前言

本書針對自然語言處理（NLP）與深度學習新手討論重要的主題。這些主題領域正以指數等級增長。本書介紹深度學習與 NLP 並強調實作，這部分是重要的學習基礎。寫作時，我們必須做出困難的決定來排除一些內容。對新手而言，我們希望這本書能提供扎實的基礎並一窺各種可能。機器學習，特別是深度學習，是相對於知識學習的經驗養成。每一章的程式範例可讓你參與體驗過程。

寫作本書時，我們使用 PyTorch 0.2，然後跟著 PyTorch 版本 0.2 到 0.4 改寫。PyTorch 1.0（*https://pytorch.org/2018/05/02/road-to-1.0.html*）會在本書出版後釋出。本書範例與 PyTorch 0.4 相容，應該也能在 PyTorch 1.0 上執行。[1]

我們在大多數內容中刻意避免數學，並非因為深度學習的數學特別難（其實不難），而是因為它會分散本書主要目標的注意力──讓新手入門。同樣的，程式與內文寫法偏向清楚闡述而非精簡，進階讀者與程式設計師可能會覺得程式能更精簡，但我們選擇盡可能說明清楚。

1 *https://pytorch.org/2018/05/02/road-to-1.0.html*。

第一章

介紹

Echo（Alexa）、Siri、Google Translate 等家喻戶曉的產品名稱至少有一個共同點，它們都是本書兩個主題之一的**自然語言處理**（*Natural Language Processing*，NLP）的應用。NLP 是指一群與語言學有關或無關的統計方法的文字理解應用技術。文字的“理解”主要來自將文字轉換成可用的計算機**表示法**（*representation*），它是向量、張、圖、樹等離散或連續的組合結構。

機器學習的主題就是從資料中（此例中的文字）學習適合任務的表示法。文字資料的機器學習應用已經有三十年以上，但過去十年間[1]一組稱為**深度學習**的機器學習技術的發展開始展現 NLP、語音、計算機視覺中的各種人工智慧（AI）的高效能。深度學習是本書的另一個主題；因此，本書研究的是 NLP 與深度學習。

參考出處列於每一章的後面。

簡單說，使用稱為**計算機圖**概念與數學最佳化技術的深度學習可有效的從資料學習表示法。Google、Facebook、Amazon 等科技公司採用計算圖架構與函式庫是深度學習與計算圖有效的證明。本書以基於 Python 的 *PyTorch* 計算圖架構實作深度學習演算法。這一章說明計算圖是什麼並解釋我們為何選擇 PyTorch 架構。

1 雖然社交網路與 NLP 的歷史很久，但 Collobert 與 Weston（2008）常被視為現代 NLP 的深度學習應用的起點。

機器學習與深度學習的範圍很大。本書與這一章主要採用**監督式**（*supervised*）**學習**，也就是以有標記的訓練範例學習。我們會說明作為本書基礎的監督式學習典範。若你還不熟悉這些術語，讀這本書就對了。這一章與後續內容不只會說明還會深入討論這些術語。若你已經熟悉前述的術語與概念，我們還是鼓勵你好好看完，原因有二：認識本書的詞彙用法與打好閱讀後續內容的基礎。

這一章的目標是：

- 認識監督式學習典範、認識術語、打好概念基礎。
- 學習如何編碼學習任務的輸入。
- 認識什麼是計算圖。
- 基本掌握 PyTorch。

讓我們開始！

監督式學習的典範

機器學習中的監督（supervision），或稱為**監督式學習**，指的是**觀察**（*observation*）的**目標**（*target*，預測的對象）有實值（ground truth）可用。以文件分類為例，目標是分類標籤[2]，觀察的是文件。在機器翻譯中，觀察的是某個語言的句子，目標是另一個語言的句子。圖 1-1 顯示這種輸入資料定義下的監督式學習的典範。

我們可如圖 1-1 所示將監督式學習的典範分為六個主要概念：

觀察（*observation*）

觀察是我們想要預測的項目。我們將觀察記為 x。我們有時稱觀察為**輸入**。

目標（*target*）

目標是對應觀察的標籤。它們通常是被預測的事物。我們採用機器學習 / 深度學習的慣例記為 y。有時這些標籤稱為**實值**。

2 分類變數是一組固定值中挑出的一個值；例如 {TRUE, FALSE}，{VERB, NOUN, ADJECTIVE, ...} 等。

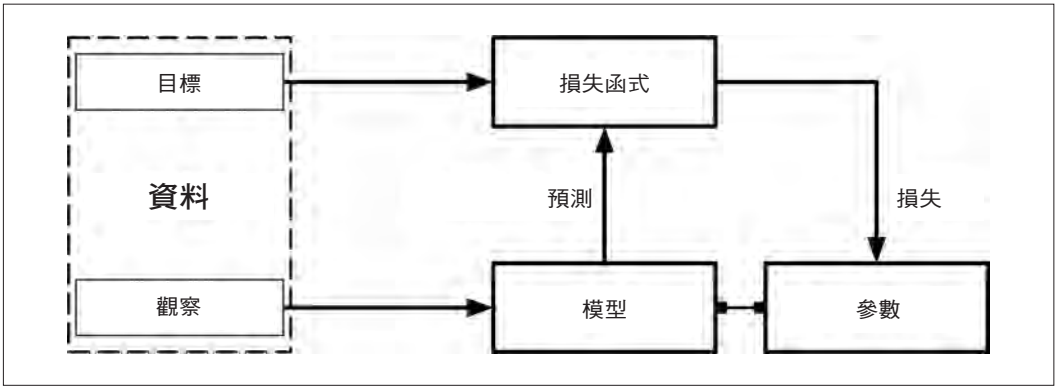

圖 1-1　監督式學習的典範，一種從有標記的輸入資料學習的概念性架構

模型（*model*）

模型是輸入觀察 x 並預測目標標籤值的數學運算式或函式。

參數（*parameter*）

有時稱為**權重**（*weight*），將模型參數化。標準記法為 w（代表 weight）或 $\hat{w}$。

預測（*prediction*）

預測又稱為**預估**，是模型輸入觀察後做出的目標猜測值。我們使用“戴帽子”記法，因此目標 y 的預測記為 $\hat{y}$。

損失函式（*loss function*）

損失函式是計算預測偏離訓練資料中的觀察目標有多遠的函式。損失函式對一個目標與其預測計算稱為**損失**的純量值。損失值越低則模型對目標的預測越好。我們將損失函式記為 L。

雖然 NLP/ 深度學習的模型設計沒有一定要以正規的數學表示，我們仍會正規的表示監督式學習典範，讓新入門的讀者熟悉標準術語與寫作風格好閱讀 arXiv 上的研究論文。

以資料集 $D = \{X_i, y_i\}_{i=1}^n$ 的 n 個範例來說，我們想要學習一個以權重 w 參數的函式（模型）f。也就是我們做出 f 的結構的假設，在此結構下，權重 w 的學習值會表現此模型的特徵。對一個輸入 X，模型的預測 $\hat{y}$ 為目標：

$$\hat{y} = f(X, w)$$

在監督式學習的訓練範例中，我們知道觀察的實目標 y。此例的損失為 $L(y, \hat{y})$。然後以監督式學習找出讓 n 個範例累積損失最小的最佳參數 / 權重 w。

以（隨機）梯度下降訓練

監督式學習的目的是挑出對輸入資料集最小損失函式的參數值。換句話說，這等於找到等式的根。我們知道**梯度下降**（*gradient descent*）是找到等式的根的常見技術。在傳統的梯度下降中，我們先猜一個根值（參數），並迭代更新參數直到對象函式（損失函式）求出低於可接受閥（threshold，又稱為 convergence criterion）的值。對大資料集而言，實作整個資料集的傳統梯度下降通常因為有限的記憶體而不可能辦到，且計算也會非常慢。此時通常會使用逼近梯度下降的**隨機梯度下降**（*stochastic gradient descent*，SGD）。這種方式會隨機挑選資料點或部分資料點進行梯度下降計算。使用單一資料點時，這種方式稱為純 SGD，使用一組（超過一個）資料點時，我們稱它為**小批 *SGD***。“純（pure）”與“小批（minibatch）”在採用方法很明顯時捨去。實務上，因雜訊導致的緩慢收斂使純 SGD 很少使用。一般 SGD 演算法有不同的變化，全都是針對快速收斂。後續的內容會在以梯度更新參數的過程中討論其中一些變化。這種迭代更新參數的程序稱為**反向傳播**（*backpropagation*）。每個反向傳播的步驟（又稱為代，epoch）由**向前傳遞**（*forward pass*）與**向後傳遞**（*backward pass*）組成。向前傳遞比較輸入與目前參數值並計算損失函式。向後傳遞以損失梯度更新參數。

以上的內容並沒有專對深度學習或神經網路[3]。圖 1-1 的箭頭方向表示訓練系統時的資料“流向”。我們在“計算圖”一節會有更多訓練與“流向”的討論，接下來先看看如何以數學表示 NLP 中的輸入與目標好讓我們訓練模型並預測結果。

觀察與目標的編碼

我們必須以數學表示觀察（文字）以供機器學習演算法使用。圖 1-2 顯示此概念。

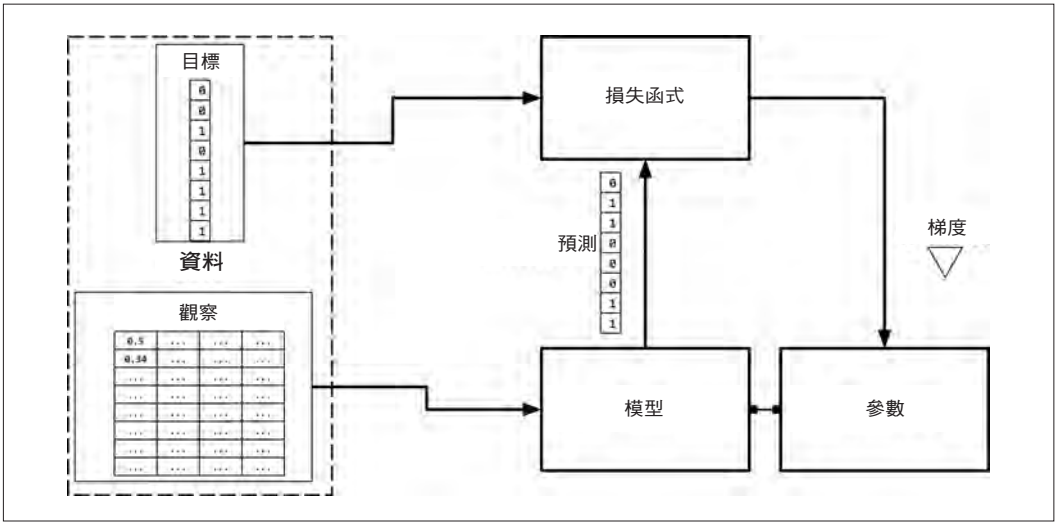

圖 1-2　觀察與目標的編碼：圖 1-1 所示的目標與觀察以向量或張量表示。這統稱為輸入的“編碼”。

表示文字的一種簡單辦法是數學向量。這種對應 / 表示的辦法有無限多種。事實上，本書大部分內容在於學習這些任務資料表示法，但我們會從啟發式的簡單計數表示法開始，它們雖然簡單但非常有效，且可作為更複雜的表示的起點。這些基於計數的表示都從固定維度的向量開始。

3　深度學習與 2006 年之前的文獻中討論的傳統神經網路的區別在於，前者指的是越來越多透過在網路中添加更多層來實現可靠性的技術。我們會在第三章和第四章討論為什麼這很重要。

獨熱表示

獨熱（*one-hot*）表示如名稱所示，從零向量開始，若該詞出現在句子或文件中，則將向量中相對應記錄設為 1。以下列兩個句子為例：

```
Time flies like an arrow.
Fruit flies like a banana.
```

將句子分解、忽略標點符號、全部視為小寫後產生 8 個詞彙：{time, fruit, flies, like, a, an, arrow, banana}。然後我們可以用八維獨熱向量表示每個詞彙。本書使用 1_w 指示一個 token/word 的獨熱表示。

片語、句子、或文件折疊後的獨熱表示只是其組成詞彙的獨熱表示的邏輯 OR。“like a banana”這樣的短句在使用圖 1-3 所示的編碼後，其獨熱表示為一個 3×8 矩陣，欄是八維獨熱向量。“折疊”或二進位編碼其中文字 / 短句由向量表示詞彙的長度，以 0 和 1 指示一個詞彙的存在與否也很常見。

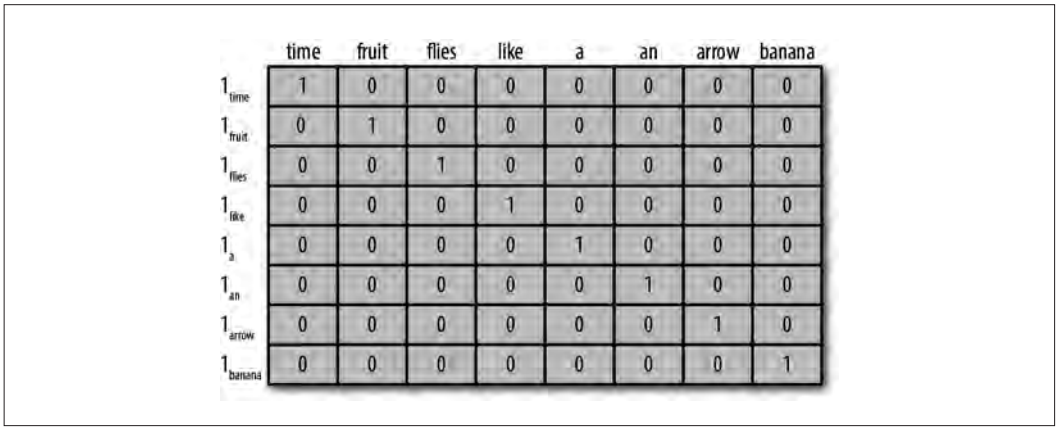

圖 1-3 “Time flies like an arrow”與“Fruit flies like a banana”句子編碼後的獨熱表示

如果你對我們折疊“flies”兩種語意（或意義）感到不妥，你是個細心的讀者！語言充滿模糊，但我們還是能用極度簡化的假設建造可用的方案。學習分辨意義的表示是可能的，但現在先不談。

雖然本書很少使用獨熱表示以外的輸入，但我們現在會介紹**詞頻**（*Term-Frequency*，TF）與**詞頻逆文件頻**（TF-IDF）表示。這是由於它們在 NLP 很常見，因為歷史因素、

以及完整性的考量。這些表示法在資訊擷取（information retrieval，IR）有很長遠的歷史，甚至還有 NLP 系統正在使用。

TF 表示法

短句、句子、文件的 TF 表示只是其組成詞彙的獨熱表示的集合。以前述的獨熱編碼繼續我們的例子，“Fruit flies like time flies a fruit”有著如此的 TF 表示：`[1, 2, 2, 1, 1, 0, 0, 0]`。注意每一筆是相對應詞彙出現在句子（語料庫）的計次。我們將一個詞 *w* 的 TF 記為 *TF(w)*。

範例 *1-1*　以 *scikit-learn* 產生“折疊”獨熱或二進位表示

```
from sklearn.feature_extraction.text import CountVectorizer
import seaborn as sns

corpus = ['Time flies flies like an arrow.',
          'Fruit flies like a banana.']
one_hot_vectorizer = CountVectorizer(binary=True)
one_hot = one_hot_vectorizer.fit_transform(corpus).toarray()
sns.heatmap(one_hot, annot=True,
            cbar=False, xticklabels=vocab,
            yticklabels=['Sentence 2'])
```

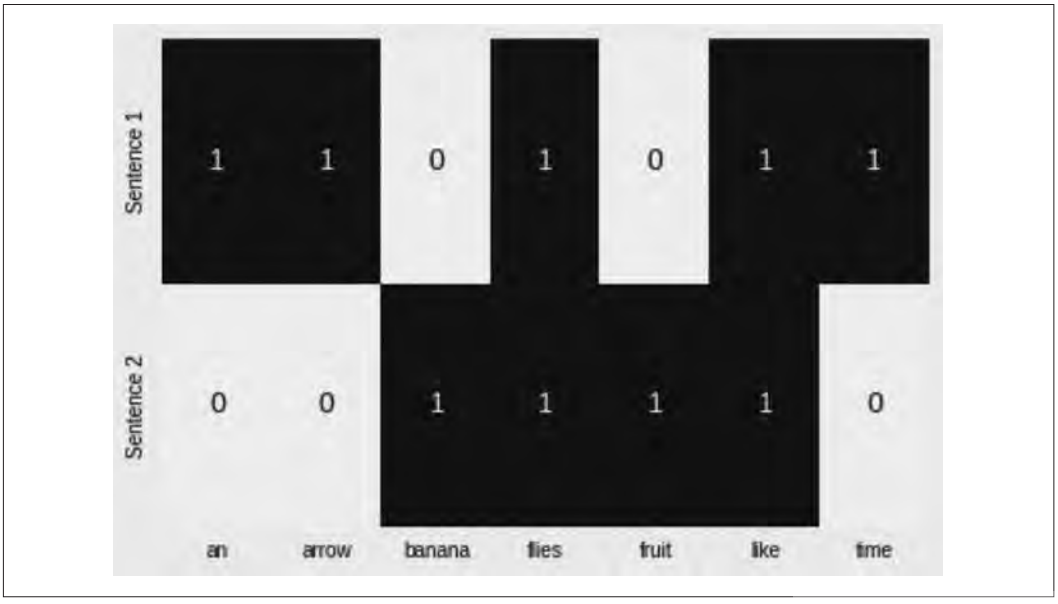

圖 1-4　範例 1-1 產生的折疊獨熱表示

TF-IDF 表示法

以一堆專利文件為例，你估計會遇到 *claim*、*system*、*method*、*procedure* 這樣的詞彙很多次。TF 表示法中，詞的權重與其頻率等比。但“claim”等常見詞彙並不會提高我們對特定專利的認識。相反的，若一個罕見詞彙（例如“tetrafluoroethylene”）少量出現、但非常可能指出該專利文件的特質，則我們會想要讓它在表示中有更高的權重。逆文件頻（IDF）是如此做的啟發法。

IDF 表示法懲罰向量表示中的常見詞並獎勵罕見詞。詞彙 *w* 的 *IDF(w)* 在語料庫中的定義為：

$$IDF(w) = \log \frac{N}{n_w}$$

是帶詞彙 *w* 的文件數量，*N* 是文件總數。TF-IDF 分數只是 *TF(w) * IDF(w)* 積。首先，注意到出現在每個文件中的極常見詞彙（也就是 $n_w = N$ ），*IDF(w)* 為 0 且 TF-IDF 分數為 0，因此完全懲罰該詞彙。其次，若一個詞彙極罕出現，或許只在一個文件中，則 IDF 會是最大可能值 log *N*。範例 1-2 顯示如何以 `scikit-learn` 產生一列英文句子的 TF-IDF 表示。

範例 1-2　以 scikit-learn 產生 TF-IDF 表示

```
from sklearn.feature_extraction.text import TfidfVectorizer
import seaborn as sns

tfidf_vectorizer = TfidfVectorizer()
tfidf = tfidf_vectorizer.fit_transform(corpus).toarray()
sns.heatmap(tfidf, annot=True, cbar=False, xticklabels=vocab,
            yticklabels= ['Sentence 1', 'Sentence 2'])
```

在深度學習中，很少看到輸入因目標是學習一種表示法，而以 TF-IDF 等啟發式表示法編碼。通常我們以使用整數索引的獨熱編碼與特殊的“嵌入查詢（embedding lookup）”層來建構神經網路的輸入。後續的章節中會有幾個例子。

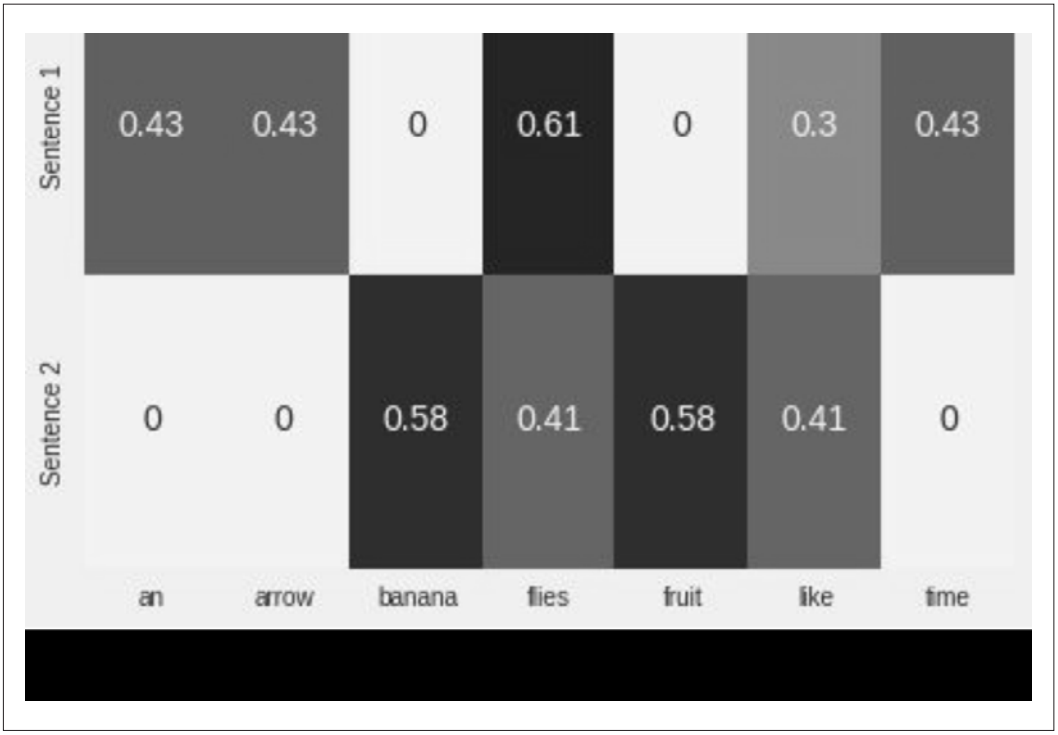

圖 1-5　範例 1-2 產生的 TF-IDF 表示

目標的編碼

如“監督式學習的典範”一節所述，目標變數的本質視要解決的 NLP 任務而定。以機器翻譯、摘要、答問為例，目標也是文字且使用前述的獨熱編碼等方式編碼。

許多 NLP 任務實際使用分類標籤，其模型必須從固定的標籤集合中進行預測。這種辦法常見的編碼方式是每個標籤使用獨特的索引，但這種簡單表示法在輸出標籤的數量非常大時會有很多問題。一個例子是語言模型設計問題，其任務是以前面的詞彙預測下一個詞彙。它的標籤空間是整個語言的詞彙，包含特殊字元、姓名時數量很容易上數十萬。我們會在後面的章節重新檢視這個問題並討論如何處理。

有些 NLP 問題涉及從文字中預測一個數值。以英文論文為例，我們可能必須給分或給可讀性分數。對一段餐廳評價，我們可能要預測星數到小數第一位。對一個使用者的推特，我們可能需要預測該使用者的年齡層。數值目標編碼有多種方式，但將目標分類到“罐”中（如“0-18”、“19-25”、“25-30”），並將它當做分類問題來處理是合理的辦法。[4] 分罐可以是平均或不平均且依資料而定。雖然其細節已經超出本書範圍，但我們提出這些議題是因為目標編碼在這種例子中大幅影響其效能，建議你參考 Dougherty et al. (1995) 與其中的參考來源。

計算圖

圖 1-1 以透過模型（數學運算式）轉換輸入成預測，並以損失函式（另一個運算式）提供回饋訊號來調整模型參數的資料流架構說明監督式學習（訓練）典範。此資料流能以計算圖資料結構實作 [5]。技術上，計算圖是數學運算式的模型的摘要。在深度學習中，計算圖的實作（例如 Theano、TnsorFlow、PyTorch 等）做了額外的記錄以實作監督式學習典範的訓練過程中取得參數梯度所需的自動微分。我們會在“PyTorch 基礎”中進一步討論。推論（或預測）只是運算式求值（計算圖中的向前流向）。讓我們看看計算圖如何做運算式的模型。以下列運算式為例：

$$y = wx + b$$

這可以寫成 $z = wx$ 與 $y = z + b$ 兩個子運算式，然後我們可以使用節點為乘法與加法等數學運算的有向無環圖（directed acyclic graph，DAG）表示原來的運算式，運算的輸入是向著節點的邊而輸出是離開節點的邊。因此，對 $y = wx + b$ 運算式，其計算圖如圖 1-6 所示。下一節會看到 PyTorch 如何讓我們直接建立計算圖，以及如何計算梯度，讓我們無需顧慮任何記錄工作。

4 “序數詞”分類是一種多類別分類問題，在標籤間存在部分順序。在我們的例子中，“0-18”分類在“19-25”前面，以此類推。

5 Seppo Linnainmaa（*http://bit.ly/2Rnmdao*）首先在其 1970 年的論文中提出自動微分！其各種變化成為 Theano、TensorFlow、PyTorch 等現代深度學習框架的基礎。

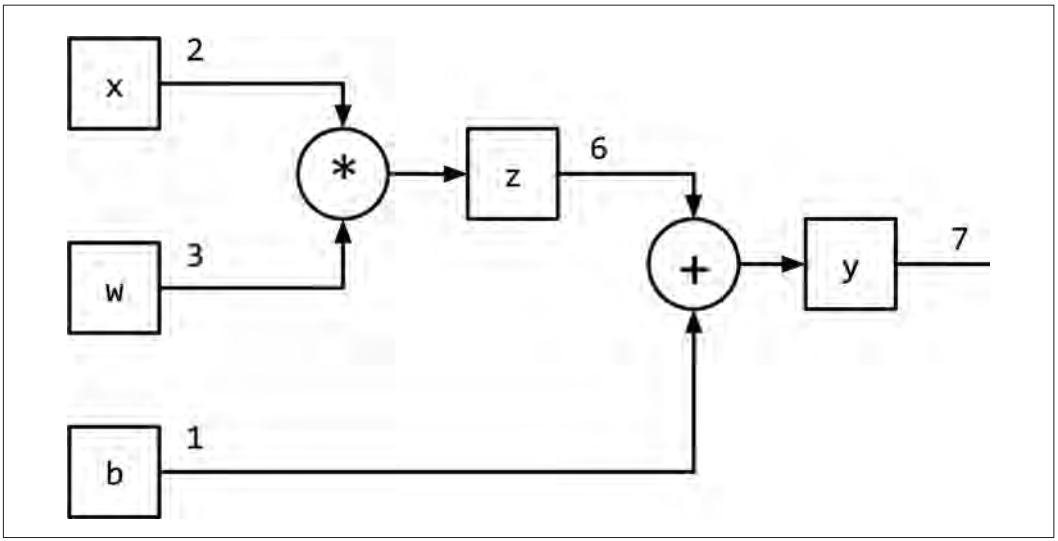

圖 1-6　使用計算圖表示 y = wx + b

PyTorch 基礎

本書大量使用 PyTorch 實作深度學習模型。PyTorch 是開源、社群推動的深度學習框架。與 Theano、Caffe、TensorFlow 不同，PyTorch 實作基於 tape 的自動微分（*http://bit.ly/2Jrntq1*），能讓我們動態的定義與執行計算圖。這對除錯特別有用，且也能用最小投入建構複雜的模型。

動態與靜態計算圖

Theano、Caffe、TensorFlow 等靜態框架要求計算圖必須先宣告、編譯後才能執行[6]。雖然這樣的實作效率非常好（適用於上線與行動應用），但對於研究與開發有時較麻煩。Chainer、DyNet、PyTorch 等現代框架實作動態計算圖以進行更有彈性、指令式的開發而無需在每次執行前都要編譯。動態計算圖特別適用於每個輸入有可能產生不同圖結構的 NLP 任務模型設計。

6　TensorFlow 的 1.7 版有個執行前不一定要將圖編譯過的“eager mode”，但 TensorFlow 還是以靜態圖為主。

PyTorch 是最佳化的張量操作函式庫，提供一系列的深度學習套件。此函式庫的核心是 *tensor*，它是保存一些多維資料的數學物件。零級（order）的 tensor 只是一個數字，或稱為**純量**（*scalar*），一級的 tensor（1st-order tensor）是個數字陣列，或稱為**向量**（*vector*）。同樣的，2nd-order tensor 是向量的陣列，或稱為**矩陣**（*matrix*）。所以 tensor 可歸納為純量的 *n* 維陣列，如圖 1-7 所示。

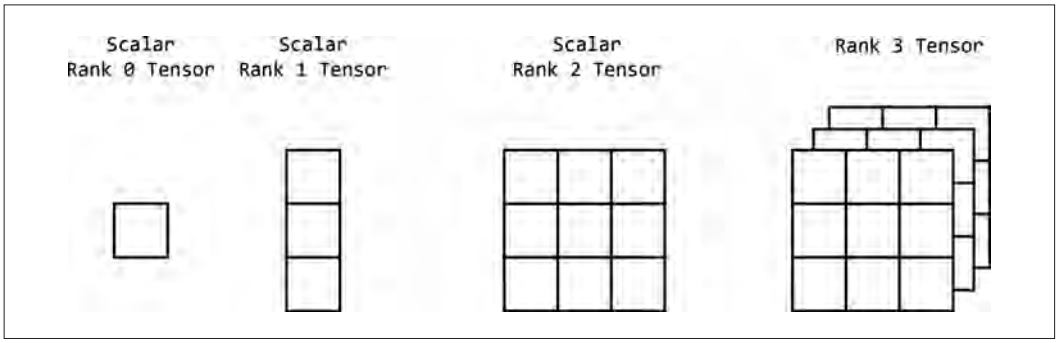

圖 1-7　歸納為多維陣列的 tensor

我們在這一節會逐步讓你熟悉 PyTorch 的各種操作，包括：

- 建立 tensor
- 操作 tensor
- 對 tensor 編索引、分割、連結
- 計算 tensor 梯度
- 使用 GPU 的 CUDA tensor

我們建議你如下一節所述在電腦上安裝 Python 3.5+ notebook 與 PyTorch 以便執行範例[7]，也建議您照著後面的練習做。

7　這一節的程式碼見本書的 GitHub 程式庫（*https://nlproc.info/PyTorchNLPBook/repo/*）下的 */chapters/chapter_1/PyTorch_Basics.ipynb*

安裝 PyTorch

安裝 PyTorch 的第一步是在 *pytorch.org*（*http://bit.ly/2DqhmCv*）選擇你的系統偏好。選擇作業系統和套件管理員（建議 Conda 或 Pip），然後選擇你使用的 Python 版本（建議 3.5+），這會產生安裝 PyTorch 的指令。例如 Conda 環境的安裝命令像這樣：

```
conda install pytorch torchvision -c pytorch
```

若有 CUDA 的圖形處理器（GPU），你應該選擇適用於 CUDA 的版本。更多資訊見 *pytorch.org*（*http://bit.ly/2DqhmCv*）的安裝指示。

建立 tensor

首先定義輔助函式 `describe(x)`，它說明 *x* 這個 tensor 的各種屬性，像是類型、維度、內容等：

Input[0]

```
def describe(x):
    print("Type: {}".format(x.type()))
    print("Shape/size: {}".format(x.shape))
    print("Values: \n{}".format(x))
```

PyTorch 可透過 `torch` 套件以多種方式建構 tensor。其中一種方式是如範例 1-3 所示，指定維度來初始化一個隨機 tensor。

範例 1-3　在 PyTorch 中以 torch.Tensor 建構一個 tensor

Input[0]

```
import torch
describe(torch.Tensor(2, 3))
```

Output[0]

```
Type: torch.FloatTensor
Shape/size: torch.Size([2, 3])
Values:
tensor([[ 3.2018e-05, 4.5747e-41, 2.5058e+25],
        [ 3.0813e-41, 4.4842e-44, 0.0000e+00]])
```

我們也可以如範例 1-4 所示，從區間 [0, 1) 的平均分佈、或標準常態分佈[8] 值隨機初始化的值建構一個 tensor。如第三章與第四章會看到的，以常態分佈值隨機初始化 tensor 很重要。

範例 1-4　建立隨機初始化的 tensor

Input[0]

```
import torch
describe(torch.rand(2, 3))  # 平均隨機
describe(torch.randn(2, 3)) # 隨機常態
```

Output[0]

```
Type: torch.FloatTensor
Shape/size: torch.Size([2, 3])
Values:
 tensor([[ 0.0242, 0.6630, 0.9787],
        [ 0.1037, 0.3920, 0.6084]])

Type: torch.FloatTensor
Shape/size: torch.Size([2, 3])
Values:
tensor([[-0.1330, -2.9222, -1.3649],
        [ 2.3648, 1.1561, 1.5042]])
```

我們也可以建立全部填入相同純量的 tensor。有內建函式可建立全零或一的 tensor，要填入指定值可使用 fill_() 方法。有底線（_）的 PyTorch 方法表示原址（in-place）操作；也就是如範例 1-5 所示，在原址修改內容而不會建立新物件。

範例 1-5　建立填值的 tensor

Input[0]

```
import torch
describe(torch.zeros(2, 3))
x = torch.ones(2, 3)
describe(x)
x.fill_(5)
describe(x)
```

8　標準常態分佈是 mean=0 與 variance=1 的常態分佈。

Output[0]

```
Type: torch.FloatTensor
Shape/size: torch.Size([2, 3])
Values:
tensor([[ 0., 0., 0.],
        [ 0., 0., 0.]])

Type: torch.FloatTensor
Shape/size: torch.Size([2, 3])
Values:
tensor([[ 1., 1., 1.],
        [ 1., 1., 1.]])

Type: torch.FloatTensor
Shape/size: torch.Size([2, 3])
Values:
tensor([[ 5., 5., 5.],
        [ 5., 5., 5.]])
```

範例 1-6 展示如何使用 Python 的 list 宣告一個 tensor。

範例 1-6　以 list 建構與初始化 tensor

Input[0]

```
x = torch.Tensor([[1, 2, 3],
                  [4, 5, 6]])
describe(x)
```

Output[0]

```
Type: torch.FloatTensor
Shape/size: torch.Size([2, 3])
Values:
tensor([[ 1., 2., 3.],
        [ 4., 5., 6.]])
```

值可如前例來自 list 或來自 NumPy 陣列。當然也可以從 PyTorch 的 tensor 建立 NumPy 的陣列。注意 tensor 的型別是 `DoubleTensor`，而不是預設的 `FloatTensor`（見下一節）。它對應於 NumPy 隨機矩陣的資料型別，`float64`，如範例 1-7 所示。

範例 *1-7*　從 *NumPy* 建構與初始化一個 *tensor*

Input[0]

```
import torch
import numpy as np
npy = np.random.rand(2, 3)
describe(torch.from_numpy(npy))
```

Output[0]

```
Type: torch.DoubleTensor
Shape/size: torch.Size([2, 3])
Values:
tensor([[ 0.8360, 0.8836, 0.0545],
        [ 0.6928, 0.2333, 0.7984]], dtype=torch.float64)
```

運用 NumPy 格式數值的函式庫時能夠在 NumPy 陣列與 PyTorch 的 tensor 間轉換很重要。

Tensor 型別與大小

每個 tensor 有相對應的型別與大小。使用 `torch.Tensor` 建構元的預設 tensor 型別是 `torch.FloatTensor`。但你可以在初始化時指定、或在事後使用型別轉換方法將 tensor 轉換為不同型別（`float`、`long`、`double` 等）。指定初始化型別有兩種方式：直接呼叫特定 tensor 型別的建構元，例如 `FloatTensor` 或 `LongTensor`，或如範例 1-8 所示使用 `torch.tensor()` 並提供 `dtype`。

範例 *1-8*　*Tensor* 屬性

Input[0]

```
x = torch.FloatTensor([[1, 2, 3],
                       [4, 5, 6]])
describe(x)
```

Output[0]

```
Type: torch.FloatTensor
Shape/size: torch.Size([2, 3])
Values:
tensor([[ 1., 2., 3.],
        [ 4., 5., 6.]])
```

Input[1]

```
x = x.long()
describe(x)
```

Output[1]

```
Type: torch.LongTensor
Shape/size: torch.Size([2, 3])
Values:
tensor([[ 1, 2, 3],
        [ 4, 5, 6]])
```

Input[2]

```
x = torch.tensor([[1, 2, 3],
                  [4, 5, 6]], dtype=torch.int64)
describe(x)
```

Output[2]

```
Type: torch.LongTensor
Shape/size: torch.Size([2, 3])
Values:
tensor([[ 1, 2, 3],
        [ 4, 5, 6]])
```

Input[3]

```
x = x.float()
describe(x)
```

Output[3]

```
Type: torch.FloatTensor
Shape/size: torch.Size([2, 3])
Values:
tensor([[ 1., 2., 3.],
        [ 4., 5., 6.]])
```

我們使用 tensor 物件的 shape 屬性與 size() 方法來存取維度值，這兩種存取值的方法相同。檢查 tensor 的形狀是對 PyTorch 程式碼除錯不可或缺的工具。