Machine Learning Techniques

Shyi-Chyi Cheng (鄭錫齊)
Email:csc@mail.ntou.edu.tw
Tel: 02-24622192-6653

章節目錄

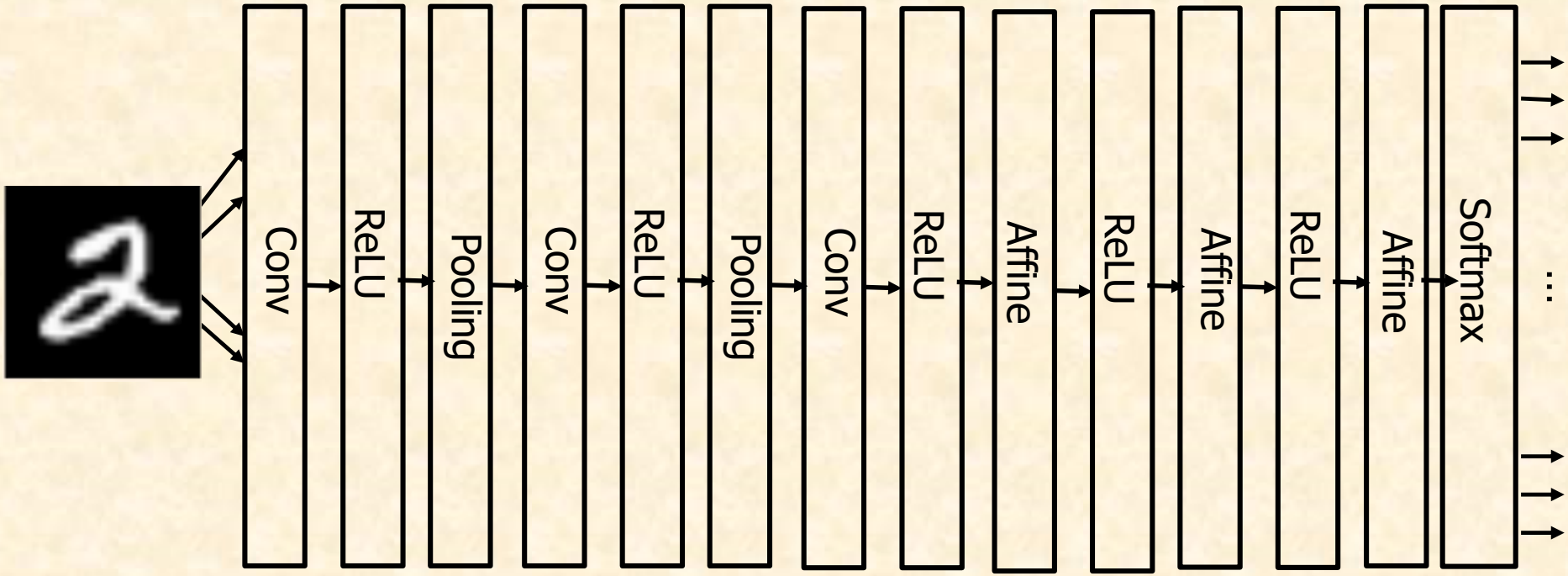
卷積神經網路(CNN)

學習重點

- CNN在全連結層的網路中，新加入卷積層及池化層。
- 使用Python之im2col(將影像展開成陣列的函數)，就可以輕鬆快速執行卷積層及池化層。
- 將CNN視覺化，可以瞭解加深層數之後，擷取出的高階資料模樣。
- CNN的代表網路包括LeNet與AlexNet
- 大數據與GPU對深度學習發展的重要貢獻。

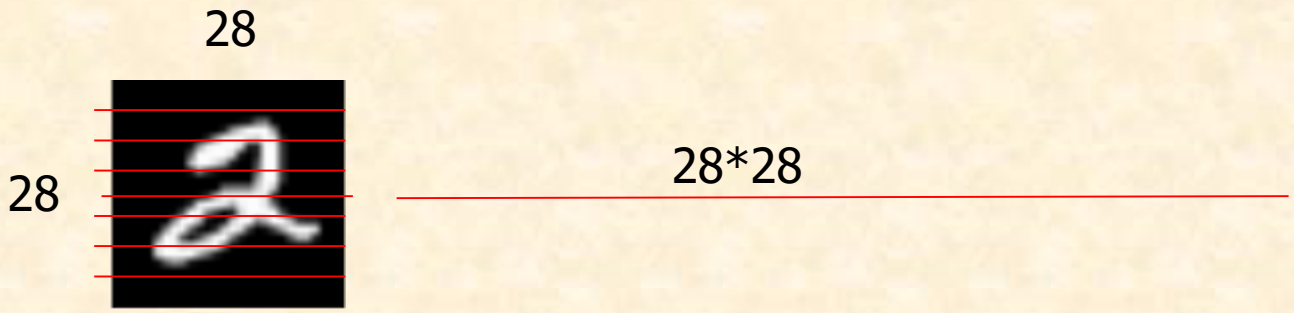
整體架構

- 卷積神經網路(convolutional neural network; CNN)已廣泛應用在各種辨識相關的應用。
- CNN在全連結(Affine)層的網路中，新加入卷積層(convolution)及池化(Pooling)層。



全連結(Affine)層的問題

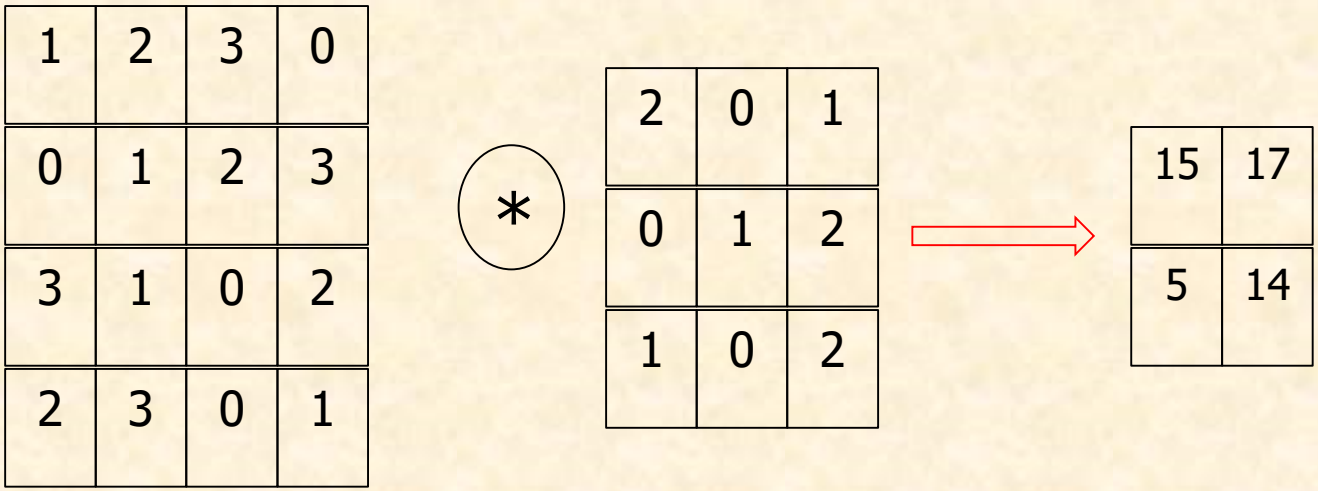
- 全連結(Affine)層忽略資料的形狀



- 例如影像資料包含水平、垂直、及顏色三維形狀，而神經網路之Affine層要求把資料排列成一維資料(平面結構)
- 資料形狀具重要的空間資訊，不應被去除
- 卷積層的目的在於維持資料的形狀資訊，以利產生包含形狀資訊的特徵圖(feature map)

卷積運算

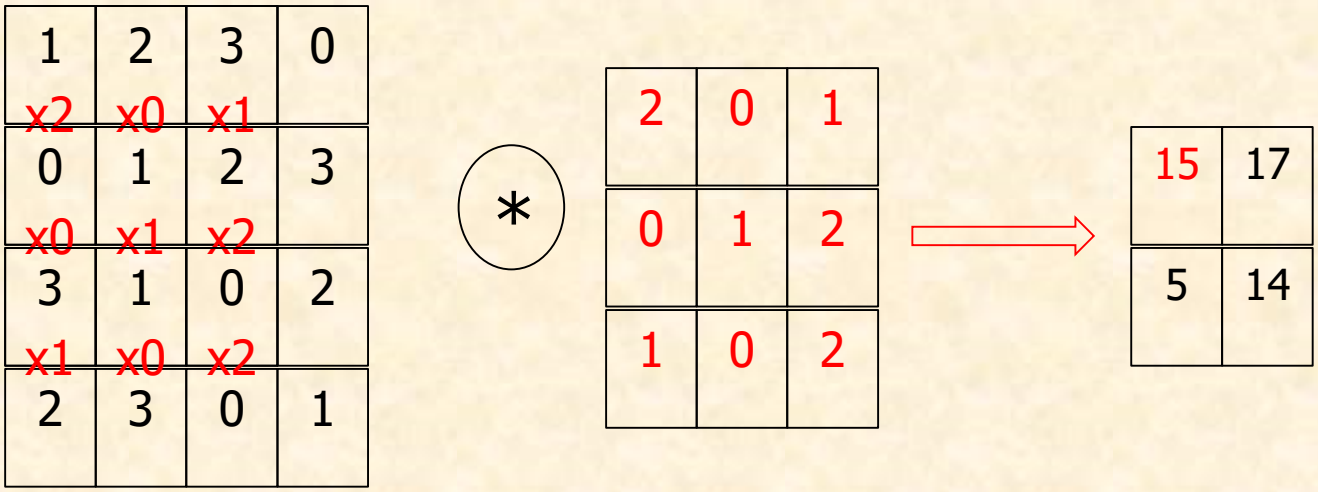
- 在卷積層執行的處理叫卷積運算
- 卷積運算是一種濾鏡(Filter)運算



輸入資料　　卷積運算　濾鏡(核，kernel)

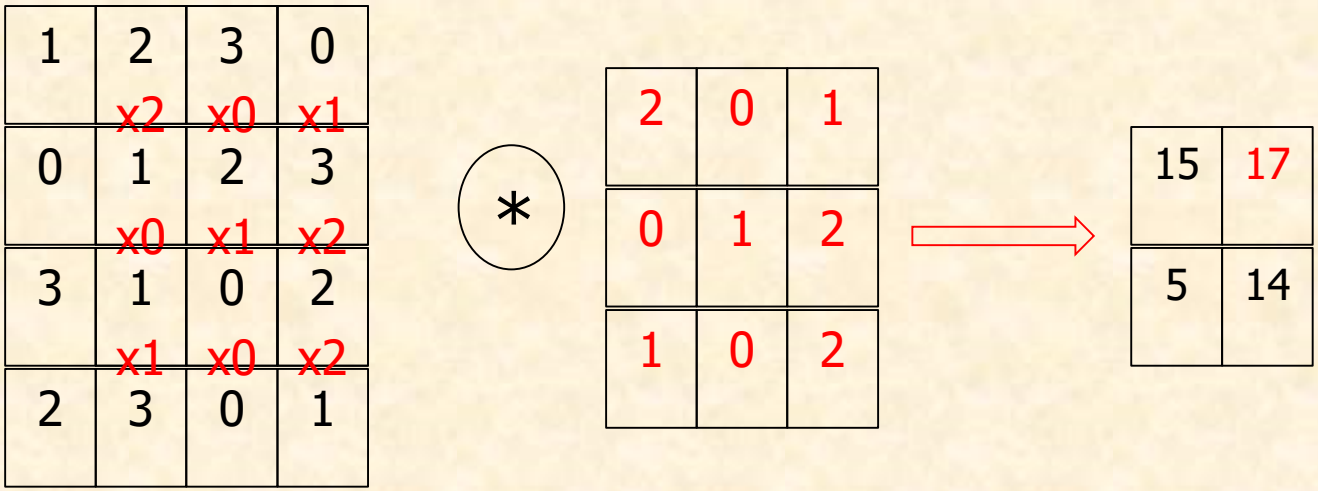
卷積運算

- 在卷積層執行的處理叫卷積運算
- 卷積運算是一種濾鏡(Filter)運算



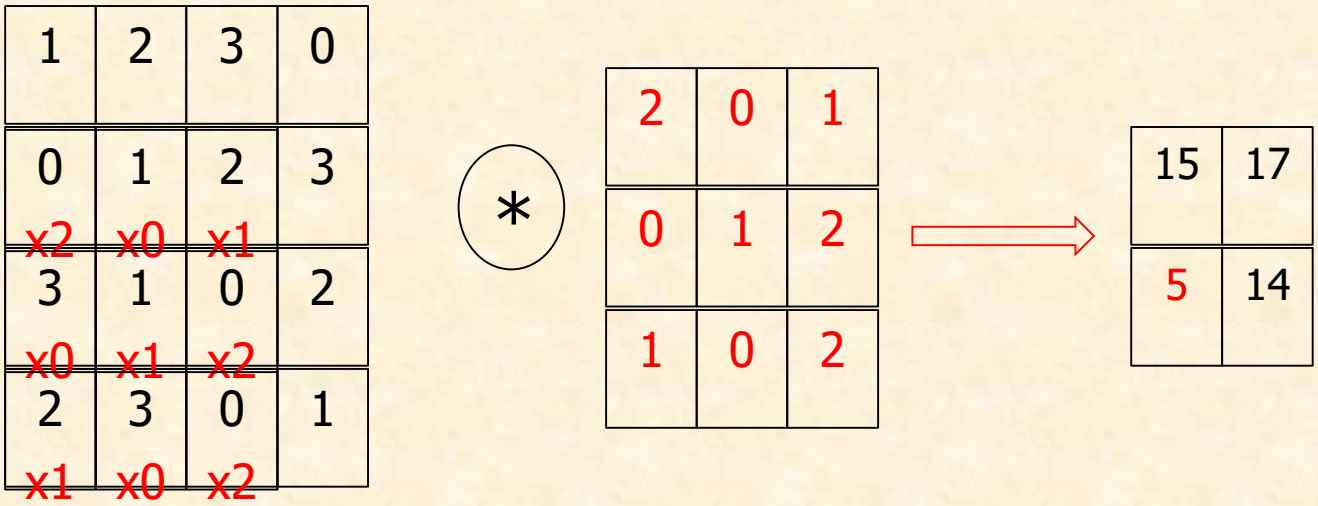
卷積運算

- 在卷積層執行的處理叫卷積運算
- 卷積運算是一種濾鏡(Filter)運算



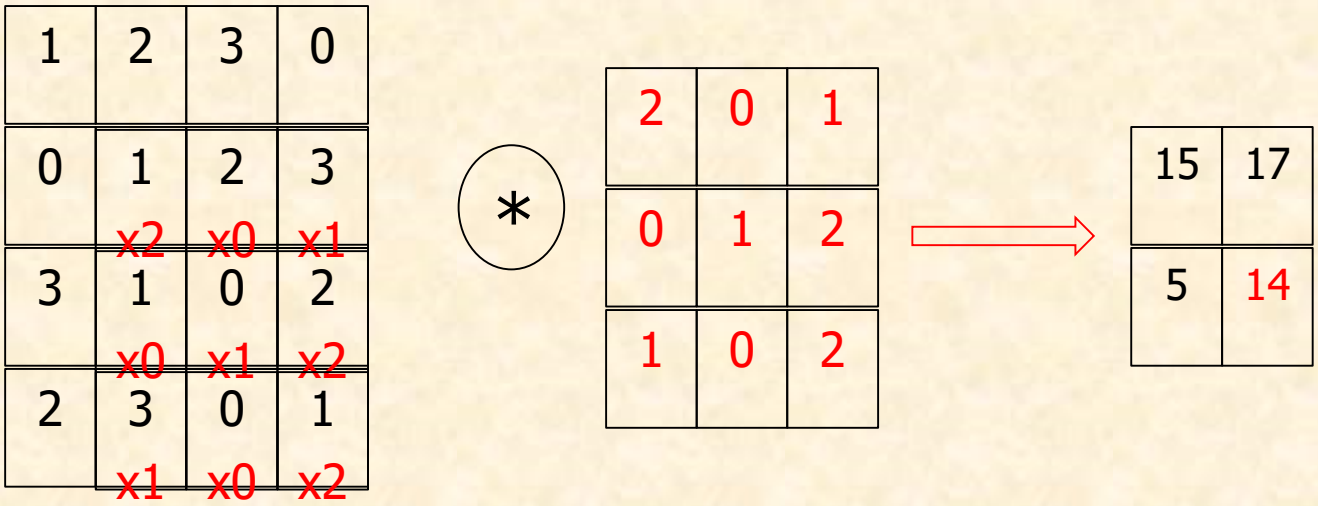
卷積運算

- 在卷積層執行的處理叫卷積運算
- 卷積運算是一種濾鏡(Filter)運算



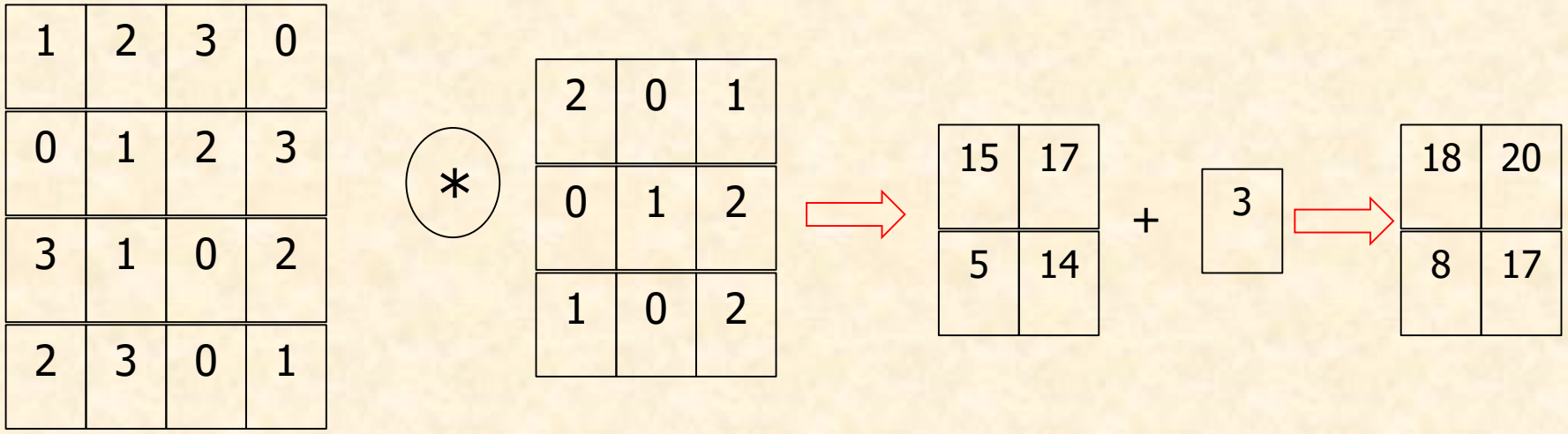
卷積運算

- 在卷積層執行的處理叫卷積運算
- 卷積運算是一種濾鏡(Filter)運算



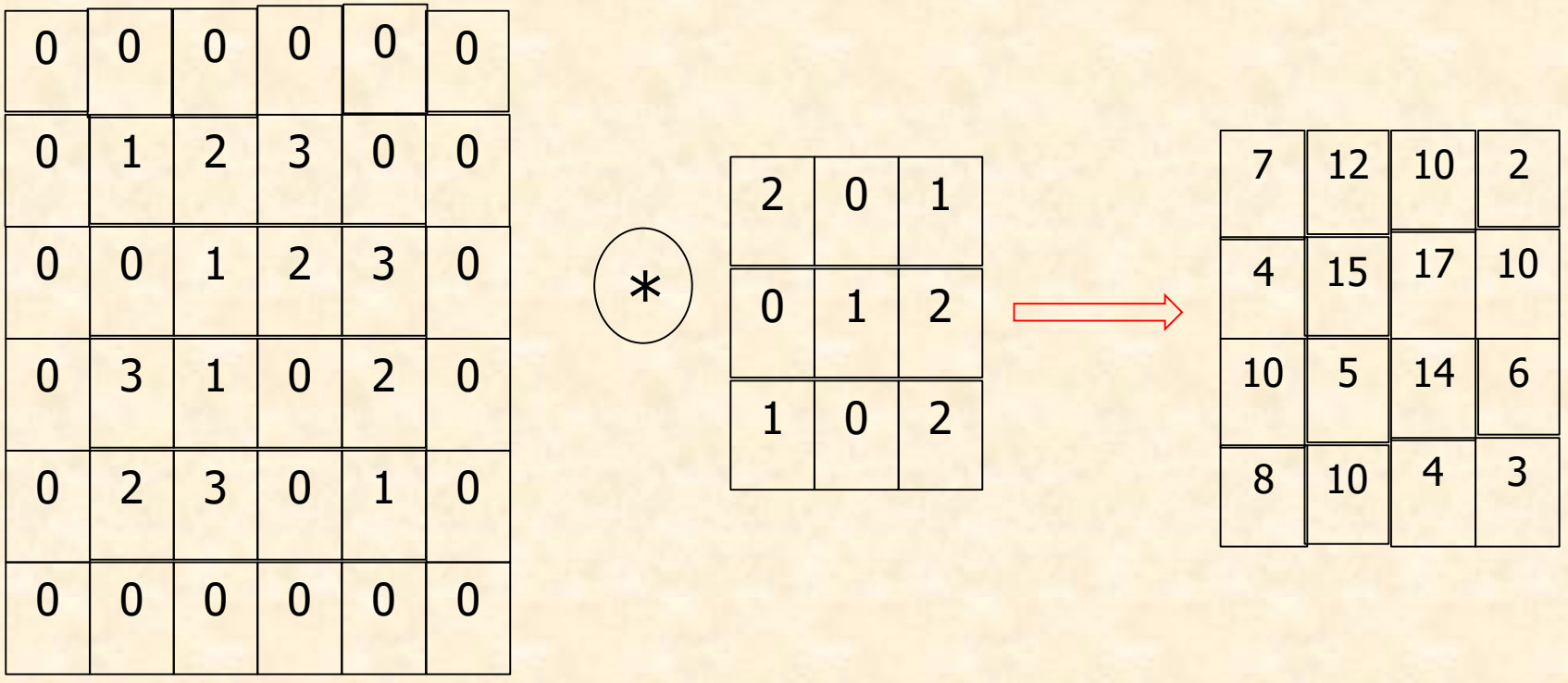
卷積運算於CNN之運作

- 在卷積層執行的處理叫卷積運算
- 卷積運算是一種濾鏡(Filter)運算



輸入資料　卷積運算　濾鏡(權重)　偏權值　神經元輸出資料

填補(Padding)



(4,4)
輸入資料(Padding: 1)

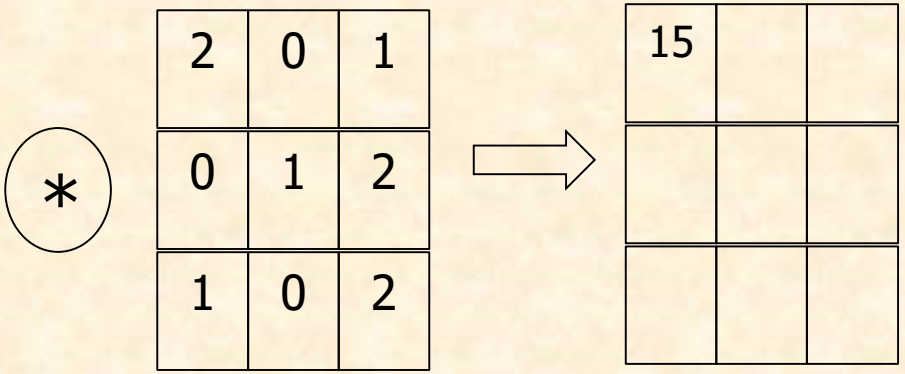
(3, 3)
濾鏡(核，kernel)

(4,4)
輸出資料

步幅(Stride)

- 套用濾鏡的位置間隔稱作步幅
- Stride =2 的例子

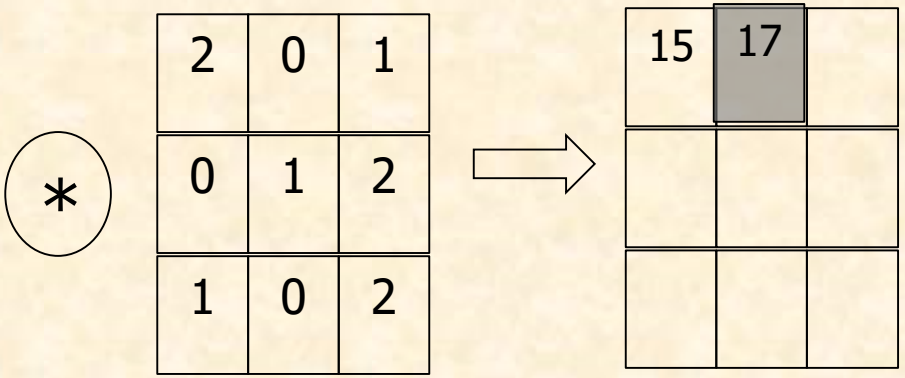
1	2	3	0	1	2	3
0	1	2	3	0	1	2
3	0	1	2	3	0	1
2	3	0	1	2	3	0
1	2	3	0	1	2	3
0	1	2	3	0	1	2
3	0	1	2	3	0	1



步幅(Stride)

- 套用濾鏡的位置間隔稱作步幅
- Stride =2 的例子

1	2	3	0	1	2	3
0	1	2	3	0	1	2
3	0	1	2	3	0	1
2	3	0	1	2	3	0
1	2	3	0	1	2	3
0	1	2	3	0	1	2
3	0	1	2	3	0	1



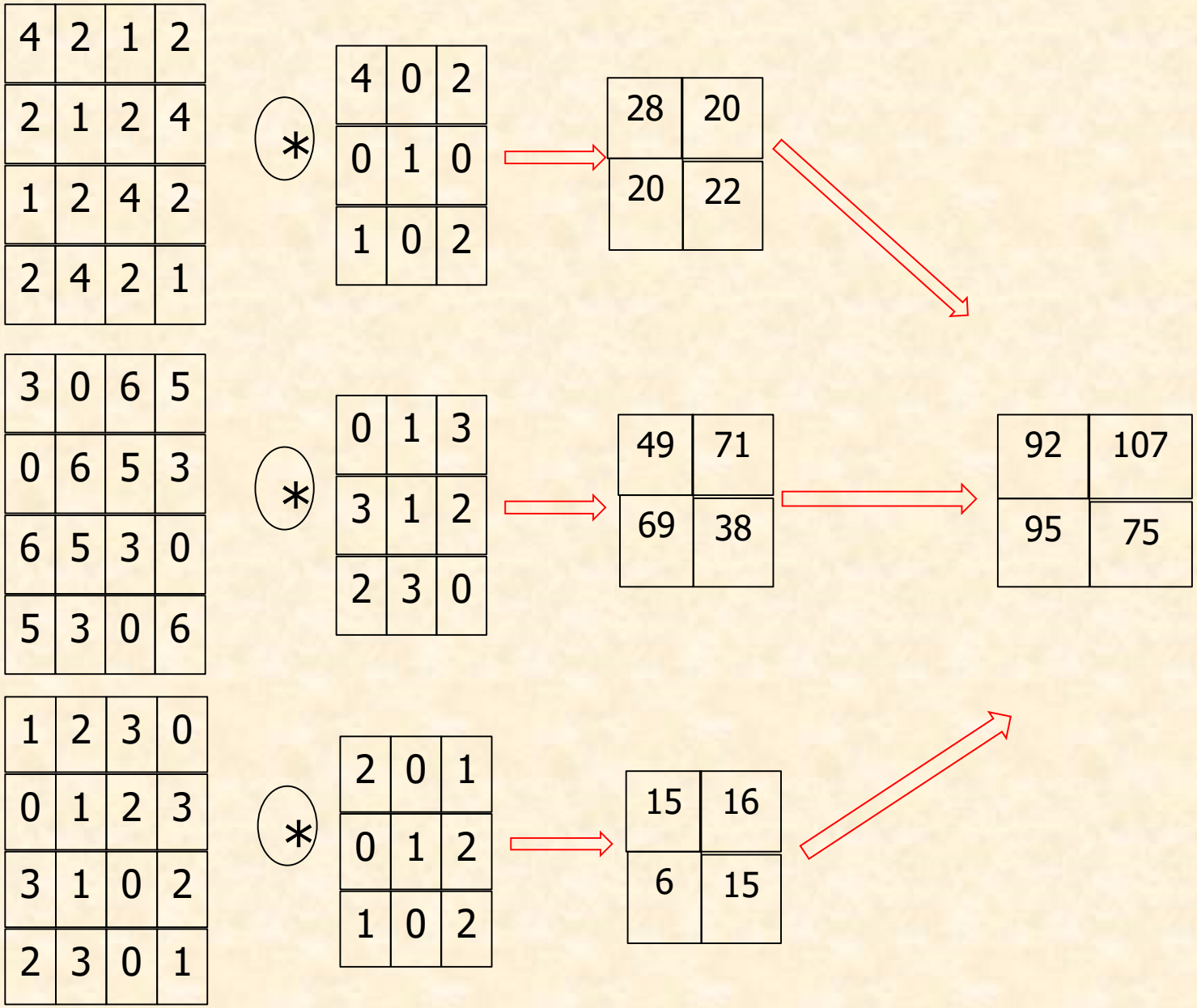
輸出資料的大小

- 令輸入資料大小為(*H, W*)，濾鏡大小為(*FH,FW*)，Padding = *P*，Stride = *S*
- 則輸出資料的大小(*OH, OW*)

$$OH = \frac{H + 2P - FH}{S} + 1$$

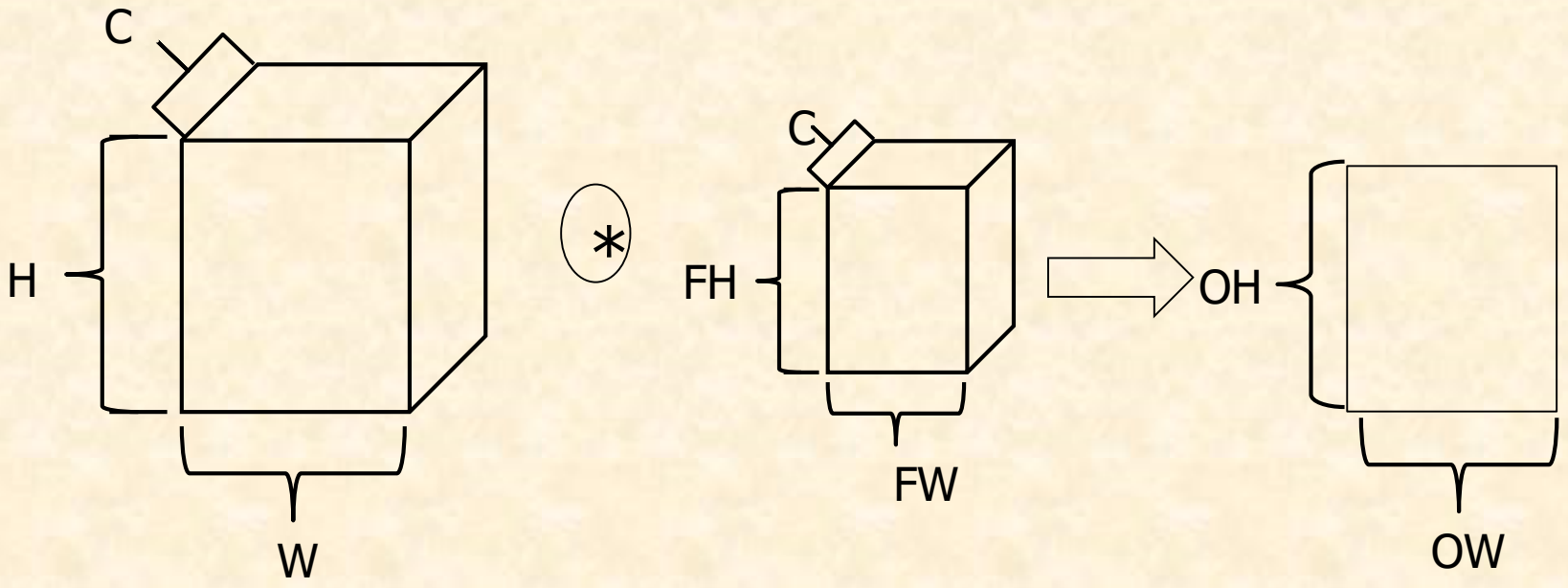
$$OW = \frac{W + 2P - FW}{S} + 1$$

三維資料的卷積運算


4 2 1 2
2 1 2 4
1 2 4 2
2 4 2 1
*
4 0 2
0 1 0
1 0 2
28 20
20 22
3 0 6 5
0 6 5 3
6 5 3 0
5 3 0 6
*
0 1 3
3 1 2
2 3 0
49 71
69 38
92 107
95 75
1 2 3 0
0 1 2 3
3 1 0 2
2 3 0 1
*
2 0 1
0 1 2
1 0 2
15 16
6 15

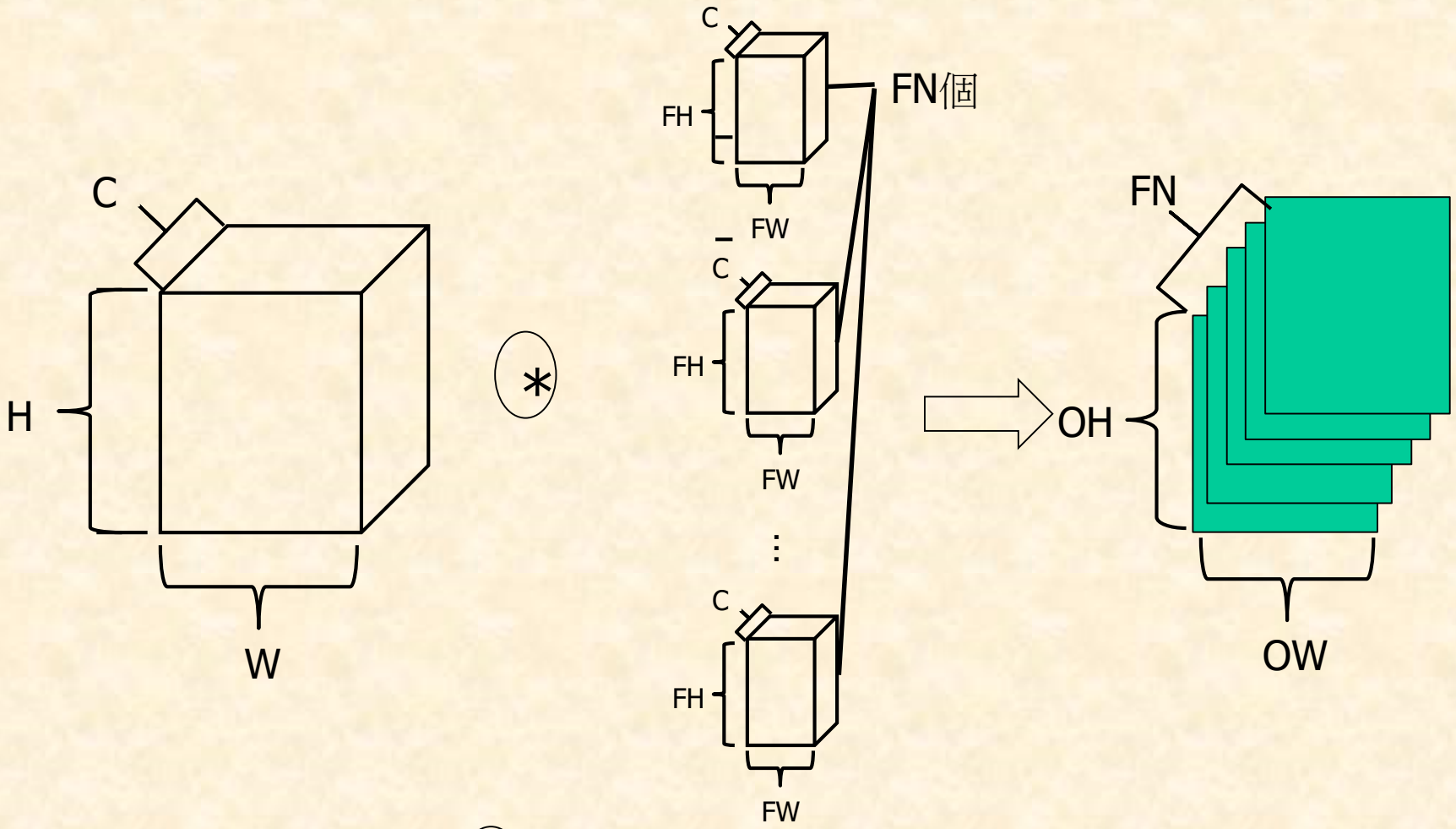
用區塊來思考

- 三維卷積運算中的資料或濾鏡想像成三維立方體區塊，比較容易思考。



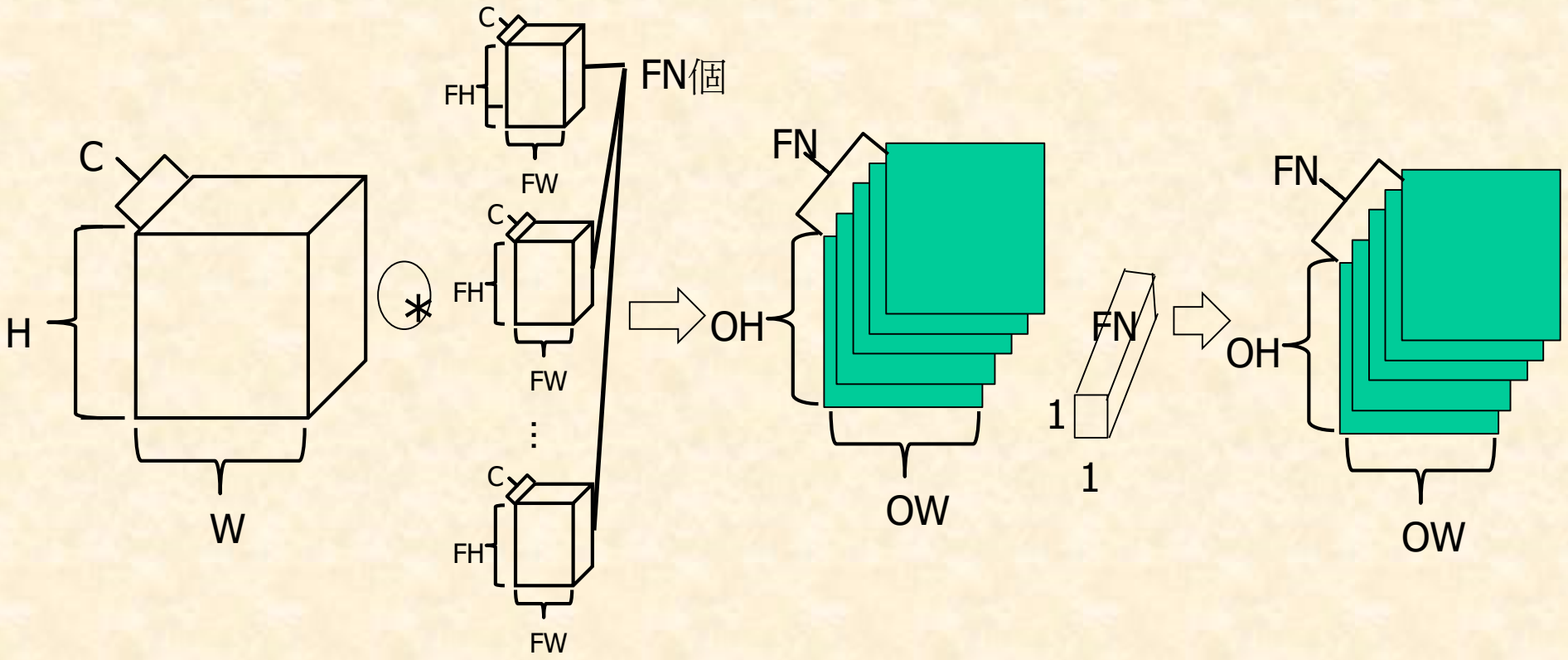
多個濾鏡的卷積運算

- 三維卷積運算中的資料或濾鏡想像成三維立方體區塊，比較容易思考。



多個濾鏡、加上偏權值的卷積運算

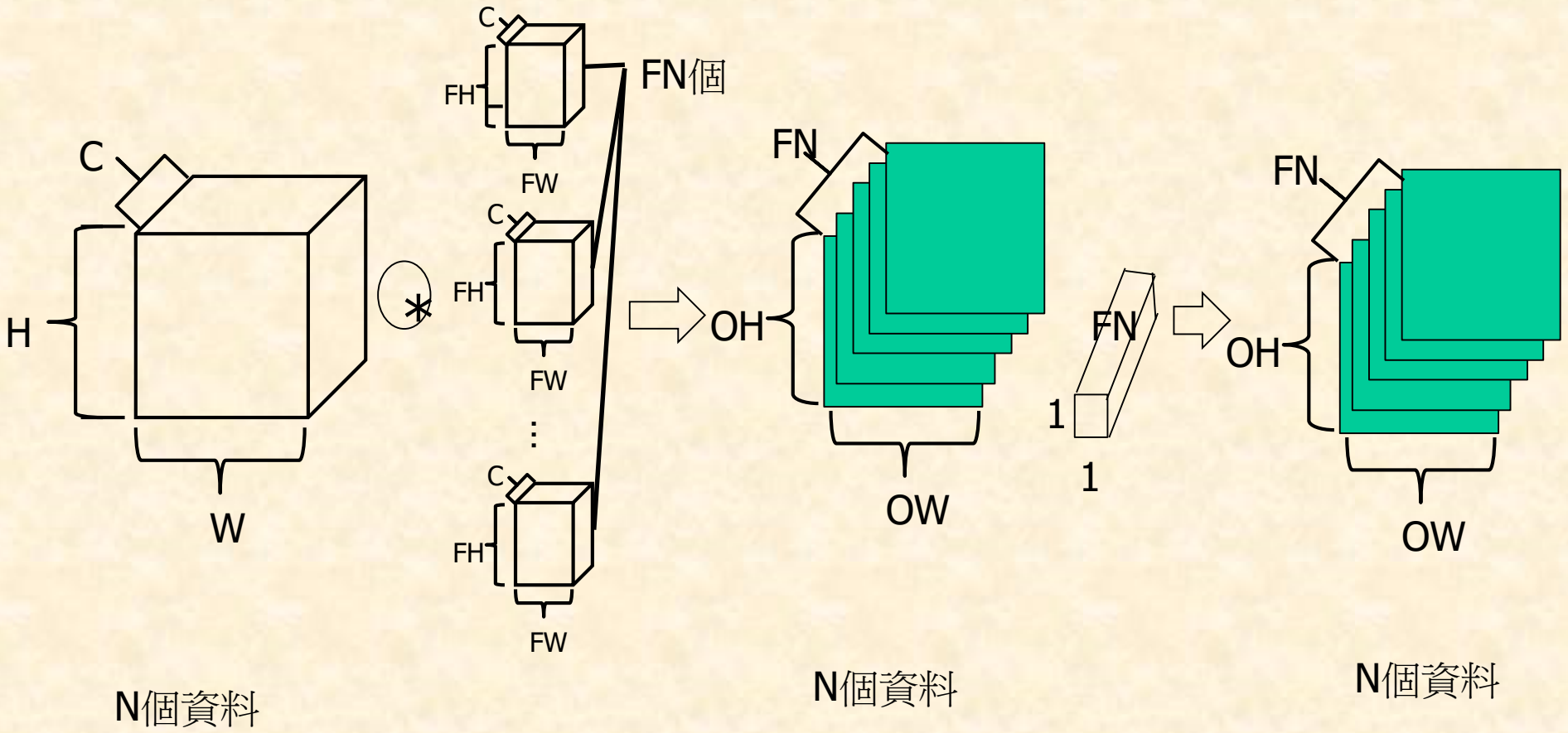
- 三維卷積運算中的資料或濾鏡想像成三維立方體區塊，比較容易思考。



(C, H, W) 輸入資料 ∗ (FN, C, FH, FW) 濾鏡 ⇨ (FN, OH, OW) + (FN, 1, 1) 偏權值 ⇨ (FN, OH, OW) 輸入資料

多個濾鏡、加上偏權值及批次處理的卷積運算

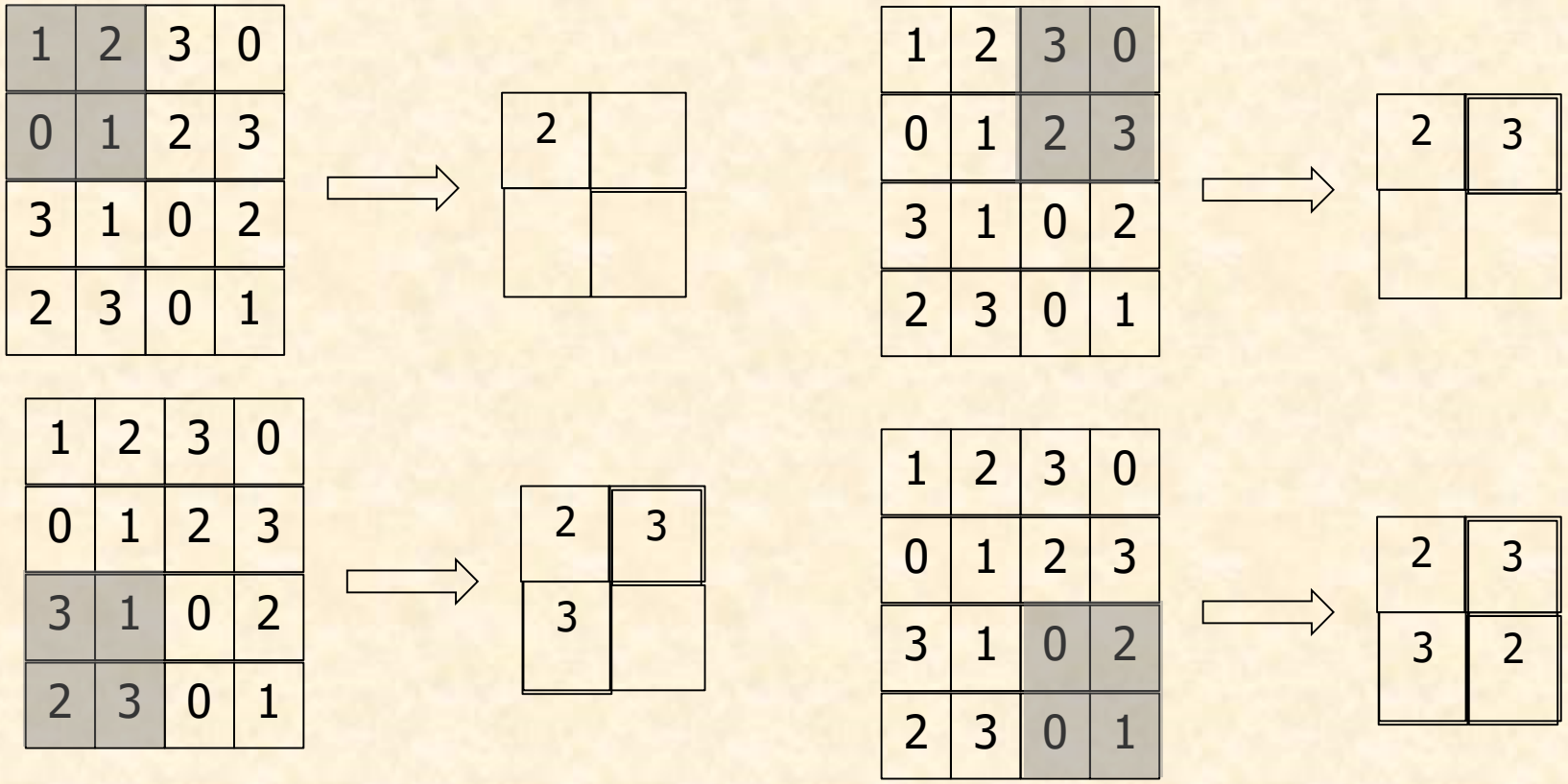
- 三維卷積運算中的資料或濾鏡想像成三維立方體區塊，比較容易思考。



(N, C, H, W) 輸入資料 * (FN, C, FH, FW) 濾鏡 ⇨ (N,FN,OH,OW) + (FN, 1, 1) 偏權值 ⇨ (N,FN, OH, OW) 輸入資料

池化層(Pooling)

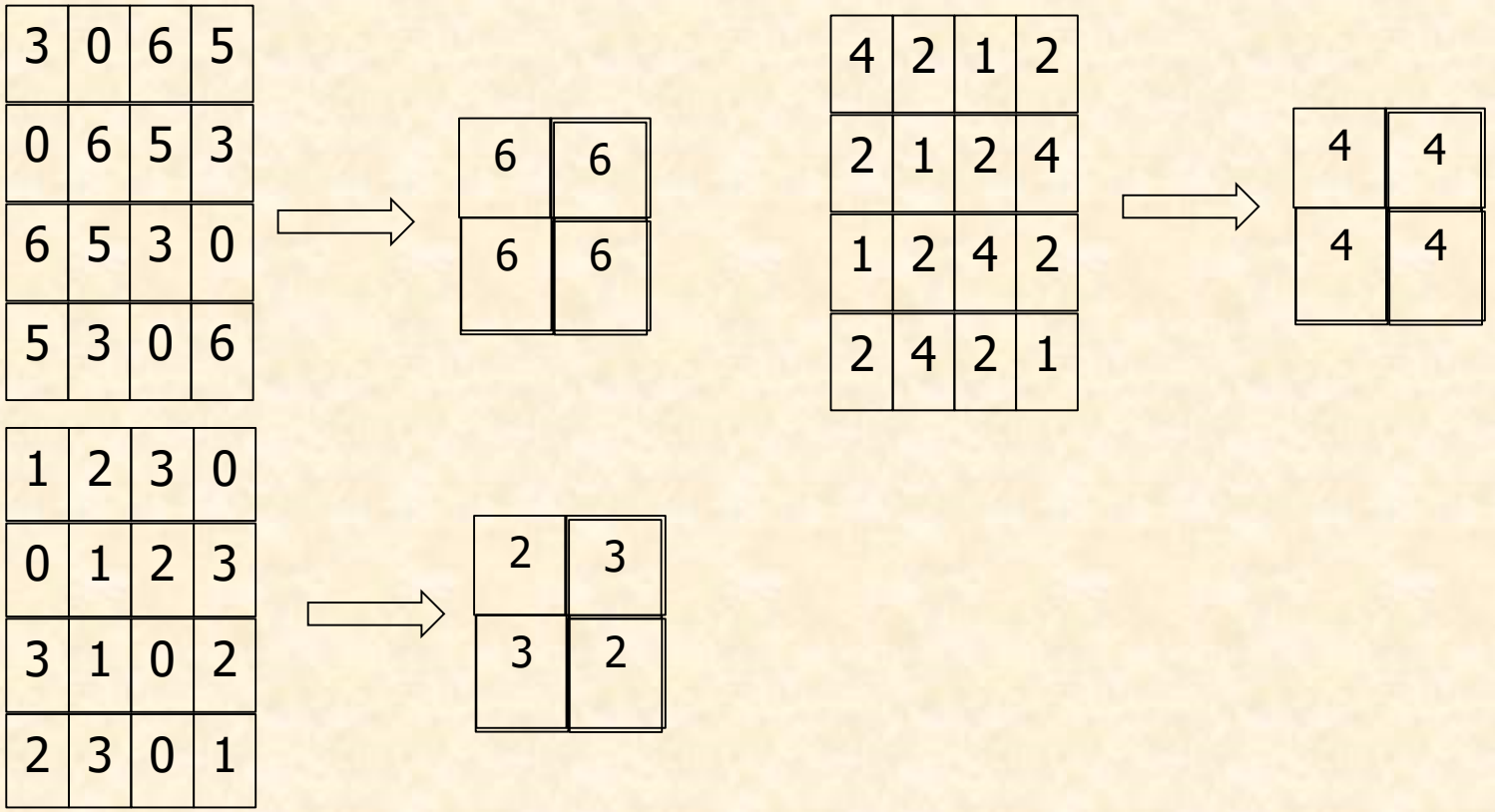
- 池化層是縮小垂直、水平空間的運算。
- 以下是最大化池化層(Max Pooling)之範例



也可使用其他的池化層函數，例如取平均值、取中值等

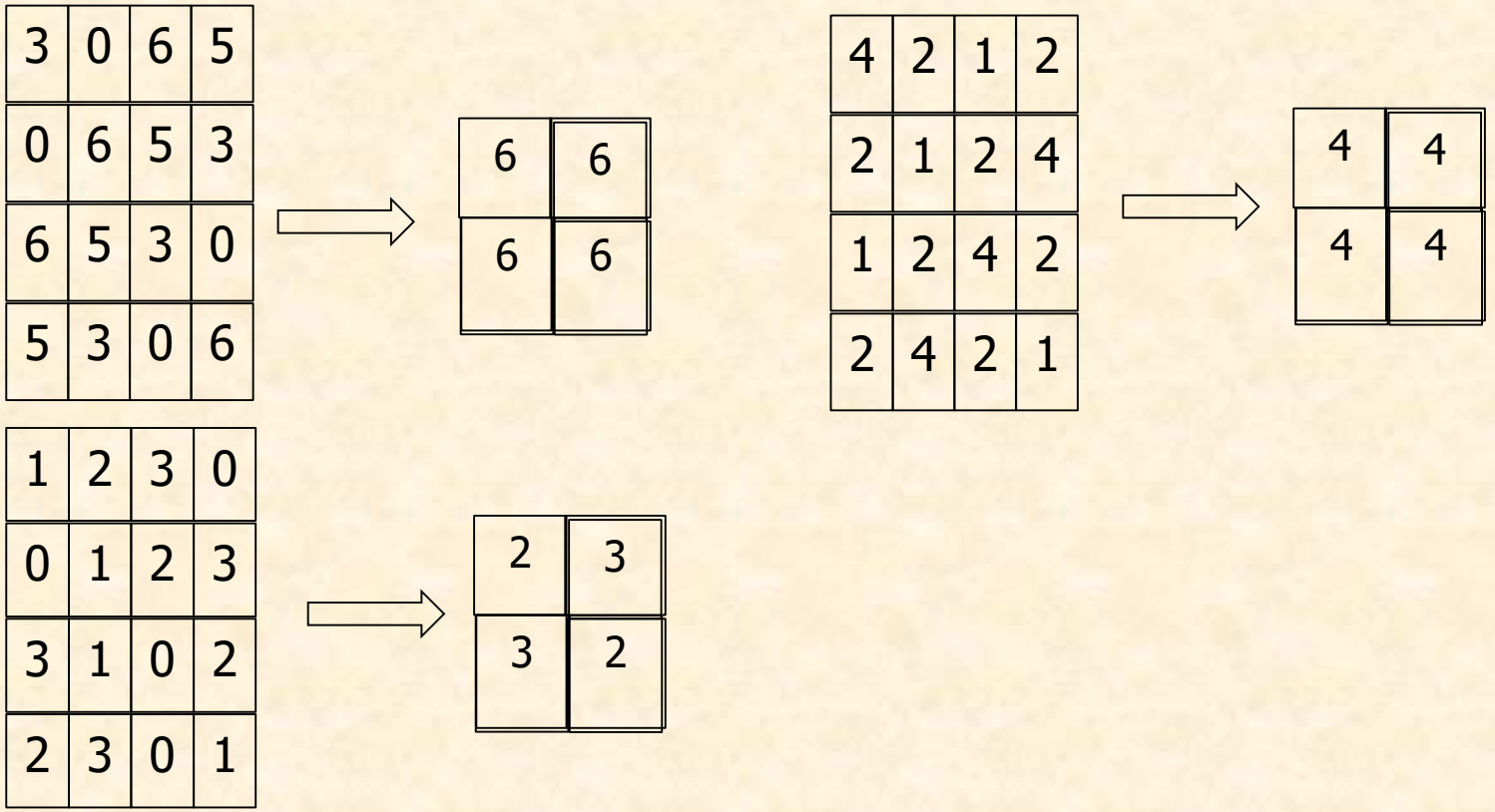
池化層的特色

- 沒有學習參數(使用制式函數)
- 色板數量不變



池化層的特色

- 對微小變化很穩健(Robust)
  - 區域內之微小變化值會被忽略。



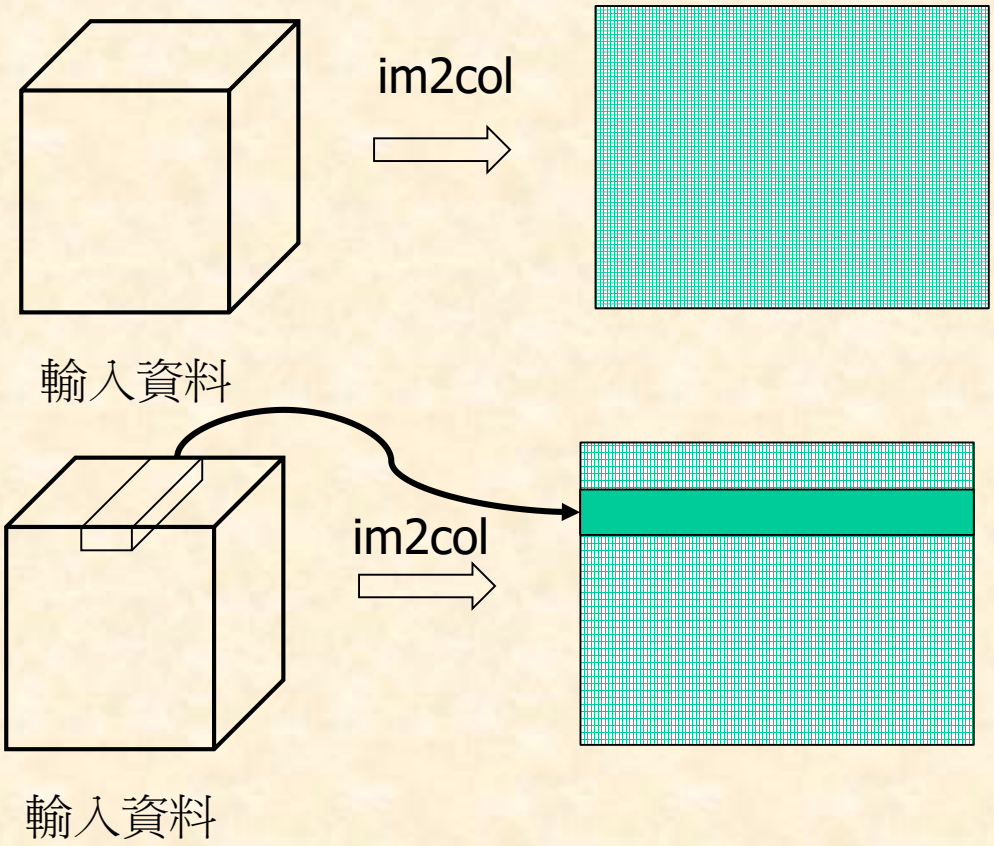
Python執行捲積層/池化層

- 製作捲積層需要用到4維陣列
  - CNN各層流動的資料是四維陣列，例如(10, 1, 28, 28)代表10個高度28、寬度28、1個色板的輸入影像
  - Python Code

```
>>> x = np.random.rand(10, 1, 28, 28)    #隨機產生資料
>>> x.shape
(10, 1, 28, 28)
>>> x[0].shape   #存取第一筆資料
(1, 28, 28)
>>>x[0, 0].shape  #存取第一筆資料的第一個色板
(28, 28)
```

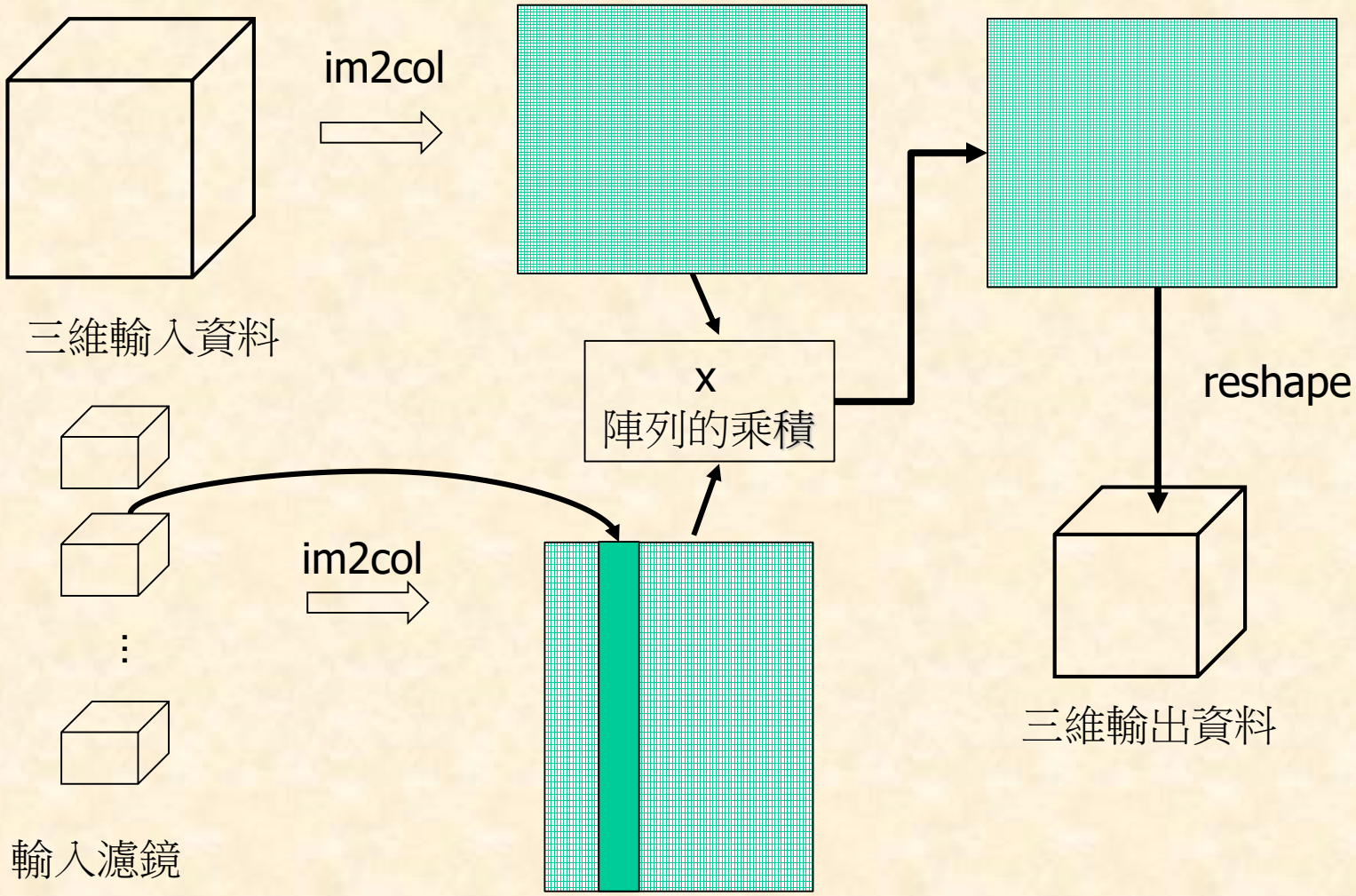
Python執行捲積層/池化層

- 利用內建函數im2col製作捲積層處理需要用到4維陣列
- im2col 是可以針對套用的濾鏡(權重)輕鬆展開資料的函數。
- im2col 可以針對套用的濾鏡的位子，輕鬆展開資料的函數。



Python執行捲積層/池化層

- 利用im2col展開輸入資料，之後只要把捲積濾鏡(權重)展開成一行，計算兩個陣列的乘積即可。

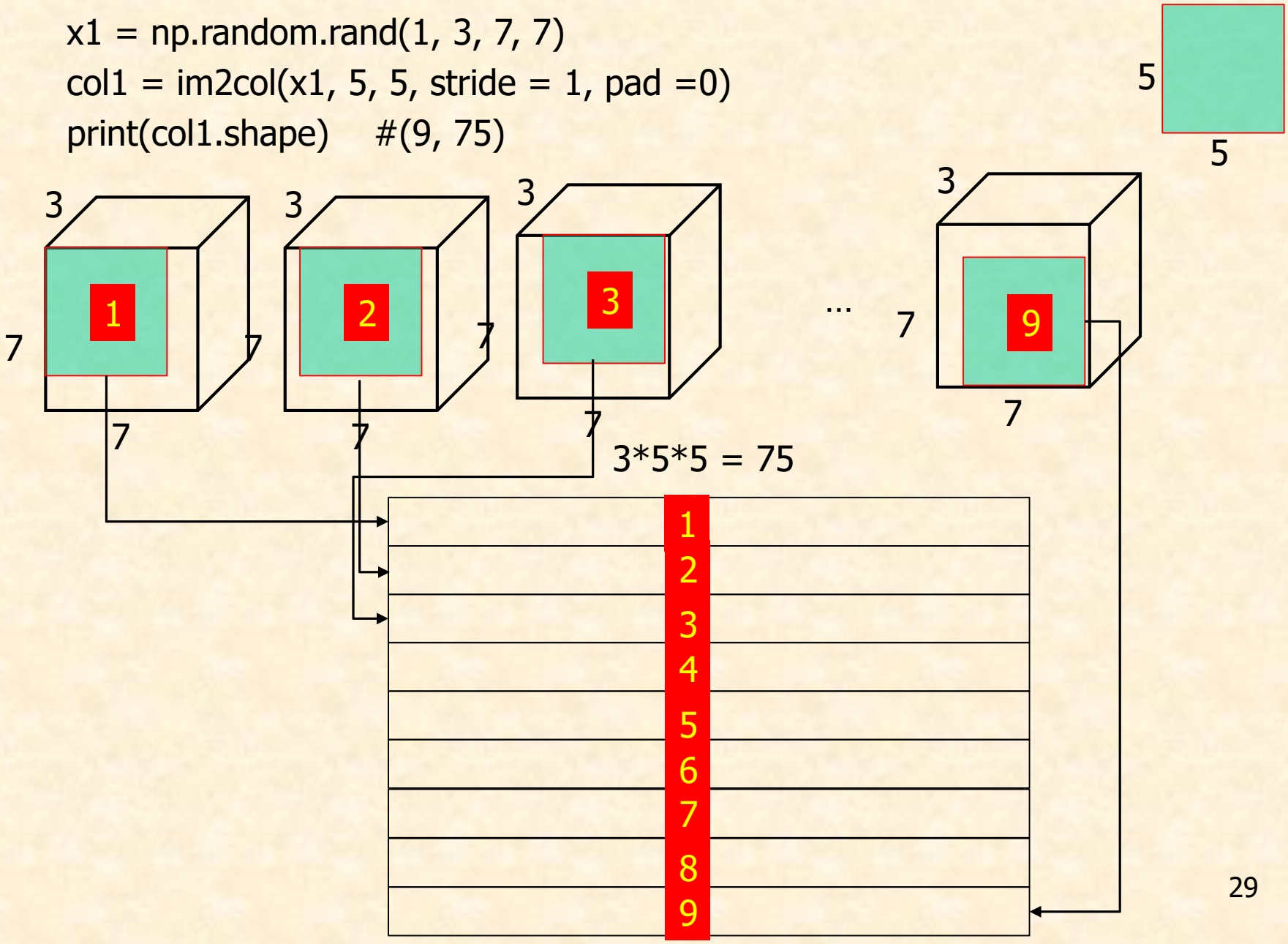


im2col Code

```
def im2col(input_data, filter_h, filter_w, stride=1, pad=0):
    """
    Parameters
    ----------
    input_data : (批次大小, 色板個數, 高度, 寬度)的4維振烈
    filter_h : 濾鏡的高度
    filter_w :濾鏡的寬度
    stride : 步輻
    pad : padding數
    Returns
    -------
    col : 2維陣列
    """
    N, C, H, W = input_data.shape
    out_h = (H + 2*pad - filter_h)//stride + 1
    out_w = (W + 2*pad - filter_w)//stride + 1
    img = np.pad(input_data, [(0,0), (0,0), (pad, pad), (pad, pad)], 'constant')
    col = np.zeros((N, C, filter_h, filter_w, out_h, out_w))
    for y in range(filter_h):
        y_max = y + stride*out_h
        for x in range(filter_w):
            x_max = x + stride*out_w
            col[:, :, y, x, :, :] = img[:, :, y:y_max:stride, x:x_max:stride]
    col = col.transpose(0, 4, 5, 1, 2, 3).reshape(N*out_h*out_w, -1)
```

im2col 函數

```
x1 = np.random.rand(1, 3, 7, 7)
col1 = im2col(x1, 5, 5, stride = 1, pad =0)
print(col1.shape)    #(9, 75)
```



使用im2col執行捲積層

```
class Convolution:
    def __init__(self, W, b, stride=1, pad=0):
        self.W = W
        self.b = b
        self.stride = stride
        self.pad = pad
    def forward(self, x):
        FN, C, FH, FW = self.W.shape
        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)
        col_W = self.W.reshape(FN, -1).T
        out = np.dot(col, col_W) + self.b
        out = out.reshape(N, out_h, out_w, -1).transpose(0, 3, 1, 2)
        return out
```

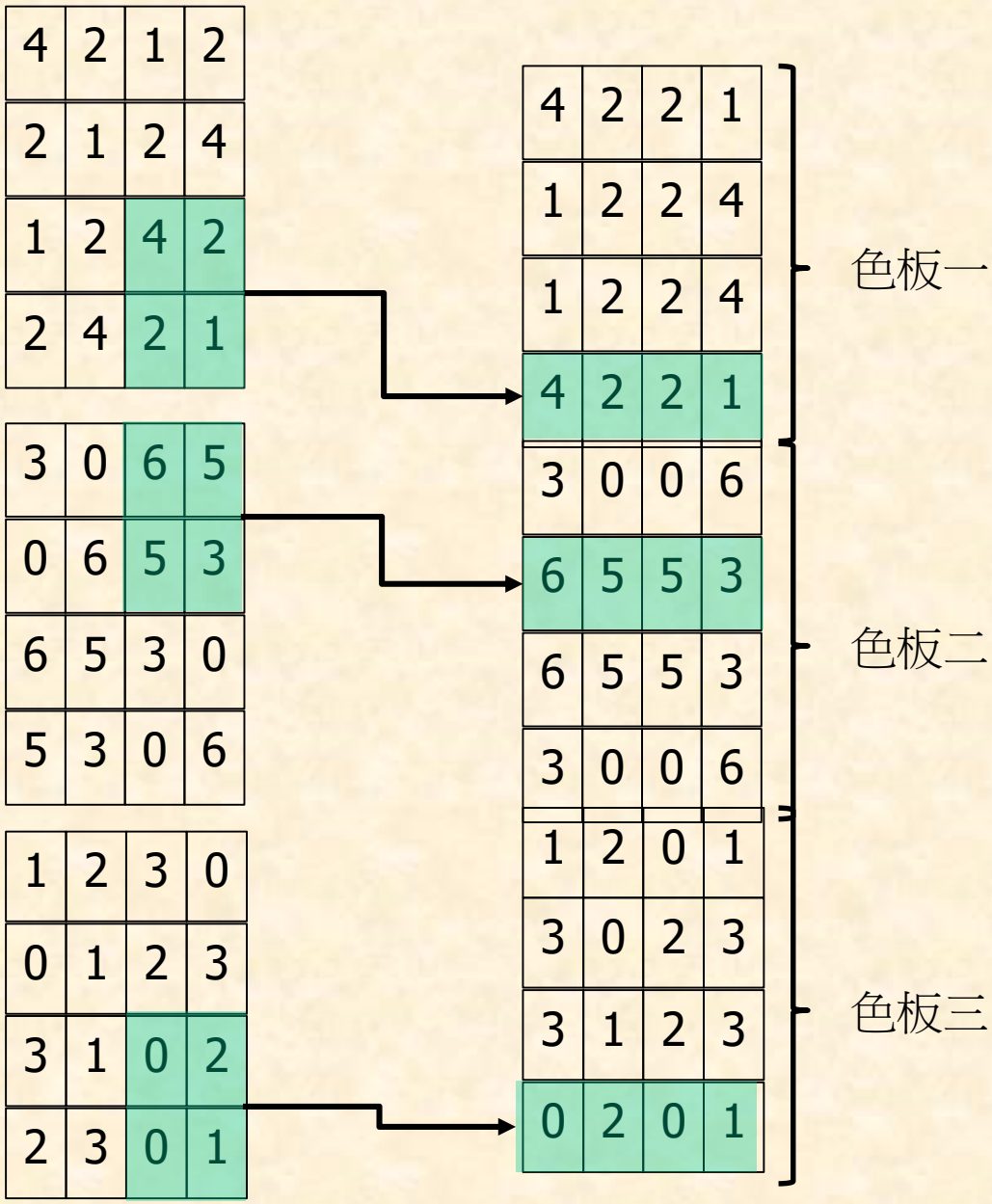
transpose(0, 3, 1, 2)

形狀 (N, H, W, C) ⟹ 形狀 (N, C, H, W)

索引值 0, 1, 2, 3 0, 3, 1, 2

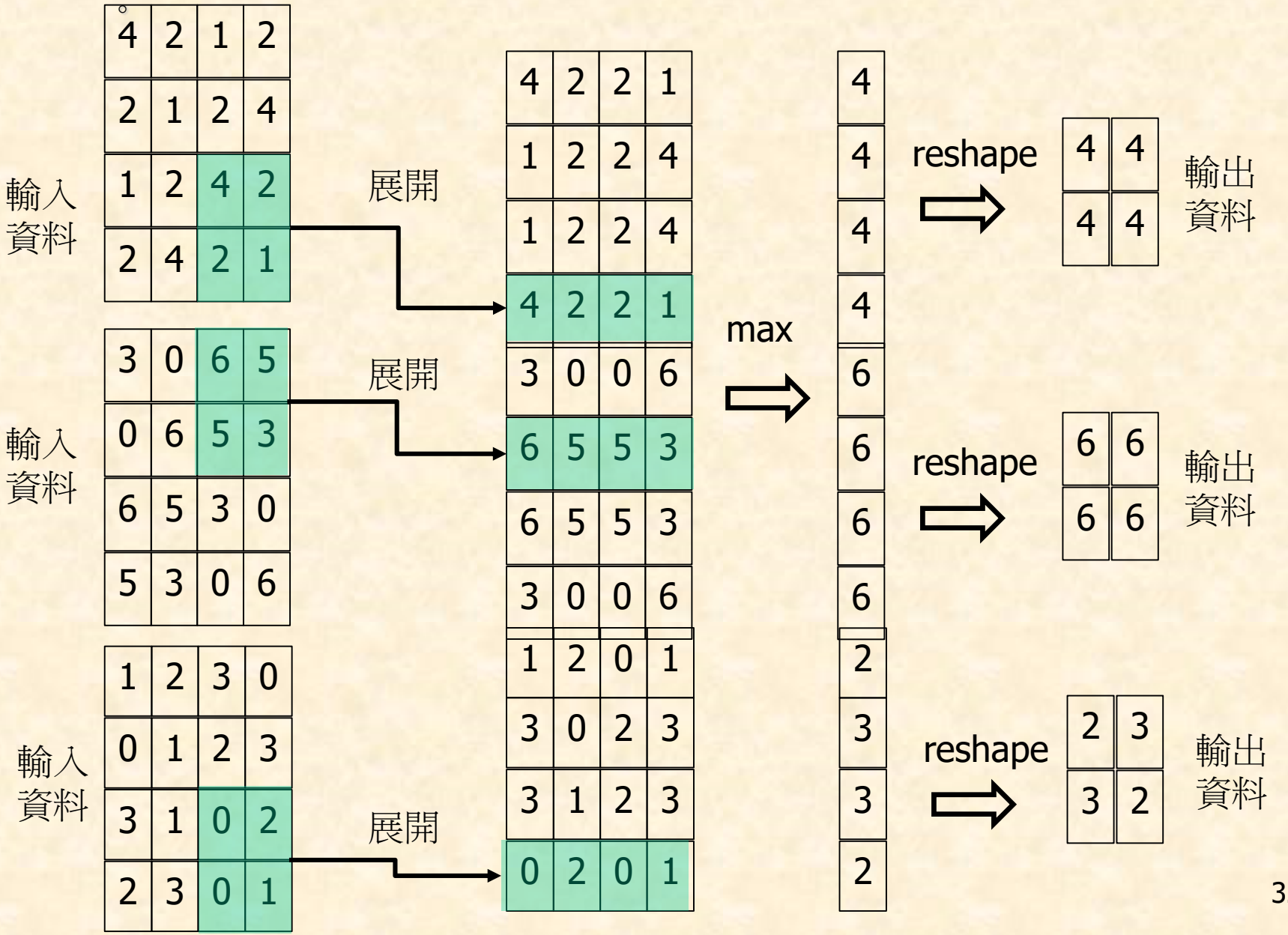
執行池化層

- 池化層的執行過程跟卷積層一樣，利用im2col展開輸入資料，但是池化層各色板的運算是彼此獨立，這點跟卷積層不一樣。



執行池化層

- 展開後，針對各列進行運算



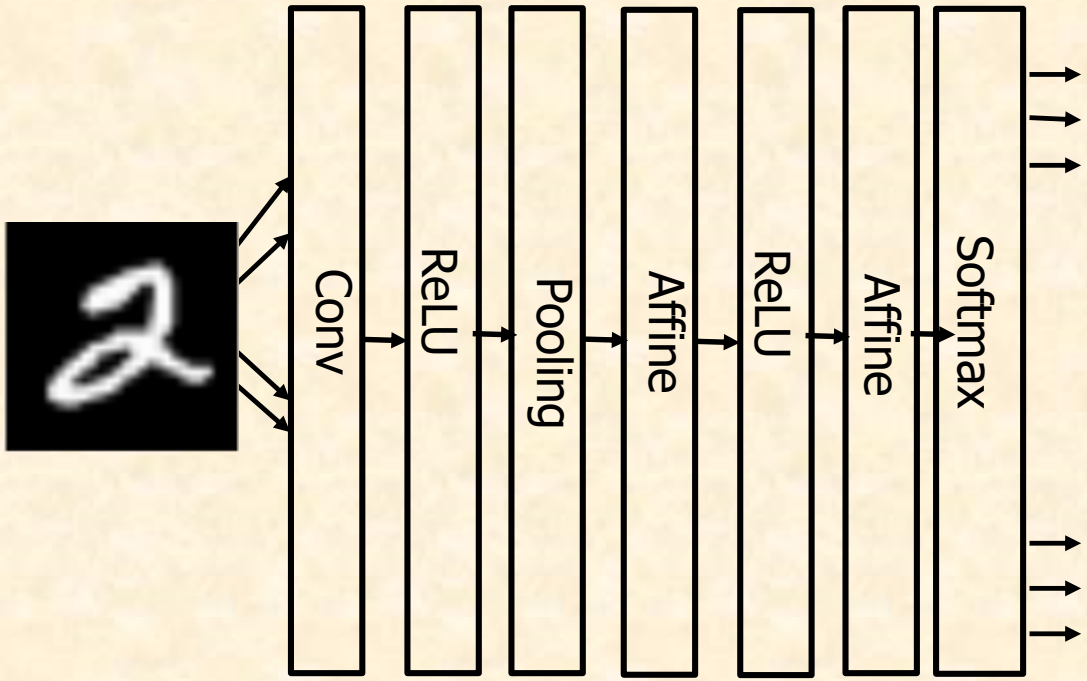
Python執行池化層

```
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
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)
        # 展開(1)
        col = im2col(x, self.pool_h, self.pool_w, self.stride, self.pad)
        col = col.reshape(-1, self.pool_h*self.pool_w)
        # 最大值 (2)
        arg_max = np.argmax(col, axis=1)
        out = np.max(col, axis=1)
        # 調整形狀 (3)
        out = out.reshape(N, out_h, out_w, C).transpose(0, 3, 1, 2)
        return out
```

Python執行CNN

- 以管線作業方式組合各層的功能，就可完成CNN之基本執行工作。

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



Python執行CNN

- 單存的CNN引數(參數)

input_dim：輸入資料的(色板、高度、寬度)維度

conv_param: 卷積層的超參數(字典)。Key 如下

--- filter_num: 濾鏡的數量

--- filter_size: 濾鏡的大小

--- stride: 步輻

--- pad: 填補

hidden_size：隱藏層(全連接)的神經元個數

output_size：輸出層(全連接)的神經元個數（MNIST的例子為10）

activation : 'relu' or 'sigmoid'

weight_init_std：初始化時的權重標準偏差的指定（e.g. 0.01）

--- ‘relu‘時採用「He的初期值」設定

--- ‘sigmoid’時採用「Xavier的初期值」設定

SimpleConvNet的初始化

```
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))
```

SimpleConvNet的初始化

```
# 權重的初始化
    self.params = {}
    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)
    self.params['W2'] = weight_init_std * \
                        np.random.randn(pool_output_size, hidden_size)
    self.params['b2'] = np.zeros(hidden_size)
    self.params['W3'] = weight_init_std * \
                        np.random.randn(hidden_size, output_size)
    self.params['b3'] = np.zeros(output_size)
```

SimpleConvNet的初始化

```
# 產生必要層
        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()
```

SimpleConvNet的predict及損失函數計算

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

        return x

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

SimpleConvNet的誤差反向傳播法及梯度計算

```
def gradient(self, x, t):
        # 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
```

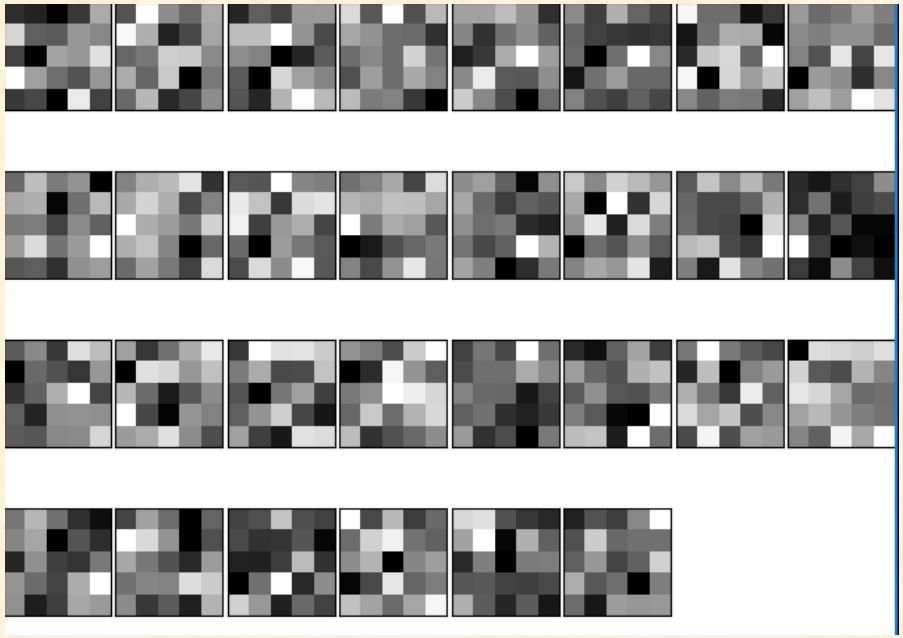
SimpleConvNet的執行結果

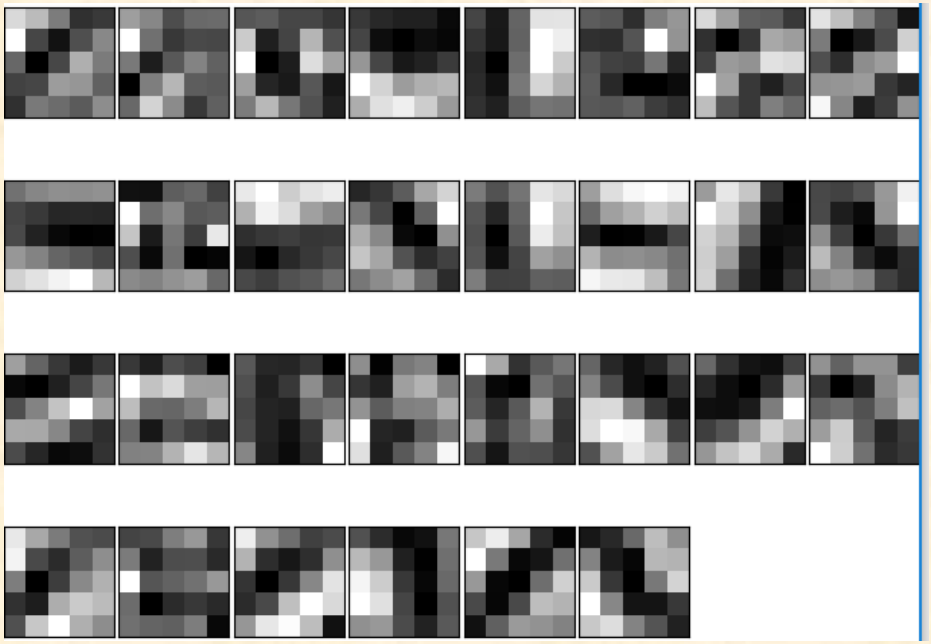
- 執行前必須先行訓練(請參考ch07/train_convent.py)
- 執行結果(MNIST DATASET)
  - 訊練資料的辨識率(inside testing): 99.28%
  - 測試資料的辨識率(outside testing): 98.96%

利用重疊更多層(深度學習)時，測試資料的辨識率可達99%

CNN的視覺化

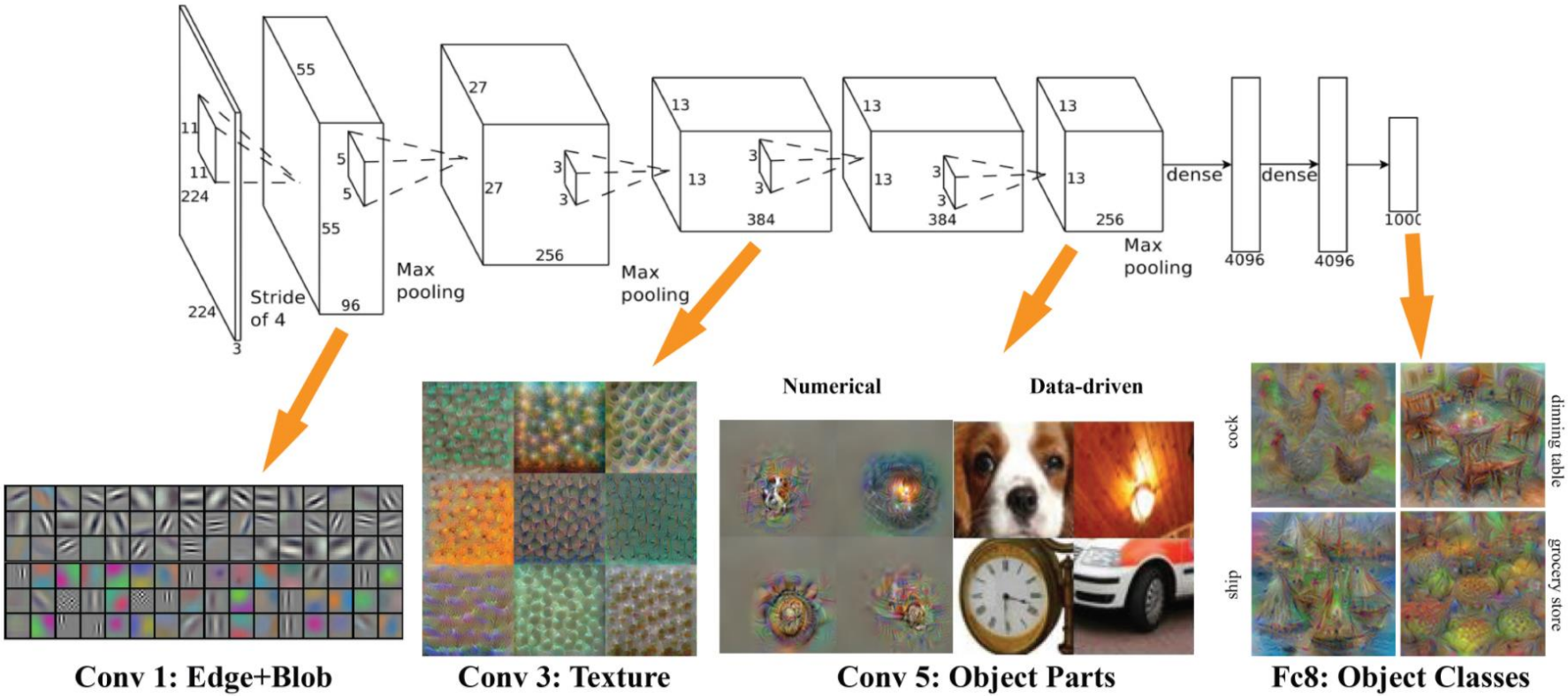
- 視覺化第一層權重
  - 形狀:(30, 1, 5, 5)





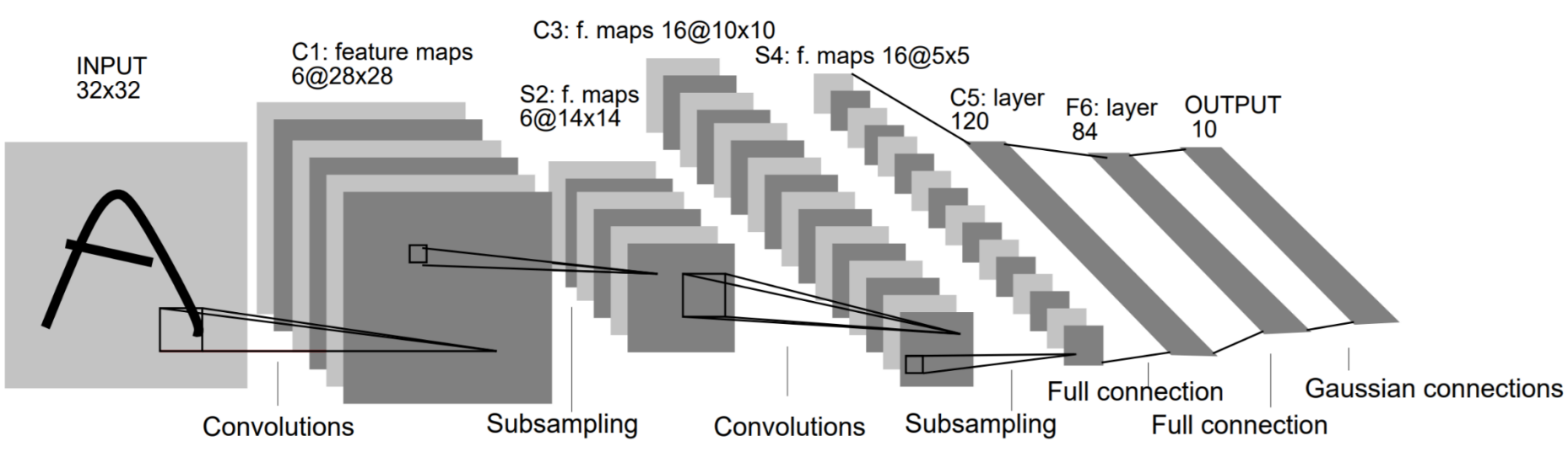
學習前: 隨意初始化濾鏡　　學習後:擷取輸入影像的各種初階結構(線邊)特徵

AlexNet的各層視覺化結果



CNN的鼻祖

- CNN的鼻祖
  - LeNet (1998)



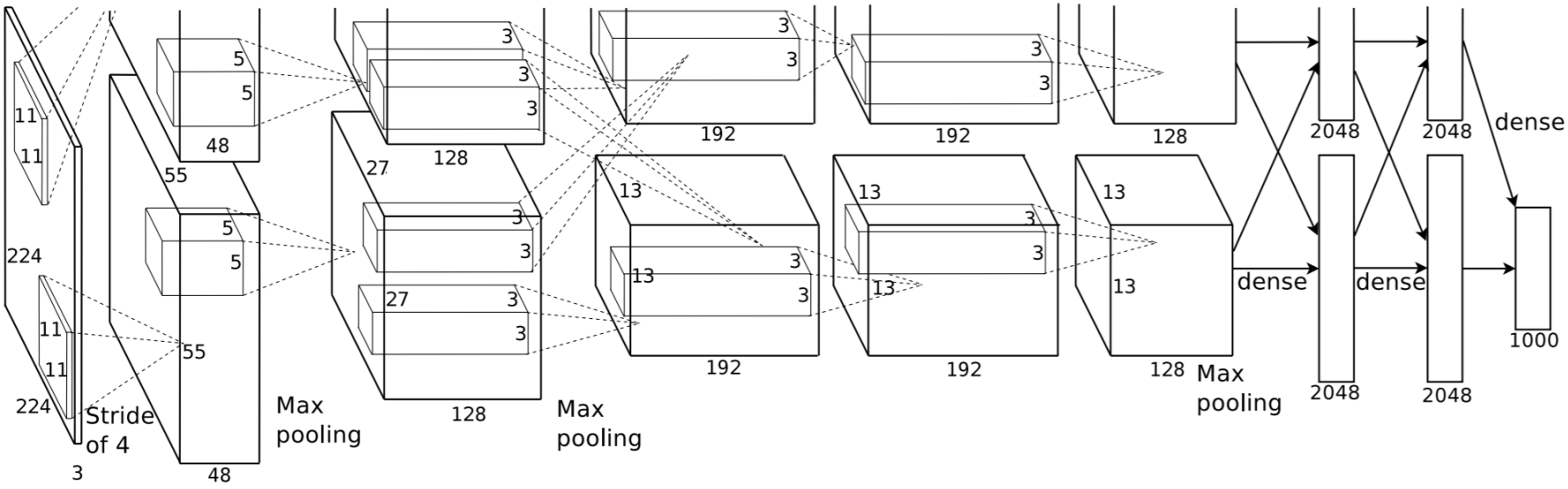
PROC. OF THE IEEE, NOVEMBER 1998

Gradient-Based Learning Applied to Document Recognition

Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner

現代版的CNN

- AlexNet (2012)



ImageNet Classification with Deep Convolutional Neural Networks

Alex Krizhevsky
University of Toronto
kriz@cs.utoronto.ca

Ilya Sutskever
University of Toronto
ilya@cs.utoronto.ca

Geoffrey E. Hinton
University of Toronto
hinton@cs.utoronto.ca

LeNet vs. AlexNet

- 結構相似
- AlexNet做了哪些改善
  - 使用ReLu函數
  - 使用LRN(Local Response Normalization)局部性正規化層。
  - 使用Dropout
- 20年前LeNet無GPU及雲端系統可供平行處理，因學習速度太慢，無法獲得該有的重視。

Any Questions?