前言

機器學習海嘯

Geoffrey Hinton 等研究者在 2006 年發表了一篇論文（*https://homl.info/136*）[1]，文中介紹如何訓練以極高準確率（>98%）來辨識手寫數字的深度神經網路，他們將這項技術稱為「深度學習」。深度神經網路是大腦皮層的（極度）簡化模型，它是由多層人工神經元組成的。當時人們普遍認為訓練深度神經網路是不可能的任務[2]，絕大多數的研究者在 1990 年代已經放棄這種想法，但這篇論文重新引起科學界的興趣，不久之後，許多新論文都證實，深度學習不僅可行，也能夠實現令人驚嘆的成就，其他的機器學習（ML）技術皆無法望其項背（在強大的計算能力和龐大的資料量的幫助下）。很快地，這股熱潮擴展至機器學習的許多其他領域。

大約十年後，機器學習已經征服整個產業了，如今，它是許多高科技產品的核心，可為你排序網路搜尋結果、驅動智慧型手機的語音辨識功能、推薦影片，甚至幫你開車。

在你的專案中的機器學習

所以，你一定對機器學習很感興趣，想要快點加入這場盛會吧？！

或許你想要讓自製的機器人擁有自己的大腦？讓它辨識人臉？或學會到處行走？

1 Geoffrey E. Hinton et al., "A Fast Learning Algorithm for Deep Belief Nets", *Neural Computation* 18 (2006): 1527–1554.

2 儘管 Yann LeCun 的深度摺積神經網路自從 1990 年代以來就可以準確地辨識圖像了，但當時它們沒有那麼通用。

或者，你的公司有大量的資料（使用者紀錄、財務資料、生產資料、機器感應器資料、熱線統計、人力資源報告…等），只要知道該在哪裡尋找它們，就能發現一些祕寶。透過機器學習，你可以完成以下及其他工作（*https://homl.info/usecases*）：

- 細分顧客，找出對每一個族群而言最好的行銷策略。
- 根據性質相似的顧客的購買情況，為每位顧客推薦相應的產品。
- 找出有詐欺可能的交易。
- 預測你的明年收入。

不管你的理由是什麼，你已經決定學習機器學習，並在專案中實現它了。這是很棒的想法！

目標和方法

本書假設你幾乎不懂機器學習，其目的是告訴你一些概念、工具，以及直覺，教你寫出可在資料中進行學習的程式。

我們將探討大量的技術，從最簡單的到最常用的（例如線性回歸）、到一些經常贏得比賽的深度學習技術。為此，我們將使用以下這些準生產 Python 框架：

- Scikit-Learn（*https://scikit-learn.org*）很容易使用，它也高效地實作了許多機器學習演算法，所以很適合當成學習機器學習的起點。它是 David Cournapeau 在 2007 年創作的，目前由法國國立計算機科學及自動化研究院（Inria）的一個研究團隊領導。
- TensorFlow（*https://tensorflow.org*）是較複雜的程式庫，其用途是進行分散式數值運算，可以高效地訓練及執行龐大的神經網路，將計算工作分配給上百個多 GPU（圖形處理單元）伺服器。TensorFlow（TF）是 Google 創造的，最初被用來支援 Google 的許多大規模機器學習應用程式，後來在 2015 年 11 月開放原始碼，在 2019 年 9 月發表 2.0 版。
- Keras（*https://keras.io*）是高階深度學習 API，可讓你非常輕鬆地訓練與運行神經網路。Keras 與 TensorFlow 同捆，依靠 TensorFlow 來執行所有的密集計算。

本書傾向實踐性質，透過可執行的具體範例和一點點理論，來幫助你直觀地瞭解機器學習。

第四章

訓練模型

到目前為止，我們都將機器學習模型與它們的訓練演算法當成黑盒子。如果你做了前幾章的習題，你應該會驚訝地發現，你不需要瞭解太多底層的知識就可以做很多事情了：你優化了一個回歸系統、改善了一個數字圖像分類器，甚至從零開始建立一個垃圾郵件分類器，完全不需要知道它們的實際動作。其實，在許多情況下，你不需要真正知道實作細節。

但是，瞭解事物如何運作可以協助你快速找出合適的模型、使用正確的訓練演算法，以及找到適合任務的超參數。瞭解底層的東西也可以協助你更高效地進行除錯，以及執行錯誤分析。最後，本章討論的主題大都是瞭解、建構、訓練神經網路（在本書的第二部分討論）必備的部分。

在本章中，我們會先瞭解線性回歸模型，它是最簡單的模型之一。我們將討論兩種全然不同訓練法：

- 直接使用「閉合形式」方程式[1]，算出讓模型最擬合訓練組（也就是可將訓練組的代價函數最小化）的模型參數。
- 使用一種稱為梯度下降（Gradient Descent，GD）的迭代優化法來逐步調整模型參數，來將訓練組的代價函數最小化，最終收斂成與第一種方法一樣的參數組。我們會看一些梯度下降的變體：批次 GD、小批次 GD 與隨機 GD。我們將在第二部分學習神經網路時反覆使用它們。

1 閉合形式方程式僅由有限數量的常數、變數和標準運算組成，例如 $a = \sin(b - c)$。它沒有無窮的總和、極限、積分…等。

接著我們要瞭解多項式回歸，它是比較複雜的模型，可擬合非線性的資料組。因為這種模型的參數比線性回歸更多，所以它比較容易過擬訓練資料，所以我們將探討如何使用學習曲線來檢測有沒有發生這種情況，然後研究一些降低過擬訓練組風險的正則化技術。

最後，我們還要探討兩種經常用來執行分類任務的模型：logistic（邏輯）回歸與 softmax 回歸。

本章有許多數學公式，它們將使用線性代數與微積分的基本概念。為了瞭解這些公式，你必須知道什麼是向量與矩陣、如何轉置它們、對它們執行乘法、取逆，以及什麼是偏導數。如果你不瞭解這些概念，請參考線上輔助教材的 Jupyter notebook 中的線性代數與微積分課程（*https://github.com/ageron/handson-ml3*）。如果你不喜歡數學，你還是要看這一章，但可以跳過公式，希望文字的部分足以幫助你理解大部分的概念。

線性回歸

我們在第 1 章看過生活滿意度的回歸模型：

$$life\_satisfaction = \theta_0 + \theta_1 \times GDP\_per\_capita$$

這個模型只是輸入特徵 `GDP_per_capita` 的線性函數。θ_0 與 θ_1 是模型的參數。

更廣泛地說，線性模型進行預測的方式，是計算輸入特徵的加權總和，再加上一個稱為**偏差項**（也稱為**截距項**）的常數，如公式 4-1 所示。

公式 *4-1*　線性回歸模型預測

$$\hat{y} = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n$$

在這個公式中：

- $\hat{y}$ 是預測值。
- n 是特徵的數量。
- x_i 是第 i 個特徵的值。
- θ_j 是第 j 個模型參數（包括偏差項 θ_0 與特徵權重 $\theta_1, \theta_2, \cdots, \theta_n$）。

你可以用更簡潔的向量形式來表示它，見公式 4-2。

公式 *4-2*　線性回歸模型預測（向量形式）

$$\hat{y} = h_{\boldsymbol{\theta}}(\mathbf{x}) = \boldsymbol{\theta} \cdot \mathbf{x}$$

在這個公式中：

- $h_{\boldsymbol{\theta}}$ 是假設函數，使用模型參數 $\boldsymbol{\theta}$。
- $\boldsymbol{\theta}$ 是模型的**參數向量**，包含偏差項 θ_0 與特徵權重 θ_1 到 θ_n。
- $\mathbf{x}$ 是實例的**特徵向量**，包含 x_0 到 x_n，x_0 一定等於 1。
- $\boldsymbol{\theta} \cdot \mathbf{x}$ 是向量 $\boldsymbol{\theta}$ 與 $\mathbf{x}$ 的內積，等於 $\theta_0 x_0 + \theta_1 x_1 + \theta_2 x_2 + \ldots + \theta_n x_n$。

在機器學習中，向量通常被稱為行（*column*）**向量**，是一個具有一個欄位的 2D 陣列。如果 $\boldsymbol{\theta}$ 與 $\mathbf{x}$ 是行向量，那麼預測值是 $\hat{y} = \boldsymbol{\theta}^\intercal \mathbf{x}$，其中 $\boldsymbol{\theta}^\intercal$ 是 $\boldsymbol{\theta}$ 的**轉置**（列向量，而不是行向量），而 $\boldsymbol{\theta}^\intercal \mathbf{x}$ 是 $\boldsymbol{\theta}^\intercal$ 與 $\mathbf{x}$ 的矩陣乘積。當然它是同一個預測結果，只是這次是以單格（cell）矩陣來表示的，而不是純量值。在本書中，我將使用這種寫法來避免在內積與矩陣乘法之間切換。

OK，它就是線性回歸模型，那麼，我們該如何訓練它？之前提過，訓練一個模型的意思，就是設定它的參數，使它最擬合訓練組。為此，我們要先評估模型擬合訓練資料的程度有多好（或多糟）。第 2 章說過，最常見的回歸模型效能指標是均方根誤差（RMSE）（公式 2-1）。因此，為了訓練線性回歸模型，我們要先找到可將 RMSE 最小化的 $\boldsymbol{\theta}$ 值。在實務上，與 RMSE 相較之下，將均方誤差（MSE）最小化不但比較簡單，也可以得到相同的結果（因為可將正函數最小化的值，也可將函數的平方根最小化）。

在訓練期間，學習演算法通常會優化一個損失函數，這個損失函數與用來評估最終模型的效能指標不一樣。這通常是因為函數比較容易優化，而且（或者）它有一些項僅在訓練期間使用（例如用來正則化）。好的效能指標應盡可能地接近最終的商業目標。好的訓練損失容易優化，而且與指標高度相關。例如，分類器通常用對數損失（log loss，稍後介紹）等代價函數來進行訓練，但是用 precision/recall 來進行評估。對數損失很容易最小化，而最小化通常可以改善 precision/recall。

訓練組 $\mathbf{X}$ 的線性回歸假設 $h_{\boldsymbol{\theta}}$ 的 MSE 是用公式 4-3 來計算的。

公式 *4-3*　線性回歸模型的 *MSE* 代價函數

$$\mathrm{MSE}(\mathbf{X}, h_{\boldsymbol{\theta}}) = \frac{1}{m}\sum_{i=1}^{m}\left(\boldsymbol{\theta}^{\mathsf{T}}\mathbf{x}^{(i)} - y^{(i)}\right)^2$$

我們已經在第 2 章介紹大部分的符號了（見第 43 頁的「符號」）。唯一的差異在於，我們使用 $h_{\boldsymbol{\theta}}$ 而非只是 h，來表明這個模型是用向量 $\boldsymbol{\theta}$ 來參數化的。為了簡化符號，我們用 $\mathrm{MSE}(\boldsymbol{\theta})$ 來取代 $\mathrm{MSE}(\mathbf{X}, h_{\boldsymbol{\theta}})$。

正規方程式

我們可以使用**封閉形式解**（*closed-form solution*）來找出將代價函數最小化的 $\boldsymbol{\theta}$ 值，換句話說，封閉形式解就是可以直接給出結果的數學公式，它稱為**正規方程式**（*Normal Equation*）（公式 4-4）。

公式 *4-4*　正規方程式

$$\widehat{\boldsymbol{\theta}} = \left(\mathbf{X}^{\mathsf{T}}\mathbf{X}\right)^{-1}\ \mathbf{X}^{\mathsf{T}}\ \mathbf{y}$$

在這個公式中：

- $\widehat{\boldsymbol{\theta}}$ 是可將代價函數最小化的 $\boldsymbol{\theta}$ 值。
- $\mathbf{y}$ 是目標值向量，裡面有 $y^{(1)}$ 至 $y^{(m)}$。

我們來製作一些看似線性的資料，用它們來測試這個方程式（圖 4-1）：

```
import numpy as np

np.random.seed(42)  # 為了讓這段範例程式可以復現
m = 100  # 實例的數量
X = 2 * np.random.rand(m, 1)  # 行向量
y = 4 + 3 * X + np.random.randn(m, 1)  # 行向量
```

接下來，我們用正規方程式來計算 $\widehat{\boldsymbol{\theta}}$。我們使用 NumPy 的線性代數模組（`np.linalg`）的 `inv()` 函式來計算逆矩陣，並使用 `dot()` 方法來做矩陣乘積：

```
from sklearn.preprocessing import add_dummy_feature

X_b = add_dummy_feature(X)  # 將 x0 = 1 加至每個實例
theta_best = np.linalg.inv(X_b.T @ X_b) @ X_b.T @ y
```

`@` 運算子執行矩陣乘法。如果 `A` 和 `B` 是 NumPy 陣列，那麼 `A @ B` 等同於 `np.matmul(A, B)`。許多其他程式庫，例如 TensorFlow、PyTorch 和 JAX，也支援 `@` 運算子。然而，你無法對著純 Python 陣列（即，串列的串列）使用 `@`。

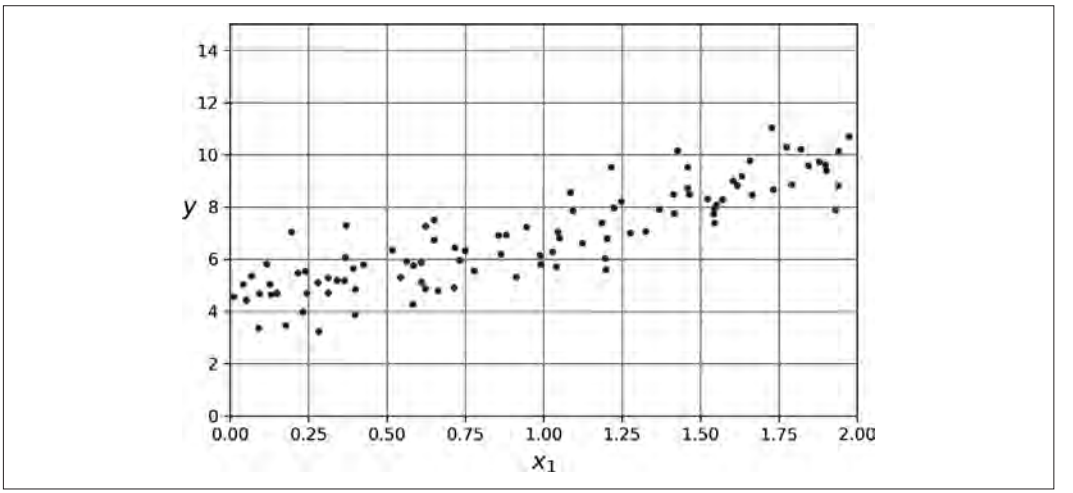

圖 4-1　隨機生成的線性資料組

我們用來產生資料的函數是 $y = 4 + 3x_1$ + 高斯（Gaussian）雜訊。來看一下方程式找到什麼：

```
>>> theta_best
array([[4.21509616],
       [2.77011339]])
```

我們本來希望 $\theta_0 = 4$ 且 $\theta_1 = 3$ 而不是 $\theta_0 = 4.215$ 且 $\theta_1 = 2.770$。雖然兩者很接近，但雜訊使得我們無法恢復原始函數的精確參數。資料組越小且雜訊越大，就越難恢復。

現在我們可以用 $\widehat{\boldsymbol{\theta}}$ 來進行預測：

```
>>> X_new = np.array([[0], [2]])
>>> X_new_b = add_dummy_feature(X_new)  # 為每個實例加上 x0 = 1
>>> y_predict = X_new_b @ theta_best
>>> y_predict
array([[4.21509616],
       [9.75532293]])
```

我們來畫出這個模型的預測（圖 4-2）：

```
import matplotlib.pyplot as plt

plt.plot(X_new, y_predict, "r-", label="Predictions")
plt.plot(X, y, "b.")
[...]  # 美化圖表，加入標籤、座標軸、網格、圖例
plt.show()
```

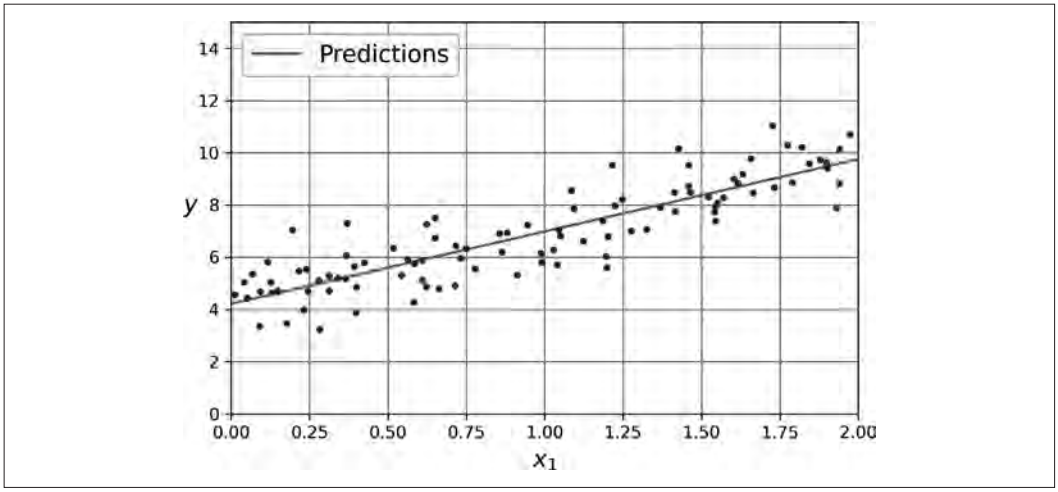

圖 4-2　線性回歸模型預測

使用 Scikit-Learn 來執行線性回歸相對簡單：

```
>>> from sklearn.linear_model import LinearRegression
>>> lin_reg = LinearRegression()
>>> lin_reg.fit(X, y)
>>> lin_reg.intercept_, lin_reg.coef_
(array([4.21509616]), array([[2.77011339]]))
>>> lin_reg.predict(X_new)
array([[4.21509616],
       [9.75532293]])
```

注意，Scikit-Learn 將偏差項（`intercept_`）從特徵權重（`coef_`）分離出來。`LinearRegression` 類別以 `scipy.linalg.lstsq()` 函式為基礎（它的名字代表「least squares（最小平方）」），你可以直接呼叫它：

```
>>> theta_best_svd, residuals, rank, s = np.linalg.lstsq(X_b, y, rcond=1e-6)
>>> theta_best_svd
array([[4.21509616],
       [2.77011339]])
```

這個函式計算 $\hat{\boldsymbol{\theta}} = \mathbf{X}^{+}\mathbf{y}$，其中 $\mathbf{X}^{+}$ 是 $\mathbf{X}$ 的**偽逆**（*pseudoinverse*）（具體來說，它是 Moore-Penrose 逆矩陣）。你可以用 `np.linalg.pinv()` 來直接計算偽逆：

```
>>> np.linalg.pinv(X_b) @ y
array([[4.21509616],
       [2.77011339]])
```

偽逆本身是用一種稱為**奇異值分解**（*Singular Value Decomposition*，SVD）的標準矩陣分解技術來計算的，這種技術可將訓練組矩陣 $\mathbf{X}$ 分解成三個矩陣 $\mathbf{U}\ \mathbf{\Sigma}\ \mathbf{V}^{T}$ 的矩陣相乘（見 `numpy.linalg.svd()`）。算出來的偽逆是 $\mathbf{X}^{+} = \mathbf{V}\mathbf{\Sigma}^{+}\mathbf{U}^{T}$。在計算矩陣 $\mathbf{\Sigma}^{+}$ 時，演算法接收 $\mathbf{\Sigma}$，並將小於某個極小閾值的值都設為零，然後將所有非零值換成它們的逆值，最後將產生的矩陣轉置。這種做法比計算正規方程式更高效，而且可以正確地處理極端案例：事實上，如果矩陣 $\mathbf{X}^{T}\mathbf{X}$ 是不可逆的（奇異的），正規方程式可能沒有用，例如當 $m < n$ 時，或有些特徵是多餘的時，但是偽逆絕對是有定義的（defined）。

計算複雜度

正規方程式計算的是 $\mathbf{X}^{T}\ \mathbf{X}$ 的逆矩陣，它是個 $(n + 1) \times (n + 1)$ 矩陣（其中 n 是特徵數量）。計算這個逆矩陣的複雜度通常大約在 $O(n^{2.4})$ 至 $O(n^{3})$ 之間，依具體的實作而定。換句話說，如果特徵的數量翻倍，計算時間要乘上大約 $2^{2.4} = 5.3$ 至 $2^{3} = 8$。

Scikit-Learn 的 `LinearRegression` 類別使用的 SVD 技術大約是 $O(n^{2})$。如果特徵數量翻倍，計算時間大約提升 4 倍左右。

當特徵數量變得很大時（例如 100,000 個），正規方程式與 SVD 技術都會變得非常慢。從正面看，它們都與訓練組的實例數量成線性關係（它們是 $O(m)$），因此只要記憶體可以容納大型的訓練組，它們就可以有效地處理大型訓練組。

此外，當你訓練好線性回歸模型之後（使用正規方程式或任何其他演算法），預測的速度會很快：計算複雜度與你想預測的實例數量及特徵數量成線性關係。換句話說，對兩倍的實例（或兩倍的特徵）進行預測所花費的時間大約也是兩倍。

接下來要介紹一種非常不同的線性回歸模型訓練法，它更適合有太多特徵，或太多訓練實例，而無法全部放入記憶體的情況。

梯度下降

梯度下降（*Gradient Descent*）是一種通用的優化演算法，它能夠找出許多問題的最佳解。梯度下降的基本概念是藉著反覆微調參數來將代價函數最小化。

假如你在起霧的深山裡迷路了，你只能感受腳底下的斜坡。為了快速走到山谷，有一種好方法是沿著最陡的斜坡向下走，這也是梯度下降的做法，它會計算誤差函式對參數向量 **θ** 的局部梯度，並且朝著梯度下降的方向前進。當梯度變成零時，你就到達最小值了！

在實務上，你可以先將 **θ** 設成隨機值（這稱為**隨機初始化**），然後逐步改善它，每次一小步，每一步都試著降低代價函數（例如 MSE），直到演算法收斂至最小值為止（見圖 4-3）。

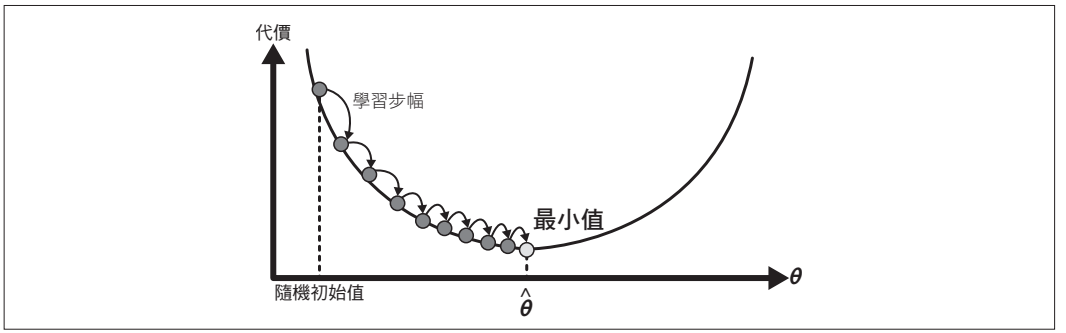

圖 4-3　在這張梯度下降圖中，模型參數的初始值都是隨機的，並且被反覆調整來將代價函數最小化；學習步幅與代價函數的斜率成正比，所以步幅會隨著參數趨近最小值而越來越小

在梯度下降中，步幅是很重要的參數，它是由**學習速度**（*learning rate*）超參數決定的。如果學習速度太小，演算法就得經歷多次反覆運算才能收斂，這將花費很久的時間（見圖 4-4）。

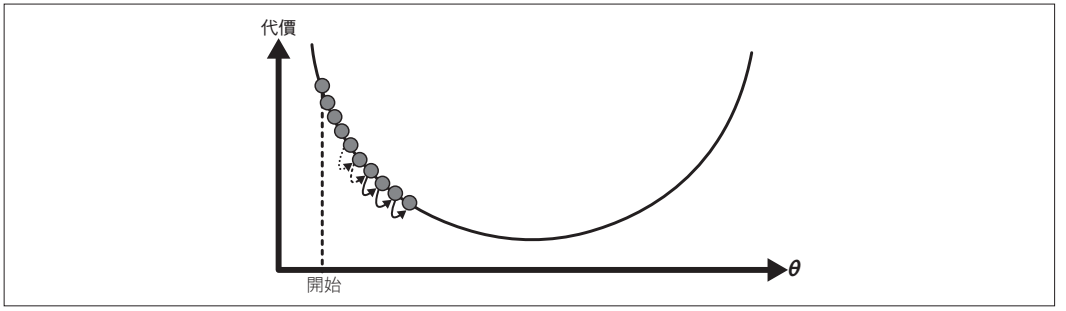

圖 4-4　學習速度太小

另一方面，學習速度太高可能會直接跳過山谷到另一邊，甚至可能跳到更高的位置，導致演算法發散，值越來越大，最終無法找到好的解（見圖 4-5）。

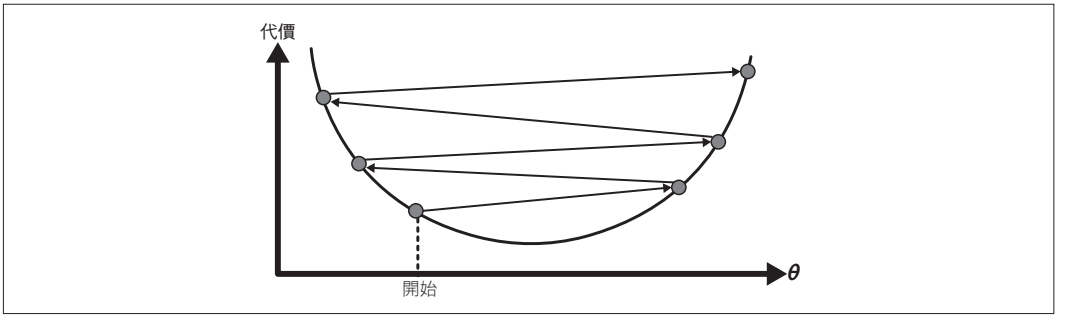

圖 4-5　學習速度太高

最後，並非所有代價函數看起來都像個漂亮的、常規的碗狀。它可能有坑洞、山嶺、高原，和各種不規則的地形，難以收斂至最小值。圖 4-6 展示梯度下降的兩大挑戰。如果隨機初始值從演算法的左邊開始，它會收斂至局部最小值，此值不像全域最小值那麼好。如果從右邊開始，它要花很長的時間來穿越平穩期，太早停止的話，永遠無法到達全域最小值。

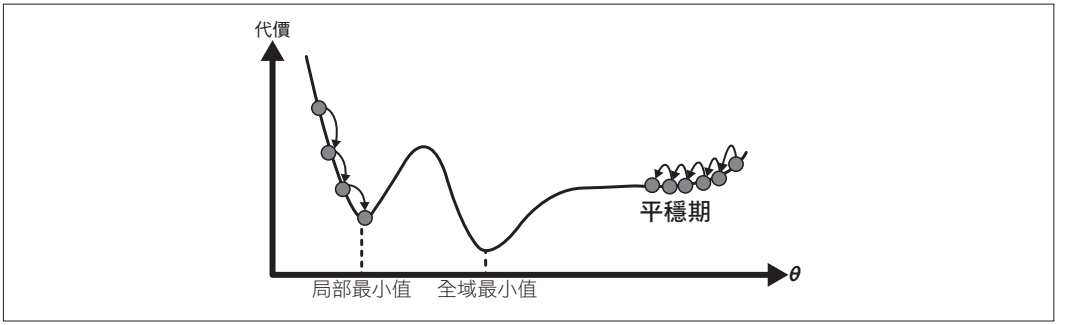

圖 4-6　梯度下降的陷阱

幸運的是，線性回歸模型的 MSE 代價函數剛好是個**凸函數**（*convex function*），凸函數的意思是，選擇曲線上的任意兩點畫一條直線的話，那條直線絕對不會與曲線交叉。凸函數意味著這條曲線沒有局部最小值，只有全域最小值。凸函數也是一個連續函數，其斜率永

遠不會突然變化[2]。這兩個特性有重大的意義，它們意味著梯度下降一定能夠非常接近全域最小值（如果你等待的時間夠久，而且學習速度不高）。

儘管代價函數長得像一個碗，但如果特徵的尺度有很大的差異，它可能成為一個橢圓形的碗。圖 4-7 展示了在特徵 1 和特徵 2 有相同尺度的訓練組上進行梯度下降（左）的情況，以及在特徵 1 的值比特徵 2 小得多的訓練組上進行的梯度下降（右）的情況[3]。

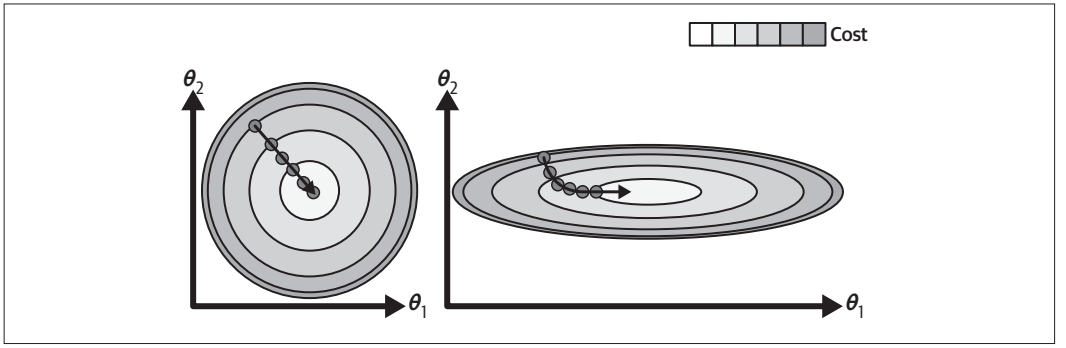

圖 4-7　進行了特徵尺度調整的梯度下降（左邊）與沒有進行特徵尺度調整（右邊）的梯度下降

如你所見，左圖的梯度下降演算法直接朝著最小值的方向前進，因此迅速到達最小值，右圖則先朝著與全域最小值方向幾乎垂直的方向前進，最終沿著幾乎平坦的山谷進行長時間的下降，雖然它最終也到達最小值，但花了很長的時間。

在使用梯度下降時，你要確保所有特徵都有相似的尺度（例如使用 Scikit-Learn 的 `StandardScaler` 類別），否則收斂的時間會長很多。

這張圖也說明一個事實：訓練模型，就是找出一組可將代價函數最小化的模型參數（使用訓練組來尋找）。它是在模型的**參數空間**裡進行搜尋。模型的參數越多，空間的維數越多，搜尋就越困難：在 300 維的乾草堆中尋找一根細針，比在 3 維空間裡困難很多。幸好，由於線性回歸的代價函數是凸的，針就在碗底。

2　嚴格說來，它的導數是 *Lipschitz continuous*。

3　因為特徵 1 比較小，所以 θ_1 必須有大很多的變化才能影響代價函數，這就是為什麼碗形沿著 θ_1 軸被延伸的原因。

批次梯度下降

為了實作梯度下降，你必須計算代價函數相對於每個模型參數 θ_j 的梯度。換句話說，你要計算微幅改變 θ_j 時，代價函數會改變多少。這種計算稱為**偏導數**，它就像詢問「如果我朝東，那麼我腳下的山坡的斜率是多少？」，然後詢問朝北的同一個問題（對所有其他維度也是如此，如果你能夠想像維度超過三的空間長怎樣的話）。公式 4-5 計算 MSE 對於參數 θ_j 的偏導數，寫成 $\partial\,\mathrm{MSE}(\boldsymbol{\theta}) / \partial\theta_j$。

公式 *4-5*　代價函數的偏導數

$$\frac{\partial}{\partial\theta_j}\mathrm{MSE}(\boldsymbol{\theta}) = \frac{2}{m}\sum_{i=1}^{m}\left(\boldsymbol{\theta}^{\intercal}\mathbf{x}^{(i)} - y^{(i)}\right)x_j^{(i)}$$

你只要使用公式 4-6 就可以一次算出這些偏導數了，不需要分別計算它們。梯度向量（寫成 $\nabla_{\boldsymbol{\theta}}\mathrm{MSE}(\boldsymbol{\theta})$）包含代價函數的所有偏導數（每個模型參數一個）。

公式 *4-6*　代價函數的梯度向量

$$\nabla_{\boldsymbol{\theta}}\,\mathrm{MSE}(\boldsymbol{\theta}) = \begin{pmatrix} \frac{\partial}{\partial\theta_0}\mathrm{MSE}(\boldsymbol{\theta}) \\ \frac{\partial}{\partial\theta_1}\mathrm{MSE}(\boldsymbol{\theta}) \\ \vdots \\ \frac{\partial}{\partial\theta_n}\mathrm{MSE}(\boldsymbol{\theta}) \end{pmatrix} = \frac{2}{m}\mathbf{X}^{\intercal}(\mathbf{X}\boldsymbol{\theta} - \mathbf{y})$$

注意，這個公式是對著整個訓練組 **X** 進行計算，並且在各個梯度下降步驟都是如此！這就是它被稱為**批次梯度下降**的原因：它在每一個步驟都使用一整批訓練資料（事實上，稱為**全梯度下降**應該更合適）。因此，當它處理龐大的訓練組時，速度將極度緩慢（但我們很快就會看到快很多的梯度下降演算法）。但是，梯度下降法可以隨著特徵的數量而擴展，使用梯度下降法來訓練具有數十萬個特徵的線性回歸模型的速度，比使用正規方程式或 SVD 分解還要快很多。

一旦取得指往上坡方向的梯度向量，你只要朝著相反的方向前進，即可朝著下坡方向前進。這意味著將 $\boldsymbol{\theta}$ 減去 $\nabla_{\boldsymbol{\theta}}\mathrm{MSE}(\boldsymbol{\theta})$。這就是學習速度 η 發揮作用的地方[4]：我們將梯度向量乘以 η 來決定下坡步幅的大小（公式 4-7）。

4　Eta (η) 是希臘字母的第七個字母。

公式 *4-7* 梯度下降步幅

$$\theta^{(\text{next step})} = \theta - \eta \nabla_{\theta} \text{MSE}(\theta)$$

我們來看一下如何快速實作這個演算法：

```
eta = 0.1  # 學習速度
n_epochs = 1000
m = len(X_b)  # 實例的數量

np.random.seed(42)
theta = np.random.randn(2, 1)  # 隨機初始化模型參數

for epoch in range(n_epochs):
    gradients = 2 / m * X_b.T @ (X_b @ theta - y)
    theta = theta - eta * gradients
```

這並不難！針對訓練組的每一次迭代都稱為一 *epoch*（期）。來看看算出來的 `theta` 是多少：

```
>>> theta
array([[4.21509616],
       [2.77011339]])
```

這正是正規方程式找到的值！梯度下降的表現很完美，但使用不同的學習速度（`eta`）會怎樣？圖 4-8 是使用三個不同學習速度的梯度下降的前 20 步。在每張圖最下面的那條線代表隨機起點，我們用越來越深的線條來表示每一個 epoch。

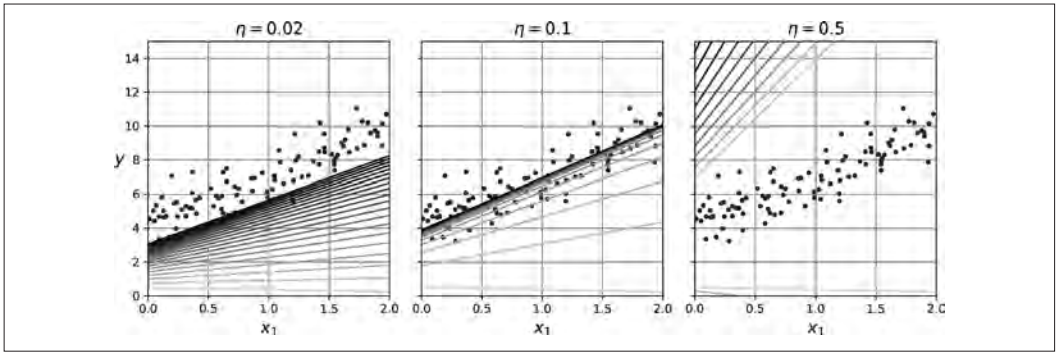

圖 4-8　使用各種學習速度的梯度下降

左圖的學習速度太低了，雖然演算法最終會到達解，但是它花很長的時間。中圖的學習速度看起來不錯，只用幾個 epoch 就收斂到解了。右圖的學習速度太高，導致演算法發散了，到處亂跳，每一步都離解越來越遠。

你可以用網格搜尋來找出好的學習速度（見第 2 章）。但是，你可能要限制 epoch 的次數，使網格搜尋法可排除收斂時間太長的模型。

你可能想知道如何設定 epoch 數？如果它太低，當演算法停止時，你仍然離最佳解很遠，但是如果它太高，你會浪費時間，因為模型參數再也不會改變。比較簡單的做法是設定很大的 epoch 數，但是在梯度向量變得很小時中斷演算法，也就是當它的範數（norm）變得小於一個很小的數字 ϵ（稱為**容限數**（*tolerance*））時，因為這會在梯度下降（幾乎）到達最小值的時候發生。

收斂速度

當代價函數是凸函數而且斜率不會突然變化（就像 MSE 代價函數的情況）時，使用固定學習速度的批次梯度下降最終會收斂至最佳解，但可能需要一段時間：根據代價函數的形狀，它能需要做 $O(1/\epsilon)$ 次迭代，才能到達在 ϵ 的範圍內的最佳解。如果你將容限數除以 10 來取得更精確的解，演算法可能會執行大約 10 倍的時間。

隨機梯度下降

批次梯度下降的主要問題是它在每一個步驟都使用整個訓練組來計算梯度，如此一來，當訓練組很大時，它的速度將十分緩慢。**隨機梯度下降**（*Stochastic Gradient Descent*）是另一個極端的做法，它會在每一個步驟從訓練組中隨機選出一個實例，並且僅用該實例來計算梯度。一次使用一個實例當然可讓演算法的速度快很多，因為它在每一次迭代時需要處理的資料少非常多。這種做法也讓我們有機會使用巨型的訓練組來進行訓練，因為在每次迭代時，記憶體只需要容納一個實例（隨機 GD 可以用外存（out-of-core）演算法來實現，見第 1 章）。

另一方面，由於這種演算法的隨機性質，它比批次梯度下降還要不規律許多：它的代價函數不會平穩地下降至最小值，而是會上下波動，只是平均而言是下降的。久而久之，它將非常接近最小值，但到達最小值後，它會繼續上下波動，永遠不會停止（見圖 4-9）。所以當演算法停止時，雖然最終參數值是好的，但不是最好的。

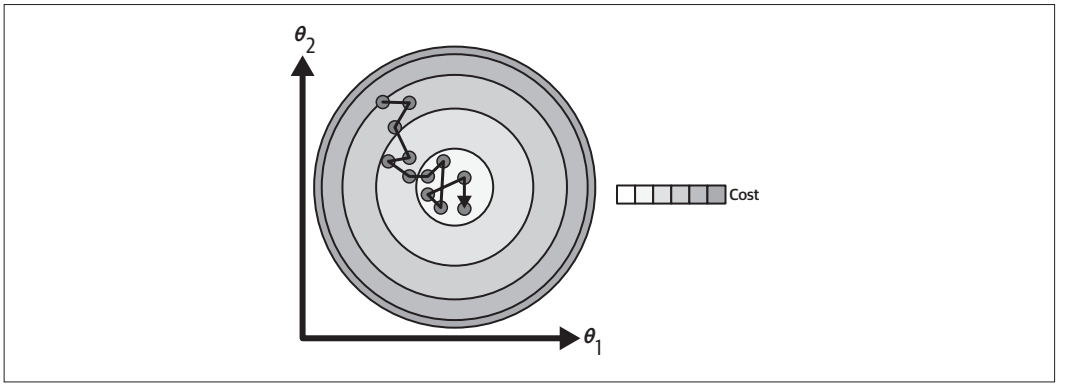

圖 4-9　隨機梯度下降的每一個訓練步驟都快很多，但也比批次梯度下降還要隨機

當代價函數非常不規則時（像圖 4-6 那樣），隨機性可幫助演算法跳出局部最小值，因此隨機梯度下降比批次梯度下降更有機會找到全域最小值。

隨機性有助於跳脫局部最佳值，但它也有壞處，因為它意味著演算法絕不會停留在最小值。有一種解決這個兩難的辦法，就是逐漸降低學習速度，最初使用大步幅（有助於快速前進並跳脫局部最小值），然後使用越來越小的步幅，使演算法停在全域最小值。這個程序類似**模擬退火**（*simulated annealing*）。模擬退火是受金屬熱處理中的退火（讓熔化的金屬慢慢冷卻）啟發的演算法。在每次迭代時決定學習速度的函數稱為**學習規劃**（*learning schedule*）。如果學習速度減少得太快，你可能會被困在局部最小值，甚至停在前往最小值的半路上。如果學習速度減少得太慢，你可能會在最小值附近跳躍很久，如果訓練太早結束，你可能只會得到次佳解。

下面的程式使用簡單的學習規劃來實作隨機梯度下降：

```
n_epochs = 50
t0, t1 = 5, 50  # 學習規劃超參數

def learning_schedule(t):
    return t0 / (t + t1)

np.random.seed(42)
theta = np.random.randn(2, 1)  # 隨機初始化

for epoch in range(n_epochs):
    for iteration in range(m):
        random_index = np.random.randint(m)
        xi = X_b[random_index : random_index + 1]
        yi = y[random_index : random_index + 1]
```

```
gradients = 2 * xi.T @ (xi @ theta - yi)  # 執行 SGD 時，不除以 m
eta = learning_schedule(epoch * m + iteration)
theta = theta - eta * gradients
```

按慣例，我們執行包含 *m* 次迭代的多個回合，每一個回合稱為一 *epoch*。批次梯度下降程式對整個訓練組迭代了 1,000 次，但這段程式只遍歷訓練組 50 次就得到相當不錯的答案：

```
>>> theta
array([[4.21076011],
       [2.74856079]])
```

圖 4-10 是訓練的前 20 步（注意每一步有多麼不規則）。

請注意，因為實例是隨機選出的，有些實例可能被各個 epoch 選出多次，有些則未被選過。如果你想要確保演算法在每個 epoch 都遍歷每一個實例，另一種做法是洗亂訓練組（務必一併洗亂輸入特徵與標籤），再遍歷每一個實例，然後再洗亂它，以此類推。然而，這種做法比較複雜，而且通常無法改善結果。

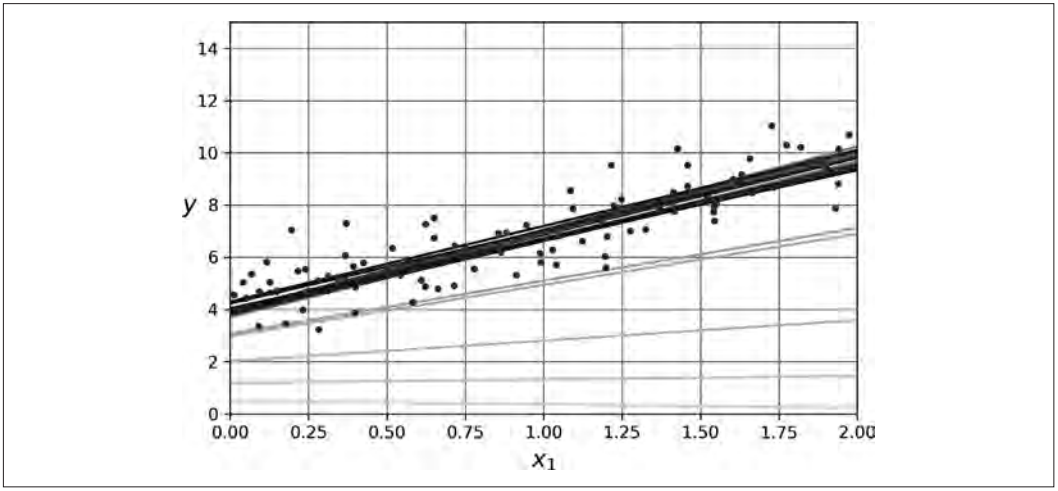

圖 4-10　隨機梯度下降的前 20 步

在使用隨機梯度下降時，訓練實例必須獨立且具有相同分布（independent and identically distributed，IID），以確保平均而言，參數會被牽引至全域最佳解。確保這一點的一種簡單方法，是在訓練期間對實例進行洗牌（例如，隨機選擇每個實例，或在每個 epoch 開始時，將訓練組洗亂）。如果不將實例洗亂（舉例來說，如果實例依標籤排序），那麼 SGD 將先優化一個標籤，然後優化下一個，以此類推，並且不會靠近全域最小值。

若要用 Scikit-Learn 的隨機 GD 來執行線性回歸，你可以使用 `SGDRegressor` 類別，它預設優化 MSE 代價函數。下面這段程式最多執行 1,000 epoch（`max_iter`），或在連續 100 個 epoch（`n_iter_no_change`）期間，損失下降不到 10^{-5}（`tol`）時停止執行。它的初始學習速度是 0.01（`eta0`），使用預設的學習規劃（與前面的不同）。最後，它沒有使用任何正則化（`penalty=None`，稍後詳述）：

```
from sklearn.linear_model import SGDRegressor

sgd_reg = SGDRegressor(max_iter=1000, tol=1e-5, penalty=None, eta0=0.01,
                       n_iter_no_change=100, random_state=42)
sgd_reg.fit(X, y.ravel())  # 使用 y.ravel() 是因為 fit() 期望收到 1D 目標
```

同樣地，它產生的解相當接近正規方程式回傳的解：

```
>>> sgd_reg.intercept_, sgd_reg.coef_
(array([4.21278812]), array([2.77270267]))
```

所有的 Scikit-Learn 估計器都可以使用 `fit()` 方法來訓練，但有些估計器也有 `partial_fit()` 方法，你可以呼叫它來用一個或多個實例執行一輪訓練（它忽略 `max_iter` 和 `tol` 等超參數）。反覆呼叫 `partial_fit()` 方法可逐漸訓練模型。這很適合在你想要更仔細地控制訓練過程時使用。其他模型則有一個 `warm_start` 超參數（有些模型兩者都有）：若設定 `warm_start=True`，那麼對著訓練過的模型呼叫 `fit()` 方法將不會重設模型，而是從上次停止的地方繼續訓練，並考慮 `max_iter` 和 `tol` 等超參數。請注意，`fit()` 會重設學習規劃所使用的迭代計數器，而 `partial_fit()` 方法不會重設。

小批次梯度下降

我們要看的最後一個梯度下降演算法稱為**小批次梯度下降**（*Mini-batch Gradient Descent*）。當你瞭解批次與隨機梯度下降之後，你可以輕鬆地理解這種演算法：小批次梯度下降的每一步都是用隨機抽取的一小組實例（稱為**小批次**（*mini-batches*））來計算梯度，而不是用完整的資料組（就像批次 GD 那樣）或一個實例（就像隨機 GD 那樣）來計算梯度。小批次 GD 相對於隨機 GD 的主要優勢在於，有一些硬體擅長進行矩陣計算，你可以利用那些硬體來提升效能，尤其是在使用 GPU 時。

小批次梯度下降在參數空間裡的軌跡不像隨機梯度下降那麼顛簸，尤其是在使用相當大的小批次時。因此，小批次 GD 最終會在比隨機 GD 更靠近最小值的地方徘徊，但它可能更難從局部最小值跳脫（在被局部最小值困擾的問題中，不像使用 MSE 代價函數的線性回

歸那樣）。圖 4-11 展示這三種梯度下降演算法在訓練期間於參數空間中的路徑。它們最後都很接近最小值，但是批次 GD 的路徑會停在最小值，而隨機 GD 與小批次 GD 會持續徘徊。但是，別忘了，批次 GD 的每一步都花了大量的時間，如果你使用優良的學習規劃，隨機 GD 與小批次 GD 也可以到達最小值。

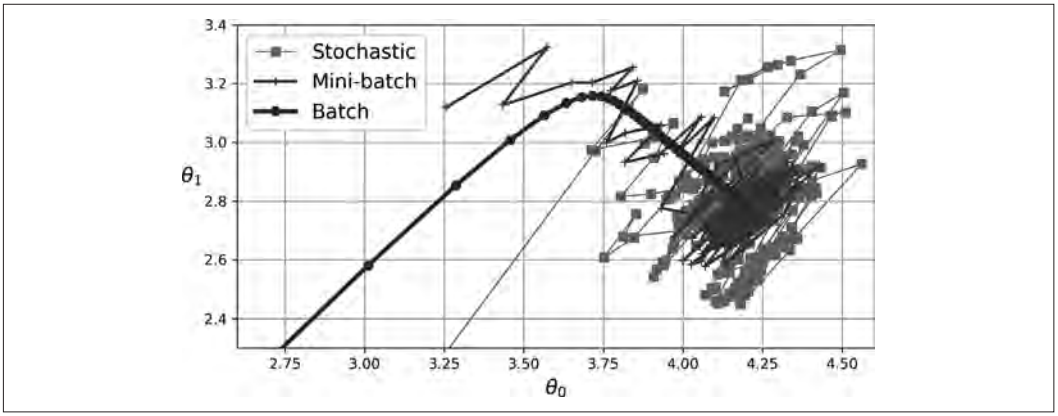

圖 4-11　梯度下降在參數空間裡的路徑

表 4-1 對迄今為止討論過的線性回歸算法進行比較[5]（*m* 是訓練實例的數量，*n* 是特徵的數量）。

表 4-1　比較線性回歸的演算法

演算法	大 *m*	支援外存	大 *n*	超參數	需要調整尺度	Scikit-Learn
正規方程式	快	否	慢	0	否	N/A
SVD	快	否	慢	0	否	`LinearRegression`
批次 GD	慢	否	快	2	是	N/A
隨機 GD	快	是	快	≥2	是	`SGDRegressor`
小批次 GD	快	是	快	≥2	是	N/A

這些演算法在訓練後幾乎沒有差異，它們最終都會產生相似的模型，並以相同的方式進行預測。

5　正規方程式只能執行線性回歸，但等一下會看到，梯度下降演算法也可以用來訓練許多其他模型。

多項式回歸

如果資料比較複雜，不僅僅是一條線呢？出人意外的是，你可以用線性模型來擬合非線性資料。其中一種簡單的做法是加入各個特徵的次方來作為新特徵，然後用這組擴展後的特徵來訓練線性模型。這種技術稱為**多項式回歸**。

我們來看一個例子。首先，我們使用簡單的**二次方程式**（例如 $y = ax^2 + bx + c$ 的方程式）和一些雜訊來產生一些非線性資料（見圖 4-12）：

```
np.random.seed(42)
m = 100
X = 6 * np.random.rand(m, 1) - 3
y = 0.5 * X ** 2 + X + 2 + np.random.randn(m, 1)
```

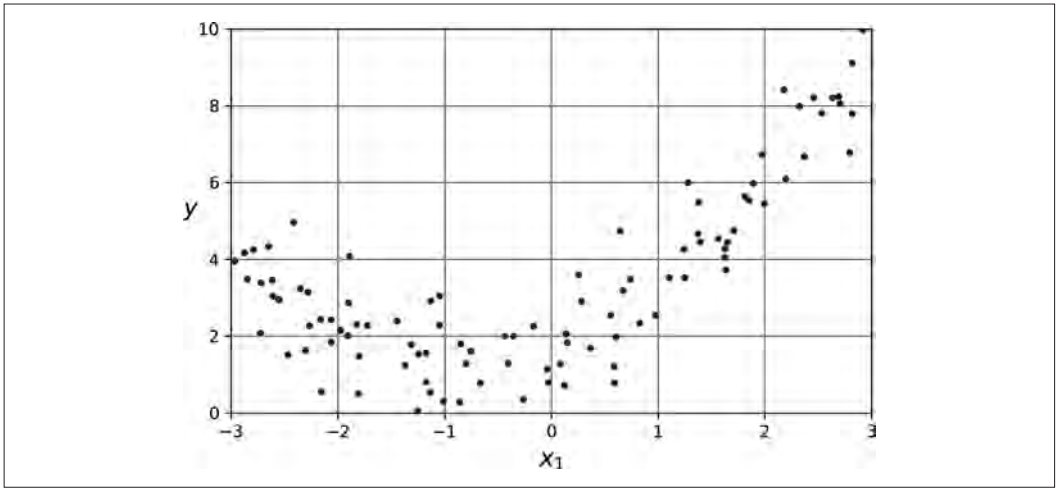

圖 4-12　生成一個非線性且有雜訊的資料組

顯然，直線不可能緊密擬合這筆資料。所以我們用 Scikit-Learn 的 `PolynomialFeatures` 類別來轉換訓練資料，在訓練組裡面加入各個特徵的平方（二次多項式）來作為新特徵（這個例子只有一個特徵）：

```
>>> from sklearn.preprocessing import PolynomialFeatures
>>> poly_features = PolynomialFeatures(degree=2, include_bias=False)
>>> X_poly = poly_features.fit_transform(X)
>>> X[0]
array([-0.75275929])
>>> X_poly[0]
array([-0.75275929, 0.56664654])
```

現在，`X_poly` 裡面不但有原始的 `X` 特徵，還有該特徵的平方。現在你可以將 `LinearRegression` 模型擬合至這個擴展過的訓練資料（圖 4-13）：

```
>>> lin_reg = LinearRegression()
>>> lin_reg.fit(X_poly, y)
>>> lin_reg.intercept_, lin_reg.coef_
(array([1.78134581]), array([[0.93366893, 0.56456263]]))
```

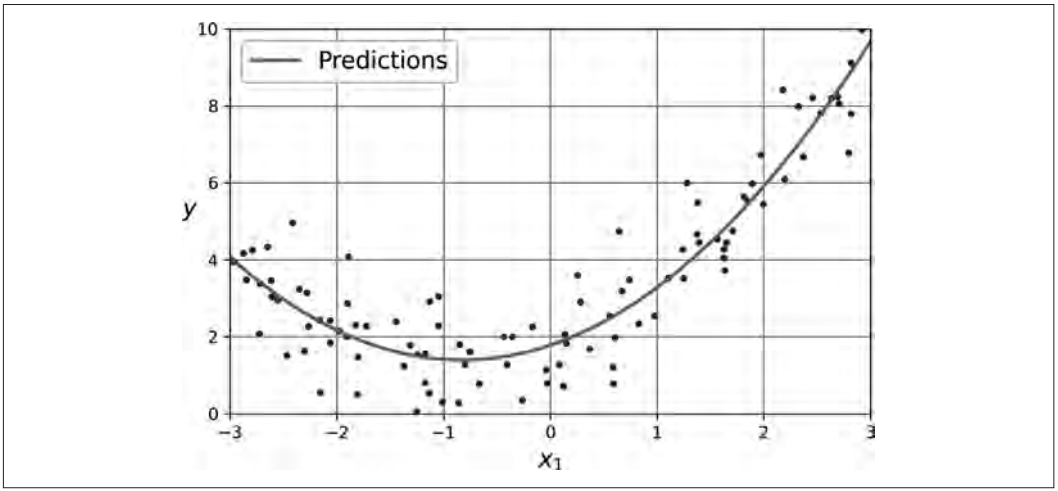

圖 4-13　多項式回歸模型預測

還不賴，這個模型估計 $\hat{y} = 0.56x_1^2 + 0.93x_1 + 1.78$，而原始的函數是 $y = 0.5x_1^2 + 1.0x_1 + 2.0 +$ 高斯雜訊。

注意，有多個特徵時，多項式回歸能夠找出特徵之間的關係，而普通線性回歸模型沒辦法做到，它能夠如此的原因是 `PolynomialFeatures` 也會加入所有特徵組合，直到給定的次方。例如，如果有兩個特徵 a 與 b，設定 `degree=3` 的 `PolynomialFeatures` 不但會加入特徵 a^2、a^3、b^2 與 b^3，也會加入 ab、a^2b 與 ab^2 等組合。

`PolynomialFeatures(degree=d)` 會將一個包含 n 個特徵的陣列轉換成一個包含 $(n + d)! / d!n!$ 個特徵的陣列，其中 $n!$ 是 n 的階乘，等於 $1 \times 2 \times 3 \times \cdots \times n$。小心特徵數量組合爆炸（combinatorial explosion）！