

# 卷積神經網路

本章的主題是卷積神經網路（convolutional neural network：CNN）。CNN 使用於影像辨識、聲音辨識等各種情況。在視覺辨識競賽中，深度學習使用的手法幾乎都是以 CNN 為基礎。本章將詳細說明 CNN 的結構，並且利用 Python 執行處理內容。

## 7.1 整體結構

我們先從 CNN 的網路結構開始介紹，讓你瞭解 CNN 的概要。CNN 和前面介紹過的神經網路一樣，就像樂高，可以利用組合各層的方式來建構。但是，在 CNN 中，還出現了新的「Convolution 層（卷積層）」與「Pooling 層（池化層）」。下一節會詳細說明卷積層與池化層，這裡先說明如何組合各層來建構 CNN。

前面說明的神經網路是，相鄰各層的所有神經元之間彼此相連，稱作全連接（*fully-connected*），我們以 Affine 層的名稱，執行過全連接層。利用 Affine 層，可以建構出 5 層全連接的神經網路，如圖 7-1 所示。

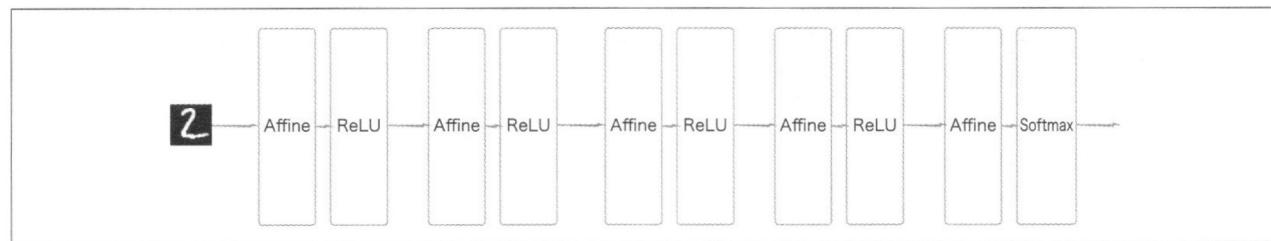


圖 7-1 以全連接層（Affine 層）建構的網路範例

圖 7-1 全連接的神經網路是在 Affine 層的後方，連接活化函數的 ReLU 層（或 Sigmoid 層）。這裡重疊了 4 層「Affine - ReLU」組合，再連接第 5 層 Affine 層，最後以 Softmax 層輸出結果（機率）。

如果用 CNN 會形成何種結構呢？CNN 的範例如圖 7-2 所示。

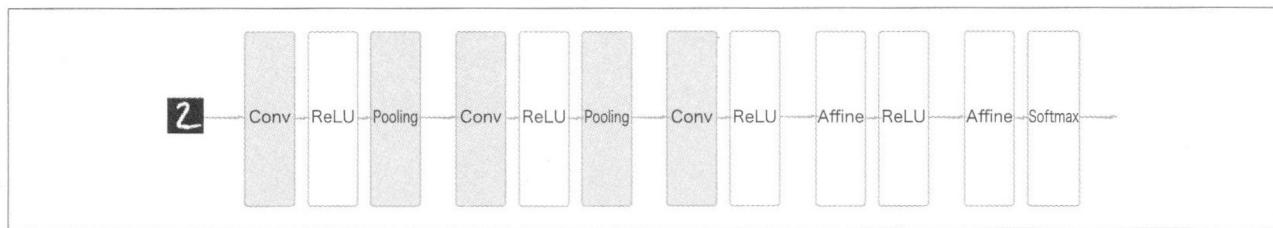


圖 7-2 以 CNN 建構的網路範例：新加入 Convolution 層與 Pooling 層（分別用灰底矩形顯示）

如圖 7-2 所示，CNN 加入了新的「Convolution 層」與「Pooling 層」。CNN 各層的連接順序是「Convolution - ReLU - (Pooling)」（有時會省略 Pooling 層）。你可以當成是用「Convolution - ReLU - (Pooling)」取代「Affine - ReLU」。

圖 7-2 的 CNN 還有其他要注意的重點，就是在接近輸出的層級，使用了原本的「Affine - ReLU」組合。此外，在最後輸出層，使用的是「Affine - Softmax」組合，這是一般 CNN 常見的結構。

## 7.2 卷積層

在 CNN 中，出現了 padding、stride 等 CNN 專用的名詞。另外，傳遞在各層的資料變成具有形狀的資料（例如，三維資料），與之前的全連接網路不同。因此，剛開始學習 CNN 時，可能會覺得完全看不懂。所以這裡我想花一點時間，徹底介紹 CNN 使用的卷積層結構。

### 7.2.1 全連接層的問題

前面介紹的全連接神經網路，使用了全連接層（Affine 層）。全連接層連接相鄰各層的所有神經元，可以決定任意輸出數量。

全連接層的問題是什麼？就是會「忽略」資料的形狀。例如，輸入資料為影像，影像通常是含有水平、垂直、色版方向的三維形狀。可是，輸入全連接層時，三維資料必須變成平面，亦即變成一維資料。實際以前面提過的 MNIST 資料集為例，輸入影像是

(1, 28, 28)，1 色版、垂直 28 像素、水平 28 像素的形狀，全連接層卻會把這個形狀變成排成一行的 784 個資料，輸入最初的 Affine 層。

影像是三維形狀，這個形狀包含了重要的空間資料。例如，類似的空間有著相似的像素值，RGB 各色版之間，具有緊密連接的關聯性，距離較遠的像素彼此沒有關係等，在三維形狀中，潛藏著這些必須瞭解的本質類型。可是，全連接層卻忽略了形狀，把所有的輸入資料當作同等神經元（相同維度的神經元）來處理，無法發揮與形狀有關的資料。

然而，卷積層（Convolution 層）能維持形狀。如果是影像，輸入資料可以當作三維資料來處理，對下一層輸出同樣是三維的資料。因此，CNN（可能）可以正確瞭解影像等含有形狀的資料。

有時，我們會把 CNN 的卷積層輸出入資料稱作特徵圖 (*feature map*)。甚至將卷積層的輸入資料稱作輸入特徵圖 (*input feature map*)，輸出資料稱作輸出特徵圖 (*output feature map*)。本書把「輸出入資料」與「特徵圖」視為同義詞。

## 7.2.2 卷積運算

在卷積層執行的處理，稱作「卷積運算」。若以影像處理來比喻卷積運算，相當於「濾鏡運算」。以下將舉個具體的例子（圖 7-3）來說明卷積運算。

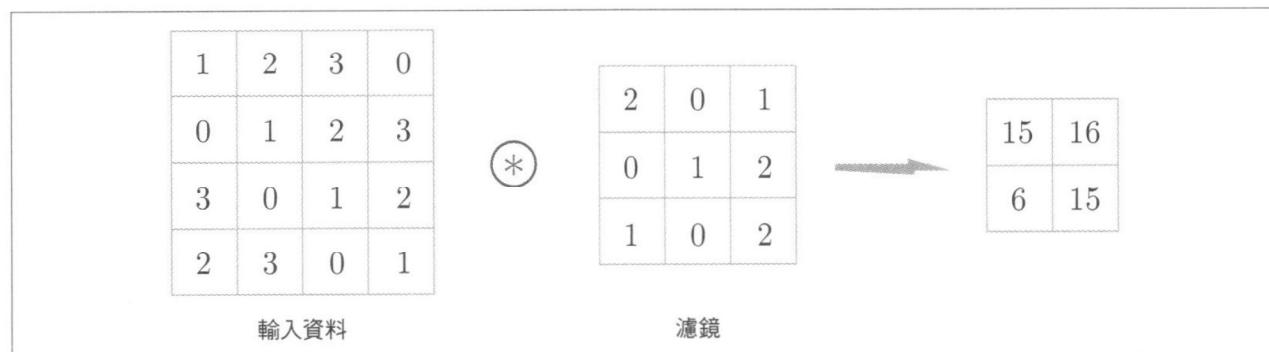


圖 7-3 卷積運算的範例：以「 $*$ 」符號代表卷積運算

如圖 7-3 所示，卷積運算是在輸入資料套用濾鏡。在這個範例中，輸入資料是擁有垂直、水平形狀的資料，濾鏡也同樣擁有垂直、水平方向的維度。資料與濾鏡的形狀以  $(height, width)$  顯示，在這個範例中，輸入大小為  $(4, 4)$ ，濾鏡大小是  $(3, 3)$ ，輸出大小為  $(2, 2)$ 。部分文獻會把這裡使用的名詞「濾鏡」，稱作「核（kernel）」。

接下來要說明在圖 7-3 的卷積運算範例中，執行了哪些計算。圖 7-4 顯示了卷積運算的計算步驟。

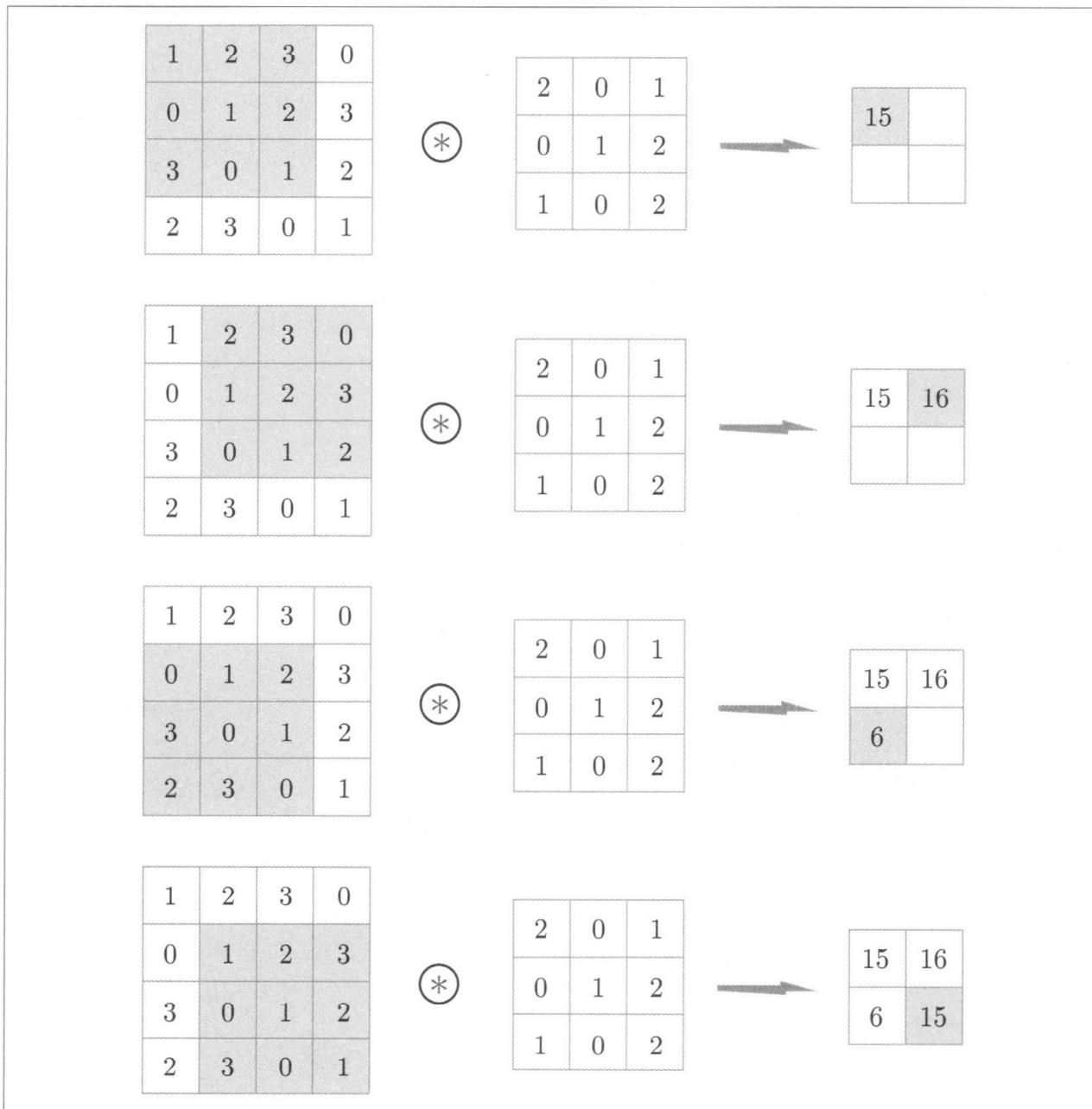


圖 7-4 卷積運算的計算步驟

卷積運算是針對輸入資料，以固定的間隔，一邊移動，一邊套用濾鏡的視窗。這裡所謂的視窗是指圖 7-4 的  $3 \times 3$  灰色部分。如圖 7-4 所示，在各個位置乘上濾鏡的元素與對應輸入的元素，並計算總和（這個部分也稱作積和運算）。再將結果儲存在對應輸出的位置。在全部的位置執行這個流程，可以得到卷積運算的輸出。

在全連接的神經網路中，除了權重參數之外，還有偏權值。CNN 的濾鏡參數對應的是前面提到的「權重」，而且在 CNN 之中，也有偏權值。圖 7-3 的卷積運算範例顯示了套用濾鏡後的階段。包含偏權值的卷積運算處理流程如圖 7-5 所示。

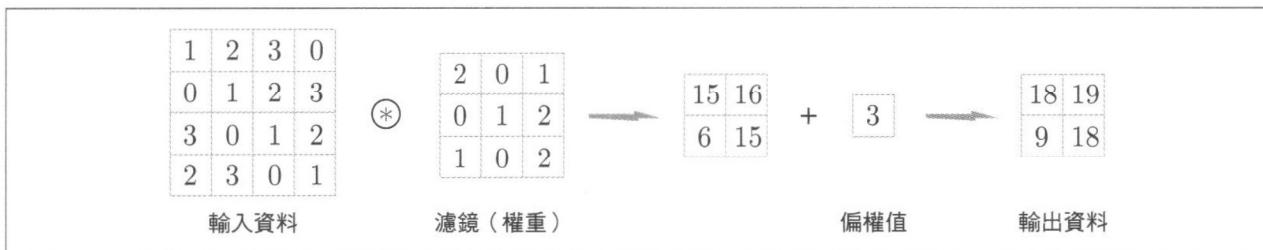


圖 7-5 卷積運算的偏權值：套用濾鏡後的元素加上固定值（偏權值）

如圖 7-5 所示，對套用濾鏡後的資料進行偏權值的加法運算。這裡可以看到，偏權值隨時都只有一個 ( $1 \times 1$ )（這個例子是，相對於 4 個套用濾鏡後的資料，偏權值只有 1 個），這個值要加在套用濾鏡後的所有元素。

### 7.2.3 填補

進行卷積處理之前，必須在輸入資料的周圍填上固定的資料（例如 0），這個動作稱作填補 (padding)，也是在卷積運算中，常用的處理。例如，在圖 7-6 的範例中，對於大小為  $(4, 4)$  的輸入資料，進行寬度 1 的填補。寬度 1 的填補是指，周圍用寬 1 像素的 0 填滿。

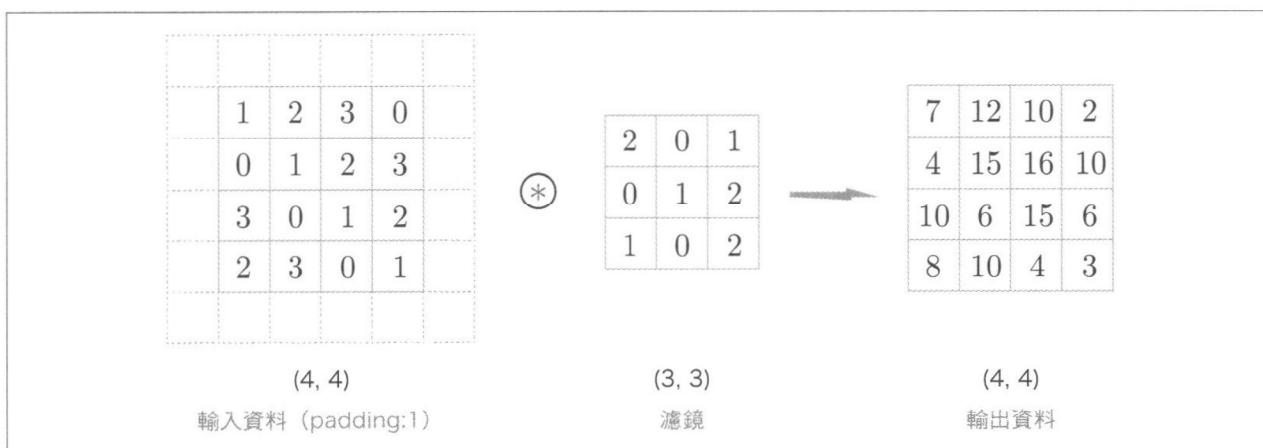


圖 7-6 卷積運算的填補處理：在輸入資料的周圍填補 0（圖中以虛線代表填補，省略掉內容中的「0」）

如圖 7-6 所示，(4, 4) 大小的輸入資料利用填補，變成 (6, 6) 的形狀。乘上 (3, 3) 大小的濾鏡，產生 (4, 4) 的輸出資料。在這個範例中，將填補設定為 1，但是填補的值可以設定成 2 或 3 等任意整數。假如圖 7-5 的範例將填補設定為 2，輸入資料的大小變成 (8, 8)，若填補為 3，輸入資料的大小會變成 (10, 10)。



使用填補的理由是為了調整輸出大小。假設在大小為 (4, 4) 的輸入資料中，套用 (3, 3) 的濾鏡時，輸出大小變成 (2, 2)，輸出資料只會縮小輸入資料的 2 個元素。如此一來，在反覆進行卷積運算的多層網路中，就會造成問題。如果每次進行卷積運算時，空間就會縮小的話，到了某個階段，輸出大小就會變成 1，而無法再進行卷積運算，使用填補就是為了避免出現這種情況。在前面的範例中，將填補的寬度設定為 1 時，相對於輸入大小為 (4, 4)，輸出大小也會保持 (4, 4)。因此，藉由卷積運算，可以維持固定的空間大小，將資料傳遞給下一層。

## 7.2.4 步幅

套用濾鏡的位置間隔稱作步幅 (stride)。到目前為止，全都是步幅為 1 的範例，假如步幅變成 2，套用濾鏡的視窗間隔會變成 2 個元素，如圖 7-7 所示。

在圖 7-7 的範例中，針對輸入大小為 (7, 7) 的資料，以步幅 2 的間隔來套用濾鏡。步幅設定為 2，輸出大小變成 (3, 3)。步幅是設定套用濾鏡的間隔。

如上面所示，步幅變大，輸出大小就會變小。然而，填補變大，輸出大小會變大。把這種關係公式化，會變成如何？接下來，要說明針對填補與步幅，計算出輸出大小。

這裡假設輸入大小為  $(H, W)$ ，濾鏡大小為  $(FH, FW)$ ，輸出大小為  $(OH, OW)$ ，填補為  $P$ ，步幅為  $S$ 。此時，可以用以下算式 (7.1) 計算輸出大小。

$$\begin{aligned} OH &= \frac{H + 2P - FH}{S} + 1 \\ OW &= \frac{W + 2P - FW}{S} + 1 \end{aligned} \tag{7.1}$$

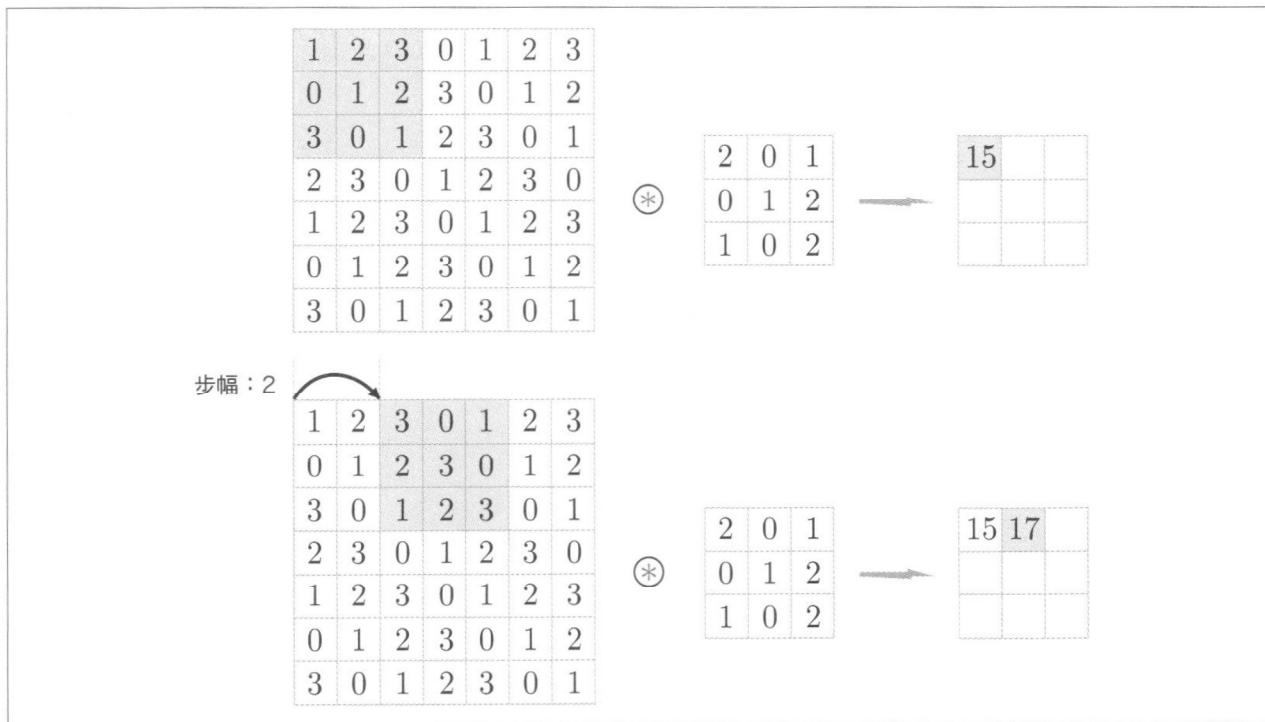


圖 7-7 步幅為 2 的卷積運算範例

接下來，使用這個算式，計算幾個範例。

#### 例 1：圖 7-6 的範例

輸入大小：(4, 4)、填補：1、步幅：1、濾鏡大小：(3, 3)

$$OH = \frac{4 + 2 \cdot 1 - 3}{1} + 1 = 4$$

$$OW = \frac{4 + 2 \cdot 1 - 3}{1} + 1 = 4$$

#### 例 2：圖 7-7 的範例

輸入大小：(7, 7)、填補：0、步幅：2、濾鏡大小：(3, 3)

$$OH = \frac{7 + 2 \cdot 0 - 3}{2} + 1 = 3$$

$$OW = \frac{7 + 2 \cdot 0 - 3}{2} + 1 = 3$$

例 3

輸入大小：(28, 31)、填補：2、步幅：3、濾鏡大小：(5, 5)

$$OH = \frac{28 + 2 \cdot 2 - 5}{3} + 1 = 10$$

$$OW = \frac{31 + 2 \cdot 2 - 5}{3} + 1 = 11$$

如同這些範例所示，在算式（7.1）帶入數值，即可計算輸出大小。雖然單純代入數值，就能計算輸出大小，但是這裡必須注意到，一定要分別設定成讓算式（7.1） $\frac{W+2P-FW}{S}$  與 $\frac{H+2P-FH}{S}$  整除的數值。假如輸出大小無法整除（結果為小數），就必須採取輸出錯誤對策。附帶一提，部分深度學習的框架遇到數值無法整除時，會變成最接近的整數，不顯示錯誤，繼續執行處理。

## 7.2.5 三維資料的卷積運算

到目前為止的卷積運算範例都是以含有垂直、水平方向的二維形狀為對象，但是如果是以影像，就必須處理除了垂直、水平方向之外，再加上色版方向的三維資料。以下將依照和前面一樣的步驟，針對包含色版的三維資料，進行卷積運算。

圖 7-8 是卷積運算的範例，圖 7-9 是計算步驟。這裡以 3 色版資料為例，顯示卷積運算的結果。與二維的情況（圖 7-3 的範例）做比較，可以得知，在深度方向（色版方向）增加了特徵圖。如果色版方向有多個特徵圖時，要依照各個色版進行輸入資料與濾鏡的卷積運算，把這些結果相加，獲得一個輸出。

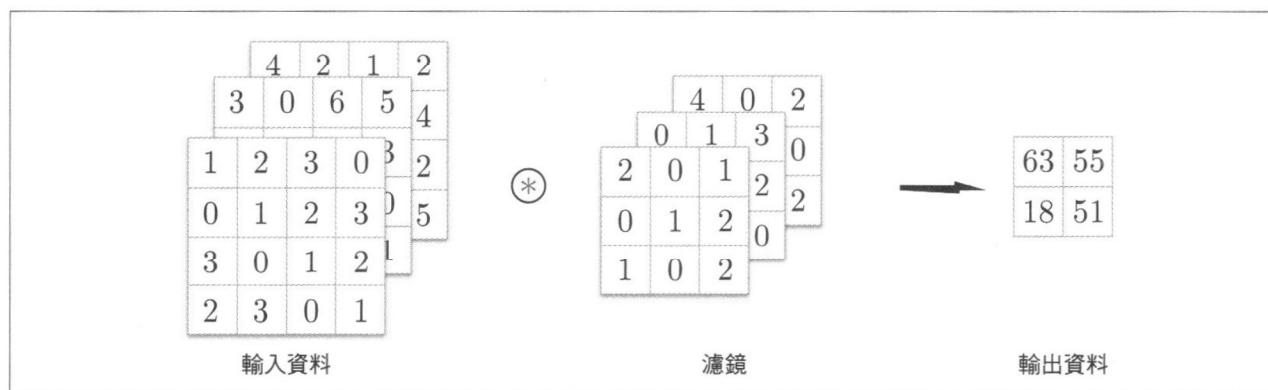


圖 7-8 三維資料的卷積運算範例

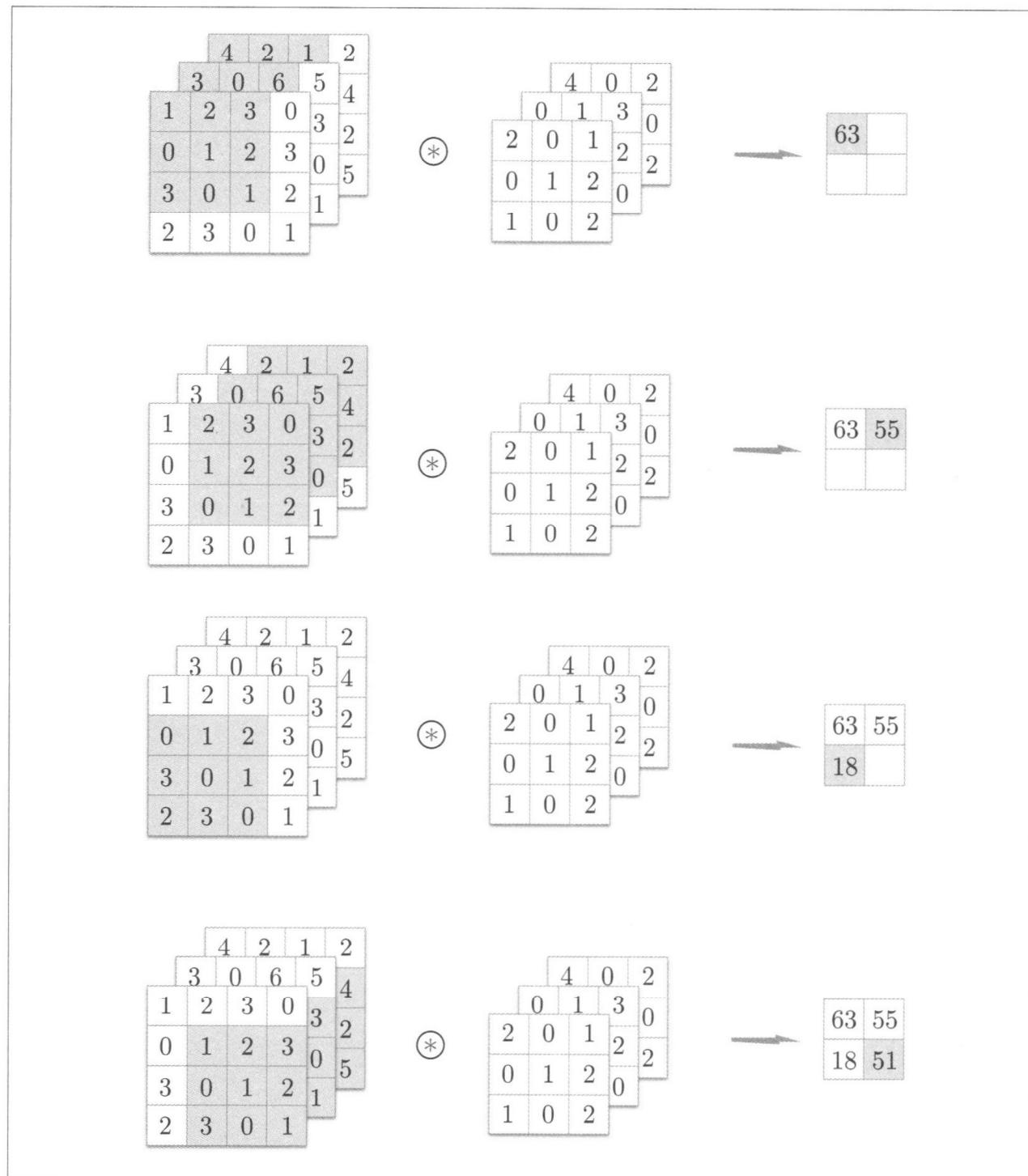


圖 7-9 三維資料的卷積運算步驟

在這個範例顯示的三維卷積運算中，必須注意到，輸入資料與濾鏡的色版數同值。就這個範例來說，輸入資料與濾鏡的色版數量都是 3。不過，濾鏡大小可以隨意設定（但是，每個色版的濾鏡大小都一樣）。這個範例的濾鏡大小是  $(3, 3)$ ，你也可以設定成  $(2, 2)$ 、 $(1, 1)$  或  $(5, 5)$  等任意值。可是，這裡要再次重申，色版數量與輸入資料的色版數必須同值，因此這個範例只能設定成 3。

### 7.2.6 用區塊來思考

你可以將三維卷積運算中的資料或濾鏡想像成立體區塊，比較容易思考。區塊是指，三維的立方體，如圖 7-10 所示。此外，把三維資料當作多維陣列時，會依照 (channel, height, width) 的順序來顯示。例如，色版數 C、高度 H、寬度 W 的資料形狀會寫成  $(C, H, W)$ 。濾鏡也同樣依照 (channel, height, width) 的順序來顯示。假設色版數 C、濾鏡高度 FH (Filter Height)、寬度 FW (Filter Width) 時，寫成  $(C, FH, FW)$ 。

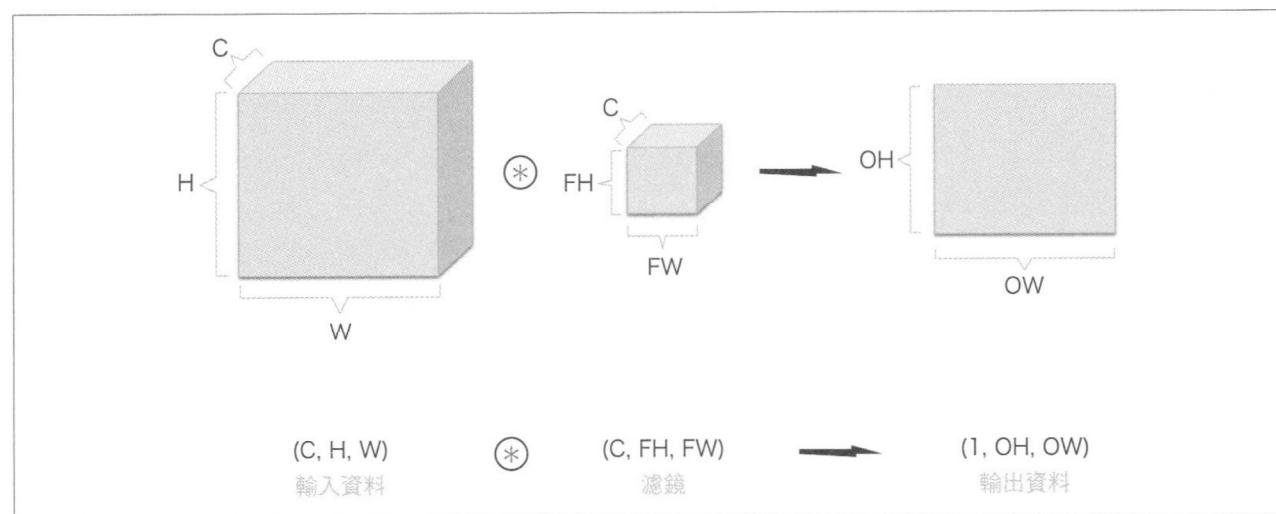


圖 7-10 以區塊來思考卷積運算，注意區塊的形狀

這個範例的資料輸出是一張特徵圖，換句話說，就是色版數量為 1 的特徵圖。如果要在色版方向，產生多個卷積運算的輸出時，該怎麼做？此時，要使用多個濾鏡（權重）。用圖表顯示，如圖 7-11 所示。

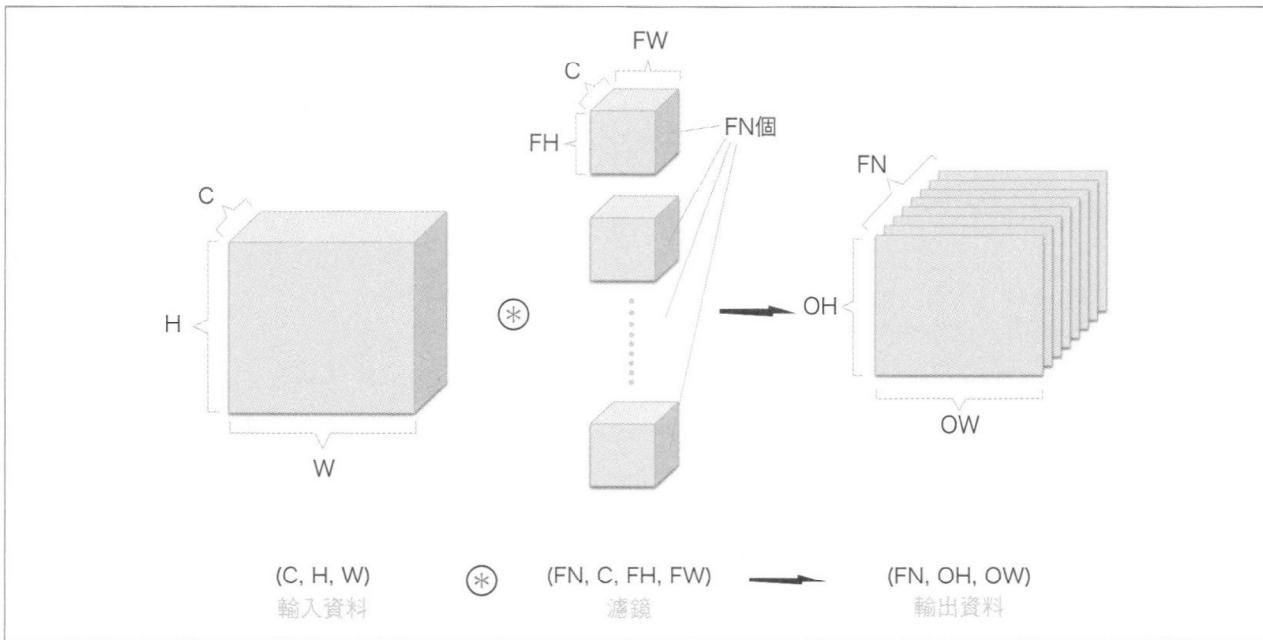


圖 7-11 多個濾鏡的卷積運算範例

如圖 7-11 所示，套用 FN 個濾鏡，也會產生 FN 個輸出的特徵圖。整合 FN 個特徵圖，就會完成形狀為 (FN, OH, OW) 的區塊。把完成的區塊傳遞給下一層，就是 CNN 的處理流程。

如圖 7-11 所示，卷積運算的濾鏡也必須考慮到濾鏡的數量。因此，濾鏡的權重資料會依照 (output\_channel, input\_channel, height, width) 的順序顯示，當作四維資料。假設有 20 個色版數量為 3、大小為  $5 \times 5$  的濾鏡，會寫成 (20, 3, 5, 5)。

在卷積運算中（和全連接層相同），也有偏權值。以圖 7-11 為例，加上偏權值的加法運算後，結果如圖 7-12 所示。

如圖 7-12 所示，偏權值是每個色版只有一個資料。這裡的偏權值形狀是 (FN, 1, 1)，濾鏡輸出結果的形狀是 (FN, OH, OW)。在這 2 個區塊的加法運算中，針對濾鏡的輸出結果 (FN, OH, OW)，在每個色版加上相同的偏權值。另外，不同形狀的區塊加法運算，利用 NumPy 的廣播，就可以輕鬆完成（請參考「1.5.5 廣播」）。

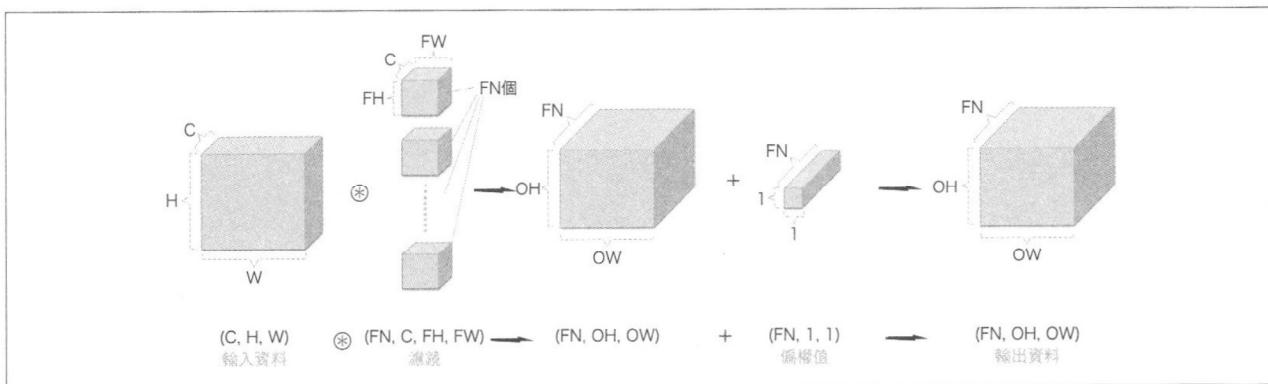


圖 7-12 卷積運算的處理流程（加上偏權值）

### 7.2.7 批次處理

在神經網路的處理中，進行了把輸入資料整合成一束的批次處理。前面介紹的全連接神經網路，也支援批次處理，能提高處理效率，以及因應學習時的小批次。

卷積運算也同樣希望能支援批次處理，因此把流動在各層的資料儲存成四維資料。具體來說是依照 (batch\_num, channel, height, width) 的順序來儲存資料。例如，針對圖 7-12，進行對應 N 個資料的批次處理，資料的形狀如圖 7-13 所示。

在圖 7-13 批次處理版的資料流程中，於各資料的開頭加上批次用的維度，把資料當成四維形狀，傳遞給各層。這裡要注意到，雖然在網路中傳遞四維資料，但這代表針對 N 個資料來進行卷積運算。換句話說，就是一次執行完 N 次處理。

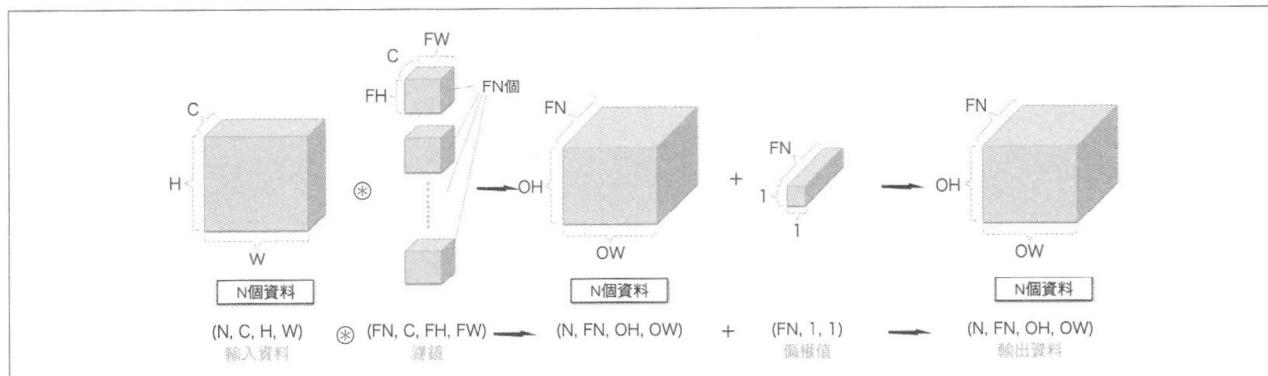


圖 7-13 卷積運算的處理流程（批次處理）

## 7.3 池化層

池化層是縮小垂直、水平空間的運算。如圖 7-14 所示，把  $2 \times 2$  的範圍整合成一個元素，縮小空間大小。

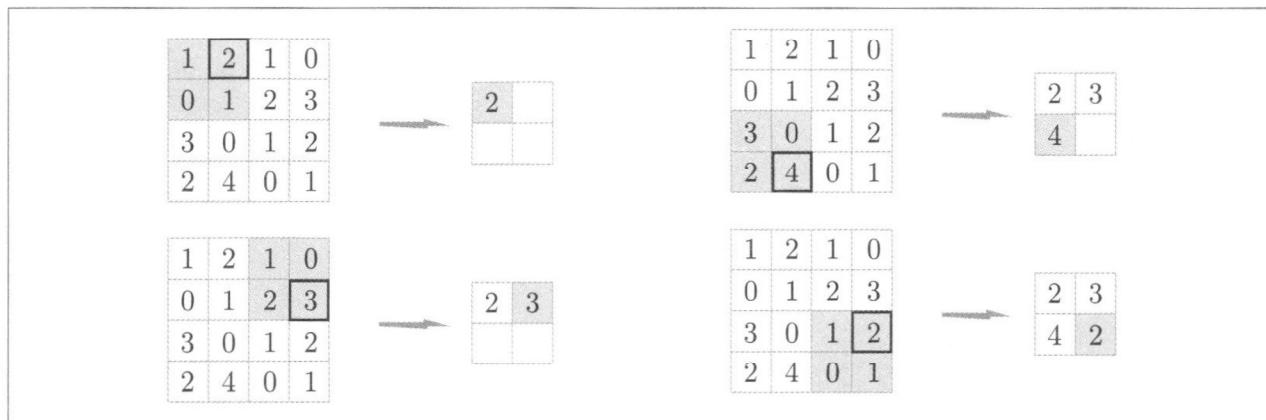


圖 7-14 最大池化的處理步驟

圖 7-14 的範例是，當步幅為 2，進行  $2 \times 2$  最大池化的處理步驟。「最大池化」是用來取得最大值，「 $2 \times 2$ 」是代表成為處理對象的區域大小。如圖所示，針對  $2 \times 2$  區域，取出最大元素。此外，這個範例的步幅設定為 2，所以  $2 \times 2$  的視窗移動間隔變成以每 2 個元素來移動。一般而言，池化的視窗大小與步幅設定成相同數值。假設  $3 \times 3$  的視窗，步幅設定為 3， $4 \times 4$  的視窗，步幅為設定為 4。



除了最大池化 (max pooling) 之外，還有平均池化 (average pooling) 等。最大池化是從目標區域中，取得最大值；而平均池化是計算目標區域的平均值。在影像辨識的領域中，主要使用的是最大池化。因此，本書提及「池化層」時，指的是最大池化。

### 7.3.1 池化層的特色

池化層有以下特色。

#### 沒有學習參數

池化層與卷積層不同，沒有學習參數。池化只進行從目標區域取得最大值（或平均值）的處理，所以沒有必須學習的參數存在。

### 色版數量不變

輸入資料與輸出資料的色版數量不會隨著池化運算而改變。如圖 7-15 所示，各個色版進行獨立的運算。

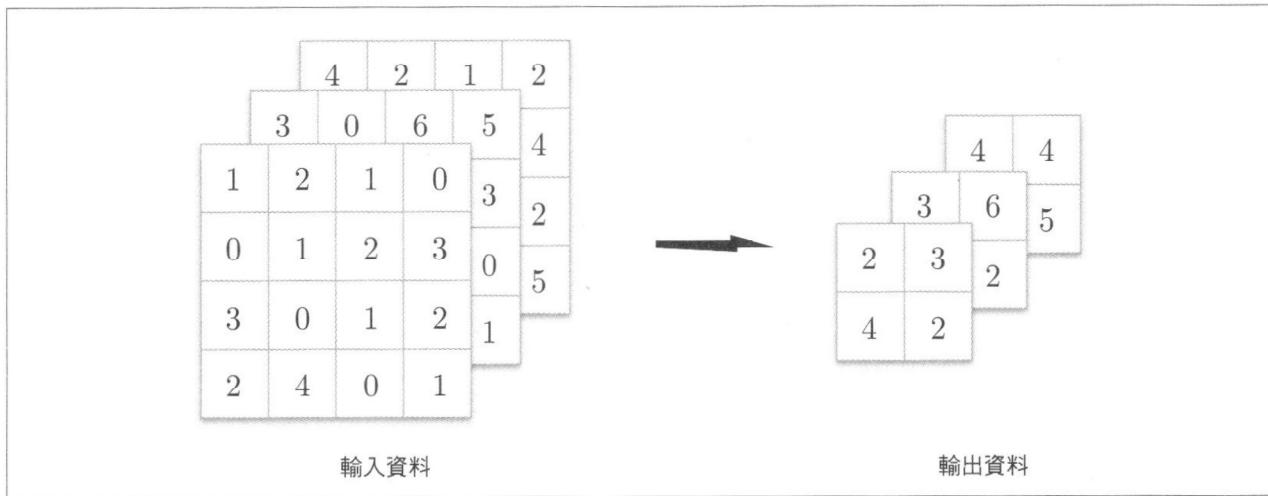


圖 7-15 池化不會改變色版的數量

### 對微小位置變化很穩健 (robust)

即使輸入資料出現小偏差，池化仍會回傳相同結果。因此，對輸入資料的微小偏差很穩健。假設有個  $3 \times 3$  的池化，如圖 7-16 所示，池化會吸收掉輸入資料的偏差（部分資料可能出現不一致的情況）。

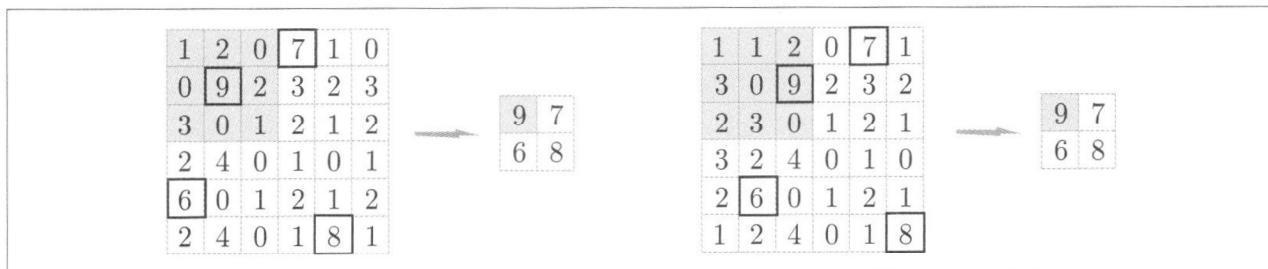


圖 7-16 即使輸入資料只往水平方向偏差 1 個元素，也能輸出相同結果（部分資料可能出現不一致的情況）

## 7.4 執行卷積層 / 池化層

前面已經詳細介紹了卷積層與池化層。以下要用 Python 執行這兩層。「第 5 章 誤差反向傳播法」說明過，這裡執行的類別中，含有 `forward` 與 `backward` 等方法，可以當作模組使用。

或許你會認為執行卷積層與池化層很複雜，其實只要使用某個「妙招」，就可以輕鬆完成。本節要說明這個妙招，把問題簡化之後，再執行卷積層。

### 7.4.1 四維陣列

前面說明過，在 CNN 各層流動的資料是四維資料。四維資料是指，資料的形狀如果是  $(10, 1, 28, 28)$ ，等於有 10 個高度 28、寬度 28、1 色版的資料。使用 Python 執行，結果如下所示。

```
>>> x = np.random.rand(10, 1, 28, 28) # 隨機產生資料
>>> x.shape
(10, 1, 28, 28)
```

這裡在存取第 1 個資料時，只寫了 `x[0]`（請注意，Python 的索引值是從 0 開始）。同樣第 2 個資料可以用 `x[1]` 來存取。

```
>>> x[0].shape # (1, 28, 28)
>>> x[1].shape # (1, 28, 28)
```

另外，存取第 1 個資料的第 1 色版空間資料時，程式如下所示。

```
>>> x[0, 0] # 或 x[0][0]
```

這樣在 CNN 就會變成要處理四維資料，使得執行卷積運算的過程變得很複雜，其實只要使用下面要說明的 `im2col` 這個「妙招」，問題就會變簡單。

### 7.4.2 利用 `im2col` 展開

執行卷積運算時，按部就班的處理，需要重疊多重 `for` 陳述式。可是這樣有點麻煩，而且在 NumPy 使用 `for` 陳述式，還有處理速度緩慢的缺點（在 NumPy 存取元素時，最好盡量別使用 `for` 陳述式）。這裡不執行 `for` 陳述式，改用方便的 `im2col` 函數來進行簡單處理。

`im2col` 是可以針對套用濾鏡（權重），輕鬆展開輸入資料的函數。如圖 7-17 所示，對三維的輸入資料套用 `im2col`，就會轉換成二維陣列（正確來說，是將含有批次數的四維資料轉換成二維）。

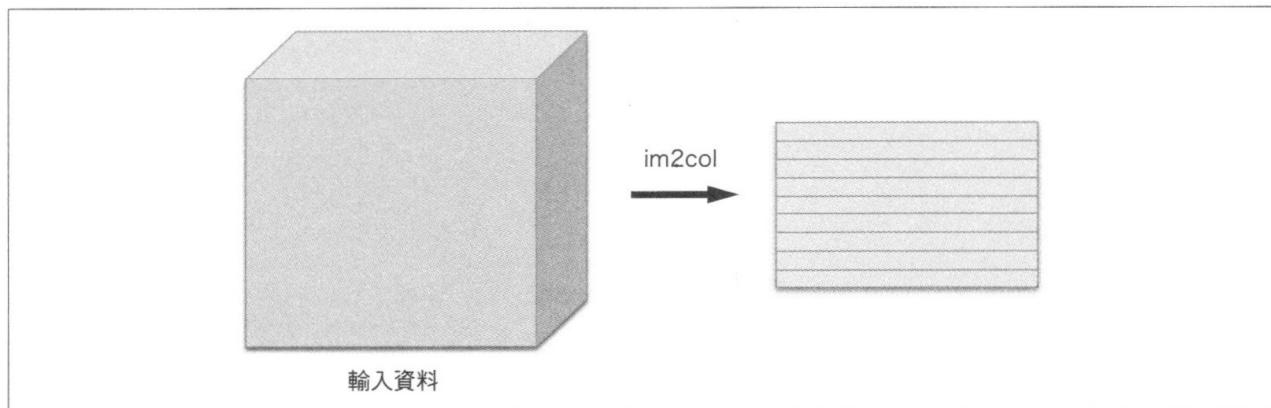


圖 7-17 `im2col` 的概略圖

`im2col` 可以針對套用濾鏡的位置，輕易展開輸入資料。具體而言，如圖 7-18 所示，對輸入資料套用濾鏡的區域（三維區塊）往水平方向展開成 1 行。`im2col` 就是對套用了濾鏡的所有位置進行這種展開處理。

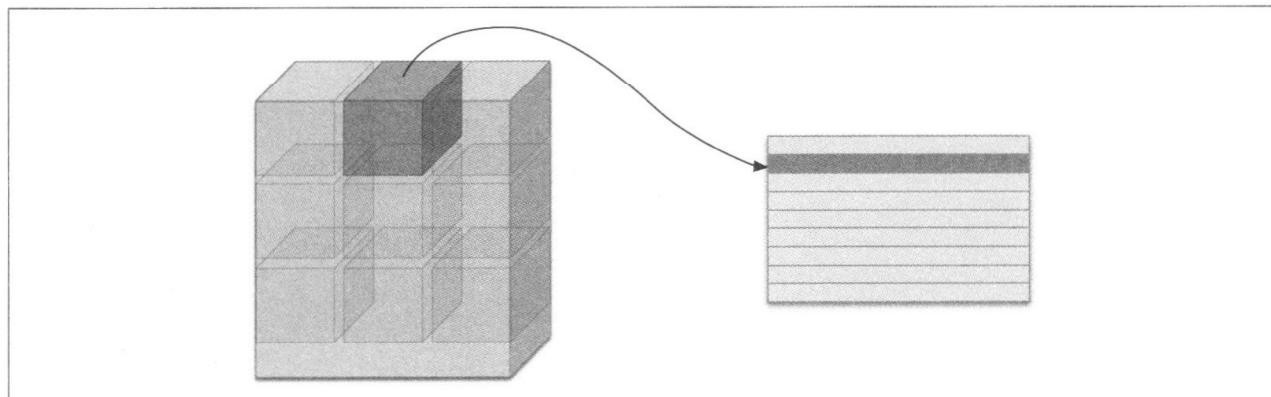


圖 7-18 從頭開始依序把套用濾鏡區域展開成 1 行

另外，圖 7-18 以易讀性為前提，把步幅設定成較大數值，避免套用濾鏡的區域重疊。實際上，進行卷積運算時，幾乎濾鏡區域都會重疊。套用濾鏡的區域重疊時，利用 `im2col` 展開，展開後的元素數量，會變得比原本區塊的元素數量還多。因此，使用 `im2col` 時，通常會產生占用比較多記憶體的缺點。可是，利用電腦統一計算大型陣列，有很多好處。例如，計算陣列的函式庫（線性代數函式庫）等，可以先將陣列計算最佳

化，再快速執行大型陣列的乘法運算。因此，利用還原陣列計算的方式，能有效運用線性代數函式庫。



`im2col` 是「image to column」的縮寫，意思是將影像轉換成陣列。在 Caffe 及 Chainer 等深度學習的框架中，含有名為 `im2col` 的函數，執行卷積層時，可以使用各個 `im2col` 來進行處理。

利用 `im2col` 展開輸入資料，之後只要把卷積層的濾鏡（權重）展開成 1 行，計算 2 個陣列的乘積（請參考圖 7-19）。這個部分與在全連接層 Affine 層進行的處理幾乎一模一樣。

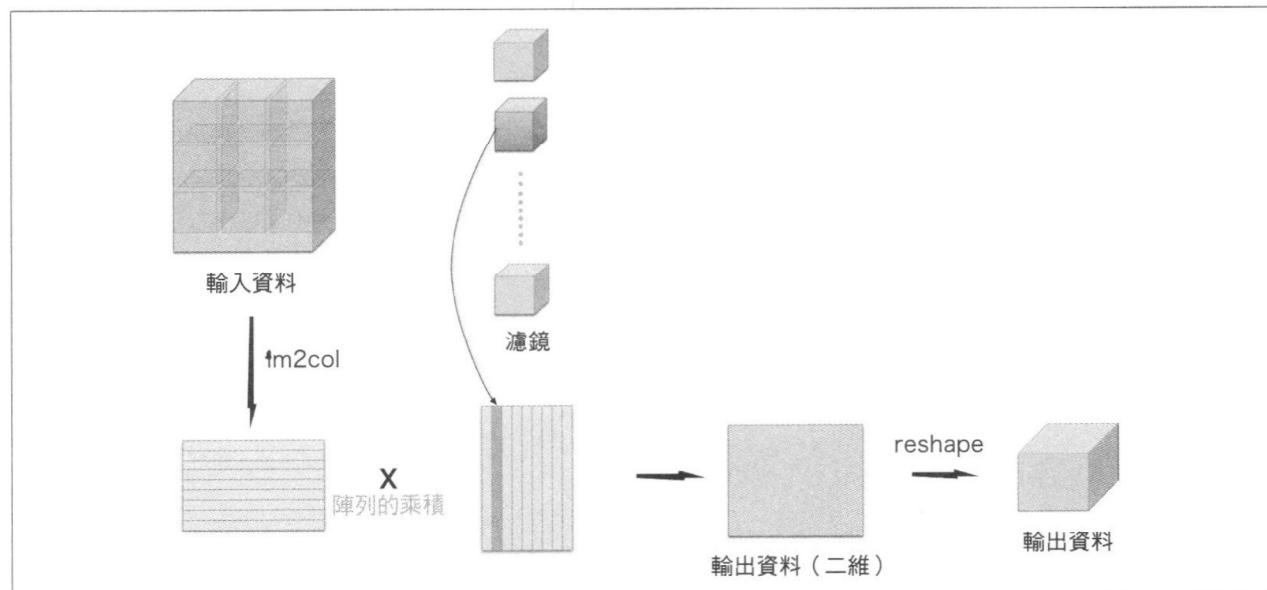


圖 7-19 卷積運算的濾鏡處理詳細說明：往垂直方向把濾鏡展開成 1 行，計算以 `im2col` 展開的資料與陣列的乘積，最後調整（`reshape`）輸出資料的大小

如圖 7-19 所示，利用 `im2col` 方法輸出的結果是二維陣列。CNN 的資料是四維陣列，所以將二維輸出資料調整成適當形狀，以上就是卷積層的執行流程。

### 7.4.3 執行卷積層

本書提供了 `im2col` 函數。請把這個 `im2col` 函數想像成在進行黑箱測試（不用在意執行的內容）。此外，`im2col` 的執行內容位於 `common/util.py`，這是一個（實質上）只有 10 行程式的簡單函數。有興趣的讀者可以自行參考。

`im2col` 是一個很方便的函數，因為它含有以下介面。

- ```
im2col(input_data, filter_h, filter_w, stride=1, pad=0)
```
- `input_data` —— 由（資料數量、色版、高度、寬度）的四維陣列形成的輸入資料
  - `filter_h` —— 濾鏡的高度
  - `filter_w` —— 濾鏡的寬度
  - `stride` —— 步幅
  - `pad` —— 填補

`im2col` 考量到「濾鏡大小」、「步幅」、「填補」，把輸入資料展開成二維陣列。接下來，讓我們來實際使用 `im2col`。

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

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

x2 = np.random.rand(10, 3, 7, 7) # 10 個資料
col2 = im2col(x2, 5, 5, stride=1, pad=0)
print(col2.shape) # (90, 75)
```

這裡顯示了 2 個範例。第 1 個是批次大小為 1，色版為 3 的  $7 \times 7$  資料，第 2 個是批次大小為 10，資料形狀和第 1 個相同的範例。分別套用 `im2col` 函數，這 2 個範例，第二維的元素數量為 75。這是濾鏡（色版 3、大小  $5 \times 5$ ）的元素數量總和。此外，批次大小為 1 的時候，`im2col` 的結果是大小為 (9, 75)。然而，第 2 個範例的批次大小為 10，所以儲存了 (90, 75) 的 10 倍資料。

接下來，要使用 `im2col`，執行卷積層。這裡以名稱為 `Convolution` 的類別來執行卷積層。

```
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 = int(1 + (H + 2*self.pad - FH) / self.stride)
out_w = int(1 + (W + 2*self.pad - FW) / self.stride)

col = im2col(x, FH, FW, self.stride, self.pad)
col_W = self.W.reshape(FN, -1).T # 展開濾鏡
out = np.dot(col, col_W) + self.b

out = out.reshape(N, out_h, out_w, -1).transpose(0, 3, 1, 2)

return out

```

卷積層的初始化方法是把濾鏡（權重）、偏權值、步幅、填補當作引數來取得。濾鏡是 (FN, C, FH, FW) 的四維形狀。此外 FN 是 Filter Number (濾鏡的數量)，C 是 Channel，FH 是 Filter Height，FW 是 Filter Width 的縮寫。

在執行卷積層的過程，以粗體字顯示重要的部分。這些粗體字部分包含以 `im2col` 展開輸入資料，濾鏡也使用 `reshape`，展開成二維陣列，然後再計算展開後的陣列乘積。

展開濾鏡的位置（原始碼中的粗體字部分）如圖 7-19 所示，將各濾鏡的區塊展開成 1 行。`reshape(FN, -1)` 設定成 -1，這是 `reshape` 的方便功能之一。`reshape` 時，設定成 -1，可以整合元素數量，使多維陣列的元素數量完美配合。例如，(10, 3, 5, 5) 形狀的陣列，全部的元素數量共有 750 個，假設 `reshape(10, -1)`，就能調整成形狀為 (10, 75) 的陣列。

另外，執行 `forward` 時，最後要將輸出大小調整成適當形狀。調整形狀時，使用的是 NumPy 的 `transpose` 函數。`transpose` 是更換多維陣列各軸順序的函數。如圖 7-20 所示，設定從 0 開始的索引值（編號），更換軸的順序。

以上是卷積層的 `forward` 處理。利用 `im2col` 展開，可以執行和全連接層 `Affine` 層幾乎一樣的處理（請參考「5.6 執行 `Affine` / `Softmax` 層」）。接下來是卷積層的反向傳播，這個部分與 `Affine` 層的執行過程有很多共通點，所以在此省略說明。不過必須注意到，卷積層進行反向傳播時，要執行 `im2col` 的反向處理，請使用本書提供的 `col2im` 函數（`col2im` 的原始碼位於 `common/util.py`）來因應這個問題。除了使用 `col2im`，其餘卷積層反向傳播執行的部分，和 `Affine` 層一樣。卷積層的反向傳播原始碼位於 `common/layer.py`，有興趣的讀者請自行參考。

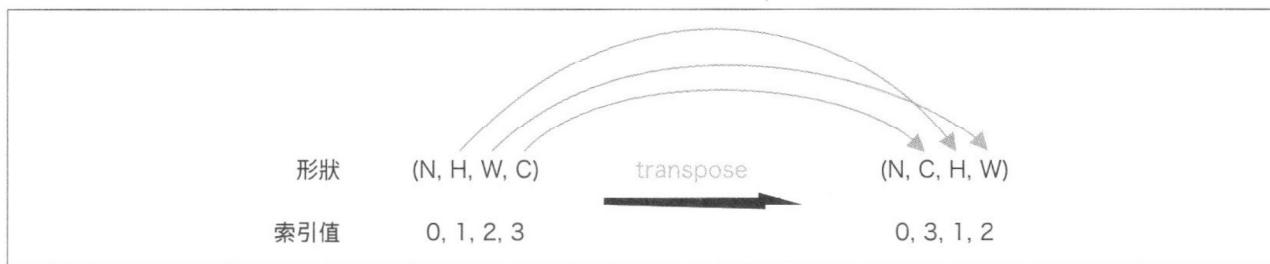


圖 7-20 利用 NumPy 的 transpose 改變軸的順序：利用索引值（編號），調整軸的順序

#### 7.4.4 執行池化層

池化層的執行過程和卷積層一樣，都是使用 im2col 展開輸入資料。但是，池化層的色版方向是獨立的，這點與卷積層不同。具體來說，如圖 7-21 所示，套用池化層的區域會依照色版獨立展開。

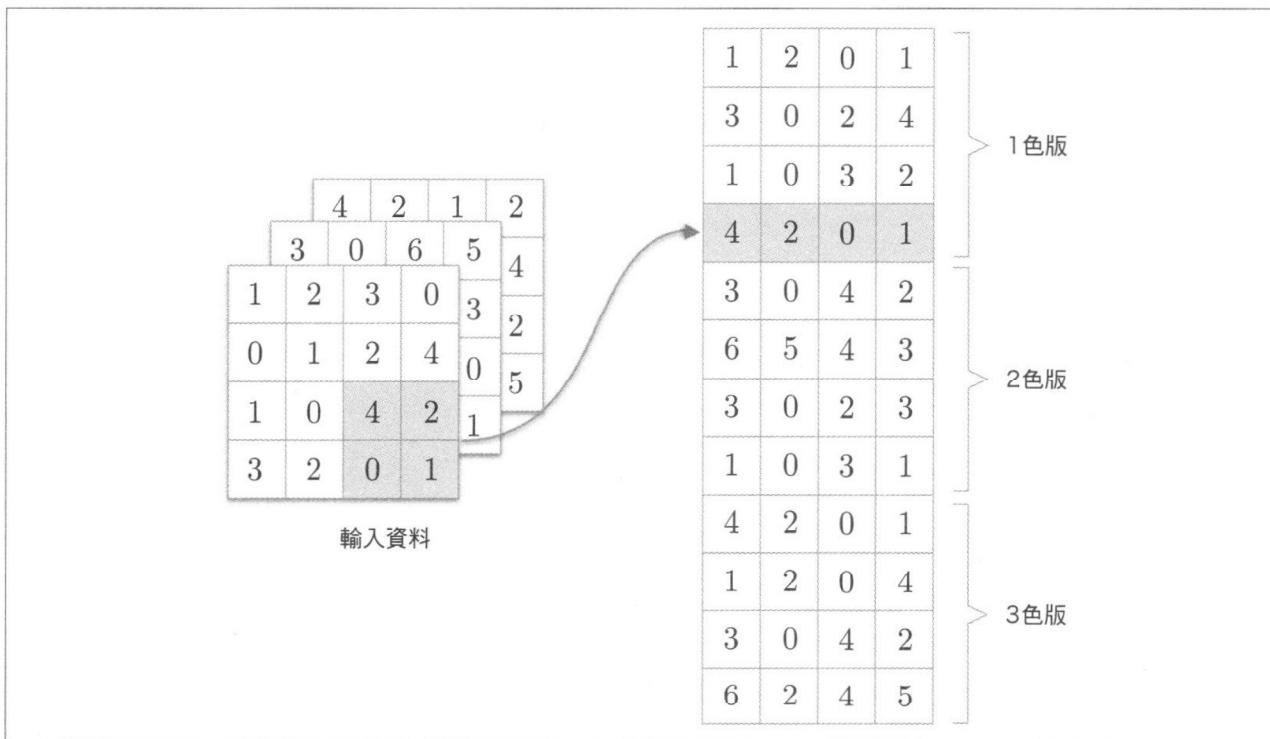


圖 7-21 針對輸入資料，展開套用了池化的區域（ $2 \times 2$  的池化範例）

只要展開一次，之後再針對展開後的陣列，計算每行的最大值，調整成適當形狀即可（圖 7-22）。

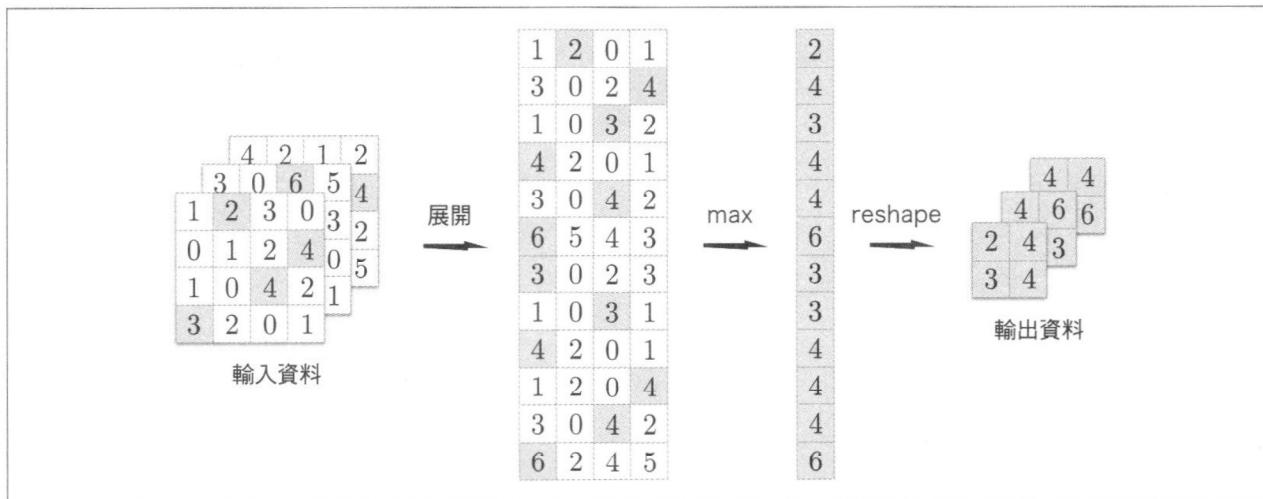


圖 7-22 池化層的執行過程：以灰色顯示套用池化區域內的最大值元素

以上是池化層的 forward 處理過程。接下來，要舉一個使用 Python 執行的實際範例。

```
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)
        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
```

池化層的執行過程如圖 7-22 所示，按照以下 3 個階段來進行。

1. 展開輸入資料
2. 計算每列的最大值

### 3. 調整成適當的輸出大小

各階段的執行過程只有 1、2 行程式，非常簡單。



利用 NumPy 的 `np.max` 方法可以計算最大值。`np.max` 可以將引數設定為 `axis`，就能利用引數算出各軸的最大值。例如，`np.max(x, axis=1)` 是計算出輸入 `x` 第一維各軸的最大值。

以上是池化層的 `forward` 處理。如這裡所示，只要將輸入資料展開成容易執行池化的形狀，後續的處理就很簡單。

至於池化層的 `backward` 處理，由於相關事項已經說明過，所以這裡省略不提。池化層的 `backward` 處理可以參考 ReLU 層執行時，使用的 `max` 反向傳播（「5.5.1 ReLU 層」）。執行池化層的原始碼位於 `common/layer.py`，有興趣的讀者請自行參考。

## 7.5 執行 CNN

由於前面已經執行了卷積層與池化層，所以接下來，要把這兩層組合起來，製作辨識手寫數字的 CNN。執行 CNN 的過程如圖 7-23 所示。

圖 7-23 的網路結構是「Convolution - ReLU - Pooling - Affine - ReLU - Affine - Softmax」。利用名為 `SimpleConvNet` 的類別來執行。

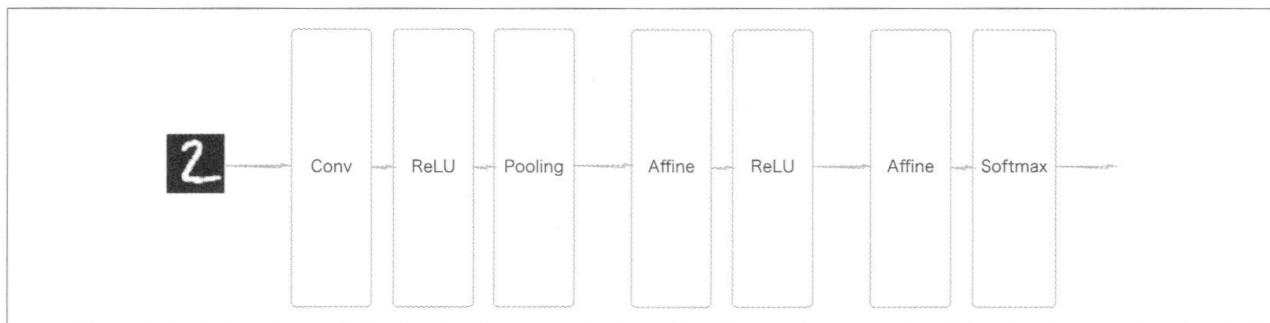


圖 7-23 單純的 CNN 網路結構

接下來，先檢視最初的 `SimpleConvNet` 初始化 (`__init__`)，利用這個部分取得以下引數。

引數

- `input_dim` —— 輸入資料的（色版、高度、寬度）維度
  - `conv_param` —— 卷積層的超參數（字典）。字典的 key 如下所示
    - `filter_num` —— 濾鏡的數量
    - `filter_size` —— 濾鏡的大小
    - `stride` —— 步幅
    - `pad` —— 填補
  - `hidden_size` —— 隱藏層（全連接）的神經元數
  - `output_size` —— 輸出層（全連接）的神經元數
  - `weight_init_std` —— 初始化時的權重標準差

卷積層的超參數是以名為 `conv_param` 的字典型態來提供。例如 `{'filter_num':30, 'filter_size':5, 'pad':0, 'stride':1}`，將所需的超參數儲存在裡面。

SimpleConvNet 的初始化過程比較長，所以分成 3 個部分來說明。接下來是初始化的第一個部分。

```

class SimpleConvNet:
    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))

```

從字典型態中，取出當作初期化引數的卷基層超參數（為了方便後續使用），再計算卷積層的輸出大小。接下來是進行權重參數初始化的部分。

```

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)

```

學習過程需要的參數是，第 1 層的卷積層，以及其餘 2 個全連接層的權重與偏權值。這些參數儲存在實例變數的 `params` 字典裡。字典型態的 `key` 是第 1 層的卷積層權重 `W1`，以及偏權值 `b1`。同樣分別利用第 2 層全連接層的權重與偏權值 `W2`、`b2`，第 3 層全連接層的權重與偏權值 `W3`、`b3` 等 `key` 來儲存參數。

最後產生必要的層級。

```

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()

```

從前面開始依序在含有字典型態（`OrderedDict`）的 `layers` 中，增加層數。只有最後的 `SoftmaxWithLoss` 層加上另一個變數 `lastLayer`。

以上就是 `SimpleConvNet` 初始化執行的處理。只要完成初始化，就可以按照以下所示，執行用來推論的 `predict` 方法與計算損失函數的 `loss` 方法。

```

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.lastLayer.forward(y, t)

```

引數  $x$  是輸入資料， $t$  是訓練標籤。用來推論的 `predict` 方法，只會從頭開始依序呼叫追加的層級，然後把結果傳遞給下一層。在計算損失函數的 `loss` 中，除了用 `predict` 方法進行的 `forward` 處理之外，最後 `SoftmaxWithLoss` 層也會進行 `forward` 處理。

接著是利用誤差反向傳播法，計算梯度，過程如下所示。

```
def gradient(self, x, t):
    # 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'] = self.layers['Conv1'].dW
    grads['b1'] = self.layers['Conv1'].db
    grads['W2'] = self.layers['Affine1'].dW
    grads['b2'] = self.layers['Affine1'].db
    grads['W3'] = self.layers['Affine2'].dW
    grads['b3'] = self.layers['Affine2'].db

    return grads
```

利用誤差反向傳播法（反向傳播）計算出參數的梯度。連續進行正向傳播與反向傳播。由於分別在各層執行了正確的正向傳播與反向傳播，所以這裡只要依照適當的順序來呼叫即可。最後，在 `grads` 字典中，儲存各權重參數的梯度。以上就是 `SimpleConvNet` 的執行過程。

接下來，試著利用 `SimpleConvNet` 學習 MNIST 資料集。學習用的原始碼和「4.5 執行學習演算法」說明過的內容一樣。因此，這裡不重複顯示（該原始碼位於 `ch07/train_convnet.py`）。

利用 `SimpleConvNet` 學習 MNIST 資料集之後，訓練資料的辨識率為 99.82%，測試資料的辨識率是 98.96%（各學習的辨識準確度會產生些許誤差）。測試資料的辨識率約 99%，對於相對較小的網路而言，辨識率算是非常高的。此外，下一章要利用重疊更多層的方式，建立測試資料辨識率超過 99% 的網路。

如上所述，卷積層與池化層是進行影像辨識時，不可缺少的模組。CNN 可以成功讀取具有空間形狀特性的影像，在手寫數字辨識方面，也能達到高準確性的辨識度。

## 7.6 CNN 的視覺化

使用於 CNN 的卷積層，究竟可以「看到什麼」？以下將透過視覺化卷積層的方式，探討 CNN 到底執行了什麼。

### 7.6.1 視覺化第 1 層權重

前面針對 MNIST 資料集，進行了單純的 CNN 學習。此時，第 1 層卷積層的權重形狀為  $(30, 1, 5, 5)$ ，亦即大小是  $5 \times 5$ ，色版為 1 的濾鏡有 30 個。濾鏡大小為  $5 \times 5$ ，色版數量為 1，代表濾鏡可以當作是 1 色版的灰階影像。接下來，把卷積層（第 1 層）的濾鏡顯示成影像。這裡要比較的是學習前與學習後的權重，結果如圖 7-24 所示（原始碼位於 ch07/visualize\_filter.py）。

如圖 7-24 所示，由於學習前的濾鏡進行了隨機初始化，所以黑白深淺沒有規則性。然而，結束學習的濾鏡變成了有規則性的影像。由此可知，藉由學習，更新成具有規則性的濾鏡，包括伴隨由白到黑漸層變化的濾鏡、具有塊狀區域（這裡稱作「塊（blob）」）的濾鏡等。

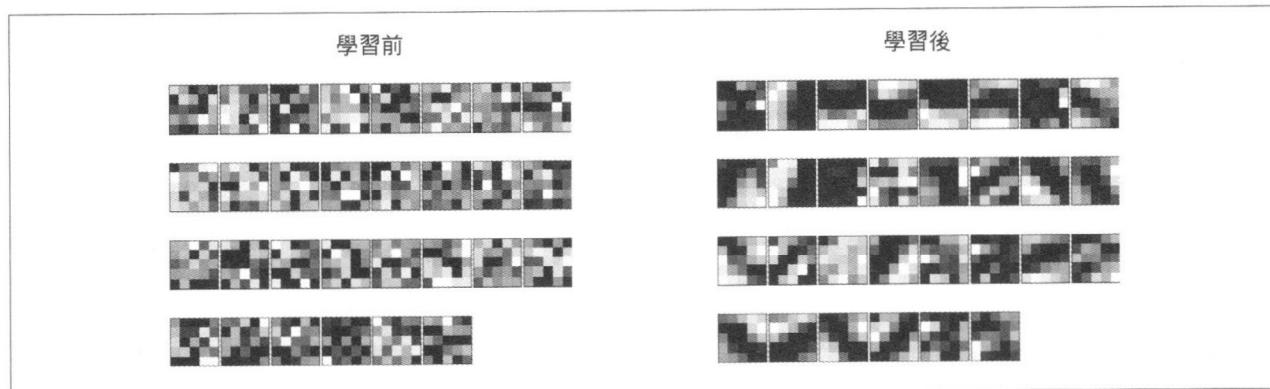


圖 7-24 學習前與學習後第 1 層卷積層的權重：權重的元素是實數，在顯示的影像中，以最小值為黑色 (0)，最大值為白色 (255) 來進行正規化

圖 7-24 右側這種具有規則性的濾鏡，「看到了什麼」？看到了邊界（顏色變化的邊緣）與塊狀（局部的塊狀區域）等。假設濾鏡的左半部分是白色，右半部分是黑色，如圖 7-25 所示，會變成顯示垂直邊界的濾鏡。

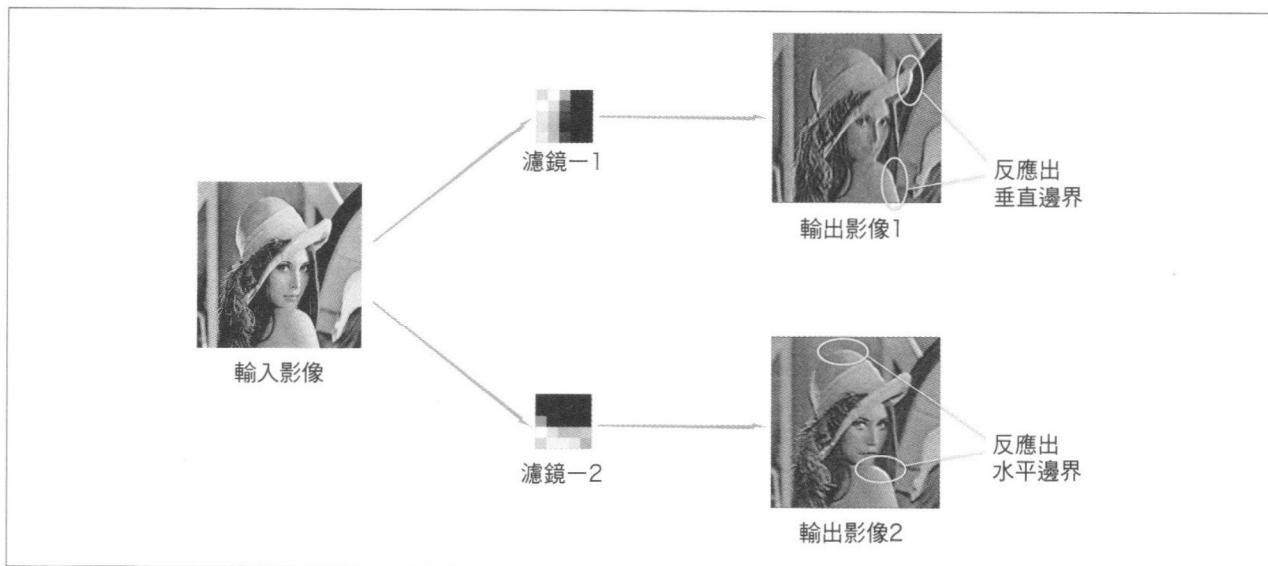


圖 7-25 反應水平邊界與垂直邊界的濾鏡：輸出影像 1 在垂直邊界出現白色像素，而輸出影像 2 在水平邊界出現許多白色像素

圖 7-25 是挑選 2 個完成學習的濾鏡，顯示在輸入影像進行卷積處理時的結果。由此可知，「濾鏡 1」反應出垂直邊界，「濾鏡 2」反應出水平邊界。

由此可知，這種卷積層的濾鏡可以擷取出邊界與塊狀等原始資料。把這種原始資料傳遞給下一層，就是由前面說明的 CNN 來執行。

## 7.6.2 利用階層結構擷取資料

上面的結果是以第 1 層卷積層為對象，在第 1 層卷積層中，擷取出邊界及塊狀等初階資料。在可以重疊多層結構的 CNN 中，各層能擷取出哪種資料？根據深度學習視覺化的研究 [17] [18] 顯示，隨著層數加深，擷取的資料（正確來說是反應強烈的神經元）會變得更抽象化。

圖 7-26 是用來辨識一般物體（車或狗等）的 8 層 CNN。在這個網路結構中，加上下一節要說明的 AlexNet 名稱。這個 AlexNet 的網路結構是重疊數層卷積層與池化層，最後透過全連接層輸出結果。圖 7-26 的區塊顯示的是中間資料，針對這些中間資料，連續套用卷積運算。

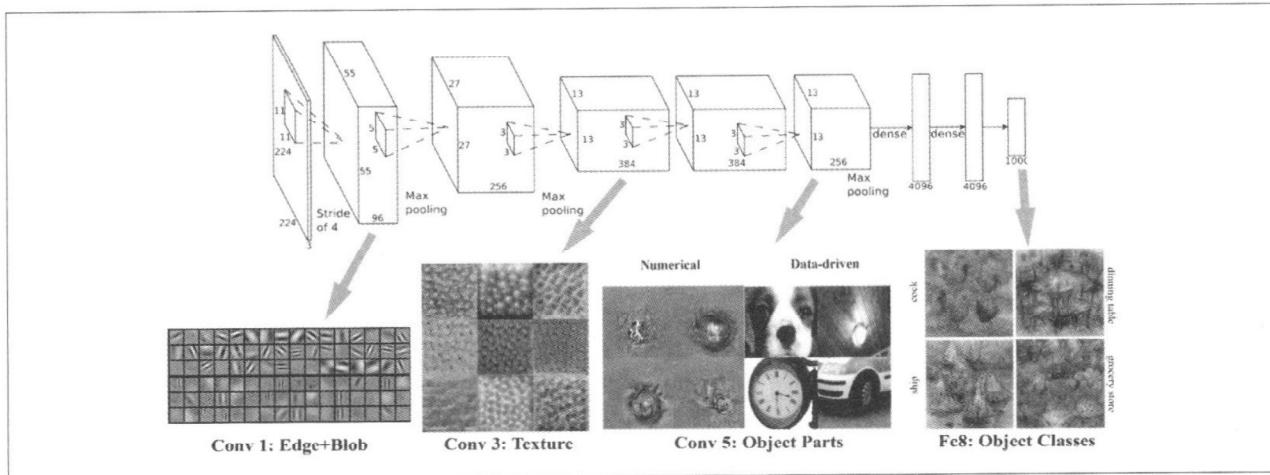


圖 7-26 利用 CNN 的卷積層擷取的資料。第 1 層是邊界及塊狀，第 3 層是紋理，第 5 層是物體的部分，利用最後的全連接層，神經元反應出物體的類別（狗或車等）（以上影像引用自文獻 [19]）

深度學習有趣之處在於，重疊多層卷積層，隨著層數加深，可以擷取出更複雜、抽象的資料，如圖 7-26 所示。最初的層級反應單純的邊界，接著反應紋理，然後反應複雜的物體部位。換句話說，隨著層數加深，神經元從單純的形狀，變化成「高階」的資料。就像瞭解物體的「意義」般，反應的對象會逐漸變化。

## 7.7 代表性的 CNN

CNN 提出了各種結構的網路。以下要介紹其中最重要的兩種網路。第一種是在 1998 年首度提出的 CNN 鼻祖 LeNet [20]。另一個是深度學習受到矚目之後，於 2012 年提出的 AlexNet [21]。

### 7.7.1 LeNet

LeNet 是 1998 年提出，用來進行手寫數字辨識的網路。如圖 7-27 所示，連續執行卷積層與池化層（正確來說是只有「分隔元素」的次取樣層），最後透過全連接層輸出結果。

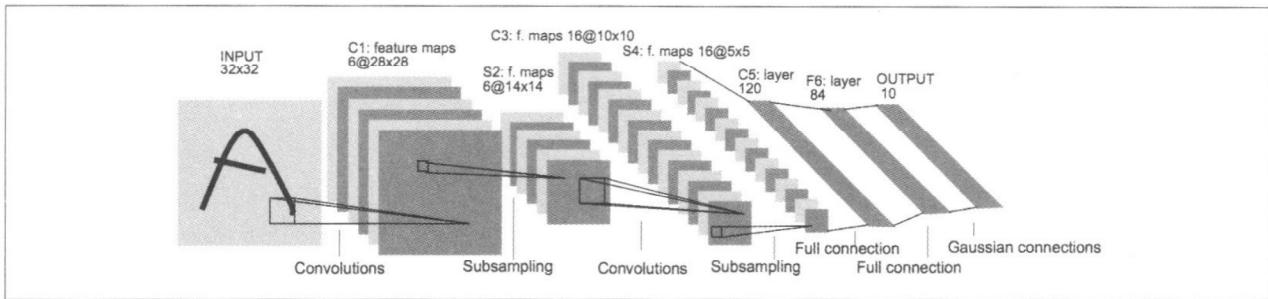


圖 7-27 LeNet 的網路結構（引用自文獻 [20]）

比較 LeNet 與「現在的 CNN」，可以發現有一些差異。第 1 個差異是活化函數。LeNet 使用的是 sigmoid 函數，而 CNN 以 ReLU 為主。另外，在自訂的 LeNet 中，會利用次取樣（subsampling）來縮小中間資料的大小，但是現在的 CNN 是以最大池化為主。

儘管 LeNet 與「現在的 CNN」有一些差異，但是兩者相去不遠。如果把 LeNet 當成是距今將近 20 年前提出的「首度 CNN」，應該會感到不可思議。

### 7.7.2 AlexNet

LeNet 問世之後，過了將近 20 幾年，才提出 AlexNet。AlexNet 在深度學習框架中，扮演著先驅者的角色。如圖 7-28 所示，基本上的網路結構與 LeNet 差不多。

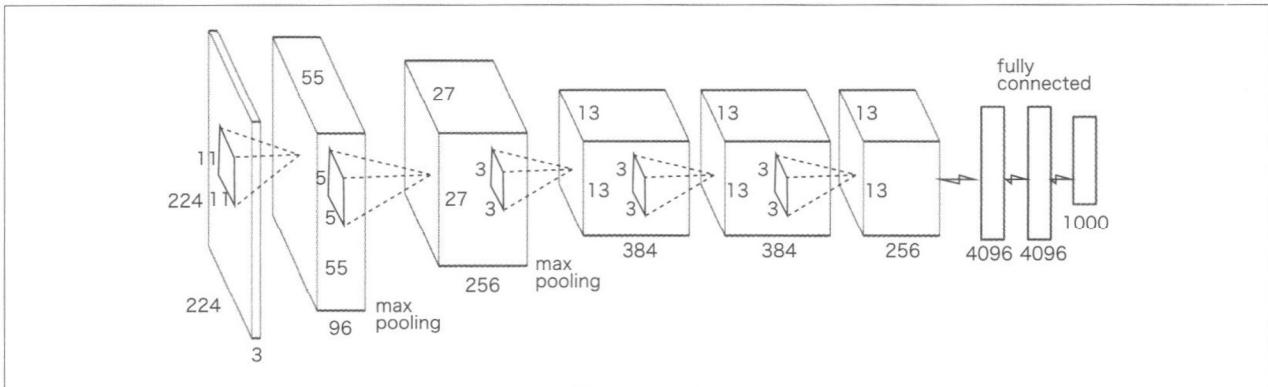


圖 7-28 AlexNet（參考文獻 [21] 製作而成）

AlexNet 同樣是重疊卷積層與池化層，最後經由全連接層輸出結果。結構與 LeNet 大同小異，但是 AlexNet 有以下這些相異點。

- 使用 ReLU 活化函數
- 使用 LRN（Local Response Normalization）局部性正規化層

- 使用 Dropout（請參考「6.4.3 Dropout」）

前面說明過，LeNet 與 AlexNet 的網路結構差別不大，但是周圍的環境與電腦技術卻有了顯著的進步。具體而言，就是任何人都可以取得大量資料。由於擅長大量平行運算的 GPU 普及，而能高速進行大量運算。因此，大數據與 GPU 成為推動深度學習發展的重要原動力。



深度學習（多層網路）通常都有許多參數。因此學習時，需要進行大量運算，而且還需要「滿足」這些參數的龐大資料。因此，GPU 與大數據可說是為解決這些問題帶來一道曙光。

## 7.8 重點整理

本章學習了 CNN，構成 CNN 的基本模組「卷積層」與「池化層」有些複雜，但是只要瞭解之後，後續就只剩如何運用的問題。這一章特別花時間說明如何執行卷積層與池化層。在處理影像的範疇，幾乎都可以使用 CNN，沒有例外。因此請徹底瞭解本章的內容，再進入最後一章。

### 本章學到的重點

- CNN 在全連結層的網路中，新加入了卷積層與池化層。
- 使用 im2col（將影像展開成陣列的函數），就可以輕鬆快速執行卷積層與池化層。
- 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 的代表性網路包括 LeNet 與 AlexNet。
- 大數據與 GPU 對深度發展有重要貢獻。