

第三章 Keras和Tensorflow簡介

目錄

3-1 Tensorflow是什麼?

3-2 Keras是什麼?

3-4 設定深度學習工作站

3-5使用 Tensorflow的第一步

3-6 剖析神經網路：了解 Keras API的核心

3-1 Tensorflow是什麼？

TensorFlow 是一個免費、開源的 Python機器學習框架，主要由 Google 所開發。

TensorFlow 的首要目標是讓工程師和研究人員可以在數值張量上進行數學運算。不過比起 NumPy，TensorFlow 在以下方面更具優勢：

- TensorFlow 能在任何可微分函數上自動計算梯度，因此非常適合用在機器學習。
- TensorFlow 不僅可在CPU上運行，也可在GPU 及 TPU 等高度平行的硬體加速器上運行。
- 用TensorFlow 建立的運算程序，可以輕易地分散到多台機器上共同執行。
- TensorFlow 的程式可以匯出為各種不同語言的程式，包括C++、JavaScript或 TensorFlow Lite等，這樣的適應能力讓 TensorFlow 應用可以輕鬆部署到各種實際場景

3-2 Keras是什麼?

Keras是一款用Python編寫而成的開源神經網路庫，也可以說是開放的高階深度學習程式庫，能搭配TensorFlow、Theano等運作，其設計目的是希望快速實現深度神經網路。

Keras將訓練模型的輸入層、隱藏層、輸出層建好架構，只需插入所需的參數或函式即可，因此可使用最少的程式碼，花費最少的時間，就完成深度學習模型的建構，開始進行訓練，修正誤差，並拿去做應用和預測。

相較之下，TensorFlow屬於低階的程式庫，雖然能夠達到更細緻更精準的模型，但是需要付出大量的時間成本，設計更多複雜的程式碼。

Keras僅處理深度學習模型的建立、訓練、預測等，然而底層的運算，如張量（矩陣）運算，則是交給「後端引擎」做配合，目前可支援的後端引擎主要是TensorFlow。

3-4 設定深度學習工作站

在動手開發深度學習應用之前，要設定好開發環境，推薦使用者在 NVIDIA GPU 上運程式碼，而不是在自己電腦上的CPU。要在 GPU 上進行深度學習，你有3種選擇：

- 直接買一張 NVIDIA GPU 顯示卡，並裝在自己的電腦上。
- 在 Google Cloud 或 AWS EC2 上使用GPU。
- 使用 Google Colaboratory（一個 notebook 服務）上的免費 GPU。

其中，Colaboratory 是最容易上手的，既不需要買硬體，也不需要安裝什麼軟體。只要在瀏覽器上開啟相關網頁，就可以開始寫程式了。不過，Colaboratory 的免費版本只適合處理工作量不大的任務。如果想要擴展任務的規模，就得考慮第一個或第二個選項。

3-4-1 Jupyter Notebook：執行深度學習專案的首選

在 Jupyter Notebook 中實作深度學習專案是很好的選擇。Jupyter 記事本被廣泛使用於資料科學和機器學習領域。Jupyter 記事本應用所產生的文件稱為 notebook，可以在瀏覽器中開啟並編輯它的內容。

Jupyter記事本的特色是提供各式各樣的純文字編輯功能，不但可以編寫 Python 程式碼，還可以註解正在做的事情，並整合了直接執行 Python 程式的能力。

notebook 檔案還允許將一長串的實作範例分解為一個個小區塊程式碼獨立執行，使開發過程具有一來一往的互動性。如果在實作後期才發生錯誤，也不必重新執行之前的所有程式碼。

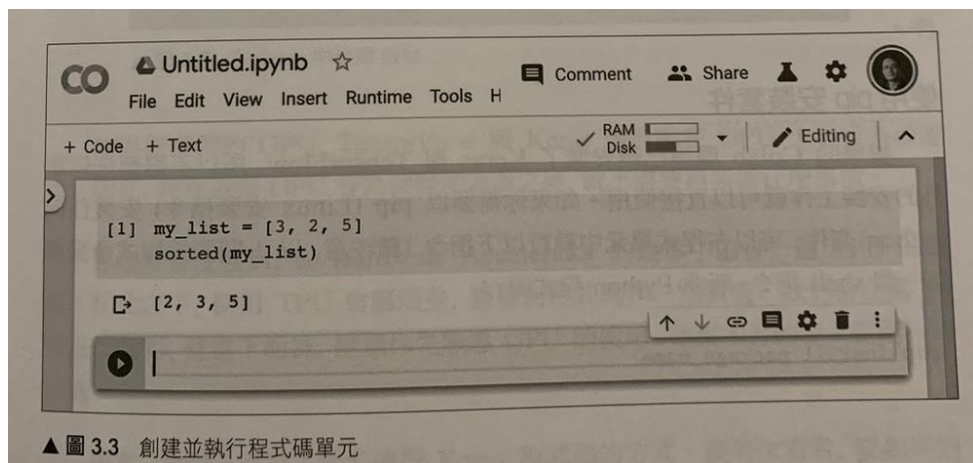
3-4-2 使用 Colaboratory

Colaboratory 是一種無需額外進行安裝，而且可以完全在雲端操作的免費 Jupyter 記事本服務。具體來說，Colaboratory 是一個網頁，可以直接在該網頁上撰寫並執行 Keras 程式碼。在 Colaboratory 上，你可以透過免費（但限量）的 GPU 甚至 TPU 來做運算，所以不必自己買 GPU。

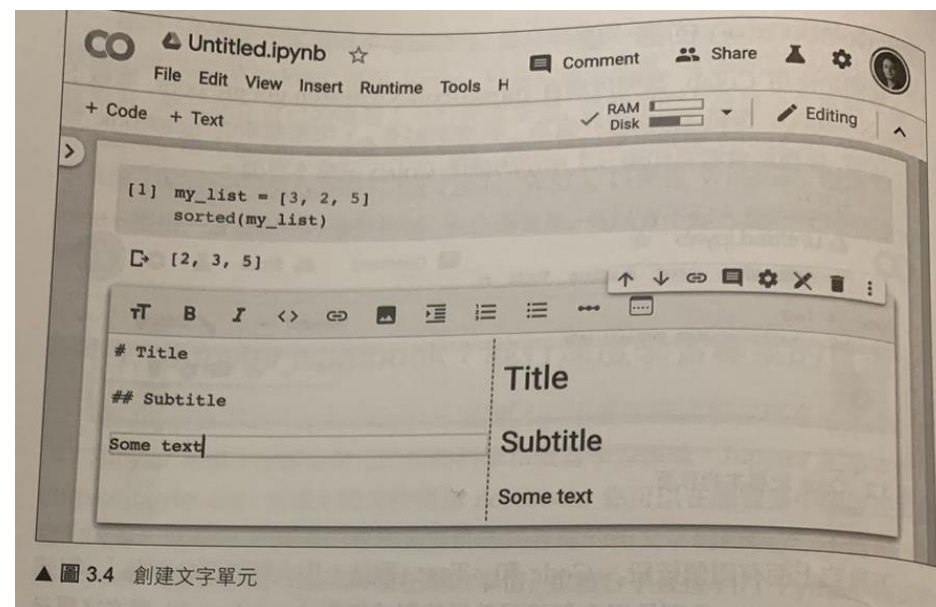
3-4-2 使用 Colaboratory

程式碼單元輸入程式之後，按下Shift+ Enter 就可以執行它

文字單元可用來說明程式內容：透過段落標題與文字解說段落，或者內嵌圖形。如此一來，就可以將原本單調的 notebook 檔案變成資訊豐富的多媒體文件。



▲ 圖 3.3 創建並執行程式碼單元



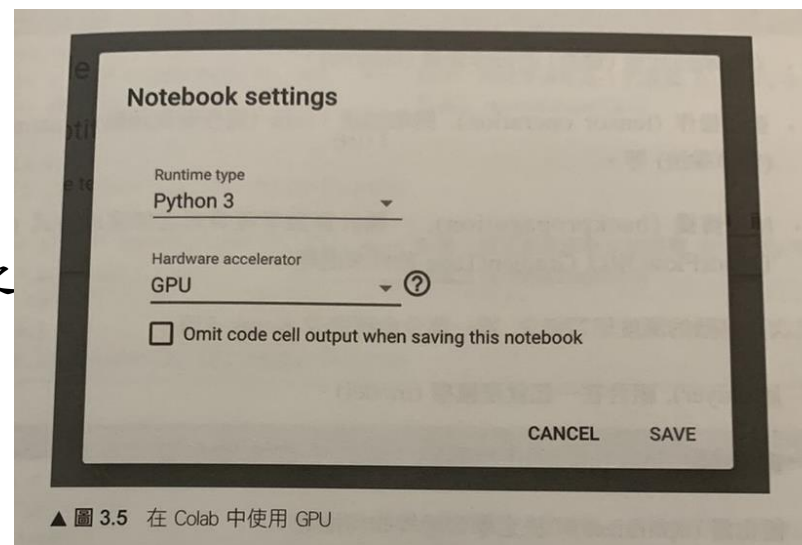
▲ 圖 3.4 創建文字單元

3-4-2 使用 Colaboratory

使用 GPU進行運算

要在 Colab 中使用 GPU 進行運算，選擇主選單中的Runtime > ChangeRuntime Type，並在 Hardware Accelerator 選項中選擇 GPU。

如果有可用的 GPU，TensorFlow 與 Keras 會自動在 GPU 上執行各項運算。因此，在你選擇 GPU 作為硬體加速器之後，就不需要再多做什麼事情



▲ 圖 3.5 在 Colab 中使用 GPU

3-5使用 Tensorflow的第一步

訓練神經網路牽涉到以下概念：

首先，低階的張量運算會貫穿整個機器學習過程，這一部分會轉換成TensorFlow API：

- 存放網路狀態（變數）的特殊張量（tensors）。
- 張量操作（tensor operation），例如加法、relu（線性整流函數）、matmul（矩陣乘法）等。
- 反向傳播（backpropagation），一種計算數學運算式之梯度的方式。

3-5 使用 Tensorflow 的第一步

其次是高階的深度學習概念，這一部分會轉換成 Keras API：

- 層（Layer），組合在一起就是模型（model）。
- 損失函數（loss function）：定義學習階段所用的回饋信號。
- 優化器（optimizer）：決定學習過程如何推進。
- 用來評估模型表現的各種評量指標（metric），例如準確度（accuracy）。
- 執行小批次隨機梯度下降（mini-batch stochastic gradient descent）的訓練迴圈（training loop）。

3-5-1 常數張量與變數

不管要利用 TensorFlow 來做什麼事情，我們都離不開張量。在建立張量時，需要為其指定一個初始值。舉例來說，我們可以創建一個完全由1或0組成的張量（程式3.1），或者是由亂數組成的張量（程式3.2）

程式 3.1 創建完全由 1 或 0 組成的張量

```
>>> import tensorflow as tf
>>> x = tf.ones(shape=(2, 1))  ← 創建一個元素值皆為 1 的張量, 在 NumPy 中
>>> print(x)                  相當於 np.ones(shape=(2,1))
tf.Tensor(
[[1.]
 [1.]], shape=(2, 1), dtype=float32)

>>> x = tf.zeros(shape=(2, 1)) ← 創建一個元素值皆為 0 的張量, 在 NumPy 中
>>> print(x)                  相當於 np.zeros(shape=2,1))
tf.Tensor(
[[0.]
 [0.]], shape=(2, 1), dtype=float32)
```

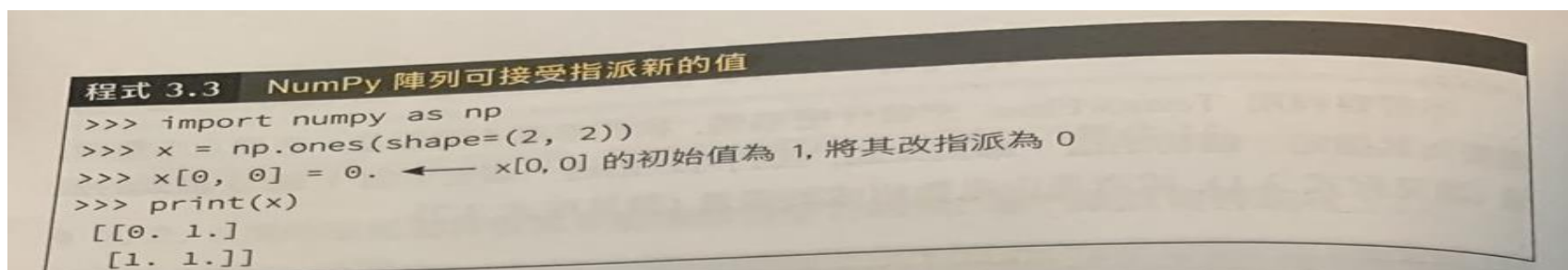
程式 3.2 創建由亂數組成的張量

```
>>> x = tf.random.normal(shape=(3, 1), mean=0., stddev=1.) ←
>>> print(x)              張量中的亂數從平均值 0、標準差 1 的常態分佈抽取而來,
tf.Tensor(                在 NumPy 中相當於 np.random.normal(size=(3,1), loc=0., scale=1.)
[[-1.6488864 ]
 [ 1.3780084 ]
 [ 0.10832578]], shape=(3, 1), dtype=float32)

>>> x = tf.random.uniform(shape=(3, 1), minval=0., maxval=1.) ←
>>> print(x)              張量中的亂數從 0 到 1 之間的均勻分佈抽取而來, 在
tf.Tensor(                NumPy 中相當於 np.random.uniform(size=(3,1), low=0., high=1.)
[[0.58475494]
 [0.3868904 ]
 [0.7811636 ]], shape=(3, 1), dtype=float32)
```

3-5-1 常數張量與變數

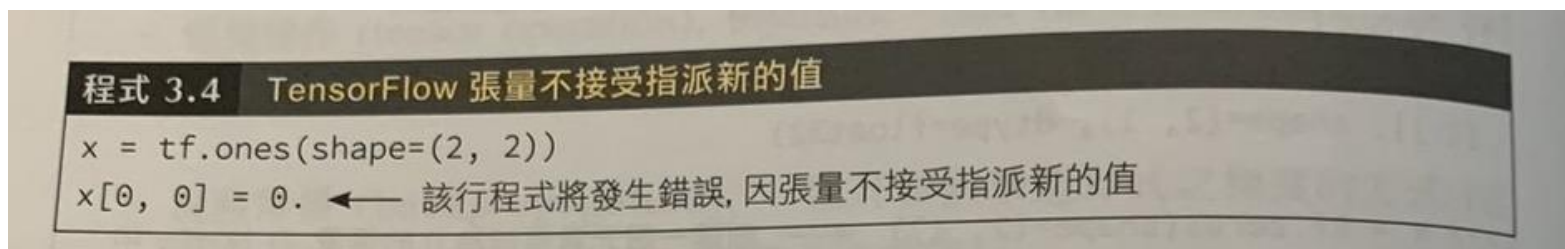
NumPy陣列與TensorFlow 張量的關鍵差異在於，TensorFlow 張量的值為常數（constant），無法接受指派新的值。例如，在 NumPy中可以執行以下操作：



程式 3.3 NumPy 陣列可接受指派新的值

```
>>> import numpy as np
>>> x = np.ones(shape=(2, 2))
>>> x[0, 0] = 0. ← x[0, 0] 的初始值為 1, 將其改指派為 0
>>> print(x)
[[0. 1.]
 [1. 1.]]
```

如果在 TensorFlow 執行同一操作，你將會得到錯誤訊息：EagerTensorobject does not support item assignment（EagerTensor 物件不接受指派）。



程式 3.4 TensorFlow 張量不接受指派新的值

```
x = tf.ones(shape=(2, 2))
x[0, 0] = 0. ← 該行程式將發生錯誤, 因張量不接受指派新的值
```

3-5-1 常數張量與變數

訓練模型時，我們需要不斷更新其狀態（為一組張量）。如果張量無法接受指派，那要如何進行更新呢？此時就要用到 `tf.Variable` 類別了，它是 TensorFlow 中負責操作可變狀態的類別。若想創建 `Variable` 物件，我們需要先指定初始值（例如：一個亂數張量）。

程式 3.5 創建 Variable 物件

```
>>> v = tf.Variable(initial_value=tf.random.normal(shape=(3, 1)))  
                                     ↑  
                               使用一個亂數張量來創建 Variable 物件  
  
>>> print(v)  
<tf.Variable 'Variable:0' shape=(3, 1) dtype=float32, numpy=  
array([[ -1.2214708 ],  
       [ -0.49477732 ],  
       [  1.7742974 ]], dtype=float32)>
```

我們可以透過 `assign()` 方法（method）來修改 `Variable` 物件的狀態，如下所示

程式 3.6 為 Variable 物件指派一個值

```
>>> v.assign(tf.ones((3, 1))) ← 將 v (程式 3.5 所創建的 Variable 物件)  
                               中的元素值改指派為 1  
  
<tf.Variable 'UnreadVariable' shape=(3, 1) dtype=float32, numpy=  
array([[1.],  
       [1.],  
       [1.]], dtype=float32)>
```


3-5-1 常數張量與變數

我們也可以對 Variable 物件中的數值進行局部修改：

程式 3.7 對 Variable 物件進行局部修改

```
>>> v[0, 0].assign(3.) ← 將第 0 列, 第 0 行的元素值改為 3
<tf.Variable 'UnreadVariable' shape=(3, 1) dtype=float32, numpy=
array([[3.], ← Variable 物件的局部值發生了改變
      [1.],
      [1.]], dtype=float32)>
```

如下所示，Variable 物件的 `assign_add()` 和 `assign_sub()` 函式等效於Python 的「+=」和「-=」算符。

程式 3.8 使用 `assign_add()` 函式

```
>>> v.assign_add(tf.ones((3, 1)))
<tf.Variable 'UnreadVariable' shape=(3, 1) dtype=float32, numpy=
array([[4.], ← Variable 物件中的每個數值都加上了 1
      [2.],
      [2.]], dtype=float32)>
```

3-5-2 張量操作：在 TensorFlow 中進行數學運算

和 NumPy 一樣，TensorFlow 也提供大量的張量操作函式，讓我們可以進行各種數學運算。來看一些例子：

程式 3.9 一些基本的數學操作

```
a = tf.ones((2, 2))  
b = tf.square(a)      ← 計算平方  
c = tf.sqrt(a)        ← 計算平方根  
d = b + c              ← 張量相加 (逐元素相加)  
e = tf.matmul(a, b)    ← 張量點積 (如 2-3-3 節所討論)  
e *= d                 ← 張量相乘 (逐元素相乘)
```

重點在於，每一項操作都是在當下馬上執行，你可以立刻將最新結果印出來（就跟在 NumPy 中一樣），我們稱這種執行方式為「即時執行」。

3-5-3 GradientTape API 的進一步說明

目前為止，TensorFlow 看起來就跟NumPy沒什麼兩樣，不過還是有些事情是NumPy做不到的：取得任意可微分函數對任意輸入項的梯度。在TensorFlow中，只需要設置一個 GradientTape 區塊，並在區塊中對（一個或多個）輸入張量進行計算，就可以取得計算結果對各個輸入張量的梯度。

程式 3.10 使用 GradientTape

```
>>> input_var = tf.Variable(initial_value=3.)  ← 建立一個 Variable 物件，其初始值為 3
>>> with tf.GradientTape() as tape:
>>>     result = tf.square(input_var)  ← 輸出結果為輸入的平方值
>>> gradient = tape.gradient(result, input_var)  ← 計算輸出對輸入之梯度
>>> print(gradient)
tf.Tensor(6.0, shape=(), dtype=float32)
```

此方法最常用來計算模型損失值（loss）對權重（weights）的梯度：`gradients = tape.gradient(loss, weights)`。

小編補充：在程式 3.10 中，輸出為輸入的平方值，以數學式來表達就是： $y=x^2$ ，其中 y 為輸出， x 為輸入。根據導數規則，輸出 y 對輸入 x 的導數為 $2x$ ，而以上程式中 x 的值為 3，故最終產生的梯度為 $2 \times 3 = 6$ 。只要在 GradientTape 區塊中建立相關運算式，未來就可用 `tape.gradient(輸出, 輸入)` 來取得梯度，無需自行推導。

3-5-3 GradientTape API 的進一步說明

到目前為止，我們只見過 `tape.gradient()` 中的輸入張量是 `Variable` 物件的例子。事實上，任何類型的張量都可以作為輸入。然而，系統預設只會追蹤 `GradientTape` 區塊中的可訓練變數（`trainable variable`，例如 `Variable` 物件）。對於常數張量，就必須用 `tape.watch()` 指定後才會進行追蹤：

程式 3.11 在 GradientTape 中使用常數張量為輸入

```
>>> input_const = tf.constant(3.) ← 建立一個常數張量
>>> with tf.GradientTape() as tape:
>>>     tape.watch(input_const) ← 用 tape.watch() 指定要追蹤
>>>     result = tf.square(input_const)
>>>     gradient = tape.gradient(result, input_const)
>>>     print(gradient)
tf.Tensor(6.0, shape=(), dtype=float32) ← 編註： 若沒加上 tape.watch()，輸出結果會是 None
```

為什麼要加上 `tape.watch()`？這是因為如果要計算並記錄所有張量之間的梯度變化，計算量跟資料量都會過於龐大。為了避免浪費資源，磁帶必須知道哪些才是需要計算跟追蹤的張量。系統預設會自動追蹤可訓練變數的原因是，計算「損失值對可訓練變數的梯度」就是梯度磁帶最主要的用途。

3-5-3 GradientTape API 的進一步說明

某物體移動距離對時間的一階梯度就是速度，二階梯度就是加速度。如果我們發現蘋果垂直落下的過程中，時間點與所在位置有以下的數學關係： $\text{position}(\text{time}) = 4.9 \times \text{time}^2$ ，那麼蘋果的加速度會是多少？使用巢狀（nested）的梯度磁帶來找出答案。

程式 3.12 利用巢狀梯度磁帶計算二階梯度

[illegible]

3-5-4 端到端的範例：使用TensorFlow 建立線性分類器

首先，我們要在平面上創建資料點，這些資料點分屬兩個不同的類別。我們會透過有著特定共變異數矩陣以及平均值的隨機分佈來抽出資料點的座標位置。直觀上，共變異數矩陣描述了點雲（point cloud，編註：一個點雲就是一群資料點）的形狀，平均值則描述這個點雲在平面上的中心位置。我們會以同一個共變異數矩陣來生成兩個點雲（但平均值不同），因此這兩個點雲的形狀相同，但中心位置不同。

程式 3.13 在平面上生成兩個類別的隨機資料點

```
num_samples_per_class = 1000  ← 設定每個類別會有 1000 個資料點
negative_samples = np.random.multivariate_normal(  ← 生成第一個類別
    mean=[0, 3],  ← 點雲的中心位置
    cov=[[1, 0.5],[0.5, 1]],  ← 產生的點雲會是橢圓形的,
    size=num_samples_per_class)  ← 從左下往右上進行延伸
positive_samples = np.random.multivariate_normal(  ← 生成另外一個類別的
    mean=[3, 0],  ← 資料點雲：形狀相同,
    cov=[[1, 0.5],[0.5, 1]],  ← 但中心位置不同
    size=num_samples_per_class)
```


3-5-4 端到端的範例：使用TensorFlow 建立線性分類器

程式 3.13 在平面上生成兩個類別的隨機資料點

```
num_samples_per_class = 1000  ← 設定每個類別會有 1000 個資料點
negative_samples = np.random.multivariate_normal(  ← 生成第一個類別
    mean=[0, 3],  ← 點雲的中心位置
    cov=[[1, 0.5],[0.5, 1]],  ← 產生的點雲會是橢圓形的,
    size=num_samples_per_class)  ← 從左下往右上進行延伸
positive_samples = np.random.multivariate_normal(  ← 生成另外一個類別的
    mean=[3, 0],  ← 資料點雲：形狀相同,
    cov=[[1, 0.5],[0.5, 1]],  ← 但中心位置不同
    size=num_samples_per_class)
```

在程式 3.13 中，negative_samples 和 positive_samples 都是shape為 (1000, 2) 的陣列（編註：1000代表資料點數量，2代表資料點的座標維度，也就是x、y座標），我們現在將這兩個陣列堆疊成 (2000, 2) 的陣列。

程式 3.14 將資料點堆疊成 shape 為 (2000, 2) 的新陣列

```
inputs = np.vstack((negative_samples, positive_samples)).astype(np.float32)
```

小編補充：np.vstack() 會沿著垂直方向堆疊陣列，因此 inputs 前 1000 列的資料為 negative_samples 中的各資料點，接著的 1000 列資料則是 positive_samples 中的 1000 個資料點。

3-5-4 端到端的範例：使用TensorFlow 建立線性分類器

接下來，要產生與資料點對應的目標值陣列（存成 `targets`），shape為（2000，1），其中的值為0或1，若`inputs[i]`屬於類別0，則`targets[i,0]`為0；若`inputs[i]`屬於類別1，則 `targets[i,0]`為1（編註：此處設定 `negative_samples`中的資料點為類別 0；`positive_samples` 中的資料點為類別 1）。

程式 3.15 產生對應的目標值 (0 或 1)

```
targets = np.vstack((np.zeros((num_samples_per_class, 1), dtype="float32"),  
                    np.ones((num_samples_per_class, 1), dtype="float32")))
```

3-5-4 端到端的範例：使用TensorFlow 建立線性分類器

接下來，使用 Matplotlib 來繪製資料點。

程式 3.16 繪製兩個類別的資料點 (參見圖 3.6)

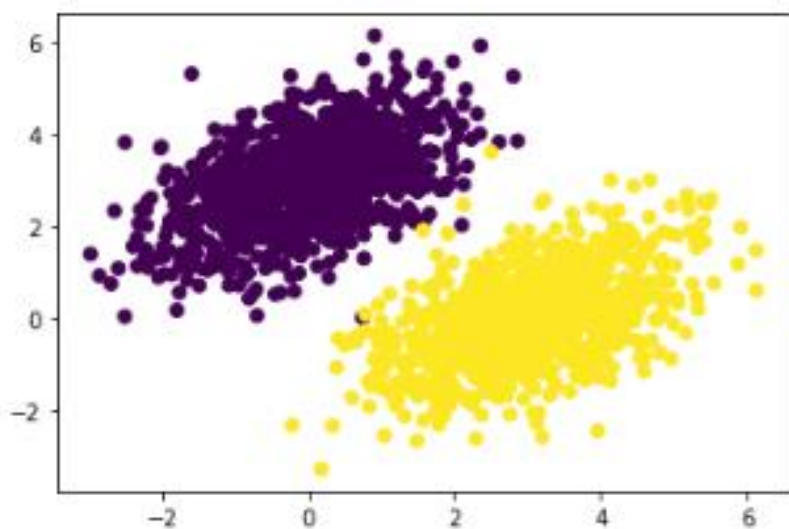
```
import matplotlib.pyplot as plt  
plt.scatter(inputs[:, 0], inputs[:, 1], c=targets[:, 0])
```

資料點的 x 座標

資料點的 y 座標

用目標值來區分不同類別資料點的顏色

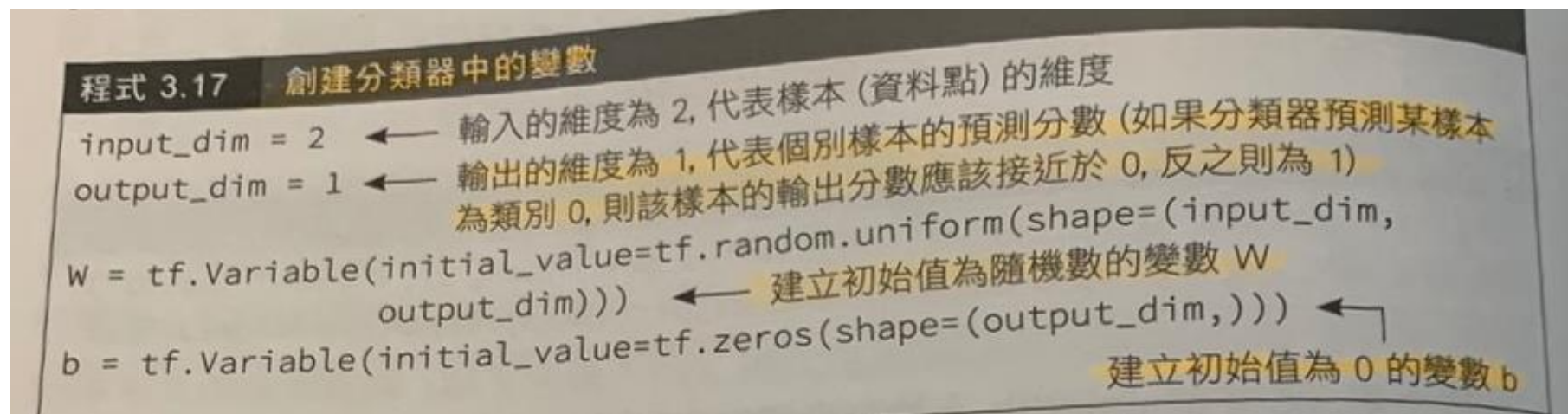
```
plt.show()
```



◀ 圖 3.6 平面上兩個類別的資料點 (深色點為類別 0；淺色點為類別 1)

3-5-4 端到端的範例：使用TensorFlow 建立線性分類器

接下來要設計一個線性分類器，並學會如何分類資料點，線性分類器就是一個以「最小化預測值與目標誤差的平方」，為目標來進行訓練的仿射變換(affine transformation)，以數學式來說，即 $\text{prediction} = W * \text{input} + b$ ，其中W和b均為變數。先來創建變數W（初始值為隨機數）和b（初始值為0）。



3-5-4 端到端的範例：使用TensorFlow 建立線性分類器

接下來便是正向傳播的函式

程式 3.18 正向傳播函式

```
def model(input):  
    return tf.matmul(inputs, W) + b
```

由於分類器處理的是 2D 輸入（編註：輸入的第0軸為資料點數目，第1 軸為個別資料點的座標值），因此 W 是由兩個純量（ w_1 和 w_2 ）所組成： $W = [[w_1], [w_2]]$ ；而 b 則是一個純量。對於任意輸入點 $[x, y]$ ，預測值會是 $\text{prediction} = [[w_1], [w_2]] \cdot [x, y] + b = w_1 * x + w_2 * y + b$ 。

接下來則是我們的損失函數。

程式 3.19 均方誤差損失函數

```
def square_loss(targets, predictions):  
    per_sample_losses = tf.square(targets - predictions) ←  
    per_sample_losses 張量的 shape 與 targets 和 predictions  
    張量一致, 內存有每個樣本 (資料點) 的損失分數  
    return tf.reduce_mean(per_sample_losses) ←  
    reduce_mean() 可計算各樣本損失分數之  
    平均值, 進而輸出單一的純量損失值
```

3-5-4 端到端的範例：使用TensorFlow 建立線性分類器

接下來便是訓練函式的部分。該函式會透過訓練資料來更新權重 W 與 b ，讓損失值達到最小。

程式 3.20 訓練函式

```
learning_rate = 0.1 ← 將學習率設為 0.1

def training_step(inputs, targets):
    with tf.GradientTape() as tape:
        predictions = model(inputs)
        loss = square_loss(targets, predictions)
        grad_loss_wrt_W, grad_loss_wrt_b = tape.gradient(loss, [W, b])
    w.assign_sub(grad_loss_wrt_W * learning_rate)
    b.assign_sub(grad_loss_wrt_b * learning_rate)
    return loss
```

在梯度磁帶區塊內進行正向傳播

計算損失值對權重 W 和 b 的梯度

更新權重

3-5-4 端到端的範例：使用TensorFlow 建立線性分類器

為了簡單起見，我們會使用批次訓練（batch training），而非小批次訓練（mini-batch training）。也就是說，我們會一次性對所有資料進行訓練（計算梯度並更新權重），而不是對多個小批次的資料做迭代。因此，執行一次訓練函式所花費的時間會比較久，畢竟我們需一次性計算 2,000 個樣本的正向傳播結果及梯度。這樣一來，我們需要的訓練次數就會減少很多，同時我們使用的學習率應該比小批次訓練來得大（在程式3.20中，我們將學習率設定成 0.1）。

程式 3.21 批次訓練迴圈

```
for step in range(40): ← 進行 40 次訓練
    loss = training_step(inputs, targets)
    print(f"Loss at step {step}: {loss:.4f}")
```

在40次訓練後，損失值似乎在 0.025 附近趨於穩定。

```
Loss at step 0: 4.7725
Loss at step 1: 0.6526
Loss at step 2: 0.2074
Loss at step 3: 0.1462
Loss at step 4: 0.1304
Loss at step 5: 0.1207
Loss at step 6: 0.1123
Loss at step 7: 0.1047
Loss at step 8: 0.0977
Loss at step 9: 0.0914
Loss at step 10: 0.0856
Loss at step 11: 0.0803
Loss at step 12: 0.0754
Loss at step 13: 0.0710
Loss at step 14: 0.0670
Loss at step 15: 0.0633
Loss at step 16: 0.0599
Loss at step 17: 0.0568
Loss at step 18: 0.0540
Loss at step 19: 0.0514
Loss at step 20: 0.0490
Loss at step 21: 0.0469
Loss at step 22: 0.0449
Loss at step 23: 0.0431
Loss at step 24: 0.0415
...
Loss at step 36: 0.0299
Loss at step 37: 0.0294
Loss at step 38: 0.0289
Loss at step 39: 0.0285
```

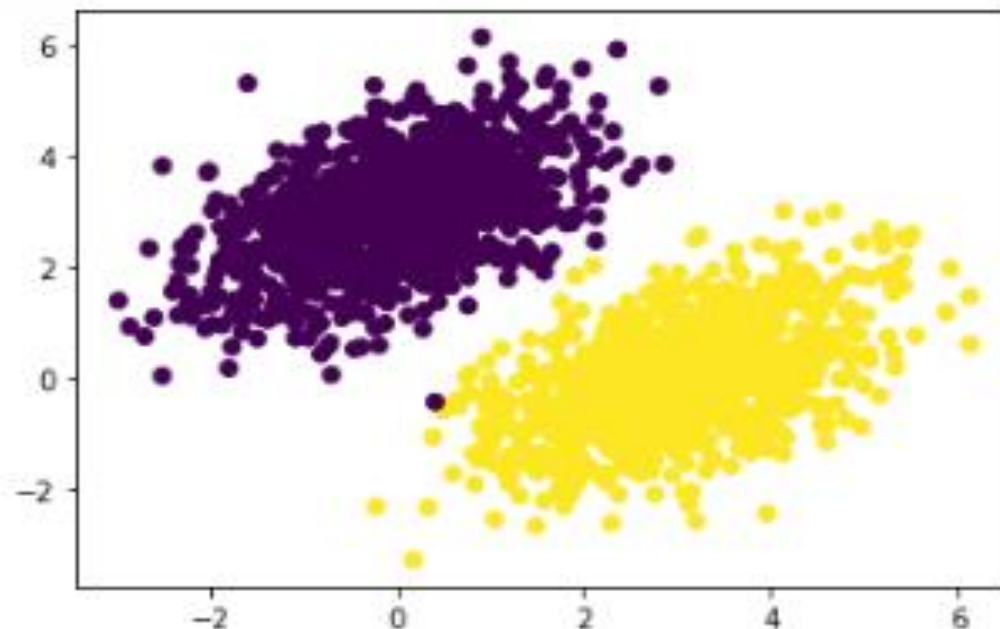
3-5-4 端到端的範例：使用TensorFlow 建立線性分類器

接下來將透過繪圖來檢查分類器的表現、由於我們的目標是0與1，因此如果某樣本的預測值小於0.5，就將其歸類為類別0，反之則歸類為類別1。

可以發現用來區分資料點類別的，其實是平面上的
一條直線： $w_1 * x + w_2 * y + b = 0.5$ 。位於這條線
上方的資料點屬於類別 0，位於下方的資料點則屬
於類別 1。以 $y = a * x + b$ 這種格式來呈現的直線方程
式，若想以相同格式來表示我們的直線，則此處的
方程式會變成 $y = -w_1 / w_2 * x + (0.5 - b) / w_2$ 。

```
predictions = model(inputs)
plt.scatter(inputs[:, 0], inputs[:, 1], c=predictions[:, 0] > 0.5)
plt.show()
```

利用預測值為資料點上色，進而與真實資料點做比較



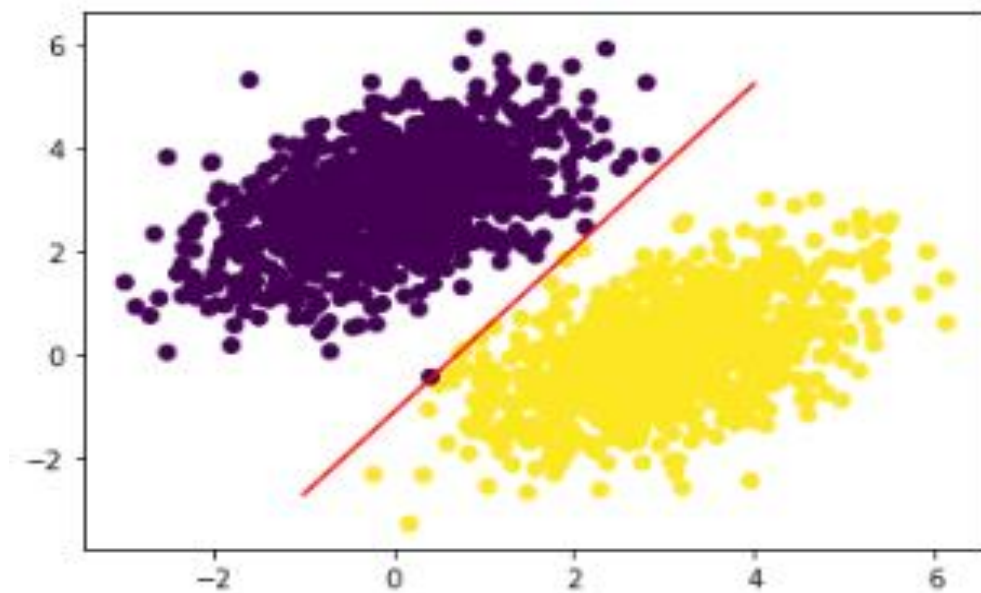
▲ 圖 3.7 預測結果似乎和真實資料一致

3-5-4 端到端的範例：使用TensorFlow 建立線性分類器

現在，把這一條分隔線畫出來（如圖 3.8所示）：

```
x = np.linspace(-1, 4, 100)  ← 在 -1 到 4 之間產生 100 個等距的數字，  
                               即 [-1, -0.95, -0.90, ..., 3.90, 3.95, 4.00]  
y = - W[0] / W[1] * x + (0.5 - b) / W[1]  ← 我們的直線方程式  
plt.plot(x, y, "-r")  ← 繪製出直線 ("-r" 表示直線的顏色為紅色)  
plt.scatter(inputs[:, 0], inputs[:, 1], c=predictions[:, 0] > 0.5)
```

以上就是訓練線性分類器的完整流程：找到一條直線的權重參數，使其可以整齊地區隔開不同類別的資料（如果是高維空間，用來區隔資料的會是一個平面，我們稱之「超平面」）。



▲ 圖 3.8 將我們的模型視覺化為一條直線

3-6 剖析神經網路：了解 Keras API的核心

3-6-1 Layer（層）：深度學習的基石

layer（層）是神經網路的基本資料處理模型（data-processing module），他可以接受一個或多個張量輸入，再輸出一個或多個張量。

層的權重可視為該層的狀態（State），在經過隨機梯度下降法（SGD）不斷更新機種（學習）之後，最終得到的權重即成為該神經網路的智慧所在。

不同格式的資料需要用不同的層處理。如果輸入資料是簡單的1D 向量資料，則多半是儲存在 2D 張量中，其 shape 為（樣本 Samples，特徵 features）），通常是用密集連接層（densely connected layer）來處理。在 Keras 中是以 Dense 類別來實作。

如果輸入資料是 2D 序列（sequence）資料，則多半是儲存在3D張量中，其 shape 為（樣本 samples，時戳 timesteps，特徵 features），通常是用循環層（recurrent layer，例如LSTM 層）來處理。

如果是3D 影像資料，則多半是儲存在4D 張量中，通常是用2D卷積層（Conv2D layer）來處理。（所有的資料都會在第0軸多加上樣本軸，因此軸數都會 +1，例如簡單的1D 向量資料就會變成 2D張量資料。）

3-6-1 Layer（層）：深度學習的基石

Keras 中的基礎 Layer 類別

Layer 類別是 Keras 的核心，每個 Keras 元件都是一個 Layer 物件或與 Layer 有密切互動。

Layer 是將一些狀態（權重）和運算（正向傳播）包在一起的物件。雖然權重可以在建構子

`_init_()`中建立，不過我們一般會使用 `build()`來建立它，而正向傳播的運算過程則是用 `call()`

方法來定義。

3-6-1 Layer（層）：深度學習的基石

在前一章，我們實作過一個名為 `NaiveDense` 的類別，其中包含了兩個權重 W 和 b ，同時進行了 $\text{output} = \text{activation}(\text{dot}(\text{input}, W) + b)$ 的運算。該類別在 Keras 中的實作方式如下：

程式 3.22 使用 Layer 類別來實作 Dense 層

```
from tensorflow import keras

class SimpleDense(keras.layers.Layer):
    ← 所有的 Keras 層都會繼承自基礎的 Layer 類別

    def __init__(self, units, activation=None):
        super().__init__()
        self.units = units
        self.activation = activation

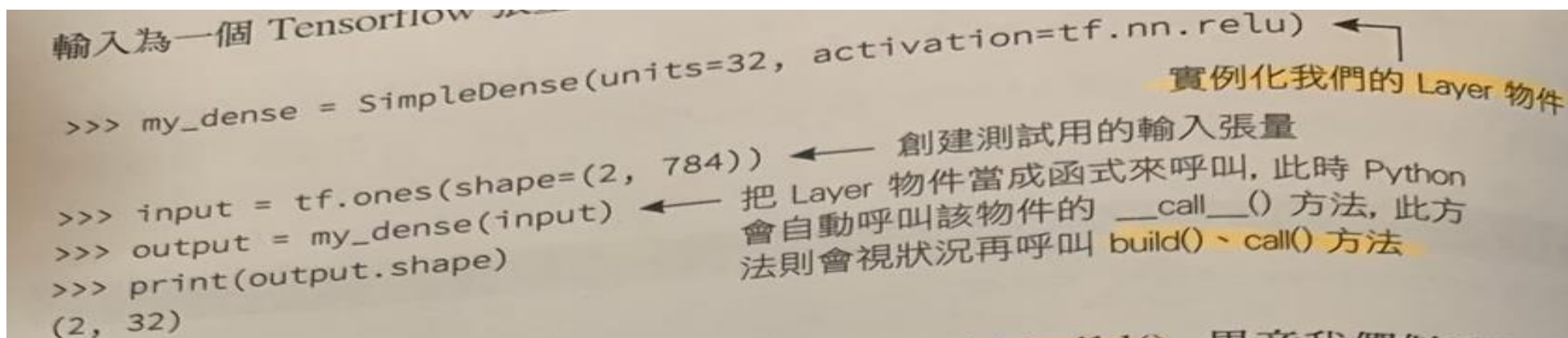
    def build(self, input_shape):
        ← 在 build() 中建立權重張量
        input_dim = input_shape[-1]
        self.W = self.add_weight(shape=(input_dim, self.units),
                                ←
                                initializer="random_normal")
        ← add_weight() 可以很輕鬆地創建權重,但其實也可以改成自行創建一個 Variable 物件,並將其指派給 layer 的 W 屬性,例如 self.W=tf.Variable(tf.random.uniform(w_shape))

        self.b = self.add_weight(shape=(self.units,),
                                initializer="zeros")

    def call(self, inputs):
        ← 在 call() 中定義正向傳播的運算過程
        y = tf.matmul(inputs, self.W) + self.b
        if self.activation is not None:
            y = self.activation(y)
        return y
```


3-6-1 Layer（層）：深度學習的基石

我們可以像使用函式一樣來使用實例化（instantiated）的 Layer 物件、其輸入為一個 Tensorflow 張量：



```
輸入為一個 Tensorflow 張量
>>> my_dense = SimpleDense(units=32, activation=tf.nn.relu)
>>> input = tf.ones(shape=(2, 784))
>>> output = my_dense(input)
>>> print(output.shape)
(2, 32)
```

實例化我們的 Layer 物件

創建測試用的輸入張量

把 Layer 物件當成函式來呼叫，此時 Python 會自動呼叫該物件的 `__call__()` 方法，此方法則會視狀況再呼叫 `build()`、`call()` 方法

為何不將所有功能寫在 `__init__` 及 `__call__` 方法中就好？這是因為我們想要在第一次呼叫 layer 物件時才即時創建其狀態（權重）張量，看看內部是怎麼運作的！

3-6-1 Layer（層）：深度學習的基石

自動推論出權重的 shape：即時建構layer 的權重

和樂高積木一樣，只有具備相容性的layer 才能扣在一起。所謂相容性，即每一layer只能接受特定 shape 的輸入張量，並輸出特定 shape 的張量。參考下面的例子：

```
from tensorflow.keras import layers  
layer = layers.Dense(32, activation="relu")
```

有著 32 個輸出單元的 dense 層

以上的 layer 會傳回一個張量，該張量第1軸的維度已經固定成 32了（編註：其 shape 為（批次量，32））。因此，該 layer 必須連接到預期輸入 shape為（批次量，32）的下游 layer。

3-6-1 Layer（層）：深度學習的基石

不過當我們使用Keras 時，並不需要擔心相容性的問題，因為模型中每一層的權重張量，都是由Keras 自動依照其第一次輸入的張量來即時建立的，換句話說，它會自動配合上一層傳給它的張量的 shape。例如以下列的程式來建構模型

```
from tensorflow.keras import models
from tensorflow.keras import layers
model = models.Sequential([
    layers.Dense(32, activation="relu"),
    layers.Dense(32)
])
```

以上程式中的 Layers物件並沒有指定任何有關輸入的 shape。相反的，它們會自動以 input 層第一次輸入的 shape 來推論各層輸入的 shape。

3-6-1 Layer（層）：深度學習的基石

在第2章所實作的 Dense層中（名稱為 NaiveDense），我們需要明確地將各層的輸入大小傳入建構子，如此一來才能創建出權重。但這並不是一個理想的做法，因為由此創建的每一層，都需要用 `input_size` 參數明確指定輸入 `shape` 來配合前一層的 `shape`。

```
model = NaiveSequential([
    NaiveDense(input_size=784, output_size=32, activation="relu"),
    NaiveDense(input_size=32, output_size=64, activation="relu"),
    NaiveDense(input_size=64, output_size=32, activation="relu"),
    NaiveDense(input_size=32, output_size=10, activation="softmax")
])
```

編註： 此層各張量的 `shape`：輸入為 `(batch, 784)`，輸出為 `(batch, 32)`，而 $y = \text{dot}(x, W) + b$ ，因此 `W` 為 `(784, 32)`，而 `b` 為 `(32,)`

如果遇到某些層輸出 `shape` 的規則很複雜時，情況會變得更糟。試想，若某層輸出的 `shape` 為 `(batch, input_size*2 if input.size%2==0 else input_size*3)`，對我們來說是非常不好計算而且容易出錯的。

3-6-1 Layer（層）：深度學習的基石

若我們將先前的 NaiveDense 層改寫為能自動推論 shape 的Keras 層，那它長得就會像之前的 SimpleDense 層一樣（程式 3.22），因此同樣也要有build()和 call（）方法。

在 SimpleDense 中，我們是以 build()方法來創建權重，該方法會接受輸入資料為其參數，當首次呼叫 SimpleDense 層時，就會自動呼叫 build()方法（透過該 Layer 子類別的_call_()方法）。基礎 Layer 類別的_call_（）方法已由Keras 定義好，其架構大致如下：

```
def __call__(self, inputs):  
    if not self.built:  
        self.build(inputs.shape) ← 首次呼叫 layer 物件時，會自動呼叫 build() 方法  
        self.built = True  
    return self.call(inputs) ← 每次呼叫 layer 物件時，都會  
                                呼叫 call() 並 return 其傳回值
```

編註：Layer 類別的 __call__() 方法已由 Keras 定義好了，在其內部還做了許多其他重要的事情，因此我們通常只要改寫 build() 和 call() 就好，而不必也不建議改寫 __call__() 方法。

3-6-1 Layer（層）：深度學習的基石

有了自動推論 Shape 的功能後，先前用來建構模型的程式就會變得簡單乾淨許多：

```
model = keras.Sequential([  
    SimpleDense(32, activation="relu"),  
    ↑  
    單元 (units) 數, 請參見程式 3.22  
    SimpleDense(64, activation="relu"),  
    SimpleDense(32, activation="relu"),  
    SimpleDense(10, activation="softmax")  
])
```

值得一提的是，Layer 類別的 `__call__()` 方法能處理的事遠不止推論 shape。它還負責了很多事，特別是 eager 執行模式和 graph 執行模式之間的路徑選擇，以及有關輸入遮罩（input masking）的部分。目前只需要記得：在實作自己的 Layer 物件時，請把正向傳播的運算另外寫在 `call()` 方法中

3-6-2從層到模型

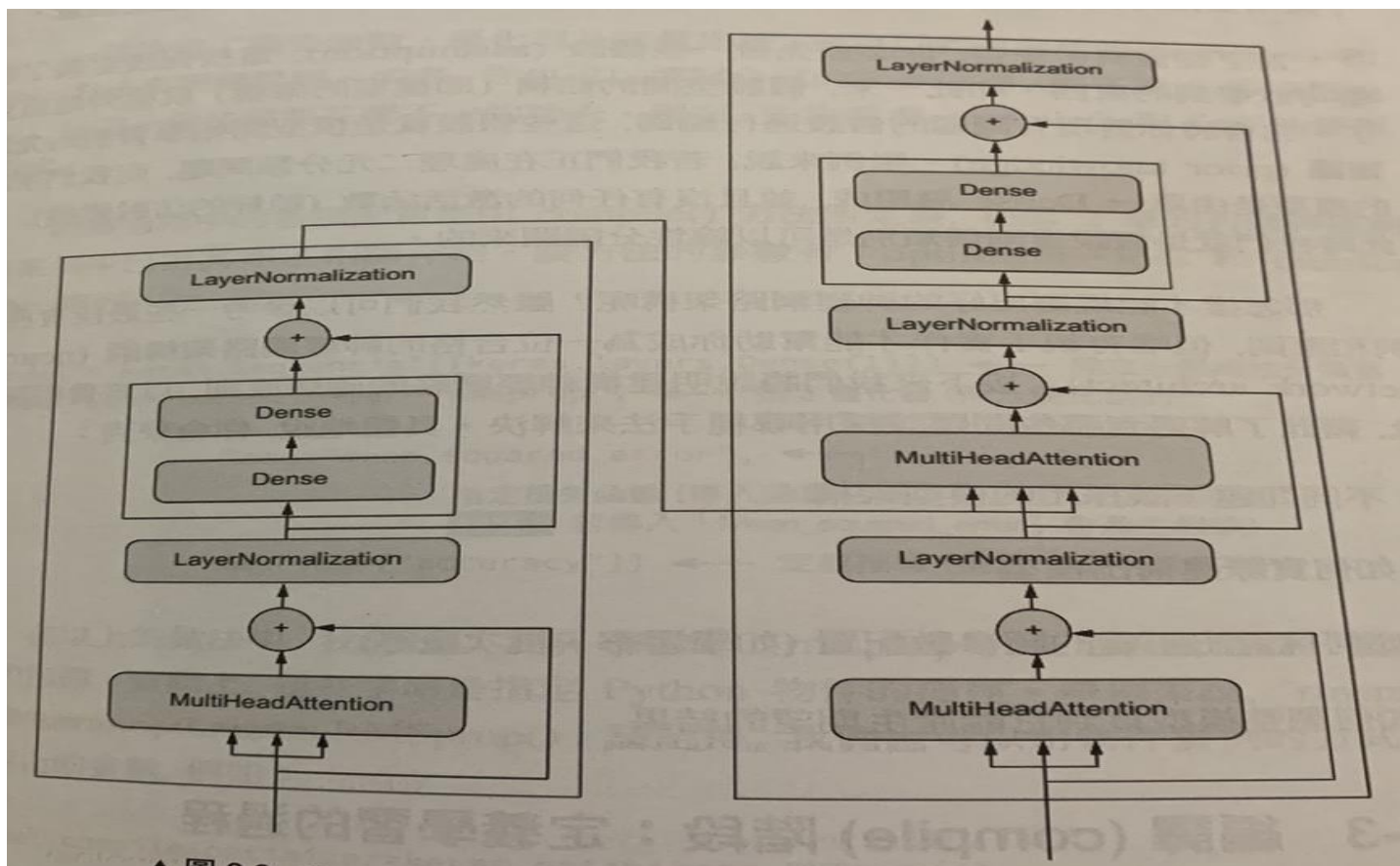
深度學習模型就是由多個層所組成的結構，在 Keras 中是以 Model 類別來建立模型物件。目前為止，我們只處理過 Sequential類別的模型（序列式模型），它是Model類別的子類別，由多個層簡單地堆疊而成，並具有單一的輸入端及單一的輸出端。在未來，我們還會接觸到種類更廣泛的神經網路拓樸（topology，層的不同組合方式）。常見的拓樸包括：

- 雙分支神經網路（Two-branch networks）（編註：中間有分支，而非只有線性連接）
- 多端口網路（Multihead networks）（編註：有多個輸入端或輸出端）
- 殘差連接（Residual connections）

（編註：某些層的輸出會多一條分支跳接到後遺的層）

3-6-2從層到模型

網路拓樸可以非常地複雜，圖3.9展示了 Transformer 模型的拓換圖，該架構常用來處理文字資料。



▲ 圖 3.9 Transformer 模型的架構 (將在第 11 章進行詳細說明)。
在接下來的幾章中, 你將慢慢了解圖中的一些內容

3-6-2從層到模型

一般來說，Keras 有兩種方式來建構以上這些非線性的模型；你可以直接創建 Model類別的子類別，也可以使用函數式 API (Functional API)、後者讓你能用更少程式碼做到更多的事情。

模型的拓模定義了一個假設空間（編註：hypothesis space，就是在該拓模下，所有權重參數位可能的組態）。選擇特定的神經網路拓模，就能將可能空間綁到特定的一系列張量運算上，借此將輸入資料轉換為對應的輸出資料。選好模型拓模之後，接下來所要做的就是搜尋一組可以讓張量運算（預測）發揮最佳效果的權重張量。

為了從資料中學習，你必須先做一些假設（assumption），這些假設定義了模型可以學到的東西。如此一來，假設空間的結構（即模型的架構）就格外地重要。它會將你對現有問題的假設進行編碼，這些假設就是模型開始學習前的先驗知識（prior knowledge）。

3-6-2從層到模型

舉例來說，若我們正在處理二元分類問題，而我們所用的模型是由單一 Dense 層組成，並且沒有任何的激活函數，那此時我們就是假設這兩個類別是可以線性分隔開來的。

那怎樣才能規劃出好的神經網路架構呢？雖然我們可以參考一些最佳實務案例和原則，但唯有親手實作才能幫助你成為一位合格的神經網路架構。接下來我們將說明建構神經網路的確切原則，以培養相關概念，藉此了解遇到哪些問題該採用哪種手法來解決。具體的說，會學到：

- 不同問題下該採用的模型架構
- 如何實際建構出模型
- 如何挑選出正確的超參數配置（如學習率、批次量等）
- 如何調整模型直到它能產生期望的結果

3-6-3 編譯（compile） 階段：定義學習的過程

確立了模型架構後，我們還需要決定3件事情：

- 損失函數（目標函数）：訓練期間要逐漸將此函數的傳回值最小化，這是衡量任務成功與否的關鍵。
- 優化器：決定如何根據損失函數來更新神經網路，使損失值變小。一般來說，會以隨機梯度下降法（SGD）所衍生的方法來執行。
- 評鼠指標：用來評量訓練階段和測試階段的模型表現，例如分類準確度。與損失值不同，訓練並不會直接使用這些指標來進行優化，因此指標不一定要是可微分的。（編註：一般來說，損失值是給優化器看的，而指標則是給我們人類看的。）

3-6-3 編譯 (compile) 階段：定義學習的過程

一旦決定了損失函數、優化器及評量指標，就可以利用內建的`compile()`和`fit()`方法來訓練模型。

現在，先來看看 `compile()` 和 `fit()`的使用方式。

訓練過程中的各項配置是由 `compile()` 方法所定義，該方法的參數有：`optimizer`、`loss` 和 `metrics`（為一串列）。

```
model = keras.Sequential([keras.layers.Dense(1)]) ← 建立一個線性分類器
model.compile(optimizer="rmsprop", ← 指定優化器 (傳入優化器的
                                名稱字串, 大小寫沒有差別)
              loss="mean_squared_error", ←
              編註： 指定損失函數 (傳入函數名稱字串, 但大小寫有差別,
              若傳入「Mean_squared_error」會產生錯誤)
              metrics=["accuracy"]) ← 定義指標 (傳入一個字串串列)
```

在以上的做法中，我們是以字串（例如："rmsprop"）來定義優化器、損失函數和指標。實際上，這些字串是指定Python物件的捷徑。舉例來說，"rmsprop"代表 `keras.optimizers.RMSprop()`。

3-6-3 編譯 (compile) 階段：定義學習的過程

我們也可以用物件實例的方式來指定以上的參數，例如：

```
model.compile(optimizer=keras.optimizers.RMSprop(),  
              loss=keras.losses.meanSquaredError(),  
              metrics=[keras.metrics.BinaryAccuracy()])
```

當你想要傳入客制化的損失函數或指標時，以上做法非常有用。該作法允許你更改物件的一些配置，例如在以下程式中，我們將 `learning_rate` 參數傳入 `optimizer` 物件，藉此來更改學習率：

```
model.compile(optimizer=keras.optimizers.RMSprop(learning_rate=1e-4),  
              loss=my_custom_loss, ← 傳入客制化的損失函數  
              metrics=[my_custom_metric_1, my_custom_metric_2]) ← 傳入客制化的指標
```

指定 `learning_rate` 參數值 (若不指定, 則採用預設值)

3-6-3 編譯 (compile) 階段：定義學習的過程

Keras 提供了種類廣泛的內建選項，基本上可以滿足你的需求：

- 優化器：

SGD (搭配/不搭配動量 momentum)

RMSprop

Adam

Adagrad 等等..

3-6-3 編譯 (compile) 階段：定義學習的過程

Keras 提供了種類廣泛的內建選項，基本上可以滿足你的需求：

- 損失函數：

CategoricalCrossentropy

SparseCategoricalCrossentropy

BinaryCrossentropy

MeanSquaredError

KLDivergence

CosineSimilarity 等等..

3-6-3 編譯 (compile) 階段：定義學習的過程

Keras 提供了種類廣泛的內建選項，基本上可以滿足你的需求：

- 評量指標：

CategoricalAccuracy

SparseCategoricalAccuracy

BinaryAccuracy

AUC

Precision

- Recall 等等… 在本書中，你將陸續看到以上選項的實際應用案例。

3-6-4 選擇損失函數

為特定問題選擇正確的損失函數是非常重要的，因為神經網路會據此調整權重參數，以減少損失值。如果損失函數與當前任務的成功與否不完全相關，那麼神經網路學習了半天，最終可能會得不到想要的結果。

假設我們隨便設定了一個目標（損失）函數：「最大化所有人類的平均幸福感」，然後經由 SGD 訓練出一個能夠使命必達的 AI，例如為了達成目標，這個AI 很可能選擇殺死大部分的人類，並將所有資源貫注於剩餘人類的幸福，因為分母（總人數）愈小，平均幸福感就愈大。

幸運的是，當遇到分類、迴歸和序列化預測等常見問題時，我們可以遵循一些簡單的準則來選擇正確的損失函數。例如：用二元交叉熵（binary crossentropy）處理二元分類問題；用分類交叉（categorical crossentropy）處理多類別分類問題等等。只有在處理全新的研究問題時，才需要開發自己的損失函數。

3-6-5 搞懂 fit () 方法

在使用 compile()之後，緊接著便要用到 fit()方法。該方法會自行實作訓練迴圈，其關鍵參數如下：

- 輸入 (inputs) 和目標值 (targets)：用來訓練的資料，包括輸入樣本和目標答案。資料會以 NumPy陣列或 TensorFlow 的 Dataset 物件之形式傳入。
- 週期數 (epochs)：訓練迴圈的重複次數，也就是要用所有資料重複訓練多少次。
- 批次量 (batch_size)：在每一週期中，進行小批次梯度下降訓練時的批次量，也就是每次訓練並更新權重時用來計算梯度的訓練樣本數量。

程式 3.23 透過 NumPy 資料來呼叫 fit()

```
history = model.fit(  
    inputs,          ← 此處的 inputs 為一 NumPy 陣列  
    targets,         ← 對應到 inputs 的目標值, 也是一個 NumPy 陣列  
    epochs=5,        ← 訓練迴圈會對資料迭代 5 次  
    batch_size=128   ← 訓練資料會切割為 128 個樣本組成的批次資料, 並用來進行訓練  
)
```

3-6-5 搞懂 fit () 方法

呼叫 fit() 會傳回一個 History 物件。該物件的 history 屬性是一個字典，其鍵（key）為 "loss" 或特定的評量指標名稱，值（value）則為每一訓練週期的損失值或指標值（儲存在串列中），如下程式所示：

```
>>> history.history
{'loss': [1.9968852996826172, ← 鍵為 'loss', 值為每一個訓練迴圈的損失值
 1.7974740266799927,
 1.6363857984542847,
 1.4866105318069458,
 1.3471424579620361],
 'binary_accuracy': [0.6675000190734863, ← 鍵為 'binary_accuracy', 值為
 0.6809999942779541, 每一個訓練迴圈的準確度
 0.6974999904632568,
 0.71100000252723694,
 0.7269999980926514]}
```

3-6-6：用驗證資料來監控損失值和指標

機器學習的目標並非取得只在訓練資料上表現良好的模型，要做到這一點相對容易，只需要跟著梯度進行優化即可。

我們的目標是取得在大部分狀況下都表現良好的模型，特別是在那些模型從未見過的資料點上。在訓練資料上表現良好，並不代表在從未見過的資料上也能表現良好。

舉例來說，你的模型有可能只是把訓練樣本和對應目標值「死背」起來，如此的模型在預測未見過資料的目標值時就毫無用處了。

為了得知模型在新資料上的表現，一般會保留訓練資料的一部分作為驗證資料（validation data）。我們並不會用驗證資料來訓練模型，但會用它來計算損失值和指標值。

3-6-6：用驗證資料來監控損失值和指標

我們可以透過 `fit()` 中的 `validation_data` 參數來傳入驗證資料。和訓練資料一樣，驗證資料可以是 NumPy 陣列或 Dataset 物件。

程式 3.24 使用 `validation_data` 參數

```
model = keras.Sequential([keras.layers.Dense(1)])
model.compile(optimizer=keras.optimizers.RMSprop(learning_rate=0.1),
              loss=keras.losses.MeanSquaredError(),
              metrics=[keras.metrics.BinaryAccuracy()])

indices_permutation = np.random.permutation(len(inputs))
shuffled_inputs = inputs[indices_permutation]
shuffled_targets = targets[indices_permutation]

num_validation_samples = int(0.3 * len(inputs))
val_inputs = shuffled_inputs[:num_validation_samples]
val_targets = shuffled_targets[:num_validation_samples]
training_inputs = shuffled_inputs[num_validation_samples:]
training_targets = shuffled_targets[num_validation_samples:]

model.fit(
    training_inputs,
    training_targets,
    epochs=5,
    batch_size=16,
    validation_data=(val_inputs, val_targets)
)
```

為了避免驗證資料中只有單一類別的樣本, 要先對訓練資料進行洗牌 (但樣本和目標值的對應關係不變)

保留 30% 的訓練資料來做驗證

訓練資料, 用來更新模型權重

驗證資料, 只用來計算驗證損失和指標

3-6-6；用驗證資料來監控損失值和指標

在驗證資料上的損失值稱為「驗證損失」，以此來和「訓練損失」做區分。

將訓練資料和驗證資料完全區隔開來是非常必要的：驗證資料是用來監看模型所學在新資料上是否能發揮作用。若模型在訓練階段就見過部分的驗證資料，那麼得到的驗證損失和指標值就會有瑕疵（編註：類似考前洩題以致成績較好）。

在訓練結束後，若想用特定資料來計算驗證損失和驗證指標值，可以使用`evaluate()`方法。

```
loss_and_metrics = model.evaluate(val_inputs, val_targets, batch_size=128)
```

該方法會對傳入的資料進行小批次驗證（批次量由 `batch_size` 指定，編註：若省略此參數則預設為 32），進而傳回由多個純量組成的串列，其中的第一個純量為驗證損失，緊接著便是驗證指標值（編註：若 `metrics` 設定了多個指標，則這裡也會有多個驗證指標值）。若沒有設定指標，則 `evaluate()` 就只會傳回驗證損失。

3-6-7 推論 (Inference) 階段：使用訓練好的模型來預測

-6-7 推論 (Inference) 階段：使用訓練好的模型來預測在訓練好模型後，就可以用它對新資料進行預測，這一階段稱為推論 (inference)。若要進行推論，只需直接在新資料上呼叫模型即可：

```
predictions = model(new_inputs) ← 接受一個 NumPy 陣列或 TensorFlow 張量, 並傳回一個 TensorFlow 張量
```

不過，以上做法會一次性處理 new_input 中的所有資料，這在資料量很大的情況下或許是不可行的一個更好的做法是使用 predict() 方法來進行推論。這樣一來就會在小批次（編註：可用 batch_size 指定大小，預設為 32）資料上迭代，並傳回存有預測值的 NumPy 陣列。本方法也可處理 TensorFlow 的 Dataset 物件。

```
predictions = model.predict(new_inputs, batch_size=128) ← 接受一個 NumPy 陣列或 Dataset 物件, 並傳回一個 NumPy 陣列
```

3-6-7推論 (Inference) 階段：使用訓練好的模型來預測

舉例來說，若我們對先前訓練過的線性模型使用 `predict()`，藉此查看模型在驗證資料上的表現，則我們會得到模型對每一個輸入樣本的預測值（為一純量分數）：

```
>>> predictions = model.predict(val_inputs, batch_size=128)
>>> print(predictions[:10])
[[0.6839796 ]
 [0.57421404]
 [0.12924099]
 [0.1350658 ]
 [0.26702708]
 [0.579858  ]
 [0.5298121 ]
 [0.07167074]
 [0.4662552 ]
 [0.18158592]]
```