

```
                input_data = data_vec[rnd_c][rnd_n].reshape(1,feature)
```

入力データ  
(1 × feature)に変形

ランダムにクラス(rnd\_c), データ(rnd\_n)を選択し, 入力

```
                # 伝播
```

```
                huntunit.Propagation( input_data )
```

```
                outunit.Propagation( huntunit.out )
```

huntunit.Propagationメソッド  
入力値(input\_data)を渡す

outunit.Propagationメソッド  
中間層の出力値(huntunit.out)を渡す



### # 教師信号

```
teach = data_vec[rnd_c][rnd_n].reshape(1,feature)
```

教師信号  
(1 × feature)  
→ 入力情報と同じ

### # 誤差

```
outunit.Error( teach )
```

outunit.Errorメソッド  
教師信号(teach)を渡す

```
hunit.Error( outunit.error )
```

hunit.Errorメソッド  
出力層から逆伝播される誤差  
(outunit.error)を渡す

### # 重みの修正

```
outunit.Update_weight()
```

```
hunit.Update_weight()
```

誤差二乗和

```
error += np.dot( ( outunit.out - teach ) , ( outunit.out - teach ).T )  
print( e , "->" , error )
```

### # 重みの保存

```
outunit.Save( "dat/BP-out.npz" )
```

outunit.Saveメソッド  
出力層の保存  
保存ファイル名("dat/BP-out.npz")を渡す

```
hunit.Save( "dat/BP-hunit.npz" )
```

hunit.Saveメソッド  
出力層の保存  
保存ファイル名("dat/BP-hunit.npz")を渡す

# 予測

```
def Predict():
```

```
    # 重みのロード
```

```
    outunit.Load( "dat/BP-out.npz" )
```

```
    hunit.Load( "dat/BP-hunit.npz" )
```

```
    # 混合行列
```

```
    result = np.zeros((class_num,class_num), dtype=np.int32)
```

```
    for i in range(class_num):
```

```
        for j in range(0,train_num):
```

```
            # 入力データ
```

```
            input_data = data_vec[i][j].reshape(1,feature)
```

```
            # 伝播
```

```
            hunit.Propagation( input_data )
```

```
            outunit.Propagation( hunit.out )
```

outunit.Loadメソッド  
出力層のロード  
ロードしたいファイル名  
("dat/BP-out.npz")を渡す

hunit.Loadメソッド  
中間層のロード  
ロードしたいファイル名  
("dat/BP-hunit.npz")を渡す

入力データ  
(1 × feature)に変形

hunit.Propagationメソッド  
入力値(input\_data)を渡す

outunit.Propagationメソッド  
中間層の出力値(hunit.out)を渡す

```
if j < 1:
```

```
# 画像の描画
```

```
plt.figure()
```

```
# 元画像の表示
```

```
plt.subplot(1,2,1)
```

```
plt.imshow(np.reshape(data_vec[i][j],(size,size)),cmap='gray')
```

```
plt.title( "Original Image" )
```

```
# 復元画像の表示
```

```
plt.subplot(1,2,2)
```

```
plt.imshow(np.reshape(outunit.out,(size,size)),cmap='gray')
```

```
# 画像の保存
```

```
plt.title( "Decode Image(" + str(i) + "," + str(j) + ")" )
```

```
file = "fig/decode-" + str(i) + "-" + str(j) + "-result.png"
```

```
plt.savefig(file)
```

```
plt.close()
```

元画像

復元画像

## # 結合係数の画像化

```
g_size = 10
```

```
plt.figure(figsize=(g_size,g_size))
```

最大(10×10)個の領域  
に結合係数を画像化

```
count = 1
```

```
for i in range(hunit_num):
```

```
    plt.subplot(g_size,g_size,count)
```

hunit.w  
feature × hunit\_num

```
    plt.imshow(np.reshape(hunit.w[:,i],(size,size)),cmap='gray')
```

```
    plt.xticks(color="None")
```

```
    plt.yticks(color="None")
```

X, Y軸上の目盛り, 目盛り線を表示しない

```
    plt.tick_params(length=0)
```

```
    count += 1
```

```
file = "fig/hunit-weight.png"
```

```
plt.savefig(file)
```

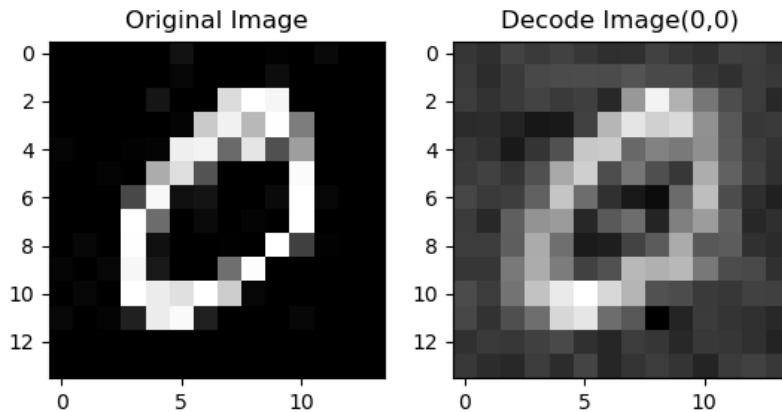
画像の保存

```
plt.close()
```

# 入力画像と出力画像①

中間層数は32個

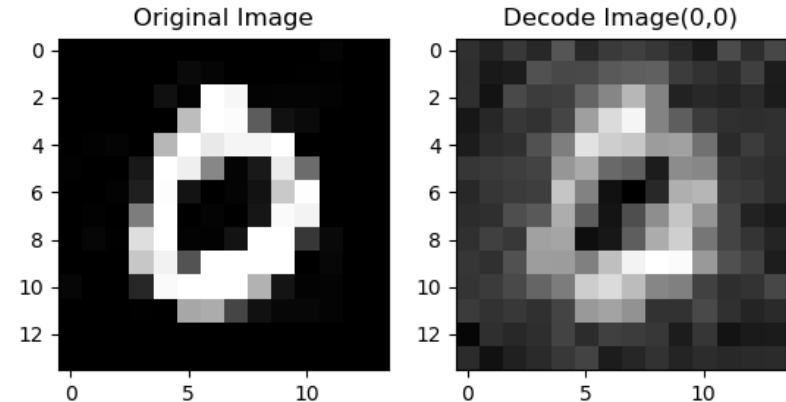
学習データ



入力画像

出力画像

テストデータ



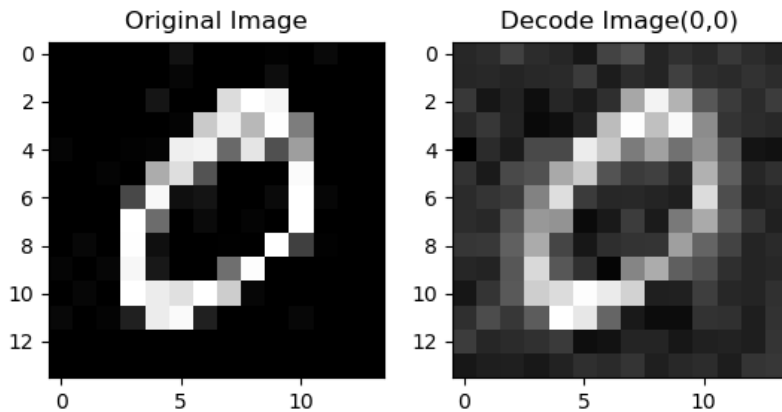
入力画像

出力画像

# 入力画像と出力画像②

中間層数は100個

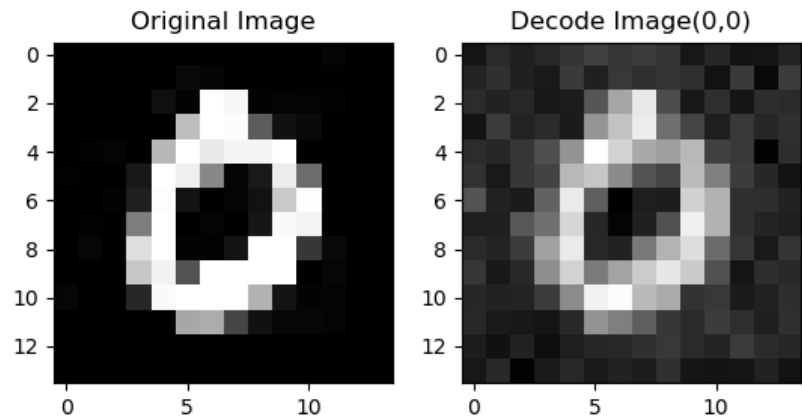
学習データ



入力画像

出力画像

テストデータ

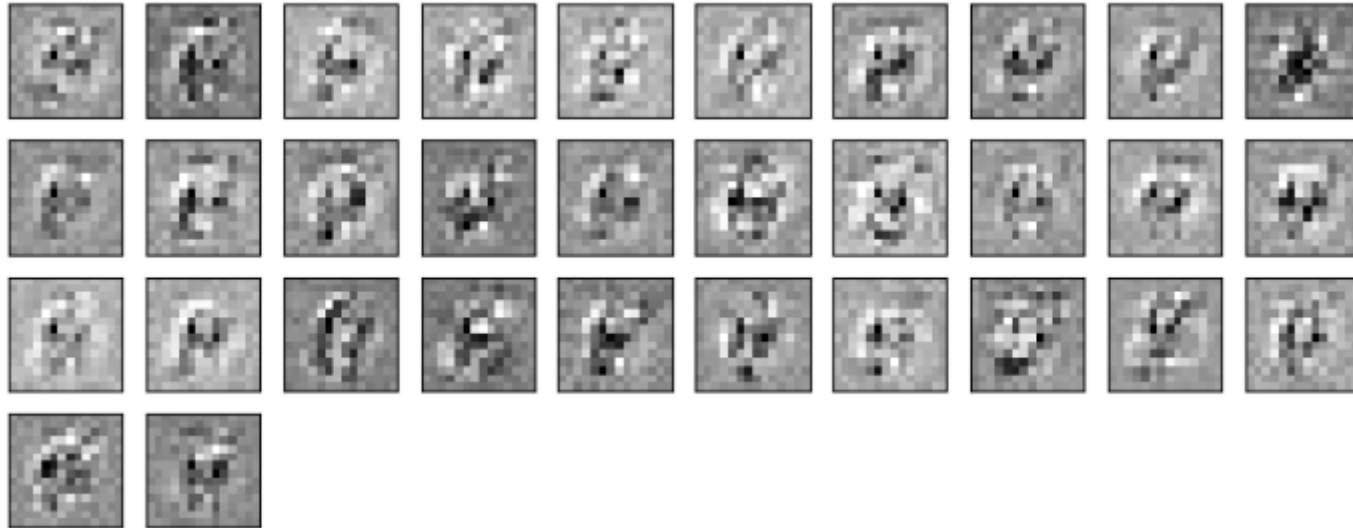


入力画像

出力画像

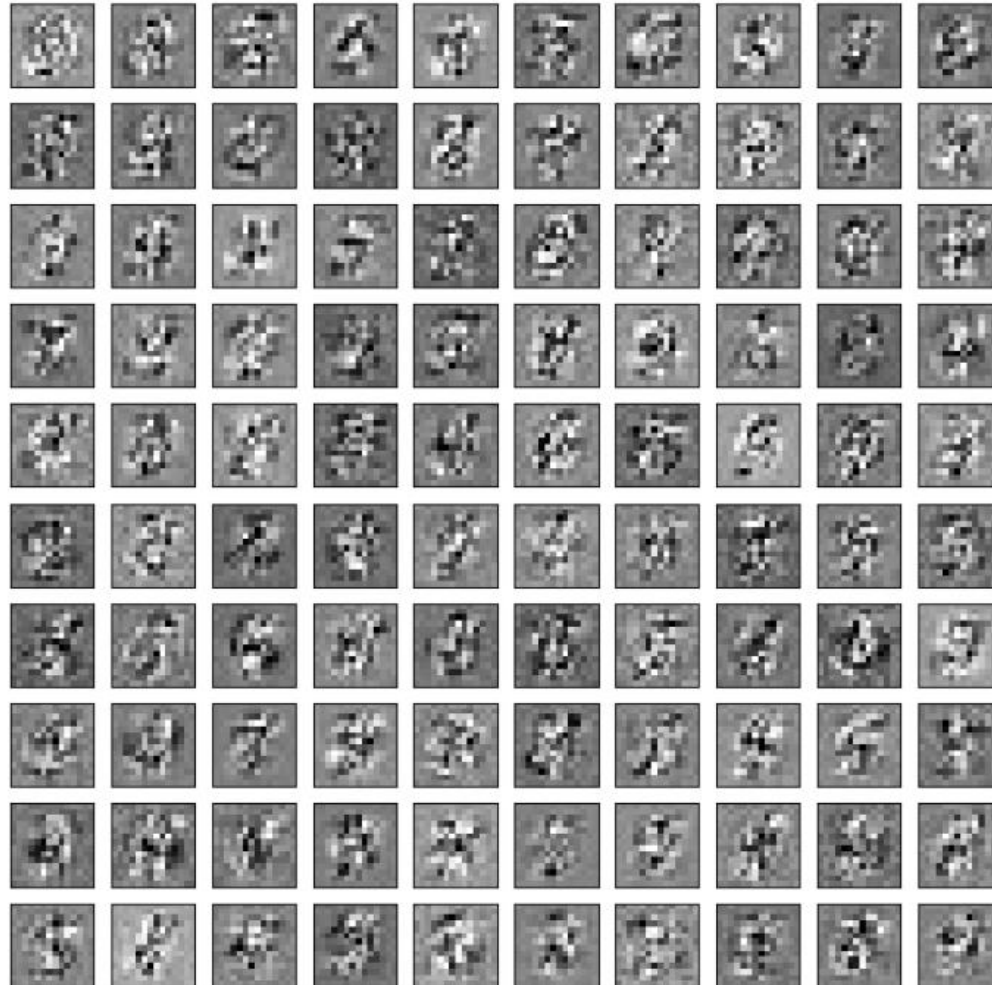
# エンコーダの結合係数の視覚化①

中間層数は32個



# エンコーダの結合係数の視覚化②

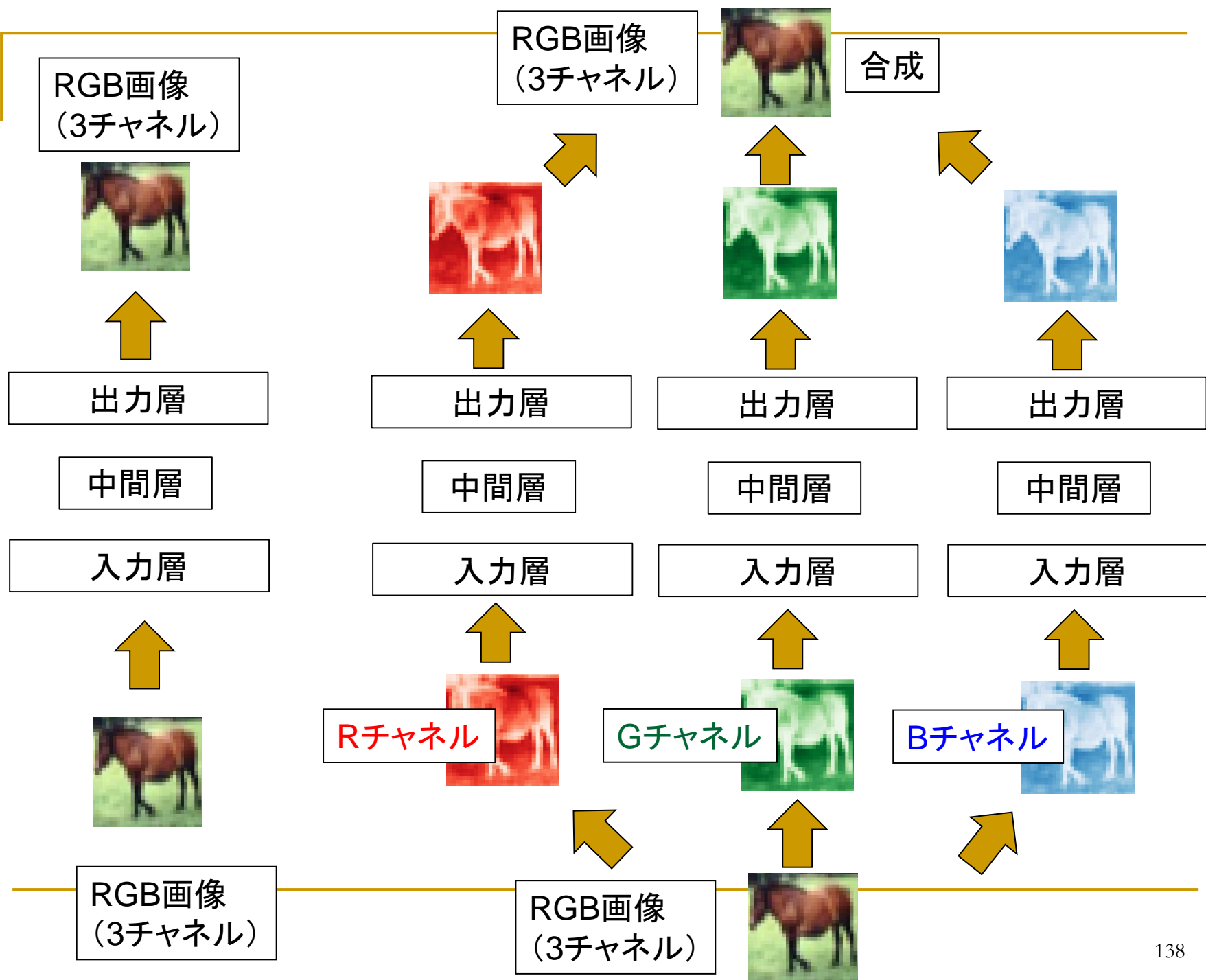
中間層数は100個





## 宿題⑩

- CIFAR-10の画像(学習データ1,000枚)を対象として, 3層のオートエンコーダを学習しなさい.
- ただし, RGB画像(3チャンネル)として学習させることは困難かと思います.
- 3チャンネルごとに(Rチャンネル, Gチャンネル, Bチャンネルごとに)3個のオートエンコーダを学習して下さい.
- 最後に, 3個のオートエンコーダからの出力を3チャンネルにまとめて, 元の画像に復元されるか確認して下さい.



# データ

```
data_vec = np.zeros((class_num,train_num,3,feature), dtype=np.float64)
```

(class\_num × train\_num × 3 × feature)

# データの読み込み

```
def Read_data( flag ):
```

```
    dir = [ "train" , "test" ]
```

```
    dir1 = [ "airplane" , "automobile" , "bird" , "cat" , "deer" , "dog" , "frog" , "horse" , "ship" ,  
            "truck" ]
```

```
    for i in range(class_num):
```

```
        for j in range(0,train_num):
```

# RGB画像で読み込み→大きさの変更→numpyに変換, ベクトル化

```
        train_file = cifar-10/" + dir[ flag ] + "/" + dir1[i] + "/" + str(j) + ".png"
```

```
        work_img = Image.open(train_file).convert('RGB')
```

```
        resize_img = work_img.resize((size, size))
```

(size,size,3)の大きさに変更

```
        work = np.resize( np.asarray(resize_img).astype(np.float64) , (size,size,3) )
```

```
        data_vec[i][j][0] = work[:, :, 0].flatten()
```

R画像

```
        data_vec[i][j][1] = work[:, :, 1].flatten()
```

G画像

```
        data_vec[i][j][2] = work[:, :, 2].flatten()
```

B画像

# 次頁の出力結果で用いたオートエンコーダの構造

## ■ 出力層

- 損失関数: 誤差二乗和
- 活性化関数: シグモイド関数
- 個数: 特徴数 ( $32 \times 32$ )

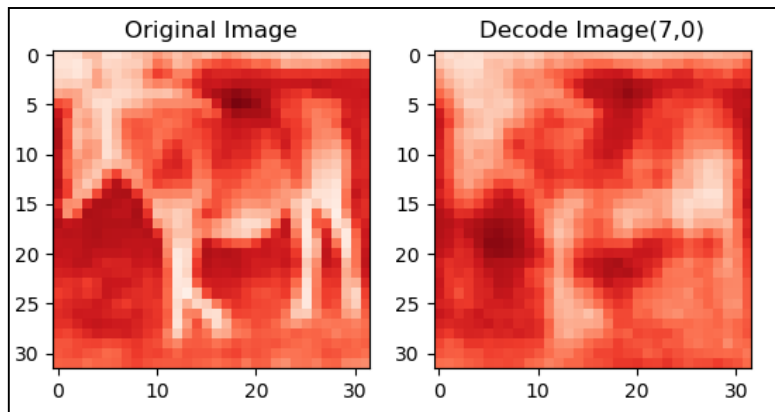
## ■ 中間層

- 活性化関数: シグモイド関数
- 個数: 100個

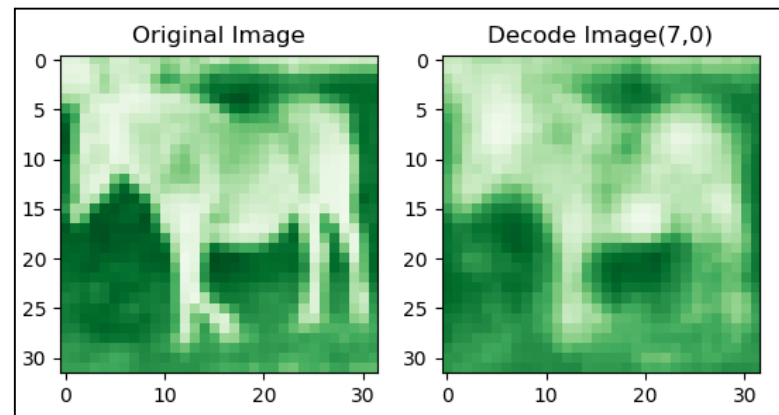
## ■ 入力層

- 個数: 特徴数 ( $32 \times 32$ )
- 入力値: 画素値を255で除算 (0~1)

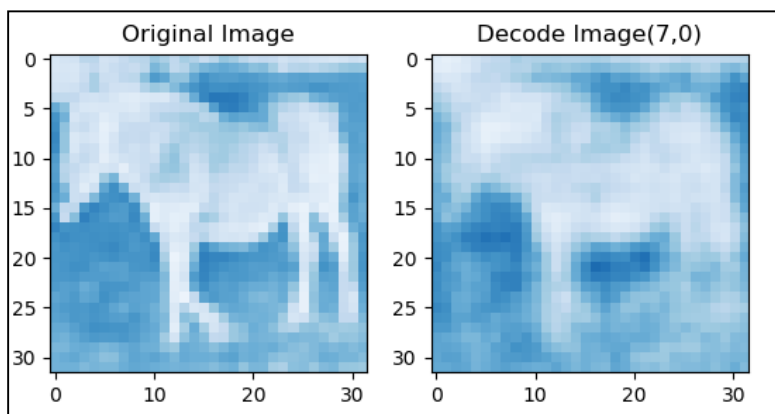
Rチャンネル



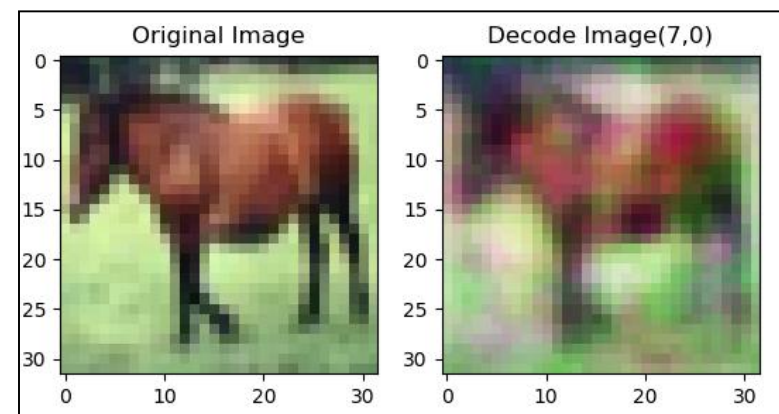
Gチャンネル



Bチャンネル



RGB画像(3チャンネル)



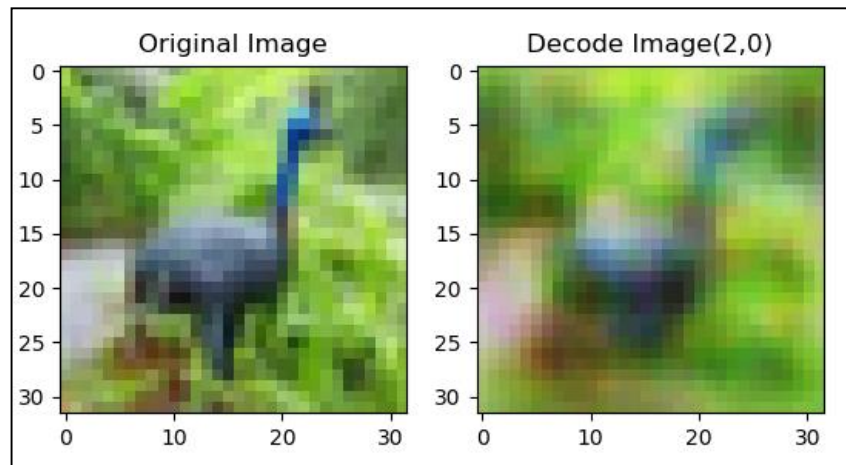
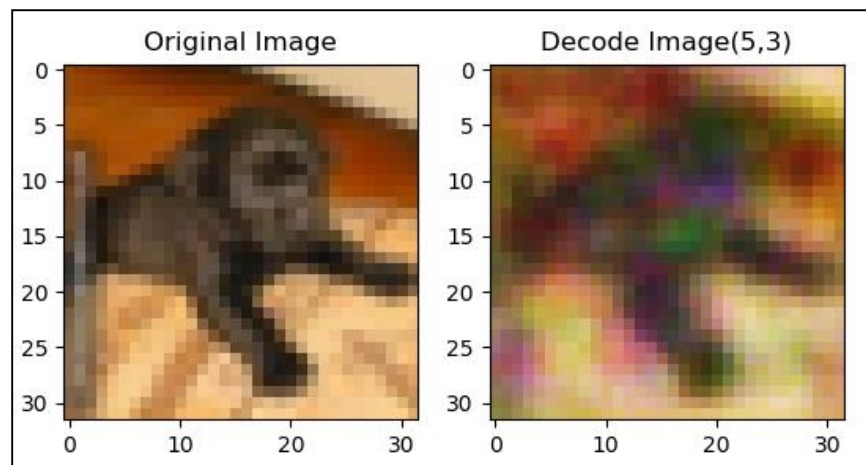
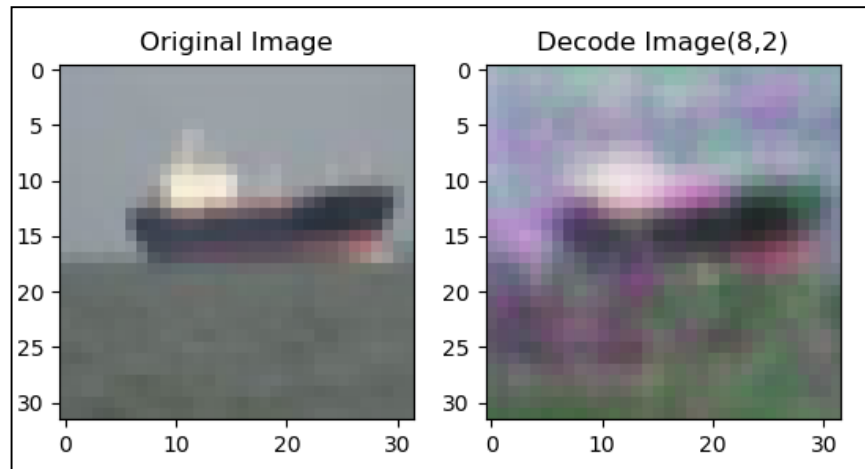
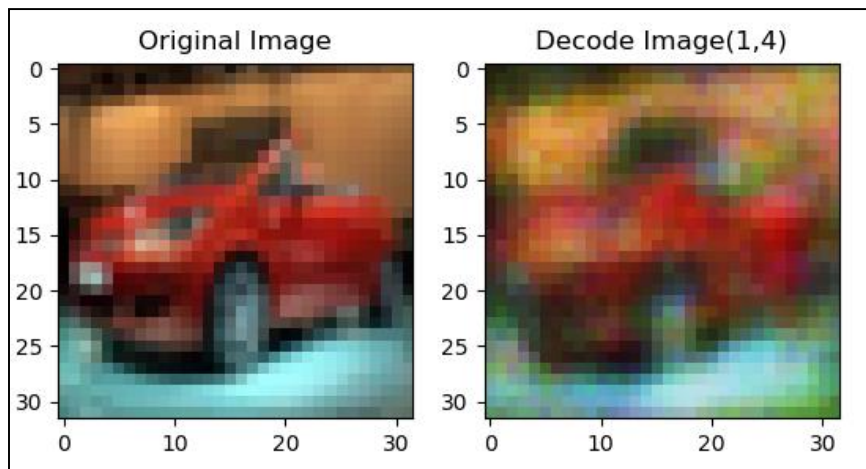
入力画像

出力画像

# 出力画像の例

入力画像

出力画像



# numpy配列をRGB画像にする場合の注意

```
vec = np.zeros((3,size*size), dtype=np.float64)
```

```
# (3, size, size)に変換
```

```
img_vec = np.resize( vec , (3,size,size) )
```

```
# (size, size, 3)に変換
```

```
img_vec = np.transpose( img_vec , (1,2,0) )
```

```
# 画像化
```

```
img = Image.fromarray(np.uint8(img_vec))
```

vec

numpy配列 (3, size\*size)

vec[0] R値

vec[1] G値

vec[2] B値

# (本日の)参考文献

- J.デイホフ:ニューラルネットワークアーキテクチャ入門, 森北出版(1992)
- P.D.Wasserman:ニューラル・コンピューティング, 理論と実際, 森北出版(1993)
- C.M.ビショップ:パターン認識と機械学習(上), シュプリンガー・ジャパン(2007)
- 岡谷貴之:深層学習, 講談社(2015)
- 瀧雅人:これならわかる深層学習入門, 講談社(2017)
- 原田達也:画像認識, 講談社(2017)