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

# 建立資料
x = np.arange(0, 6, 0.1) # 從 0 到 6，以 0.1 為單位產生資料
y = np.sin(x)

# 繪製圖表
plt.plot(x, y)
plt.show()
```

以下將利用 NumPy 的 arange 方法，產生 [0, 0.1, 0.2,…, 5.8, 5.9] 等資料，把這個部分當作 x。以這個 x 的各個元素為對象，套用 NumPy 的 sin 函數 np.sin()，把 x、y 的資料列提供給 plt.plot 方法，繪製圖表。最後，以 plt.show() 顯示圖表後結束。執行以上程式，會出現如**圖 1-3** 的影像。

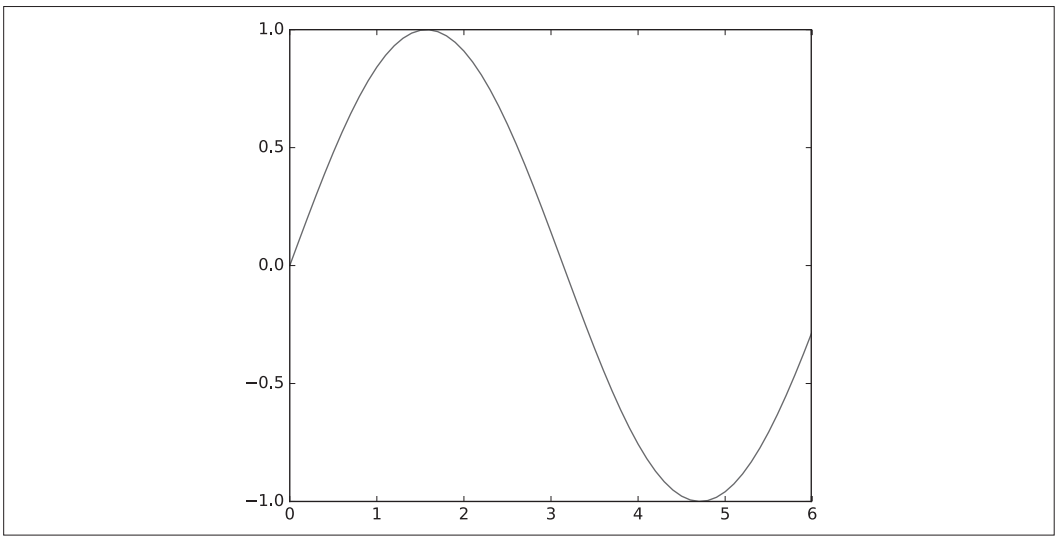

圖 1-3　sin 函數的圖表

1.6.2　pyplot 的功能

除了前面的 sin 函數，下面也試著用 cos 函數來繪圖，並且利用 pyplot 的其他功能，加上標題與 x 軸標籤名稱。

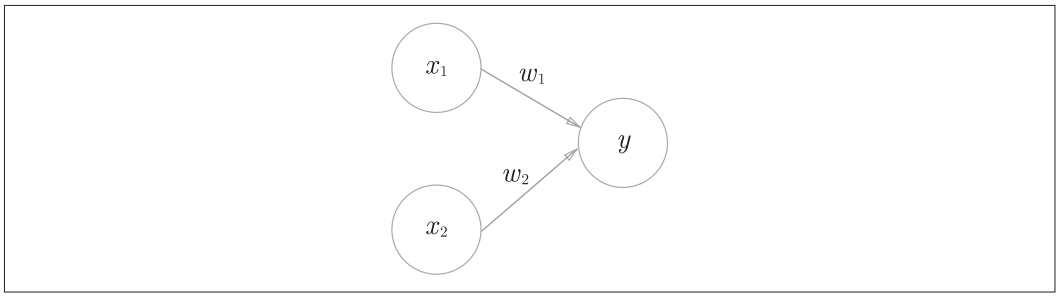

圖 2-1　接收 2 個輸入訊號的感知器

感知器的運作原理只有這一個！若用算式來顯示上述內容，可以變成以下式子（2.1）。

$$y = \begin{cases} 0 & (w_1x_1 + w_2x_2 \leqq \theta) \\ 1 & (w_1x_1 + w_2x_2 > \theta) \end{cases} \tag{2.1}$$

感知器擁有各輸入訊號的原有權重，而該權重是控制各訊號重要性的元素。換句話說，權重愈重，對應該權重的訊號重要性愈高。

權重相當於電流中的「電阻」。電阻是決定電流是否容易通過的參數，電阻愈低，通過的電流愈大。然而，感知器的權重是，數值愈大，通過的訊號愈大。就控制訊號傳遞（傳遞難度）而言，電阻與權重的作用是一樣的。

2.2　單純的邏輯電路

2.2.1　及閘（AND Gate）

接下來，要用感知器來思考一些簡單的問題。以下將以邏輯電路為主題，先思考什麼是及閘（AND Gate）。及閘是 2 輸入 1 輸出的閘道。圖 2-2 這種輸入訊號對應輸出訊號的表格，稱作「真值表」。及閘如圖 2-2 所示，當 2 個輸入為 1 時，才會輸出 1，其他是輸出 0。

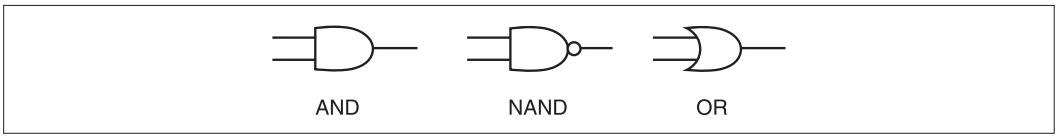

圖 2-9　代表及閘（AND Gate）、反及閘（NAND Gate）、或閘（OR Gate）的符號

如果要製作互斥或閘，該如何用及閘、反及閘、或閘來配線呢？請稍微動腦想看看。提示是分別在**圖 2-10** 的「？」代入及閘、反及閘、或閘其中一種，就可以完成互斥或閘。

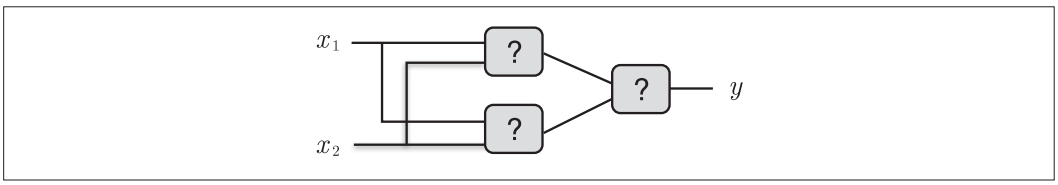

圖 2-10　在「？」插入及閘、反及閘、或閘其中一種，完成互斥或閘！

上一節說明過感知器的極限，正確來說，就是「單層感知器無法表現互斥或閘」或「單層感知器無法分離非線性區域」。由此可知，組合（層疊）感知器，就可以執行互斥或閘。

利用**圖 2-11** 的電路，即可形成互斥或閘。以下將以 x_1 與 x_2 代表輸入訊號，y 表示輸出訊號。x_1 與 x_2 是反及閘與或閘的輸入，反及閘與或閘的輸出會變成及閘的輸入。

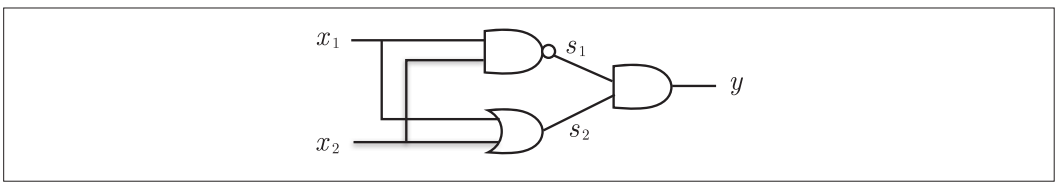

圖 2-11　搭配及閘、反及閘、或閘，完成互斥或閘

讓我們來確認，**圖 2-11** 的電路真的可以完成互斥或閘嗎？以下把反及閘的輸出當作 s_1，或閘的輸出為 s_2，試著填滿真值表。結果如**圖 2-12** 所示，檢視 x_1、x_2、y，就可以發現，的確變成互斥或閘的輸出了。

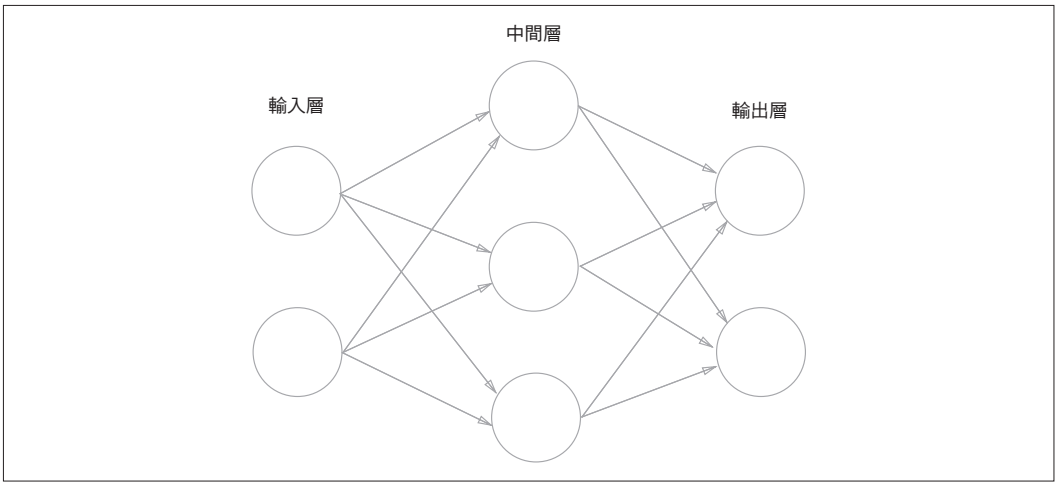

圖 3-1　神經網路的範例

圖 3-1 的網路是由 3 層構成。但是擁有權重的層數，其實只有 2 層，因此稱作「雙層網路」。部分書籍是根據構成網路的層數，把圖 3-1 稱作「三層網路」，請特別注意這一點。本書是將實質擁有權重的層數，亦即從輸入層、隱藏層、輸出層的總計數量減 1，來表示網路名稱。

單看圖 3-1，會覺得形狀與上一章的感知器一樣。事實上，神經元的連接方式，與上一章看到的感知器沒有什麼兩樣。那麼，神經網路如何傳達訊息呢？

3.1.2　複習感知器

接下來，要說明神經網路的訊號傳遞方法。在此之前，我想先從複習感知器開始。讓我們來思考圖 3-2 的網路結構。

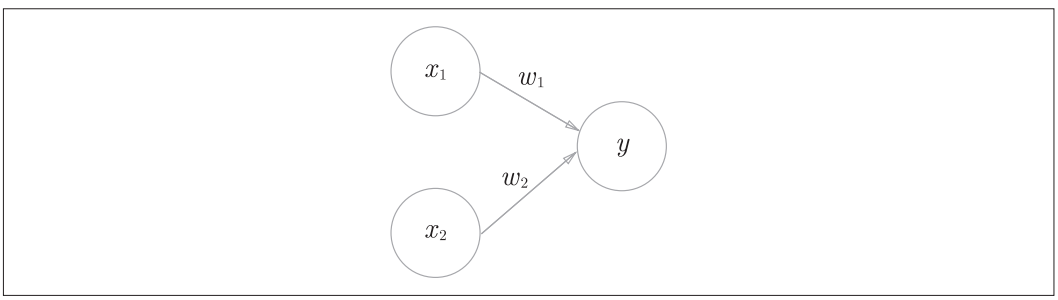

圖 3-2　複習感知器

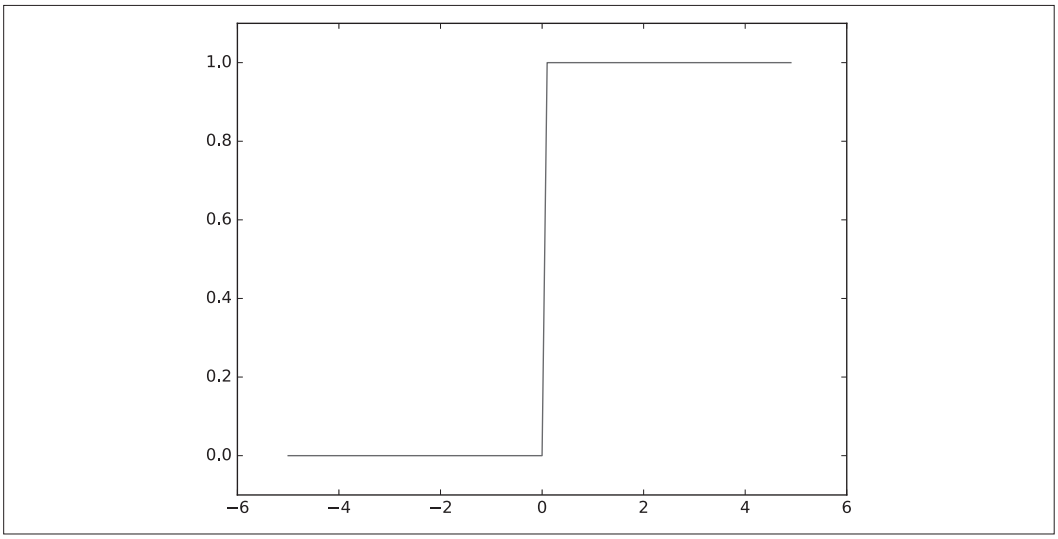

圖 3-6　階梯函數的圖表

如圖 3-6 所示，階梯函數是以 0 為界線，輸出從 0 切換到 1（或從 1 切換到 0）。由於切換數值會形成階梯狀，才命名為階梯函數，如圖 3-6 所示。

3.2.4　執行 sigmoid 函數

接下來，要執行 sigmoid 函數。算式（3.6）的 sigmoid 函數，在 Python 中，可以寫成以下這樣。

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

這裡的 np.exp(-x) 是對應算式的 $\exp(-x)$。這個部分並不困難，但是請注意到，在引數 x 輸入 NumPy 陣列，仍可以計算出正確的結果。事實上，在 sigmoid 函數中，輸入 NumPy 陣列後，會執行以下正確的運算。

```
>>> x = np.array([-1.0, 1.0, 2.0])
>>> sigmoid(x)
array([ 0.26894142,  0.73105858,  0.88079708])
```

sigmoid 函數對應 NumPy 陣列的處理，祕密就藏在 NumPy 的廣播中（詳細說明請參考「1.5.5 廣播」）。利用廣播的功能，執行純量（scalar value）與 NumPy 陣列的運算時，是以純量與 NumPy 陣列的各個元素之間來進行運算。以下舉個具體的範例來說明。

```
>>> t = np.array([1.0, 2.0, 3.0])
>>> 1.0 + t
array([ 2.,  3.,  4.])
>>> 1.0 / t
array([ 1.        ,  0.5       ,  0.33333333])
```

在上述的範例中，純量（這個範例是指 1.0）與 NumPy 陣列之間，進行數值運算（+ 或 / 等）。結果，純量與 NumPy 陣列的各個元素進行運算，運算結果輸出成 NumPy 陣列。剛才提到的 sigmoid 函數，是由 np.exp(-x) 產生 NumPy 陣列，所以 1 / (1 + np.exp(-x)) 的結果，是 NumPy 陣列的各個元素之間進行運算。

接下來，將 sigmoid 函數繪製成圖表。繪圖用的程式和前面的階梯函數幾乎一樣，唯一的差別在於，輸出 y 的函數改變成 sigmoid 函數。

```
x = np.arange(-5.0, 5.0, 0.1)
y = sigmoid(x)
plt.plot(x, y)
plt.ylim(-0.1, 1.1) # 設定 y 軸的範圍
plt.show()
```

執行上述原始碼，可以得到**圖 3-7** 的圖表。

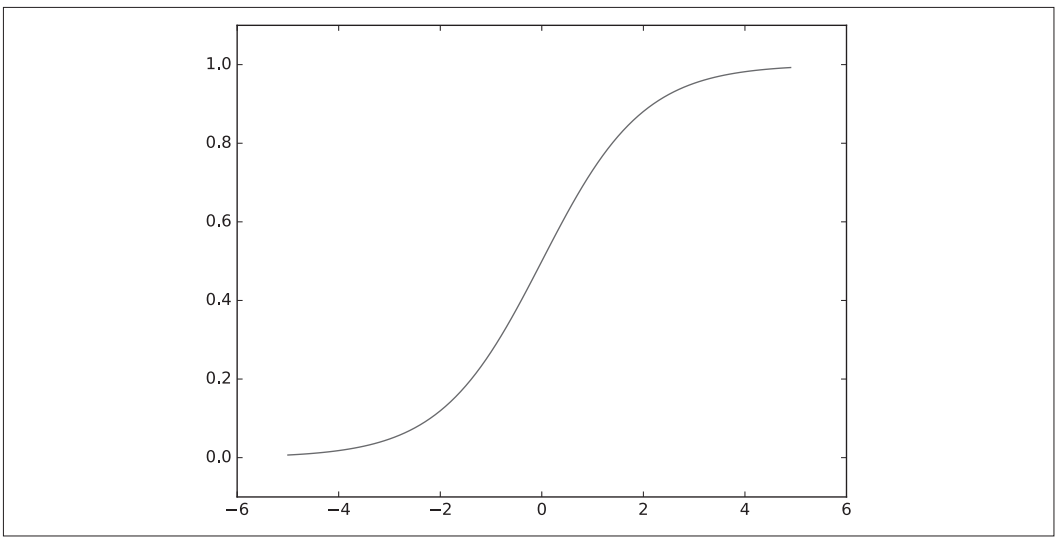

圖 3-7　sigmoid 函數的圖表

3.2.5 比較 sigmoid 函數與階梯函數

讓我們來比較一下 sigmoid 函數與階梯函數。階梯函數與 sigmoid 函數如圖 3-8 所示。這兩種函數的差別是什麼？有何共通性？請試著觀察圖 3-8，思考看看。

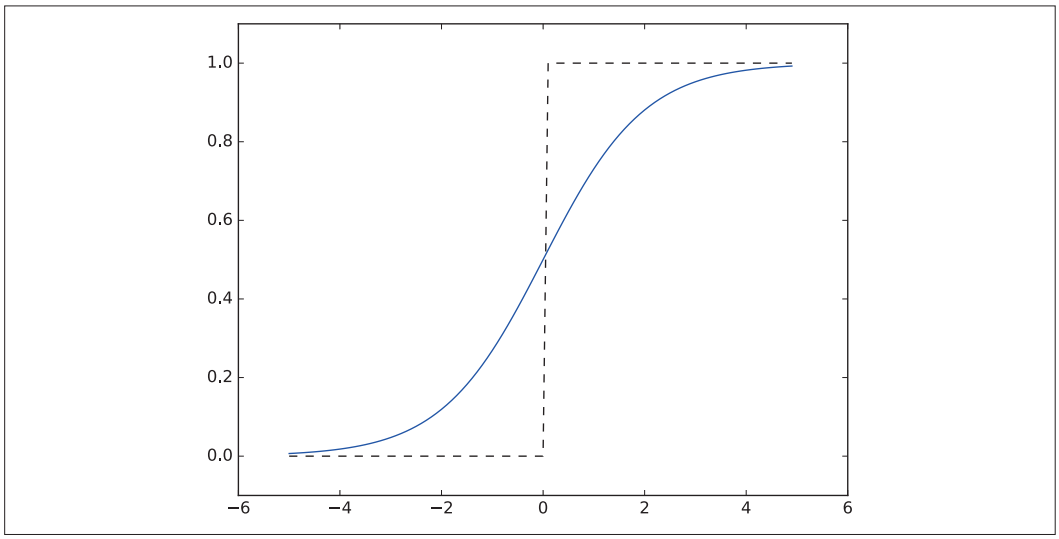

圖 3-8 階梯函數與 sigmoid 函數（虛線是階梯函數）

檢視圖 3-8，首先注意到的部分，應該是「平滑度」的差異。sigmoid 函數是平滑曲線，針對輸入產生連續性的輸出；然而，階梯函數是以 0 為界線，明確改變輸出。學習神經網路時，sigmoid 函數的平滑度具有重要的意義。

另外，（與前面提到的平滑度有關）階梯函數只能回傳 0 或 1 其中一個值，而 sigmoid 函數可以回傳實數，例如 0.731…或 0.880…等，這點也不一樣。換句話說，在感知器中，傳遞著 0 或 1 這兩個值的訊號，但是在神經網路中，傳遞的是連續性的實數訊號。

階梯函數好比是「鹿威」（譯註：又稱添水，在日本，這是由竹子製成，藉由流水通過發出的響聲來驅趕動物的裝置），而 sigmoid 函數就像是「水車」。相對於階梯函數的動作類似「鹿威」，只有水流過與不流過（0 或 1）等兩個動作；sigmoid 函數如同「水車」一樣，依照流過去的水量比例，調整下次流過的水量。

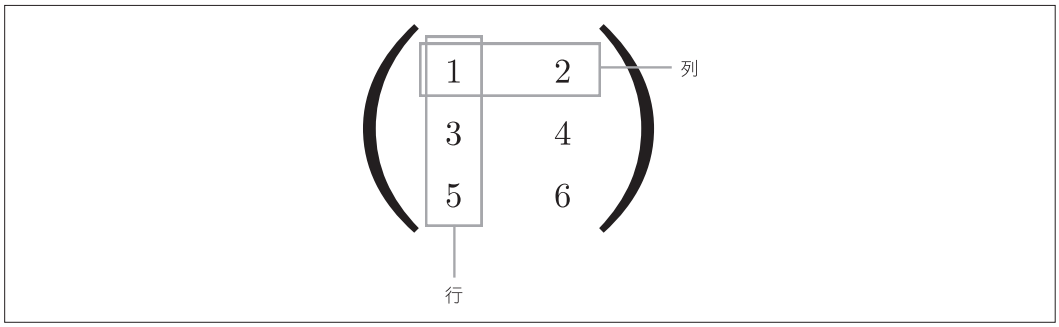

圖 3-10　水平排列稱作「列」，垂直排列稱作「行」

3.3.2　矩陣的乘積

接下來，要說明矩陣（二維陣列）的乘積。矩陣乘積是指，假設 2×2 的矩陣，會執行如圖 3-11 的運算（定義成依照以下步驟計算）。

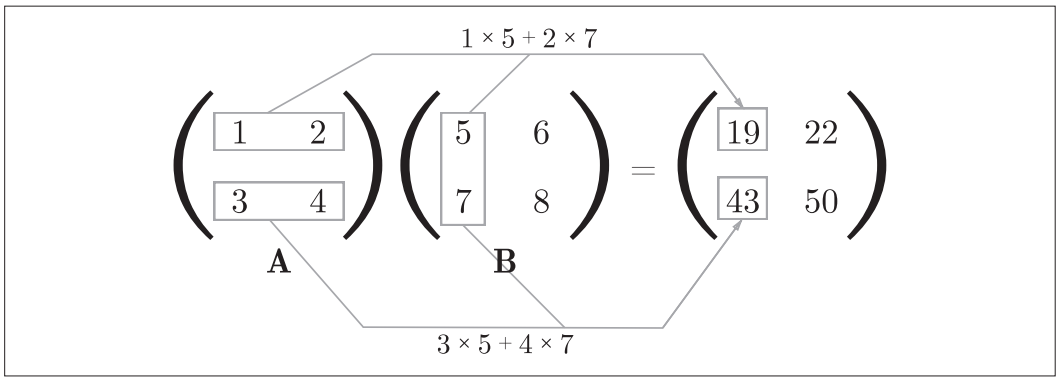

圖 3-11　計算矩陣的積

如這個範例所示，矩陣的乘積是，利用左邊矩陣的列（水平方向）與右邊矩陣的行（垂直方向）之間，各個元素的積與和來進行運算。運算結果將儲存成新多維陣列的元素。例如，**A** 的第 1 列與 **B** 的第 1 行之乘積結果為第 1 列第 1 行的元素，**A** 的第 2 列與 **B** 的第 1 行會變成第 2 列第 1 行的元素。另外，在本書的算式顯示中，以粗體字表示矩陣。例如，矩陣顯示成 **A**，與純量（例如 a 或 b）的元素做區隔。接下來，利用 Python 執行運算結果，如下所示。

4.1.1 驅動資料

機器學習的核心是資料。從資料中尋找答案，從資料中找出類型，從資料中說故事，這些都是利用機器學習來進行，沒有資料，一切無從開始。因此，機器學習的核心，就是「資料」。這種驅動資料的方法，擺脫了以「人」為中心的手法。

一般要解決某個問題時，尤其必須找出某個類型時，通常是由人類絞盡腦汁思考，找出答案。「這個問題看起來有這樣的規則性吧！」、「不，原因可能出在其他場所」像這樣，把人類的經驗或直覺當作線索，嘗試錯誤，試著找出答案。然而，機器學習的手法是盡量避免人為介入，試著從收集到的資料中，找出答案（類型）。神經網路或深度學習除了原本機器學習使用的手法，還擁有可以避免人為介入的重要性質。

讓我們來思考一個具體的問題。假設要執行可以辨識「5」這個數字的程式。「5」這個數字是一張手寫影像，如**圖 4-1** 所示，因此目標是執行可以分辨 5 或不是 5 的程式。這個問題看起來很單純，但是可以使用何種演算法呢？

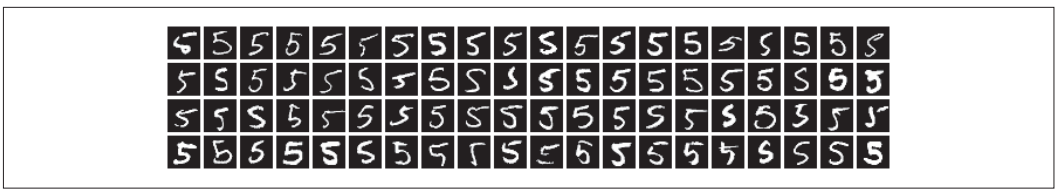

圖 4-1　手寫數字「5」的範例：每個人的筆跡（寫法）都不同

如果你打算自行思考，設計出能正確分類「5」的程式，就會發現，這個問題沒有想像中容易。我們可以輕鬆辨識「5」，但是要清楚描述按照何種規則來辨識「5」，卻非常困難。另外，檢視**圖 4-1** 就會發現，每個人的筆跡不同，找出「5」的規則，其實是十分艱鉅的工作，非常花時間。

與其「琢磨」出從零開始辨識「5」的演算法，倒不如有效運用資料，找出解決之道。其中一種方法，就是從影像中擷取**特徵量**，利用機器學習技術，思考學習該特徵量類型的方法。這裡所謂的特徵量是指，可以從輸入資料（輸入影像）中，精準擷取出本質資料（重要資料）的轉換器。影像的特徵量常當作向量來描述。在電腦視覺領域中，知名的特徵量包括 SIFT、SURF、HOG 等。使用這種特徵量，將影像資料轉換成向量，就可以利用機器學習用的辨識器 SVM、KNN 等，學習轉換後的向量。

這種機器學習的方法，是由「機器」從收集到的資料中，找出規則性。相較於從零開始思考演算法，較能有效率地解決問題，也可以減輕對「人」的負擔。但是，必須注意到，把影像轉換成向量時使用的特徵量，還是由「人」設計。也就是說，如果沒有根據問題，使用適當的特徵量（或沒有設計特徵量），就無法獲得良好的結果。例如，辨識狗臉與辨識「5」的特徵量不同，必須由人來思考其他的特徵量。換句話說，即使是利用特徵量與機器學習的方法，仍得配合問題，由「人」來決定適合的特徵量。

到目前為止，針對機器學習問題，說明了兩種方法。用圖來表示這兩種方法，結果如**圖 4-2** 上半部所示。相對而言，神經網路（深度學習）的方法，如**圖 4-2** 下半部所示，以沒有人類介入的一個區塊來顯示。

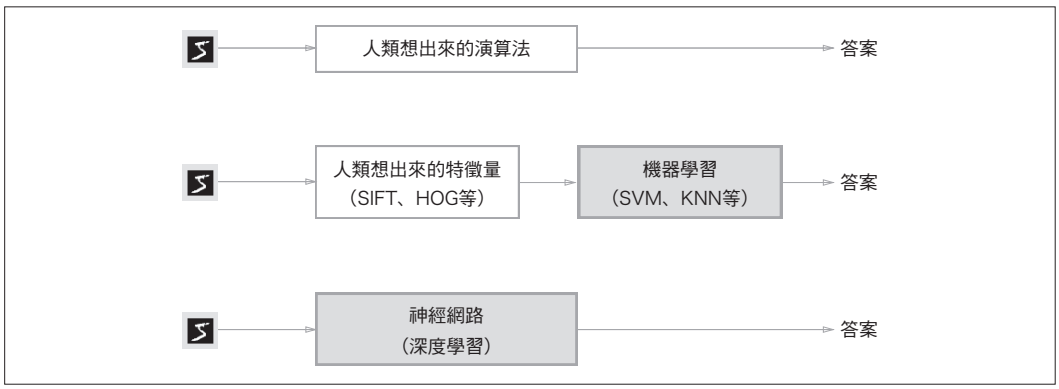

圖 4-1　從「人工」建立原則到「機器」自動從資料中學習的典範轉移過程：以灰色顯示沒有人類介入的區塊

如**圖 4-2** 所示，神經網路是「原封不動」學習影像。第 2 種方法，亦即利用特徵量與機器學習的範例，是由人來設計特徵量，但是神經網路連包含在影像中的重要特徵量，也是由「機器」來學習。

深度學習稱作「end-to-end machine learning」，這裡的 *end-to-end* 是指，「從其中一端到另一端」，代表從未處理的資料（輸入）中，獲得目的結果（輸出）之意。

神經網路的優點是，全部的問題都可以透過相同流程來解決。例如，不管要解決的問題是辨識「5」或者辨識「狗」，還是辨識「人臉」，神經網路都會徹底學習取得的資料，試著找出問題的類型。換句話說，神經網路面對任何問題，都能把資料視為未處理資料，以「end-to-end」的方式來學習。

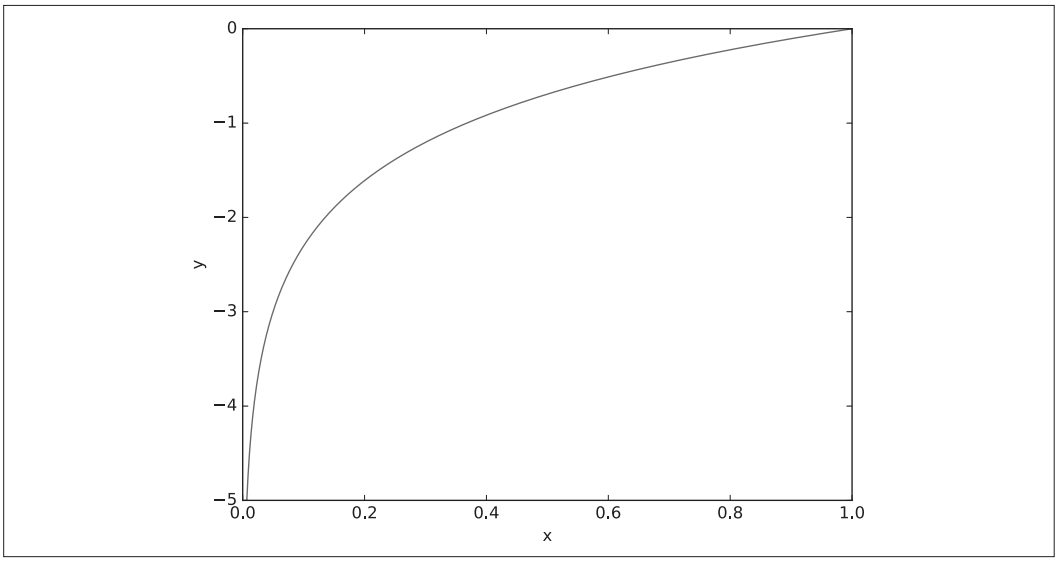

圖 4-3　自然對數 $y=\log x$ 的圖表

如**圖 4-3** 所示，x 為 1 的時候，y 等於 0，隨著 x 趨近於 0，y 值將會愈來愈小。因此，算式（4.2）對應正確答案標籤的輸出愈大，愈趨近於 0。當輸出為 1 的時候，交叉熵誤差變成 0。另外，對應正確答案標籤的輸出愈小，算式（4.2）的值愈大。

接下來，要實際執行交叉熵誤差。

```
def cross_entropy_error(y, t):
    delta = 1e-7
    return -np.sum(t * np.log(y + delta))
```

引數 y 與 t 是 NumPy 陣列，進行 np.log 運算時，要先加上微小值 delta 再計算。因為當 np.log(0) 時，np.log(0) 會變成代表負無限大的 -inf，這樣就無法繼續運算，所以才會加上微小值，防範發生負無限大的情形。接下來，讓我們利用 cross_entropy_error(y, t)，進行簡單的運算。

```
>>> t = [0, 0, 1, 0, 0, 0, 0, 0, 0, 0]
>>> y = [0.1, 0.05, 0.6, 0.0, 0.05, 0.1, 0.0, 0.1, 0.0, 0.0]
>>> cross_entropy_error(np.array(y), np.array(t))
0.51082545709933802
>>>
>>> y = [0.1, 0.05, 0.1, 0.0, 0.05, 0.1, 0.0, 0.6, 0.0, 0.0]
>>> cross_entropy_error(np.array(y), np.array(t))
2.3025840929945458
```

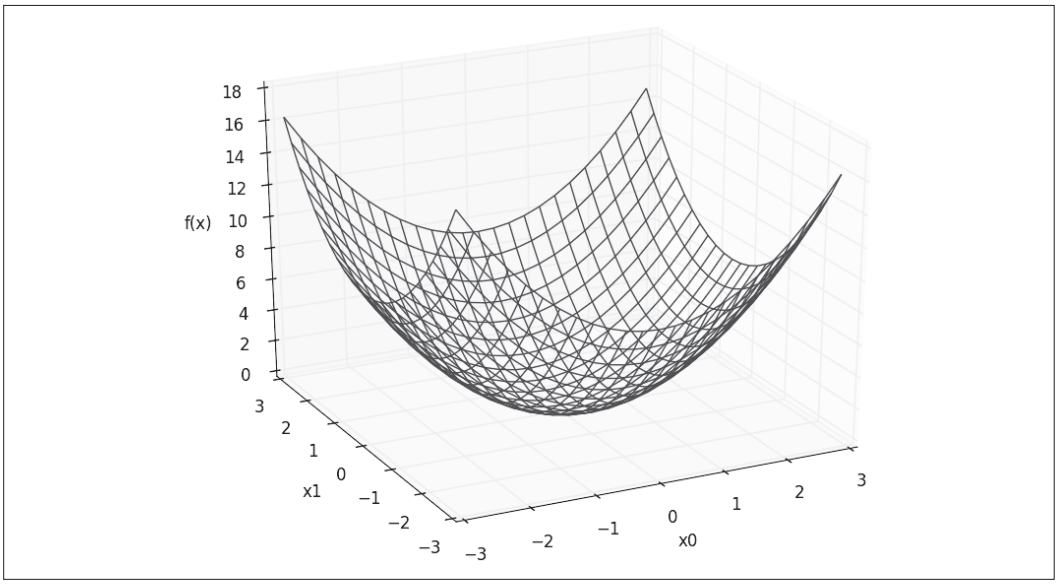

圖 4-8 $f(x_0, x_1) = x_0^2 + x_1^2$**的圖表**

接著，要計算出算式（4.6）的微分。這裡必須注意到，算式（4.6）有 2 個變數。因此，「要對哪個變數進行微分？」亦即必須區分要計算 x_0 與 x_1 哪一個變數的微分。另外，由多個變數形成的函數微分，稱作**偏微分**。用算式來顯示偏微分，可以寫成 $\frac{\partial f}{\partial x_0}$、$\frac{\partial f}{\partial x_1}$。

如何才能計算出偏微分？以下將實際試著解開 2 個偏微分的問題。

問題 1：當 $x_0 = 3$、$x_1 = 4$ 時，計算 x_0 的偏微分 $\frac{\partial f}{\partial x_0}$。

```
>>> def function_tmp1(x0):
...     return x0*x0 + 4.0**2.0
...
>>> numerical_diff(function_tmp1, 3.0)
6.00000000000378
```

問題 2：當 $x_0 = 3$、$x_1 = 4$ 時，計算 x_1 的偏微分 $\frac{\partial f}{\partial x_1}$。

```
>>> def function_tmp2(x1):
...     return 3.0**2.0 + x1*x1
...
>>> numerical_diff(function_tmp2, 4.0)
7.999999999999119
```