對本書的讚譽

《生成深度學習》是一本關於生成建模中各種深度學習套件的實用入門書。如果你喜歡寫點程式並希望將深度學習應用在工作中的創意從業者，這本書保證適合你。

——*David Ha*，策略部門主管，*Stability AI*

這本書超級棒，深入介紹了各種最新生成式深度學習背後的所有主要技術。你會找到許多直觀解釋和巧妙類比——還有易讀的教學範例程式來支援。這是對 *AI* 中這項最迷人領域的一趟令人興奮的探索之旅！

——*François Chollet*，*Keras* 創始人

David Foster 能清晰解釋各種複雜概念，再搭配直觀的視覺化圖像、範例程式與練習題。本書是給學生和從業者的絕佳資源！

——*Suzana Ilić*，負責任 *AI* 計畫經理，*Microsoft Azure OpenAI*

在各種 *AI* 技術中，生成式 *AI* 是下一個會對世界產生巨大影響的革命性階段。本書完整介紹了這一領域，以及其無窮潛力和潛在風險。

——*Connor Leahy*，*Conjecture CEO* 和 *EleutherAI* 聯合創辦人

預測世界代表在所有模態上了解這個世界。就這個意義而言，生成式 *AI* 正在解決智慧的最核心問題。

——*Jonas Andrulis*，*Aleph Alpha* 創辦人與 *CEO*

生成式 *AI* 正在重塑無數產業並催生新一代的創新工具。本書是進入生成建模並運用這項革命性技術的完美指南。

——*Ed Newton-Rex*，*Stability AI* 的聲學部門副總和作曲家

David 教會了我關於機器學習的所有知識，他擅長解釋各種底層概念。《生成深度學習》是我研究生成式 *AI* 的首選資源，一直與我最喜愛的技術書籍一起放在我工作桌旁的書架上。

——*Zack Thoutt*，*AutoSalesVelocity* 產品總監

生成式 *AI* 很可能對社會產生深遠的影響。這本書以易於理解且不省略任何技術細節的方式完整介紹了這個領域。

——*Raza Habib*，*Humanloop* 聯合創辦人

當有人問我如何入門生成式 *AI* 的時候，我都會推薦 *David* 的書。第二版很棒，涵蓋了像是擴散模型和 *Transformer* 等最厲害的模型。對於任何想要發揮運算創意的人來說，本書絕對必備啦！

——*Tristan Behrens* 博士，*AI* 專家和 *KI Salon Heilbronn* 的 *AI* 駐場音樂藝術家

本書的技術知識非常豐富，每當有任何關於生成式 *AI* 的想法時，它都是我首選的參考文獻。這本書應該放在每位資料科學家的書架上。

——*Martin Musiol*，*generativeAI.net* 創辦人

本書完整且詳細介紹了生成模型的分類。我發現這本書最棒的地方之一在於它既涵蓋了各種背後的重要理論，又透過實務範例來加強讀者對模型的理解。我必須說，本書關於 *GAN* 的章節是我讀過的最佳解釋之一，並針對模型微調提供了相當直觀的方法。本書收錄了各種形式的生成式 *AI*，包括文字、圖像和音樂。對於所有正準備入門生成式 *AI* 的人來說是非常棒的資源。

——*Aishwarya Srinivasan*，*Google Cloud* 資料科學家

第一篇

認識生成深度學習

本書第一篇會概略介紹生成建模與深度學習——也就是想要入門生成深度學習一定要理解的兩個重點領域！

第 1 章會定義何謂生成建模，並用一個簡單範例來理解適用於所有生成模型的一些關鍵概念。另外也會約略提到生成模型家族的主要分類，這會在本書第二篇深入探討。

第 2 章簡介了建置更複雜生成模型所需的深度學習工具與技術。確切而言，我們會使用 Keras 來建置第一個深度神經網路範例——多層感知器（MLP）。接著延續這個範例來加入卷積層與其他改良做法，並觀察其效能差異。

讀完本書第一篇之後，你會對本書後續所介紹的各項技術之關鍵核心概念有相當扎實的理解。

什麼是生成建模？

生成建模的廣義定義如下：

> 生成建模是機器學習的一個分支，其中所訓練出的模型必須能夠產生與給定資料集相似的全新資料。

這在實務上代表著什麼呢？假設我們有一個包含許多馬匹照片的資料集。我們可以根據這個資料集來訓練一個生成模型，要求它去擷取馬匹圖像像素之間的複雜關係規則。然後，我們就能對這個模型抽樣來產生全新又逼真、而且不存在於原始資料集中的馬匹圖像。這個過程請參考圖 1-1。

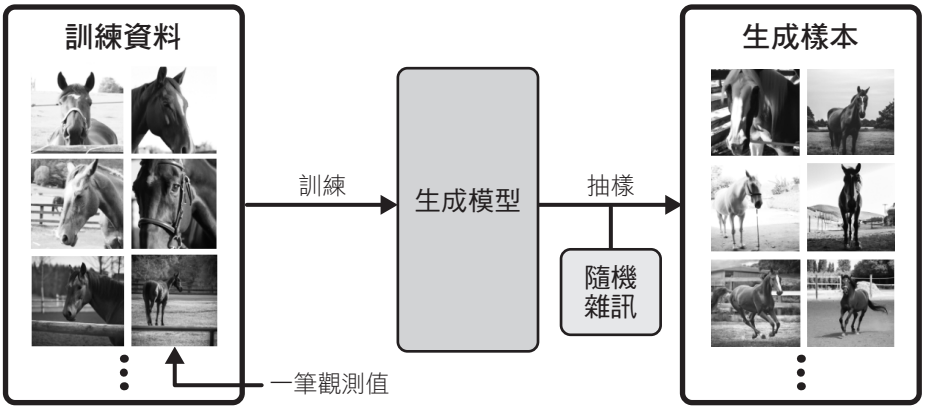

圖 1-1　訓練生成模型來生成逼真的馬匹照片

為了建置一個生成模型，我們需要一個資料集，其中包含了許多想要生成的實體範例。這稱為訓練資料，而其中的一個資料點則稱為觀測值（*observation*）。

每個觀測值由許多特徵（*feature*）組成。對於圖像生成問題，這些特徵通常是個別的像素值；但如果是文字生成問題，特徵可以是一個單詞或一些字母。我們的目標是建立一個可以生成全新特徵的模型，這些特徵就好像是根據與原始資料相同的規則來產生的。觀念上來說，要生成這樣的圖像可說是不可能的任務，因為考量到個別像素值的指派方式這必然是天文數字，但要能夠構成我們所要模擬實體的方式卻又少得驚人。

生成模型必須是機率性（*probabilistic*），而不是決定性（*deterministic*），因為我們希望能夠讓輸出具備多種不同的變化，而非每次都得到相同的結果。如果我們的模型只是某個固定的算式，例如把訓練資料集的所有像素值取平均，這就談不上是生成了。生成模型必定會包含某種隨機元素來影響模型所生成的個別樣本。

換言之，我們可想像有個未知的機率分配來決定為什麼有些圖像比較容易在訓練資料集中找到，而其他則無法。我們的目標是建置一個能夠盡量模仿這個分配的模型，並從中抽樣來生成全新且獨一無二的觀測值，看起來如同它們原本就屬於原始訓練資料集一樣。

生成建模與判別建模

為了真正理解生成建模的目標和其重要性，將其與對應的判別建模（*discriminative modeling*）進行比較是很有用的。如果你已學過機器學習，之前所遇到的大多數問題可能都是判別性的。為了理解這種差異，來看看以下範例吧。

假設我們有一個繪畫資料集，其中一些是梵谷的作品，其他則是其他藝術家的作品。只要資料量足夠，我們就能訓練一個判別模型來預測某張畫的作者是否就是梵谷。這個模型將有可能學會某些顏色、形狀和紋理，足以指出某張畫作是否出自這位荷蘭大師之手，而只要遇到具備這些特徵的畫作時，模型也會相應地調高其預測結果的權重。圖 1-2 是判別建模的處理流程——請觀察它與圖 1-1 中的生成模型流程的不同之處。

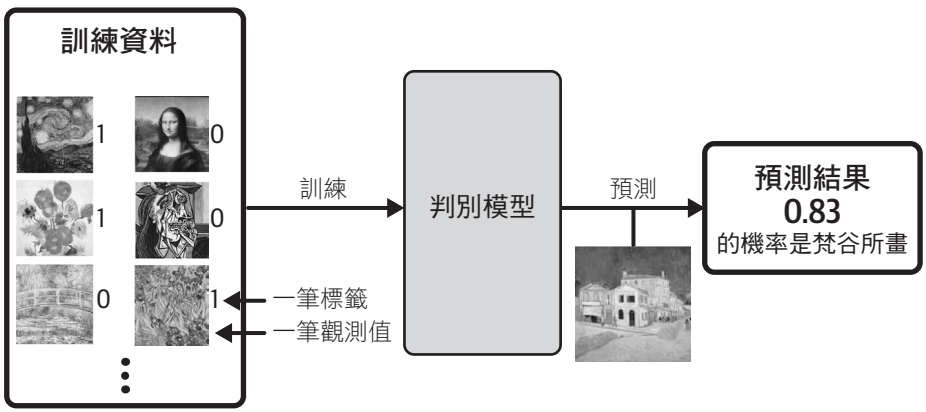

圖 1-2　訓練判別模型來預測指定圖像是否由梵谷所畫

在進行判別建模時，訓練資料中的每個觀測值都有一個標籤（*label*）。對於像上述藝術家鑑別這樣的二元分類問題來說，梵谷的畫作會被標註為 1，而非梵谷的畫作則被標註為 0。這樣一來，模型就能接續學習如何區分這兩組資料，並輸出一個機率值來表示它認為某筆新觀測值為標籤 1 的機率——也就是這幅畫是出自梵谷的機率。

相較之下，生成建模不需要已標註好的資料集，因為它的重點在於如何生成全新的圖像，而非試著去預測指定圖像的標籤。

接著用數學符號來正式定義這些建模類型：

判別建模會去估計 $p(y \mid \mathbf{x})$。

也就是說，判別建模的目標是找出給定某個觀測值 $\mathbf{x}$ 的情況下，標籤為 y 的機率。

生成建模會去估計 $p(\mathbf{x})$。

也就是說，生成建模的目標是找出某個觀測值 $\mathbf{x}$ 的機率。對這個分配中進行抽樣就能生成新的觀測值。

條件生成模型

請注意，我們也可以建立一個生成模型來建模條件機率 $p(\mathbf{x} \mid y)$——也就是觀察到某個觀測值 $\mathbf{x}$ 的標籤為 y 的機率。

例如，如果資料集中包含了不同類型的水果，就能要求生成模型專門去生成一張蘋果的圖像。

需要注意的一點是，即使我們能夠建立一個完美的判別模型來辨識梵谷畫作，它還是不知道如何產生一幅如同出自梵谷之手的畫作。它只能輸出針對既有圖像的機率，因為它就是被訓練來做這件事的。因此還需要訓練一個生成模型，再對這個模型來抽樣就很有機會生成看起來屬於原始訓練資料集的圖像。

生成建模的崛起

時至今日，判別建模一直是機器學習大幅進展的推動主力。這是因為對於任何鑑別性問題來說，對應的生成建模問題通常更難解決。例如，要訓練一個模型來預測一幅畫的作者是不是梵谷，要比從頭開始訓練一個能生成梵谷風格畫作的模型來得簡單太多了。

同樣地，要訓練一個模型來預測一段文字是否是查爾斯·狄更斯所寫，要比建立一個能夠生成一段狄更斯風格文章的模型來得更加容易。直到最近，大多數生成性挑戰都是無法實現的，許多人甚至懷疑它們是否真的能被解決。創造力一直以來都被視為是完全專屬於我們人類的能力，AI 完全無法比肩。

然而，隨著機器學習技術的成熟，這種假設已逐漸崩解。過去十年來，這個領域中許多最有趣的進展都是關於如何將機器學習應用於生成建模任務的各種新方法。例如，圖 1-3 是人臉圖像生成從 2014 年以來的驚人進展。

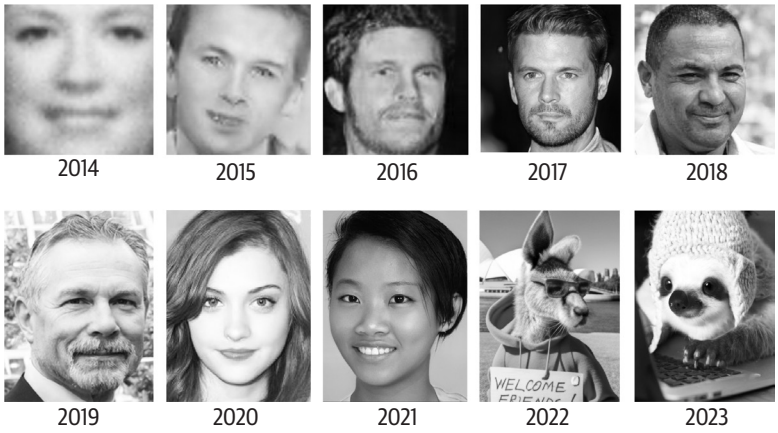

圖 1-3　生成建模應用於人臉生成技術在過去十年中已有大幅改進（Brundage et al., 2018）[1]

除了容易處理之外，判別建模向來也比生成建模更被廣泛應用於各產業的實務問題上。例如，醫生有可能借助一個能夠預測指定視網膜圖像是否呈現出青光眼跡象的模型，但如果換成能夠生成全新眼球後側圖像的模型就不一定有這個必要了。

然而，就算是這種情況也開始變化了，針對特定業務問題提供生成服務的公司已如雨後春筍般激增。例如，現在已可透過 API 來根據指定主題生成原創部落格文章、以任何你想得到的設定來生成各種產品圖像，或者撰寫符合品牌和目標訊息的社交媒體內容和廣告文案。生成式 AI 在遊戲設計和電影製作等產業中也已有了明確的積極應用，其中已訓練了各種能為產業加值的影片與音樂生成模型。

第五章

自迴歸模型

本章目標

本章學習內容如下：

- 了解自迴歸模型為何非常適合用於生成像是文字這樣的序列資料。
- 學習如何處理和分詞文字資料。
- 了解遞歸神經網路（RNN）的架構設計。
- 使用 Keras 從頭建置並訓練長短期記憶網路（LSTM）。
- 使用 LSTM 生成全新文字。
- 認識其他的 RNN 變體，包括門控遞歸單元（GRU）和雙向循環單元。
- 了解如何將圖像資料以像素序列來處理。
- 了解像素卷積神經網路（PixelCNN）的架構設計。
- 使用 Keras 從頭開始建置 PixelCNN。
- 使用 PixelCNN 來生成圖像。

到目前為止，我們已經探討了兩種與潛在變數有關的生成模型 —— 變分自動編碼器（VAE）和生成對抗網路（GAN）。在這兩種情況下，都對資料分布引入了一個易於從中抽樣的新變數，好讓模型學習如何將該變數解碼回到原始領域中。

現在把注意力轉到自迴歸模型 —— 這是將生成建模問題簡化為序列流程來處理的一系列模型。自迴歸模型會根據序列中的先前數值來條件化預測，而非使用潛在的隨機變數。

當 RNN 首次問世時，其遞歸層非常簡單，只由一個 tanh 運算子組成，用於確保在時步之間傳遞的資訊會被縮放到 –1 和 1 之間。然而，這種方法已被證明存在梯度消失問題，因此對於較長序列的資料表現不佳。

LSTM 單元首次出現在 1997 年 Sepp Hochreiter 和 Jürgen Schmidhuber 所提出的論文中[1]。作者在這篇論文中描述了 LSTM 不會碰到一般 RNN 所苦惱的梯度消失問題，並且可用長達數百個時步的序列來訓練。從那時開始，LSTM 網路的架構已被不斷改良，並且像是門控循環單元等變體（本章後續會介紹），現也已被廣泛應用，且已被作為 Keras 的現成層來提供。

LSTM 已經被應用於與序列資料相關的各種問題上，包括時間序列預測、情感分析和聲音分類。本章將使用 LSTM 來處理文字生成這項挑戰。

執行本範例的程式碼

本範例的 Jupyter notebook 程式碼請由本書 GitHub 取得：
notebooks/05_autoregressive/01_lstm/lstm.ipynb

食譜資料集

我們要使用來自 Kaggle 的 Epicurious Recipes 資料集（*https://oreil.ly/laNUt*）。這是一個包含了超過 20,000 份食譜的資料集，還附帶了營養資訊和成分清單等元資料。

請執行本書 GitHub 中的 Kaggle 資料集下載腳本來下載這個資料集，如範例 5-1。這會把食譜和相關元資料儲存到本地端的 */data* 資料夾中。

範例 *5-1*　下載 *Epicurious Recipe* 資料集

```
bash scripts/download_kaggle_data.sh hugodarwood epirecipes
```

範例 5-2 說明如何載入並篩選資料，只留下具備標題與描述的食譜。範例 5-3 則是一段處理完成後的食譜文字。

範例 *5-2*　載入資料

```
with open('/app/data/epirecipes/full_format_recipes.json') as json_data:
    recipe_data = json.load(json_data)
```

```
filtered_data = [
    'Recipe for ' + x['title']+ ' | ' + ' '.join(x['directions'])
    for x in recipe_data
    if 'title' in x
    and x['title'] is not None
    and 'directions' in x
    and x['directions'] is not None
]
```

範例 *5-3*　來自食譜資料集的一段文字

```
Recipe for Ham Persillade with Mustard Potato Salad and Mashed Peas  | Chop enough
parsley leaves to measure 1 tablespoon; reserve. Chop remaining leaves and stems
and simmer with broth and garlic in a small saucepan, covered, 5 minutes.
Meanwhile, sprinkle gelatin over water in a medium bowl and let soften 1 minute.
Strain broth through a fine-mesh sieve into bowl with gelatin and stir to dissolve.
Season with salt and pepper. Set bowl in an ice bath and cool to room temperature,
stirring. Toss ham with reserved parsley and divide among jars. Pour gelatin on top
and chill until set, at least 1 hour. Whisk together mayonnaise, mustard, vinegar,
1/4 teaspoon salt, and 1/4 teaspoon pepper in a large bowl. Stir in celery,
cornichons, and potatoes. Pulse peas with marjoram, oil, 1/2 teaspoon pepper, and
1/4 teaspoon salt in a food processor to a coarse mash. Layer peas, then potato
salad, over ham.
```

在說明如何使用 Keras 來建置 LSTM 網路之前，首先得先岔出來簡單談一下文字資料的結構，以及它與本書中到目前為止所用的圖像資料到底有何不同。

處理文字資料

文字資料與圖像資料之間有一些關鍵性差異，代表許多適用於圖像資料的方法換成文字資料之後就沒那麼好用了。說明如下：

- 文字資料由多個離散區塊所組成（可能是字元或單詞），而圖像中的像素則是連續顏色光譜中的各個點。我們很容易就能把一個綠色像素變得更藍一點，但是要讓貓這個詞變得更接近狗，這就不太知道怎麼做了。這代表圖像資料要進行反向傳播是相當簡單的，因為我們可以計算出損失函數之於個別像素的梯度，從而確定像素顏色要朝著哪個方向來改變才能讓損失最小化。但如果是離散的文字資料，就無法直接以同樣的方式來進行反向傳播了，所以需要找出解決這個問題的方法才行。

- 文字資料具有時間維度但沒有空間維度，而圖像資料則是具有兩個空間維度但沒有時間維度。在文字資料中，詞的順序非常重要，把詞倒著講是沒有意義的，但圖像內容通常在翻轉之後也不會受影響。此外，模型通常需要捕捉各單詞之間的長期依賴性，例如回答問題或者延續某個代詞的上下文。但圖像資料中的所有像素是可以同時處理的。
- 文字資料對個別單位（單詞或字元）的微小變化相當敏感。相比之下，圖像資料對於單一像素的變化就沒那麼敏感——就算某些像素被改變了，一幅房屋的圖片還是可被辨識為房屋——但如果是文字資料，哪怕只是改了幾個詞都可能讓整段文句的意思大大不同，甚至變得毫無意義。這使得想要訓練出一個能夠生成流暢文字的模型非常困難，因為每個單詞對於段落的整體意義來說都非常重要。
- 文字資料有其規則式的語法結構，但圖像資料則沒有一套用於指定像素值的規則。例如，不論在任何情況下，“The cat sat on the having”這句話在語法上都是沒有意義的。再者，要對語義規則建模是相當困難的；例如，“I am in the beach”這句話並不合理，雖然它在語法上並沒有錯。

基於文字的生成深度學習之相關進展

直到不久之前，大多數最複雜的生成式深度學習模型都專注於圖像資料，因為上述提到的許多挑戰即便是最厲害的技術都無法解決。然而，多虧了 Transformer 模型架構，文字生成深度學習領域在過去五年已有驚人的進展，後續在第 9 章會深入介紹。

考慮到這些因素，現在來看看要採取哪些步驟才能把文字資料轉換為適合訓練 LSTM 網路的正確形狀吧。

標記化

第一步是對文字進行清理和分詞。分詞（*tokenization*）是指將文字分割成個別單元，例如單詞或字元的過程。

如何分詞將取決於你希望這個文字生成模型能做到怎樣的目標。使用單詞和字元標記各有其優缺點，而你的選擇將會影響在建模之前應該如何清理文字與模型輸出。

如果使用單詞標記：

- 所有文字都可轉換為小寫來確保句子開頭的大寫單詞與句子中間出現的相同單詞是以相同方式來標記。然而這在某些情況下可能不是最好的做法；例如，某些專有名詞（例如姓名或地點）可能要保持大寫使得它們能被獨立標記。
- 文字詞彙（*vocabulary*，訓練集中的不同單詞集合）可能會變得非常大，其中某些單詞的出現狀況可能非常稀疏，甚至只出現一次。比較好的做法是把稀疏單詞換成代表未知單詞的標記，而不是獨立標記它們，這樣做可減少神經網路所需學習的權重數量。
- 單詞可以進行詞幹（*stemmed*）處理，代表把它們縮減為最簡單的形式，好讓同一個動詞的不同時態可被標記在一起。例如，*browse*、*browsing*、*rowses* 和 *browsed* 可全部被整理為 *brows*。
- 標點符號也可被分詞，或全部刪除。
- 使用單詞標記代表模型將永遠無法預測出未包含在訓練詞彙表中的單詞。

如果使用字元標記：

- 模型能夠生成由字元組成的序列，形成未包含在訓練詞彙表中的新詞——這在某些情況下可能是很棒的效果，但在其他情況下就不一定了。
- 大寫字母可以轉換為對應的小寫，或者單獨標記也可以。
- 使用字元標記的詞彙表通常會小巧許多。這有助於提升模型的訓練速度，因為最終輸出層所需學習的權重較少。

以下範例將使用小寫單詞分詞，且不進行詞幹處理。另外還會對標點符號進行分詞，因為我們希望模型能夠預測何時應該結束句子或使用逗號等標點符號。

範例 5-4 的程式碼進行了文字的清理和分詞。

範例 *5-4*　標記化

```
def pad_punctuation(s):
    s = re.sub(f"([{string.punctuation}])", r' \1 ', s)
    s = re.sub(' +', ' ', s)
    return s
```

```
text_data = [pad_punctuation(x) for x in filtered_data] ❶

text_ds = tf.data.Dataset.from_tensor_slices(text_data).batch(32).shuffle(1000) ❷

vectorize_layer = layers.TextVectorization( ❸
    standardize = 'lower',
    max_tokens = 10000,
    output_mode = "int",
    output_sequence_length = 200 + 1,
)

vectorize_layer.adapt(text_ds) ❹
vocab = vectorize_layer.get_vocabulary() ❺
```

❶ 將標點符號補全，以將它們視為單獨的單詞。

❷ 將其轉換為 TensorFlow Dataset。

❸ 建立一個 Keras `TextVectorization` 層，將文字轉換為小寫，為最常見的 10,000 個單詞分配對應的整數標記，並將序列修剪或填充到 201 個標記的長度。

❹ 將 `TextVectorization` 層應用於訓練資料。

❺ `vocab` 變數用於儲存單詞標記的清單。

範例 5-5 是經過分詞處理後的食譜。訓練模型時使用的序列長度是訓練過程的一個參數。本範例使用的序列長度為 200，因此要把食譜填充或裁剪到比這個長度再加 1 以便建立目標變數（下一段詳述）。為了滿足這個期望長度，向量的尾端要填 0。

停止標記

標記 `0` 稱為停止標記（*stop token*）表示文字字串已經結束。

範例 5-5　經過分詞處理的範例 5-3 食譜

```
[  26   16  557    1    8  298  335  189    4 1054  494   27  332  228
  235  262    5  594   11  133   22  311    2  332   45  262    4  671
    4   70    8  171    4   81    6    9   65   80    3  121    3   59
   12    2  299    3   88  650   20   39    6    9   29   21    4   67
  529   11  164    2  320  171  102    9  374   13  643  306   25   21
    8  650    4   42    5  931    2   63    8   24    4   33    2  114
   21    6  178  181 1245    4   60    5  140  112    3   48    2  117
```

```
  557    8  285  235    4  200  292  980    2  107  650   28   72    4
  108   10  114    3   57  204   11  172    2   73  110  482    3  298
    3  190    3   11   23   32  142   24    3    4   11   23   32  142
   33    6    9   30   21    2   42    6  353    3 3224    3    4  150
    2  437  494    8 1281    3   37    3   11   23   15  142   33    3
    4   11   23   32  142   24    6    9  291  188    5    9  412  572
    2  230  494    3   46  335  189    3   20  557    2    0    0    0
    0    0    0    0    0]
```

在範例 5-6 中，可以看到單詞標記的部分清單以及它們對應的索引。該層將數字 0 保留為填充（就是停止標記），而數字 1 用於未包含在前 10,000 個單詞中的未知單詞（例如 persillade）。其他單詞則按照出現頻率高低來分配標記。詞彙表要納入多少單詞中也是訓練過程的一個參數。包含的單詞越多，其中的未知標記就會越少；然而，你的模型也需要更大的容量才能容納更大的詞彙表。

範例 5-6　*TextVectorization* 層的詞彙表

```
0:
1: [UNK]
2: .
3: ,
4: and
5: to
6: in
7: the
8: with
9: a
```

建立訓練資料集

我們的 LSTM 將被訓練在給定某一點之前的一串單詞之後，可預測出序列中的下一個單詞。例如，我們可以將 *grilled chicken with boiled* 作為標記送入模型，並期望它輸出適當的下一個單詞（例如 *potatoes* 而不是 *bananas*）。

因此，只需將整個序列向後移動一個標記就能建立目標變數了。

資料集生成步驟可藉由範例 5-7 的程式碼完成。

範例 5-7　建立訓練資料集

```
def prepare_inputs(text):
    text = tf.expand_dims(text, -1)
    tokenized_sentences = vectorize_layer(text)
    x = tokenized_sentences[:, :-1]
```

```
    y = tokenized_sentences[:, 1:]
    return x, y

train_ds = text_ds.map(prepare_inputs) ❶
```

❶ 建立訓練資料集，由食譜標記輸入以及向後移動一個標記的相同向量（目標）所組成。

LSTM 架構

LSTM 模型的整體架構如表 5-1。模型的輸入是一系列整數標記，輸出是每個詞在長度 10,000 的詞彙中下一次出現的機率。如果要詳細了解其運作原理，需要介紹兩種新類型的網路層：`Embedding` 和 `LSTM`。

表 5-1　LSTM 的模型摘要

Layer (type)	Output shape	Param #
InputLayer	(None, None)	0
Embedding	(None, None, 100)	1,000,000
LSTM	(None, None, 128)	117,248
Dense	(None, None, 10000)	1,290,000

Total params	2,407,248
Trainable params	2,407,248
Non-trainable params	0

LSTM 的輸入層

請注意，`Input` 層不需要事先指定序列長度。批大小和序列長度都可彈性調整（代表形狀為 `(None, None)`）。這是因為所有的後續層都無法得知對於要送入的序列長度。

嵌入層

嵌入層本質上是一個查找表，將每個整數標記轉換為長度為 `embedding_size` 的向量，如圖 5-2。這些查找向量是由模型作為權重來學習的。因此，該層所需學習的權重數量等於詞彙表的大小乘以嵌入向量的維度（即 10,000 × 100 = 1,000,000）。

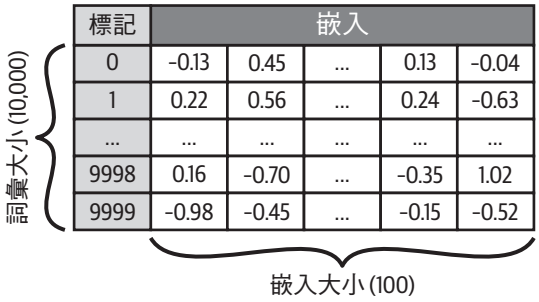

圖 5-2　嵌入層是各個整數標記的查找表

我們將每個整數標記嵌入到一個連續向量中，因為這可讓模型透過反向傳播來學習每個單詞的表示。單純對每個輸入標記進行獨熱編碼也是可以的，但是嵌入層是更受歡迎的做法，因為它使嵌入本身就能夠被訓練，從而使模型在嵌入標記來提高效能方面變得更加靈活。

因此，輸入層會把形狀為 `[batch_size, seq_length]` 的整數序列張量送入 `Embedding` 層，後者再輸出形狀為 `[batch_size, seq_length, embedding_size]` 的張量。然後，該張量再被送入 `LSTM` 層（圖 5-3）。

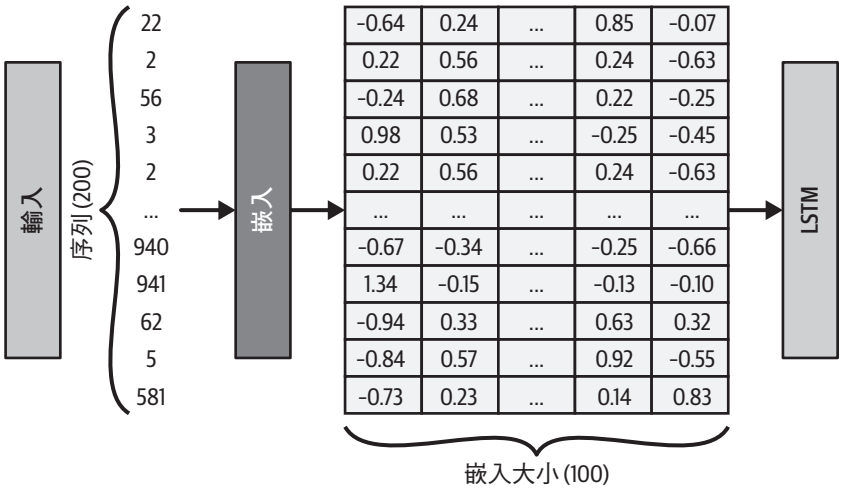

圖 5-3　流經嵌入層的單一序列

LSTM 層

要理解 LSTM 層，首先要先知道一般遞歸層的運作原理。

遞歸層具有特殊的特性，能夠處理序列輸入資料 $x_1, ..., x_n$。它包含了一個單元，當序列 x_t 中的每個元素依序透過它時，它會以每次一個時步來更新自身的隱藏狀態 h_t。

隱藏狀態是一個長度等於單元中單位數量的向量，可將其視為單元對序列的當前理解。在時步 t，單元會運用上一個隱藏狀態 h_{t-1} 和來自當前時步 x_t 的資料產生一個更新後的隱藏狀態向量 h_t。這個循環過程會一直持續到序列結束為止。一旦序列結束，該層會輸出單元的最終隱藏狀態 h_n，並送入網路的下一層，過程如圖 5-4。

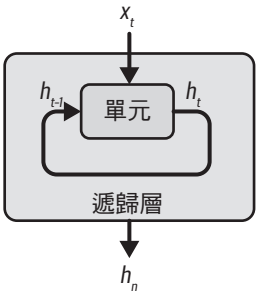

圖 5-4　遞歸層的簡單示意圖

為了更詳細地解釋這一點，讓我們展開這個過程，以便觀察單個序列是如何透過該層的（圖 5-5）。

單元權重

重點在於，本圖中的所有單元共享了相同的權重（因為它們根本就是同一個單元）。本圖與圖 5-4 是一樣的；只是改用不同的方式來描述遞歸層的機制。

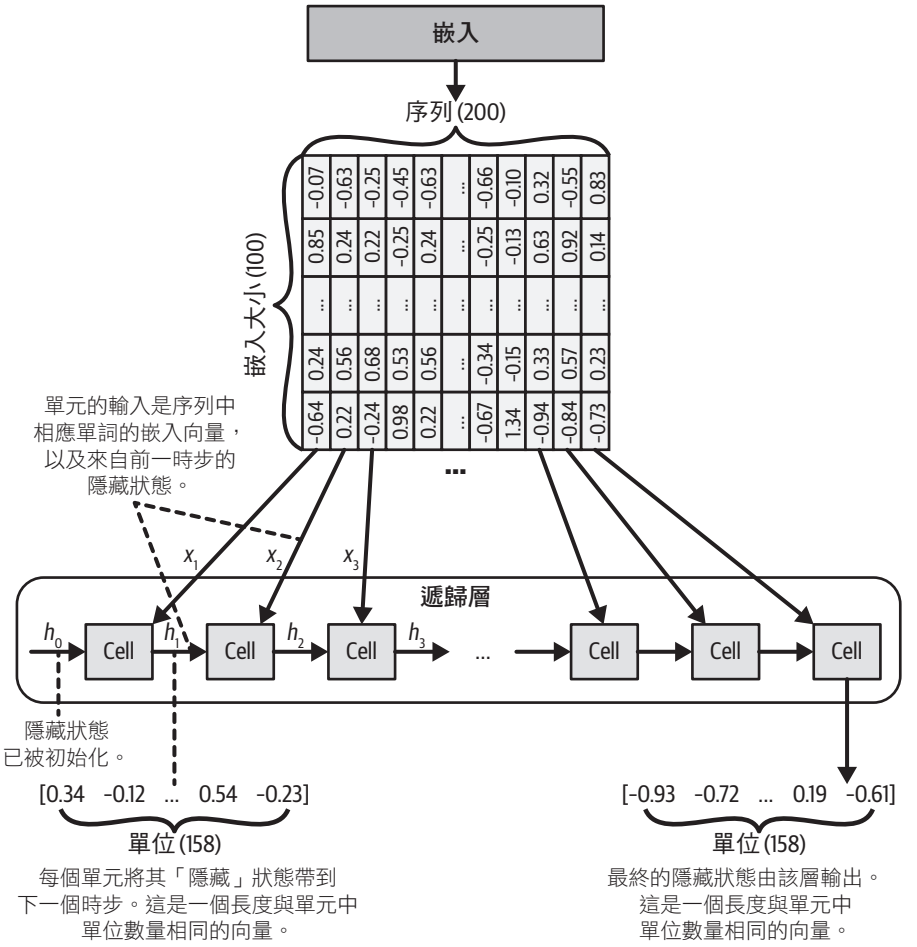

圖 5-5　單一序列如何流經遞歸層

整個遞歸過程現在的表示方式是在每個時步中另外畫出該單元的複製，並顯示隱藏狀態如何在透過單元時被不斷更新。我們可以清楚地看到先前的隱藏狀態是如何與序列的當前資料點（即當前的嵌入詞向量）混合以產生下一個隱藏狀態。該層的輸出就是該單元在處理輸入序列中的每個詞之後的最終隱藏狀態。

實際上，將單元輸出的結果稱為隱藏狀態是一個不太恰當的命名慣例——它並非真的隱藏，你也不應該這樣去理解。事實上，最後一個隱藏狀態就是該層的整體輸出，本章稍後就會運用這一點來存取個別時步的隱藏狀態。

第七章

能量模型

本章目標

本章學習內容如下：

- 了解如何制定深度能量模型（EBM）。
- 學習如何使用朗之萬動力學（Langevin dynamics）從 EBM 抽樣。
- 用對比散度法自行訓練 EBM。
- 分析 EBM，包括觀察朗之萬動力學的抽樣過程。
- 認識限制波茲曼機器等其他類型的 EBM。

能量模型是一種借助了物理系統建模概念的生成模型廣泛類別，此概念即可以用波茲曼分布——一種將實值能量函數正規化在 0 和 1 之間的特定函數——來表示事件機率。此分布最初由路德維希・波茲曼於 1868 年為了表示熱平衡中的氣體而提出。

本章將學習如何利用這個概念來訓練一個能夠生成手寫數字圖像的生成模型。我們將探索幾個新概念，包括訓練 EBM 的對比散度法與用於抽樣的朗之萬動力學。

簡介

讓我們用一個小故事來說明能量模型背後的關鍵概念。

朗奧文長跑俱樂部

黛安·米克斯是一位住在虛構法國小鎮朗奧文的長跑隊教練。她以卓越的培訓能力聞名，即使是最平庸的運動員也能被她培養成世界級跑者而享負盛名（圖 7-1）。

圖 7-1　正在訓練優秀運動員的長跑教練（圖片使用 Midjourney 生成）

黛安的做法是以評估每位運動員的體力程度為基礎。多年來與各種能力的運動員合作讓她鍛鍊出極為準確的直覺，只需看一眼完賽後的運動員便知道他們還剩下多少精力。運動員剩餘的體力越低越好，因為最優秀的運動員總是在比賽中全力以赴！

為了不斷精進自身技術，她會定期透過比對自己對知名選手和對俱樂部中最優秀的選手的判斷結果來訓練自己。她還確保對這兩組人馬的預測差異越大越好，這樣當她說在俱樂部中找到了真正的優秀選手時，人們才會認真看待。

黛安真正厲害的地方在於能夠將一名平庸的跑者改造成頂級跑者。方法很簡單——首先，測量運動員目前的體力程度，並找出運動員要在下一次比賽中進步所必須要進行的調整。接著，做完調整後黛安會再次測量運動員的剩餘體力，並期待會比之前剩得更少，因為這說明了選手在跑道上的表現有所提升。這個評估最佳所需調整方式並朝著正確方向逐步邁進的過程會持續進行下去，直到選手最終與世界級跑者並駕齊驅。

多年後，黛安從教練一職退休並出版了一本關於培養優秀運動員的書籍——她將這個系統命名為「朗奧文的黛安培訓法」。

黛安・米克斯和朗奧文長跑俱樂部的故事概括了能量模型背後的關鍵。在用 Keras 實作實際範例之前，先讓我們進一步探討這個理論。

能量模型

能量模型會運用波茲曼分布（方程式 7-1）來針對真實資料生成分布建模，其中 $E(x)$ 為觀測 x 的能量函數（或分數）。

方程式 7-1　波茲曼分布

$$p(\mathbf{x}) = \frac{e^{-E(\mathbf{x})}}{\int_{\hat{\mathbf{x}} \in \mathbf{X}} e^{-E(\hat{\mathbf{x}})}}$$

實際上，這相當於訓練出一個神經網路 $E(x)$ 來為可能性高的觀測值輸出較低的分數（$p\mathbf{x}$ 接近 1），並為可能性低的觀測值輸出高分（$p\mathbf{x}$ 接近 0）。

用這種方式建模資料有兩個挑戰。首先，不清楚該如何用模型抽樣新的觀測值，它可以為觀測值打分數，但是該如何生成低分（即可信度高）的觀測值呢？

其次，方程式 7-1 的正規化分母中含有一個積分項，除了最簡單的問題之外其他都很棘手。如果無法計算這個積分，那麼就無法用最大似然估計法來訓練模型，因為 $p\mathbf{x}$ 必須是一個有效的機率分布。

能量模型背後的關鍵是，能夠使用近似技術來確保我們永遠不需要計算棘手的分母。這與正規化流完全相反，其中我們必須竭盡全力確保應用於標準高斯分布上的變換不會改變輸出仍是有效機率分布的事實。

依照 Du 和 Mordatch 等人於 2019 年發表的「Implicit Generation and Modeling with Energy-Based Models」[1] 論文中提出的想法，我們可以透過一種名為對比散度法的技術（用於訓練）、以及名為朗之萬動力學的技術（用於抽樣）來避開棘手的分母問題。在本章建構 EBM 的部分將詳細討論這些技術。

首先要建立資料集並設計一個簡單的神經網路來表示實值能量函數 $E(x)$。

執行本範例的程式碼

本範例的 Jupyter notebook 程式碼請由本書 Github 取得：
notebooks/07_ebm/01_ebm/ebm.ipynb

此程式碼是根據 Phillip Lippe 的優質深度能量生成模型教學（*https://oreil.ly/kyO9B*）修改而來。

MNIST 資料集

在此使用包含手寫黑白數字圖像的標準 MNIST 資料集（*https://oreil.ly/mSvhc*）。資料集的範例如圖 7-2。

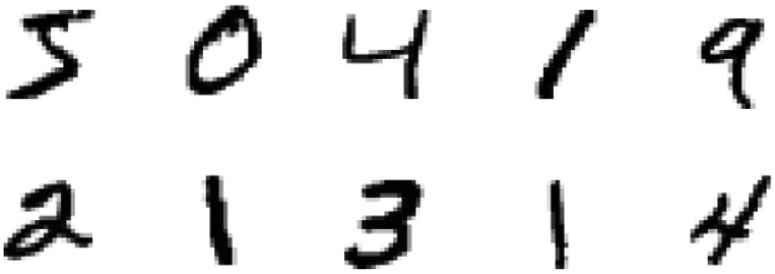

圖 7-2　MNIST 資料集的圖像範例

這個資料集已經整合在 TensorFlow 中，請依照範例 7-1 下載。

範例 *7-1*　下載 *MNIST* 資料集

```
from tensorflow.keras import datasets
(x_train, _), (x_test, _) = datasets.mnist.load_data()
```

跟之前一樣，要將像素縮放至 [-1, 1] 之間，並加入填充讓圖像大小為 32 × 32。還要將它轉換成 TensorFlow 資料集，如範例 7-2。

範例 *7-2*　預處理 *MNIST* 資料集

```
def preprocess(imgs):
    imgs = (imgs.astype("float32") - 127.5) / 127.5
    imgs = np.pad(imgs , ((0,0), (2,2), (2,2)), constant_values= -1.0)
    imgs = np.expand_dims(imgs, -1)
    return imgs

x_train = preprocess(x_train)
```

```
x_test = preprocess(x_test)
x_train = tf.data.Dataset.from_tensor_slices(x_train).batch(128)
x_test = tf.data.Dataset.from_tensor_slices(x_test).batch(128)
```

有了資料集便可以來建構用於表示能量函數 $E(x)$ 的神經網路了。

能量函數

能量函數 $E_\theta(x)$ 是一個具備能夠將輸入影像 x 轉換為純量值的參數 θ 之神經網路。整個網路都會使用一種名為 *swish* 的觸發函數，說明如下。

Swish 觸發函數

swish 是 Google 於 2017 年推出的 ReLU 替代方案[2]，定義如下：

$$\text{swish}(x) = x \cdot \text{sigmoid}(x) = \frac{x}{e^{-x}+1}$$

swish 看起來跟 ReLU 很像，主要差別在於 swish 是平滑的，有助於減緩梯度消失問題，這對能量模型來說尤其重要。圖 7-3 為 swish 函數圖。

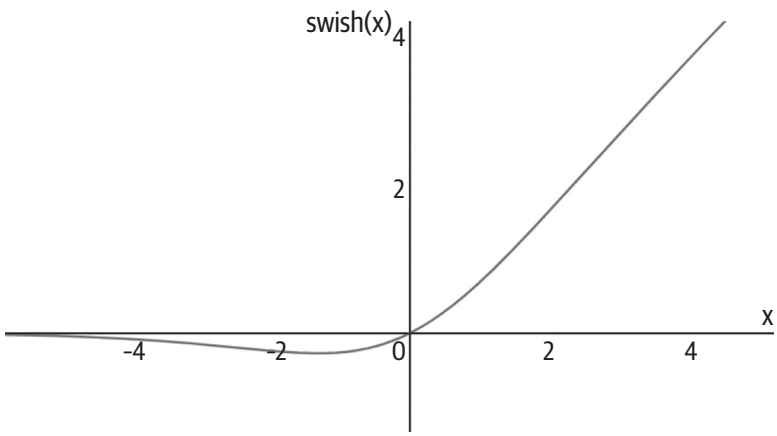

圖 7-3　swish 觸發函數

本範例的網路是一組彼此堆疊的 Conv2D 層，隨著通道數量的增加同時逐漸減少圖像大小。最後一層是具有線性觸發的單一個全連接單元，這樣就能讓網路輸出一個介於 $(-\infty, \infty)$ 之間的值。範例 7-3 為建置本網路的程式碼。

範例 7-3　建置能量函數 *E(x)* 神經網路

```
ebm_input = layers.Input(shape=(32, 32, 1))
x = layers.Conv2D(
    16, kernel_size=5, strides=2, padding="same", activation = activations.swish
)(ebm_input) ❶
x = layers.Conv2D(
    32, kernel_size=3, strides=2, padding="same", activation = activations.swish
)(x)
x = layers.Conv2D(
    64, kernel_size=3, strides=2, padding="same", activation = activations.swish
)(x)
x = layers.Conv2D(
    64, kernel_size=3, strides=2, padding="same", activation = activations.swish
)(x)
x = layers.Flatten()(x)
x = layers.Dense(64, activation = activations.swish)(x)
ebm_output = layers.Dense(1)(x) ❷
model = models.Model(ebm_input, ebm_output) ❸
```

❶ 能量函數是由多個具備 swish 觸發函數的 `Conv2D` 層堆疊而成。

❷ 最後一層為具備線性觸發函數的單一全連接單元。

❸ 將輸入圖像轉換成純量值的 Keras `Model` 函式。

使用朗之萬動力學抽樣

能量函數只會針對特定輸入給出一個分數，該如何利用這個函數來生成低分的新樣本呢？

我們要用一種名為*朗之萬動力學*的技術，它利用了能夠計算出相對於輸入的能量函數梯度這項特性。從樣本空間任一處開始沿著所得梯度的反方向緩緩移動，如此就能逐漸降低能量函數。如果神經網路訓練正確，那麼應該會看到這個隨機雜訊轉變成一個類似訓練集觀測值的圖像！

隨機梯度朗之萬動力學

穿越樣本空間時需要對輸入加入少量的隨機雜訊，否則可能會陷入區域最小值。正因如此，此技術叫做隨機梯度朗之萬動力學[3]。

圖 7-4 是將梯度下降過程在二維空間中視覺化呈現，第三軸則是能量函數值。此路徑是雜訊按照能量函數 $E(x)$ 相對於輸入 x 的負梯度而降低的下坡。MNIST 圖像資料集中有 1024 個像素，所以會是在一個 1024 維度的空間中，但原理是一樣的！

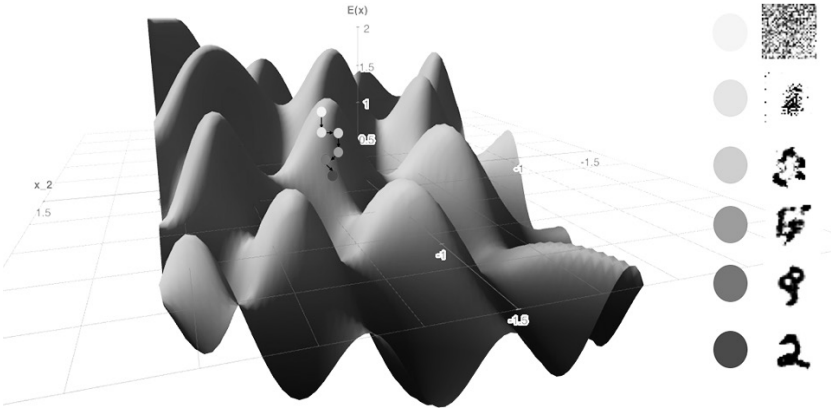

圖 7-4　使用朗之萬動力學的梯度下降

要特別提及此梯度下降法與平常用來訓練神經網路的梯度下降之間的差異。

在訓練神經網路時，會用反向傳播計算出損失函數相對於網路參數（即權重）的梯度。接著往負梯度一點一點地更新參數，以便在多次迭代後逐漸最小化損失。

而使用朗之萬動力學時，則會讓神經網路權重維持固定計算出輸出相對於輸入的梯度。接著往負梯度逐漸更新輸入以便在多次迭代後逐漸最小化輸出（能量分數）。

兩者都用了同樣的概念（梯度下降），但應用於不同函數並涉及了不同實體。

朗之萬動力學可用以下方程式表示：

$$x^k = x^{k-1} - \eta \nabla_x E_\theta\left(x^{k-1}\right) + \omega$$

在此，$\omega \sim \mathcal{N}(0, \sigma)$ 和 $x^0 \sim \mathcal{U}(-1, 1)$ 是需要調整的步長超參數，太高的話會讓學習步驟跳過最小值；太低的話則會讓演算法收斂太慢。

$x^0 \sim \mathcal{U}(-1, 1)$ 是指範圍 $[-1, 1]$ 的均勻分布。

請依照範例 7-4 編寫朗之萬抽樣函數。

範例 *7-4*　朗之萬抽樣函數

```
def generate_samples(model, inp_imgs, steps, step_size, noise):
    imgs_per_step = []
    for _ in range(steps): ❶
        inp_imgs += tf.random.normal(inp_imgs.shape, mean = 0, stddev = noise) ❷
        inp_imgs = tf.clip_by_value(inp_imgs, -1.0, 1.0)
        with tf.GradientTape() as tape:
            tape.watch(inp_imgs)
            out_score = -model(inp_imgs) ❸
        grads = tape.gradient(out_score, inp_imgs) ❹
        grads = tf.clip_by_value(grads, -0.03, 0.03)
        inp_imgs += -step_size * grads ❺
        inp_imgs = tf.clip_by_value(inp_imgs, -1.0, 1.0)
        return inp_imgs
```

❶ 根據指定步數重複執行。

❷ 在圖像中加入少量雜訊。

❸ 將圖像透過模型以取得能量分數。

❹ 計算相對於輸入的輸出梯度。

❺ 在輸入影像中加入少量梯度。

用對比散度法訓練模型

知道如何從樣本空間抽樣全新的低能量點之後，接著來看模型訓練吧。

由於能量函數並非輸出機率，在此就無法使用最大似然估計法；它輸出的是一個在樣本空間中積分結果不為 1 的分數。相反地，我們要用 Geoffrey Hinton 首次於 2002 年為了訓練非正規化分數模型而提出的對比散度法[4]。

這次要最小化的值為資料的負對數似然：

$$\mathscr{L} = -\mathbb{E}_{x \sim \text{data}}\left[\log p_\theta(\mathbf{x})\right]$$

當 $p_\theta(\mathbf{x})$ 的形式為波茲曼分布，能量函數為 $E_\theta(\mathbf{x})$，則該值的梯度表示如下（完整推導請參考 Oliver Woodford 的「Notes on Contrastive Divergence」）[5]：

$$\nabla_\theta \mathscr{L} = \mathbb{E}_{x \sim \text{data}}\left[\nabla_\theta E_\theta(\mathbf{x})\right] - \mathbb{E}_{x \sim \text{model}}\left[\nabla_\theta E_\theta(\mathbf{x})\right]$$

這一看就很合理，因為我們希望模型能針對實際觀測值輸出較大的能量分數負值，並為生成的假觀測值輸出較大的能量分數正值，藉此極大化這兩個極端之間的對比。

換句話說，我們可以計算真假樣本的能量分數之間的差異，並用它作為損失函數。

要計算假樣本的能量分數，需要能夠精準地從分布 $p_\theta(\mathbf{x})$ 抽樣，但由於分母計算過於複雜所以無法做到。相反地，我們可以用朗之萬抽樣過程來生成一組具低能量分數的觀測值。這個過程需要無限個步驟才能生成一個完美的樣本（顯然很不切實際），因此改成執行幾個步驟就好，並假設這已足以產生有意義的損失函數。

另外要從前幾次迭代保留一些緩衝樣本作為下一批的起點，讓它不完全是隨機雜訊。範例 7-5 為產生抽樣緩衝的程式碼。

範例 *7-5*　抽樣緩衝

```
class Buffer:
    def __init__(self, model):
        super().__init__()
        self.model = model
        self.examples = [
            tf.random.uniform(shape = (1, 32, 32, 1)) * 2 - 1
            for _ in range(128)
        ] ❶

    def sample_new_exmps(self, steps, step_size, noise):
        n_new = np.random.binomial(128, 0.05) ❷
        rand_imgs = (
            tf.random.uniform((n_new, 32, 32, 1)) * 2 - 1
        )
        old_imgs = tf.concat(
            random.choices(self.examples, k=128-n_new), axis=0
        ) ❸
        inp_imgs = tf.concat([rand_imgs, old_imgs], axis=0)
        inp_imgs = generate_samples(
            self.model, inp_imgs, steps=steps, step_size=step_size, noise = noise
        ) ❹
        self.examples = tf.split(inp_imgs, 128, axis = 0) + self.examples ❺
        self.examples = self.examples[:8192]
        return inp_imgs
```

❶ 用一批隨機雜訊來初始化抽樣緩衝。

❷ 每次平均有 5% 的觀測值為重新生成（即隨機雜訊）。

❸ 其餘則從現有緩衝中隨機抽樣。

❹ 將觀測值串聯起來並用朗之萬抽樣器跑過一遍。

❺ 將生成樣本加入緩衝，緩衝區的最大長度改為 8192 筆觀測值。

圖 7-5 為對比散度法的其中一個訓練步驟。實際觀測值的分數會被演算法壓低，而假觀測值的分數則被拉高，且無須在每個步驟之後將這些分數正規化。

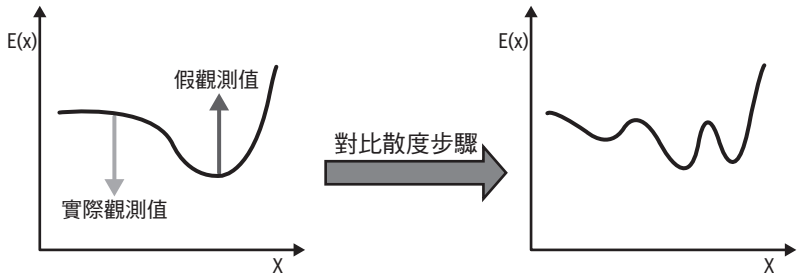

圖 7-5　對比散度法其中一個步驟

請依照範例 7-6 在自訂的 Keras 模型中編寫對比散度演算法的訓練步驟。

範例 *7-6*　使用對比散度法訓練 *EBM*

```
class EBM(models.Model):
    def __init__(self):
        super(EBM, self).__init__()
        self.model = model
        self.buffer = Buffer(self.model)
        self.alpha = 0.1
        self.loss_metric = metrics.Mean(name="loss")
        self.reg_loss_metric = metrics.Mean(name="reg")
        self.cdiv_loss_metric = metrics.Mean(name="cdiv")
        self.real_out_metric = metrics.Mean(name="real")
        self.fake_out_metric = metrics.Mean(name="fake")

    @property
    def metrics(self):
        return [
            self.loss_metric,
            self.reg_loss_metric,
            self.cdiv_loss_metric,
            self.real_out_metric,
            self.fake_out_metric
        ]

    def train_step(self, real_imgs):
```

```
            real_imgs += tf.random.normal(
                shape=tf.shape(real_imgs), mean = 0, stddev = 0.005
            ) ❶
            real_imgs = tf.clip_by_value(real_imgs, -1.0, 1.0)
            fake_imgs = self.buffer.sample_new_exmps(
                steps=60, step_size=10, noise = 0.005
            ) ❷
            inp_imgs = tf.concat([real_imgs, fake_imgs], axis=0)
            with tf.GradientTape() as training_tape:
                real_out, fake_out = tf.split(self.model(inp_imgs), 2, axis=0) ❸
                cdiv_loss = tf.reduce_mean(fake_out, axis = 0) - tf.reduce_mean(
                    real_out, axis = 0
                ) ❹
                reg_loss = self.alpha * tf.reduce_mean(
                    real_out ** 2 + fake_out ** 2, axis = 0
                ) ❺
                loss = reg_loss + cdiv_loss
            grads = training_tape.gradient(loss, self.model.trainable_variables) ❻
            self.optimizer.apply_gradients(
                zip(grads, self.model.trainable_variables)
            )
            self.loss_metric.update_state(loss)
            self.reg_loss_metric.update_state(reg_loss)
            self.cdiv_loss_metric.update_state(cdiv_loss)
            self.real_out_metric.update_state(tf.reduce_mean(real_out, axis = 0))
            self.fake_out_metric.update_state(tf.reduce_mean(fake_out, axis = 0))
            return {m.name: m.result() for m in self.metrics}

        def test_step(self, real_imgs): ❼
            batch_size = real_imgs.shape[0]
            fake_imgs = tf.random.uniform((batch_size, 32, 32, 1)) * 2 - 1
            inp_imgs = tf.concat([real_imgs, fake_imgs], axis=0)
            real_out, fake_out = tf.split(self.model(inp_imgs), 2, axis=0)
            cdiv = tf.reduce_mean(fake_out, axis = 0) - tf.reduce_mean(
                real_out, axis = 0
            )
            self.cdiv_loss_metric.update_state(cdiv)
            self.real_out_metric.update_state(tf.reduce_mean(real_out, axis = 0))
            self.fake_out_metric.update_state(tf.reduce_mean(fake_out, axis = 0))
            return {m.name: m.result() for m in self.metrics[2:]}

    ebm = EBM()
    ebm.compile(optimizer=optimizers.Adam(learning_rate=0.0001), run_eagerly=True)
    ebm.fit(x_train, epochs=60, validation_data = x_test,)
```

❶ 在真實圖像中加入少量隨機雜訊，以避免對訓練資料集過度擬合。

❷ 從緩衝中抽樣一組假圖像。

❸ 將真假圖像用模型跑過以產生各自的分數。

❹ 對比散度損失就是真假觀測分數之間的差異。

❺ 加入正則化損失以避免分數過高。

❻ 計算相對於網路權重的損失函數梯度以進行反向傳播。

❼ 在驗證期間使用 `test_step`，並計算隨機雜訊和訓練集資料分數之間的對比散度。可用作評估模型訓練情況的指標（見下一節）。

分析能量模型

訓練過程中的損失曲線和輔助指標如圖 7-6。

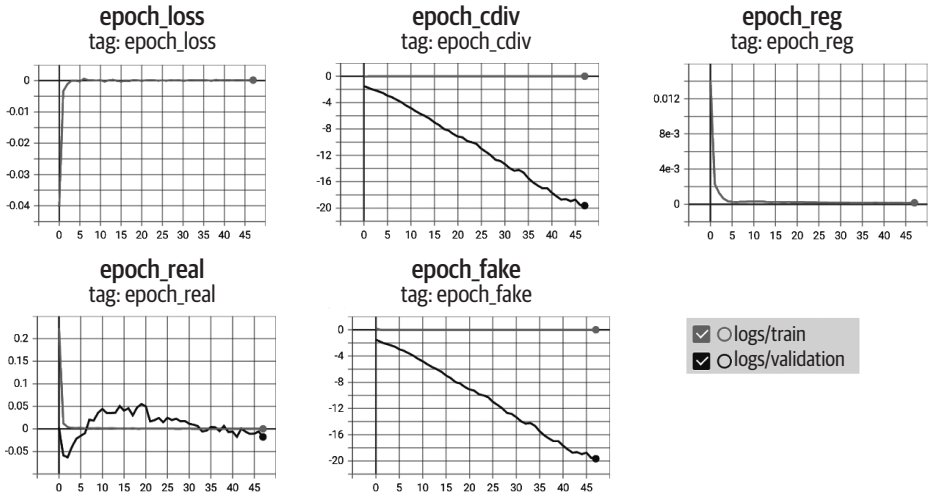

圖 7-6　能量模型訓練過程中的損失曲線和指標

首先，請注意在訓練步驟中計算的損失在各回合幾乎維持不變且偏小。雖然模型不斷改進，但緩衝區中生成圖像品質也在提高，這些圖像需要與訓練集中的真實圖像進行比較，因此不應期待訓練損失會顯著下降。

為此我們還設立了一個驗證過程來評估模型效能，該過程不會從緩衝區抽樣，而是對一個隨機雜訊樣本進行評分，並將它與訓練資料集中的樣本分數進行比較。如果模型確實有在改進，就會看到對比散度隨著回合推進而下降（即分辨隨機雜訊與真實圖像的能力變強了），如圖 7-6。

從能量模型生成新樣本只需要從某個起始點（隨機雜訊）開始多次執行朗之萬抽樣器就好，如範例 7-7。觀測值會依照之於輸入的評分函數梯度而被強迫下降，以致從雜訊中浮現出一個可信度高的觀測值。

範例 *7-7*　用能量模型生成新觀測值

```
start_imgs = np.random.uniform(size = (10, 32, 32, 1)) * 2 - 1
gen_img = generate_samples(
    ebm.model,
    start_imgs,
    steps=1000,
    step_size=10,
    noise = 0.005,
    return_img_per_step=True,
)
```

經過 50 個訓練回合後，抽樣器生成的範例觀測值如圖 7-7。

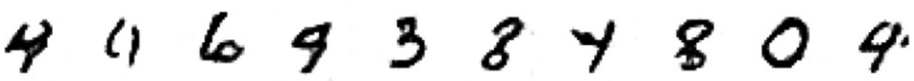

圖 7-7　朗之萬抽樣器所產生的樣本，其中運用了 EBM 模型來引導梯度下降過程

我們也可以在朗之萬抽樣過程中擷取目前觀測的圖像來重現單一觀測的生成過程，如圖 7-8。

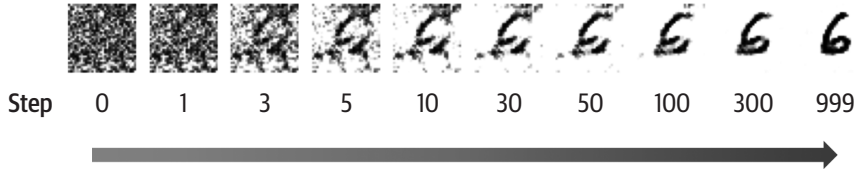

圖 7-8　觀測值在朗之萬抽樣過程中不同步驟的快照

其他能量模型

上述範例是一個透過對比散度與朗之萬抽樣所訓練的深度 EBM。然而，早期的 EBM 並未使用朗之萬抽樣，而是仰賴於其他技術和架構。

第十一章

音樂生成

本章目標

本章學習內容如下：

- 了解如何將音樂生成視為序列預測問題，以應用 Transformer 這類自迴歸模型。
- 了解如何使用 `music21` 套件解析和標記化 MIDI 檔來建立訓練資料集。
- 學習使用正弦位置編碼。
- 使用多個處理音符和音長的輸入及輸出訓練音樂生成 Transformer。
- 了解如何處理複音音樂，包括網格標記和事件型標記。
- 訓練 MuseGAN 模型來生成多軌音樂。
- 使用 MuseGAN 調整生成小節的各種屬性。

音樂創作是一個複雜且富含創意的過程，涉及結合不同的音樂元素，如旋律、和聲、節奏和音色等。長久以來它被視為人類獨有的行為，但近期技術的飛快發展已使得生成既悅耳又具長期結構的音樂變得可能。

Transformer 是最受歡迎的音樂生成技術之一，因為音樂可以被視為序列預測問題。這些模型被改良成可以透過將音符視為一系列的標記（類似於語句中的單詞）來生成音樂。Transformer 模型會根據先前音符來預測序列的下一個音符，從而生成一段音樂。

分析音樂生成 Transformer

首先，將 START 音符標記和 0.0 音長標記送入網路，藉此從頭開始生成一段音樂（也就是讓模型認為它是從樂曲的開頭開始）。接著，使用與第 9 章中用於生成文字序列的迭代技術來生成一段樂句，步驟如下：

1. 模型會根據目前序列（音符和音長）預測兩種分布，一個是下一個音符，另一個是下一個音長。
2. 從這兩個分布抽樣，並使用溫度參數來控制抽樣過程的變化程度。
3. 將所選音符和音長附加到對應的輸入序列中。
4. 使用新的輸入序列重複此過程，直到生成所需數量。

圖 11-8 為模型在訓練過程的不同時期從無到有的一些音樂範例。音符和音長的溫度為 0.5。

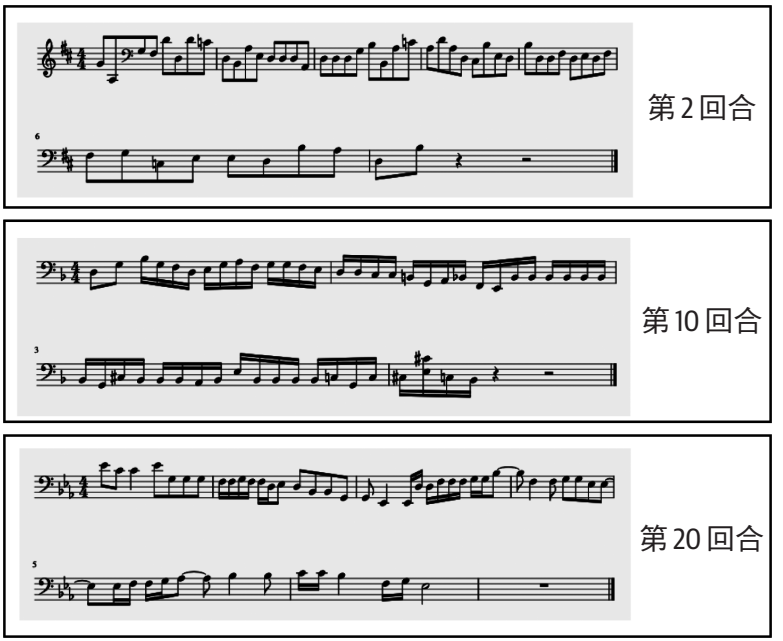

圖 11-8　僅指定 START 音符標記和 0.0 音長標記時，模型生成的樂句範例

本節大部分將集中在分析音符預測上，而非音長，因為就巴哈的大提琴組曲而言，和聲的複雜性更難以捉摸，因此更值得研究。然而，你也可以將相同的分析應用於模型的節奏預測上，這對其他可用來訓練此模型的音樂風格（例如打擊樂）可能更為相關。

圖 11-8 中的樂句段落有幾點需要特別注意。首先，請注意樂句如何隨著訓練的進展變得越來越複雜。一開始，模型保守地選擇了同一組音符和節奏。到第 10 回合，模型開始生成一小串音符，到了第 20 回合，模型產生出有趣的節奏並牢牢地建立於特定音調上（降 E 大調）。

再來，我們可將每個時步的預測分布繪製成熱圖，藉此來分析音符依時間的分布狀況。圖 11-9 為圖 11-8 中第 20 回合範例的熱圖。

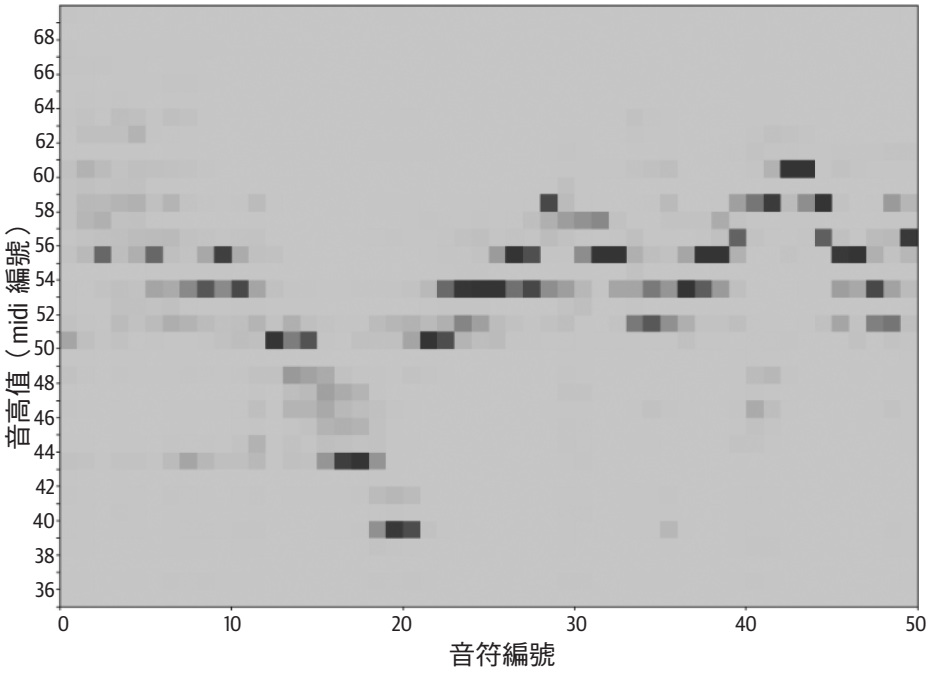

圖 11-9　下一個可能音符隨著時間推移之分布狀況（於第 20 回合）：方塊顏色越深表示模型對於下一個音符之音高的確定程度越高