CHAPTER

1

自然語言處理簡介

自然語言處理（NLP, Natural Language Processing）是理解與處理當今世界大量非結構化資料的一種重要工具。最近許多 NLP 任務開始廣泛運用深度學習的做法，因為深度學習演算法在許多頗具挑戰性的任務中（例如圖片分類、語音辨識和實際文字生成等）都有相當傑出的表現。TensorFlow 則是目前最直觀、最有效率的深度學習框架之一。本書可以幫助深度學習開發人員，學會如何運用 NLP 和 TensorFlow 處理大量的資料。

本章會介紹一下 NLP，並回答「自然語言處理究竟是什麼？」這個問題。此外，也會介紹 NLP 一些最重要的用途。還會介紹 NLP 的傳統做法，以及最近的深度學習做法，其中包括所謂的**全連結神經網路**（FCNN, Fully-Connected Neural Network）。最後會介紹一下各章重點，以及使用到的一些技術工具。

「自然語言處理」究竟是什麼？

根據 IBM 的資料，2017 年的每一天都會生成 2.5 EB（1 EB= 1,000,000,000 GB）的資料，而且就在筆者撰寫本書的此時，這個數字仍在不斷增加。從這個角度來看，如果把這些資料平均分配給世界上所有人來處理，每個人每天大概就要處理 300 MB 的資料。所有這些資料其中很大一部分都是非結構化的文字和語音，因為人們每天都會創造出數以百萬計的電子郵件、社群媒體文字，或是電話的通話內容。

這些統計數字提供了一個很好的基礎，讓我們可以為 NLP 做出定義。簡而言之，NLP 的目標就是讓機器理解我們用口頭說出的、或是用文字寫出來的語言。如今 NLP 早已無所不在，已成為人類生活很重要的一部分。像 Google 助理、Cortana 和 Apple Siri 這樣的**虛擬助理**（VA），主要就是由 NLP 系統所構成。每當有人向

的面向，例如訊息檢索與知識表達等。其中各個面向都有可能讓 QA 系統的開發變得非常困難。

- **機器翻譯**（MT, Machine Translation）：MT 的任務就是把句子或一段文字從來源語言（source，例如德語）轉換成目標語言（target，例如英語）。這是一項非常具有挑戰性的任務，因為不同語言的構詞型態與結構很可能具有高度的差異，這也就表示，兩種語言之間並不是一對一的轉換。此外，兩種語言之間單詞與單詞的關係，有可能是一對多、一對一、多對一或多對多的關係。在 MT 相關文獻中，這就是所謂的「**單詞對齊**（word alignment）」**問題**。

最後，為了開發出可協助人們完成日常任務的系統（例如虛擬助理或聊天機器人），許多不同類型任務可能必須一起搭配執行。就像前一個例子，當使用者問道：「你能告訴我附近有哪家很棒的義大利餐廳嗎？」，處理過程就需要完成好幾種不同的 NLP 任務，例如語音轉文字的轉換、語義與情感分析、問題答覆與機器翻譯。

在**圖 1.1** 中，我們針對不同 NLP 任務提供了一個分層式分類架構，把這些任務分成好幾種不同的類型。首先任務可區分成兩大類：分析型任務（analysis，分析現有文字）與生成型任務（generation，生成全新文字）。然後再把分析型任務切分成三種不同的類別：句法（syntactic，和語言結構相關的任務）、語義（semantic，和意義相關的任務）、實務（pragmatic，比較難解決的一些開放性問題）：

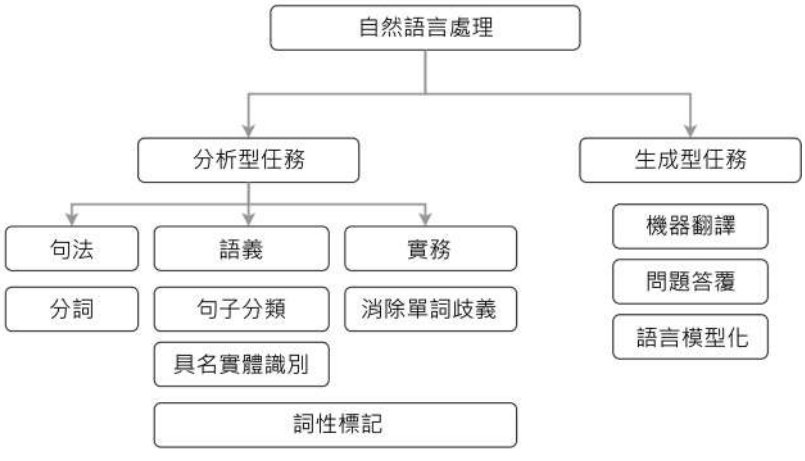

圖 1.1：NLP 常見任務的分層式分類結構

的權重和偏差，*sigma* 則代表 S 型激活函數。接著會在 FCNN 的頂部放一個分類器（classifier，例如 softmax 分類器），讓 FCNN 能夠善用隱藏層所學習到的特徵，對輸入進行分類。

分類器基本上也可算是 FCNN 的一部分，它跟其他隱藏層一樣，都具有一些權重值 W_s 和偏差值 b_s。如此一來，我們就可以計算出 FCNN 的**最終輸出** = $softmax(W_s * h + b_s)$。舉例來說，如果使用的是 softmax 分類器，分類層輸出的分數就會是歸一化的表達方式；具有最高 softmax 值的輸出節點，就會被視為輸入資料的相應標籤。透過這種方式，我們就可以把分類的誤差損失（loss），定義為預測輸出標籤與實際輸出標籤之間的差異。定義損失函數（loss function）其中的一個方式，就是採用均方損失（mean squared loss）來進行計算。就算你並不瞭解損失函數實際上有多複雜，也不需要太過擔心，後面的章節會有更多的說明。

接下來，我們就可以使用標準的隨機最佳化工具（例如 SGD 隨機梯度遞減演算法）來最佳化神經網路的參數 W、b、W_s 和 b_s，以設法降低所有輸入資料的分類誤差。**圖 1.5** 顯示的就是本段內容針對三層 FCNN 所說明的架構。在「**第 3 章：Word2vec｜學習單詞內嵌**」會逐步介紹如何把這種模型運用於 NLP 任務的相關細節。

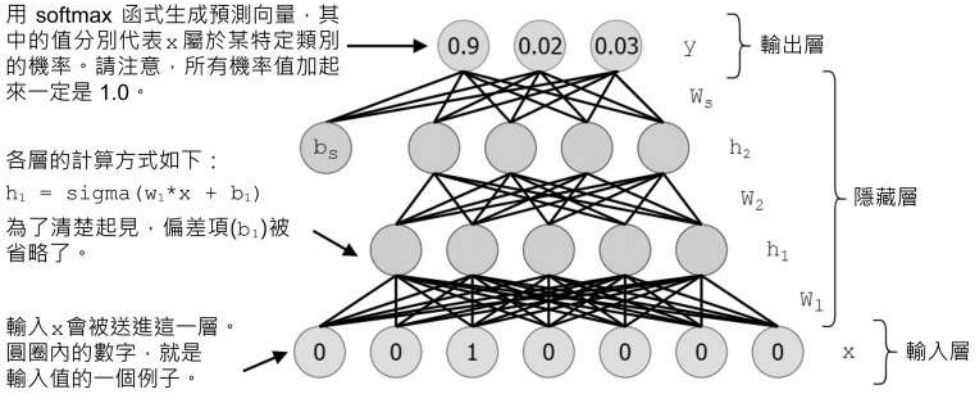

圖 1.5：全連結神經網路（FCNN）的一個範例

我們以一個範例來瞭解一下如何把神經網路用於情感分析任務。假設有一個資料集，其中輸入資料全都是針對某部電影表達正面或負面意見的一個句子，而相應標籤則代表該句子實際上是正面（1）或負面（0）的評價。另外，有一個測試資

接著，計算圖會被分解成一些子圖（subgraph），然後再進一步分解成更細的片段。雖然分解計算圖這個工作，在這個範例中似乎有點微不足道，但現實世界那些具有許多隱藏層的實際應用中，所分解出來的計算圖數量很有可能會呈指數式增長。此外，有能力把計算圖分解成許多個片段，這對於平行執行（例如有多個設備）來說十分重要。執行 graph 圖（或 subgraph 子圖，如果 graph 圖被切分成許多 subgraph 子圖的話）可被視為一個單一的**任務**，這樣的一個任務會被指派給單一的 TensorFlow 伺服器。

不過，實際上每個任務都會先被分解成兩個部分，再分別由兩個單一的 worker 工作程序來執行：

- 其中一個 worker 會使用參數當前的值，來執行 TensorFlow 操作（這個 worker 叫做「操作執行程序 [operation executor]」）
- 另一個 worker 負責儲存參數，並使用執行後所獲得的新值，對參數進行更新（這個 worker 叫做「參數伺服器 [parameter server]」）

TensorFlow 客戶端程式的一般工作流程，如**圖 2.2** 所示：

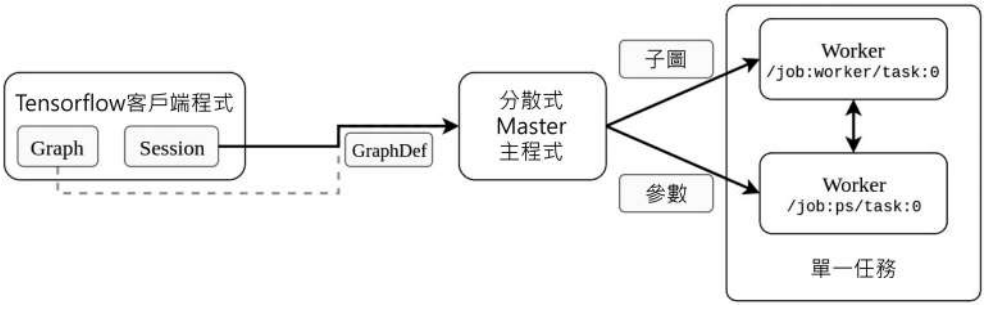

圖 2.2：TensorFlow 客戶端程式的一般執行流程

圖 2.3 說明的是 graph 圖分解的過程。除了把 graph 圖拆開來之外，TensorFlow 還會插入發送（send）和接收（receive）節點，以協助「參數伺服器」和「操作執行程序」之間進行溝通。你可以這樣理解：發送節點會在資料可取得時，把資料發送出去；接收節點則會持續監聽，並在相應發送節點發出資料時，把資料抓進來：

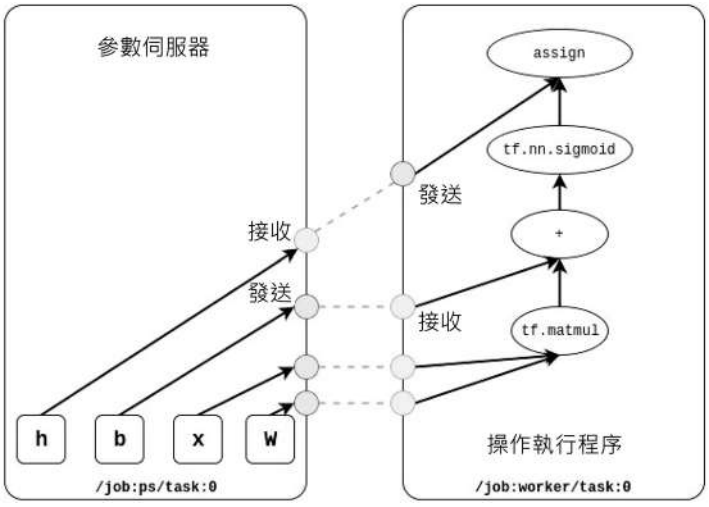

圖 2.3：TensorFlow graph 圖的分解過程

最後，一旦完成計算，session 就會把更新後的資料從參數伺服器取回到 client 客戶端程式。TensorFlow 的整體架構如圖 2.4 所示。這裡的說明是根據 TensorFlow 的官方文件而來，這些文件可以在 `https://www.tensorflow.org/extend/architecture` 找到。

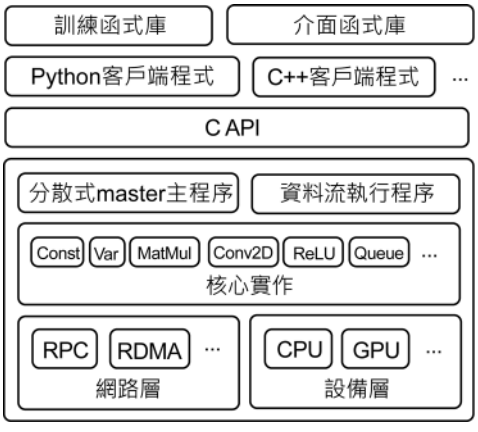

圖 2.4：TensorFlow 整個框架的架構（**https://www.tensorflow.org/extend/architecture**）

神經網路所使用的非線性激活函數

神經網路能在許多任務中取得良好的表現，非線性激活函數就是其中的一個原因。神經網路每一層輸出之後，通常都會進行一個非線性的激活函數轉換（也就是會有一個激活層，但最後一層除外）。

非線性轉換可協助神經網路學習資料中所存在的各種非線性關係。這對於現實世界各種複雜的問題來說非常有用，因為除了線性的關係以外，一般資料通常都含有比較複雜的非線性關係。如果在各層之間並未採用非線性激活函數，那麼深度神經網路也就只不過是許多堆疊起來的線性層而已。但許多線性層所堆疊起來的結構，基本上都可以壓縮成單一個比較大的線性層。換句話說，如果不是採用非線性激活函數，創建出許多層的神經網路就沒有意義了。

我們可以透過一個範例，觀察一下非線性激活函數的重要性。首先，回想一下在「**S 型範例**」中所看到的神經網路相關計算。如果暫時忽略掉 b，就會變成下面這樣：

```
h = sigmoid(W*x)
```

假設有一個三層的神經網路（W1、W2 和 W3 分別代表各層的權重），其中每一層都用前一層的計算結果來繼續進行計算；我們可以把全部的計算總結如下：

```
h = sigmoid(W3*sigmoid(W2*sigmoid(W1*x)))
```

但如果拿掉其中的非線性激活函數（也就是 sigmoid 函數），就會得到：

```
h = (W3 * (W2 * (W1 *x))) = (W3*W2*W1)*x
```

因此，在沒有非線性激活函數的情況下，三層的結構就可以降減為單一線性層。

接下來，我們會列出神經網路中最常用的兩種非線性激活函數，以及在 TensorFlow 中實作的方法：

```
# x 的 S 型激活函數，相當於用 1 / (1 + exp(-x)) 來進行計算
tf.nn.sigmoid(x,name=None)
# x 的 ReLU 激活函數，相當於用 max(0,x) 來進行計算
tf.nn.relu(x, name=None)
```

```
# 執行最佳化處理程序
loss, _ = session.run([tf_loss,tf_loss_minimize],feed_dict={
          tf_inputs: train_inputs[step*batch_size: (step+1)*batch_size,:],
          tf_labels: labels_one_hot})
train_loss.append(loss)
# 用來計算單一階段內損失的平均值

test_accuracy = []
# 測試階段
for step in range(test_inputs.shape[0]//batch_size):
  test_predictions = session.run(tf_predictions,feed_dict={tf_inputs:
                     test_inputs[step*batch_size: (step+1)*batch_size,:]})
  batch_test_accuracy = accuracy(test_predictions,test_labels[step*batch_size:
                        (step+1)*batch_size])
  test_accuracy.append(batch_test_accuracy)

print('Average train loss for the %d epoch: %.3f\n'%
      (epoch+1,np.mean(train_loss)))
print('\tAverage test accuracy for the %d epoch: %.2f\n'%
      (epoch+1,np.mean(test_accuracy)*100.0))
```

在這段程式碼中，用來計算準確度的 `accuracy(test_predictions, test_labels)` 是一個函式，它會把一些預測結果與相應標籤當成輸入，然後計算出準確度（有多少預測結果與實際標籤是相符的）。這個函式的定義也包含在練習檔案中。

如果一切順利，應該可以看到類似圖 2.10 所呈現的情況。經過 50 個階段之後，測試準確度應該就可以達到 98%左右：

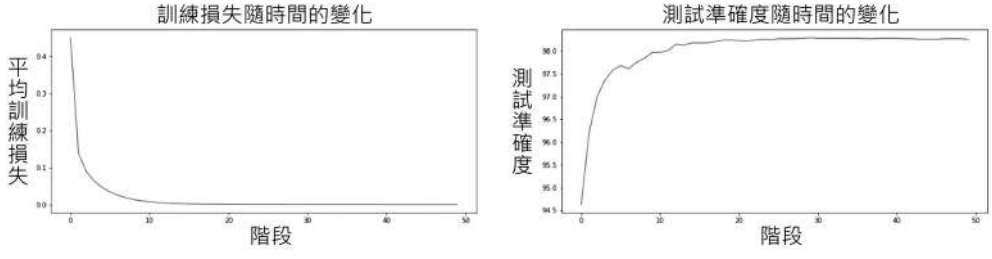

圖 2.10：MNIST 數字分類任務的訓練損失與測試準確度

舉例來說，我們可以選擇每一小塊裡的最大值或平均值，來做為轉換的方式。圖 5.2 中，說明了池化（pooling）操作如何讓 CNN 產生出平移不變的效果：

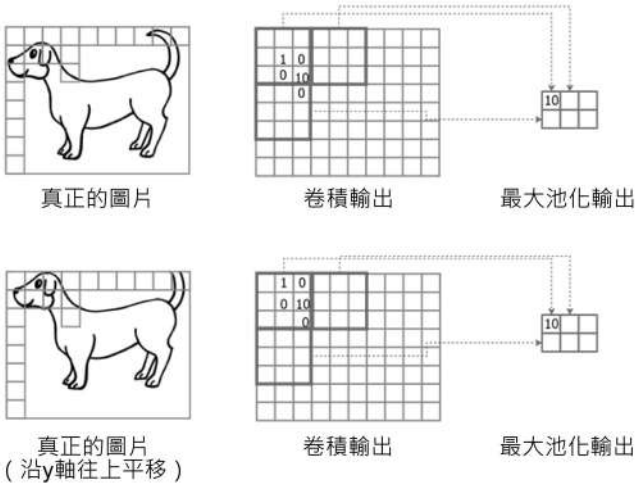

圖 5.2：池化操作如何讓資料產生出平移不變的效果

在這裡，我們有一張原始的圖片，還有另一張沿著 y 軸略微平移的圖片。觀察一下這兩張圖片的卷積輸出結果，就可以看到 10 這個值出現在卷積輸出中稍微不同的位置。但由於使用了最大池化操作（取每個小區域內的最大值），因此最後還是可以獲得相同的輸出結果。稍後我們還會再詳細討論這些操作。

最後，輸出會被送進一組全連結層，然後再把輸出轉送到最終的分類／迴歸層（例如句子／圖片分類）。全連結層的權重占了 CNN 所有權重其中很大的一部分，因為卷積層的權重相對而言少了很多。但目前也發現，採用全連結層的 CNN 比沒有採用全連結層的模型表現好很多。這可能是因為卷積層的尺寸比較小，學習到的多半是比較局部的特徵，而全連結層則可提供關於這些局部特徵如何連結在一起的整體概念，藉以生成預期中的最終輸出。圖 5.3 顯示的就是一個可用來針對圖片進行分類的典型 CNN：

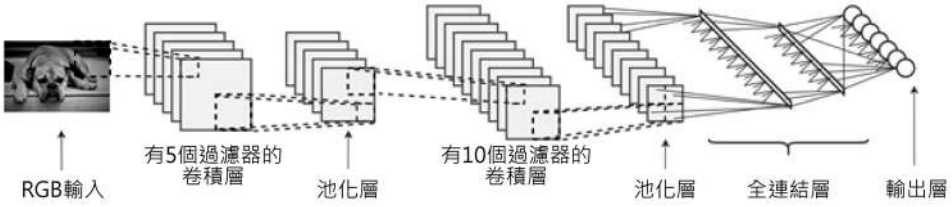

圖 5.3：一個典型的 CNN 架構

從圖中可明顯看出，CNN 透過其設計，在學習期間保留了輸入的空間結構。換句話說，對於二維輸入來說，CNN 可以讓大多數層具有二維性，直到最後比較靠近輸出層時才使用全連結層。CNN 可保留住空間結構的天性，使它可以進一步利用輸入中有價值的空間訊息，並以較少的參數建立對輸入的理解。空間訊息的價值，可參見圖 5.4 的說明：

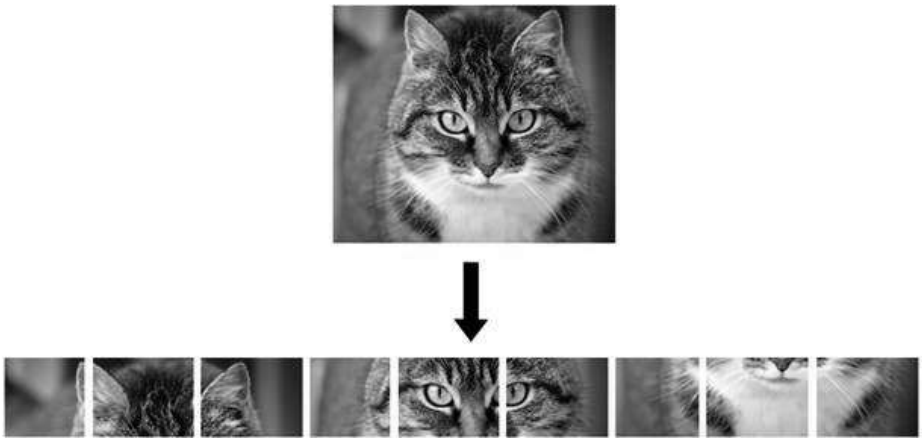

圖 5.4：如果把圖片展開成一維向量，往往會丟失掉一些重要的空間相關訊息

正如你所看到的，如果貓的二維圖片被展平成一維的向量，耳朵就不再靠近眼睛，鼻子也遠離了眼睛。這也就表示，在展平的過程中，往往會破壞一些有用的空間相關訊息。

CNN 卷積神經網路的力量

CNN 是一種非常具有通用性的模型，它在許多類型的任務中，都具有非常卓越的表現。這種多才多藝的特性，主要是源自於 CNN 可同時提取特徵並進行學習，從而導致更高的效率與通用性。我們就來討論一些 CNN 實際應用的例子。

想要解決梯度消失的問題，並沒有那麼簡單。實際上並沒有什麼簡單的方法可重新調整梯度的尺度，讓它能夠隨時間正確傳播。只有少數幾種技術，可以在某種程度上解決梯度消失的問題，例如仔細選擇權重的初始值（例如採用 Xavier 初始化方法），或者是採用以動量為基礎的最佳化方法（也就是除了更新當前梯度值之外，再額外添加一個項，用來累計過去所有的梯度值，這個項可稱為**速度項**）。不過，後來也出現好幾種從根本上解決這個問題的做法，例如採用不同的 RNN 結構，稍後在「**第 7 章：LSTM 長短期記憶網路**」就會看到。

另一方面，假設把 w_3 的初始值設成非常大的值（比如說 1000.00）。到了 $n = 100$ 時間步驟時，梯度就會變成非常巨大（大約接近 10 的 300 次方）。這一定會造成數值不穩定的問題，最後在 Python 中只會得到諸如 `Inf` 或 `NaN`（即非數值）之類的結果。這就是所謂「**梯度爆炸**（exploding gradient）」的問題。

如果問題的損失曲面（loss surface）太過複雜，也有可能會造成梯度爆炸的問題。由於深度神經網路經常存在大量的參數（權重），如果輸入維度較多，經常可以看到非常複雜的非凸形（nonconvex）損失曲面。**圖 6.7** 就顯示了一個 RNN 的損失曲面，其中可以看到一個曲率非常高的牆壁。如果最佳化方法碰到這樣的牆壁，梯度可能就會爆炸或衝過頭，如圖中的實線所示。這有可能會導致損失最小化的效果極差，或是出現數值不穩定的情況。

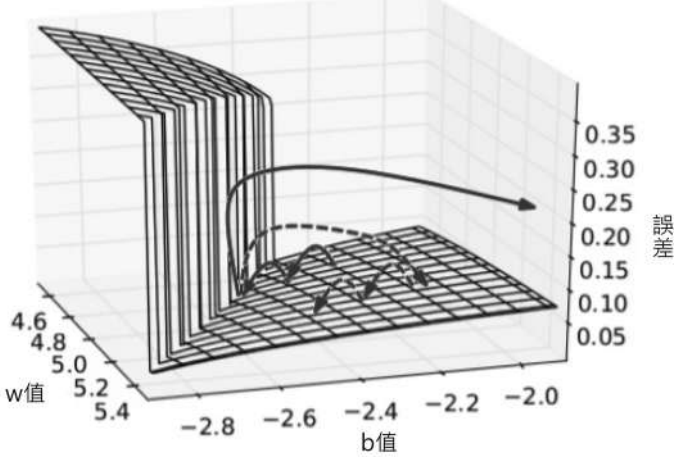

圖 6.7：梯度爆炸現象

資料來源：本圖取自 Pascanu、Mikolov 和 Bengio 的論文「On the difficulty of training recurrent neural networks（關於訓練遞迴神經網路的困難）」

在我們的範例中，我們要輸出的是下面這個句子：

They named it Luna.（**他們把它取名為魯那。**）

以這個句子來說，我們已經不再需要倒數第二個神經元的資訊，因為它包含的是小狗怎麼吠的訊息，但最後這個句子關注的是小狗的名字。因此，我們可以在最後一個句子的預測中，忽略掉這個神經元（內含 *bark* → *loud* 這個關係）。這正是 o_t 所要做的事情；在計算 LSTM 單元的最終輸出時，它會忽略不必要的記憶，只從單元狀態中取出有用的記憶。現在，我們終於可以在**圖 7.10** 中，展示 LSTM 單元的完整結構了：

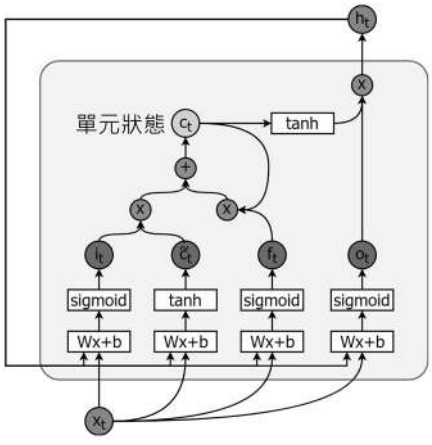

圖 7.10：LSTM 的完整結構

在這裡總結一下 LSTM 單元內所有操作相關的方程式：

$$i_t = \sigma\left(W_{ix}x_t + W_{ih}h_{t-1} + b_i\right)$$

$$f_t = \sigma\left(W_{fx}x_t + W_{fh}h_{t-1} + b_f\right)$$

$$\tilde{c}_t = \tanh\left(W_{cx}x_t + W_{ch}h_{t-1} + b_c\right)$$

$$c_t = f_t c_{t-1} + i_t \tilde{c}_t$$

$$o_t = \sigma\left(W_{ox}x_t + W_{oh}h_{t-1} + b_o\right)$$

$$h_t = o_t \tanh\left(c_t\right)$$

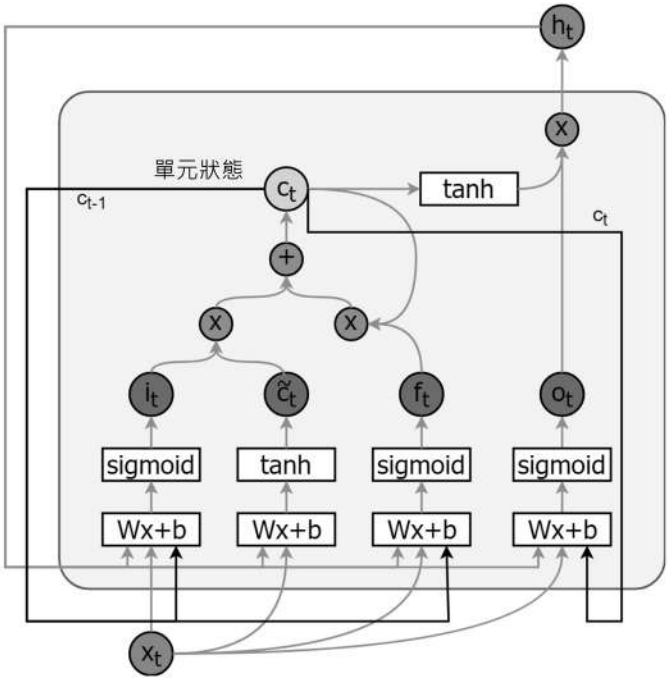

圖 7.17：帶有窺孔連結的 LSTM（窺孔連結以黑色表示，其他連結則以灰色表示）

閘控遞迴單元

GRU（Gated Recurrent Units，**閘控遞迴單元**）可視為標準 LSTM 架構的一種簡化。正如我們所見，LSTM 有三種不同的閘和兩個不同的狀態。就算狀態的尺寸不大，光是這一項也需要用到大量的參數。因此，科學家們研究了好幾種減少參數數量的方法。GRU 就是其中一種這樣的做法。

與 LSTM 相比，GRU 有好幾個主要的差異。

首先，GRU 把兩個狀態（單元狀態和最終隱藏狀態）組合成單一個隱藏狀態 h_t。如此一來，由於不再具有兩種不同的狀態，這個簡單修改的副作用就是可以去除輸出閘。還記得嗎，輸出閘只是用來決定要把多少單元狀態讀入最終隱藏狀態。這個操作大大減少了單元內的參數數量。