

# 與學習有關的技巧

本章要說明在神經網路的學習中，成為關鍵的重要部分。本章挑選的主題包括，找出最佳參數的最佳化手法、權重參數的預設值、超參數的設定方法等，每個都是在神經網路的學習中，十分重要的主題。另外，還會扼要說明並執行 Weight decay 與 Dropout 等正規化手法，當作過度學習的解決對策。最後，簡單說明近年來眾多研究中常用到的 Batch Normalization 手法。使用本章描述的手法，可以快速進行神經網路（深度學習）的學習，提高辨識準確度。接下來，開始進入本章的內容。

## 6.1 更新參數

神經網路的學習目的，就是找出可以盡量縮小損失函數的參數，亦即找出最佳參數，解決這種問題的做法，稱作最佳化 (*optimization*)。可惜的是，神經網路的最佳化是一個非常困難的問題。因為參數空間十分複雜，很難輕易找到最佳解答（得不到解開算式，瞬間取得最小值的方法）。此外，在多層網路中，參數的數量變得非常龐大，情況會更嚴重。

我們到目前為止，為了找出最佳參數，計算了參數的梯度（微分）。重複執行利用參數的梯度，往梯度方向更新參數，最後會逐漸趨近最佳參數，這種方法稱作準確率梯度下降法 (*stochastic gradient descent*)，簡稱 SGD，是非常單純的手法。比起盲目搜尋參數空間，這樣的做法的確比較「聰明」。可是，畢竟 SGD 只是單純的方法（視問題而異），還有比 SGD 更聰明的方法存在。以下要說明 SGD 的缺點，並且介紹其他最佳化手法。

### 6.1.1 冒險家的故事

在進入正題之前，我們先用一個「比喻」來形容我們在最佳化遇到的狀況。

有位瘋狂的冒險家，他在廣大的乾燥地帶旅行，尋找深邃的谷底。他的目標是，到達最深的谷底，他稱之為「深淵」，這就是他旅行的目的。此外，他還給了自己兩個嚴格的「限制」。一個是不看地圖，另一個是蒙上眼睛。因此，他不曉得在這片廣大的土地中，是否存在著最低的谷底。再加上，外面什麼都看不到。在這種嚴苛的條件下，這位冒險家該如何找到「深淵」？如何前進，才可以快速找到「深淵」？

搜尋最佳參數時，我們面臨的狀況，和這位冒險家一樣，盡是一片漆黑的世界。我們必須在廣大且複雜的地形，沒有地圖，而且蒙起眼睛的情況下，找到「深淵」。你應該不難想像，這是非常困難的問題。

在這種困難狀況中，最重要的是地面的「傾斜」狀態。冒險家看不到周遭的景色，卻可以瞭解目前地面的傾斜狀況（地面斜度能從腳底往上傳遞）。於是，朝著目前傾斜最明顯的方向前進，就是 SGD 的策略。重複進行這個步驟，遲早有一天，可以到達「深淵」，勇敢的冒險家這麼想著。

### 6.1.2 SGD

在你感受到最佳化問題的難度之後，這裡先複習一下 SGD。我們可以用算式（6.1）快速顯示出 SGD。

$$\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial L}{\partial \mathbf{W}} \quad (6.1)$$

這裡的更新權重參數為  $\mathbf{W}$ ，與  $\mathbf{W}$  有關的損失函數梯度為  $\frac{\partial L}{\partial \mathbf{W}}$ 。 $\eta$  代表學習率，實際上使用的是事先決定的 0.01 或 0.001 等數值。另外，算式中的  $\leftarrow$  代表利用右邊的值來更新左邊的值。如算式（6.1）所示，SGD 是往梯度方向前進一定距離的單純方法。接下來，把 SGD 當作 Python 的類別來執行（考量到後續的易用性，而利用名為 SGD 的類別來執行）。

```
class SGD:
    def __init__(self, lr=0.01):
        self.lr = lr
```

```
def update(self, params, grads):
    for key in params.keys():
        params[key] -= self.lr * grads[key]
```

初始化時的引數 `lr` 代表 learning rate（學習率），並且把學習率當作實例變數。另外，還定義了 `update(params, grads)` 方法。在 SGD 中，會重複呼叫這個方法。引數 `params` 與 `grads` 是（和前面的神經網路執行過程一樣）字典變數。如同 `params['W1']`、`grads['W1']` 等，分別儲存權重參數與梯度。

使用 SGD 類別，可以依照以下所示，更新神經網路的參數（下面顯示的原始碼是虛擬原始碼，無法實際運作）。

```
network = TwoLayerNet(...)
optimizer = SGD()

for i in range(10000):
    ...
    x_batch, t_batch = get_mini_batch(...) # 小批次
    grads = network.gradient(x_batch, t_batch)
    params = network.params
    optimizer.update(params, grads)
    ...
```

這裡出現的變數名稱 `optimizer` 是指，「進行最佳化者」的意思。SGD 就是扮演這個角色，而更新參數是由 `optimizer` 來負責執行。我們這裡要做的事情，只有把參數與梯度資料傳遞給 `optimizer`。

將進行最佳化的類別分離出來再執行，能輕易把功能模組化。例如，後續馬上就要執行的其他最佳化手法 Momentum。Momentum 也擁有共通的 `update(params, grads)` 方法，這樣只要把 `optimizer = SGD()` 改成 `optimizer = Momentum()`，就可以將 SGD 切換成 Momentum 了。



在大部分的深度學習框架中，都包含了各種最佳化手法，並且提供輕鬆切換的結構。例如，在 Lasagne 深度學習框架中，於 `updates.py`（請參考以下連結）的檔案裡，把最佳化手法當作函數來統一執行。使用者可以從中選擇想要的最佳化手法。

<http://github.com/Lasagne/Lasagne/blob/master/lasagne/updates.py>

### 6.1.3 SGD 的缺點

SGD 很單純，也容易執行，但是遇到部分問題，可能會變得沒有效率。以下將藉由思考函數最小值的問題，指出 SGD 的缺點。

$$f(x, y) = \frac{1}{20}x^2 + y^2 \quad (6.2)$$

算式 (6.2) 顯示的函數如圖 6-1 所示，是往  $x$  軸方向延伸成「碗」形狀的函數。實際上，算式 (6.2) 的等高線變成往  $x$  軸方向延伸的橢圓形。

接下來，要檢視算式 (6.2) 顯示的函數梯度。若用圖來顯示梯度，結果如圖 6-2 所示。這個梯度的特色是，往  $y$  軸方向變大，往  $x$  軸方向變小。換句話說， $y$  軸方向是大幅傾斜， $x$  軸方向為平緩傾斜。另外，這裡還要注意到，算式 (6.2) 的最小值位於  $(x, y) = (0, 0)$ ，但是圖 6-2 顯示的梯度，在大部分的位置都沒有指到  $(0, 0)$  的方向。

針對圖 6-1 形狀的函數套用 SGD。探索的位置（預設值）是從  $(x, y) = (-7.0, 2.0)$  開始。結果如圖 6-3 所示。

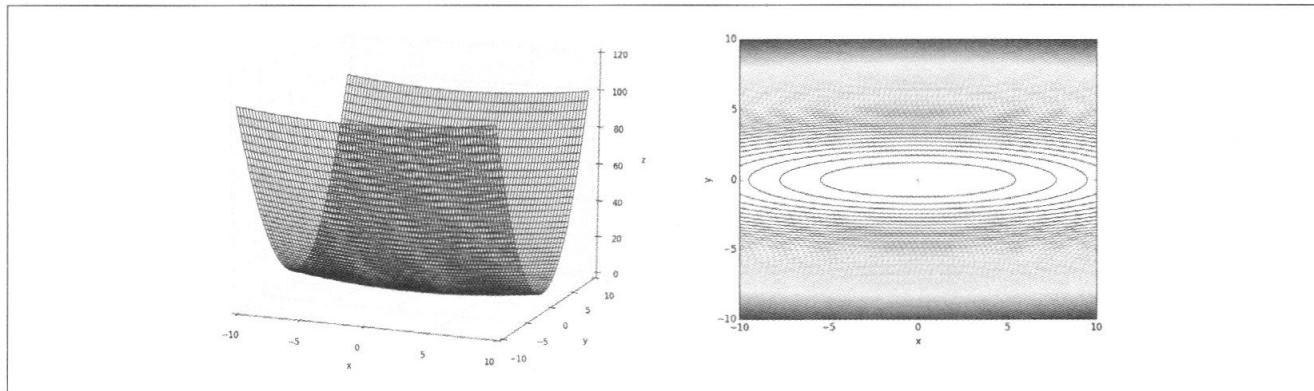


圖 6-1  $f(x, y) = \frac{1}{20}x^2 + y^2$  的圖（左圖）與等高線（右圖）

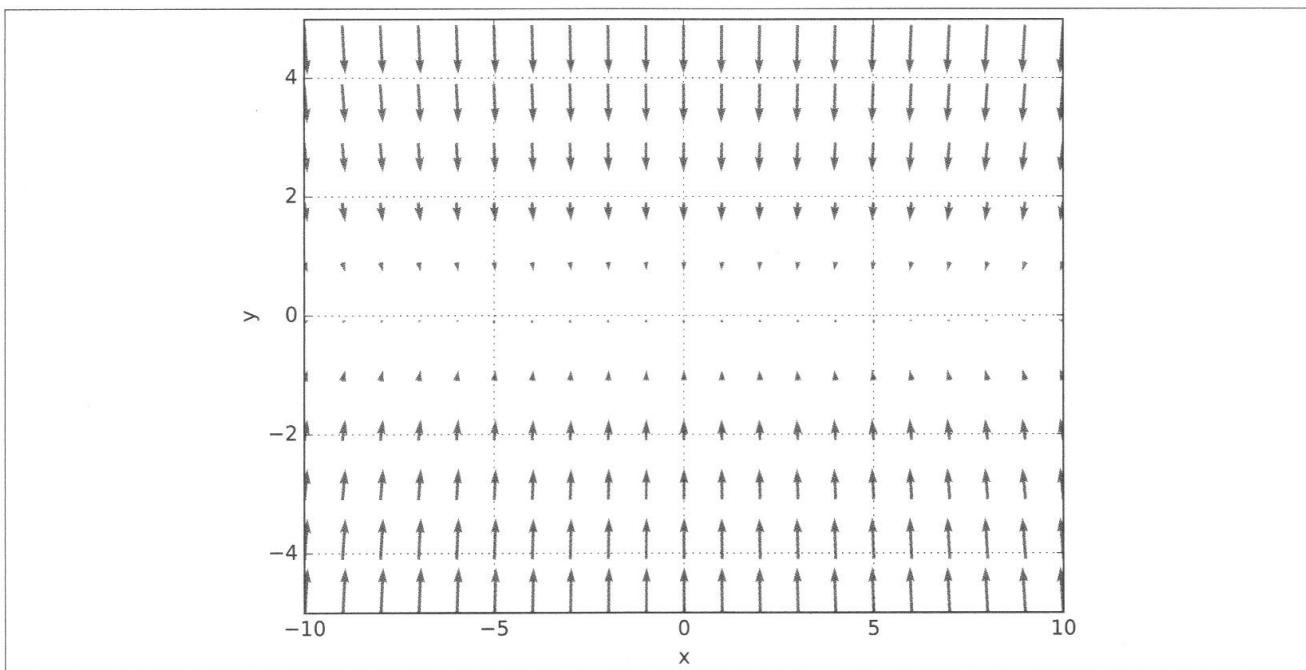


圖 6-2  $f(x, y) = \frac{1}{20}x^2 + y^2$  的梯度

SGD 如圖 6-3 所示，呈鋸齒狀移動。這是非常沒有效率的路徑。換句話說，SGD 的缺點是，函數的形狀如果沒有等向性，非延伸形狀的函數，就會以沒有效率的路徑來進行探索。因此，需要尋找比 SGD 這種單純往梯度方向前進，更聰明的方法。SGD 形成無效率探索路徑的根本原因在於，梯度方向原本就不是指向最小值。

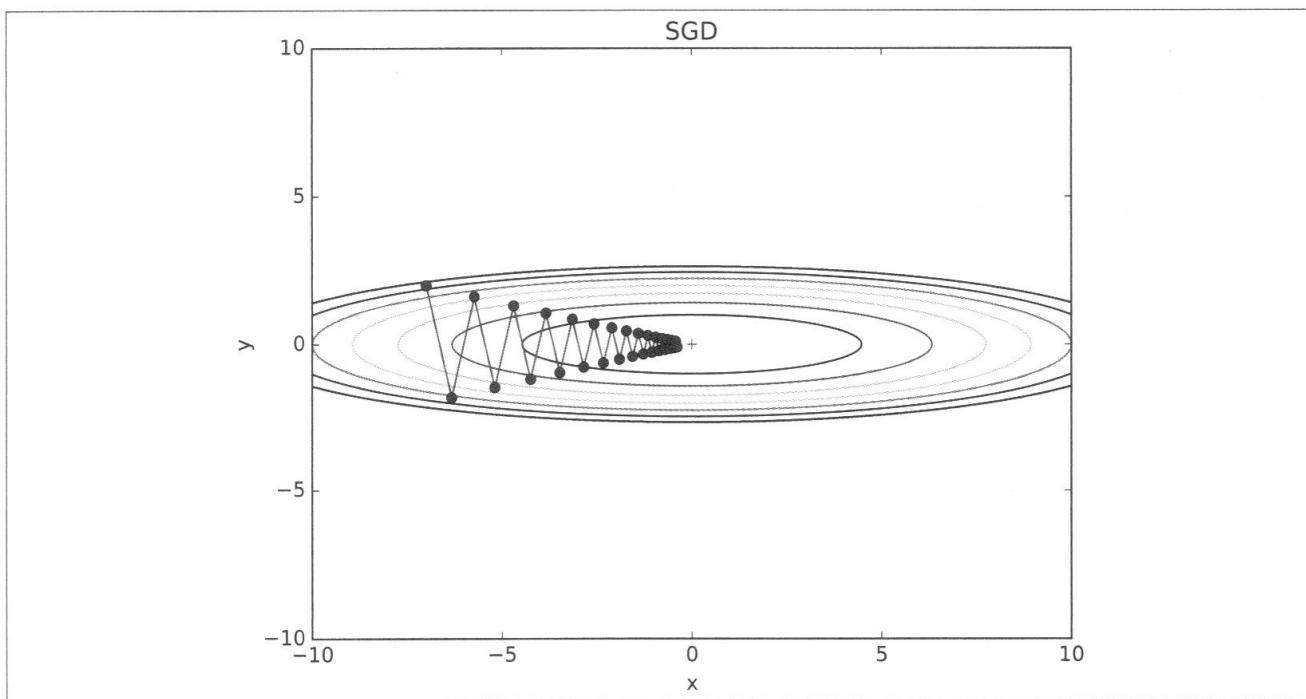


圖 6-3 SGD 最佳化的更新路徑：以鋸齒狀往最小值 (0, 0) 移動，效率不佳

為了改善 SGD 的缺點，接下來要介紹 Momentum、AdaGrad、Adam 等取代 SGD 的 3 種手法。先分別簡單說明，再顯示算式與 Python 的執行過程。

### 6.1.4 Momentum

Momentum 這個字是「運動量」的意思，與物理有關係。Momentum 手法可以用以下算式表示。

$$\mathbf{v} \leftarrow \alpha \mathbf{v} - \eta \frac{\partial L}{\partial \mathbf{W}} \quad (6.3)$$

$$\mathbf{W} \leftarrow \mathbf{W} + \mathbf{v} \quad (6.4)$$

和前面的 SGD 一樣， $\mathbf{W}$  是更新的權重參數， $\frac{\partial L}{\partial \mathbf{W}}$  是與  $\mathbf{W}$  有關的損失函數梯度， $\eta$  代表學習率。這裡出現了新的變數  $\mathbf{v}$ ，站在物理的角度， $\mathbf{v}$  是對應「速度」。算式 (6.3) 是代表，物體往梯度方向受力，並將這個力量加上物體速度的物理定律。Momentum 可以想像成球在地面滾動的狀態，如圖 6-4 所示。



圖 6-4 Momentum 的示意圖：球在傾斜的地面上滾動

另外，算式 (6.3) 出現了  $\alpha \mathbf{v}$ ，這是當物體沒有受力時，逐漸減速的功能 ( $\alpha$  設定為 0.9 等數值)。站在物理學的角度，相當於地面摩擦力或空氣阻力。以下是 Momentum 的執行過程（原始碼位於 common/optimizer.py）。

```
class Momentum:
    def __init__(self, lr=0.01, momentum=0.9):
        self.lr = lr
        self.momentum = momentum
        self.v = None

    def update(self, params, grads):
        if self.v is None:
            self.v = {}
        for key, val in params.items():
            self.v[key] = np.zeros_like(val)
```

```

for key in params.keys():
    self.v[key] = self.momentum*self.v[key] - self.lr*grads[key]
    params[key] += self.v[key]

```

實例變數  $v$  是保持物體的速度。 $v$  在初始化時，沒有保持任何數值，但是當 `update()` 首度呼叫時，會把和參數相同結構的資料當作字典變數維持下來。剩下是算式（6.3）與（6.4）的簡單執行過程。

接下來，試著使用 Momentum，解開算式（6.2）的最佳化問題，結果如圖 6-5 所示。

圖 6-5 的更新路徑呈現出球在碗內打轉般的移動狀態。由此可以看出，與 SGD 相比，減少了「起伏程度」。這是因為  $x$  軸方向的受力非常小，但是承受的施力為同方向，所以固定往相同方向加速。相對來說， $y$  軸方向受力較大，卻因為正負方向穿插受力而彼此抵銷，使得  $y$  軸方向的速度不穩定。與 SGD 相比，比較快接近  $x$  軸方向，可以減少起伏程度。

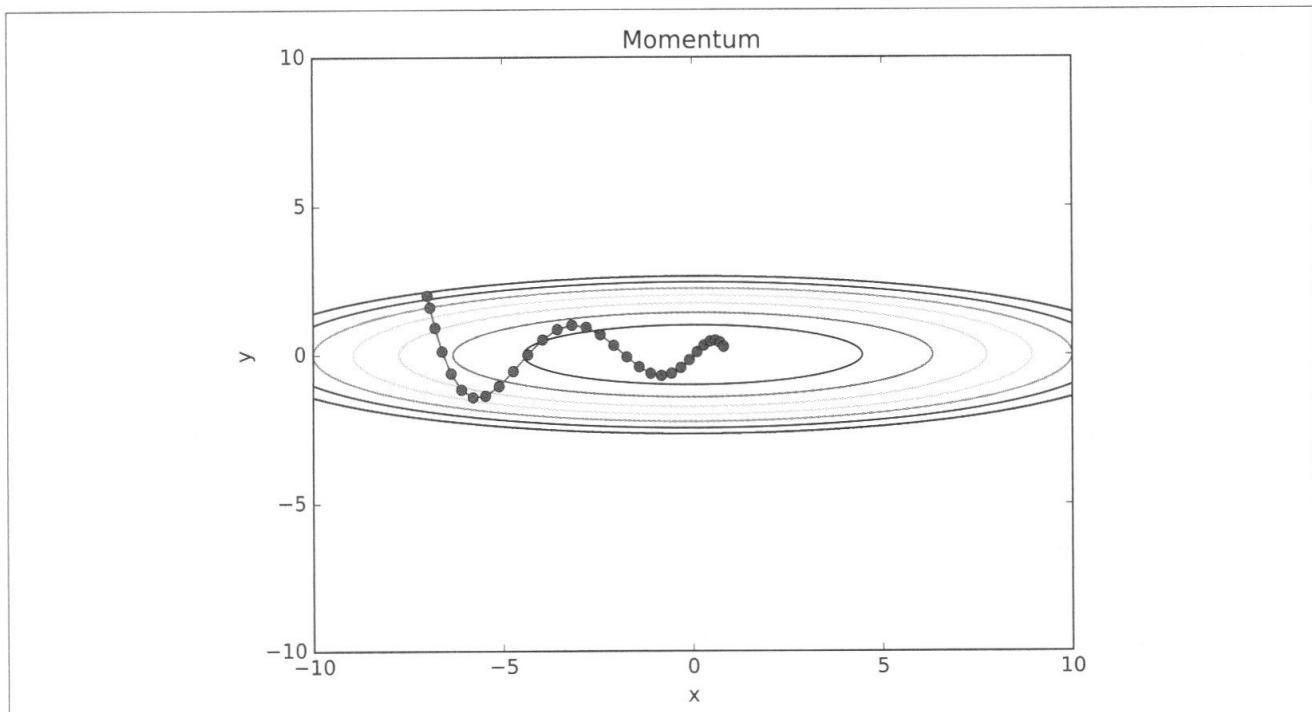


圖 6-5 Momentum 最佳化的更新路徑

### 6.1.5 AdaGrad

在神經網路的學習中，學習率（算式中顯示為  $\eta$ ）的值非常重要。學習率太小，會花費太多時間在學習上；相對來說，學習率太大，就會往外擴散，無法正確學習。

有個與學習率有關的有效技巧，稱作學習率衰減 (*learning rate decay*)。這是隨著學習過程來縮小學習率的方法。事實上，在神經網路的學習中，經常使用開始為「大」學習，接著是「小」學習的手法。

逐漸降低學習率的概念，相當於統一降低參數的「整個」學習率。進一步發展之後，就成為 AdaGrad [6]。AdaGrad 是針對「每個」參數，準備「客製」值。

AdaGrad 是適應各個參數的元素，一邊調整學習率，一邊學習的手法（AdaGrad 的 Ada 有「適應」的意思，源自於 Adaptive）。接下來，用算式來顯示 AdaGrad 的更新方法。

$$\mathbf{h} \leftarrow \mathbf{h} + \frac{\partial L}{\partial \mathbf{W}} \odot \frac{\partial L}{\partial \mathbf{W}} \quad (6.5)$$

$$\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{1}{\sqrt{\mathbf{h}}} \frac{\partial L}{\partial \mathbf{W}} \quad (6.6)$$

和前面 SGD 一樣， $\mathbf{W}$  是更新的權重參數， $\frac{\partial L}{\partial \mathbf{W}}$  是與  $\mathbf{W}$  有關的損失函數梯度， $\eta$  代表學習率。這裡出現新的變數  $\mathbf{h}$ 。如算式 (6.5) 所示， $\mathbf{h}$  維持為前面提到的梯度值平方和（算式 (6.5) 的  $\odot$  代表矩陣各元素相乘）。更新參數時，乘上  $\frac{1}{\sqrt{\mathbf{h}}}$ ，調整學習規模。這是代表在參數的元素中，經常變動（放大）的元素，學習率會變小的意思。由於經常變動的參數，學習率會逐漸變小，所以我們可以針對參數的各個元素，執行學習率衰減。



AdaGrad 是把過去的梯度當作平方和，全都記錄下來。因此，在不斷學習之下，更新幅度會變小。實際上，無限學習下去，更新量會變成 0，完全不再更新。改善這個問題的手法，就是 RMSProp [7]。RMSProp 並非把過去所有的梯度一律相加，而是逐漸忘記過去的梯度，以大幅反應新梯度資料的方式加總。專有名詞稱作「指數移動平均」，以指數函數減少過去梯度的規模。

接下來，要執行 AdaGrad，依照下列程式執行處理（原始碼位於 `common/optimizer.py`）。

```
class AdaGrad:
    def __init__(self, lr=0.01):
        self.lr = lr
        self.h = None

    def update(self, params, grads):
        if self.h is None:
            self.h = {}
        for key, val in params.items():
            self.h[key] = np.zeros_like(val)

        for key in params.keys():
            self.h[key] += grads[key] * grads[key]
            params[key] -= self.lr * grads[key] / (np.sqrt(self.h[key]) + 1e-7)
```

這裡必須注意到，最後一行加上了小數值 `1e-7` 這一點。這是為了避免當 `self.h[key]` 出現 0 時，發生除以 0 的問題。在大部分的深度學習框架中，這種小數值也可以設定成參數，但是這裡使用固定值 `1e-7`。

接下來，試著使用 AdaGrad，解開算式（6.2）的最佳化問題，結果如圖 6-6 所示。

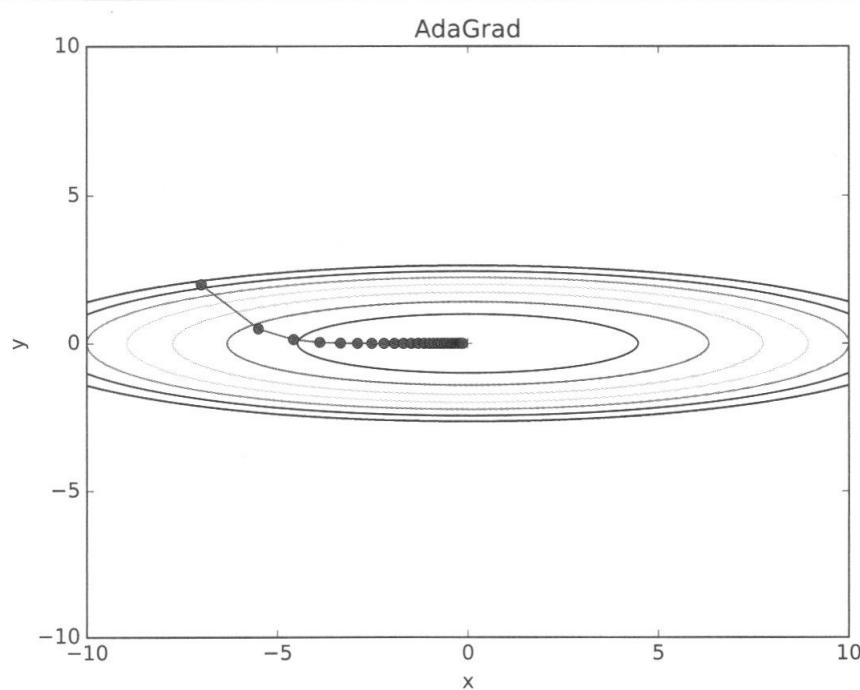


圖 6-6 利用 AdaGrad 最佳化更新路徑

檢視圖 6-6 的結果，可以看到有效率地往最小值移動。由於往  $y$  軸方向的梯度較大，所以剛開始會大幅移動，根據大幅移動的比例，縮小更新步驟，讓朝著  $y$  軸方向的更新程度變弱，減少起伏。

### 6.1.6 Adam

Momentum 是依照球在碗內來回滾動的物理定律為基準來移動，而 AdaGrad 是依照各個參數的元素，調整適應的更新步驟。假如將 Momentum 與 AdaGrad 這兩種手法融合，結果會變成如何？這就是 Adam [8] 手法的基本雛型<sup>1</sup>。

Adam 是在 2015 年提出的新手法，這個理論有點複雜，但是直覺來說，就是將 Momentum 與 AdaGrad 融合在一起。組合這兩種手法的優點，可以有效探索參數空間。另外，還能進行超參數的「偏權值校正（偏離校正）」，這也是 Adam 的特色之一。這裡不深入探討，詳細說明請參考原著論文 [8]。另外，關於 Python 的執行，在 common/optimizer.py 中，已經用 Adam 類別來處理，有興趣的讀者，可以自行參考。

接下來，要使用 Adam 解決算式 (6.2) 的最佳化問題。結果如圖 6-7 所示。

在圖 6-7 中，使用 Adam 的更新過程就像球在碗內滾動般移動。與 Momentum 的動作類似，但是與使用 Momentum 時相比，減少了球左右搖晃的幅度。這是因為 Adam 可以適應調整學習的更新程度。



Adam 會設定 3 個超參數，一個是前面說明過的學習率（在論文中，顯示為  $\alpha$ ）。另外 2 個是第一時刻係數  $\beta_1$  以及第二時刻係數  $\beta_2$ 。根據論文的說明，標準的設定值是  $\beta_1$  為 0.9， $\beta_2$  是 0.999，使用這個設定值，大部分的情況都可以順利執行。

### 6.1.7 該使用何種更新手法？

到目前為止介紹了 4 種更新方法。以下要比較這 4 種方法（原始碼位於 ch06/optimizer\_compare\_naive.py）。

<sup>1</sup> 這裡的 Adam 手法，是以直覺方式來說明，並非完全正確。詳細說明請參考原著論文。

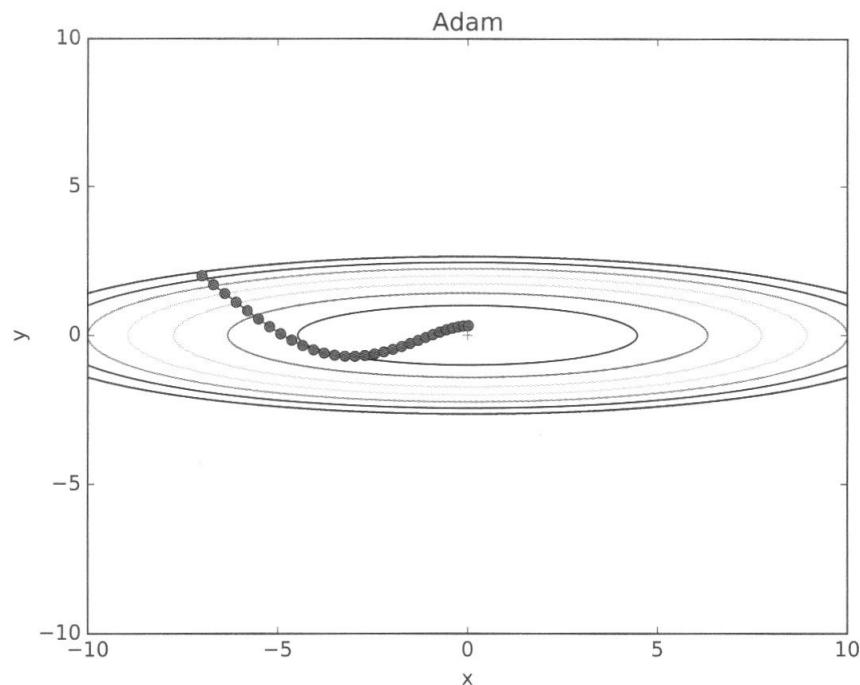


圖 6-7 利用 Adam 最佳化的更新路徑

如圖 6-8 所示，可以瞭解，使用不同方法，會以不同路徑來更新。單獨看這張圖，會覺得 AdaGrad 是最好的手法，但是這裡必須注意到，結果會隨著要解決的問題而改變。當然，超參數（學習率等）的設定值也會影響結果。

到目前為止，說明了 SGD、Momentum、AdaGrad、Adam 等 4 種手法，究竟該使用哪一種比較好？可惜（目前）沒有一種手法能完美解決所有問題。每種手法各有特色，也有各自擅長與不擅長解決的問題。

在眾多研究中，至今仍繼續使用 SGD。Momentum 與 AdaGrad 也是值得嘗試的手法。最近有許多研究者或技術人員偏愛使用 Adam。本書主要使用的是 SGD 與 Adam，請你依照個人喜好，試著嘗試各種手法。

### 6.1.8 利用 MNIST 資料集比較更新手法

以下將以辨識手寫數字為對象，比較前面說明過的 SGD、Momentum、AdaGrad、Adam 等 4 種手法。確認使用每種手法的學習進展有何不同。以下先顯示結果，如圖 6-9 所示（原始碼位於 `ch06/optimizer_compare_mnist.py`）。

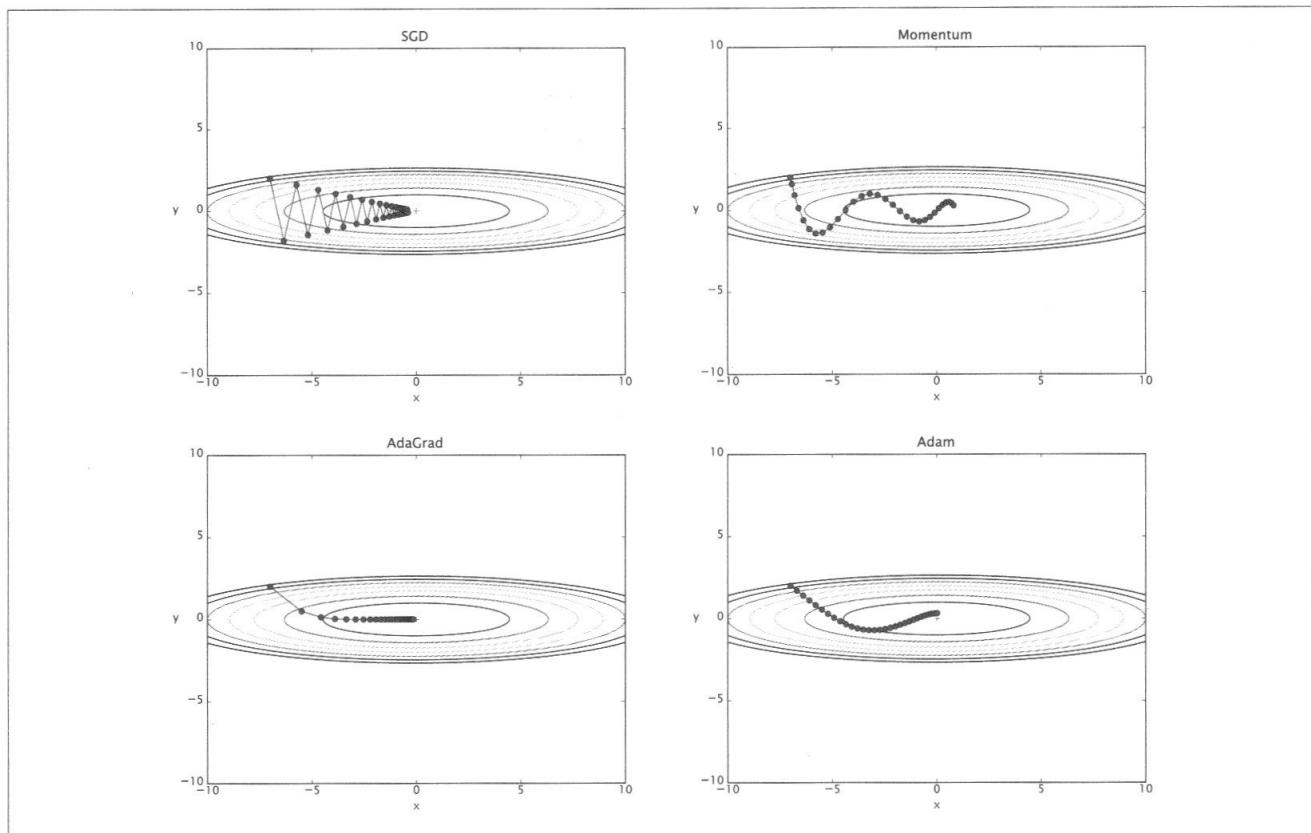


圖 6-8 比較最佳化手法：SGD、Momentum、AdaGrad、Adam

這個實驗以 5 層神經網路，各層擁有 100 個神經元的網路為對象，活化函數使用的是 ReLU。

檢視圖 6-9 的結果，可以瞭解其他方法比 SGD 的學習速度更快。仔細檢視這 3 種手法，會發現 AdaGrad 的學習速度稍微快一點。但是我們必須注意到，這個實驗結果會受到學習率的超參數、神經網路的結構（深度有幾層等）影響，而產生變化。但是，一般而言，與 SGD 相比，其他 3 種手法的學習速度的確較快，有時最後的辨識效能也比較好。

## 6.2 權重的預設值

在神經網路的學習中，特別重要的就是權重的預設值。事實上，權重的預設值應該設定成哪種數值，常會影響到神經網路的學習成功與否。本節將針對建議的權重預設值來說明，利用實驗，確認實際的神經網路學習速度。

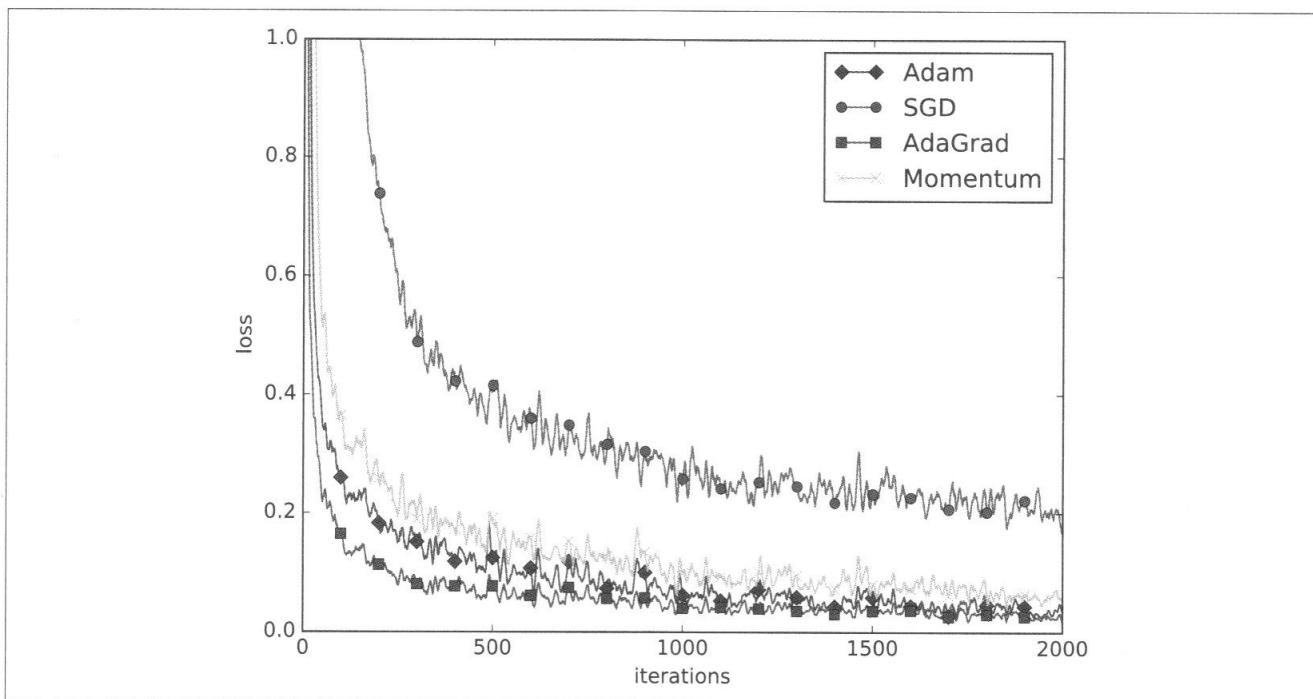


圖 6-9 利用 MNIST 資料集比較四種更新手法：水平軸是重複學習的次數（iterations），垂直軸是損失函數的值（loss）

## 6.2.1 權重的預設值變成 0 ?

後面將會介紹 Weight decay（權重衰減）手法，當作控制過度學習，提高一般化功能的技巧。簡單來說，Weight decay 是以縮小權重參數值為目的，進行學習的手法。縮小權重值，可以避免過度學習。

如果想要縮小權重值，預設值也盡量從小數值開始，這是屬於正攻法。事實上，到目前為止，權重的預設值如 `0.01 * np.random.randn(10, 100)` 所示，使用的是將常態分布產生的值，縮小 0.01 倍的小數值（標準差為 0.01 的常態分布）。

想要縮小權重值，把權重的預設值全都設定成 0 不就好了？在此先回答這個問題，其實將權重的預設值變成 0，是很糟糕的想法。事實上，當權重的預設值變成 0，就不會正確學習。

為什麼權重的預設值不能為 0？正確來說是權重為何不能設定成均一值？因為在誤差反向傳播法中，所有的權重值都會均一（同樣）更新。假設在雙層神經網路中，把第 1 層與第 2 層的權重設定為 0，進行正向傳播時，因為輸入層的權重為 0，會傳遞相同值給第 2 層神經元。第 2 層神經元全都輸入相同值，在反向傳播時，第 2 層的權重會更新成一樣（請回想起「乘法節點的反向傳播」）。因此，以均一值更新，權重會變成具有對稱性的數值（重複值）。這樣擁有眾多權重也毫無意義。為了避免「權重變成均一值」，正確來說是要破壞權重的對稱結果，需要隨機的預設值。

## 6.2.2 隱藏層的活性化分布

觀察隱藏層的活性化（Activation）<sup>2</sup>（活化函數之後的輸出資料）分布，可以得到許多資料。以下想進行一個簡單的實驗，觀察隱藏層的活性化會隨著權重的預設值產生何種變化。這裡進行的實驗是，在 5 層神經網路（活化函數使用 sigmoid 函數）中，傳遞隨機產生的輸入資料，利用分布圖繪製各層的活性化資料分布。另外，這個實驗參考了史丹佛大學的課程「CS231n」[5]。

此實驗用的原始碼位於 ch06/weight\_init\_activation\_histogram.py。以下只顯示部分程式。

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

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

x = np.random.randn(1000, 100) # 1000 個資料
node_num = 100          # 各隱藏層的節點（神經元）數量
hidden_layer_size = 5 # 隱藏層有 5 層
activations = {}        # 在這裡儲存 Activation 的結果

for i in range(hidden_layer_size):
    if i != 0:
        x = activations[i-1]

    w = np.random.randn(node_num, node_num) * 1

    z = np.dot(x, w)
    a = sigmoid(z) # sigmoid 函數！
    activations[i] = a
```

2 這裡把活化函數之後的輸出資料稱作「活性化（Activation）」，但是根據文獻記載，在各層之間流動的資料，也稱作「活性化（Activation）」。

這裡以 5 層且每層有 100 個神經元的神經網路為對象，再以常態分布隨機產生 1,000 個資料當作輸入資料，傳遞給 5 層神經網路。活化函數使用的是 sigmoid 函數，各層的活性化結果儲存在 `activations` 變數中。這個程式要注意的重點是，權重的規模。這次使用了標準差為 1 的常態分布，我們要觀察改變規模（標準差）時，活性化分布會產生何種變化，這也是此實驗的目的。接下來，把儲存在 `activations` 變數中的各層資料繪製成分布圖。

```
# 繪製分布圖
for i, a in activations.items():
    plt.subplot(1, len(activations), i+1)
    plt.title(str(i+1) + "-layer")
    plt.hist(a.flatten(), 30, range=(0,1))
plt.show()
```

執行此原始碼，可以得到圖 6-10 的分布圖。

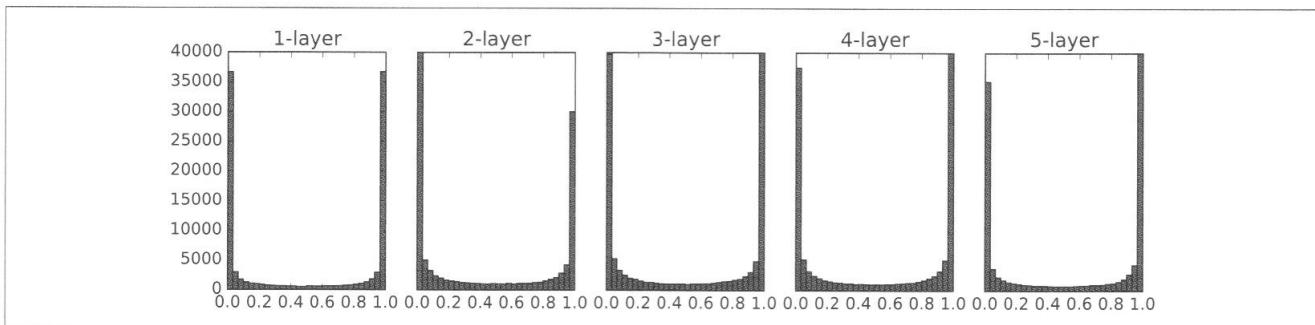


圖 6-10 使用標準差 1 的常態分布當作權重的預設值時，各層的活性化分布

檢視圖 6-10，能看出各層的活性化分布偏向 0 與 1。這裡使用的 sigmoid 函數是呈現 S 型曲線的函數，但是隨著 sigmoid 函數的輸出趨近於 0（或趨近 1），該微分值也會趨近於 0。因此，在偏向 0 與 1 的資料分布中，反向傳播的梯度值會逐漸變小、消失。這就是所謂的梯度消失 (*gradient vanishing*) 問題。在層數較多的深度學習中，梯度消失會造成嚴重的問題。

接著把權重的標準差當成 0.01 來進行相同實驗。在實驗的原始碼中，只要把設定權重預設值的部分更換成以下原始碼即可。

```
# w = np.random.randn(node_num, node_num) * 1
w = np.random.randn(node_num, node_num) * 0.01
```

接著，一起來檢視結果。如果是標準差 0.01 的常態分布，各層的活性化分布如圖 6-11 所示。

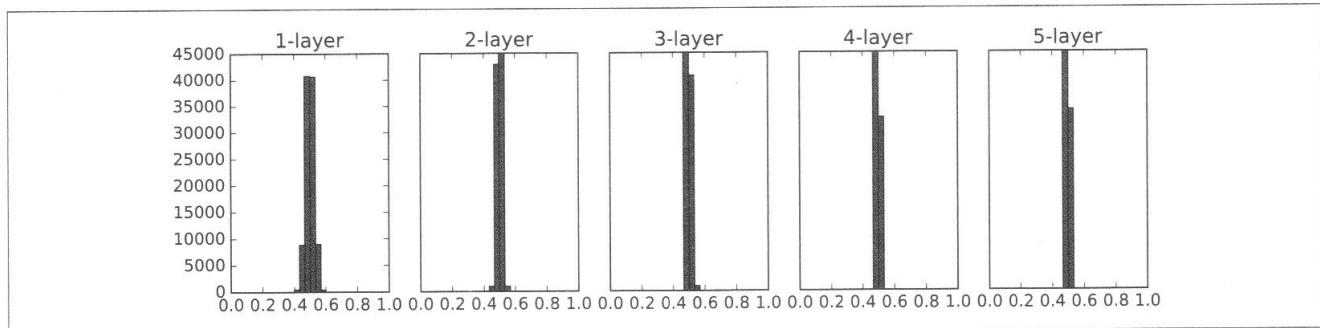


圖 6-11 使用標準差 0.01 常態分布當作權重預設值時，各層的活性化分布

這次集中分布在 0.5 附近。和前面範例偏向 0 與 1 的情況不同，沒有引起梯度消失問題。可是，活性化分布出現特定偏差，在表現力方面，將會產生重大問題。因為，當多個神經元輸出幾乎一模一樣的數值時，就沒有存在的意義。例如，100 個神經元輸出幾乎相同數值，代表只用 1 個神經元，就可以表現出一樣的情況。因此，活性化的偏差有著「表現力受限」的問題。



各層的活性化分布需要擁有適當的廣泛度。因為在各層傳遞具有適當多樣性的資料，才能有效進行神經網路的學習；相對來說，單一偏差的資料會造成梯度消失、「表現力受限」等問題，因而無法順利學習。

接下來，要使用 Xavier Glorot 在論文 [9] 中推薦的權重預設值（通稱「Xavier 預設值」）。現在，在一般深度學習的框架中，標準都會使用「Xavier 預設值」。例如，在 Caffe 框架中，權重的預設值設定成 xavier 引數，就可以使用「Xavier 預設值」。

另外，在 Xavier 的論文中，以讓各層的活性化呈現廣泛分布為目的，導出了適當的權重值。其得到的結論是，假設上層節點有  $n$  個，要使用具有  $\frac{1}{\sqrt{n}}$  標準差的常態分布來初始化<sup>3</sup>（圖 6-12）。

<sup>3</sup> 在 Xavier 的論文中，提出了考量到上層輸入節點個數及下層輸出節點個數的設定值。但是，如同上面的說明，在執行 Caffe 等框架時，簡化成只利用上層輸入節點來進行計算。

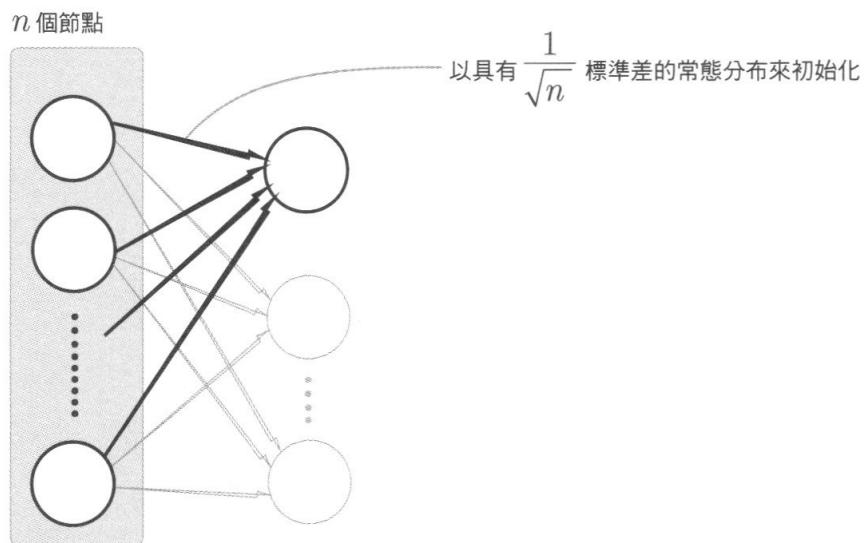


圖 6-12 Xavier 的預設值：假設上層有  $n$  個節點，以具有  $\frac{1}{\sqrt{n}}$  標準差的常態分布來初始化

使用「Xavier 預設值」，上層的節點數量愈多，當作目標對象的節點預設值來設定的權重就愈小。接下來，就利用「Xavier 預設值」進行實驗。你只要把實驗用程式中的權重預設值更換成以下內容即可（這裡將實驗單純化，每層的節點數量皆為 100 個）。

```
node_num = 100 # 上一層的節點數量
w = np.random.randn(node_num, node_num) / np.sqrt(node_num)
```

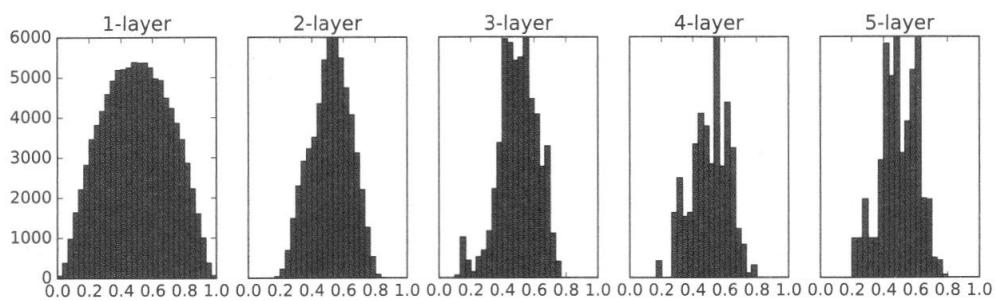


圖 6-13 使用「Xavier 預設值」當作權重的預設值時，各層的活性化分布

使用「Xavier 預設值」的結果如圖 6-13 所示。從結果中可以得知，愈往上層，會變成略微不規則的形狀，但是與之前相比，分布變得比較廣。由於各層傳遞的資料有著適當的廣度，sigmoid 函數的表現力沒有受到限制，能有效進行學習。



圖 6-13 的上層分布形狀變得有些不規則。這裡的不規則形狀可以用  $\tanh$  函數（雙曲線函數）取代  $\text{sigmoid}$  函數來改善。實際上，使用  $\tanh$  函數，能形成整齊的鐘型分布。 $\tanh$  函數和  $\text{sigmoid}$  函數一樣，都是呈 S 型曲線的函數，但是  $\tanh$  函數在原點  $(0, 0)$  呈對稱的 S 型曲線，而  $\text{sigmoid}$  函數是在  $(x, y) = (0, 0.5)$  呈對稱的 S 型曲線。一般而言，活化函數使用的函數最好擁有原點對稱的性質。

### 6.2.3 ReLU 的權重預設值

「Xavier 預設值」是在活化函數為線性的前提之下，所導出來的結果。而  $\text{sigmoid}$  函數與  $\tanh$  函數是在中央附近左右對稱的線性函數，所以很適合「Xavier 預設值」。然而，使用 ReLU 時，最好利用 ReLU 專用的預設值，就是 Kaiming He 推薦的「He 預設值」[10]。「He 預設值」是在上層節點數量為  $n$  個時，使用標準差為  $\sqrt{\frac{2}{n}}$  的常態分布。假設「Xavier 預設值」為  $\sqrt{\frac{1}{n}}$ ，ReLU 的負領域會變成 0，因此可以（直覺）解釋成，為了變得更廣泛，需要加倍的係數。

接著，要檢視活化函數使用 ReLU 時，呈現出來的活性化分布。首先是標準差為 0.01 的常態分布（後續簡寫成「 $\text{std} = 0.01$ 」），接著是「Xavier 預設值」，再來是 ReLU 專用的「He 預設值」，針對這三種情況，進行實驗，並且顯示結果（圖 6-14）。

從實驗結果可以得知，如果「 $\text{std} = 0.01$ 」，各層活性化的值會變成非常小<sup>4</sup>。在神經網路上，傳遞非常小的資料，代表反向傳播時，權重的梯度也同樣會變小。這將造成嚴重問題，事實上在此種情況下，幾乎無法繼續學習。

接著是「Xavier 預設值」的結果，隨著層數加深，偏差逐漸變大。當層數愈深，活性化的偏差就愈大，使得學習時出現「梯度消失」的問題。然而，「He 預設值」會讓各層分布的廣度變均勻。即使層數變深，資料廣度仍會保持均一，所以在反向傳播時，也能傳遞適當的值。

整理以上內容，活化函數為 ReLU 時，使用「He 預設值」，若是  $\text{sigmoid}$  或  $\tanh$  等 S 型曲線，則使用「Xavier 預設值」，是目前最好的做法。

<sup>4</sup> 各層的活性化分布平均如下。第 1 層：0.0396、第 2 層：0.00290、第 3 層：0.000197、第 4 層：1.32e-5、第 5 層：9.46e-7。

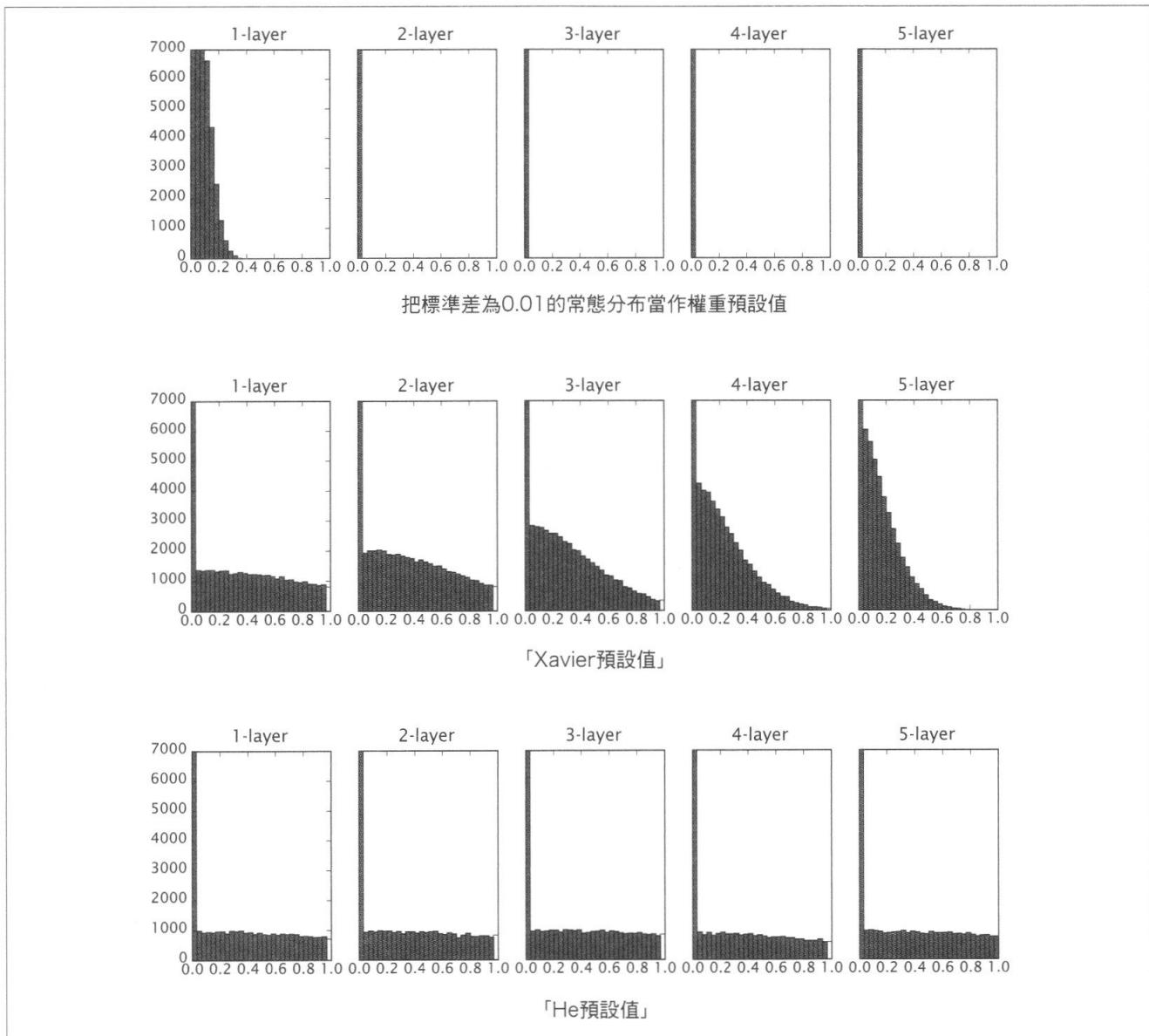


圖 6-14 使用 ReLU 當作活化函數，觀察不同權重預設值的活性化分布變化

## 6.2.4 利用 MNIST 資料集比較權重預設值

讓我們以實際資料為對象，提供不同的權重預設值，檢視對神經網路的學習會造成何種影響。以下利用「 $\text{std} = 0.01$ 」、「Xavier 預設值」、「He 預設值」等三種情況來進行實驗（原始碼位於 `ch06/weight_init_compare.py`），結果如圖 6-15 所示。

在這個實驗中，以 5 層神經網路（各層有 100 個神經元）為對象，使用 ReLU 當作活化函數。檢視圖 6-15 的結果可以得知，當「 $\text{std} = 0.01$ 」時，完全不會學習。因為，和剛才觀察的活性化分布一樣，正向傳播傳遞的數值非常小（集中在 0 的資料），使得反向

傳播時，計算出來的梯度也會變小，因而幾乎不會更新權重。相對來說，使用 Xavier 或 He 預設值時，可以順利學習。而且「He 預設值」的學習速度比較快。

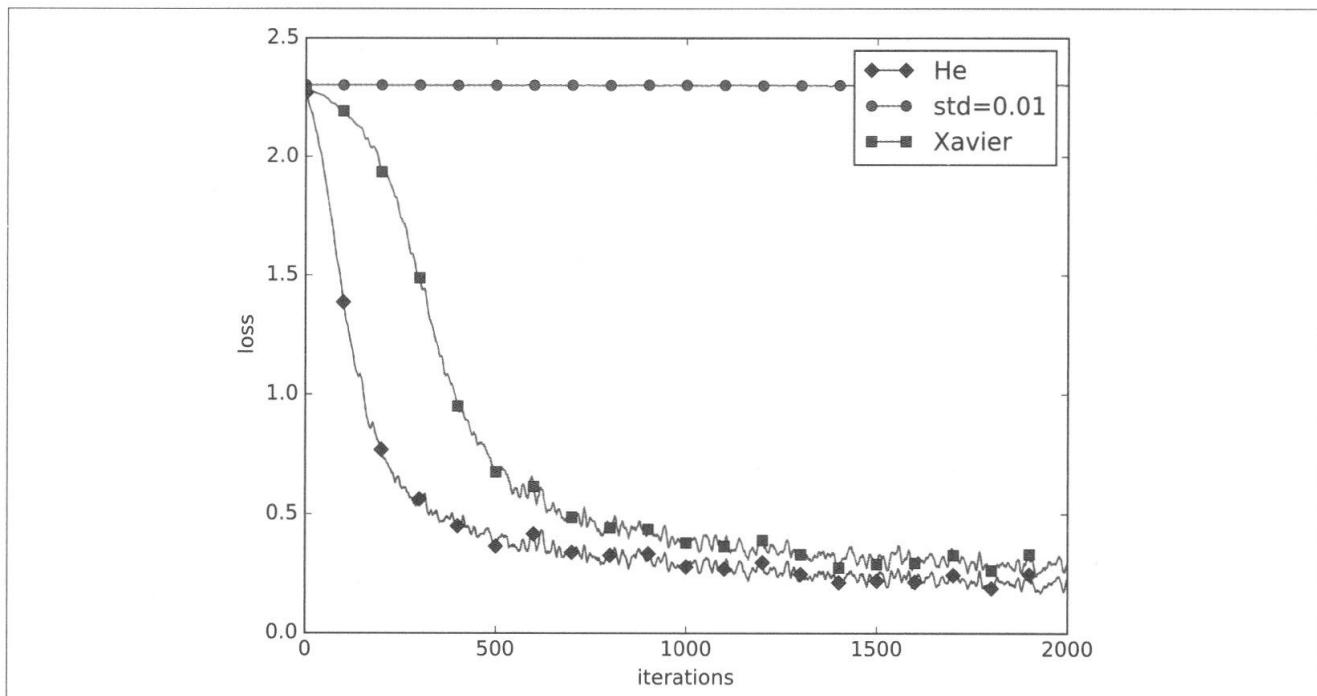


圖 6-15 針對 MNIST 資料集，比較「權重的預設值」：水平軸是學習的重複次數（iterations），垂直軸是損失函數值（loss）

如上所示，在神經網路的學習中，權重的預設值成為非常重要的關鍵。通常權重的預設值會影響神經網路的學習成功與否。權重的預設值是很容易被忽略的關鍵，但是任何事物都是開始（預設值）最重要。因此，在本節的最後，我想再次重申權重預設值的重要性。

## 6.3 Batch Normalization

在上一節「6.2 權重的預設值」中，觀察了各層的活性化分布。這裡學到的重點是，只要妥善設定權重的預設值，就能讓各層的活性化分布具有適當廣度，順利進行學習。為了讓各層擁有適當的廣度，何不「強制性」調整活性化的分布？其實，以這種概念為基礎的手法，稱作 Batch Normalization [11]。

### 6.3.1 Batch Normalization 的演算法

Batch Normalization（以下簡稱 Batch Norm）是 2015 年提出的手法。儘管 Batch Norm 才剛問世不久，是很新的手法，卻已經有許多研究者、技術人員在使用。事實上，檢視機器學習的競賽結果，利用 Batch Norm 獲得優秀成果的案例比比皆是。

為何 Batch Norm 如此受到矚目？因為 Batch Norm 有以下優點。

- 可以快速學習（因為能增加學習率）
- 不會過度依賴預設值（不會對預設值產生過度反應）
- 控制過度學習（減少 Dropout 等必要性）

由於深度學習需要花許多時間，所以第一個優點就很吸引人。另外，不用過度在意預設值，還可以控制過度學習，這些都可以解決在深度學習的學習階段，令人頭痛的問題。

Batch Norm 的概念如前面所述，可以調整各層的活性化分布，變得具有適當廣泛性。因此，這裡把進行資料分布正規化的層級，亦即 Batch Normalization 層（以下簡稱「Batch Norm」層），插入神經網路中，如圖 6-16 所示。

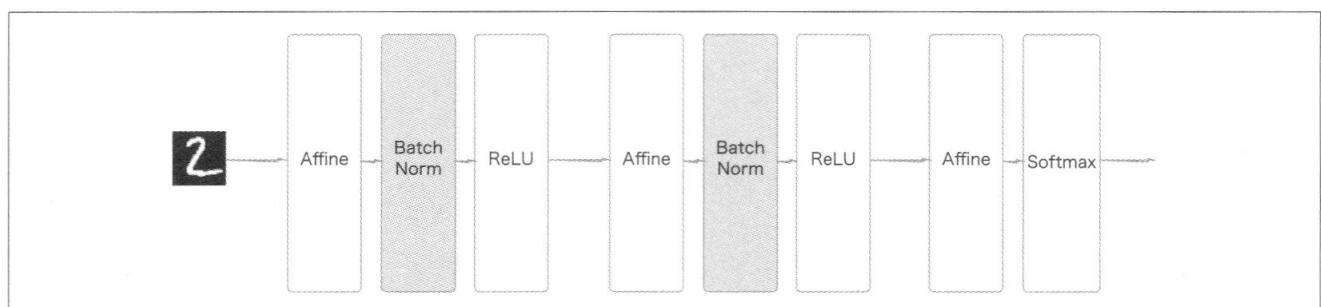


圖 6-16 使用 Batch Normalization 的神經網路範例（Batch Norm 層顯示為灰底）

Batch Norm 顧名思義，是在進行學習時，以小批次為單位，依照各個小批次來進行正規化。具體來說，就是進行讓資料分布平均為 0，分散為 1 的正規化處理。以算式顯示的結果如下所示。

$$\begin{aligned}
 \mu_B &\leftarrow \frac{1}{m} \sum_{i=1}^m x_i \\
 \sigma_B^2 &\leftarrow \frac{1}{m} \sum_{i=1}^m (x_i - \mu_B)^2 \\
 \hat{x}_i &\leftarrow \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}
 \end{aligned} \tag{6.7}$$

這裡針對  $m$  個小批次  $B = \{x_1, x_2, \dots, x_m\}$  輸入資料，計算平均  $\mu_B$ 、分散  $\sigma_B^2$ 。再進行讓輸入資料的平均為 0，分散為 1（適當分布）的正規化處理。此外，算式 (6.7) 的  $\epsilon$  是微小數值（例如， $10^{-7}$  等）。這是用來防止發生除以 0 的情況。

算式 (6.7) 執行的處理是將小批次的輸入資料  $\{x_1, x_2, \dots, x_m\}$  轉換成平均 0、分散 1 的資料  $\{\hat{x}_1, \hat{x}_2, \dots, \hat{x}_m\}$ ，非常簡單。在活化函數之前（或之後）插入這種處理步驟<sup>5</sup>，可以減少資料分布的偏差。

此外，Batch Norm 層是針對正規化後的資料，以原有的規模與移動來進行轉換。以算式顯示，結果如下所示。

$$y_i \leftarrow \gamma \hat{x}_i + \beta \tag{6.8}$$

$\gamma$  與  $\beta$  是參數。最初從  $\gamma=1$ 、 $\beta=0$  開始，藉由學習調整成適當值。

以上就是 Batch Norm 演算法。這種演算法將成為神經網路上的正向傳播。使用第 5 章說明過的計算圖，可以顯示 Batch Norm，如圖 6-17 所示。

導出 Batch Norm 的反向傳播過程有些複雜，所以這裡省略不提。但是，利用圖 6-17 計算圖來思考，應該就能輕易導出 Batch Norm 的反向傳播。Frederik Kratzert 的部落格「Understanding the backward pass through Batch Normalization Layer」[13] 有詳細說明，有興趣的讀者，可以自行參考。

<sup>5</sup> 應該在活化函數的前或後插入 Batch Normalization 的探討（及實驗），請參考文獻 [11] 或 [12]。

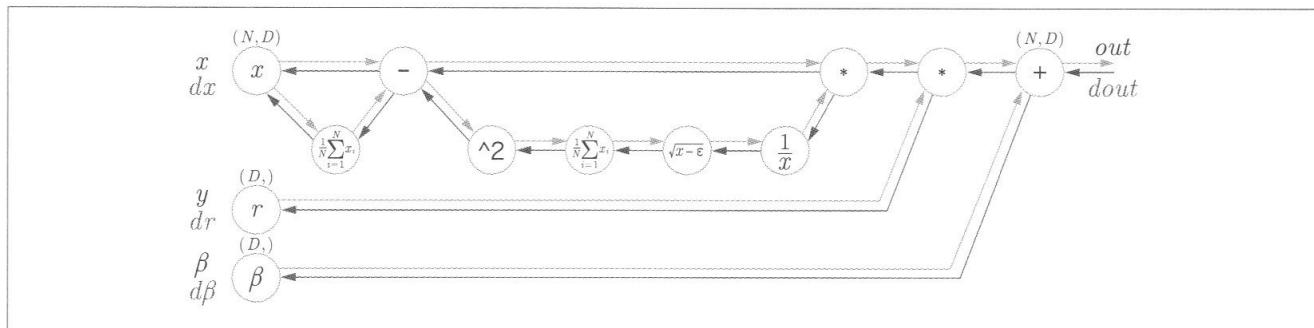


圖 6-17 Batch Normalization 的計算圖（引用自文獻 [13]）

### 6.3.2 Batch Normalization 的評價

接下來，要利用 Batch Norm 層，開始進行實驗。首先，使用 MNIST 資料集，檢視使用了 Batch Norm 層與沒有使用時，學習的速度產生何種變化（原始碼位於 ch06/batch\_norm\_test.py）。結果如圖 6-18 所示。

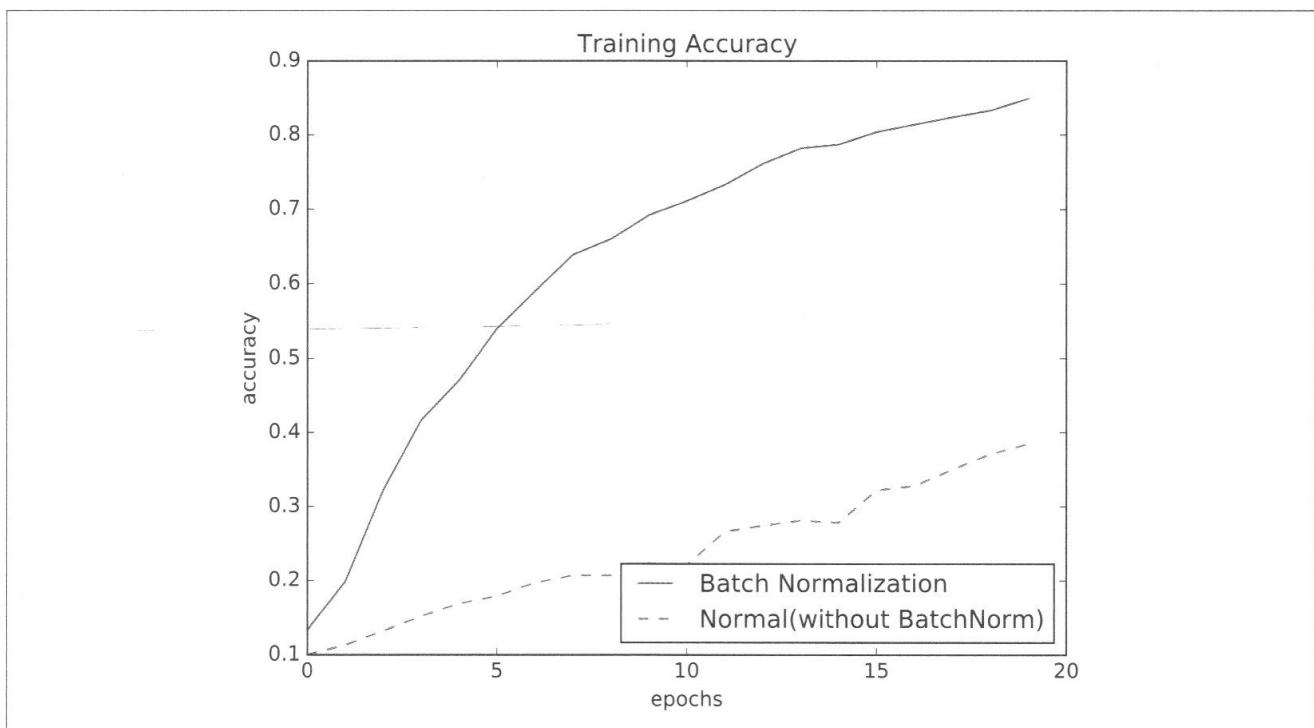


圖 6-18 Batch Norm 的效果：利用 Batch Norm，提升學習速度

從圖 6-18 的結果可以得知，Batch Norm 提升了學習速度。接下來，給予各種預設值，觀察學習速度的變化。圖 6-19 是將權重預設值的標準差改變成不同數值後的學習過程。

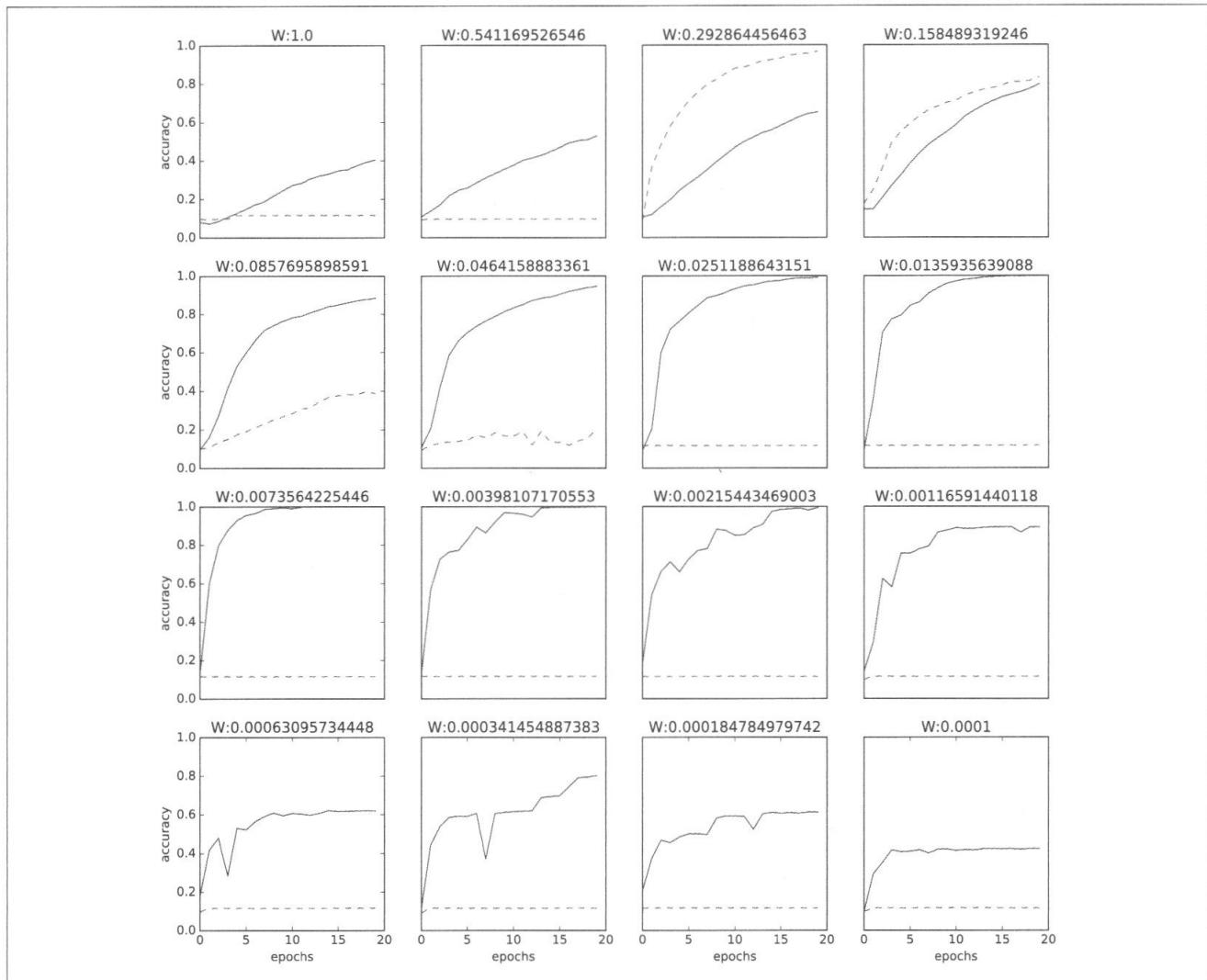


圖 6-19 圖中的實線是使用 Batch Norm 後的結果，虛線是沒有使用 Batch Norm 的結果：圖的標題顯示為權重預設值的標準差

我們可以看到，幾乎所有範例在使用了 Batch Norm 之後，學習速度都比較快。事實上，沒有使用 Batch Norm 時，就得給予適當的預設值，否則完全不會學習。

如以上說明，使用 Batch Norm，可以促進學習速度，而且權重的預設值會變穩健（「預設值變得穩健」是指，不會過度依賴預設值）。由於 Batch Norm 具備這些優秀特性，所以能運用在各種場合。

## 6.4 正規化

在機器學習的問題中，經常出現過度學習（*overfitting*）的情況。過度學習是指，過度適應訓練資料，而無法順利對應不含訓練資料的其他資料。機器學習的目標是擁有一般化的能力，即使遇到沒有包含在訓練資料或未曾見過的資料，也可以正確辨識。這種複雜且表現力高的模型，的確可以設計出來，所以控制過度學習的技巧，就顯得很重要。

### 6.4.1 過度學習

以下列出主要兩個造成過度學習的原因。

- 擁有大量參數，表現力高的模型
- 訓練資料太少

以下將刻意滿足這兩個條件，引起過度學習。這裡把 MNIST 資料集的訓練資料從原本的 60,000 個限縮成 300 個，並且提高網路的複雜性，變成 7 層網路，每層的神經元為 100 個，活化函數使用 ReLU。

以下擷取實驗用的程式（檔案是 ch06/overfit\_weight\_decay.py）。首先是載入資料的程式。

```
(x_train, t_train), (x_test, t_test) = load_mnist(normalize=True)
# 為了製造過度學習而減少學習資料
x_train = x_train[:300]
t_train = t_train[:300]
```

接下來是進行訓練的程式。和前面的程式一樣，按照每個循環週期（epoch），分別計算出所有訓練資料與測試資料的辨識準確度。

```
network = MultiLayerNet(input_size=784, hidden_size_list=[100, 100, 100,
100, 100], output_size=10)
optimizer = SGD(lr=0.01) # 以學習率為 0.01 的 SGD 更新參數

max_epochs = 201
train_size = x_train.shape[0]
batch_size = 100

train_loss_list = []
train_acc_list = []
test_acc_list = []

iter_per_epoch = max(train_size / batch_size, 1)
```

```

epoch_cnt = 0

for i in range(1000000000):
    batch_mask = np.random.choice(train_size, batch_size)
    x_batch = x_train[batch_mask]
    t_batch = t_train[batch_mask]

    grads = network.gradient(x_batch, t_batch)
    optimizer.update(network.params, grads)

    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)

    epoch_cnt += 1
    if epoch_cnt >= max_epochs:
        break

```

在 `train_acc_list`、`test_acc_list` 中，儲存了以循環週期（epoch）為單位（看完所有訓練資料的單位）的辨識準確度。接下來，把這些清單（`train_acc_list`、`test_acc_list`）畫成圖表，結果如圖 6-20 所示。

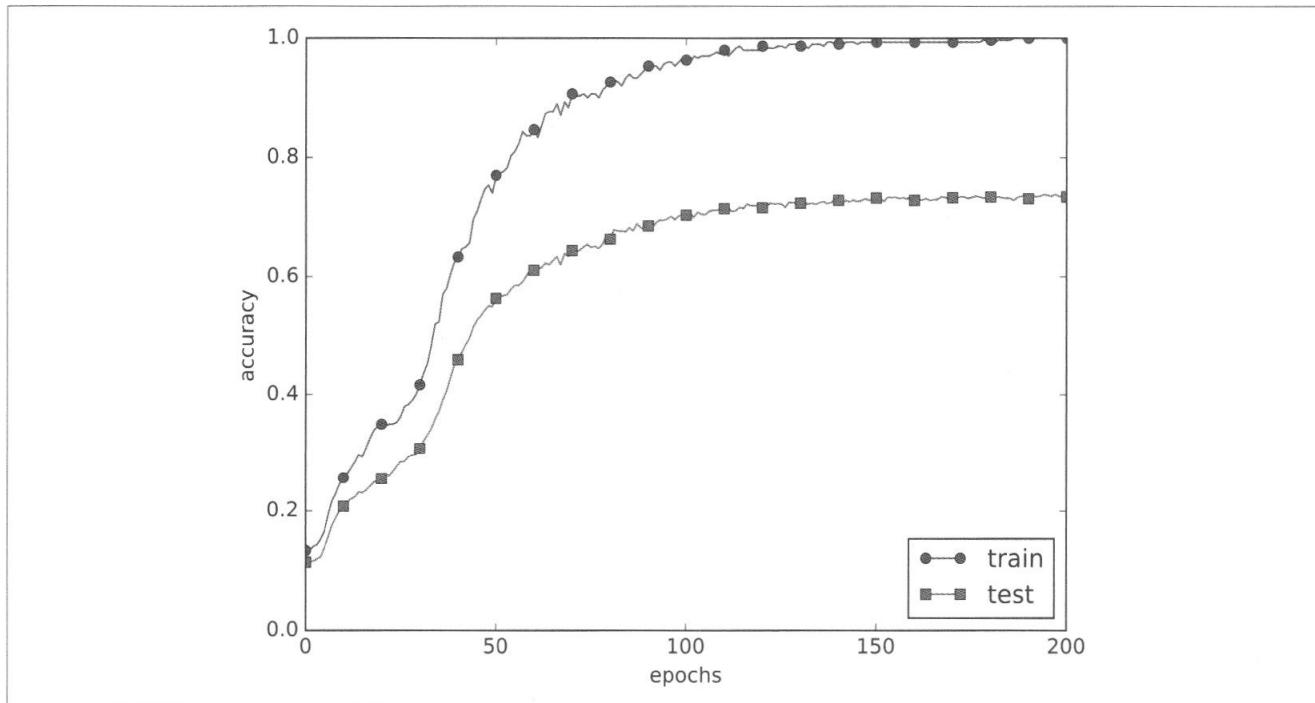


圖 6-20 訓練資料（train）與測試資料（test）的辨識準確度變化

使用訓練資料，計算出來的辨識準確度，超過 100 epoch 之後，幾乎是 100%。可是，測試資料與 100% 辨識準確度有著很大的差距。這種辨識準確度的明顯差異，其實是過度適應訓練資料的結果。從這張圖可以看出，結果無法妥善對應不含訓練資料的一般化資料（測試資料）。

## 6.4.2 Weight decay

過去常用來控制過度學習的手法是 *Weight decay*（權重衰減）。在學習的過程中，針對擁有較大權重的部分，課以罰金，藉此控制過度學習。原本過度學習就是權重參數值太大時，常發生的問題。

再複習一下，神經網路的學習是以減少損失函數的值為目的。例如，權重的平方範數（L2 norm）與損失函數相加，就可以抑制權重變大。用符號來顯示，假設權重為  $\mathbf{W}$ ，L2 norm 的 Weight decay 是  $\frac{1}{2}\lambda\mathbf{W}^2$ ，將  $\frac{1}{2}\lambda\mathbf{W}^2$  與損失函數相加。這裡的  $\lambda$  是控制正規化強度的超參數。 $\lambda$  值愈大，會對取得的大權重值，課以較重的罰金。另外， $\frac{1}{2}\lambda\mathbf{W}^2$  開頭的  $\frac{1}{2}$  是為了把  $\frac{1}{2}\lambda\mathbf{W}^2$  的微分結果變成  $\lambda\mathbf{W}$  的調整用定數。

Weight decay 是針對全部的權重，將  $\frac{1}{2}\lambda\mathbf{W}^2$  加入損失函數中。因此，計算權重的梯度時，將誤差反向傳播法的結果與正規化項目的微分  $\lambda\mathbf{W}$  相加。



L2 norm 是對應各元素的平方和。以算式顯示，假設權重  $\mathbf{W} = (w_1, w_2, \dots, w_n)$ ，可以用  $\sqrt{w_1^2 + w_2^2 + \dots + w_n^2}$  計算出 L2 norm。此外，除了 L2 norm 之外，還有 L1 norm 及  $L^\infty$  norm。L1 norm 是絕對值的和，相當於  $|w_1| + |w_2| + \dots + |w_n|$ 。 $L^\infty$  norm 又稱 Max norm，相當於各元素中，最大的絕對值。L2 norm、L1 norm、 $L^\infty$  norm 都可以當作正規化項目，各有特色，但是這裡只執行了一般常用的 L2 norm。

接下來，開始進行實驗。針對剛才的實驗，以  $\lambda=0.1$  套用 Weight decay。結果如圖 6-21 所示（對應 Weight decay 的網路位於 common/multi\_layer\_net.py，實驗用的原始碼位於 ch06/overfit\_weight\_decay.py）。

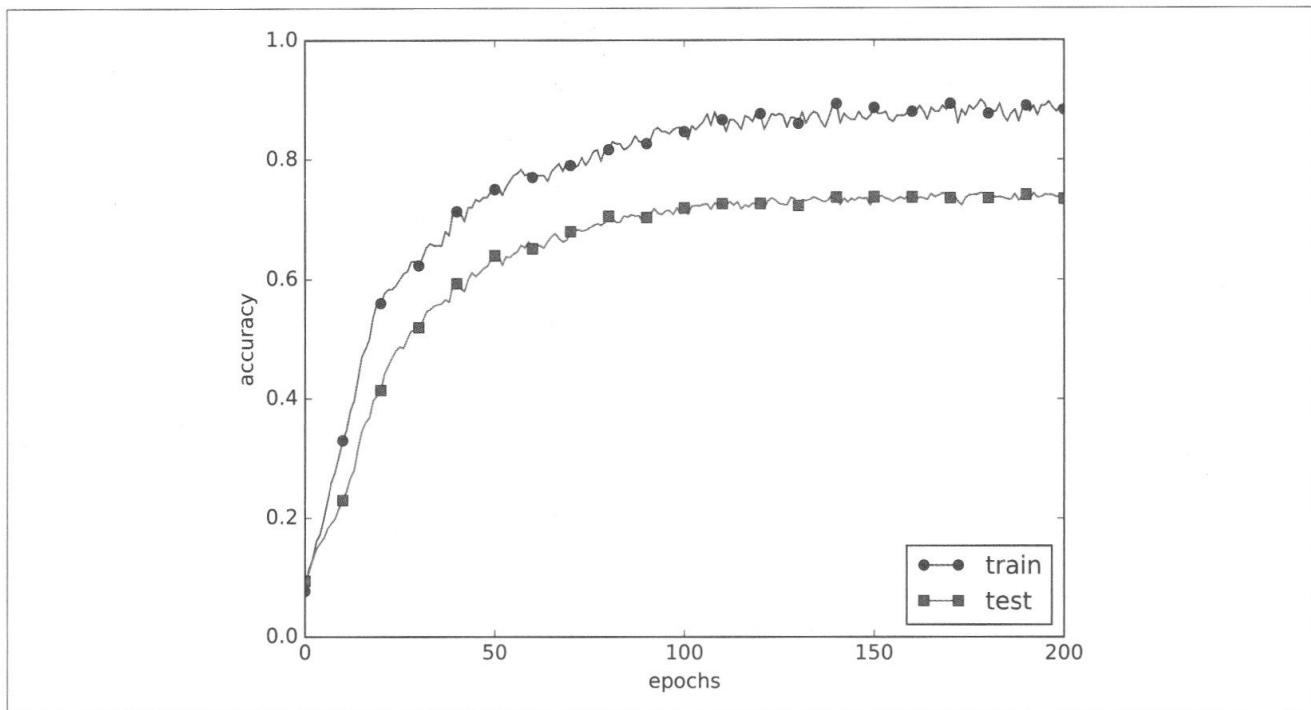


圖 6-21 使用 Weight decay 的訓練資料（train）與測試資料（test）的辨識準確度變化

如圖 6-21 所示，訓練資料與測試資料的辨識精準度之間，存在著「差距」。但是與沒有使用 Weight decay 的圖 6-20 相比，差距縮小了，代表過度學習受到控制。此外，訓練資料的辨識準確度沒有達到 100% (1.0)，也是必須注意到的重點。

### 6.4.3 Dropout

剛才說明了在損失函數中，加上權重 L2 norm 的 Weight decay，當作控制過度學習的方法。Weight decay 可以輕鬆執行，也能控制一定程度的過度學習。可是，當神經網路的模型變複雜時，單憑 Weight decay，很難解決過度學習問題。因此，還有另一個常用的手法，就是 Dropout [14]。

Dropout 是一邊隨機消除神經元，一邊學習的手法。訓練時，隨機選出隱藏層的神經元，再將挑選出來的神經元刪除。刪除後的神經元，如圖 6-22 所示，無法進行訊號傳遞。此外，訓練時，每次傳遞資料的時候，會隨機挑選刪除的神經元。測試時，會傳遞全部的神經元訊號，但是各神經元的輸出，要乘上訓練時刪除的神經元比例再傳遞。

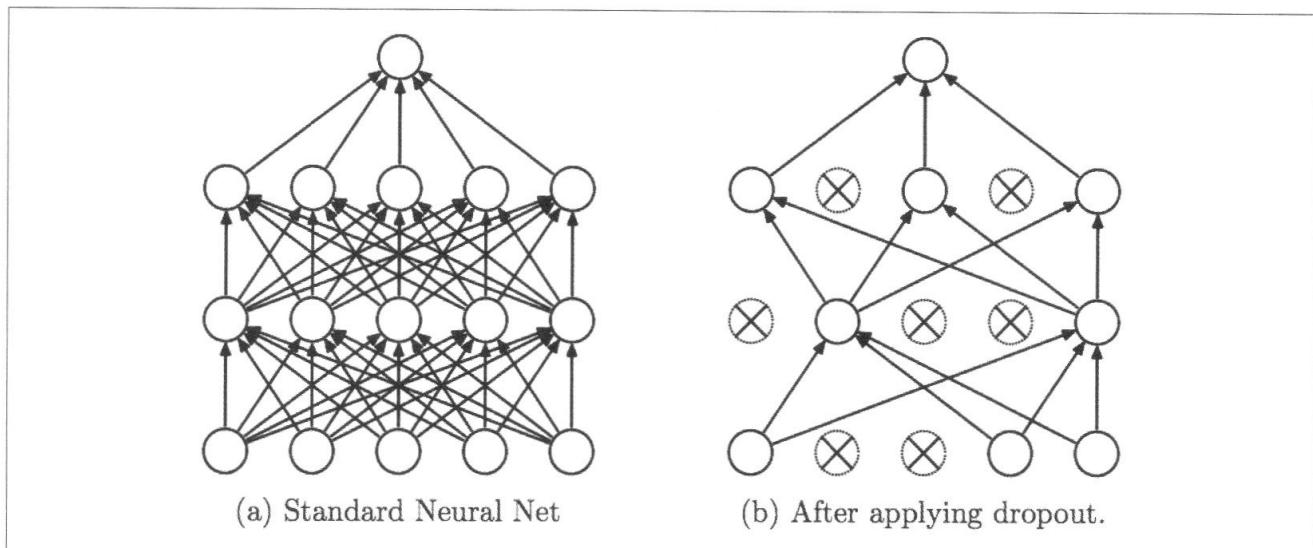


圖 6-22 Dropout 概念圖（引用自文獻 [14]）：左邊是一般的神經網路，右邊是套用了 Dropout 的神經網路。Dropout 是隨機選取神經元再刪除，藉此停止之後的訊號傳遞

接著要執行 Dropout。這裡為了方便瞭解，列出了執行過程的細節。可是，在訓練的過程中，進行適當計算時，正向傳播只要純粹傳遞資料即可（不乘上刪除比例也可以），因此在神經網路的框架中，會採取這種處理方式。若要快速執行處理，可以參考以 Chainer 執行的 Dropout。

```
class Dropout:  
    def __init__(self, dropout_ratio=0.5):  
        self.dropout_ratio = dropout_ratio  
        self.mask = None  
  
    def forward(self, x, train_flg=True):  
        if train_flg:  
            self.mask = np.random.rand(*x.shape) > self.dropout_ratio  
            return x * self.mask  
        else:  
            return x * (1.0 - self.dropout_ratio)  
  
    def backward(self, dout):  
        return dout * self.mask
```

這裡的重點是，每次正向傳播時，把刪除的神經元當作 `False` 儲存在 `self.mask` 中。`self.mask` 是隨機產生和 `x` 相同形狀的陣列，只有這個值比 `dropout_ratio` 大的元素當作 `True`。反向傳播時的動作與 ReLU 相同。換句話說，在正向傳播時，通過訊號的神經元，於反向傳播時，也會直接傳遞訊號；在正向傳播時，沒有通過訊號的神經元，在反向傳播時，訊號就會在此停住。

以下將利用 MNIST 資料集驗證 Dropout 的效果。原始碼位於 ch06/overfit\_dropout.py。此外，在原始碼中，使用 Trainer 類別，簡化處理過程。



在 common/trainer.py 中，執行 Trainer 類別。使用這個類別，可以用 Trainer 類別取代和前面一樣的神經網路學習。詳細內容請參考 common/trainer.py 與 ch06/overfit\_dropout.py。

接下來要進行 Dropout 的實驗，和前面一樣，使用 7 層神經網路（每層的神經元有 100 個，活化函數為 ReLU），一邊套用 Dropout，另一邊不套用 Dropout。結果如圖 6-23 所示。

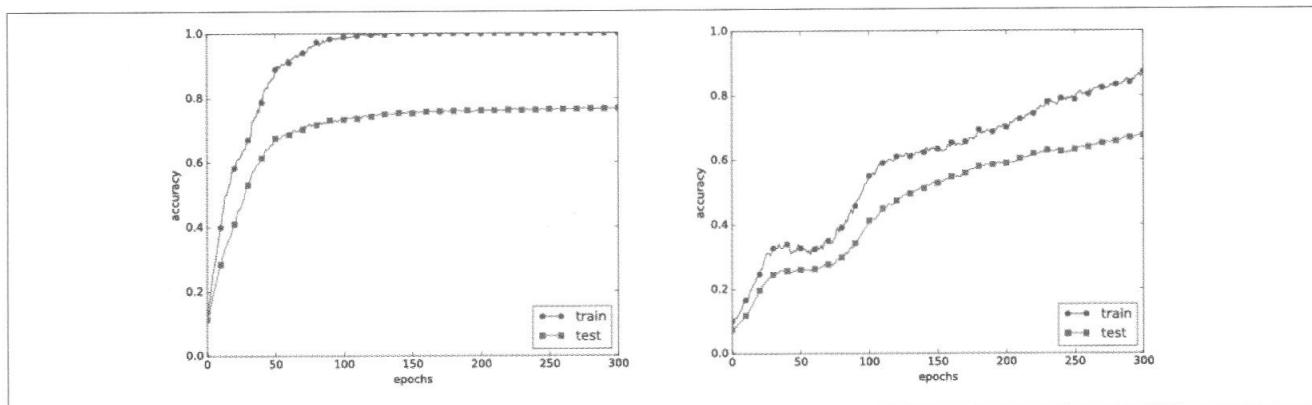


圖 6-23 左圖是沒有 Dropout，右圖有 Dropout (dropout\_rate=0.15)

由圖 6-23 可得知，使用 Dropout，訓練資料與測試資料的辨識準確度差距縮小了。此外，訓練資料的辨識準確度沒有達到 100%。因此，使用 Dropout 之後，即使是表現複雜的神經網路，也可以控制過度學習問題。



在機器學習中，常用到整體學習（ensemble learning）。整體學習是指，個別學習多個模型，推論時，將這些輸出平均。按照神經網路的描述，假設有 5 個相同結構（或類似結構）的神經網路，分別進行學習，在測試時，提出這 5 個輸出的平均值。從實驗中可以得知，進行整體學習，能讓神經網路的辨識準確度提高數 %。

整體學習與 Dropout 有著類似關係。我們可以把在學習時，Dropout 隨機刪除神經元的情況，解釋成每次學習不同模型。推論時，針對神經元的輸出，乘上刪除比例（例如，0.5 等），取出模型的平均值。換句話說，我們可以把 Dropout 視為（類似）用一個神經網路，達到和整體學習一樣效果。

## 6.5 驗證超參數

在神經網路中，除了權重與偏權值等參數，也常提到超參數 (*hyper-parameter*)。這裡所謂的超參數是指，各層的神經元數量、批次大小、參數更新時的學習率、Weight decay 等。如果沒有妥善設定超參數，就會變成效能不佳的模型。超參數的值非常重要，但是決定超參數時，通常都伴隨著許多錯誤嘗試。以下將說明可以有效找出超參數的方法。

### 6.5.1 驗證資料

到目前為止，使用的資料集，是分成訓練資料與測試資料等兩種資料來運用。利用訓練資料進行學習，使用測試資料評估一般化能力，藉此瞭解是否過度適應訓練資料（有沒有發生過度學習），還有一般化能力的程度。

接下來，要設定各種超參數值進行驗證。但是這裡必須特別注意到，不可以使用測試資料來評估超參數的效能。這點非常重要，卻是容易被忽略的地方。

為什麼不能用測試資料評估超參數的效能？因為，若使用測試資料調整超參數，超參數的值會對測試資料產生過度學習。這種做法等於使用測試資料來確認超參數值的「好壞」，結果調整成只適合測試資料的超參數，變成無法適應其他資料、一般化能力低的模型。

因此，調整超參數時，必須準備超參數專用的確認資料。超參數的調整用資料一般稱作驗證資料 (*validation data*)。利用這種驗證資料來評估超參數的好壞。



訓練資料是用來進行參數（權重及偏權值）學習，而驗證資料是用來評估超參數的效能。最後（理想狀態是一次）使用測試資料檢查一般化能力。

有些資料集已經先將資料分成訓練資料、驗證資料、測試資料。也有部分資料集只提供訓練資料和測試資料，甚至有些完全沒有分別。此時，就需要使用者手動劃分資料。以 MNIST 資料集為例，最簡單取得驗證資料的方法，就是從訓練資料中，先分出 20% 的資料，當作驗證資料。以程式顯示，結果如下所示。

```
(x_train, t_train), (x_test, t_test) = load_mnist()

# 混合訓練資料
x_train, t_train = shuffle_dataset(x_train, t_train)
```

```
# 分割驗證資料
validation_rate = 0.20
validation_num = int(x_train.shape[0] * validation_rate)

x_val = x_train[:validation_num]
t_val = t_train[:validation_num]
x_train = x_train[validation_num:]
t_train = t_train[validation_num:]
```

在分割訓練資料之前，先混合輸入資料與訓練標籤。這是因為部分資料集可能有特別的資料傾向（例如，數字「0」到「10」依序排列等）。另外，這裡使用的 `shuffle_dataset` 函數，利用了 `np.random.shuffle`，在 `common/util.py` 中，可以看到執行過程。

接著，就用驗證資料來檢視超參數的最佳化手法。

### 6.5.2 超參數的最佳化

執行超參數最佳化的關鍵在於，逐漸縮小「優良」超參數存在的範圍。逐漸縮小範圍是指，剛開始先設定概略的範圍，從該範圍中，隨機選出超參數（取樣），利用樣本值評估辨識準確度。反覆重複數次，觀察辨識準確度的結果，從結果中，縮小「優良」超參數的範圍。重複執行以上步驟，即可逐漸篩選出適當的超參數範圍。



根據報告顯示，進行神經網路的超參數最佳化時，和格點搜尋（grid search）等規律搜尋方式相比，隨機取樣的搜尋方式，效果比較好 [15]。這是因為在眾多超參數之中，每種超參數對最後辨識準確度造成影響程度並不一樣。

「概略」設定超參數的範圍，效果比較好。這裡的「概略」是指，以「10 的次方」來設定範圍，例如  $0.001 (10^{-3})$  到  $1,000 (10^3)$  左右（也可以寫成「以對數刻度（log scale）來設定」）。

進行超參數最佳化時，必須注意到，深度學習的學習階段需要花費許多時間（例如，幾天或幾週等）。因此，尋找超參數時，必須盡早排除可能不適合的超參數。在超參數最佳化的過程中，縮小學習用的循環週期（epoch），能有效縮短每次評估所需的時間。

以上就是超參數的最佳化，經過整理歸納後，可以列出以下步驟。

#### 步驟 0

設定超參數的範圍。

### 步驟 1

從設定的超參數範圍中，隨機取樣。

### 步驟 2

使用步驟 1 取樣的超參數值進行學習，利用驗證資料評估辨識準確度（但是要設定成較小的 epoch 值）。

### 步驟 3

重複執行步驟 1 與步驟 2 多次（100 次等），從辨識準確度的結果中，縮小超參數的範圍。

重複上述操作，縮小超參數的範圍，縮小至一定程度後，從該範圍中，選出一個超參數的值。這就是一種超參數最佳化的手法。



這裡說明的超參數最佳化手法，是一種實踐性的方法。但是，與其說這是一種科學化手法，其實倒比較像是實踐者的「智慧結晶」。進行超參數最佳化時，如果希望採取更洗練的手法，可以使用貝葉斯優化 (*Bayesian optimization*)。貝葉斯優化是以貝葉斯定律為主的數學（理論），進行更嚴謹、有效率的最佳化。詳細內容請參考論文「Practical Bayesian Optimization of Machine Learning Algorithms」[16] 等。

### 6.5.3 執行超參數最佳化

接下來，我想使用 MNIST 資料集，進行超參數最佳化。這裡要以學習率 (learning rate) 以及控制 Weight decay 強度的係數（後續稱作「Weight decay 係數」），當作尋找超參數最佳化的對象。此外，這裡的問題設定與解決問題的手法，參考了史丹佛大學的課程「CS231n」[5]。

前面說明過，超參數的驗證是從  $0.001 (10^{-3})$  到  $1,000 (10^3)$  的對數刻度範圍中，隨機取樣，進行驗證。在 Python 可以寫成 `10 ** np.random.uniform(-3, 3)`。以下 Weight decay 係數以  $10^{-8}$  到  $10^{-4}$  為範圍，學習率以  $10^{-6}$  到  $10^{-2}$  為範圍，開始進行實驗。此時，隨機取樣超參數的程式如下所示。

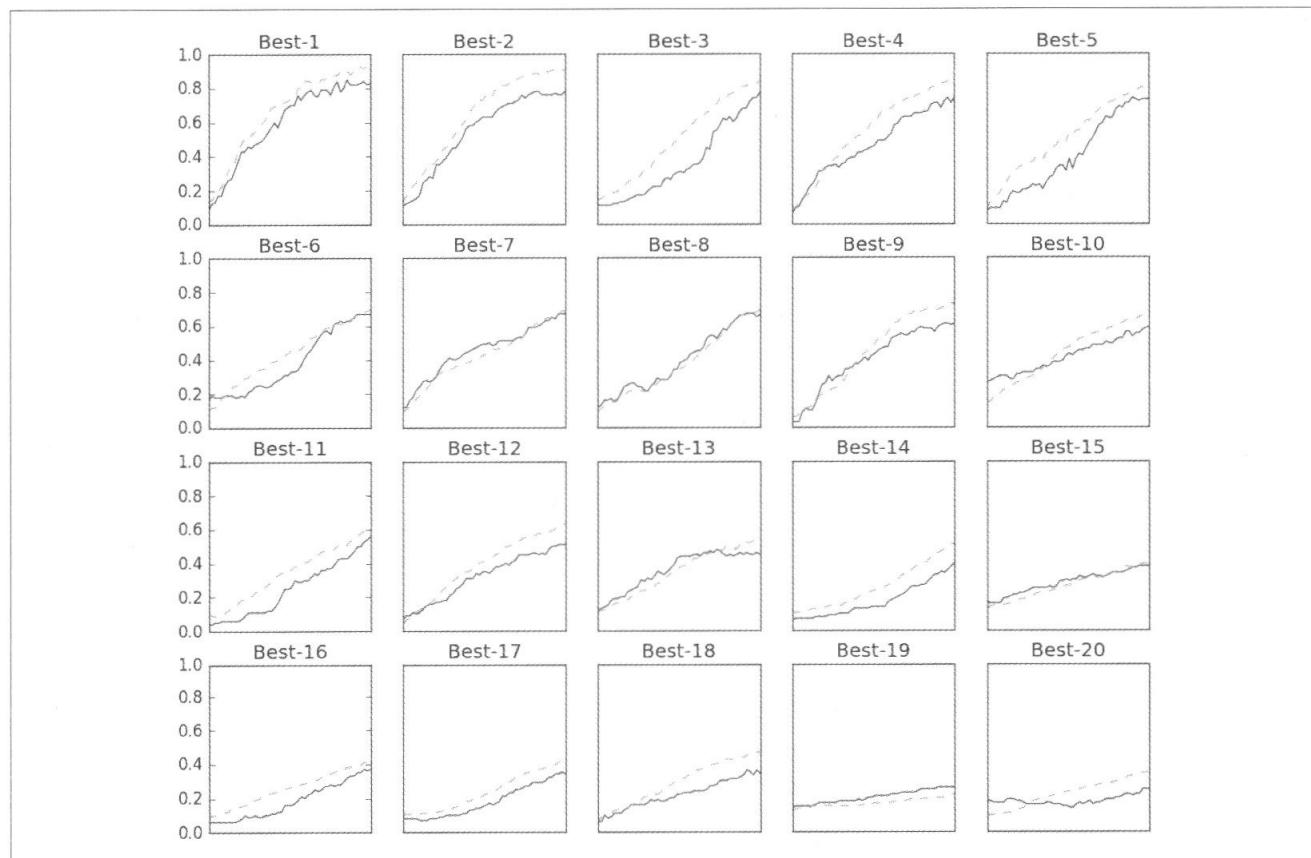
```
weight_decay = 10 ** np.random.uniform(-8, -4)
lr = 10 ** np.random.uniform(-6, -2)
```

隨機取樣，再使用該值進行學習。後續以各種超參數值反覆進行學習，觀察優良的超參數位於何處。這裡省略詳細的執行過程，只顯示結果。超參數最佳化的原始碼位於 ch06/hyperparameter\_optimization.py，請自行參考。

另外，Weight decay 係數以  $10^{-8}$  到  $10^{-4}$  為範圍，學習率以  $10^{-6}$  到  $10^{-2}$  為範圍，進行實驗之後，結果如圖 6-24 所示。

**圖 6-24** 依照辨識準確度的高低順序，顯示驗證資料的學習變化。檢視這張圖，可以得知，到「Best-5」為止，都能順利學習。因此，觀察到「Best-5」為止的參數值（學習率與 Weight decay 係數），結果如下所示。

```
Best-1 (val acc:0.83) | lr:0.0092, weight decay:3.86e-07
Best-2 (val acc:0.78) | lr:0.00956, weight decay:6.04e-07
Best-3 (val acc:0.77) | lr:0.00571, weight decay:1.27e-06
Best-4 (val acc:0.74) | lr:0.00626, weight decay:1.43e-05
Best-5 (val acc:0.73) | lr:0.0052, weight decay:8.97e-06
```



**圖 6-24** 實線是驗證資料的辨識準確度，虛線是訓練資料的辨識準確度

從結果中可以得知，順利學習的範圍是，學習率為 0.001 到 0.01，Weight decay 係數是  $10^{-8}$  到  $10^{-6}$ 。像這樣，觀察順利學習的超參數範圍，逐漸縮小超參數值的範圍，在縮小後的範圍內，重複相同的步驟，適當的超參數範圍就會變小，到了一定程度之後，挑選出一個最後的超參數值。

## 6.6 重點整理

本章介紹了幾個在進行神經網路的學習時，非常重要的技巧。包括參數的更新方法、提供權重預設值的方法、還有 Batch Normalization 及 Dropout 等，每一種對現在的神經網路而言，都是不可缺少的技術。此外，這裡學到的技術，也經常用在最先進的深度學習中。

### 本章學到的重點

- 在參數的更新方法，除了 SGD 之外，還有 Momentum、AdaGrad、Adam 等知名的手法。
- 提供權重預設值的方法，在進行正確的學習時，非常重要。
- 使用「Xavier 預設值」及「He 預設值」，當作權重的預設值，效果比較好。
- 使用 Batch Normalization 可以提升學習速度，而且不會過度依賴預設值，比較穩定。
- Weight decay 與 Dropout 是控制過度學習用的正規化技術。
- 尋找超參數的有效方法是，逐漸縮小優良值存在的範圍。