

誤差反向傳播法

上一章說明了神經網路的學習。此時，神經網路權重參數的梯度，正確來說，與權重參數有關的損失函數梯度，是利用數值微分計算出來。數值微分很簡單，執行起來也不難，缺點是運算比較花時間。而本章要學習的「誤差反向傳播法」，是能以良好效率計算出權重參數梯度的方法。

筆者（個人）認為，要正確瞭解誤差反向傳播法有兩種方法。一種是利用「算式」，另一種是用「計算圖 (*computational graph*)」。前者是一般的方法，尤其多數與機器學習有關的書籍，都是以算式為主來說明。的確，利用算式來說明，是最嚴謹又簡潔的方法。可是立刻就以算式來思考，可能忽略掉本質，或迷失在算式中。因此，本章要利用計算圖，以「視覺化」方式，說明誤差反向傳播法。實際撰寫程式，可以加深理解，讓你覺得「豁然開朗」。

利用計算圖來說明誤差反向傳播法的點子，其實是參考 Andrej Karpathy 的部落格 [4]，以及他和 Fei-Fei Li 教授在史丹佛大學開辦的深度學習課程「CS231n」[5]。

5.1 計算圖

計算圖是利用圖表呈現計算過程。這裡所謂的圖，是指當作資料結構的圖，利用多個節點與邊線來呈現（連接節點的直線稱作「邊線」）。本節利用簡單的問題，讓你熟悉什麼是計算圖。從簡單的問題開始，逐步說明，最後介紹誤差反向傳播法。

5.1.1 用計算圖解答

接下來，要使用「計算圖」解開簡單的問題。我們要作答的題目是用心算就可以算出答案的簡單問題，但是這裡的目的是讓你熟悉計算圖。學會計算圖的用法，就能在後續複雜的計算中，發揮威力，因此請一定要在此徹底學會計算圖。

問題 1：假設太郎在超市買了 2 顆每顆單價為 100 元的蘋果。請計算出要支付的金額。此外，營業稅是 10%。

計算圖是利用節點與箭頭來呈現計算過程。節點以○顯示，在○之中寫上運算內容。另外，把計算途中的結果，寫在箭頭上方，各個節點的計算結果以左往右傳送來代表。用計算圖解答問題 1，結果如圖 5-1 所示。

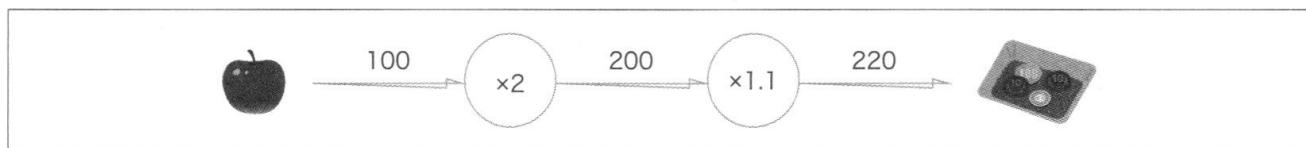


圖 5-1 利用計算圖回答問題 1

如圖 5-1 所示，最初的蘋果 100 元流向「 $\times 2$ 」的節點，變成 200 元，然後再傳給下個節點。接下來，200 元流向「 $\times 1.1$ 」的節點，變成 220 元。因此，從計算圖的結果，可以知道答案是 220 元。

另外，在圖 5-1 中，用○把「 $\times 2$ 」或「 $\times 1.1$ 」當作一個演算，但是也可以用○來表示乘法「 \times 」運算。此時，如圖 5-2 所示，「2」與「1.1」分別代表「蘋果的顆數」及「營業稅」等變數，顯示在○之外。

接著是下一個問題。

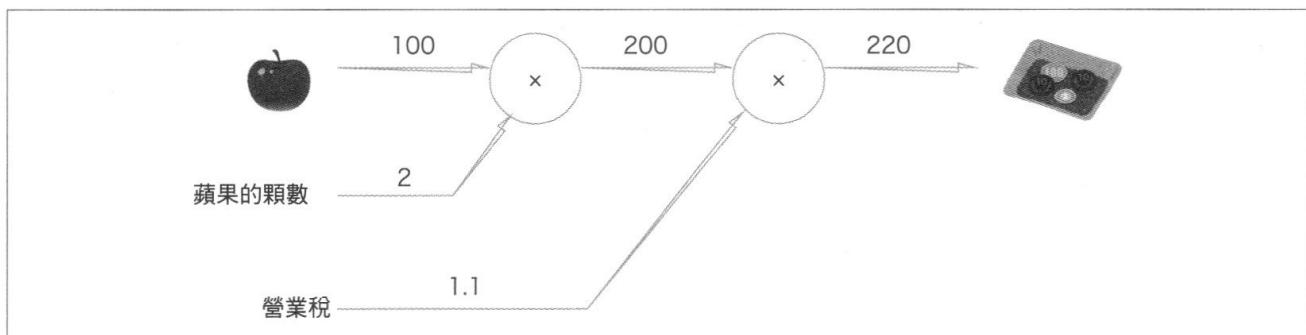


圖 5-2 利用計算圖解答問題 1：把「蘋果的顆數」與「營業稅」當作變數，顯示在○之外

問題 2：太郎在超市買了 2 顆蘋果與 3 顆橘子。蘋果每顆 100 元，橘子每顆 150 元，假設營業稅為 10%，請計算出要支付的金額。

和問題 1 一樣，利用計算圖解開問題 2。計算圖如圖 5-3 所示。

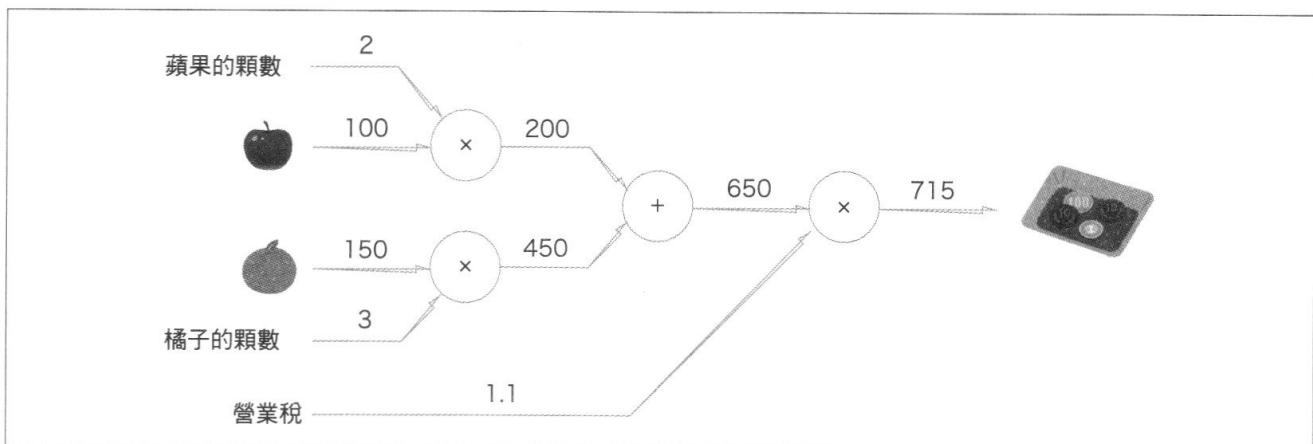


圖 5-3 利用計算圖回答問題 2

在這個問題中，新加入了加法節點「+」，合計蘋果與橘子的金額。畫出計算圖之後，由左往右開始計算。請想像成電流通過電路般，由左往右傳遞計算結果。當計算結果到達最右邊時，就在此結束。圖 5-3 的答案是 715 元。

和前面一樣，使用計算圖解答問題是依照以下流程在進行。

1. 建立計算圖
2. 在計算圖上，由左往右進行計算

第 2 個「由左往右進行計算」的步驟，稱作正向傳播 (*forward propagation*)。正向傳播是從計算圖的起點開始往終點方向傳播。既然有正向傳播，應該也可以想到反方向也可以傳播吧！用圖來解釋，就是由右往左的傳播。事實上，這種傳播稱作反向傳播 (*backward propagation*)。反向傳播在前面計算微分時，發揮著重要的作用。

5.1.2 局部性計算

計算圖的特色是，利用傳播「局部性計算」的方式，可以獲得最終的結果。局部性的意思是「與自己有關的小範圍」。局部性計算是指，不論整體執行了什麼，都可以從與自己有關的資料中，輸出下個結果（之後的結果）。

以下舉個局部性計算的具體範例來說明。假設在超市買了 2 顆蘋果，其餘也買了很多東西。此時，可以畫出如圖 5-4 的計算圖。

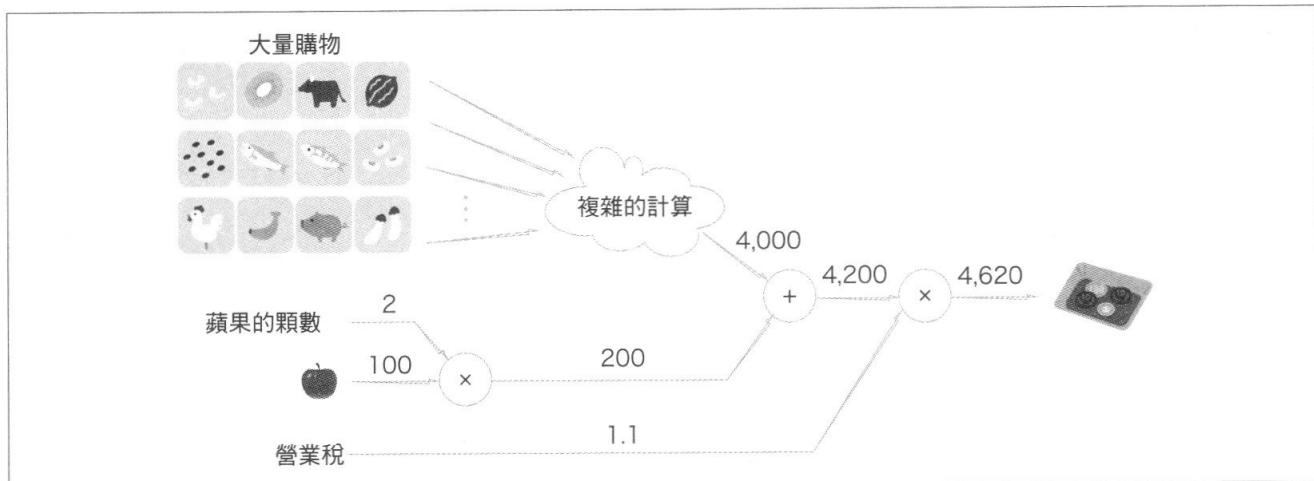


圖 5-4 購買 2 顆蘋果及其他大量物品的範例

如圖 5-4 的計算圖所示，大量購物（經過複雜的計算）之後，金額為 4,000 元。這裡的重點是，在各個節點的計算，都屬於局部性計算。例如，蘋果與其他物品的購物金額合計 ($4,000 + 200 \rightarrow 4,200$)，不用思考 4,000 這個數字是如何計算出來的，只要想成是 2 個數字相加即可。換句話說，各節點應該做的是，進行與自己有關的計算，在這個範例中，指的是將輸入的 2 個數字相加，不需要考量整體。

這樣就能在計算圖中，集中精神在局部性計算上。即使整體計算非常複雜，各個步驟做的事情，也只有目標節點的「局部性計算」。局部性計算很單純，但是藉由傳遞結果的方式，就可以獲得構成整體的複雜計算結果。



例如，組裝汽車是非常複雜的工作，通常會利用「生產線」來分工合作。各負責人員（負責的機器）執行單純化工作，將工作成果交給下個負責人員，最後製作出車子。計算圖也一樣，把複雜的計算分割成「單純的局部性計算」，像生產線一樣，把計算結果傳遞給下個節點。將複雜計算分解之後，也是由單純計算構成的，這一點跟組裝汽車很類似。

5.1.3 為什麼用計算圖來解答？

到目前為止，使用了計算圖解答了 2 個問題，但是計算圖的優點是什麼？其中一個優點是，剛才說明過的「局部性計算」。就算整體計算非常複雜，利用局部性計算，讓各個

節點專注於單純計算，就能將問題單純化。另外，還有一個優點是，利用計算圖，可以維持所有過程中的計算結果（例如，計算 2 顆蘋果的金額是 200 元，加上營業稅之前的金額是 650 元）。可是，這些理由可能還不足以說服你，為什麼要使用計算圖來解答。事實上，使用計算圖最大的原因是，利用反向傳播，可以有效計算「微分」。

在說明計算圖的反向傳播之前，讓我們再次思考問題 1。問題 1 是要計算購買 2 顆蘋果，含營業稅後，最後支付的金額。假設我們想知道，當蘋果的價格上漲時，最後的支付金額會產生何種變化。這等於要計算「與蘋果價格有關的支付金額微分」。用符號來表示，蘋果的價格是 x ，支付金額為 L 時，相當於要計算 $\frac{\partial L}{\partial x}$ 。這是代表當蘋果的價格「略微」上漲時，支付金額會增加多少？

前面說明過，利用計算圖進行反向傳播，就可以計算出「與蘋果價格有關的支付金額微分」。只先顯示結果，即可像圖 5-5 一樣，利用計算圖上的反向傳播，計算微分（如何進行反向傳播，以下馬上就會說明）。

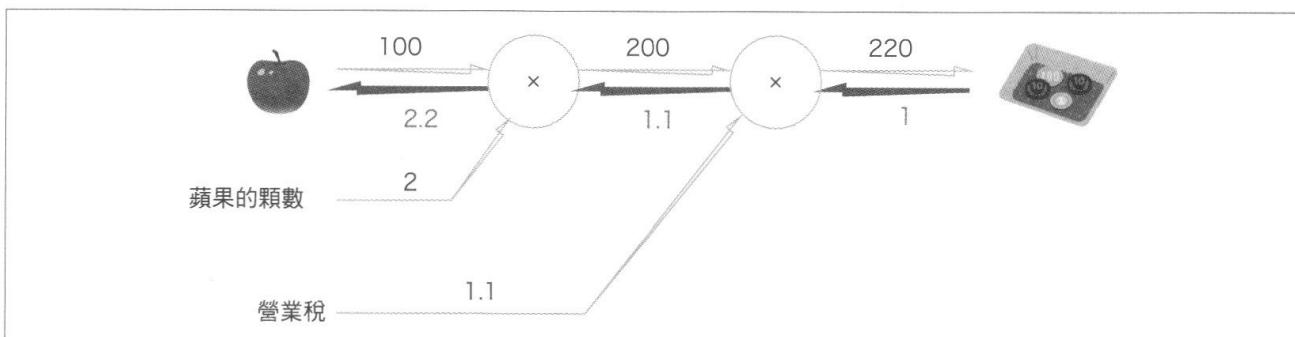


圖 5-5 利用反向傳播傳遞微分值

如圖 5-5 所示，反向傳播是利用與正向傳播相反的箭頭（粗線）來顯示。反向傳播是傳遞「局部性微分」，並且將微分值寫在箭頭的下方。此時，反向傳播是由右到左「 $1 \rightarrow 1.1 \rightarrow 2.2$ 」傳遞微分值。從這個結果中，可以說「與蘋果價格有關的支付金額微分」值，就是 2.2。這是代表蘋果每上漲 1 元，最後支付金額會增加 2.2 元（正確來說，蘋果的價格增加了某個微小值之後，最終增加的金額是該微小值的 2.2 倍）。

這裡只計算了與蘋果價格有關的微分，依照相同的步驟，也可以計算出「與營業稅有關的支付金額微分」或「與蘋果顆數有關的支付金額微分」。此時，過程中計算出來的微分（傳遞到中途的微分）結果可以共享，能有效率地計算多個微分。這種計算圖的優點，就是利用正向傳播與反向傳播，快速計算出各變數的微分值。

5.2 連鎖律

前面進行的計算圖正向傳播是把計算結果往正方向，亦即由左向右傳遞。此時，進行的計算和平常一樣，所以感覺很自然。然而，在反向傳播中，和正向傳播相反，由右向左傳遞「局部性微分」（最初你可能會感到疑惑）。傳遞「局部性微分」的原理是依靠連鎖律（chain rule）。因此，接下來要說明連鎖律，這樣就能清楚瞭解，這是對應計算圖上的反向傳播。

5.2.1 計算圖的反向傳播

以下立刻舉一個使用計算圖的反向傳播範例。假設要計算 $y=f(x)$ ，其反向傳播如圖 5-6 所示。

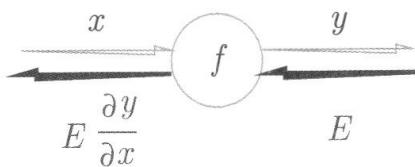


圖 5-6 計算圖的反向傳播：在正向與反向，乘上局部性微分

如圖 5-6 所示，反向傳播的計算步驟是，針對訊號 E ，乘上節點的局部性微分 ($\frac{\partial y}{\partial x}$)，接著傳遞給下一個節點。這裡所謂的局部性微分是指，計算在正向傳播中 $y=f(x)$ 的微分，這代表要計算出與 x 有關 y 的微分 ($\frac{\partial y}{\partial x}$)。假設 $y=f(x)=x^2$ ，就是 $\frac{\partial y}{\partial x}=2x$ 。將局部性微分乘以上層傳遞過來的值（這個範例是指 E ），再傳遞給前面節點。

這是用反向傳播進行計算的步驟，可以有效計算出目標微分值，也是反向傳播的重點。我們可以利用連鎖律的原理，說明這種做法可不可行。因此，下面要說明何謂連鎖律。

5.2.2 何謂連鎖律？

在說明連鎖律之前，必須先從合成函數開始介紹。合成函數是指，由多個函數構成的函數。例如， $z=(x+y)^2$ 是由 2 個式子構成，如以下算式（5.1）所示。

$$\begin{aligned} z &= t^2 \\ t &= x + y \end{aligned} \tag{5.1}$$

連鎖律是指，合成函數的微分性質，而且定義如下。

用合成函數顯示某個函數時，該合成函數的微分可以利用構成合成函數的各個函數微分之積來表示。

這就是連鎖律的原理（或許你會覺得好像很難）其實是非常簡單的性質。以算式(5.1)為例來說明， $\frac{\partial z}{\partial x}$ （與 x 有關的 z 微分）可以利用 $\frac{\partial z}{\partial t}$ （與 t 有關的 z 微分）與 $\frac{\partial t}{\partial x}$ （與 x 有關的 t 微分）之積來表示。用算式顯示，結果如算式(5.2)所示。

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial t} \frac{\partial t}{\partial x} \quad (5.2)$$

算式(5.2)剛好如以下所示， ∂t 「抵銷」，所以能輕鬆記住。

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\cancel{\partial t}} \cancel{\frac{\partial t}{\partial x}}$$

接著，使用連鎖律，求出算式(5.2)的微分 $\frac{\partial z}{\partial x}$ 。在此之前，要先計算出算式(5.1)的局部性微分（偏微分）。

$$\begin{aligned} \frac{\partial z}{\partial t} &= 2t \\ \frac{\partial t}{\partial x} &= 1 \end{aligned} \quad (5.3)$$

如算式(5.3)所示， $\frac{\partial z}{\partial t}$ 是 $2t$ ，所以 $\frac{\partial t}{\partial x}$ 是 1 。這是解析微分的公式，所計算出來的結果。最後利用算式(5.3)的微分之積算出 $\frac{\partial z}{\partial x}$ 。

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial t} \frac{\partial t}{\partial x} = 2t \cdot 1 = 2(x + y) \quad (5.4)$$

5.2.3 連鎖律與計算圖

接下來，以計算圖顯示算式(5.4)計算的連鎖律。2次方用「**2」的節點來表示，可以畫出如圖5-7所示的計算圖。

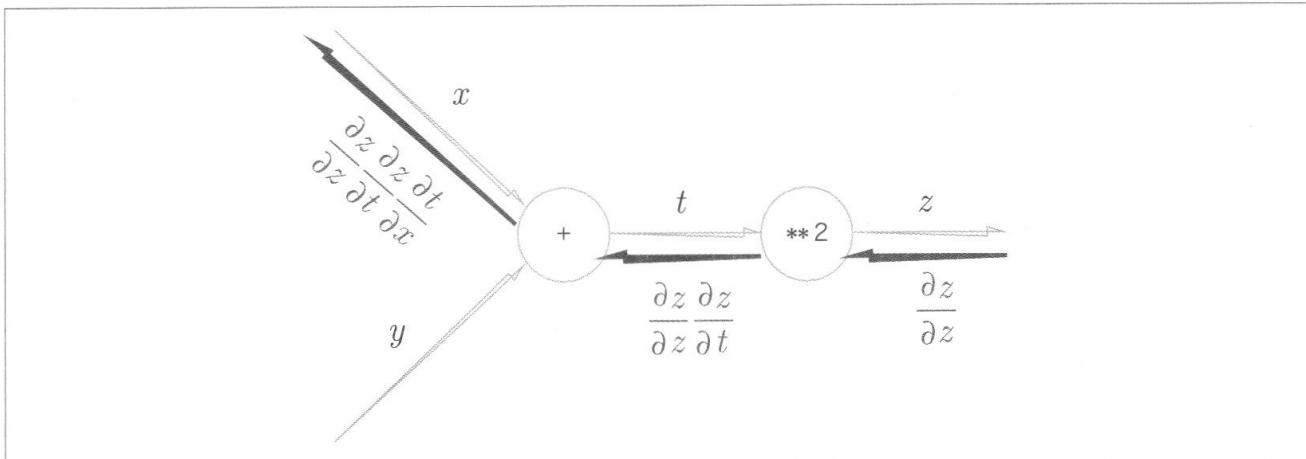
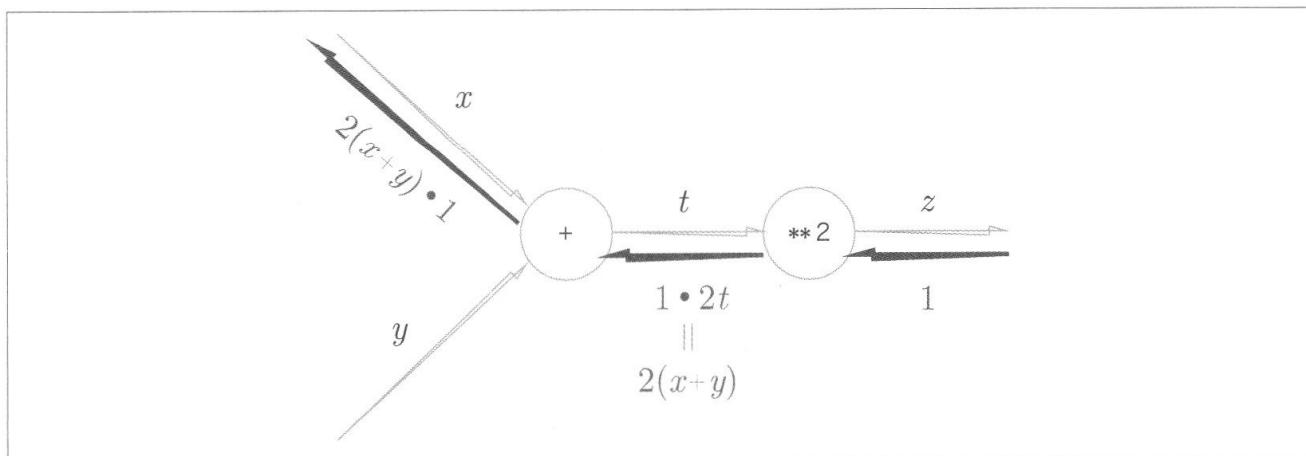


圖 5-7 算式 (5.4) 的計算圖：往正向與反向乘上局部性微分再傳遞

如圖 5-7 所示，計算圖的反向傳播是由右往左傳遞訊號。在反向傳播的計算步驟中，針對節點的輸入訊號，乘上節點的局部性微分（偏微分），再傳遞給下一個節點。例如，傳給「**2」的反向傳播輸入是 $\frac{\partial z}{\partial z}$ ，因此乘上局部性微分 $\frac{\partial z}{\partial t}$ （正向傳播時，輸入是 t ，輸出是 z ，所以這個節點（局部性）的微分是 $\frac{\partial z}{\partial t}$ ），傳遞給下個節點。另外，在圖 5-7 中，反向傳播的最初訊號 $\frac{\partial z}{\partial z}$ ，沒有出現在前面的算式，但是由於 $\frac{\partial z}{\partial z} = 1$ ，所以在前面的算式中省略。

圖 5-7 應該注意到，最左邊的反向傳播結果。根據連鎖律， $\frac{\partial z}{\partial z} \frac{\partial z}{\partial t} \frac{\partial t}{\partial x} = \frac{\partial z}{\partial t} \frac{\partial t}{\partial x} = \frac{\partial z}{\partial x}$ 成立，對應「與 x 有關的 z 微分」。換句話說，反向傳播進行的計算，是由連鎖律的原理構成的。

另外，在圖 5-7 帶入算式 (5.3) 的結果，如圖 5-8 所示， $\frac{\partial z}{\partial x}$ 可以當成是 $2(x+y)$ 來計算。

圖 5-8 根據計算圖的反向傳播結果， $\frac{\partial z}{\partial x}$ 是 $2(x+y)$

5.3 反向傳播

上一節說明了計算圖的反向傳播是基於連鎖律而成立。以下要以「+」及「×」等運算為例，說明反向傳播的結構。

5.3.1 加法節點的反向傳播

一開始，先來思考加法節點的反向傳播。這裡以 $z = x + y$ 算式為對象來介紹反向傳播。 $z = x + y$ 的微分可以依照以下方式（解析）計算。

$$\begin{aligned}\frac{\partial z}{\partial x} &= 1 \\ \frac{\partial z}{\partial y} &= 1\end{aligned}\tag{5.5}$$

如算式 (5.5) 所示， $\frac{\partial z}{\partial x}$ 與 $\frac{\partial z}{\partial y}$ 皆為 1。因此，用計算圖來顯示，可以畫出如圖 5-9 的結果。

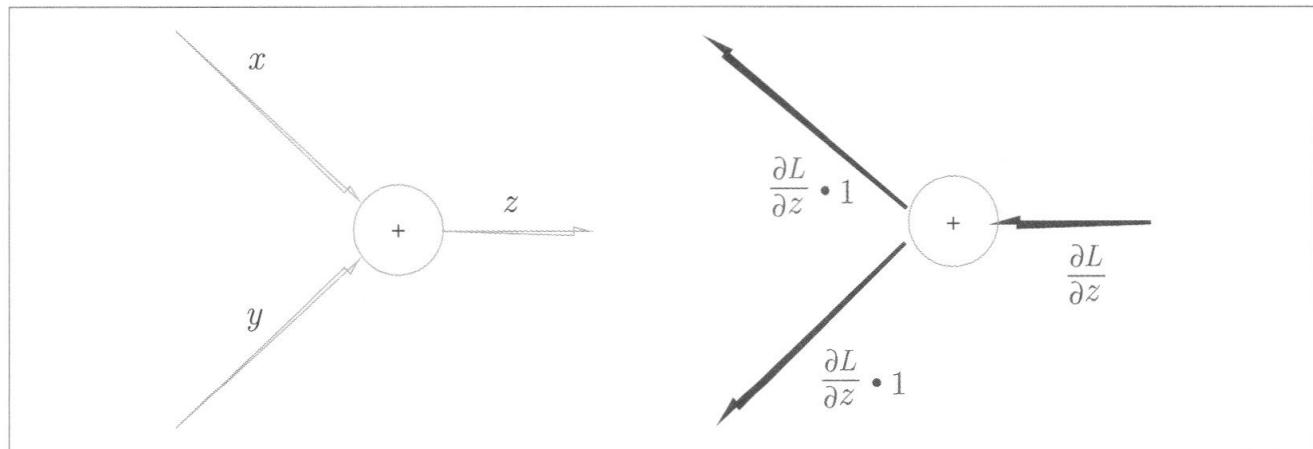


圖 5-9 加法節點的反向傳播：左圖是正向傳播，右圖是反向傳播。如右圖的反向傳播所示，加法節點的反向傳播是直接把上層的值，傳遞給下層

如圖 5-9 所示，反向傳播時，上層傳來的微分（這個例子是指 $\frac{\partial L}{\partial z}$ ）乘上 1，再傳給下層。換句話說，加法節點的反向傳播只要乘上 1，所以輸入值維持不變，只要傳給下個節點即可。

在這個範例中，上層傳來的微分值為 $\frac{\partial L}{\partial z}$ ，因為假設這是最後輸出值為 L 的大型計算圖，如圖 5-10 所示。 $z = x + y$ 的計算存在這個大型計算圖的某個位置，而且由上層傳來 $\frac{\partial L}{\partial z}$ ，接著分別往下層傳遞 $\frac{\partial L}{\partial x}$ 與 $\frac{\partial L}{\partial y}$ 。

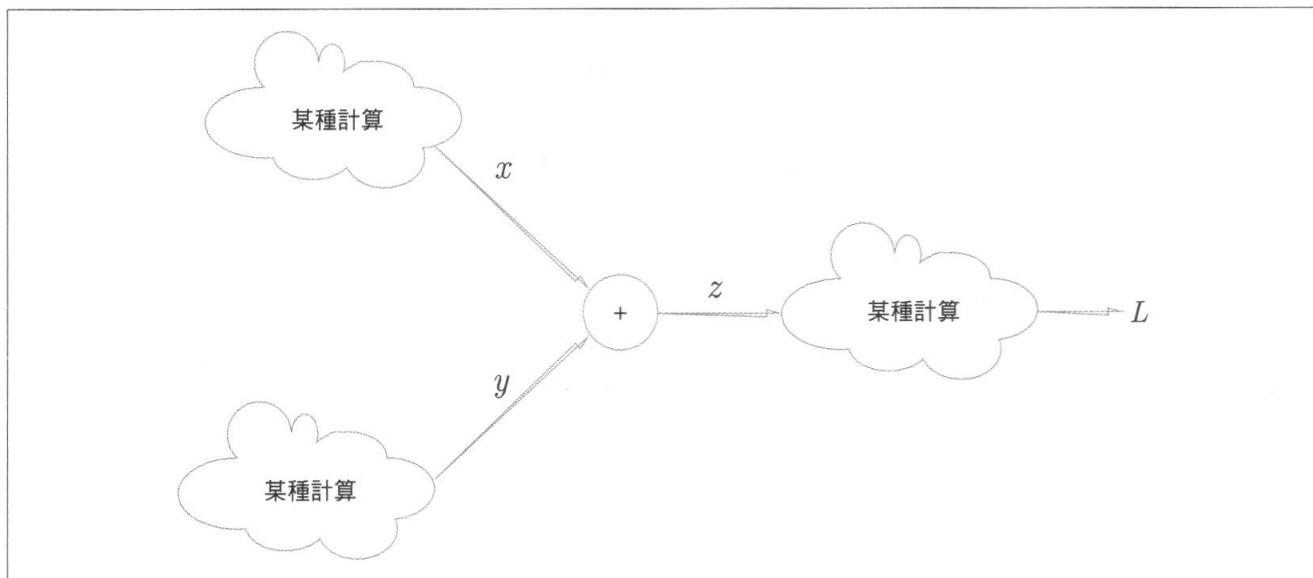


圖 5-10 這次的加法節點存在於最後輸出結果的一部分。進行反向傳播時，從最右邊的輸出開始，以反方向，將局部性微分從一個節點傳遞給另一個節點

接下來，舉一個實際範例來說明加法的反向傳播。假設「 $10 + 5 = 15$ 」，進行反向傳播時，上層傳來的值為 1.3。繪製成設計圖後，如圖 5-11 所示。

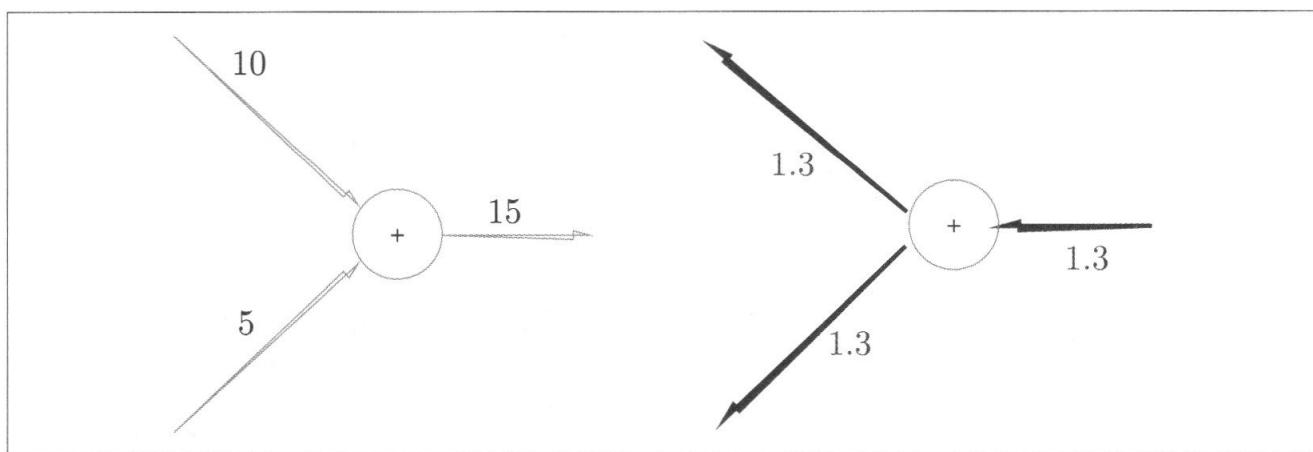


圖 5-11 加法節點的反向傳播具體範例

加法節點的反向傳播只將輸入訊號輸出給下個節點，結果如圖 5-11 所示，將 1.3 傳給下個節點。

5.3.2 乘法節點的反向傳播

接著要說明乘法節點的反向傳播。這裡要思考的是算式 $z=xy$ 。這個式子的微分如以下算式（5.6）所示。

$$\begin{aligned}\frac{\partial z}{\partial x} &= y \\ \frac{\partial z}{\partial y} &= x\end{aligned}\tag{5.6}$$

從算式（5.6）可以畫出以下計算圖。

乘法的反向傳播是將上層的值乘上正向傳播輸入訊號的「相反值」，再往下層傳遞。相反值是指，如圖 5-12 所示，假設正向傳播時是 x 訊號，反向傳播是 y 訊號；若正向傳播時是 y 訊號，反向傳播就是 x 訊號，形成這種相反關係。

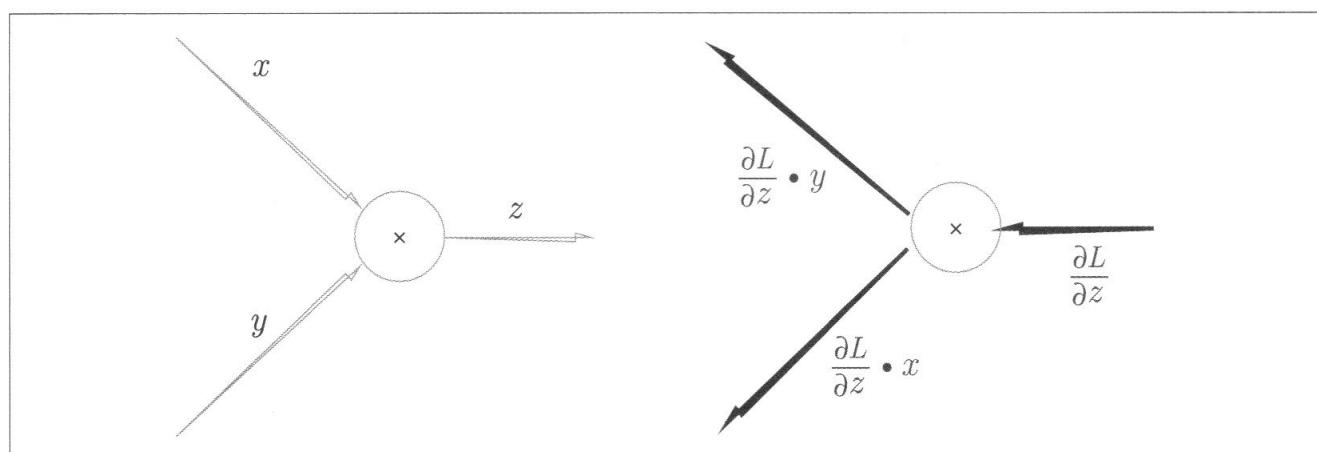


圖 5-12 乘法的反向傳播：左圖是正向傳播，右圖是反向傳播

接下來，讓我們來看一個實際的範例。假設要計算「 $10 \times 5 = 50$ 」，反向傳播時，上層將 1.3 傳遞過來。若用計算圖繪製這個部分，結果如圖 5-13 所示。

乘法的反向傳播是乘上輸入訊號的相反值，所以可以分別計算 $1.3 \times 5 = 6.5$ 、 $1.3 \times 10 = 13$ 。另外，在加法的反向傳播中，上層的值直接傳遞給下層，所以不需要正向傳播的輸入訊號值。但是，乘法的反向傳播卻需要正向傳播時的輸入訊號值。因此，進行乘法節點的反向傳播時，要維持正向傳播的輸入訊號。

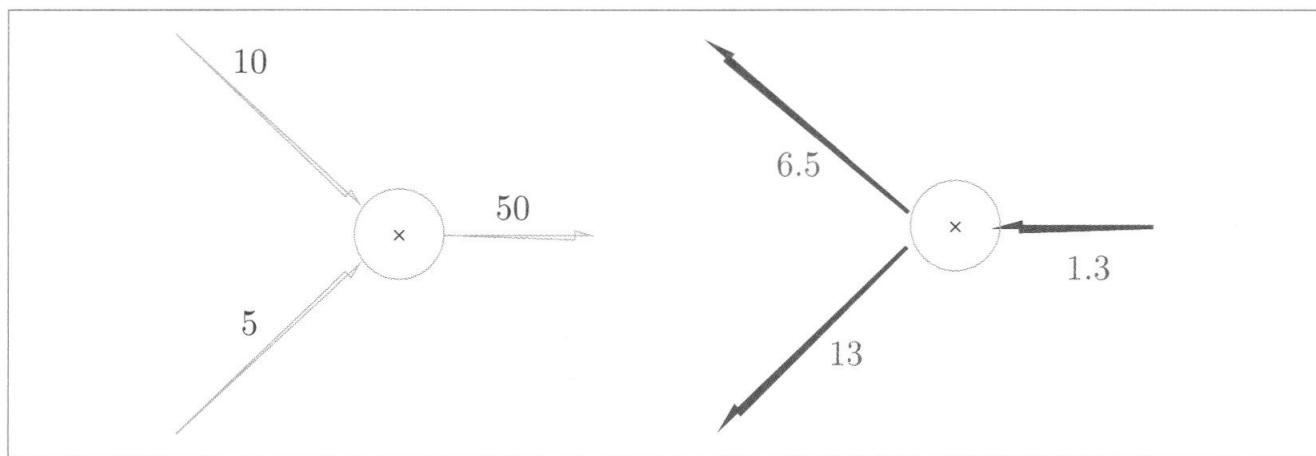


圖 5-13 乘法節點的反向傳播具體範例

5.3.3 購買蘋果的範例

讓我們再次思考本章最初提到的購買蘋果範例（2 顆蘋果與營業稅）。這裡要解答的問題是，蘋果的價格、蘋果的顆數、營業稅等 3 個變數會分別對最後的支付金額造成何種影響。等同於要計算「與蘋果價格有關的支付金額微分」、「與蘋果顆數有關的支付金額微分」、「與營業稅有關的支付金額微分」。利用計算圖的反向傳播解答這個問題，結果如圖 5-14 所示。

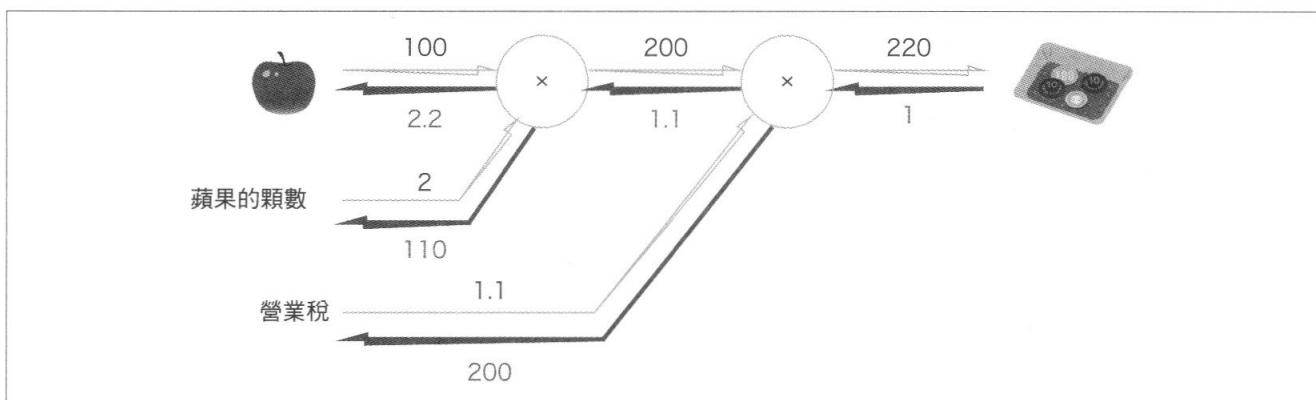


圖 5-14 購買蘋果的反向傳播範例

如同前面說明過，在乘法節點的反向傳播中，是將輸入訊號顛倒之後，往下層傳遞。根據圖 5-14 的結果所示，蘋果價格的微分是 2.2，蘋果顆數的微分是 110，營業稅的微分是 200。假設營業稅與蘋果價格都增加了相同量，可以解釋成，營業稅以 200 的大小影響最終的支付金額，蘋果價格以 2.2 的大小影響最終的支付金額。但是，在這個範例

中，營業稅與蘋果價格的單位不同，才會形成這種結果（營業稅的 1 是 100%，蘋果價格的 1 是 1 元）。

接下來，把「購買蘋果與橘子」的反向傳播，當作最後練習問題。在圖 5-15 的四方型內輸入數字，請分別計算各個變數的微分（答案在後面幾頁）。

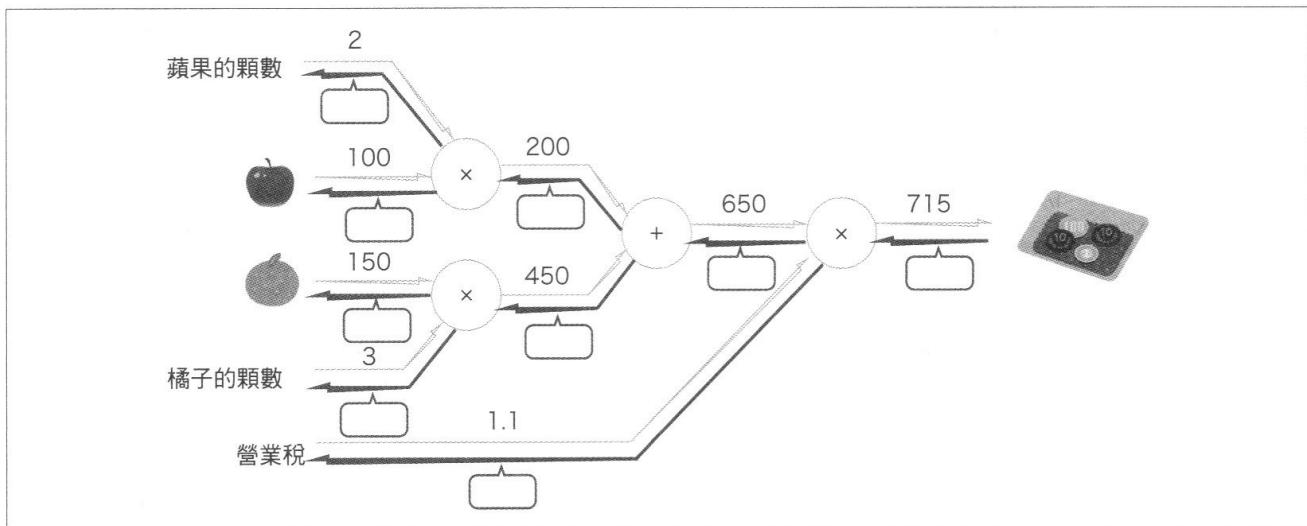


圖 5-15 購買蘋果與橘子的反向傳播範例：在四方型內輸入數字，完成反向傳播

5.4 執行單純的層級

這裡就用 Python 執行本節說明過的「購買蘋果」範例。計算圖的乘法節點以「乘法層（MulLayer）」，加法節點以「加法層（AddLayer）」的名稱來執行。



下一節要用一個類別來執行構成神經網路的「層（Layer）」。而這裡提到的「層」是指神經網路的功能單位。例如，sigmoid 函數的 Sigmoid 或矩陣乘積的 Affine 等，都是以層為單位來執行。因此，這裡也以「層」為單位，進行乘法節點與加法節點的計算。

5.4.1 執行乘法層

層是用來執行 forward() 或 backward() 等共通方法（介面）的部分。forward() 是對應正向傳播，backward() 是對應反向傳播。

接著要執行乘法層。乘法層可以當成是名為 `MulLayer` 的類別，執行以下處理（原始碼位於 `ch05/layer_naive.py`）

```
class MulLayer:
    def __init__(self):
        self.x = None
        self.y = None

    def forward(self, x, y):
        self.x = x
        self.y = y
        out = x * y

        return out

    def backward(self, dout):
        dx = dout * self.y # x 與 y 相反
        dy = dout * self.x

        return dx, dy
```

在 `__init__()` 中，進行實例變數 `x` 與 `y` 的初始化，用來保持正向傳播時的輸入值。在 `forward()` 中，取得 `x` 與 `y` 等 2 個引數，相乘之後輸出。然而，在 `backward()` 中，針對上層傳來的微分 (`dout`)，乘上正向傳播的「相反值」，再傳遞給下層。

以上就是 `MulLayer` 執行的處理。接著，使用 `MulLayer`，執行前面提過的「購買蘋果」（2 顆蘋果與營業稅）範例。上一節使用計算圖的正向傳播與反向傳播，進行了如圖 5-16 的計算。

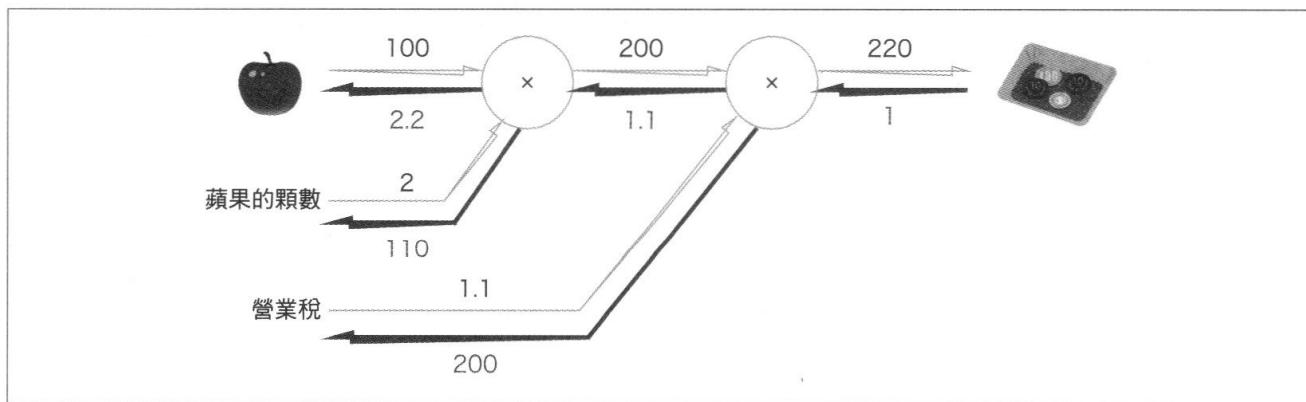


圖 5-16 購買 2 顆蘋果

使用乘法層，可以執行圖 5-16 的正向傳播，如下所示（原始碼位於 ch05/buy_apple.py）。

```
apple = 100
apple_num = 2
tax = 1.1

# layer
mul_apple_layer = MulLayer()
mul_tax_layer = MulLayer()

# forward
apple_price = mul_apple_layer.forward(apple, apple_num)
price = mul_tax_layer.forward(apple_price, tax)

print(price) # 220
```

另外，與各變數有關的微分，可以用 `backward()` 來計算。

```
# backward
dprice = 1
dapple_price, dtax = mul_tax_layer.backward(dprice)
dapple, dapple_num = mul_apple_layer.backward(dapple_price)

print(dapple, dapple_num, dtax) # 2.2 110 200
```

`backward()` 的呼叫順序與 `forward()` 的順序相反。此外，請注意 `backward()` 的引數是輸入「對應正向傳播時的輸出變數微分」。例如，`mul_apple_layer` 這個乘法層，在順向傳播時，輸出 `apple_price`，但是在逆向傳播時，是把 `apple_price` 的微分值 `apple_price` 設定成引數。此外，這個程式的執行結果，和圖 5-16 一致。

5.4.2 執行加法層

接著是在加法節點的加法層。加法層的執行過程如下所示。

```
class AddLayer:
    def __init__(self):
        pass

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

    def backward(self, dout):
        dx = dout * 1
```

```
dy = dout * 1
return dx, dy
```

加法層不需要特別初始化，所以 `__init__()` 沒有任何動作（`pass` 是指「不執行任何處理」的指令）。在加法層的 `forward()` 中，取得 2 個引數 `x`、`y`，相加之後再輸出。在 `backward()` 中，直接把上層傳來的微分（`dout`）傳遞給下層。

接下來，讓我們使用加法層與乘法層，執行如圖 5-17 所示，購買 2 顆蘋果與 3 顆橘子的計算。

利用 Python 執行 圖 5-17 的計算圖，結果如下所示（原始碼位於 `ch05/buy_apple_orange.py`）。

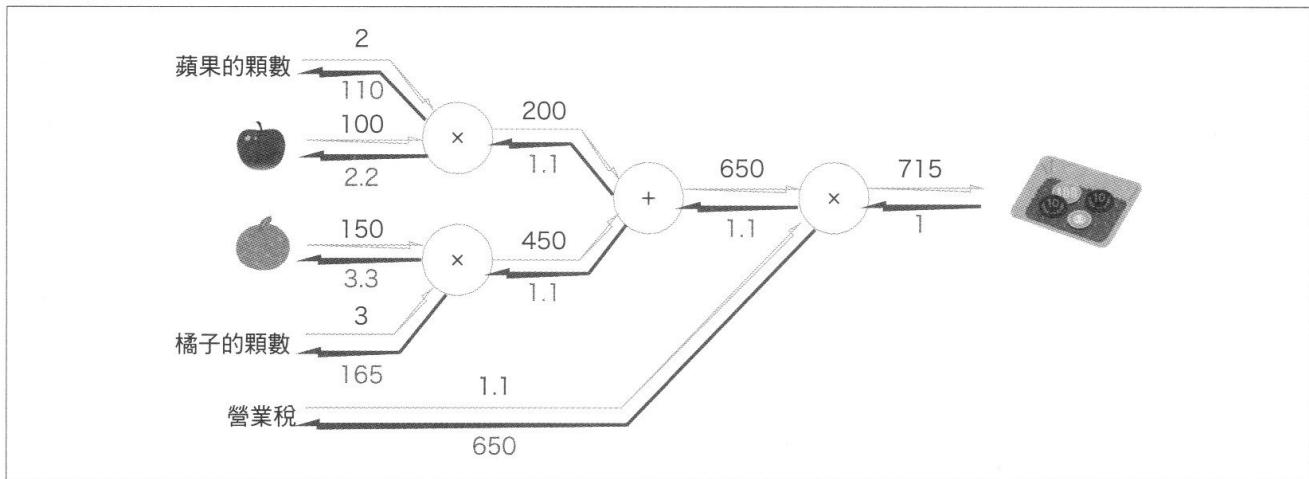


圖 5-17 購買 2 顆蘋果與 3 顆橘子

```
apple = 100
apple_num = 2
orange = 150
orange_num = 3
tax = 1.1

# layer
mul_apple_layer = MulLayer()
mul_orange_layer = MulLayer()
add_apple_orange_layer = AddLayer()
mul_tax_layer = MulLayer()

# forward
apple_price = mul_apple_layer.forward(apple, apple_num) #(1)
orange_price = mul_orange_layer.forward(orange, orange_num) #(2)
all_price = add_apple_orange_layer.forward(apple_price, orange_price) #(3)
```

```

price = mul_tax_layer.forward(all_price, tax) #(4)

# backward
dprice = 1
dall_price, dtax = mul_tax_layer.backward(dprice) #(4)
dapple_price, dorange_price = add_apple_orange_layer.backward(dall_price) #(3)
dorange, dorange_num = mul_orange_layer.backward(dorange_price) #(2)
dapple, dapple_num = mul_apple_layer.backward(dapple_price) #(1)

print(price) # 715
print(dapple_num, dapple, dorange, dorange_num, dtax) # 110 2.2 3.3 165 650

```

這裡的執行過程有點長，其實每個指令都很單純。建立必要的層級，依照適當的順序呼叫正向傳播的方法 `forward()`。接著再以與正向傳播相反的順序，呼叫反向傳播的 `backward()` 方法，就可以獲得要計算的微分。

在計算圖中，可以輕鬆完成每一層的處理（這裡是指乘法與加法），利用這種方法，即可計算出複雜的微分。接下來，要執行在神經網路中的層級。

5.5 執行活化函數層

我想將計算圖的思考方法套用在神經網路上。這裡把構成神經網路的「層（layer）」當作一個類別來執行處理。首先，要執行活化函數 ReLU 與 Sigmoid 層。

5.5.1 ReLU 層

當作活化函數使用的 ReLU（Rectified Linear Unit）可以表示成以下算式（5.7）。

$$y = \begin{cases} x & (x > 0) \\ 0 & (x \leq 0) \end{cases} \quad (5.7)$$

利用算式（5.7），計算與 x 有關的 y 微分，如算式（5.8）所示。

$$\frac{\partial y}{\partial x} = \begin{cases} 1 & (x > 0) \\ 0 & (x \leq 0) \end{cases} \quad (5.8)$$

如算式（5.8）所示，如果正向傳播的輸入 x 比 0 大，反向傳播會將上層的值直接傳遞給下層。相反來說，正向傳播時的 x 比 0 小，反向傳播會停止往下層傳遞訊號。以計算圖顯示，結果如圖 5-18 所示。

接下來要執行 ReLU 層。假設在神經網路的層級中，於 `forward()` 或 `backward()` 的引數輸入 NumPy 陣列。此外，ReLU 層執行的處理，位於 `common/layers.py`。

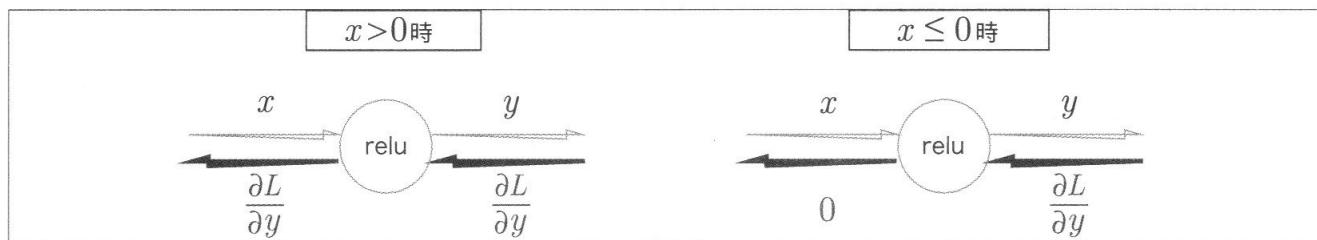


圖 5-18 ReLU 層的計算圖

```
class Relu:
    def __init__(self):
        self.mask = None

    def forward(self, x):
        self.mask = (x <= 0)
        out = x.copy()
        out[self.mask] = 0

        return out

    def backward(self, dout):
        dout[self.mask] = 0
        dx = dout

        return dx
```

ReLU 類別含有當作實例變數的 `mask` 變數。`mask` 變數是由 `True/False` 形成的 NumPy 陣列，假如正向傳播的輸入 `x` 元素小於 0 為 `True`，其餘（比 0 大的元素）為 `False`。如下範例所示，`mask` 變數保持由 `True/False` 形成的 NumPy 陣列。

```
>>> x = np.array( [[1.0, -0.5], [-2.0, 3.0]] )
>>> print(x)
[[ 1. -0.5]
 [-2.  3.]]
>>> mask = (x <= 0)
>>> print(mask)
[[False  True]
 [ True False]]
```

如圖 5-18 所示，如果正向傳播時的輸入值小於 0，反向傳播的值就會變成 0。因此，在反向傳播中，使用正向傳播的 `mask`，針對上層傳來的 `dout`，將 `mask` 元素為 `True` 的位置設定為 0。



ReLU 層的功能類似電路中的「開關」。正向傳播時，電流通過，開關變成 ON，電流沒有通過，開關變成 OFF。反向傳播時，開關為 ON 時，電流直接通過，如果是 OFF，電流就不再通過。

5.5.2 Sigmoid 層

接下來要執行 sigmoid 函數。sigmoid 函數是用算式 (5.9) 表示的函數。

$$y = \frac{1}{1 + \exp(-x)} \quad (5.9)$$

用計算圖顯示算式 (5.9)，結果如圖 5-19 所示。

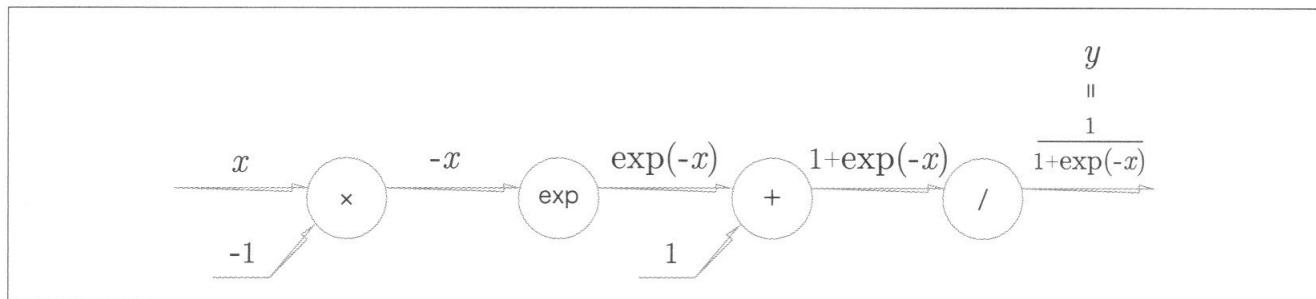


圖 5-19 Sigmoid 層的計算圖（只有正向傳播）

在圖 5-19 中，除了「×」與「+」節點之外，還出現了新的「exp」與「/」節點。「exp」節點執行 $y = \exp(x)$ 計算，「/」節點執行 $y = \frac{1}{x}$ 計算。

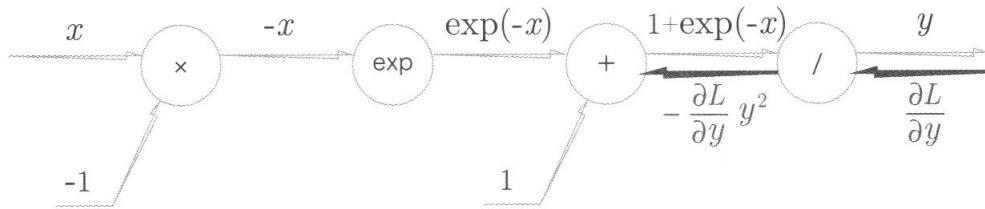
如圖 5-19 所示，算式 (5.9) 是由局部性計算的傳播所構成。接下來，要進行圖 5-19 的計算圖反向傳播。這裡將按照反向傳播的執行流程來說明（兼具重點整理）。

步驟 1

「/」節點表示 $y = \frac{1}{x}$ ，這個微分可以解析成以下算式。

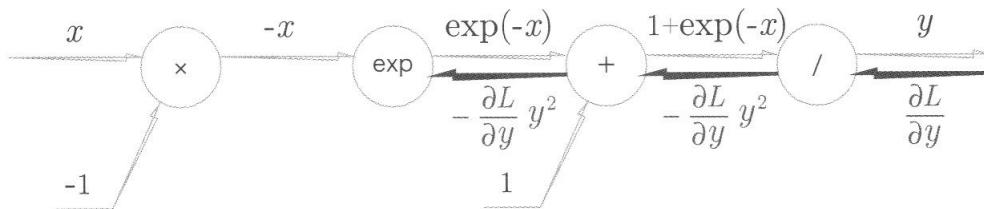
$$\begin{aligned} \frac{\partial y}{\partial x} &= -\frac{1}{x^2} \\ &= -y^2 \end{aligned} \quad (5.10)$$

根據算式 (5.10)，進行反向傳播時，對上層的值乘上 $-y^2$ （正向傳播輸出的 2 平方並加上負號的值），傳遞給下層。計算圖如下所示。



步驟 2

「+」節點是直接將上層的值傳遞給下層，計算圖如下所示。

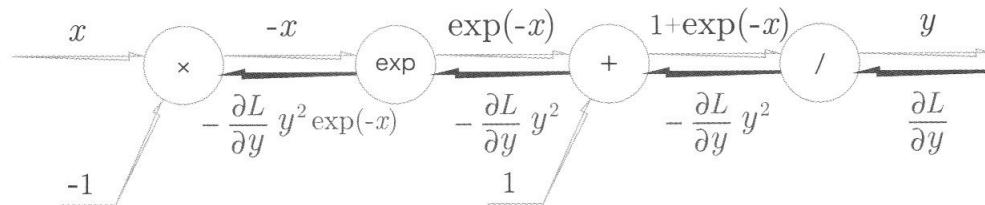


步驟 3

「exp」節點是 $y = \exp(x)$ ，顯示微分的算式如下所示。

$$\frac{\partial y}{\partial x} = \exp(x) \quad (5.11)$$

在計算圖中，針對上層的值，乘上正向傳播時的輸出（這個範例是 $\exp(-x)$ ），再傳遞給下層。



步驟 4

「 \times 」節點是乘上正向傳播的「相反」值。因此，這裡乘上 -1 。

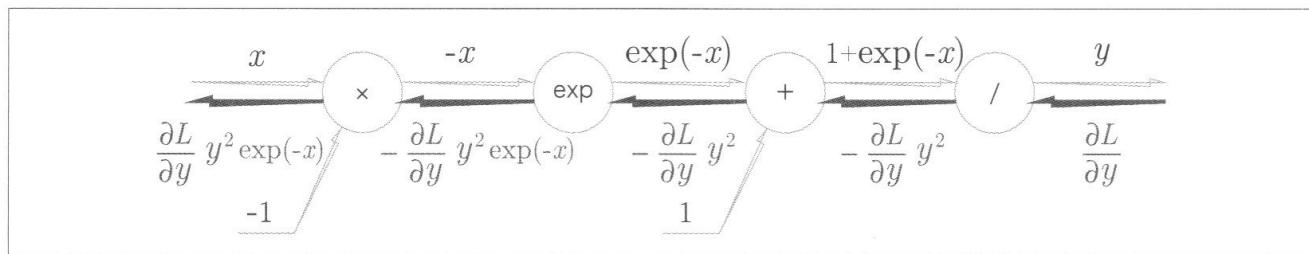


圖 5-20 Sigmoid 層的計算圖

如上所示，利用圖 5-20 的計算圖，可以進行 Sigmoid 層的反向傳播。從圖 5-20 的結果中可以得知，反向傳播的輸出為 $\frac{\partial L}{\partial y} y^2 \exp(-x)$ ，再將這個值傳給下層的節點。請注意， $\frac{\partial L}{\partial y} y^2 \exp(-x)$ 只有正向傳播的輸入 x 與輸出 y 才可以計算出來。因此，圖 5-20 的計算圖可以畫成經過群組化後的「sigmoid」節點，如圖 5-21 所示。

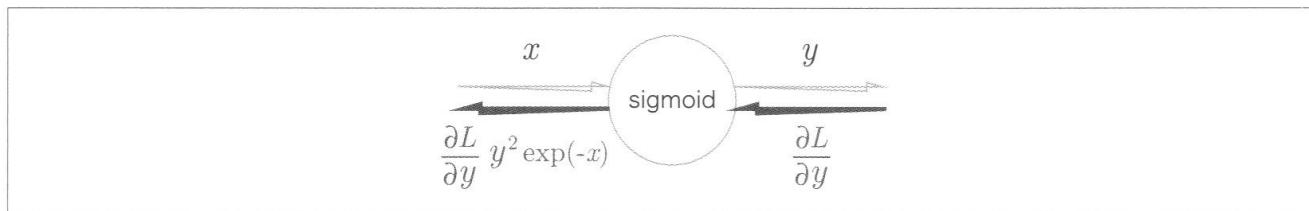


圖 5-21 Sigmoid 層的計算圖（精簡版）

圖 5-20 的計算圖與圖 5-21 精簡版的計算圖，計算結果完全一樣。但是，精簡版的計算圖可以省略在反向傳播途中的計算，計算效率比較好。此外，將節點群組化，不用在意 Sigmoid 層的細節，可以將注意力集中在輸入與輸出上，也是非常重要的關鍵。

另外， $\frac{\partial L}{\partial y} y^2 \exp(-x)$ 可以整理如下。

$$\begin{aligned}
 \frac{\partial L}{\partial y} y^2 \exp(-x) &= \frac{\partial L}{\partial y} \frac{1}{(1 + \exp(-x))^2} \exp(-x) \\
 &= \frac{\partial L}{\partial y} \frac{1}{1 + \exp(-x)} \frac{\exp(-x)}{1 + \exp(-x)} \\
 &= \frac{\partial L}{\partial y} y(1 - y)
 \end{aligned} \tag{5.12}$$

因此，顯示在圖 5-21 的 Sigmoid 層反向傳播，可以只從正向傳播的輸出開始計算。

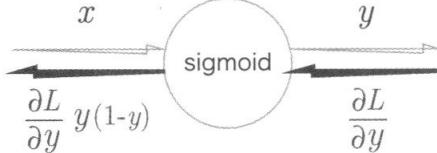


圖 5-22 Sigmoid 層的計算圖：利用正向傳播的輸出 y ，可以進行反向傳播的計算

接下來，利用 Python 執行 Sigmoid 層。參考圖 5-22，可以執行以下處理（這個部分位於 common/layers.py）。

```
class Sigmoid:
    def __init__(self):
        self.out = None

    def forward(self, x):
        out = 1 / (1 + np.exp(-x))
        self.out = out

        return out

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

        return dx
```

在這個過程中，先將正向傳播時的輸出保持實例變數 out。在反向傳播時，使用 out 變數進行計算。

5.6 執行 Affine / Softmax 層

5.6.1 Affine 層

在神經網路的正向傳播中，使用了矩陣乘積（在 NumPy 是 `np.dot()`），計算含權重訊號的總和（詳細說明請參考「3.3 多維陣列的運算」）。例如，前面使用過 Python 執行了以下處理，不曉得你是否還記得？

```

>>> X = np.random.rand(2)    # 輸入
>>> W = np.random.rand(2,3) # 權重
>>> B = np.random.rand(3)   # 偏權值
>>>
>>> X.shape # (2,)
>>> W.shape # (2, 3)
>>> B.shape # (3,)
>>>
>>> Y = np.dot(X, W) + B

```

X 、 W 、 B 的形狀分別是 $(2,)$ 、 $(2, 3)$ 、 $(3,)$ 的多維陣列。這樣神經元的加權總和可以用 $Y = np.dot(X, W) + B$ 計算出來。利用活化函數轉換 Y ，再傳遞給下一層，這就是神經網路的正向傳播。再複習一下，計算矩陣的乘積時，要讓對應維度的元素數量一致。例如， X 與 W 的乘積如圖 5-23 所示，必須讓對應維度的元素數量一致。這裡的矩陣形狀以括弧顯示，如 $(2, 3)$ （這是為了對應 NumPy 的 `shape` 輸出）。

$$\begin{matrix} \mathbf{X} & \cdot & \mathbf{W} & = \mathbf{O} \\ (2,) & & (2, 3) & (3,) \\ \text{一致} \end{matrix}$$

圖 5-23 矩陣乘積要讓對應的維度元素數量一致



利用神經網路的正向傳播，計算矩陣乘積，在幾何學中，稱作「仿射轉換（Affine Transformation）」。因此，這裡以「Affine 層」的名稱，執行仿射轉換處理。

接下來，使用計算圖顯示矩陣乘積與偏權值的和。計算乘積的節點顯示為「dot」，可以用圖 5-24 的計算圖來表示 $np.dot(X, W) + B$ 的計算。此外，各變數的上面部分，也顯示了該變數的形狀（例如，在計算圖上 X 的形狀顯示成 $(2,)$ ， $X \cdot W$ 的形狀顯示為 $(3,)$ ）。

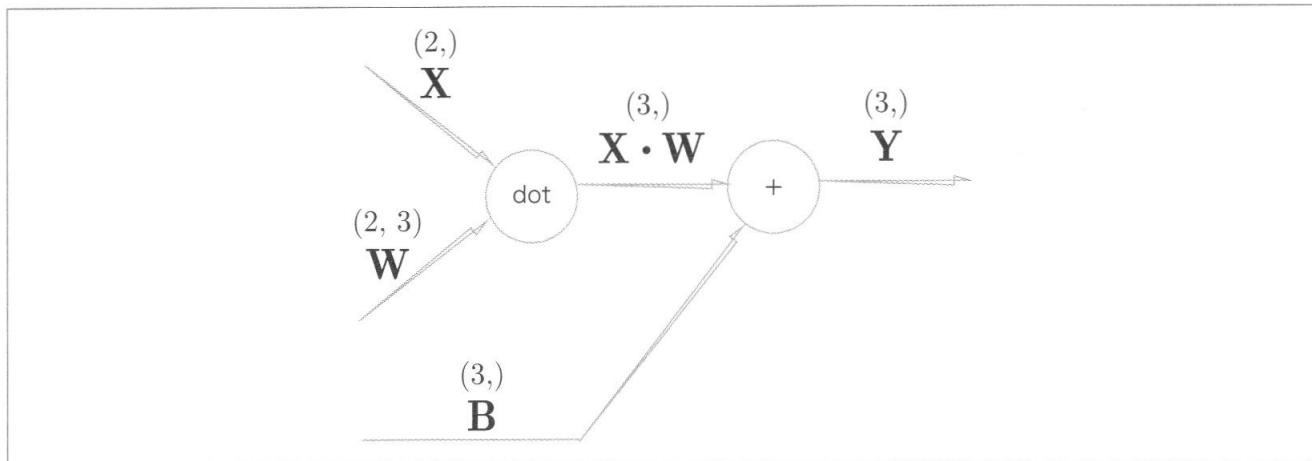


圖 5-24 Affine 層的計算圖：注意變數是矩陣。在各變數的上面部分顯示該變數的形狀

圖 5-24 是比較單純的計算圖。但是，請注意 \mathbf{X} 、 \mathbf{W} 、 \mathbf{B} 是矩陣（多維陣列）。到目前為止，看到的計算圖都是在節點之間傳遞「純量」，但是這個範例卻是在節點之間傳遞「矩陣」。

接下來，要思考圖 5-24 計算圖的反向傳播。計算以矩陣為對象的反向傳播時，只要依序寫下矩陣元素，就可以和以純量為對象的計算圖一樣，按照相同步驟來思考。實際寫出來，能獲得以下算式（這裡省略導出算式（5.13）的過程）。

$$\begin{aligned}\frac{\partial L}{\partial \mathbf{X}} &= \frac{\partial L}{\partial \mathbf{Y}} \cdot \mathbf{W}^T \\ \frac{\partial L}{\partial \mathbf{W}} &= \mathbf{X}^T \cdot \frac{\partial L}{\partial \mathbf{Y}}\end{aligned}\tag{5.13}$$

在算式（5.13）中， \mathbf{W}^T 的 T 代表轉置。轉置是指，將 \mathbf{W} 的 (i, j) 元素轉換成 (j, i) 。實際用算式表示，結果如下所示。

$$\begin{aligned}\mathbf{W} &= \begin{pmatrix} w_{11} & w_{21} & w_{31} \\ w_{12} & w_{22} & w_{32} \end{pmatrix} \\ \mathbf{W}^T &= \begin{pmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \end{pmatrix}\end{aligned}\tag{5.14}$$

如算式（5.14）所示， \mathbf{W} 的形狀為 $(2, 3)$ 時， \mathbf{W}^T 的形狀變成 $(3, 2)$ 。

接下來，根據算式 (5.13)，畫出計算圖的反向傳播。結果如圖 5-25 所示。

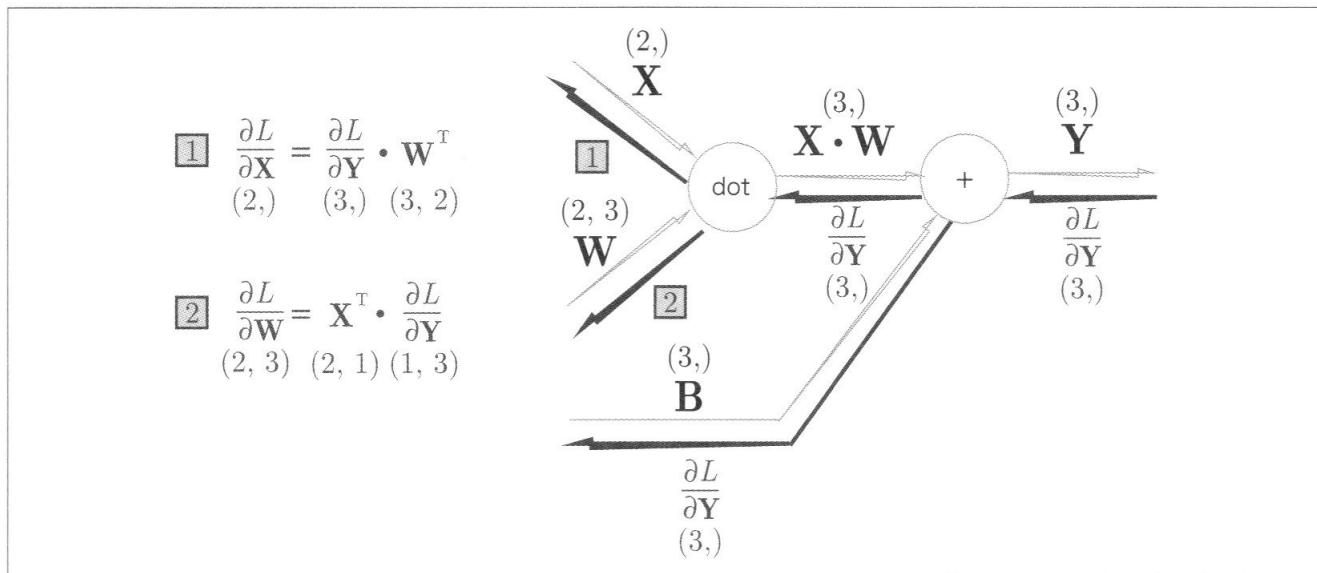


圖 5-25 Affine 層的反向傳播：注意變數為多維陣列。進行反向傳播時，在各變數的下面部分顯示該變數的形狀

在圖 5-25 的計算圖中，請注意各變數的形狀。尤其是， \mathbf{X} 與 $\frac{\partial L}{\partial \mathbf{X}}$ 的形狀相同， \mathbf{W} 與 $\frac{\partial L}{\partial \mathbf{W}}$ 形狀相同這一點。利用以下算式來顯示 \mathbf{X} 與 $\frac{\partial L}{\partial \mathbf{X}}$ 形狀相同，就能明白。

$$\begin{aligned}\mathbf{X} &= (x_0, x_1, \dots, x_n) \\ \frac{\partial L}{\partial \mathbf{X}} &= \left(\frac{\partial L}{\partial x_0}, \frac{\partial L}{\partial x_1}, \dots, \frac{\partial L}{\partial x_n} \right)\end{aligned}\quad (5.15)$$

為什麼要注意矩陣的形狀？這是因為矩陣的乘積必須讓對應的維度元素數量一致，確認這一點，就能導出算式 (5.13)。假設 $\frac{\partial L}{\partial \mathbf{Y}}$ 的形狀是 $(3,)$ ， \mathbf{W} 的形狀是 $(2, 3)$ ，思考當 $\frac{\partial L}{\partial \mathbf{X}}$ 的形狀變成 $(2,)$ 時， $\frac{\partial L}{\partial \mathbf{Y}}$ 與 \mathbf{W} 的乘積（圖 5-26），自然可以導出算式 (5.13)。

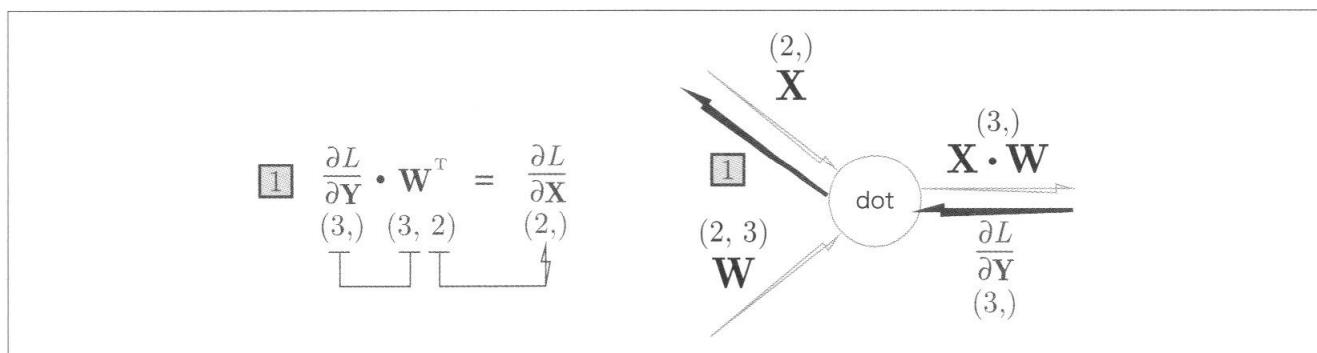


圖 5-26 矩陣的乘積（「dot」節點）的反向傳播可以利用組合乘積的方式，讓矩陣對應的維度元素數量一致

5.6.2 批次版 Affine 層

前面說明的 Affine 層是以單一輸入資料 \mathbf{X} 為對象，接下來要思考，若要一次正向傳播 N 個資料時，亦即批次版的 Affine 層（整合性資料稱作「批次」），該如何處理。

先用計算圖顯示批次版 Affine 層。批次版的 Affine 層如圖 5-27 所示。

和前面說明的差別只在輸入 \mathbf{X} 的形狀變成 $(N, 2)$ 而已。其餘和前面一樣，只要在計算圖上，進行矩陣計算即可。另外，反向傳播時，注意矩陣的形狀，就可以和前面一樣，導出 $\frac{\partial L}{\partial \mathbf{X}}$ 與 $\frac{\partial L}{\partial \mathbf{W}}$ 。

進行偏權值的加法運算時，要特別注意。正向傳播的偏權值加法要針對 $\mathbf{X} \cdot \mathbf{W}$ ，分別加入偏權值。例如， $N=2$ 時（有 2 個資料），偏權值要分別針對這 2 個資料（針對各自的計算結果）進行加總。具體範例如下所示。

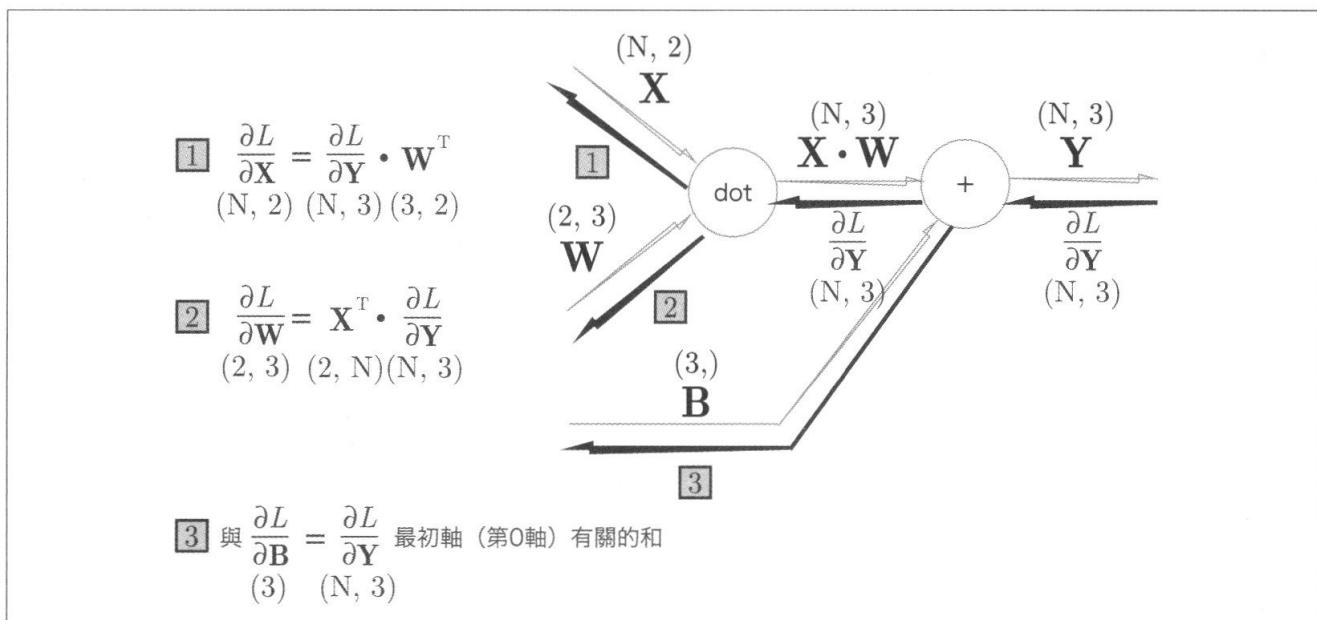


圖 5-27 批次版 Affine 層的計算圖

```
>>> X_dot_W = np.array([[0, 0, 0], [10, 10, 10]])
>>> B = np.array([1, 2, 3])
>>>
>>> X_dot_W
array([[ 0,  0,  0],
       [10, 10, 10]])
>>> X_dot_W + B
array([[ 1,  2,  3],
       [11, 12, 13]])
```

正向傳播的偏權值加法是分別針對各個資料（第 1 個資料、第 2 個資料、…）相加。因此，反向傳播時，各個資料的反向傳播值，必須集中在偏權值的元素中。若用程式顯示，結果如下所示。

```
>>> dY = np.array([[1, 2, 3], [4, 5, 6]])
>>> dY
array([[1, 2, 3],
       [4, 5, 6]])
>>>
>>> dB = np.sum(dY, axis=0)
>>> dB
array([5, 7, 9])
```

這個範例假設資料有 2 個 ($N=2$)。偏權值的反向傳播是，按照各個資料統計這 2 個資料的微分。因此，利用 `np.sum()`，針對第 0 軸（以資料為單位的軸），計算 (`axis=0`) 的總和。

根據以上資料，執行以下的 `Affine` 處理。另外，位於 `common/layers.py` 的 `Affine` 處理，還考慮到輸入資料為張量（四維資料）的情況，因此與以下說明的內容有些微差異。

```
class Affine:
    def __init__(self, W, b):
        self.W = W
        self.b = b
        self.x = None
        self.dW = None
        self.db = None

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

        return out

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

        return dx
```

5.6.3 Softmax-with-Loss 層

最後要說明輸出層的 Softmax 函數。Softmax 函數（當作複習）是將輸入值正規化後再輸出。例如，辨識手寫數字時，Softmax 層的輸出如圖 5-28 所示。

在圖 5-28 中，Softmax 層是把輸入值正規化，亦即輸出的和變成 1，再輸出。另外，辨識手寫數字時，為了進行 10 類別分類，所以 Softmax 層的輸出變成 10 個。



神經網路的處理分成推論（*inference*）與學習等兩個階段。在神經網路的推論中，一般不使用 Softmax 層。例如圖 5-28 的神經網路進行推論時，會把最後 Affine 層的輸出當作辨識結果。另外，沒有進行神經網路正規化的輸出結果（在圖 5-28 是指 Sofmax 的上層 Affine 層的輸出），稱作「評分（score）」。換句話說，假如神經網路的推論只產生一個答案時，將只對評分（score）的最大值有興趣，因此不需要 Softmax 層。然而，神經網路的學習需要用到 Softmax 層。

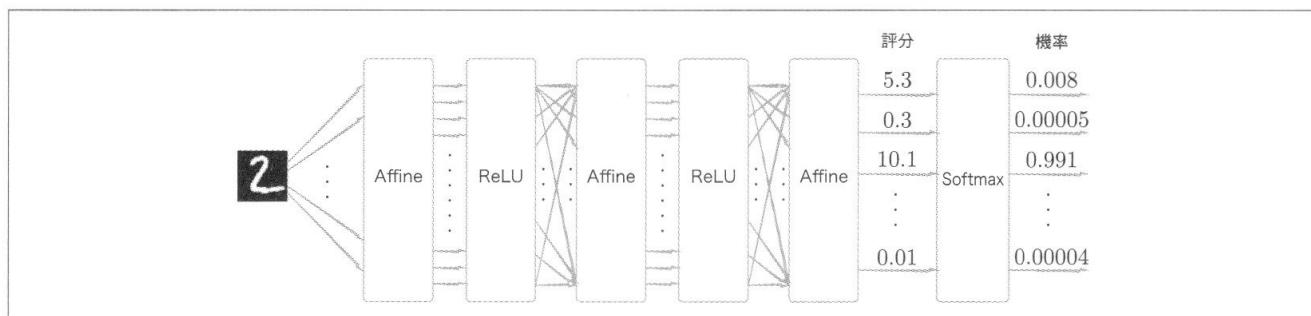


圖 5-28 輸入影像是利用 Affine 層與 ReLU 層進行轉換，利用 Softmax 層將 10 個輸入正規化。在這個範例中，「0」的評分是 5.3，根據 Softmax 層，轉換成 0.008（0.8%）。另外，「2」的評分是 10.1，所以轉換成 0.991（99.1%）

接下來要執行 Softmax 層，但是這裡要執行的是，包含損失函數的交叉熵誤差（cross entropy error）之「Softmax-with-Loss 層」。以計算圖顯示 Softmax-with-Loss 層（Softmax 函數與交叉熵誤差），如圖 5-29 所示。

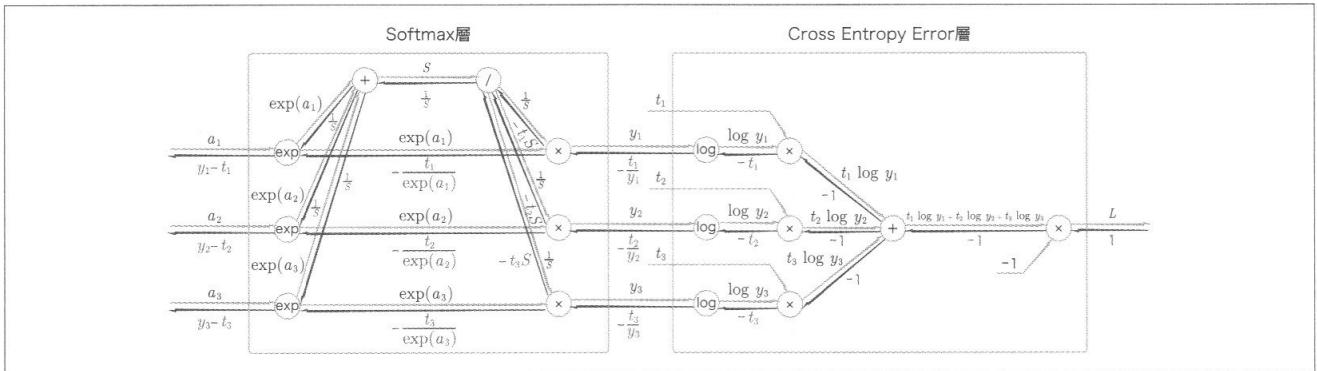


圖 5-29 Softmax-with-Loss 層的計算圖

如上所示，Softmax-with-Loss 層有點複雜，這裡只顯示結果。對導出 Softmax-with-Loss 層的過程有興趣的讀者，請參考「附錄 A Softmax-with-Loss 層的計算圖」。

將圖 5-29 的計算圖簡化之後，可以畫成圖 5-30。

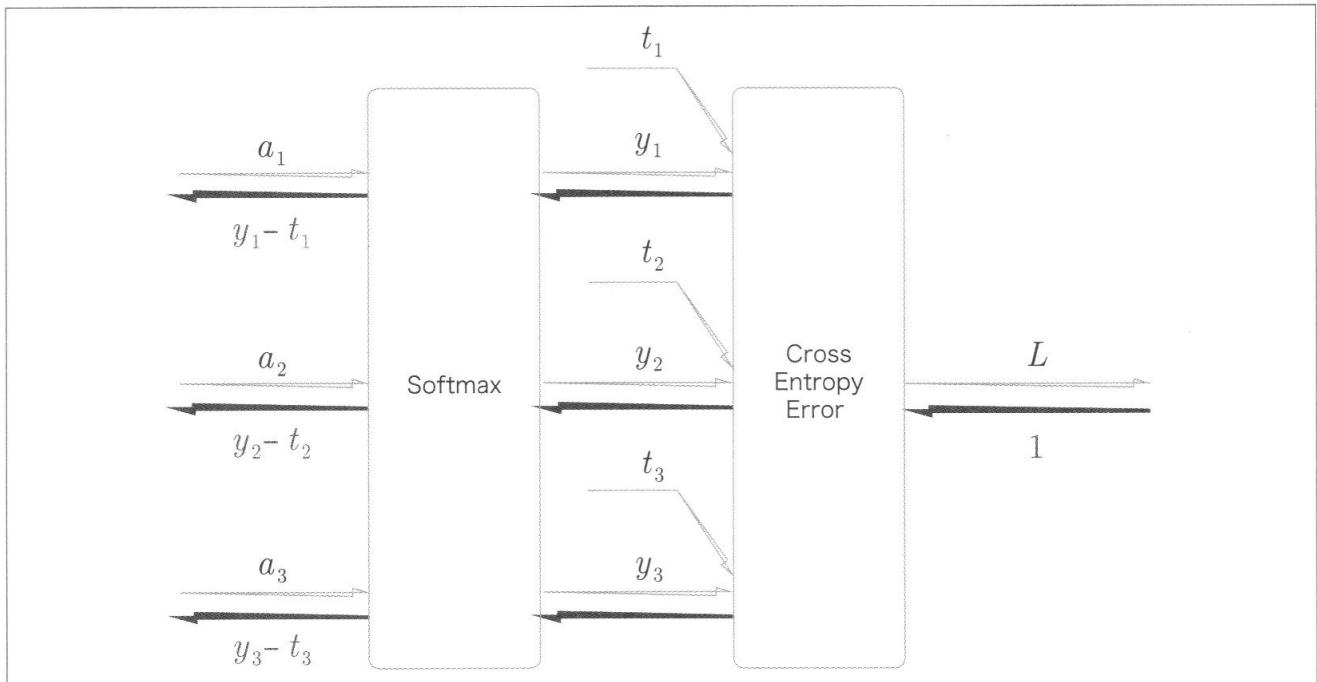


圖 5-30 「簡易版」Softmax-with-Loss 層的計算圖

在圖 5-30 的計算圖中，Softmax 函數顯示為 Softmax 層，交叉熵誤差顯示為 Cross Entropy Error 層。假設這裡要進行 3 個類別分類，從上層接收 3 個輸入（評分）。如圖 5-30 所示，Softmax 層將輸入 (a_1, a_2, a_3) 正規化，輸出 (y_1, y_2, y_3) 。Cross Entropy Error 層取得 Softmax 的輸出 (y_1, y_2, y_3) 與訓練資料 (t_1, t_2, t_3) ，從這些資料中，輸出損失 L 。

圖 5-30 必須注意到反向傳播的結果。來自 Softmax 層的反向傳播，形成 $(y_1 - t_1, y_2 - t_2, y_3 - t_3)$ 的「整齊」結果。 (y_1, y_2, y_3) 是 Softmax 層的輸出， (t_1, t_2, t_3) 是訓練資料，所以 $(y_1 - t_1, y_2 - t_2, y_3 - t_3)$ 是 Softmax 層的輸出與訓練資料的差分。因為，在神經網路的反向傳播中，這個差分的誤差會傳遞給上一層。對於神經網路的學習而言，是非常重要的性質。

神經網路的學習目的是，調整權重參數，讓神經網路的輸出（Softmax 的輸出）趨近訓練資料。因此，神經網路的輸出與訓練資料的誤差，必須有效率地傳遞給上一層。剛才 $(y_1 - t_1, y_2 - t_2, y_3 - t_3)$ 的結果等於 Softmax 層輸出與訓練資料的差，直接表現出目前神經網路的輸出與訓練資料的誤差。



使用「交叉熵誤差」當作「Softmax 函數」的損失函數，進行反向傳播時，可以導出 $(y_1 - t_1, y_2 - t_2, y_3 - t_3)$ 的「整齊」結果。事實上，這種「整齊」結果並非偶然，而是交叉熵誤差這個函數的特別設計。另外，在迴歸問題中，於輸出層使用「恆等函數」以及當作損失函數的「均方誤差」（請參考「3.5 輸出層的設計」），也是基於相同道理。換句話說，使用「均方誤差」當作「恆等函數」的損失函數時，反向傳播會得到 $(y_1 - t_1, y_2 - t_2, y_3 - t_3)$ 的「整齊」結果。

讓我們舉個實際的範例。假設訓練資料為 $(0, 1, 0)$ ，Softmax 層的輸出是 $(0.3, 0.2, 0.5)$ 。正確答案標籤的機率是 0.2 (20%)，此時的神經網路無法進行正確辨識。在這種情況下，Softmax 層的反向傳播會傳遞較大的誤差 $(0.3, -0.8, 0.5)$ 。由於傳遞了較大的誤差，使得上一層從這個大誤差中，學習到比 Softmax 層更大的內容。

再舉另一個範例，假設訓練資料為 $(0, 1, 0)$ ，Softmax 層的輸出是 $(0.01, 0.99, 0)$ （這個神經網路可以正確辨識）。此時，Softmax 層的反向傳播會得到 $(0.01, -0.01, 0)$ 的小誤差，並且將小誤差傳遞給上一層。由於誤差小，所以上一層學習到的內容將小於 Softmax 層。

接下來，要執行 Softmax-with-Loss 層。Softmax-with-Loss 層的執行過程如下所示。

```
class SoftmaxWithLoss:
    def __init__(self):
        self.loss = None # 損失
        self.y = None    # softmax 的輸出
        self.t = None    # 訓練資料 (one-hot vector)

    def forward(self, x, t):
        self.t = t
```

```

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

    return self.loss

def backward(self, dout=1):
    batch_size = self.t.shape[0]
    dx = (self.y - self.t) / batch_size

return dx

```

在執行過程中，使用了「3.5.2 執行 softmax 函數時的注意事項」以及「4.2.2 交叉熵誤差」出現過的函數，`softmax()` 與 `cross_entropy_error()`。因此，執行過程非常簡單。另外，在反向傳播時，必須注意到，以批次個數（`batch_size`）除以傳播值，將每個資料的誤差傳遞給上一層這一點。

5.7 執行誤差反向傳播法

組合上一節執行過的各個層級，就能像組合樂高般，建構出神經網路。因此，這裡要一邊組合前面執行過的各層，一邊建構神經網路。

5.7.1 神經網路的學習總圖

以下說明有點長，在開始實際執行之前，讓我們先再次確認神經網路的學習總圖。接著再列出神經網路的學習步驟。

前提

神經網路具有可適應的權重與偏權值，調整權重與偏權值，以適應訓練資料，這種過程稱作「學習」。神經網路的學習可以分成以下 4 個步驟。

步驟 1（小批次）

從訓練資料中，隨機取出部分資料。

步驟 2（計算梯度）

計算與各權重參數有關的損失函數梯度。

步驟 3（更新參數）

往梯度方向微量更新權重參數。

步驟 4（重複）

重複步驟 1、步驟 2、步驟 3。

前面說明過的誤差反向傳播法，就是步驟 2 的「計算梯度」。上一章使用了數值微分來計算梯度，數值微分可以輕易執行，但是相對來說，計算時間比較久。使用誤差反向傳播法，可以快速計算出梯度，這點與需要花時間的數值微分不同。

5.7.2 執行對應誤差反向傳播法的神經網路

接下來，開始執行。以下是把雙層神經網路當作 TwoLayerNet 來執行。首先，整理這個類別的實例變數與方法，如表 5-1 與表 5-2 所示。

表 5-1 TwoLayerNet 類別的實例變數

實例變數	說明
params	這是維持神經網路參數的字典變數（實例變數）。 params['W1'] 是第 1 層的權重，params['b1'] 是第 1 層的偏權值。 params['W2'] 是第 2 層的權重，params['b2'] 是第 2 層的偏權值。
layers	這是維持神經網路層級的有序字典變數。 以 layers['Affine1']、layers['Relu1']、layers['Affine2'] 等有序字典變數，保持各層。
lastLayer	神經網路的最後一層。 在這個範例中，是指 SoftmaxWithLoss 層。

表 5-2 TwoLayerNet 類別的方法

方法	說明
__init__(self, input_size, hidden_size, output_size, weight_init_std)	進行初始化。 引數從頭開始依序是輸入層的神經元數量、隱藏層的神經元數量、輸出層的神經元數量、權重初始化時的常態分布規模。
predict(self, x)	進行辨識（推論）。 引數 x 是影像資料。
loss(self, x, t)	計算損失函數。 引數 x 是影像資料，t 是正確答案標籤。
accuracy(self, x, t)	計算辨識準確度。
numerical_gradient(self, x, t)	利用數值微分計算權重參數的梯度（同上一章）。
gradient(self, x, t)	利用誤差反向傳播法計算權重參數的梯度。

這個類別的執行過程有些冗長，但是大部分的內容與上一章的「4.5 執行學習演算法」共通。與上一章不同的部分在於，這裡主要使用的是層級。利用層級傳播，就能達到處理辨識結果（`predict()`），計算梯度（`gradient()`）的目的，這就是 `TwoLayerNet`。

```

import sys, os
sys.path.append(os.pardir)
import numpy as np
from common.layers import *
from common.gradient import numerical_gradient
from collections import OrderedDict

class TwoLayerNet:

    def __init__(self, input_size, hidden_size, output_size,
                 weight_init_std=0.01):
        # 權重初始化
        self.params = {}
        self.params['W1'] = weight_init_std * \
            np.random.randn(input_size, hidden_size)
        self.params['b1'] = np.zeros(hidden_size)
        self.params['W2'] = weight_init_std * \
            np.random.randn(hidden_size, output_size)
        self.params['b2'] = np.zeros(output_size)

        # 產生各層
        self.layers = OrderedDict()
        self.layers['Affine1'] = \
            Affine(self.params['W1'], self.params['b1'])
        self.layers['Relu1'] = Relu()
        self.layers['Affine2'] = \
            Affine(self.params['W2'], self.params['b2'])

        self.lastLayer = SoftmaxWithLoss()

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

        return x

    # x: 輸入資料，t: 訓練資料
    def loss(self, x, t):
        y = self.predict(x)
        return self.lastLayer.forward(y, t)

    def accuracy(self, x, t):
        y = self.predict(x)

```

```

y = np.argmax(y, axis=1)
if t.ndim != 1 : t = np.argmax(t, axis=1)

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

# x: 輸入資料 , t: 訓練資料
def numerical_gradient(self, x, t):
    loss_W = lambda W: self.loss(x, t)

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

    return grads

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

    return grads

```

請注意上面用粗體字標示的程式。尤其是把神經網路的層當作 `OrderedDict` 這一點極為重要。`OrderedDict` 是有序字典。「有序」是指，可以記住新增至字典內的元素順序。因此，神經網路的正向傳播，只要按照順序呼叫出各層的 `forward()` 方法，就能處理完畢。另外，反向傳播也只要依照相反方向來呼叫各層即可。`Affine` 層及 `ReLU` 層各自在

內部正確處理正向傳播與反向傳播，而這裡進行的是，按照正確順序連結各層，並且依序（或相反順序）呼叫出各層即可。

把神經網路的構成元素當作「層」來執行，就可以輕鬆建構出神經網路。這種以「層」來模組化的優點非常強大。假如想要製作其他較大的網路，例如 5 層、10 層、20 層、…等，只要增加必要的層級，就可以製作神經網路（就像組合樂高一樣）。之後，在各層內部執行正向傳播與反向傳播，即可正確計算出辨識處理及學習需要的梯度。

5.7.3 誤差反向傳播法的梯度確認

到目前為止，已經說明了兩種計算梯度的方法：一種是利用數值微分，另一種是以解析算式的方式來計算梯度。後者所謂的解析，是利用誤差反向傳播法，即使有大量參數，也能快速完成計算。因此，接下來，請利用誤差反向傳播法計算梯度，別用比較花時間的數值微分。

數值微分的計算時間比較久。只要（正確）執行誤差反向傳播法，就不需要執行數值微分。那麼，數值微分有何用處？事實上，數值微分是用來確認誤差反向傳播法是否正確執行。

數值微分的優點是，執行過程很簡單。因此，數值微分比較不會出現錯誤，然而誤差反向傳播法較為複雜，所以一般而言，比較容易發生問題。因此，我們經常利用比較數值微分與誤差反向傳播法的結果，確認誤差反向傳播法是否正確執行。另外，確認數值微分計算結果與誤差反向傳播法計算出來的梯度一致（嚴格來說是幾乎近似值），這種步驟就稱作梯度檢查 (*gradient check*)。接下來，梯度檢查的執行過程如下所示（原始碼位於 ch05/gradient_check.py）。

```
import sys, os
sys.path.append(os.pardir)
import numpy as np
from dataset.mnist import load_mnist
from two_layer_net import TwoLayerNet

# 載入資料
(x_train, t_train), (x_test, t_test) = \load_mnist(normalize=True, one_hot_
label=True)

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

x_batch = x_train[:3]
t_batch = t_train[:3]
```

```

grad_numerical = network.numerical_gradient(x_batch, t_batch)
grad_backprop = network.gradient(x_batch, t_batch)

# 計算各權重的絕對誤差平均值
for key in grad_numerical.keys():
    diff = np.average( np.abs(grad_backprop[key] - grad_numerical[key]) )
    print(key + ":" + str(diff))

```

和前面一樣，載入 MNIST 資料集。使用部分訓練資料，確認數值微分算出來的梯度與誤差反向傳播法計算的結果所產生的誤差。這裡先算出各權重參數的元素差絕對值，再計算出平均值當作誤差。執行上面的程式之後，會輸出以下結果。

```

b1:9.70418809871e-13
W2:8.41139039497e-13
b2:1.1945999745e-10
W1:2.2232446644e-13

```

從結果可以得知，數值微分與誤差反向傳播法計算出來的梯度誤差非常小。例如，第 1 層偏權值的誤差是 9.7×10^{-13} (0.0000000000097)。由此可知，利用誤差反向傳播法計算出來的梯度是正確的，提高了誤差反向傳播法執行過程正確無誤的可信度。



數值微分與誤差反向傳播法的計算結果，誤差為 0 的情況非常罕見，這是電腦的運算精確度有限（例如，32 位元的浮點數）所造成。因為數值精確度的限制，使得剛才的誤差通常不會是 0。但是只要正確執行，誤差會趨近於 0。假如，誤差的數值較大，即可判斷誤差反向傳播法的執行過程有錯誤。

5.7.4 使用誤差反向傳播法學習

最後，要實際使用反向誤差傳播法來進行神經網路的學習。與前面不同的部分在於，這裡是利用誤差反向傳播法計算梯度。以下只列出程式，省略說明（原始碼位於 ch05/train_neuralnet.py）。

```

import sys, os
sys.path.append(os.pardir)
import numpy as np
from dataset.mnist import load_mnist
from two_layer_net import TwoLayerNet

# 載入資料
(x_train, t_train), (x_test, t_test) = \
    load_mnist(normalize=True, one_hot_label=True)

```

```

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

iters_num = 10000
train_size = x_train.shape[0]
batch_size = 100
learning_rate = 0.1
train_loss_list = []
train_acc_list = []
test_acc_list = []

iter_per_epoch = max(train_size / batch_size, 1)

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

    # 利用誤差反向傳播法計算梯度
    grad = network.gradient(x_batch, t_batch)

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

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

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

```

5.8 重點整理

本章學習了用視覺化方式顯示計算過程的計算圖。使用計算圖，說明神經網路執行的誤差反向傳播法，還有以層為單位，在神經網路進行處理。例如，ReLU 層、Softmax-with-Loss 層、Affine 層、Softmax 層等。在各層執行 `forward` 與 `backward` 等方法，正向或反向傳播資料，可以快速計算出權重參數的梯度。利用「層」進行模組化，可以在神經網路中，隨意組合各層，輕鬆製作出想要的網路。

本章學到的重點

- 使用計算圖，可以用視覺化方式掌握計算過程。
- 計算圖的節點是由局部性計算所構成，局部性計算能構成整個計算。
- 計算圖的正向傳播是進行一般運算。利用計算圖的反向傳播，可以計算出各節點的微分。
- 把神經網路的構成元素當作「層」來執行處理，可以快速計算出梯度（誤差反向傳播法）。
- 比較數值微分與誤差反向傳播法，可以確認誤差反向傳播法的執行過程有沒有錯誤（梯度檢查）。