让"高高在上"的AI技术回归供人们学习和使用的层面

基于代码理解机器学习和深度学习在实际应用中的意义

程序员的AI书

从代码开始

张力柯　潘　晖　编著

让“高高在上”的AI技术回归供人们学习和使用的层面

基于代码理解机器学习和深度学习在实际应用中的意义

程序员的AI书

从代码开始

张力柯　潘　晖　编著

作者简介

张力柯

腾讯某AI实验室负责人、AI系统设计专家。在操作系统内核、网络安全、搜索引擎、推荐系统、大规模分布式系统、图像处理、数据分析等领域具有丰富的实践经验。

于美国德克萨斯大学圣安东尼奥分校获得计算机科学博士学位，曾先后在美国微软、BCG、Uber及硅谷其他创业公司担任研发工程师及项目负责人等。

潘　晖

阿里巴巴某算法中心小组负责人。在推荐系统、自然语言处理、图像处理、数据分析等领域具有丰富的实践经验。

于美国佛罗里达理工大学获得计算机科学博士学位，曾先后在中国微软、美团、腾讯从事算法研发和管理工作。发表过多篇论文，拥有多项专利，曾获得2018年腾讯互动娱乐事业群技术突破奖等奖项。

程式設計師的AI書從程式碼開始

張力柯　潘　暉　編著

電子工業出版社
Publishing House of Electronics Industry
北京・BEIJING

內容簡介

隨著AI技術的普及，如何快速理解、掌握並應用AI技術，成為絕大多數程式設計師亟需解決的問題。本書基於Keras框架並以程式碼實現為核心，詳細解答程式設計師學習AI演算法時的常見問題，對機器學習、深度神經網路等概念在實際專案中的應用建立清晰的邏輯體系。

本書分為上下兩篇，上篇（第1～4章）可幫助讀者理解並獨立開發較簡單的機器學習應用，下篇（第5～9章）則聚焦於AI技術的三大熱點領域：推薦系統、自然語言處理（NLP）及影像處理。其中，第1章透過具體例項對Keras的機器學習實現進行快速介紹並給出整體概念；第2章從簡單的神經元開始，以實際問題和程式碼實現為引導，逐步過渡到多層神經網路的具體實現上，從程式碼層面講解神經網路的工作模式；第3章講解Keras的核心概念和使用方法，幫助讀者快速入門Keras；第4章講解機器學習中的常見概念、定義及演算法；第5章介紹推薦系統的常見方案，包括協同過濾的不同實現及Wide＆Deep模型等；第6章講解迴圈神經網路（RNN）的原理及Seq2Seq、Attention等技術在自然語言處理中的應用；第7～8章針對影像處理的分類及目標識別進行深度討論，從程式碼層面分析Faster RCNN及YOLO v3這兩種典型識別演算法；第9章針對AI模型的工程部署問題，引入TensorFlow Serving並進行介紹。

本書主要面向希望學習AI開發或者轉型演算法的程式設計師，也可以作為Keras教材，幫助讀者學習Keras在不同領域的具體應用。

未經許可，不得以任何方式複製或抄襲本書之部分或全部內容。

版權所有，侵權必究。

圖書在版編目（CIP）資料

程式設計師的AI書：從程式碼開始 / 張力柯，潘暉編著.—北京：電子工業出版社，2020.2

ISBN 978-7-121-38270-3

I.①程… II.①張…②潘… III.①人工智慧－程式設計 IV.①TP18

中國版本圖書館CIP資料核字（2020）第014136號

責任編輯：張國霞　特約編輯：顧慧芳

印　　刷：三河市君旺印務有限公司

裝　　訂：三河市君旺印務有限公司

出版發行：電子工業出版社

北京市海淀區萬壽路173信箱　郵編100036

開　　本：787×980　1/16　印張：20　字數：430千字

版　　次：2020年2月第1版

印　　次：2020年2月第1次印刷

印　　數：5000冊　定價：109.00元

凡所購買電子工業出版社圖書有缺損問題，請向購買書店調換。若書店售缺，請與本社發行部聯絡，聯絡及郵購電話：（010）88254888，88258888。

質量投訴請發郵件至zlts@phei.com.cn，盜版侵權舉報請發郵件至dbqq@phei.com.cn。

本書諮詢聯絡方式：010-51260888-819，faq@phei.com.cn。

推薦序一

認識力柯兄多年，一直認為他是一員虎將——能用程式碼說話，便絕不打無謂的嘴仗；能用技術與產品直接證明，便絕不空談「形式」和「主義」。這次，透過力柯兄寫的這本書，一如既往地看到他「心有猛虎」的一面：直截了當、大開大合。

在機器學習或者說工業界AI火起來的這幾年，程式設計師這一受眾群體一直缺少優秀的教程。有些教程過於淺顯，很難稱其為「入門教程」，只能稱其為科普書；有些教程則過於貼近理論推導，對夯實讀者的數理基礎大有裨益，也能給研究生提供參考，但對程式設計師來說，則理論有餘而實踐不足，常常令注重工程實踐的他們一頭霧水：畢竟不是每個程式設計師都有耐心、有必要一門一門地撿回微積分、機率統計、偏微分方程、線性代數和數值計算等基礎學科的知識，再真正實現一個屬於自己的模型。如何在數學理論和工程實踐之間找到一個平衡點，讓具有工程背景的廣大讀者從中獲得實際的價值，而非進行簡單的腦力或數學訓練，這一直是我評價機器學習教程時最為看重的要素。現在，我有幸從力柯兄的成書中找到了這些要素，實乃幸事！

這是一本寫給程式設計師看的機器學習指南。它有針對性地從程式設計師的視角切入（而非像市面上的大多數機器學習教程一樣，從數學的角度切入），介紹了工業界流行的若干模型及應用場景，同時涵蓋了神經網路的原理和基礎實現、Keras庫的使用方法和TensorFlow的部署方案，可謂有的放矢。另外，本書章節不多，卻簡短有力。這不是一本科普讀物，不存在淺嘗輒止；也不是一本百科全書，不存在天書符號。這是一本有程式碼的書，是一本談工程實現的書。我認為，這正是機器學習領域所缺少的那一類教程。

本書的上篇，讓我不由得想起多年前力柯兄剛從矽谷回國高就時，與我圍繞「怎樣的面試題對於機器學習程式設計師是合適的」這一話題展開的討論。那時AI正在升溫，無數有著各種背景、能力和水平的人都在嘗試接觸AI方面的內容，但對於人才的選拔和錄用，卻似

乎沒有一個行業內的公認標準和規範。力柯兄的面試題十分簡單粗暴，要求面試者僅使用一些基礎的Python庫去實現一個深度神經網路。這聽起來有點讓人匪夷所思，但事後細想，卻是大道至簡。這可以讓人拋去繁雜的模型，迴歸神經網路最本質的前向傳播和反向傳播，將一切都落實在程式碼層面。雖然需要運用的數學知識不過是一點高等數學的皮毛，卻可以同時從工程和數學兩個角度考察候選人的基本功。這幾年間，機器學習和深度學習教程及相關公開課越來越多，我閱課無數，竟發現很少有一門課能夠沉下心來，仔仔細細地告訴讀者和學員，搭建和實現這些神經網路的基礎元素從何而來，又為何如此。而本書的上篇，尤其是在第2章中，一絲不苟地介紹了神經元、啟用函式和損失函式，從偏微分方程層面嚴謹地推導反向傳播，又從程式碼層面給出了那道面試題的答案。這都讓我不由得敬佩力柯兄在工程上的執著。本書的下篇，則是標準的深度學習入門。

至此，我不再「劇透」，因為當你從實戰角度閱讀這些章節時，會有一種不斷髮現珍寶的驚喜感，而我更願意把這些「珍寶」留給本書的讀者。

周竟舸，Pinterest機器學習平臺技術負責人

2019年12月

推薦序二

近十餘年，計算機領域中令人矚目的亮點就是以深度學習為代表的一系列突破。無論是人臉檢測還是影像識別，抑或文字翻譯或無人駕駛，這些在過去幾十年裡讓電腦科學家苦苦思索卻不能解的種種難題，在深度學習的幫助下，竟被一一攻克，這不能不說是人類科技史上一顆耀眼的明星。

AI技術的突飛猛進，卻使傳統程式設計師產生不少困惑：過去常用的資料結構、排序搜尋、連結串列陣列等，現在變成了模型、卷積、權重和啟用函式……無論是要開發AI應用，還是和演算法研究人員共同工作，他們都存在同一個問題：如何學習AI技術？如何理解AI演算法人員常用的名詞和概念？更重要的是，如何把AI相關的程式碼和自己的軟體開發經驗聯絡起來？

現在市面上已經有很多深度學習和機器學習教程了，其中也不乏從例項入手、以程式碼實現為重點的書籍，但並沒有一本書真正地從程式設計師的視角來看待深度學習技術。或者，我們也可以這麼說，大部分相關書籍的重點是講解深度學習理論，所用的例項是解釋深度學習理論的實際應用。儘管有不少書籍在講解理論和程式碼時詳盡而深入，卻沒有涉及核心問題：要解決這個問題，為什麼非用深度學習或機器學習不可？沒有這些方法就不能做嗎？用深度學習處理該問題的優勢是什麼？是十全十美，還是存在問題？

開啟本書，我驚喜地發現它並非像市面上的其他書籍那樣，直接把各種新鮮概念放到讀者面前並強迫他們接受。它一開始就沒有在機器學習概念上過多糾纏，而是先快速展示了簡短的AI實現程式碼的結構和流程，然後帶出一些常常讓初學者疑惑的問題，針對這些問題再帶出新的內容。我們可以看到，本書每一章都用到了類似的形式：闡述一個領域中的實際問題，提出不同的解決方法，簡要探討不同的方法，找到人們難以解決的問題，然後解釋機器學習或深度學習處理這些問題的原理。讀者瞭解到的並非單純的機器學習理論，而是不同領

域的具體技術挑戰和相關演算法的解決方案，從而理解機器學習的真正意義。

必須要說的是，作者在美國工作多年，養成了求真務實和獨立思考的習慣，我們從書中能感受到他獨特的風格，並有愉悅的閱讀體驗。本書在理論講解方面也沒有概念堆砌的枯燥無味，作者常常加入一些對技術的調侃和個人見解，以供讀者思考。在程式碼解析階段，作者著眼於整體框架與流程，把重點放在理論中的網路結構如何在實際程式碼中實現，而不會浪費篇幅在程式碼的語言細節上。

閱讀本書，不但是對不同領域AI開發的學習，也是一次以資深程式設計師的視角去審視相關程式碼實現的體驗。本書無論是對於應用開發程式設計師，還是對於演算法研究人員，都相當有價值，非常值得閱讀。

喻傑博士，華為智慧車雲技術長

2019年12月

推薦序三

隨著AlphaGo在人機大戰中一舉成名，關於機器學習的研究開始備受人們關注。機器學習和神經網路已經被廣泛應用於網際網路的各個方面，例如搜尋、廣告、無人駕駛、智慧家居，等等。AI井噴式發展的主要動力如下。

其一，資料的積累。各大IT公司都擁有了自己的資料平臺，資料積累的速度越來越快。各大高校針對不同的機器學習任務，積累了多樣化的資料集。

其二，計算機效能指數級的增長。從當初的CPU到GPU，再從GPU到專門為AI設計的晶片，都提供了強大而高效的平行計算能力，大大推動了AI演算法的進步。

其三，AI理論及模型的突破，例如卷積網路、長短期記憶等。

其四，深度學習開源框架日趨完善。TensorFlow是當前領先的深度學習開源框架，越來越多的人在使用它從事計算機視覺、自然語言處理、語音識別和一般性的預測分析工作。TensorFlow整合的Keras是為人類而非機器設計的API，易於學習和使用。

這是一本非常適合程式設計師入門和實踐深度學習的書，理論和實踐並重，使用Keras作為機器學習框架，側重於AI 演算法實現。

本書以從程式碼出發，再回歸AI相關原理為宗旨，深入淺出、循序漸進地講解了Keras及常見的深度學習網路，還講解了深度學習在不同領域的應用及模型的部署與服務。讀者在一步步探索AI演算法奧秘的同時，也在享受解決問題的喜悅和成就感，並開啟深度學習之旅。

衷心地希望有志於AI學習的讀者抓住機會，早做準備，成為AI時代的弄潮兒。

王昀績，Google AI高階研究員

2019年12月

上篇

第1章　機器學習的Hello World

機器學習作為近年來的熱點技術，不但是彙集傳統統計學、資料探勘、平行計算、大資料等多個領域的交叉學科，也對傳統的程式設計開發方式形成了一定衝擊，整個開發的模式、過程及思考角度與傳統的程式碼實現有相當大的差別。從某個角度來說，很多軟體工程師在接觸機器學習時，所面臨的最大困難並不是對其概念和原理無法理解，而是難以轉換自己的程式設計思維方式，從傳統的面向具體邏輯流程的實現，變為面向資料和結果擬合的實現。因此，我們在這裡改變傳統的機器學習入門方式，先從機器學習程式碼的開發和使用入手，讓讀者對如何用機器學習的方式解決問題有直觀的瞭解。

本章作為入門級的內容，內容精簡。讀者應備好電腦，嘗試執行自己的第1個機器學習程式。

1.1　機器學習簡介

機器學習主要有三大類別：監督學習（Supervised Learning）、無監督學習（Unsupervised Learning）、增強學習（Reinforcement Learning），下面對這三大類別進行簡要介紹。

1.監督學習

監督學習是機器學習中應用最廣泛也最可靠的技術。簡單來說，監督學習的目的就是透過標註好的資料進行模型訓練，從而期望利用訓練好的模型對新的資料進行預測或分類。在這裡，「監督」（Supervised）這個詞意味著我們已經有標註好的已知資料集。

監督學習的應用場景非常廣泛，常見的垃圾郵件過濾、房價預測、圖片分類等都是適合它的領域，但其最大弱點就是需要大量標註資料，前期投入成本極高。

2.無監督學習

相對於需要大量標註資料的監督學習，無監督學習無須標註資料就能達到某個目標。注意，並不是所有場景都適合採用無監督學習，無監督學習經常被用於以下兩方面。

◎　聚類（Clustering）：在聚類場景下使用無監督學習的頻率可能是最高的。例如給出一堆圖片，把相似的圖片劃分在一起。我們既可以預設一個類別總數進行自動劃分（即半監督學習，Semi-supervised）；也可以預設一個差異閾值，然後對所有圖片進行自動聚類。

◎　降維（Dimensionality Reduction）：在資料特徵過多、維度過高時，我們通常需要把高維資料降到合理的低維空間處理，並期望保留最重要的特徵資料。主成分分析（Principal Component Analysis，PCA）就是其中最為常見的演算法應用。

3.增強學習

無論是監督學習還是無監督學習，其訓練基礎都來源於資料本身。而增強學習最大的特點就是需要與環境有某種互動關係，這也促使人們在增強學習的研究中利用類似電子遊戲的環境來模擬互動並進行AI訓練。例如，DeepMind在2015年提出的利用DQN學習ATARI遊戲的操作，以及OpenAI的Gym等。

增強學習的實現和應用場景比較特殊，儘管某些大型公司已經在推薦系統、動態定價等場景中嘗試應用增強學習，但仍只限於實驗性質，有興趣的讀者可以自行閱讀其他資料進行學習，在本書中不對增強學習進行講解。

1.2　機器學習應用的核心開發流程

我們經常聽到機器學習的研究人員開口「特徵」，閉口「模型」，也聽過他們調侃自己是「調參」師，然而，他們口中的這些術語到底指什麼呢？若想了解這些術語，就先要清楚機器學習應用開發的核心流程。

圖1-1把機器學習應用的核心開發流程劃分為4個階段，實際上，基本上所有機器學習專案的開發流程都是按照這4個階段來劃分和實施的。

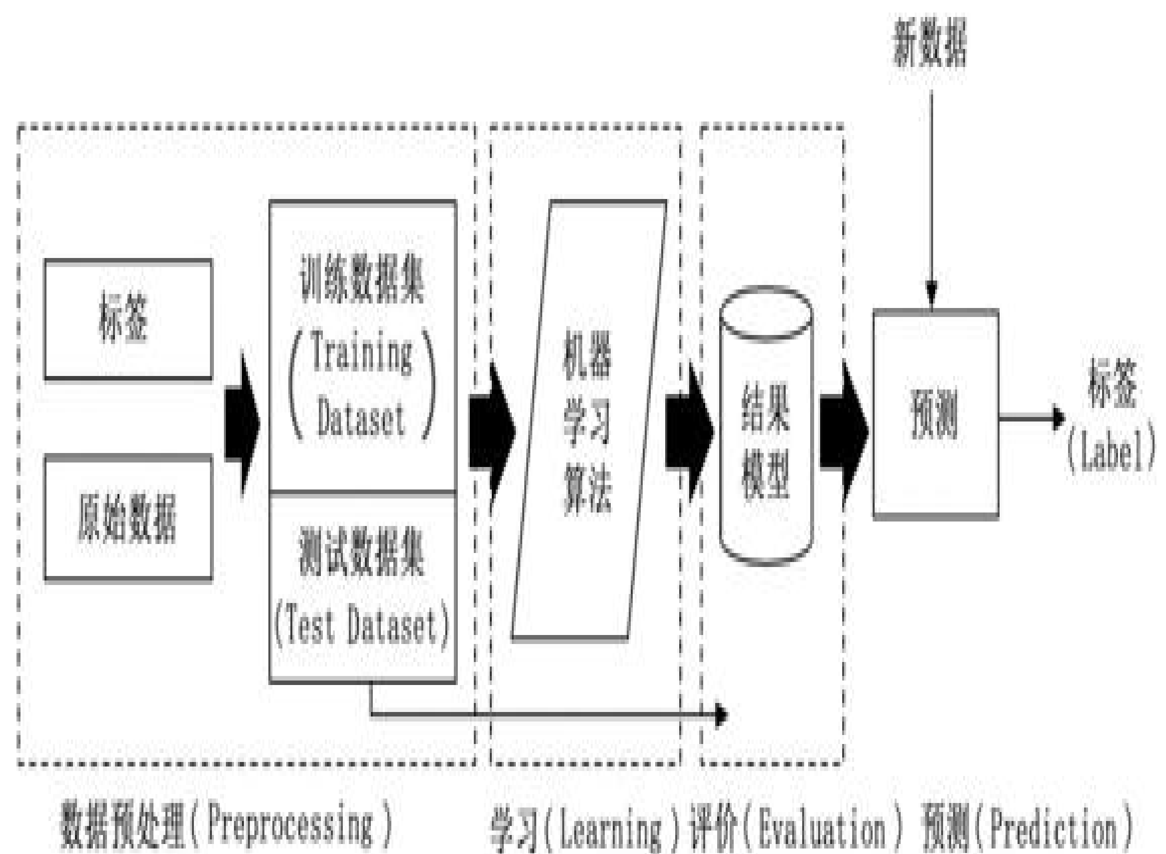


圖1-1　機器學習的基本流程

下面對圖1-1所示的4個階段進行解釋。

1.資料預處理（Preprocessing）

機器學習的第1階段就是處理原始資料。從圖1-1可以看出，我們需要處理帶有標籤的原始資料，形成用於模型訓練的訓練資料集和用於驗證模型效果的測試資料集，包含如下兩項核心工作。

（1）特徵提取。我們要處理的原始資料往往是以多種形式存在的，可以是來自MySQL 資料庫的不同欄位，也可以是原始的文字檔案或影像、影片、音訊等多媒體檔案。然而，絕大部分機器學習演算法的輸入通常是某種浮點數矩陣形式或者向量形式的。把這些多樣化的原始資料轉化為符合演算法資料的資料格式，是工作的第1步，我們通

常可以把這一步稱為「特徵提取」。而從原始資料中挑選出來進行轉換，並最終用於機器學習的數值就被稱為「特徵值」。例如，我們要識別花的種類，從花的圖片中將花瓣顏色、花瓣形狀、花瓣長度、花瓣寬度、葉片長度、葉片形狀等屬性數值化，這些數值就被定義為花的「特徵」。注意，我們在這裡是透過經驗去選取我們覺得重要且有用的特徵值的，因為在現實場景下可能會有成百上千種資料型別（例如一個業務中所有資料表的欄位）備選，我們不太可能將它們全部用於機器學習，必須有所取捨，這就是特徵工程的目的。然而，依賴經驗選擇並不是100%可靠的。在實際工作中，我們需要和業務專家一起不斷討論和嘗試驗證，直到確認我們選取的特徵值的確能達到業務要求。

（2）資料清洗。有時，我們哪怕是把資料轉化為符合演算法輸入的形式，也會出現很多問題，例如，需要輸入的某些特徵可能不存在、不同特徵的數值區間差別太大、某些特徵可能是文字形式等，這時我們需要根據情況處理這些不規範的特徵值。例如，將不存在的特徵設為0或取平均值，對文字形式的特徵進行編碼，或者對數值區間較大的特徵進行歸一化（Normalization）等。資料清洗的目的是讓演算法訓練所用的資料集儘量理想化，不包含不必要的幹擾數值，從而提高模型訓練的精度。

在資料集處理完成後，我們通常需要把資料集劃分為訓練資料集和測試資料集，這樣的劃分通常是隨機的，隨機挑選 80%的資料用於訓練，將剩下的 20%用於測試驗證。資料集其實在不同的機器學習框架中均有對應的API進行快速劃分，這裡不做詳述。

2.學習（Learning）

這個階段也是我們常說的訓練階段。我們已經在資料預處理階段構建了合適的資料集，在這個階段就需要根據自己的最終目標選擇合適的演算法模型，並根據我們的資料集進行合理的引數設定，開始模型訓練。

什麼是模型訓練？我們在中學都學過多元函式，例如：

$$y = ax_1 + bx_2 + cx_3$$

其中，y 就是標籤，在資料集中已經標註好；x_1、x_2、x_3就是前面提到的特徵值。我們可以這樣描述：

$$花的類別=a\times花蕊顏色+b\times花瓣顏色+c\times葉片顏色$$

這裡，y就是花的類別，x_1、x_2、x_3就分別是花蕊顏色、花瓣顏色和葉片顏色。

假設這個公式正確，我們接下來需要做的就是透過訓練集中的資料輸入，把a、b、c反推出來。當然，一般來說我們不可能找到100%完美的a、b、c，使以上演算法得到的結果和訓練集中所有的輸入都一致，所以我們只能讓結果儘量接近原始資料，並設定損失函式（Loss Function），設法使整體誤差最小。在實際場景中，我們通常使用MSE（Mean Squared Error，均方誤差）作為損失函式的誤差計算。

當然，上面的例子非常簡單，所要推導的引數也不多。在實際的深度學習模型中，我們需要推導的引數是以萬甚至百萬為單位的，這些引數通常被稱為「權重」（Weight）並被存為特定格式的檔案（不同框架所存的權重檔案格式各不相同）。我們將在第2章深入接觸深度學習並瞭解其訓練過程。

3.評價（Evaluation）

我們在學習階段將誤差降到足夠小之後，就可以停止訓練，將訓練好的模型用在資料預處理階段生成的測試資料集上驗證效果。因為測試資料集的所有資料都沒有在訓練階段出現過，所以我們可以把測試資料視為「新」資料，用來模擬真實環境的輸入，從而預估模型被部署到真實環境後的效果。

4.預測（Prediction）

我們在評價階段確認模型達到了預期的準確率和覆蓋率（召回率）之後，就可以將模型部署上線。注意，在小規模研究中，我們可以直接使用訓練後的模型；但在真正大流量的產品環境中，我們往往需要使用專門的模型服務框架（如TensorFlow Serving[4]）將模型轉換為專有的格式，並在該框架下進行高效服務。我們將在第9章學習如何在TensorFlow Serving上進行模型的部署與驗證。

1.3　從程式碼開始

「Talk is cheap, show me the code」這句話無論是對於程式開發還是對於機器學習都是適用的。對於一個軟體工程師來說，原理、分析和推導都比不上一段真實的程式碼更容易讓人理解，下面就讓我們直接嘗試用程式碼來解決問題吧。

1.3.1　搭建環境

本節希望讓讀者儘快接觸真實開發環境下的程式碼，因此將使用TensorFlow中的Keras作為開發框架（現在Keras已是TensorFlow的一部分）。此外，我們需要一些額外的Python包作為支援。我們選用了Python 3.7以上版本，並假設讀者已經安裝好Python 3.7。下面執行安裝本章程式碼執行所需依賴庫的命令：

```
pip install tensorflow
pip install numpy scipy pandas matplotlib
```

1.3.2　一段簡單的程式碼

考慮一個簡單的問題：對正負數分類，對正數返回1，對負數返回0。

這用傳統的程式碼實現起來非常簡單，如下所示：

```
def get_number_class(num):
    return 1 if num > 0 else 0
```

但用機器學習該如何實現呢？

下面用TensorFlow自帶的Keras實現對正負數分類：

```
1   import tensorflow as tf
2   from tensorflow.keras.models import Sequential
3   from tensorflow.keras.layers import Dense
4
5   model = Sequential()
6   model.add(Dense(units=8, activation='relu', input_dim=1))
7   model.add(Dense(units=1, activation='sigmoid'))
8   model.compile(loss='mean_squared_error', optimizer='sgd')
9
10  x = [1, 2, 3, 10, 20, -2, -10, -100, -5, -20]
11  y = [1.0, 1.0, 1.0, 1.0, 1.0,  0.0, 0.0, 0.0, 0.0, 0.0]
12  model.fit(x, y, epochs=10, batch_size=4)
13
14  test_x = [30, 40, -20, -60]
15  test_y = model.predict(test_x)
16
17  for i in range(0, len(test_x)):
18    print('input {} => predict: {}'.format(test_x[i], test_y[i]))
```

我們看看這18行程式碼都做了什麼。

第1～3行：引入依賴庫TensorFlow和其自帶的Keras相關庫。

第5～8行：這4行建立了一個簡單的兩層神經網路模型。在第5行定義了一個Sequential型別的網路，顧名思義是按順序層疊的網路。在第6行加入第1層網路，輸入input_dim為1，意味著只有1個輸入（因為

我們只判斷一個數字是否為正數）；但是定義了8個輸出（units=8），這個數字其實是隨意定的，可以視為其中神經元的個數（我們將在第2章解釋什麼是神經元），一般神經元的個數越多，效果越好（訓練時間也越長）。演算法科研人員會耗費大量的時間來確定這些數字，以找到最佳的個數。第7行再疊加一層，接收前面的輸出（8個），但作為最後一層，這次只定義一個輸出（units=1），這個輸出決定了最後的分類結果。第8行對整個模型進行最後的配置，選擇mean_squared_error（平均方差）作為損失函式計算，選擇隨機梯度下降（Stochastic Gradient Descent，SGD）作為梯度最佳化方式。這裡將不從理論角度解釋隨機梯度下降的原理，但在第2章中將透過程式碼來手工實現隨機梯度下降。

第10～12行：設定了兩組資料x和y作為訓練集，其中，x包括10個正數，y包括對應的10個分類結果。可以看到，對於x中的每個正數，y中對應的值都是1.0，負數x[*i*]所對應的y[*i*]則是0。然後我們呼叫model.fit函式進行訓練，設定epochs為10（訓練10次），batch_size為4（每次都隨機挑選4組資料）。

第14～18行：這裡構建了4個輸入資料進行測試並列印結果。我們構建了陣列test_x，該陣列包含4個整數，然後呼叫model.predict方法對test_x進行預測。在第17～18行將列印每個輸入所對應的預測結果。

以上18行程式碼的執行結果如下：

```
10/10 [==============================] - 0s 6ms/sample - loss: 0.2353
Epoch 2/10
10/10 [==============================] - 0s 205us/sample - loss: 0.1688
Epoch 3/10
10/10 [==============================] - 0s 182us/sample - loss: 0.1493
Epoch 4/10
10/10 [==============================] - 0s 175us/sample - loss: 0.1341
Epoch 5/10
10/10 [==============================] - 0s 176us/sample - loss: 0.1234
Epoch 6/10
10/10 [==============================] - 0s 177us/sample - loss: 0.1161
Epoch 7/10
10/10 [==============================] - 0s 171us/sample - loss: 0.1103
Epoch 8/10
```

```
10/10 [==============================] - 0s 173us/sample - loss: 0.1051
Epoch 9/10
10/10 [==============================] - 0s 180us/sample - loss: 0.1013
Epoch 10/10
10/10 [==============================] - 0s 172us/sample - loss: 0.0978
input 30 => predict: [0.95692104]
input 40 => predict: [0.9840995]
input -20 => predict: [0.03060886]
input -60 => predict: [3.1581865e-05]
```

可以看到，在最後的預測結果中，所有正數的預測值都非常接近1，所有負數的預測值都小於0.01。前面解釋過，機器學習是一種機率預測，不可能完全精確，以上結果對正負數做了較為明確的劃分，是可以接受的。

經過上面的程式碼實現，讀者可能仍然對其中涉及的一些內容有些茫然，比如什麼是梯度下降，神經元又是什麼，什麼叫加一層、減一層，具體的引數又是怎樣訓練的，等等，在第2～4章會詳細解釋這些內容。

上述程式碼是基於Keras實現的，Keras是機器學習研究主流的開發框架，封裝了很多常見的演算法和模型。第2章將拋開這些框架，用最基本的Python程式碼從最簡單的神經網路開始，嘗試實現上面的功能。

1.4　本章小結

本章簡明扼要地解釋了機器學習的大致概念和主要實施流程，然後迅速進入實際程式碼階段，說明瞭開發環境所需的工具，並透過一

個簡單的數字分類例項引入了簡明的Keras程式碼實現，對其進行詳細解釋，讓讀者對機器學習的開發有一個初步的感性認識。

但是本章並沒有解釋任何原理如神經元、梯度下降、損失函式、啟用函式等概念。第2章將手工進行程式碼實現，讓讀者對這些概念從軟體工程師的角度去學習和思考。

1.5 本章參考文獻

[1] HBO「Silicon Valley」, https://en.wikipedia.org/wiki/Silicon_Valley_(TV_series)

[2] Google Deepmind,「Human Level control through deep reinforcement learning」, Nature, 2015

[3] OpenAI Gym, https://gym.openai.com/

[4] https://github.com /tensorflow/serving

第2章　手工實現神經網路

第1章快速介紹了機器學習的基本概念和開發流程，並透過一段程式碼展示瞭如何用Keras機器學習框架實現簡單的數字分類。

本章將拋開機器學習框架，用純粹的Python程式碼實現簡單的神經網路，並復現第1章中Keras程式碼的結果。

> 本章以程式碼為主，理論為輔。讀者一定要自行執行書中的程式碼，如果對部分內容不理解，則也不用擔心，本章最後一個程式碼示例會解答所有問題。

2.1　感知器

2.1.1　從神經元到感知器

我們首先要了解神經網路的歷史。

1943年，W.McCullock和W.Pitts發表了一篇討論簡單的人類大腦細胞的論文*A Logical Calculus of the Ideas Immanent in Nervous Activity* [1]，在該論文中將Neuron定義為大腦中相互連線的神經細胞，用於處理和傳遞各種化學訊號和生物電訊號。在該論文中，這樣的神經元被定義為一個簡單的邏輯閘，該邏輯閘接收多個訊號輸入並將其整合到一起，如果訊號疊加值超過某個閾值，該神經元就會輸出訊號到下一個神經元。

1957年，F.Rosenblatt在*The Perceptron, a Perceiving and Recognizing Automaton* [2]一文中提出了感知器（Perceptron）的概念及對應的演算法，該演算法能夠自動學習權重，使得其與輸入的乘積能夠決定一個神經元是否產生輸出。更嚴謹地說，我們可以把這個問題描述為一個二分類問題，把類別定義為1（輸出）和0（不輸出）。然後，我們可以把輸入和權重的乘積進行疊加，最終產生輸出，在輸出

值上可以定義某種轉換函式（Transfer Function）來將結果轉換到我們需要的分類上，這個轉換函式通常又被稱為啟用函式（Activation Function）。整個流程如下所示：

$$w = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_m \end{bmatrix}, x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix}, z = w^T \cdot x = w_1 \cdot x_1 + w_2 \cdot x_2 + \cdots + w_m \cdot x_m$$

$$\emptyset(z) = \begin{cases} 1, z > \text{閾值} \\ 0, \text{其他} \end{cases}$$

其中，我們把w稱為權重，把x稱為輸入特徵，把z稱為網路輸入（NetworkInput），∅则是決定輸出的啟用函式。

由此，我們應該能夠理解為什麼要把φ稱為啟用函式。這是因為對於早期的腦神經研究來說，要麼輸出（1），要麼不輸出（0），如果輸出的話，我們就認為該神經元被啟用了，所以就有了啟用函式的概念。

同時，在w、x、z、φ這幾組資料中可以看到：

◎　在監督訓練階段，w不可知，是需要訓練和學習的目標。我們透過訓練集中已知的x和φ輸出值來反推w的值，這個反推的過程被稱為反向傳播，通常使用梯度下降（Gradient Descent）演算法，這在後面會進行詳細講解。這個過程需要反覆迴圈多次，計算量通常較大，需要大量的計算資源和時間。

◎　在實際執行和預測階段，我們用訓練好的w權重和實時輸入來計算最後的φ輸出值，因為基本上只進行少數幾次矩陣運算，所以

複雜度比反向傳播要低很多。

我們在圖2-1中可以看到基本的感知器工作流程：神經元（Neuron）收到輸入x，輸入x和權重w相乘後疊加合併形成網路輸入；網路輸入被傳送到啟用函式，生成輸出（Output，1或0）。如果在訓練階段，則輸出將被用於和真實輸出進行誤差計算，並依此更新權重w。

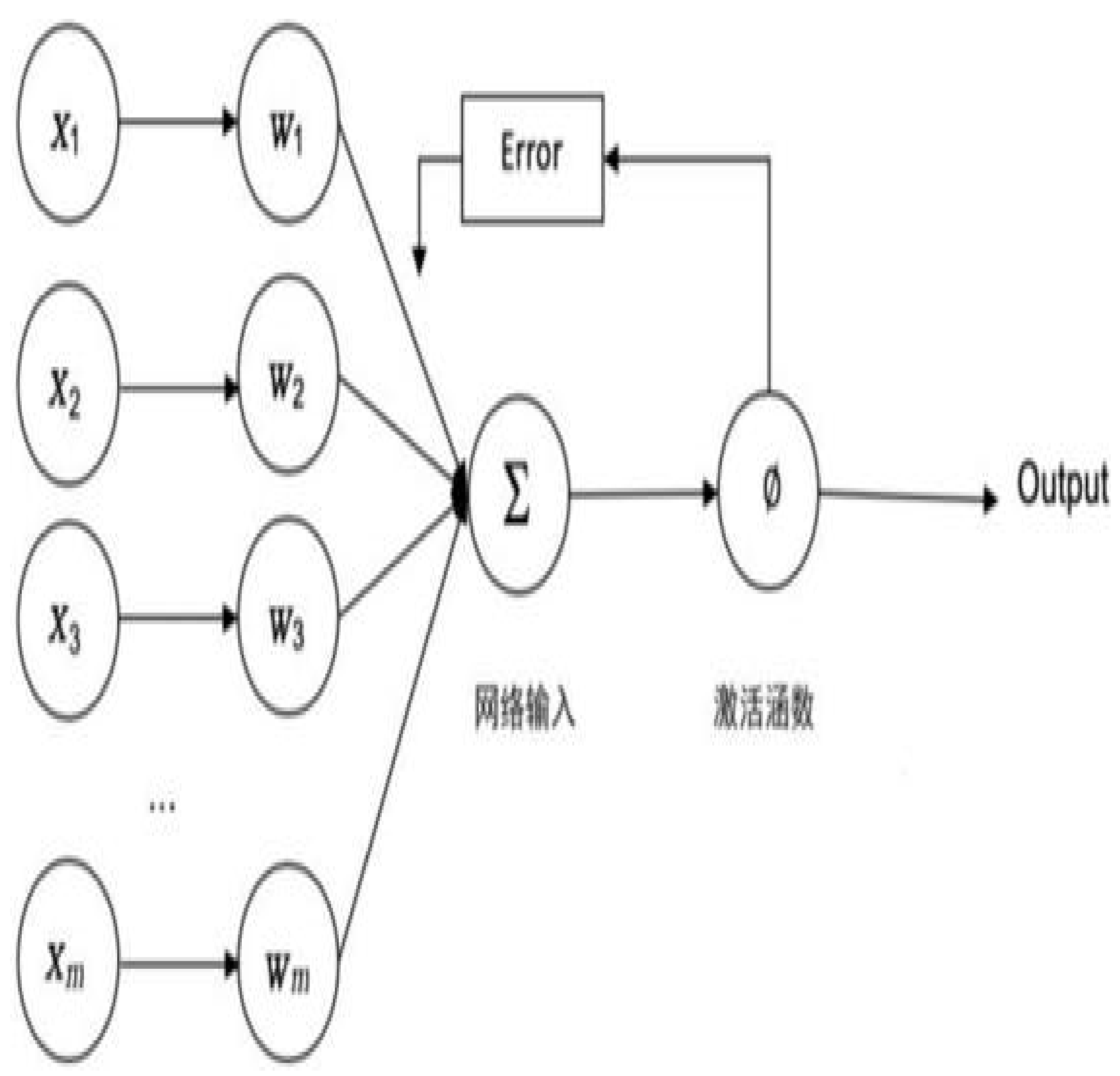


圖2-1　感知器

這裡還沒談到神經網路。到目前為止，我們只需知道神經網路由神經元構成，而感知器指的是神經元的工作方式。要了解神經網路，則首先要了解單個神經元是如何工作的。

2.1.2 實現簡單的感知器

我們現在按照圖2-1的思路來實現一個簡單的感知器。為了展示基本的感知器思路，這段程式碼不涉及任何第三方庫，甚至不使用Numpy。這裡只設一個權重w，並設定一個bias引數。因此network_input為wx+bias（注意，bias也可被視為權重w_0，對應一個輸入為1的常數。把bias視為權重引數之一有利於統一計算），同時，我們使用上面定義的$\emptyset(z)$作為啟用函式，根據網路輸入的值來輸出1或0，這就是最簡單的感知器模型實現。

我們的訓練集和測試資料與第1章的示例相同，用10組資料作為訓練集，用4組陣列作為測試集，整體的程式碼（Simple_perceptron.py）實現如下：

```
1   class Perceptron(object):
2       def __init__(self, eta=0.01, iterations=10):
3           self.lr = eta
4           self.iterations = iterations
5           self.w = 0.0
6           self.bias = 0.0
7
8
9       def fit(self, X, Y):
10          for _ in range(self.iterations):
11              for i in range(len(X)):
12                  x = X[i]
13                  y = Y[i]
14                  update = self.lr * (y - self.predict(x))
15                  self.w += update * x
16                  self.bias += update
17
18
19      def net_input(self, x):
20          return self.w * x + self.bias
21
22
23      def predict(self, x):
24          return 1.0 if self.net_input(x) > 0.0 else 0.0
25
26
27
28  x = [1, 2, 3, 10, 20, -2, -10, -100, -5, -20]
29  y = [1.0, 1.0, 1.0, 1.0, 1.0,  0.0, 0.0, 0.0, 0.0, 0.0]
30
31  model = Perceptron(0.01, 10)
32  model.fit(x, y)
33
34  test_x = [30, 40, -20, -60]
35  for i in range(len(test_x)):
36      print('input {} => predict: {}'.format(test_x[i],
37  model.predict(test_x[i])))
38
39  print(model.w)
40  print(model.bias)
```

我們來看看這些程式碼都做了什麼。

第1～6行：設定初始化引數lr（用於調整訓練時的步長，即learning rate學習率）、iterations（迭代次數）、權重w與bias。

第9～29行：這是模型訓練的核心程式碼。根據iterations的設定，我們用同樣的訓練資料反覆訓練給定的次數，在每一次訓練中都根據每一對資料對引數進行調整，即模型訓練。在訓練集中有真實標籤y，註明當前輸入x的真實類別，然後就可以根據我們設定的y』(預測值)=φ(wx+bias)來獲得y』的預測值。這個預測是透過第23～24行的predict函式實現的，它實際上是根據第19～20行的network_input輸入做出的判斷。

在第14行，我們首先獲得真實值y與預測值y'的偏差，卻並不直接用這個偏差來計算和調整w、bias，而是乘以一個較小的引數lr。我們希望每次都對w和bias進行微調，透過多次迭代和調整後讓w和bias接近最優解（Optimized Value），所以我們希望每次調整的幅度都不要太大。這是一個較難兩全的事情：lr值過小可能會導致訓練時間過長，難以在實際實驗中判斷是否收斂；lr值過大則容易造成步長過大而無法收斂。在第14行獲得需要更新的update值之後，我們可以按照在本章參考文獻[2]中給出的感知器的學習率來計算如何調整w和bias（這裡暫時不去仔細分析這兩個公式是怎麼得來的，因為不同演算法的實現各不相同，我們會在2.2.2節講解梯度下降時再去分析）：

$$\Delta w = lr \cdot (y - y') \cdot x$$

$$\Delta \text{bias} = lr \cdot (y - y')$$

第15～16行：僅僅是每次都把w和bias根據上面的誤差更新進行調整。

那麼，這樣一個簡單模型的執行效果如何呢？我們在第28～39行引入了在第1章中使用的訓練資料和測試資料，看看訓練效果如何。執行該程式碼，我們可以看到以下輸出：

```
input 30 => predict: 1.0
input 40 => predict: 1.0
input -20 => predict: 0.0
input -60 => predict: 0.0
0.01
0.01
```

可以看到，對於輸入的4組資料，執行結果完全正確。

最後兩行輸出的是w和bias的數值。

以上便是神經網路的最初起源及用Python進行的具體實現。當然，這個例子是非常簡單的：①我們只處理單個輸入；②分類也僅僅是很簡單的二分；③我們的啟用函式是一個非常直接的二分輸出，在實際情況下基本不可能這麼設定。但是，這基本解釋了神經網路執行的基本原理和模式。

本章後面幾節將仔細講解基於梯度下降的演算法實現（即如何基於梯度下降來調整權重）和相關的一些概念。當然，正如本書反覆強調的，我們將透過Python程式碼來實現所有這些概念，而不會停留在名詞解析層面。

2.2 線性迴歸、梯度下降及實現

2.2.1 分類的原理

在2.1節中，我們用程式碼實現了簡單的感知器，學習了基本的神經網路原理實現。然而，在其中的程式碼實現裡，我們提到感知器在修正權重w和偏移值bias時的修改規則如下：

$$\Delta w = lr \cdot (y - y') \cdot x$$

$$\Delta \text{bias} = lr \cdot (y - y')$$

這是整個訓練過程中的核心，又是怎麼得來的呢？本節將解釋這個問題。

首先，我們要理解為什麼要定義網路輸入為$wx+b$。

實際上，我們做了一個假設：預測資料是線性可分的，因此我們希望用一個簡單的斜線來判斷所輸入的資料落在哪個區間（類別）。

什麼叫線性可分呢？請看圖2-2。

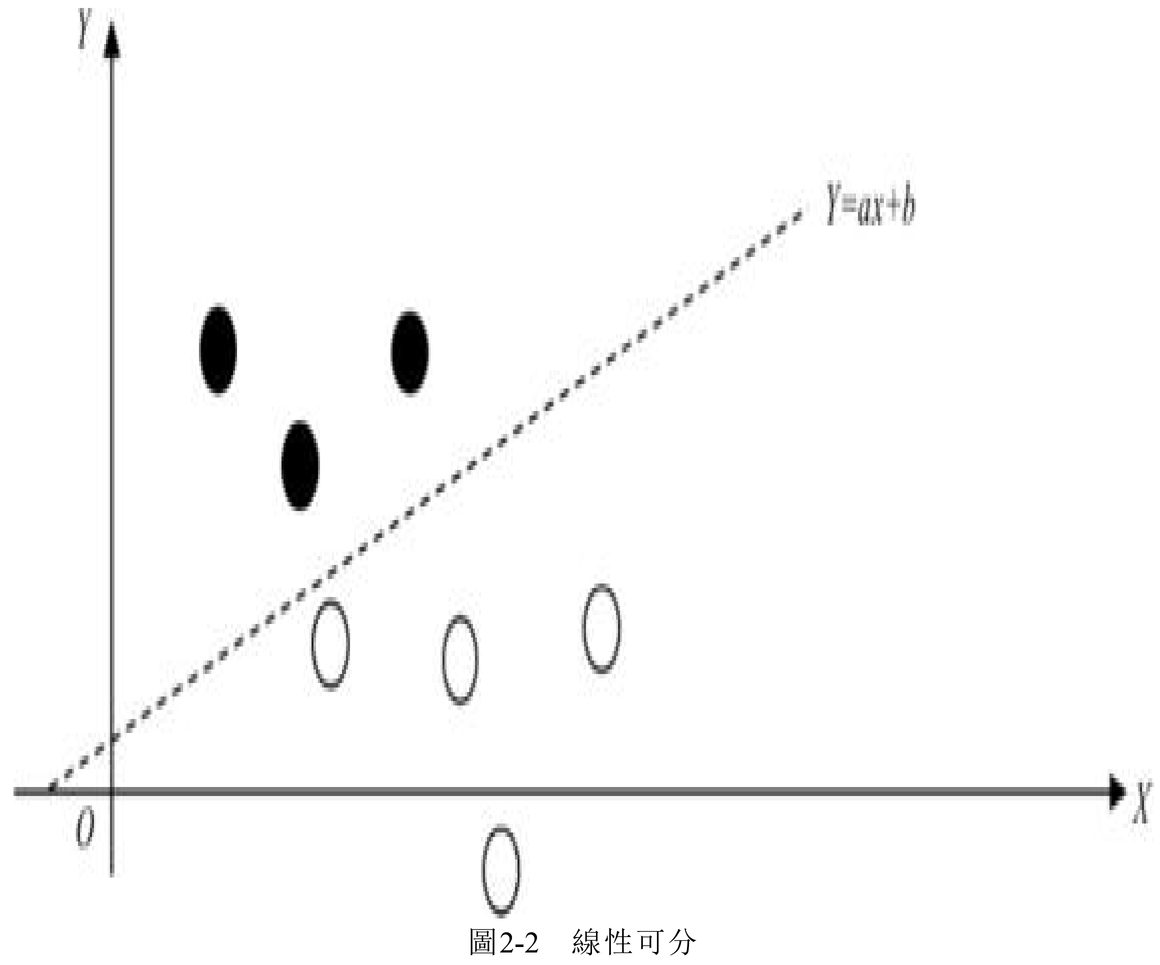


圖2-2　線性可分

由圖2-2可以看到，實心圓和空心圓能夠被一條直線分開。如果我們的資料能被一條直線分為兩類（或多個類），我們便稱其為線性可分。圖2-2實際上是二維座標的資料劃分，我們將在2.4節透過實現一個神經網路來處理。對於在2.1節中所舉的正負數分類問題，這時解決

起來就更簡單了，可以將其視為對x軸上的點進行劃分，如圖2-3所示。

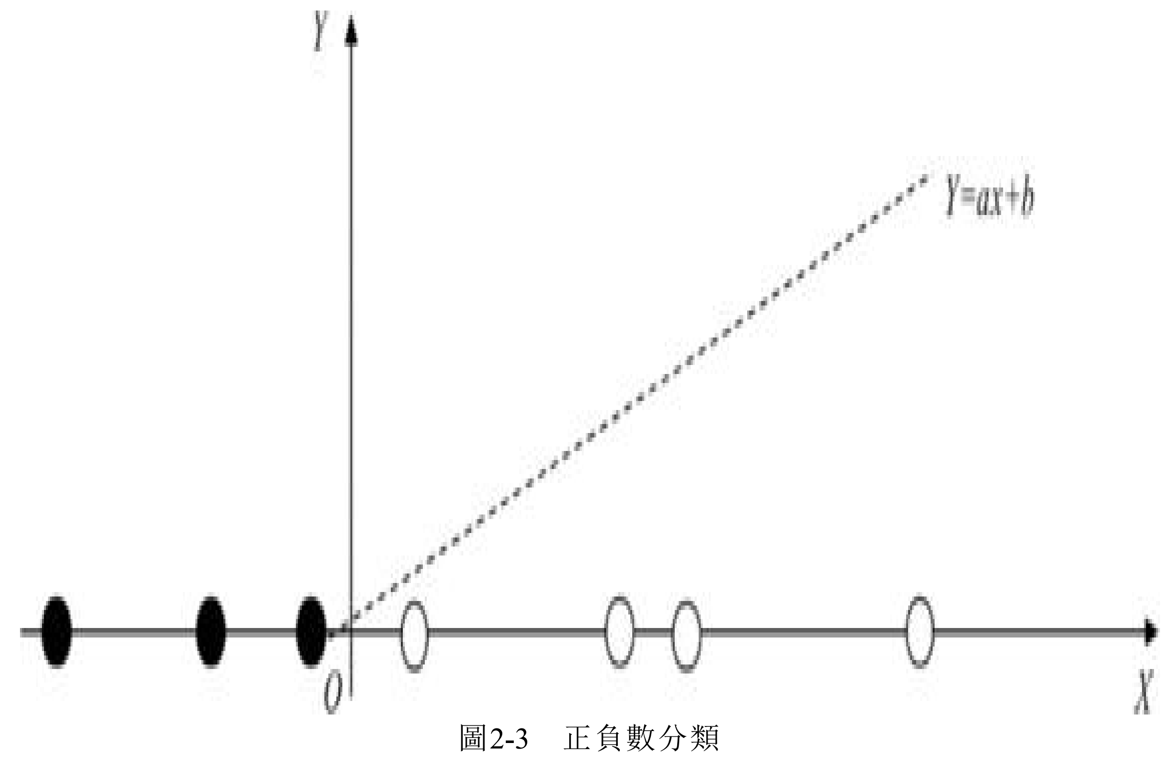


圖2-3　正負數分類

所以，現在我們知道了為什麼要定義y〃(預測值)=φ($wx+b$)，那麼我們來看真正的關鍵問題：怎麼得到w和b？

這實際上是一個線性迴歸（Linear Regression）問題，線性迴歸可以處理多於一個輸入的形式，例如$y'=\varphi(w_1 \cdot x_1+w_2 \cdot x_2+w_3 \cdot x_3)$。在這個例子中，我們實際上預設了$x_0=1$、$w_0$=bias，即$y'=\varphi(w_0 \cdot x_0+w_1 \cdot x_1)=\varphi(w_0+w_1 \cdot x_1)=\varphi(w \cdot x+b)$。注意，這是一個常見的技巧，給輸入引數增加一個固定為1的常量，可以讓我們不用單獨處理bias引數，而能將它作為權重值統一計算，2.4節會講解如何計算。

2.2.2　損失函式與梯度下降

在2.1節中假設：

$$\emptyset(z) = \begin{cases} 1, z > 阈值 \\ 0, 其他 \end{cases}$$

這是在參考文獻[2]中感知器的啟用函式中設定的，但這在實際應用中幾乎不可能這麼簡單判定。在感知器之後，人們提出了Adaptive Linear Neuron（簡稱Adaline）的概念，即其啟用函式和輸入一致：

$$\emptyset(w^{\mathrm{T}}x) = w^{\mathrm{T}}x$$

這樣，在Adaline中，啟用函式的輸出就是一個連續值，而不是如前面感知器那樣的簡單二分，這樣的變化可以讓我們定義兩個重要的概念：損失函式和梯度下降。

損失函式即如何計算圖2-1中的Error值。我們通常使用MSE來計算，即

$$\mathrm{MSE} = \frac{\sum_{i=1}^{n}(y_i - y_i')^2}{N} = \frac{\sum_{i=1}^{n}(y_i - (wx_i + b))^2}{N}$$

其中，y_i為真實值，y_i'為預測值。

用程式碼實現如下：

```
def cost_function(self, X, Y, weight, bias):
    n = len(X)
    total_error = 0.0
    for i in range(n):
        total_error += (Y[i] - (weight*X[i] + bias))**2
    return total_error / n
```

在獲得了損失函式之後，我們便要應用梯度下降來調整w和bias（記為b）。怎麼調整？其實思路很簡單，計算出損失函式針對w和bias的梯度變化Δw和Δb，然後從w和bias中分別減去Δw和Δb即可，如此反覆，直到最後的損失函式值足夠小。

根據上面的MSE公式，我們假設f為MSE，分別對w和b求導可以得到：

$$\Delta w = \frac{\partial f}{\partial w} = \frac{\partial \frac{\sum_{i=1}^{n}(y_i - (wx_i + b))^2}{N}}{\partial w} = \frac{\sum_{i=1}^{n} -2x_i \cdot (y_i - (wx_i + b))}{N}$$

$$\Delta b = \frac{\partial f}{\partial b} = \frac{\partial \frac{\sum_{i=1}^{n}(y_i - (wx_i + b))^2}{N}}{\partial b} = \frac{\sum_{i=1}^{n} -2(y_i - (wx_i + b))}{N}$$

以上即梯度Δw和Δb，然後我們只需要更新w（$w - \Delta w$）、b（$b - \Delta b$）即可。其程式碼實現如下：

```
def update_weights(self, X, Y, weight, bias, learning_rate):
    dw = 0
    db = 0
    n = len(X)

    for i in range(n):
        dw += -2 * X[i] * (Y[i] - (weight*X[i] + bias))
        db += -2 * (Y[i] - (weight*X[i] + bias))

    weight -= (dw / n) * learning_rate
    bias -= (db / n) * learning_rate

    return weight, bias
```

在上面的程式碼中要注意的是，我們並沒有直接從*w*和bias中減去梯度值，而是將梯度值乘以learning rate，以調整訓練步長。

在我們完整實現一個基於線性函式的神經網路之前，還有一點要處理，即資料歸一化。

在前面的感知器的實現中並沒有這一步，因為感知器的「啟用函式」實際上是一個單位階躍函式（Unit Step Function），它能夠把任何輸入值都對映為0或1這兩個值。然而，對於Adaline，因為啟用函式就是網路輸入本身，而*x*的值遠超過[0,1]區間，因此我們需要處理輸入值的範圍，讓最終的輸出能夠落在[0,1]區間（另一種做法是直接改變啟用函式，使其能把任意輸入都對映到[0,1]區間，這會在後面解釋）。我們通常會把輸入值對映到[0,1]區間，輸入則變為

$$x_i = \frac{x_i - x_{\min}}{x_{\max} - x_{\min}}$$

2.2.3 神經元的線性迴歸實現

現在我們就能來完整實現應用了梯度下降的單神經元網路了（不包括隱藏層）：

```
1    class LinearRegression(object):
2        def __init__(self, eta=0.01, iterations=10):
3            self.lr = eta
4            self.iterations = iterations
5            self.w = 0.0
6            self.bias = 0.0
7
8        def cost_function(self, X, Y, weight, bias):
9            n = len(X)
10           total_error = 0.0
11           for i in range(n):
12               total_error += (Y[i] - (weight*X[i] + bias))**2
13           return total_error / n
14
15       def update_weights(self, X, Y, weight, bias, learning_rate):
16           dw = 0
17           db = 0
18           n = len(X)
19
20           for i in range(n):
21               dw += -2 * X[i] * (Y[i] - (weight*X[i] + bias))
22               db += -2 * (Y[i] - (weight*X[i] + bias))
23
24           weight -= (dw / n) * learning_rate
25           bias -= (db / n) * learning_rate
26
27           return weight, bias
28
29
30       def fit(self, X, Y):
31           cost_history = []
32
33           for i in range(self.iterations):
34               self.w, self.bias = self.update_weights(X, Y, self.w, self.bias,
35   self.lr)
36
37               # 计算误差，用于观察和监控训练过程
```

```
38              cost = self.cost_function(X, Y, self.w, self.bias)
39              cost_history.append(cost)
40
41              if i % 10 == 0:
42                  print("iter={:d}    weight={:.2f}    bias={:.4f}
43  cost={:.2}".format(i, self.w, self.bias, cost))
44
45          return self.w, self.bias, cost_history
46
47      def predict(self, x):
48          x = (x+100)/200
49          return self.w * x + self.bias
50
51
52  x = [1, 2, 3, 10, 20, 50, 100, -2, -10, -100, -5, -20]
53  y = [1.0, 1.0, 1.0, 1.0, 1.0,  1,0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0]
54
55  model = LinearRegression(0.01, 500)
56
57  X = [(k+100)/200 for k in x]
58
59  model.fit(X, y)
60
61  test_x = [90, 80,81, 82, 75, 40, 32, 15, 5, 1, -1, -15, -20, -22, -33, -45,
62  -60, -90]
63  for i in range(len(test_x)):
64      print('input {} => predict: {}'.format(test_x[i],
65  model.predict(test_x[i])))
```

透過前面的討論，我們可以很容易理解上面這段程式碼了。

第8～27行是損失函式和利用梯度調整*w*和bias的計算。

在第30～39行的fit實現中，我們實際上並沒有用cost function所得到的cost來控制訓練次數，而是簡化為訓練固定次數，即反覆呼叫update_weight調整*w*和bias。

第47～49行負責使用*w*和bias進行預測分類。其中，第1步先將輸入值*x*歸一化，因為我們從輸入數值可以看到最小值為-100，最大值為200。然後根據我們對啟用函式的設定，將*wx*+bias直接作為輸出返回。

第52～59行是訓練過程。可以看到，我們使用了比之前的例子更多的資料以期獲得可接受的結果，在第57行對所有訓練資料都進行了資料歸一化，使其落在[0,1]區間，然後呼叫fit函式訓練500次（每次都使用所有資料）。

第65行呼叫predict方法，使用訓練得到的*w*和bias對在第61行中所定義的輸入*x*進行預測。

程式碼執行結果如下：

```
iter=0    weight=0.01    bias=0.0117    cost=0.57
…
iter=100    weight=0.25    bias=0.4164    cost=0.24
…
iter=300    weight=0.29    bias=0.4336    cost=0.23
…
iter=490    weight=0.32    bias=0.4207    cost=0.23
input 90 => predict: 0.7238852731029148
input 80 => predict: 0.7078964168036423
input 81 => predict: 0.7094953024335697
input 82 => predict: 0.7110941880634969
input 75 => predict: 0.6999019886540062
input 40 => predict: 0.6439409916065527
input 32 => predict: 0.6311499065671348
input 15 => predict: 0.6039688508583716
input 5 => predict: 0.5879799945590992
input 1 => predict: 0.5815844520393902
input -1 => predict: 0.5783866807795357
input -15 => predict: 0.5560022819605543
input -20 => predict: 0.5480078538109181
input -22 => predict: 0.5448100825510637
input -33 => predict: 0.5272223406218639
input -45 => predict: 0.5080357130627371
input -60 => predict: 0.4840524286138284
input -90 => predict: 0.4360858597160111
```

在上面的執行結果中，我們可以看到訓練損失的確在收斂、下降，但在超過300之後已經沒有太大變化了。

對於所有測試資料，我們可以看到閾值基本上在0.58左右，正數越大越趨近於1，負數越小越趨近於0，符合我們的預期。

關於線性迴歸的內容，會在第4章深入討論。

2.3 隨機梯度下降及實現

在2.2.3節線性迴歸的實現中，我們看到在update_weights（更新權重）方法中，實際上每一次的更新都將所有輸入資料處理了一次，將所有輸入都作為一輪訓練：

```
for i in range(n):
    dw += -2 * X[i] * (Y[i] - (weight*X[i] + bias))
    db += -2 * (Y[i] - (weight*X[i] + bias))
```

因為上例的訓練資料很少，所以每次都載入所有資料進行訓練並沒有什麼關係。然而，在實際的生產環境中訓練資料可達到千萬級別，要每次都載入這些資料進行訓練是很不現實的。但我們可以換一種方式，每次只用一條隨機資料來訓練，這便是隨機梯度下降的原理。

當然，這樣會讓訓練時間變長，所以人們在實際環境中通常會採用mini batch方法，也就是每次更新權重時既不使用所有資料，也不隨機挑選一條資料，而是隨機挑選一個子集來訓練。例如，如果有1000條資料，那麼每次都可以隨機挑選16條或者64條資料進行訓練。對mini batch的大小設定是個很有趣的研究課題，和模型本身、資料特點等都有關係，這裡不必細究。下面修改2.2節的線性迴歸實現，看看採用mini batch方法是如何進行訓練的。

```
1   def update_weights(self, X, Y, weight, bias, learning_rate):
2         dw = 0
3         db = 0
4         n = len(X)
5
6         indexes = [0:n]
7         random.shuffle(indexes)
8         batch_size = 4
9
10        for k in range(batch_size):
11            i = indexes[k]
12            dw += -2 * X[i] * (Y[i] - (weight*X[i] + bias))
13            db += -2 * (Y[i] - (weight*X[i] + bias))
14
15        weight -= (dw / n) * learning_rate
16        bias -= (db / n) * learning_rate
17
18      return weight, bias
```

如上所示，我們只需要修改update_weights方法即可。在第6行設定了一個陣列用於儲存訓練資料的索引下標；在第7行將有序索引陣列隨機打亂；在第8行設定每次訓練都取4條資料。於是在第10～11行，我們只需從已經隨機打亂的資料索引表中取前4條進行訓練即可。

訓練結果如下：

```
...
iter=470    weight=0.32    bias=0.4220    cost=0.23
iter=480    weight=0.32    bias=0.4213    cost=0.23
iter=490    weight=0.32    bias=0.4207    cost=0.23
input 90 => predict: 0.7238852731029148
input 80 => predict: 0.7078964168036423
input 81 => predict: 0.7094953024335697
input 82 => predict: 0.7110941880634969
input 75 => predict: 0.6999019886540062
input 40 => predict: 0.6439409916065527
input 32 => predict: 0.6311499065671348
input 15 => predict: 0.6039688508583716
input 5 => predict: 0.5879799945590992
input 1 => predict: 0.5815844520393902
input -1 => predict: 0.5783866807795357
input -15 => predict: 0.5560022819605543
input -20 => predict: 0.5480078538109181
input -22 => predict: 0.5448100825510637
input -33 => predict: 0.5272223406218639
input -45 => predict: 0.5080357130627371
input -60 => predict: 0.4840524286138284
input -90 => predict: 0.4360858597160111
```

把這組結果和在2.2節中用全批次資料訓練的結果相比，可以看到預測效果差別不大，這也證明瞭mini batch方法的有效性。至於隨機梯度下降，它只是batch_size為1時的特殊情況而已。

2.4 單層神經網路的Python實現

機器學習演算法隨著硬體算力的提升而演化。早期的機器學習以神經元為切入點，產生了簡單的引數調整和擬合演算法。然而要模擬較為複雜的計算，顯然不是單個神經元能做到的。我們也看到，神經元的提出是為了模擬人類大腦的神經突觸工作模式，那麼在理解單個神經元的工作原理後，下一步自然就是模擬多個神經元了，即神經網路的演算法。

2.4.1 從神經元到神經網路

前面實現的都是基於圖2-1的感知器結構，利用不同的演算法來實現對權重的訓練，重點是如何用梯度下降來修正權重，達到不斷逼近最優解的目的。

但並不能說這是一個真正的神經網路。一般來說，神經網路包含輸入層、隱藏層和輸出層，而前面講解的內容，只是隱藏層中一個神經元的權重調整方法而已。

如圖2-4所示，其中的左圖是圖2-1所示的感知器神經元，右圖是一個完整的單層神經網路。神經元實際上只是神經網路中的一小部分而已（用粗線標識）。神經網路的輸出不再由單個神經元決定，而由多個神經元的輸出共同決定。

在圖2-4中只設計了一組神經元，我們把這組神經元稱為神經網路的隱藏層（Hidden Layer）。早期的神經網路基本上只包括一個隱藏層，少數包括多個隱藏層，如多層感知器（Multilayer Perceptron，MLP)，如圖2-5所示。

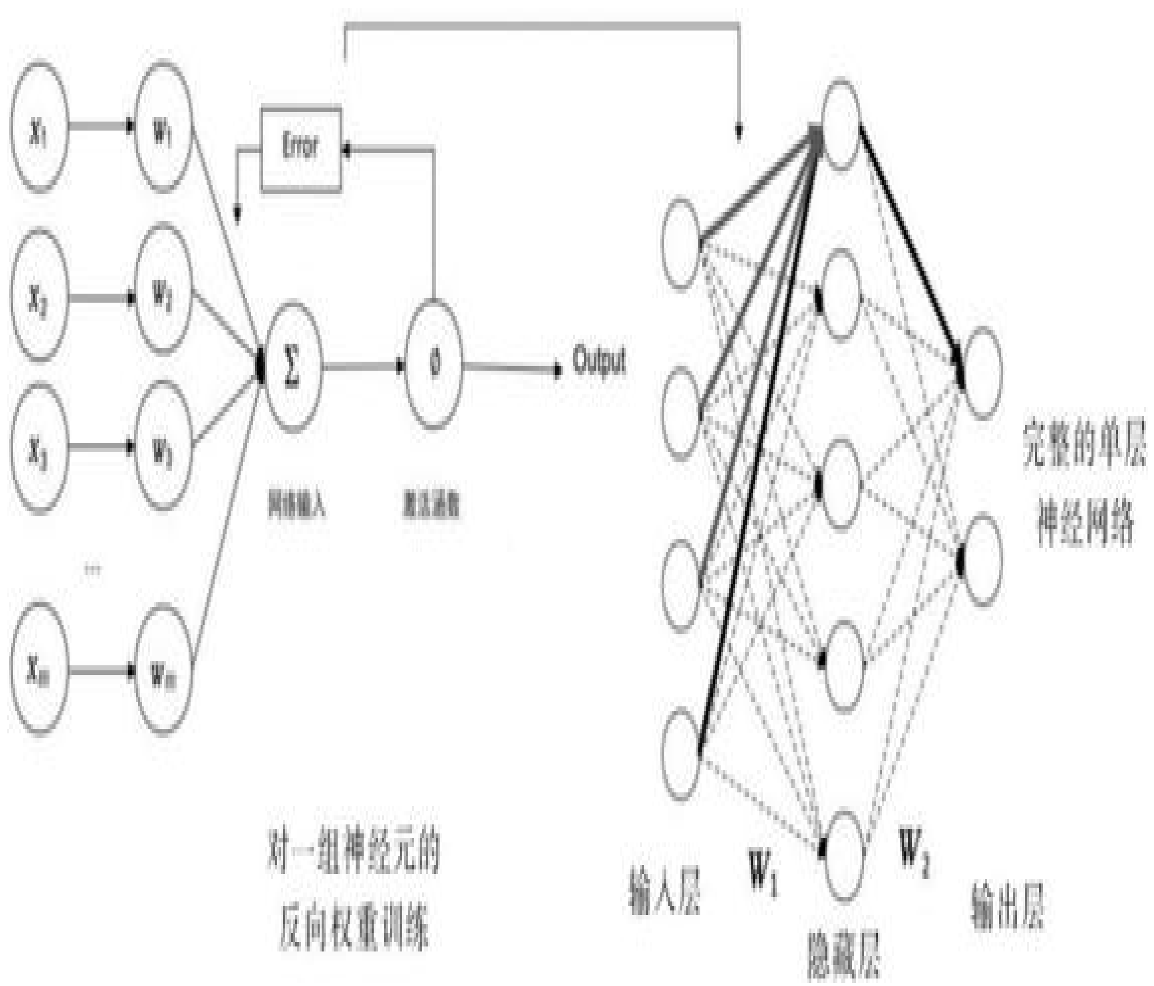


圖2-4　從神經元到神經網路

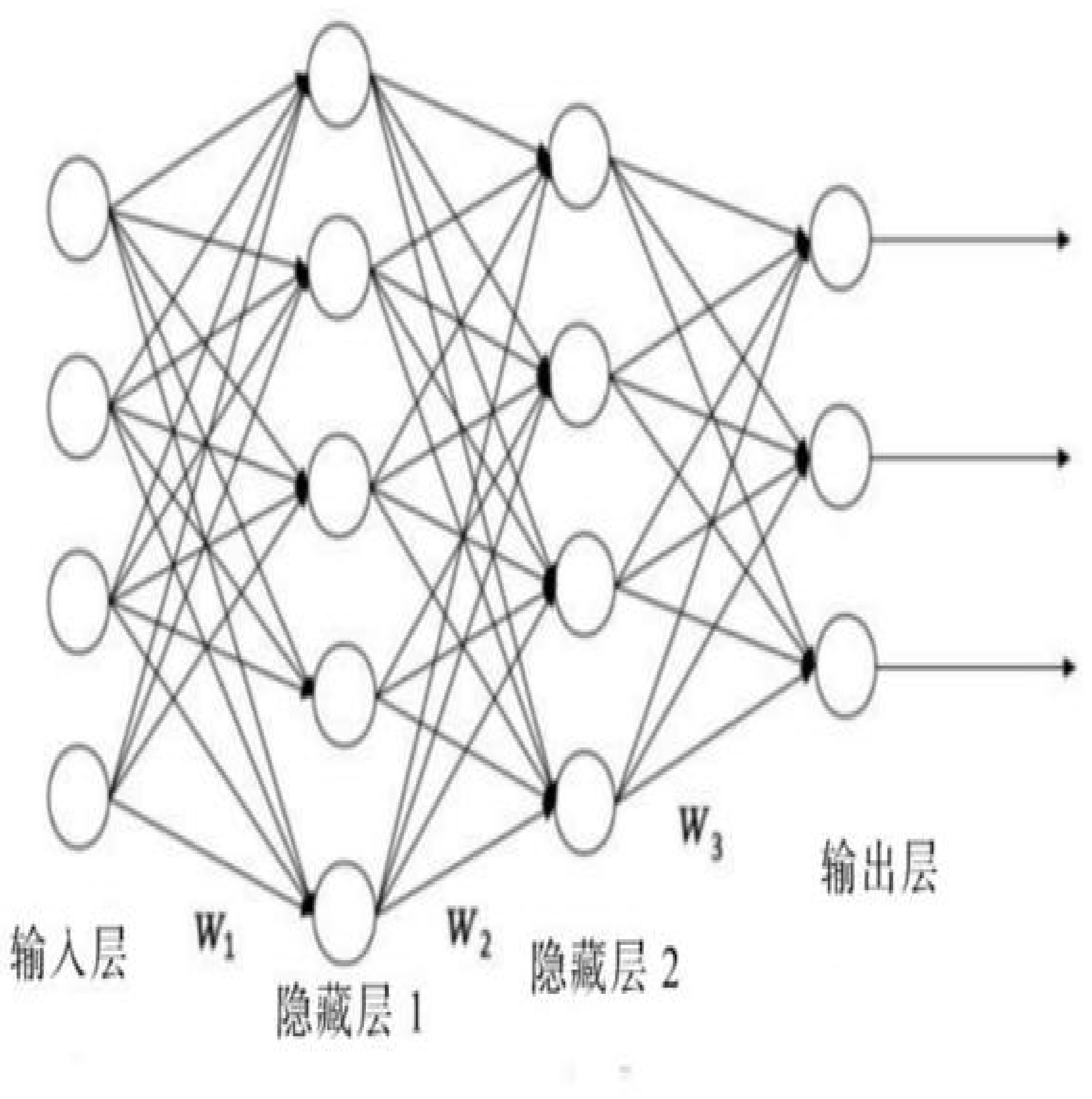


圖2-5　多層感知器，包括多個隱藏層

如果我們在圖2-5的基礎上再增加層數，則這樣形成的網路往往被稱為深度網路（Deep Network），這也是深度學習的由來。我們通常把隱藏層的數量稱為網路的深度（Depth），把每一層的神經元個數稱為寬度（Width）。於是我們在研究中有一個有趣的問題：對於同樣數量的神經元，是使用更大的寬度、減少深度，還是增加深度、減少寬度？如何達到最佳平衡？這是機器學習中仍然引人深思的問題，目前並沒有標準答案。在實際應用中，我們更多地結合網路的寬度和深

度，透過實驗達到最佳效果，如在*Wide & Deep Learning for Recommendation Systems*[3]一文中所提及的方式。

2.4.2 單層神經網路：初始化

和單個神經元的訓練相比，神經網路確實要複雜許多，但實際上也只是計算次數和引數變得更多，其原理和訓練方式實質上是一樣的。我們下面親手實現一個類似圖2-4的單層神經網路，看看在引入隱藏層之後，到底是如何進行訓練和預測的。

首先，定義問題。這次不再對正負數進行分類，而是對平面座標點進行分類，如圖2-6所示。

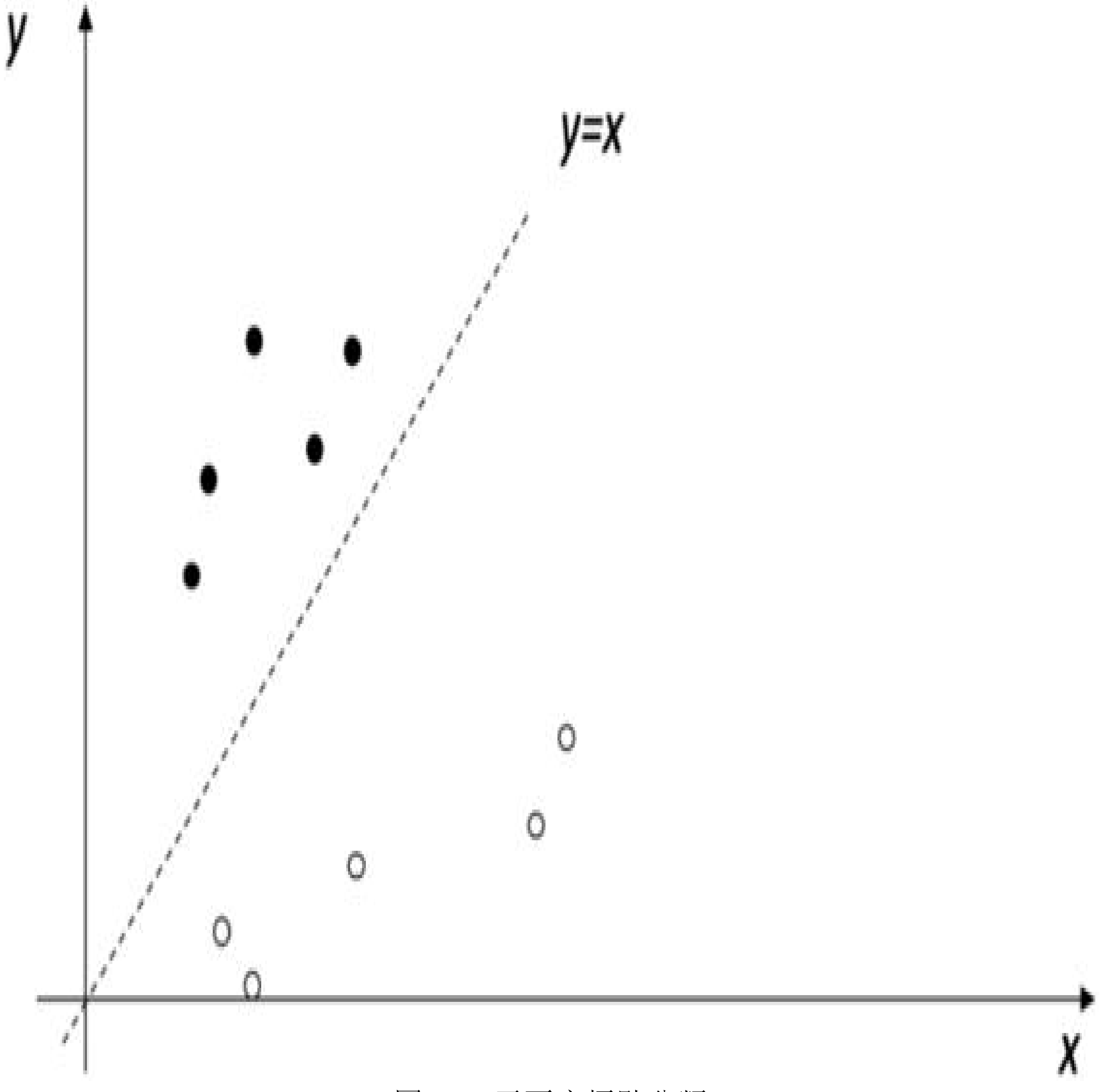


圖2-6　平面座標點分類

在圖2-6中，斜線$y=x$將平面座標點分為兩類，線上方為0，線下方為1。我們希望設計一個神經網路來對任意座標點(x,y)進行自動分類。仿照圖 2-4，這裡的輸入層包括x、y兩個輸入，輸出則包括0和1這兩個類別，因此網路結構如圖2-7所示。

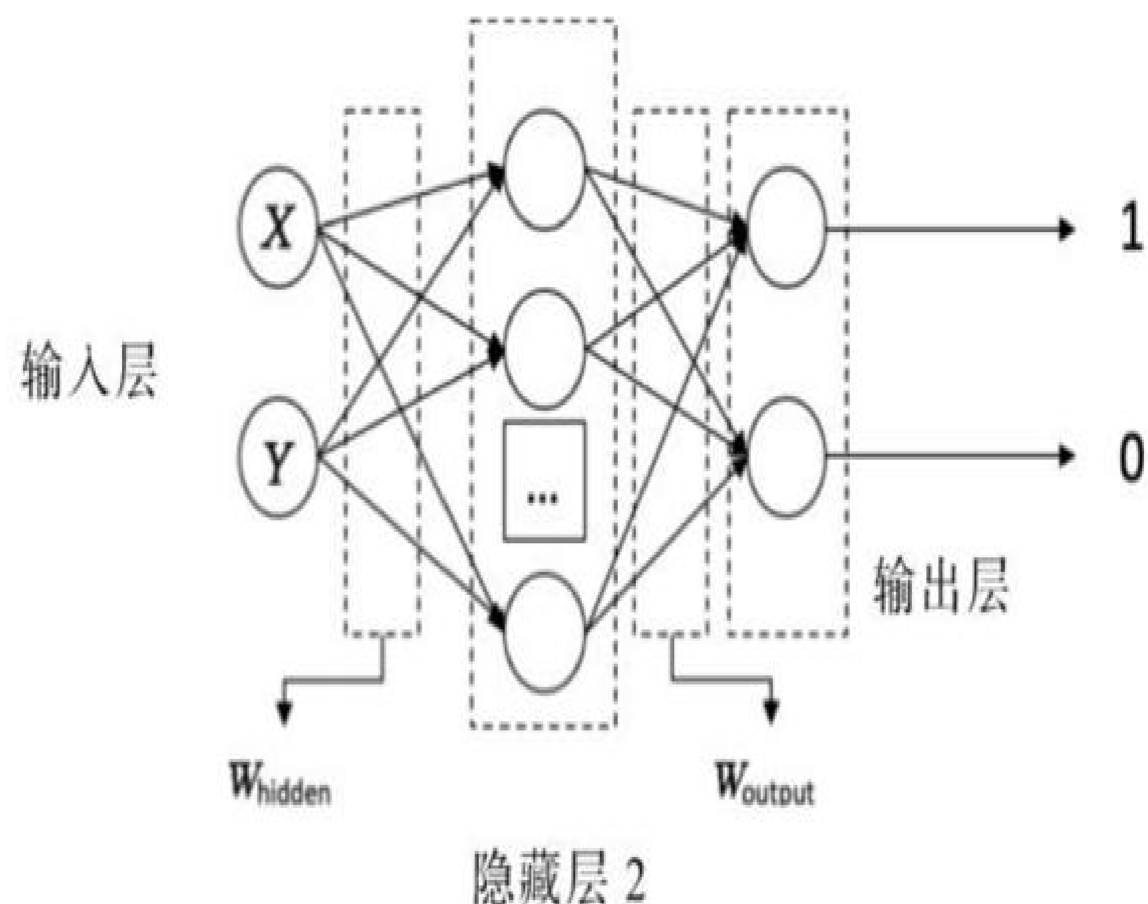


圖2-7　待實現的單層神經網路，其中隱藏層的神經元個數可設定

可以看到，圖 2-7 中神經網路的輸出是由隱藏層的多個神經元的輸出確定的。同時，因為我們的輸出不再只依靠一個神經元（雖然可以設定隱藏層只用一個神經元），因此除了隱藏層中每個神經元對應的輸入權重都需要計算（W_{hidden}），輸出層中的每個輸出對於每個隱藏層的神經元的輸入也有對應的權重需要計算（W_{output}）。

下面看看具體如何實現以上網路。

因為該分類的資料特點比較明顯，所以我們可以先手工建立兩組資料分別用於訓練與測試：

```
dataset = [[2.7810836,4.550537003,0],
    [3.396561688,4.400293529,0],
    [1.38807019,1.850220317,0],
    [3.06407232,3.005305973,0],
    [7.627531214,2.759262235,1],
    [5.332441248,2.088626775,1],
    [6.922596716,1.77106367,1]]


test_data = [[1.465489372,2.362125076,0],
        [8.675418651,-0.242068655,1],
        [7.673756466,3.508563011,1]]
```

然後定義輸入和輸出的個數。因為輸入包括x和y兩個座標，輸出包括1和0兩個類別，因此各自都為2。

```
n_inputs = 2
n_outputs = 2
```

接著我們就要開始定義神經網路了，可以透過建立一個initialize_network函式來實現：

```
def initialize_network(n_inputs, n_hidden, n_outputs):
    network = list()
    hidden_layer = [{'weights': [random() for i in range(n_inputs + 1)]} for i in range(n_hidden)]
    output_layer = [{'weights': [random() for i in range(n_hidden + 1)]} for i in range(n_outputs)]
    network.append(hidden_layer)
    network.append(output_layer)
    return network
```

在上面的程式碼中，我們將網路模型定義為一個包含兩個陣列的list，兩個陣列分別對應圖2-7中的兩組權重：W_{hidden}和W_{output}。

對於隱藏層的權重，參考圖2-7，我們可以看到隱藏層的每個神經元（總數由n_hidden定義）都接收了所有輸入，其數量為n_inputs+1個（其中比輸入多出來的一項為bias，也可將其理解為大多數線性迴歸所定義的$y = w_0 + w_1 \cdot x_1 + w_2 \cdot x_2 + \cdots + w_k \cdot x_k$中的$w_0$）。這裡一個全連線層（每個輸出都和所有輸入直接相關）的權重引數總數為(n_inputs+1)·n_hidden。

同樣，對output的權重採用同樣的處理，注意，output層的權重引數總數是(n_hidden+1)·n_output。

2.4.3 單層神經網路：核心概念

我們接下來開始訓練網路。怎麼訓練呢？其實和前面的流程非常相似。在前面的兩段程式碼示例simple_perceptron和linear_regression中，因為程式碼過於簡單，所以在訓練過程中沒有明確分離出一些概

念的實現。例如在simple_perceptron程式碼示例中並沒有定義損失函式和更新權重的單獨方法，而是單獨實現了net_input和predict（實際上相當於啟用函式）。而在linear_regression的程式碼中強調了損失函式和更新權重update_weight的概念，用於解釋梯度下降最佳化，卻沒有提及網路輸入和啟用函式。

所以實際上，對於任何模型訓練，其關鍵都是實現如下4個核心函式。

◎ net_input：計算神經元的網路輸入。

◎ activation：啟用函式，將神經元的網路輸入對映到下一層的輸入空間。

◎ cost_function：計算誤差損失。

◎ update_weights：更新權重。

這裡還需要引入兩個概念：前向傳播（Forward Propagation）和反向傳播（Back Propagation）。

◎ 前向傳播：指將資料輸入神經網路中，每個隱藏層的神經元都接收網路輸入，透過啟用函式進行處理，然後進入下一層或者輸出的過程。在前面linear_regression的例子中，predict方法實際上就是一個簡單的前向傳播方法。

◎ 反向傳播：指對網路中的所有權重都計算損失函式的梯度，這個梯度會在最佳化演算法中用來更新權值以最小化損失函式。實際上，它指代所有基於梯度下降利用鏈式法則（Chain Rule）來訓練神經網路的演算法，以幫助實現可遞迴迴圈的形式來有效地計算每一層的權重更新，直到獲得期望的效果。在前面的linear_regression例子中，我們把反向傳播計算梯度的內容和更新權重放在了一起。

瞭解了以上概念，可以知道實際上我們在前面已經實現過相關內容，只是沒有清晰地對每個環節都進行模組化。而這裡因為不再是單個神經元，所以計算環節更加複雜，對每個關鍵環節都進行模組化是非常必要的。我們來看看每一步是怎麼實現的。

2.4.4 單層神經網路：前向傳播

首先是每個神經元的網路輸入：

```
def net_input(weights, inputs):

    total_input = weights[-1]

    for i in range(len(weights)-1):

        total_input += weights[i] * inputs[i]

    return total_input
```

注意，weights實際上包含了一個類似bias的額外引數，即weights的個數比輸入（inputs）要多一個。因此我們首先使用weights[-1]對total_input賦值，然後新增每個輸入和對應權重的乘積，其形式為total_input = weights[:-2]·inputs + weights[-1]。

然後是啟用函式activation：

```
def activation(total_input):
    return 1.0/ (1.0 + exp(-total_input))
```

這裡使用了sigmoid啟用函式，用於將網路輸入對映到(-1,1)區間。啟用函式有多種形式和演算法，在第3章會做詳細解釋，這裡不再贅述，只需把它當作一個區間對映的函式即可。

定義完上述兩個函式後，我們便可以定義前向傳播的實現：

```
def forward_propagate(network, row):
    inputs = row
    for layer in network:
        outputs = []
        for neuron in layer:
            total_input = net_input(neuron['weights'], inputs)
            neuron['output'] = activation(total_input)
            outputs.append(neuron['output'])
        inputs = outputs
    return inputs
```

可以看到，前向傳播其實就是對於每一層，都把上一層的輸出作為下一層的輸入，進行迴圈計算。因為這是全連線網路，所以每個神經元都接收所有inputs，進行相同的net_input處理，將獲得的total_input結果再輸入啟用函式activation中，獲得該神經元的最終結果，然後把結果新增到該層的輸出中。我們把當前層的輸出（outputs）作為下一層的輸入（inputs），持續迭代下去，直到最後把輸出返回。

2.4.5　單層神經網路：反向傳播

可以看到，每個神經元所進行的計算過程都是一樣的，唯一影響結果的就是其中的權重引數（neuron['weights']）。下面就要進行反向傳播和權重更新的實現，幫助每個神經元都調整自己的相關引數。

這裡和前面最大的差別在於，啟用函式不再是直接的線性方程，而是使用了sigmoid啟用函式，那麼我們在計算梯度變化時的求導就需要有所變化。簡單看一下sigmoid啟用函式的形式和求導結果：

$$\text{sigmoid}(z) = \emptyset(z) = \frac{1}{1+\mathrm{e}^{-z}}$$

$$\frac{\partial \emptyset}{\partial z} = \frac{\partial\left(\frac{1}{1+\mathrm{e}^{-z}}\right)}{\partial z} = z(1-z)$$

結合forward_propagation函式的實現，我們可以看到，這裡$\emptyset(z)$中的輸入z其實就是前一層的輸出（每一層的輸入都是上一層的輸出）。

在本章第1個簡單神經元simple_perceptron及後面的linear_regression實現中，我們看到對權重的調整是這樣的：

```
update = self.lr * (y - self.predict(x))
self.w += update * x
```

當感知器只有一個神經元時，對權重的調整很簡單：

$$w' = w + lr \cdot x \cdot (y - y')$$

對於沒有隱藏層的單個神經元感知器來說，是有明確的結果（y）和預測結果（y'）的，誤差結果由cost_function（MSE）確定。根據前面Linear Regression中的推導，在將MSE對w求導後，可得到$\Delta w = x \cdot (y - y')$，因此可以進一步概括為

$$w' = w + lr \cdot \Delta w$$

那麼問題就變為如何計算Δw。

回到要在圖2-7中實現的單層神經網路，對於其中的輸出層，其處理方式和單神經元感知器類似，因為輸出的也是最終預測值y'，用MSE作為損失函式計算即可。注意，輸出層的輸入並不是原始輸入值x、y，而是上一層（隱藏層）的輸出。而對於隱藏層來說，我們並不能直接計算它的輸出誤差，因為並不存在輸出層的真實值y，只能透過鏈式法則來間接推導。換句話說，假設隱藏層有一個權重w，我們希望對其進行修正，那麼只能從最後輸出的損失函式計算出的誤差倒推（反向傳播），如圖2-8所示。

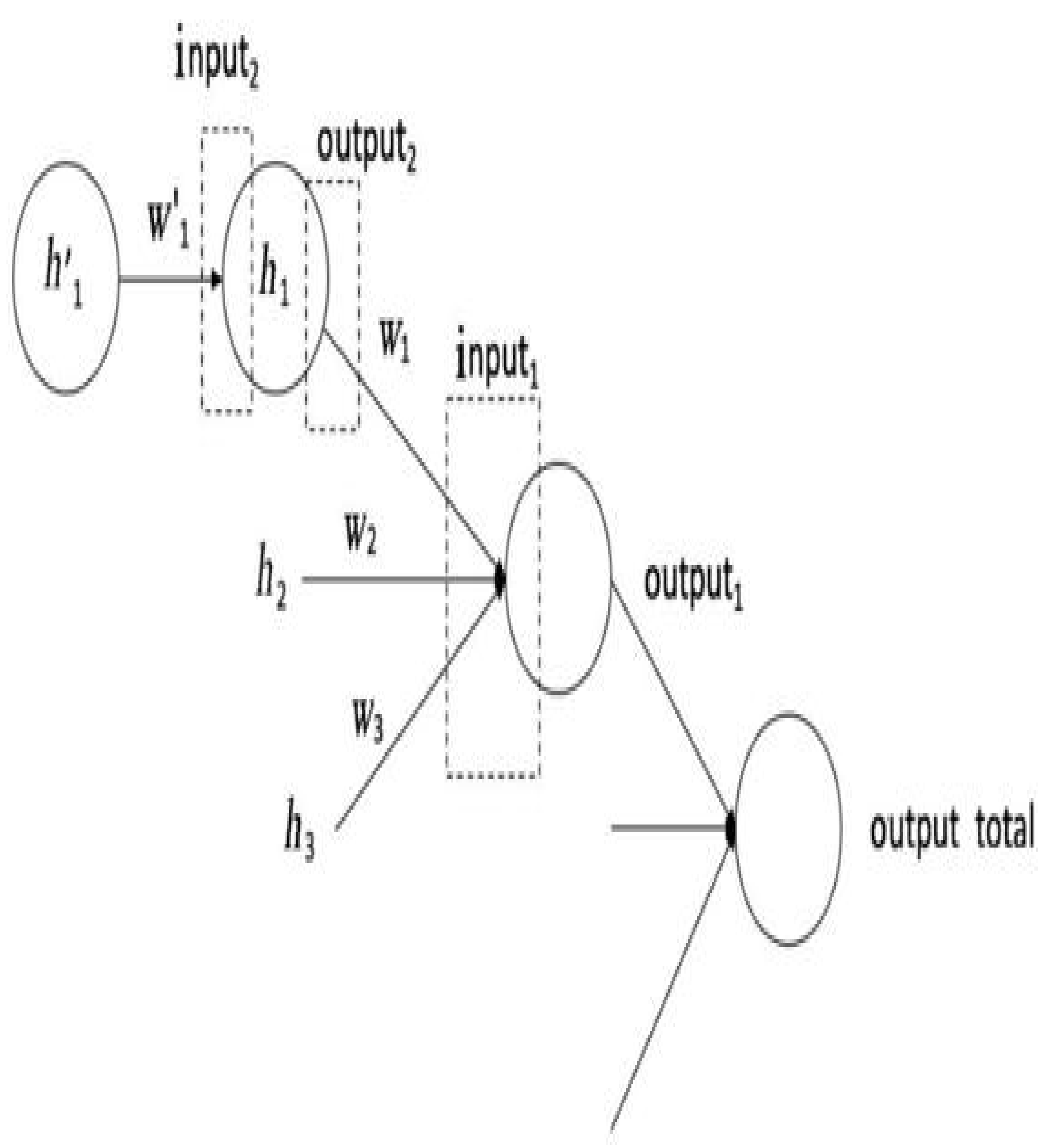


圖2-8　鏈式法則示意圖

根據圖2-8的示意，如果要計算最終output_total的誤差和w_1的關係，則可以得到這樣一個公式：

$$\Delta w = \frac{\partial E_{\text{output_total}}}{\partial w_1} = \frac{\partial E_{\text{output_total}}}{\partial \text{output}_1} \cdot \frac{\partial \text{output}_1}{\partial \text{input}_1} \cdot \frac{\partial \text{input}_1}{\partial w_1}$$

其中：

$$\frac{\partial E_{\text{output_total}}}{\partial \text{output}_1} = \frac{\partial(\Sigma(y - \text{output}_i)^2)}{\partial \text{output}_1} = -2 \cdot (y - \text{output}_1)$$

$$\frac{\partial \text{output}_1}{\partial \text{input}_1} = \frac{\partial\left(\text{sigmoid}(\text{input}_1)\right)}{\partial \text{input}_1} = \text{input}_1(1 - \text{input}_1)$$

$$\frac{\partial \text{input}_1}{\partial w_1} = \frac{\partial(w_1 \cdot h_1 + w_2 \cdot h_2 + w_3 \cdot h_3)}{\partial w_1} = h_1$$

這樣最終可以得到：

$$\Delta w_1 = -2 \cdot (y - \text{output}_1) \cdot \text{input}_1(1 - \text{input}_1) \cdot h_1$$

隱藏層可以繼續迭代下去，例如對圖2-8中的w_1'，可用類似的做法。我們記

$$\delta(h1) = \frac{\partial E_{\text{output_total}}}{\partial \text{output}_1} \cdot \frac{\partial \text{output}_1}{\partial \text{input}_1} = -2 \cdot (y - \text{output}_1) \cdot \text{input}_1(1 - \text{input}_1),$$

則

$$\Delta w_1^{'} = \delta(h_1) \cdot \frac{\partial \text{input}_1}{\partial \text{output}_2} \cdot \frac{\partial \text{output}_2}{\partial \text{input}_2} \cdot \frac{\partial \text{input}_2}{\partial w_1'}$$

$$= \delta(h_1) \cdot w_1 \cdot \frac{\partial \left(\text{sigmoid}(\text{input}_2)\right)}{\partial \text{input}_2} \cdot h_1'$$

$$= \delta(h_1) \cdot w_1 \cdot \text{input}_2(1 - \text{input}_2) \cdot h_1'$$

實際上，我們沒有必要單獨計算每個Δw，只需沿用前面的運算結果$\delta(h1)$，再和當前神經元的屬性相乘即可。那麼我們在迴圈迭代時，只需儲存$\delta(h1)$就可以大幅度提高運算效率，不必從頭計算。這就是反向傳播的核心要點。

這樣就理順了反向傳播中更新權重的全過程，下面看看它是怎麼具體實現的。

同樣，首先定義cost_function函式：

```
def cost_function(expected, outputs):
    n = len(expected)
    total_error = 0.0
    for i in range(n):
        total_error += (expected[i] - outputs[i])**2
    return total_error
```

需要指出一點：在反向傳播計算中，cost_function函式並不是必需的，根據cost_function函式進行求導才是必需的。但清晰地定義cost_function函式有助於我們理解整個實現。

然後是sigmoid啟用函式的導數實現：

```
def transfer_derivative(output):
    return output * (1.0 - output)
```

二者完成後，便可以完成最重要的反向傳播實現。先來看看下面的實現：

```
1   def backward_propagate(network, expected):
2       for i in reversed(range(len(network))):
3           layer = network[i]
4           errors = list()
5
6           if i == len(network) - 1:
7               for j in range(len(layer)):
8            neuron = layer[j]
9            error = -2 * (expected[j] - neuron['output'])
10           errors.append(error)
11          else:
12           for j in range(len(layer)):
13               error = 0.0
14               for neuron in network[i+1]:
15                   error += (neuron['weights'][j] * neuron['delta'])
16               errors.append(error))
17
18          for j in range(len(layer)):
19              neuron = layer[j]
20              neuron['delta'] = errors[j] * transfer_derivative(neuron['output'])
```

讓我們看看以上程式碼都做了什麼。

第1行：引入兩個引數，一個是需要更新的網路模型network（記住，network實際上是一個list，每個item都是一組引數，其中包含了權重和其他屬性）；另一個是期望值expected，包含結果的真實分類。

第2～4行：從網路的最後一層（輸出層）開始計算，獲得當前層（layer）並設定變數errors為一個list，errors將儲存當前層中每個神經元的預測值的誤差。注意，這個誤差是從輸出層開始不斷迭代累積形成的，而不是和真實目標值*y*的絕對誤差。

第6行：做了一個判斷，對輸出層和隱藏層做了不同的處理。

第7～10行：對最後一層（輸出層）進行處理。對輸出層中的每個神經元（neuron），根據前面定義的公式 $\Delta w = -2 \cdot (y - \mathrm{output}_1) \cdot \mathrm{input}_1(1 - \mathrm{input}_1) \cdot h_1$，將第1部分 $-2 \cdot (y - \mathrm{output}_1)$ 存入errors中。

第18～20行：在這3行裡，如果當前是輸出層，則在當前層的神經元中儲存了 $\Delta w = -2 \cdot (y - \mathrm{output}_1) \cdot \mathrm{input}_1(1 - \mathrm{input}_1)$，其中，$\mathrm{input}_1(1 - \mathrm{input}_1)$ 是sigmoid啟用函式對輸入值的導數。然後進入倒數第2層（隱藏層）。這樣，邏輯過程就很簡明瞭，實際上在每個神經元的delta屬性中都會儲存前面所計算的 $\delta(h)$。

再回到第12～16行，根據前面推導的公式，我們在這裡計算的是前一層的 $\sum \delta(h) \cdot w$，注意，這裡需要將前一層和當前神經元相關的所有輸出誤差全部累加（因為這是全連線網路，所以意味著當前層的每個神經元的輸出都會作用於下一層的每個神經元的輸入）。

於是，當再次進入第18～20行時，隱藏層乘上了對應sigmoid的導數，這時在neuron['delta']中最後儲存的是 $\sum \delta(h) \cdot w \cdot \mathrm{output} \cdot (1 - \mathrm{output})$。要獲得最終的 Δw，則只需最後再乘以當前神經元的inputs即可（即

前一層的output屬性），我們將在更新權重時實現這最後一步，請看下面的程式碼：

```
1  def update_weights(network, row, learning_rate):
2      for i in range(len(network)):
3          inputs = row[:-1]
4          if i != 0:
5              inputs = [neuron['output'] for neuron in network[i-1]]
6          for neuron in network[i]:
7              for j in range(len(inputs)):
8                  neuron['weights'][j] -= learning_rate *
9  neuron['delta'] * inputs[j]
10             neuron['weights'][-1] -= learning_rate * neuron['delta']
```

在update_weights函式中，我們看到，首先仍然是在第2行遍歷網路的所有層。和前面back_propagate函式不一樣的是，這裡不是倒序遍歷，而是順序遍歷（因為初始輸入值在第1層）。

另外，update_weight函式需要在back_propagate函式之後呼叫。因為在back_propagate函式中，我們在每一層所有神經元的delta屬性裡都儲存了

$$\sum \delta(h) \cdot w \cdot \text{output} \cdot (1 - \text{output})$$

，需要乘以該神經元的net_inputs（即前一層的output屬性所存數值，上一部分已經做了詳盡推導，這裡不再贅述）。

因此在第3～5行中，初始化inputs為輸入資料row（實際上是一組訓練資料），如果不是輸入層（即第1層輸入），則將輸入換為前一組的輸出。

從第6行開始，對該層的所有神經元都進行遍歷，對該神經元的每一個輸入inputs[*j*]所對應的權重neuron['weights'][*j*]都減去Δw，其

中，Δw為在back_propagate函式中計算的neuron['delta']·inputs[j]，於是就有了第8行的計算：

```
neuron['weights'][j] -= learning_rate*neuron['delta']*inputs[j]
```

最後，我們對額外的權重引數bias進行處理，因為其輸入被設定為1，所以在第10行設定最後一個權重為learning_rate·neuron['delta']。

2.4.6 網路訓練及調整

現在，我們已經定義了所有核心的反向傳播權重調整中所需要的函式，可以來實現具體的訓練程式碼了：

```
1   def train_network(network, training_data, learning_rate, n_epoch, _outputs):
2       for epoch in range(n_epoch):
3           sum_error = 0
4           for row in training_data:
5               outputs = forward_propagate(network, row)
6               expected = [0 for i in range(n_outputs)]
7               expected[row[-1]] = 1
8               sum_error += cost_function(expected, outputs)
9               backward_propagate(network, expected)
10              update_weights(network, row, learning_rate)
11      print('>epoch: %d, learning rate: %.3f, error: %.3f' % (epoch,
12  learning_rate, sum_error))
```

在定義好前面的函式後，真正的模型訓練只有短短12行，而且淺顯易懂，如下所述。

第1行：在訓練函式的定義中，我們需要指定網路模型物件、訓練資料、學習率、訓練次數和輸出型別的數量。

第2行：根據給定的訓練次數n_epoch進行迴圈訓練。

第3行：sum_error是當前訓練週期（每個週期都使用全部訓練集來訓練）的誤差，這實際上對訓練本身沒有影響，只是檢查一下損失（Loss）。

第4行：遍歷所有訓練資料，取其中一組開始訓練。

第5行：進行前饋計算（前向傳播），獲得最終輸出。注意，這個輸出包括在n_outputs中定義的類別個數，對每個類別都生成一個機率。在本例中輸出的是一個長度為2的一維陣列，代表0、1兩個分類。這還不算是最終的預測結果，需要在這兩個分類中選擇機率最大的一個作為預測結果。

第6～7行：做了一個小的技巧性實現，我們需要對每組輸入資料都建立對應的期望輸出（expected）。第6行首先對期望輸出置0；第7行根據輸入資料的最後一個數值（也就是ground truth標籤，表示具體是哪個類別）將期望輸出中的對應位置置1。

第8行：透過cost_function計算誤差。

第9～10行：首先呼叫back_propagate設定每個神經元的delta屬性，再透過update_weights調整權重。

第11～12行：按照習慣，我們在每個訓練週期結束時都需要顯示一些必要的引數，供檢視進度。

那麼模型到底是怎麼預測的呢？其實和前面的forward_propagate類似，只是最後要把機率最大的分類選出來，其實現如下：

```
def predict(network, row):
    outputs = forward_propagate(network, row)
    return outputs.index(max(outputs))
```

最後，可以執行其程式碼：

```
network = initialize_network(n_inputs, 1, n_outputs)
train_network(network, training_data = dataset, learning_rate = 0.5, n_epoch
= 20, n_outputs = n_outputs)

for row in test_data:
    result = predict(network, row)
    print('expected: %d, predicted: %d\n' % (row[-1], result))
```

在上面的程式碼中首先呼叫了initialize_network對網路模型進行初始化，這裡對隱藏層只設了一個神經元；然後呼叫train_network訓練網路模型；在訓練完成後，我們用測試資料test_data中的每一組進行驗證，呼叫predict函式並把預測結果和測試資料的標籤列進行對比。

因為network權重的初始值是隨機的，所以我們執行3次程式碼看看結果：

第1次	expected: 0, predicted: 1 expected: 1, predicted: 1 expected: 1, predicted: 1
第2次	expected: 0, predicted: 1 expected: 1, predicted: 1 expected: 1, predicted: 1
第3次	expected: 0, predicted: 0 expected: 1, predicted: 1 expected: 1, predicted: 1

可以看到，前兩次均有一組資料預測錯誤，最後一組資料預測全部正確。怎麼提高正確率呢？我們首先可以嘗試增加神經網路的寬度，在原有的僅有一個神經元的基礎上再加一個，變為兩個神經元，也就是在呼叫initialize_network時，將n_hidden引數設為2：

```
network = initialize_network(n_inputs, 2, n_outputs)
```

這時再連續執行三次，可以看到三次的測試結果都完全正確（結果完全一致，這裡省略執行結果展示）。

另一種思路是增加深度，在原有的單層隱藏層上再加一層，這涉及initialize_network的改動，我們來試試：

```
def initialize_network(n_inputs, n_hidden, n_outputs):
    network = list()
    hidden_layer1 = [{'weights': [random() for i in range(n_inputs + 1)]}
for i in range(n_hidden)]
    hidden_layer2 = [{'weights': [random() for i in range(n_hidden + 1)]}
for i in range(n_hidden)]
    output_layer = [{'weights': [random() for i in range(n_hidden + 1)]} for
i in range(n_outputs)]
    network.append(hidden_layer1)
    network.append(hidden_layer2)
    network.append(output_layer)
    return network
```

在上面新改動的initialize_network中，我們把之前的單個hidden_layer改為了hidden_layer1和hidden_layer2，這兩個新的隱藏層的神經元個數一致，都由n_hidden指定。

我們保持n_hidden=1，執行後發現，增加層數後的預測效果反而不如以前：

第 1 次	expected: 0, predicted: 0 expected: 1, predicted: 0 expected: 1, predicted: 0
第 2 次	expected: 0, predicted: 0 expected: 1, predicted: 0 expected: 1, predicted: 0
第 3 次	expected: 0, predicted: 1 expected: 1, predicted: 1 expected: 1, predicted: 1

為什麼增加深度後反而效果不好呢？增加深度後，訓練的引數個數和梯度迭代的次數也增加了，是不是需要更多的訓練和調整呢？我們嘗試把n_epoch的次數從20提高到200後看看：

第1次	expected: 0, predicted: 0 expected: 1, predicted: 1 expected: 1, predicted: 1
第2次	expected: 0, predicted: 0 expected: 1, predicted: 1 expected: 1, predicted: 1
第3次	expected: 0, predicted: 0 expected: 1, predicted: 0 expected: 1, predicted: 0
第4次	expected: 0, predicted: 0 expected: 1, predicted: 0 expected: 1, predicted: 0

可以看到，訓練次數從20提高到200後效果略好，那麼我們再提高到2000呢？

訓練次數提高到2000後，我們發現測試正確率為100%（不再重複展示結果）。

因此我們看到，對於本章例子中的簡單數字分類，增加網路深度雖然也能提高預測準確率，但同時對計算能力的要求大幅度增加。相對而言，保持一個隱藏層，簡單增加神經元的做法見效更快，而且避免過多增加計算能力需求。

在很長一段時間裡，機器學習都停留在強調寬度、增加神經元階段。關於增加深度，沒有太多考慮，向深度神經網路（Deep Neural Network，DNN）方向發展即可。這是因為演算法本身沒有找到合適的突破場景，沒有找到深度神經網路能夠真正發揮作用的地方；另外，硬體和軟體都沒有提供足夠的計算能力來滿足深度神經網路在實踐中的需要。

但隨著卷積神經網路（Convolutional Neural Network，CNN）在影像分類上的突破，基於深度學習（Deep Learning）的深度神經網路已經成為當前的主流方案，因此相應誕生了各種開發框架，充分發揮硬體和軟體的作用可幫助深度神經網路的構建、訓練和應用。第3章將介紹Keras開發框架，方便大家系統瞭解深度神經網路開發框架的基本使用方法，為後續進行推薦系統、自然語言處理、影像識別等方面的學習和應用做好準備。

2.5　本章小結

本章希望讀者能夠在無須瞭解任何機器學習框架和相關數學背景的前提下，從程式碼入手，直接用最基本的Python程式碼來實現簡單的單層網路，完成機器學習的分類和預測。

本章從最簡單的感知器原理講起，用Python實現了感知器分類，然後迅速進入對線性分類的概念介紹中，用圖示闡述基於機器學習分類的基本原理，並引入了兩個關鍵概念：損失函式與梯度下降；隨後用實際的Python程式碼實現線性迴歸演算法，展示了梯度下降的具體實現過程，並引入隨機梯度下降和mini batch的概念，透過程式碼表現瞭如何對全量資料訓練進行最佳化；最後從單神經元感知器的實現進化到完整神經網路的設計和開發，透過實現包含一個隱藏層的完整神經網路程式碼，將本章的概念結合起來，以座標點的分類為例，使用純Python程式碼實現了包括前向傳播、反向傳播、鏈式法則和權重更新，並可自定義隱藏層的神經元個數的一個靈活的全連線神經網路。在此之上，我們透過實驗對比了增加網路寬度和深度的實際效果，並做了簡單的分析和對比，引出了深度神經網路的概念。

2.6 本章參考文獻

[1] W.McCullock, W.Pitts, 「A Logical Calculus of the Ideas Immanent in Nervous Activity」, 1943

[2] F.Rosenblatt, 「The Perceptron, a Perceiving and Recognizing Automaton」, 1957

[3] H.T.Cheng,et.al, 「Wide & Deep Learning for Recommendation Systems」, 2016

第3章　上手Keras

前兩章介紹了機器學習的基本概念，並透過Python程式碼手動實現了簡單的梯度下降神經網路。可以看到，儘管神經網路的程式碼實現並不是特別複雜，但當我們的層數增多並需要定義不同的引數和演算法實現時，就不能再靠原生的Python程式碼去實現了。無論是對於學術研究還是對於工業級開發，都需要一個封裝了不同的網路結構、引數配置、演算法實現的機器學習開發框架，才能在這基礎上進行真正的工作。

截至2019年10月，比較流行的機器學習開發框架包括TensorFlow、Keras、PyTorch、MXNet等。本書主要介紹基於Keras的機器學習開發，因此在本章首先介紹Keras，方便大家在後續的章節中進行學習和實戰。

本章儘管程式碼不少，但基本上是作為對圖表及概念的補充和說明，不要求讀者必須執行。

3.1　Keras簡介

如前面所提到的，機器學習開發框架有很多產品，並且都在不斷改進和演化。從某種角度而言，現在的機器學習開發框架，很像多年前Windows平臺上開發工具之間的競爭，我們很難說清楚未來哪個框架會成為業界標準，甚至不知道會不會有這樣一個標準。

就2019年來說，在易用性上，PyTorch和Keras當仁不讓，而PyTorch在自定義網路結構方面更靈活一些。從正式的工程開發角度來說，TensorFlow可以說是目前業界的首選。如果我們在GitHub上搜尋一下Keras、PyTorch、TensorFlow，則可以看到引用Keras的程式碼遠超過PyTorch，當然，在生產環境中應用TensorFlow的數量更是遠遠超過Keras。

然而大家公認的是，TensorFlow的開發介面並不友好，它的API介面和基於靜態圖（Graph）的設計思想對需要靈活、方便地除錯的演算法研究來說不太方便，更適合作為演算法確定後的具體最佳化和實現。實際上，Google近期在TensorFlow的推薦活動中，也是反覆建議用Keras做模型研究和實現，用TensorFlow做生產環境開發和部署[1]。

因此在本書中，我們選擇Keras作為演算法框架。要注意一點，Keras本身並不能算是一個完全的機器學習框架，實際上它可以被視為某種底層的機器學習演算法庫的高階「封裝」，對使用者來說是一個更簡單、方便的介面。在原生的Keras中，它實際上需要使用者自行指定並設定對應的機器學習演算法庫（Theano或者TensorFlow）。而在Google現在的TensorFlow 2.0及1.1*x*版本中都已經自帶了Keras，使用者無須自行配置。TensorFlow自帶的Keras可以作為社群版Keras的「超集」，然而，讀者需要注意：二者在一些微小細節上並非完全相容。如果是自定義網路層這樣的較複雜的實現，則二者的程式碼往往並不能直接遷移。為了簡化大家的配置及方便學習，這裡直接使用TensorFlow自帶的Keras。

3.2 Keras開發入門

在第2章的神經網路實現程式碼中，我們看到，要自行實現一個神經網路，需要完成前向傳播、反向傳播、權重更新、梯度計算等一系列工作，不但反覆易錯，而且實驗新的演算法需要較多改動。Keras則提供了良好、易用的介面，讓我們能夠快速搭建一個較為複雜的網路模型。

在Keras中構建和訓練一個神經網路和我們在前面用Python自行實現的步驟類似，但具有更友好的介面定義。

3.2.1 構建模型

我們可以透過如下程式碼直接構建一個網路模型：

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation

model = Sequential([
   Dense(4, input_shape=(2,)),
   Activation('sigmoid'),
   Dense(1),
   Activation('sigmoid'),
])
```

以上程式碼定義瞭如圖3-1所示的網路。

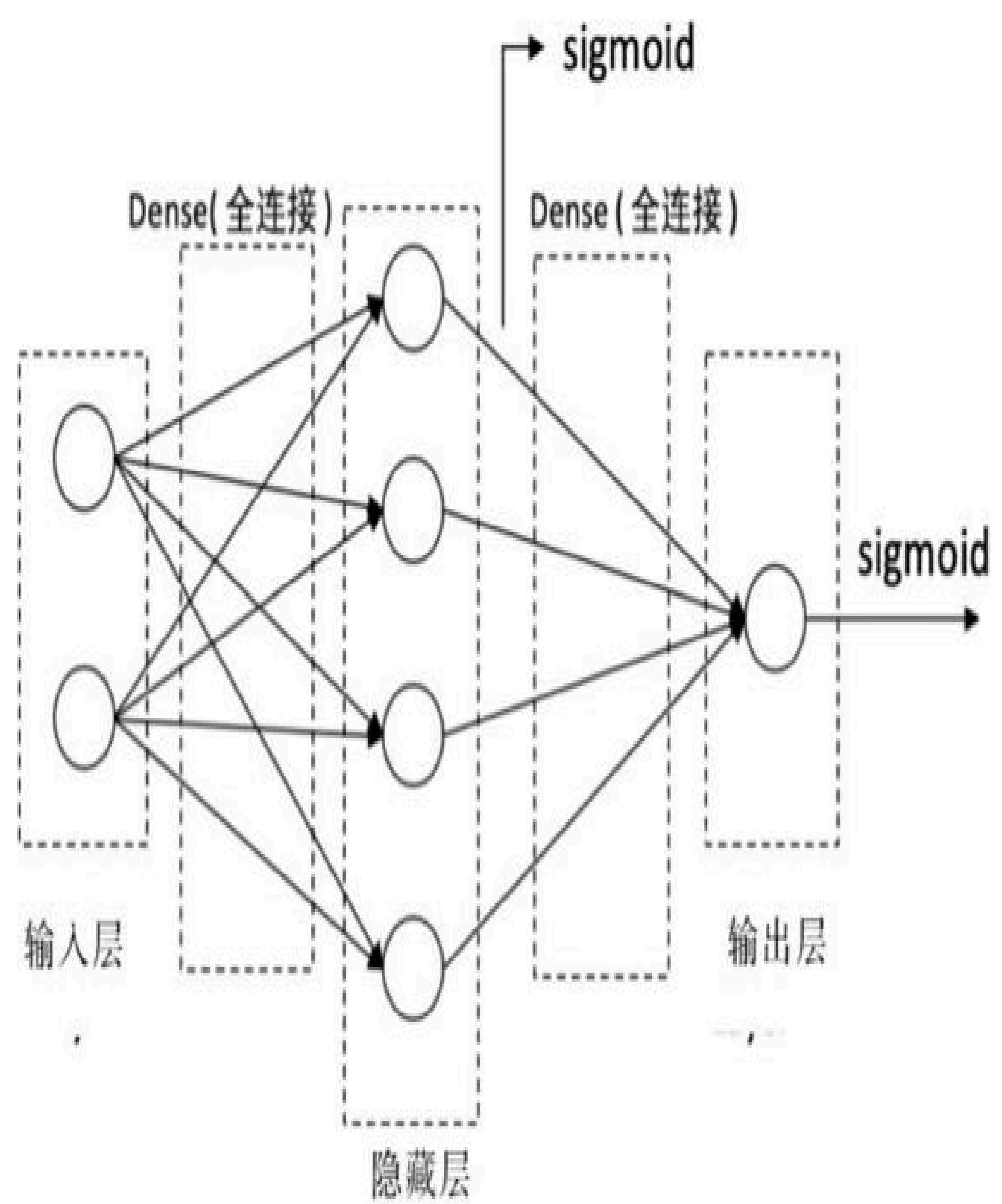


圖3-1　用Keras中的Sequential Model構建簡單的單層神經網路

首先，我們用Dense(4,input_shape=(2,))定義一個全連線層（Dense Layer），其中包含4個神經元，輸入的是一個長度為2的一維陣列。這樣實際上我們就定義了輸入層為兩個輸入，隱藏層的每個神經元的輸入（net_input）都為全連線形式。

然後，我們定義該全連線層的啟用函式（輸出）為sigmoid，這樣就完成了對整個隱藏層的定義。

最後，我們定義輸出層也為全連線形式，輸出包括一個sigmoid啟用函式的結果。因為Sequential Model是一個順序疊加的網路結構，每一層的輸入都是前一層的輸出，所以這裡不必再重複定義輸出層和全連線層的輸入格式，網路會根據前面層的資訊來自動生成。

注意，為了更形象、清晰地表示網路結構，Keras提供了方便的plot_model函式以供呼叫：

```
from tensorflow.keras.utils import plot_model
plot_model(model, to_file='training_model.png', show_shapes=True)
```

執行上述程式碼，在程式碼目錄中可以看到生成了一個新的圖片training_model.png，這是Keras生成的網路結構圖，如圖3-2所示。

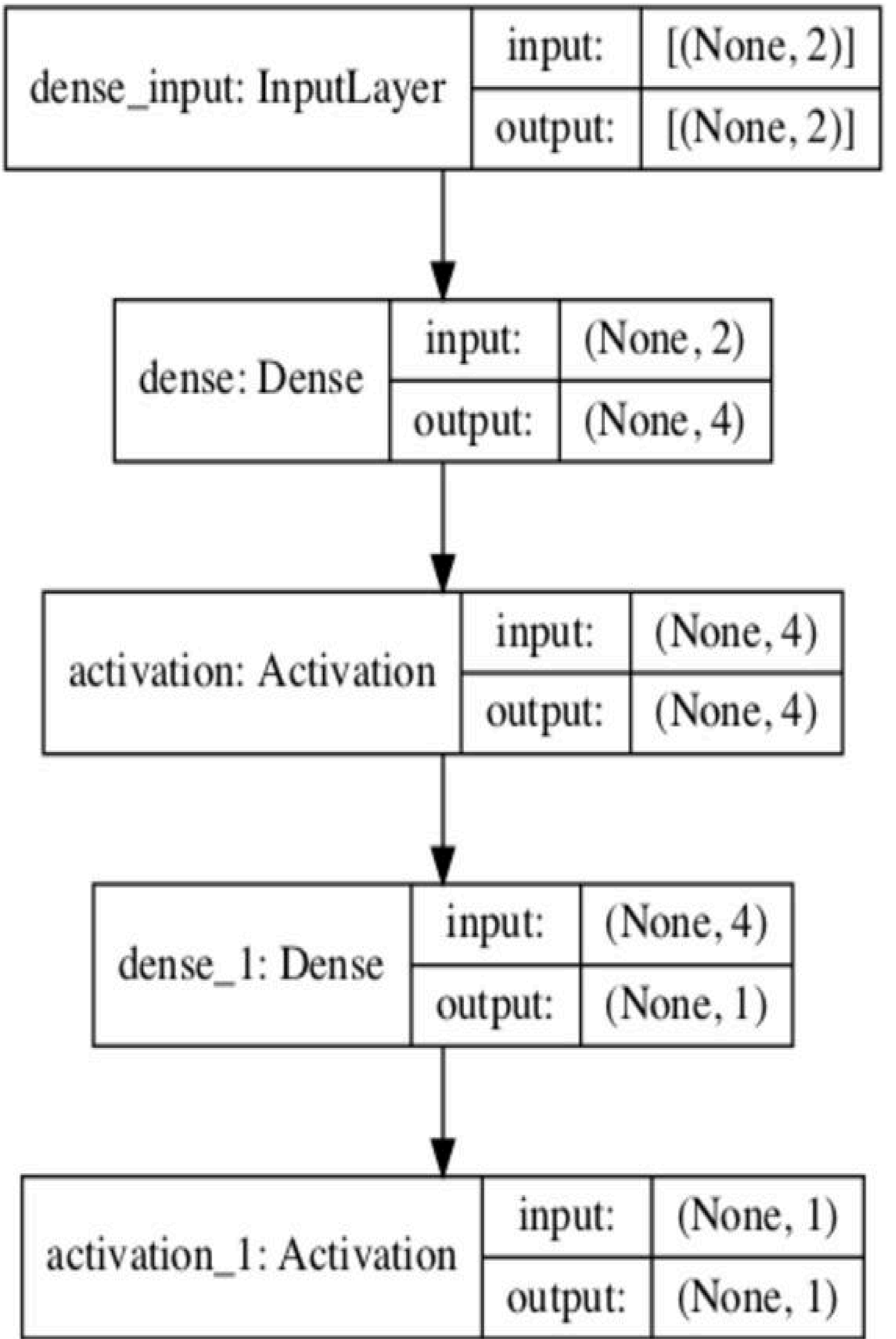

dense_input: InputLayer
input:
[(None, 2)]
output:
[(None, 2)]
dense: Dense
input:
(None, 2)
output:
(None, 4)
activation: Activation
input:
(None, 4)
output:
(None, 4)
dense_1: Dense
input:
(None, 4)
output:
(None, 1)
activation_1: Activation
input:
(None, 1)
output:
(None, 1)

圖3-2　Keras生成的網路結構圖

現在只是完成了對網路模型的定義，參考我們在第2章實現的簡單網路，要真正構建模型還缺乏如learning rate、loss等定義。在Keras中，這些可以透過呼叫compile函式指定：

```
    model.compile(optimizer=tf.train.AdamOptimizer(0.001),loss='mse',metrics=['a
ccuracy'])
```

這裡指定了learning rate為0.001，loss（即損失函式，最終結果和真實結果之間的誤差計算方式）為MSE，將預測準確率作為評價標準。

3.2.2　訓練與測試

我們在上面定義了一個簡單模型：接收兩個輸入，產生一個(0,1)區間的輸出。那麼，我們怎麼使用它呢？

我們很自然地想到繼續用它來實現座標分類：給定一個座標(x,y)，將x小於y的作為一類，將x不小於y的作為一類。上面這個模型能否完成呢？程式碼如下：

```
1  training_number = 100
2  training_data = np.random.random((training_number, 2))
3  labels = [(1 if data[0]<data[1] else 0) for data in training_data ]
4  model.fit(training_data, labels, epochs=20, batch_size=32)
5
6  test_number = 100
7  test_data = np.random.random((test_number, 2))
8  expected = [(1 if data[0]<data[1] else 0) for data in test_data ]
9  error = 0
10 for i in range(0,test_number):
11     data = test_data[i].reshape(1,2)
12     pred = 0 if model.predict(data) < 0.5 else 1
13
14     if (pred != expected[i]):
15         error+=1
16
17 print("totoal errors:{}, accuracy:{}".format(error, 1.0-error/test_number))
```

上面這段程式碼是對該模型的完整訓練和測試。

第1～2行：定義了訓練資料為100條，然後利用NumPy的random函式生成一個隨機的形狀為(100,2)的隨機數向量，即100條訓練資料，其中的每條資料都包括兩個隨機數。

第3行：對訓練資料生成了對應的標籤，對於每條訓練資料，如果第1個數小於第2個數，則標籤為1，否則為0。

第4行：開始訓練，這裡選擇訓練次數為20，batch_size為32（mini batch的概念在第2章講到隨機梯度下降時已經進行了解釋）。

從第6行開始：對訓練好的模型進行測試。首先也定義100條測試資料，第6～8行的程式碼和前面建立訓練資料及標籤的程式碼相同。

第9行：初始化錯誤次數為0。

第10行：開始對測試集中的每條資料都進行測試。因為模型接收的是一個shape為(2,)的向量（類似$[[x_1, x_2]]$的形式，而不能只是$[x_1, x_2]$）。

第11行：對原始資料進行轉置。

第12行：呼叫模型的predict方法並對輸出進行調整。因為模型的輸出經過sigmoid啟用函式對映到(0,1)區間後是類似0.91234、0.25768的機率值，所以我們需要把它再轉換為1、0兩個值。

第14～17行：計算預測值（prediction）和期望值不符的錯誤數，最後列印結果。

執行一次會發現，錯誤率其實挺高：

```
totoal errors:41, accuracy:0.59
```

如何改進？在不修改網路的前提下，無非是要麼增加訓練次數，要麼增加訓練資料。即對第4行model.fit中的epochs引數進行修改，或者對test_number引數進行修改。

實驗結果如表3-1所示。

表3-1　實驗結果

参数设定	total Errors	accuracy
Epochs=500	17	0.83
Epochs=1000	1	0.99
Training_number=1000	28	0.72
Training_number=10000	0	1.00

很快就可以看到，提高訓練次數和增加訓練樣本都很有效。在訓練次數從原本的20次增加到1000次後，在100個測試樣本中只有1個錯誤，準確率達到99%。而在訓練樣本從100條增加到10000條後，準確率更是達到了驚人的100%。

3.3 Keras的概念說明

從上面的例項介紹中，我們認識到Keras使用起來非常簡捷，它就是圍繞著模型和層來構建的，並對其提供了各種可配置的引數，包括啟用函式、梯度下降、損失函式等方面的不同實現策略。那麼，Keras到底為我們提供了多少可選擇的內容呢？讓我們來看看其中的主要內容。

3.3.1 Model

Keras中的Model實際上只包括兩種：Sequential Model及利用input tensor、output tensor建立的Model。

我們在前面的例子中已經展示了Sequential Model的用法。顧名思義，Sequential Model很適合建立層層疊加的神經網路，尤其是如卷積神經網路這樣純粹增加深度的神經網路（在第7章介紹）。然而，對於

更復雜的由多種網路組合而成的如LSTM（在第6章會介紹）和類似Faster RCNN（在第8章介紹）這樣的網路，我們無法透過在前面的網路上直接增加新的隱藏層來實現，往往需要把不同網路的輸入進行組合後，再輸出到新的網路結構，這時就不能再使用Sequential Model了，而需要用Model類來建立。

先來看一組程式碼：

```
1  from tensorflow.keras.models import Model, Sequential
2  from tensorflow.keras.layers import Input, Dense, Activation,
3  concatenate
4  from tensorflow.keras.utils import plot_model
5
6  model1 = Sequential()
7  model1.add(Dense(32, input_shape=(32,), activation='sigmoid'))
8  plot_model(model1, to_file='m1.png', show_shapes=True)
9
10 a = Input(shape=(32,))
11 b = Dense(1, activation='sigmoid')(a)
12 model2 = Model(inputs=a, outputs=b)
13 plot_model(model2, to_file='m2.png', show_shapes=True)
```

在上面的程式碼中，model1透過Sequential方式建立，model2透過自定義Model的inputs和outputs引數建立，然而我們在檢視輸出的模型結構圖時會看到二者其實是一樣的，如圖3-3所示。

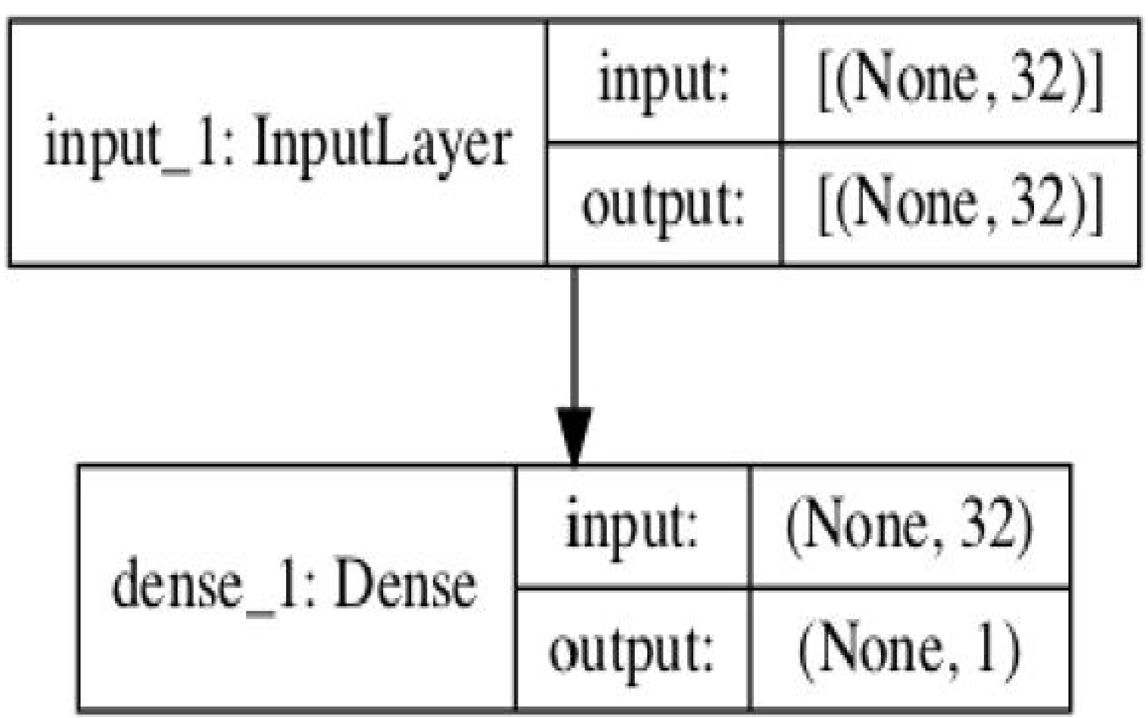


圖3-3　上面的程式碼所生成的兩個模型

然而，如果我們希望有兩個輸入input1和input2，其中，input1需要經過一次全連線層運算後輸出一個數值，再和input2一起輸入新的網路層，該怎麼辦呢？也就是說，我們希望實現一個類似圖3-4的網路。

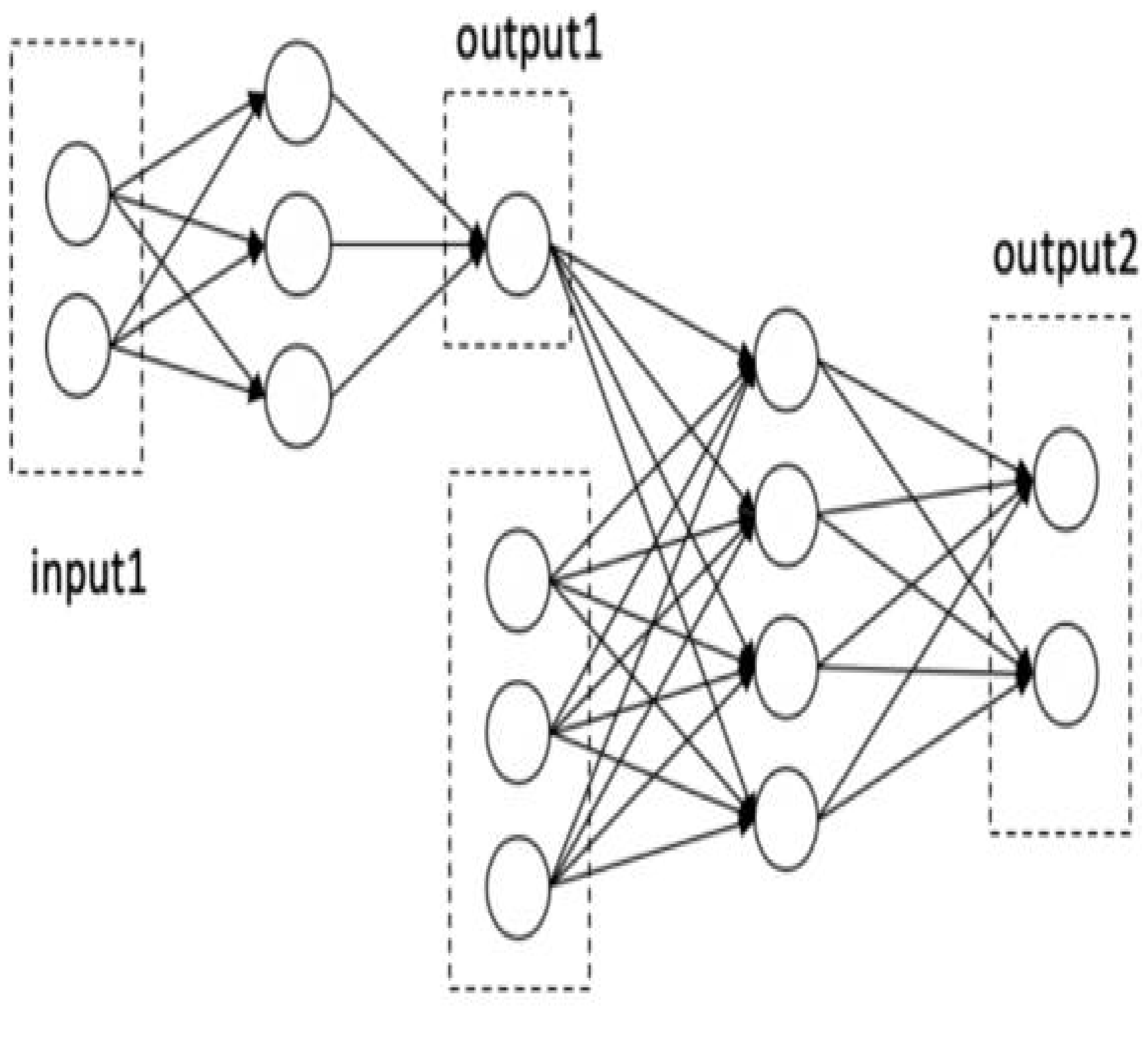


圖3-4　多個不同輸入的神經網路

對如圖3-4所示的網路，我們可以這樣實現：

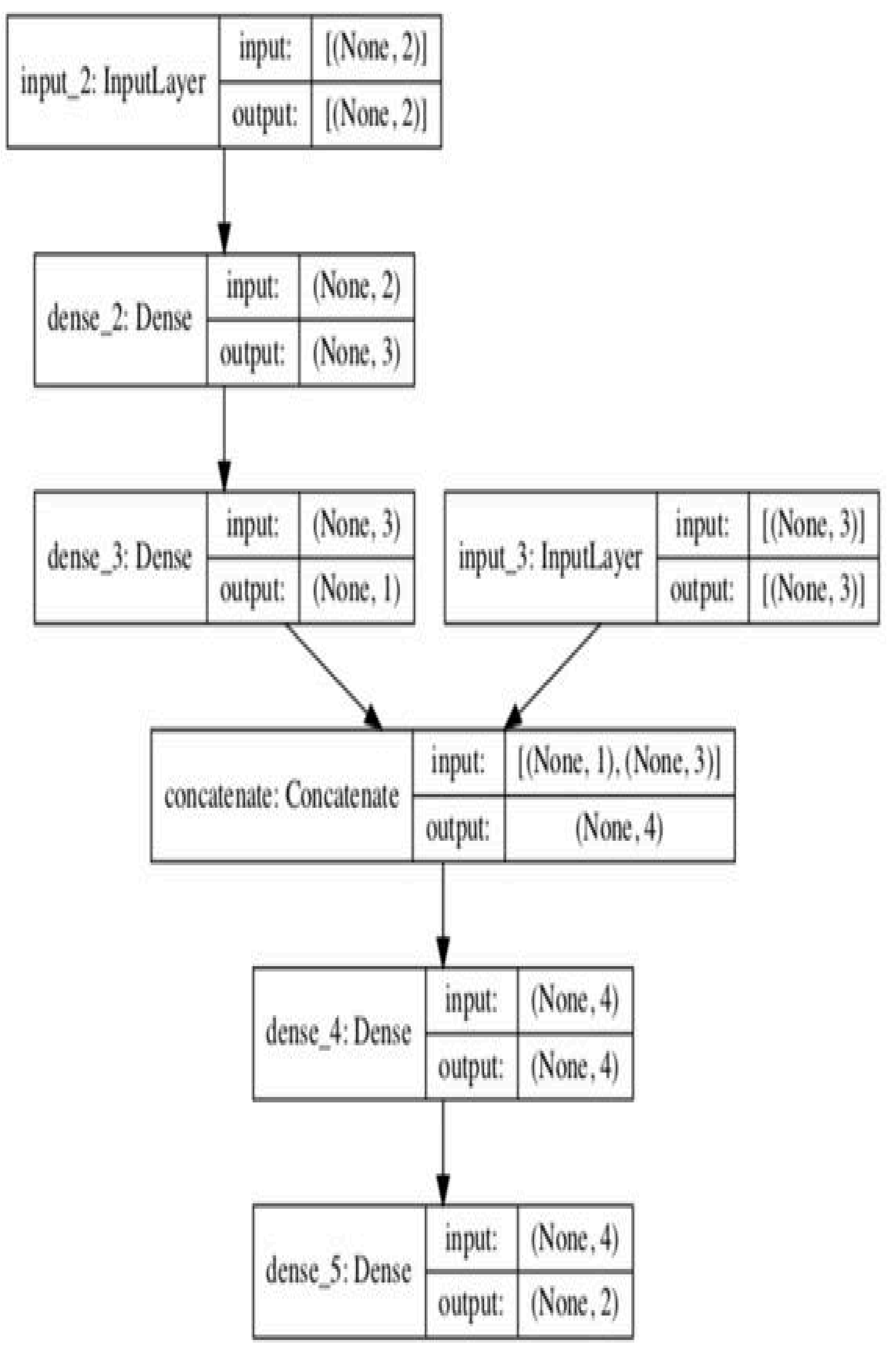
```
1   input1 = Input(shape=(2,))
2   h1 = Dense(3, activation='sigmoid')(input1)
3   output1 = Dense(1, activation='sigmoid')(h1)
4
5   input2 = Input(shape=(3,))
6   new_input = concatenate([output1, input2])
7   h2 = Dense(4, activation='sigmoid')(new_input)
8   output2 = Dense(2, activation='sigmoid')(h2)
9   model3 = Model(inputs=[input1, input2], outputs=output1, output2)
10  plot_model(model3, to_file='m3.png', show_shapes=True)
```

我們看看以上程式碼都做了什麼。

第1～3行：實現從input1到output1的流程。

第5～9行：定義新的輸入input2，並透過concatenate函式拼接output1和input2，形成新的輸入new_input，然後是實現隱藏層和輸入層output2的工作。

第10行：繪製網路結構圖，可以看到所生成的網路（見圖3-5）與圖3-4一致。



input_2: InputLayer
input:
[(None, 2)]
output:
[(None, 2)]
dense_2: Dense
input:
(None, 2)
output:
(None, 3)
dense_3: Dense
input:
(None, 3)
output:
(None, 1)
input_3: InputLayer
input:
[(None, 3)]
output:
[(None, 3)]
concatenate: Concatenate
input:
[(None, 1), (None, 3)]
output:
(None, 4)
dense_4: Dense
input:
(None, 4)
output:
(None, 4)
dense_5: Dense
input:
(None, 4)
output:
(None, 2)

圖3-5　Keras自定義模型生成的圖3-4結構的網路圖

我們也可以同時輸出output1的結果，只需將最後model3的定義修改如下：

```
model3 = Model(inputs=[input1, input2], outputs=[output1, output2])
```

除了基本的Sequential和Model型別，使用者也可以在Python中透過繼承Model類的形式實現自定義模型。在Keras官方網站[2]給出如下一個例子：

```
class SimpleMLP(Model):

    def __init__(self, use_bn=False, use_dp=False, num_classes=10):
        super(SimpleMLP, self).__init__(name='mlp')
        self.use_bn = use_bn
        self.use_dp = use_dp
        self.num_classes = num_classes

        self.dense1 = keras.layers.Dense(32, activation='relu')
        self.dense2 = keras.layers.Dense(num_classes, activation='softmax')
        if self.use_dp:
            self.dp = keras.layers.Dropout(0.5)
        if self.use_bn:
            self.bn = keras.layers.BatchNormalization(axis=-1)

    def call(self, inputs):
        x = self.dense1(inputs)
        if self.use_dp:
            x = self.dp(x)
        if self.use_bn:
            x = self.bn(x)
                return self.dense2(x)
```

可以看到，首先在新的SimpleMLP類的__init__方法中，我們可以直接自定義新的層，並透過構建引數來做一些控制。但是在定義層之後，我們並不會直接將其拼接，比如並沒有在__init__方法中做類似dense2(dense1(x))的運算，僅僅完成了對層的定義。

真正的運算其實被放到了 call()方法裡。我們可以將 call()方法看作模型的前向傳播實現，這裡和前面的例子類似，僅僅是把層之間的資料處理封裝在call方法中而已。

對該自定義模型的處理也很簡單，用下面的偽程式碼即可：

```
model = SimpleMLP()
model.compile(...)
model.fit(...)
```

Keras的Model還包括一些通用的屬性和介面，例如inputs、outputs、layers、get_weights()、save_weights()、compile()、fit()等，可以查閱Keras官方網站，在此不再贅述。

3.3.2 Layer

在前面的Keras例子中我們看到，Keras中的深度學習模型實際上就是各種不同層的疊加和拼接。在前面的例子中已經出現過最常見的Dense、Activation等基本的層型別，那麼Keras還支援哪些層型別呢？

下面講解Keras中核心的層型別。若想了解每個層的構建引數，則可以參考Keras官方網站中的詳細說明。

1．Dense

Dense為最常見的全連線層，實現了output=activation(dot(input, kernel) + bias)的基本操作。該層的構造引數較多，最常見的引數如下。

◎ units：神經元的個數。

◎ activation：選擇啟用函式，預設為None。如果不設定的話，則輸出將不使用任何啟用函式，輸出等於輸入，即activation(x) = x。

◎ use_bias：決定是否使用bias，預設為True，表示會使用bias引數。

對該層的使用，我們在前面的例子裡已經見過多次，這裡對其輸入資料再說明一下。在全連線層（及其他層）的建構函式中並沒有指定輸入資料的結構，這是在建立模型時定義的，例如我們在3.3.1節的程式碼中所看到的：

```
model1 = Sequential()
model1.add(Dense(32, input_shape=(32,), activation='sigmoid'))
…
input1 = Input(shape=(2,))
h1 = Dense(3, activation='sigmoid')(input1)
```

可以看到，model1透過add方法接收了一個額外引數input_shape作為輸入資料的格式定義。也可以透過function call形式專門定義一個輸入層，作為計算中的引數傳給全連線層。無論採用哪種形式，其實都是定義了形狀為(*, 32)或(*, 2)的輸入資料作為模型的接收引數。對於模型而言，實際上傳入的是(batch_size, input_dimension)這樣的形式。例如在前面的keras_sample程式碼中，我們在呼叫model.fit函式訓練時，傳入的資料是(1000,2)這樣的形式，其中的輸入層實際上是一個(2,)的向量。讓我們再回顧一下：

```
model = Sequential([
Dense(4, input_shape=(2,)),
…
])
training_number=1000
training_data = np.random.random((training_number, 2))
…
model.fit(training_data, labels, epochs=20, batch_size=32)
```

2．Activation

在上面的程式碼中，我們看到Activation（啟用函式）有以下兩種形式：

```
model.add(Dense(32))
model.add(Activation('sigmoid'))
```

```
model.add(Dense(32,activation='sigmoid'))
```

這兩種形式其實是等價的。

另外，如果Activation層作為網路的第1層，那麼必須指定input_shape引數。

Keras中的主要Activation如下。

（1）softmax(x, axis=-1)。和sigmoid函式類似，softmax函式也是一種把輸入對映在[0,1]區間的運算。但sigmoid function是處理單個輸入，而softmax函式是處理一組輸入（多分類）。對於k個輸入[x_0, x_1, …, x_{k-1}]，我們可以定義x_i對映為

$$\delta(x_i) = \frac{e^{x_i}}{\sum_j e^{x_i}}$$

其程式碼實現也很直接：

```
import numpy as np
def softmax(inputs):
    return np.exp(inputs) / float(sum(np.exp(inputs)))
```

（2）elu(*x*, alpha=1.0)。其中，*x*是輸入向量，alpha是下面公式中的對應值：

$$R(z) = \begin{cases} z, z > 0 \\ \alpha(e^z - 1), z \leqslant 0 \end{cases}$$

程式碼實現：

```
def elu(x, alpha=1.0):
  return x if x > 0 else alpha*(math.exp(x)-1)
```

對應的曲線圖如圖3-6所示。

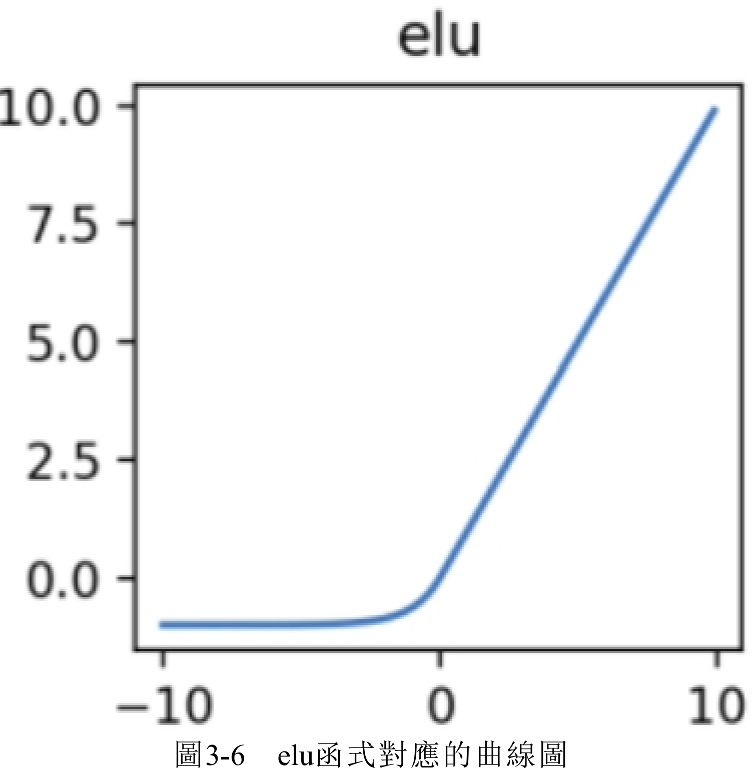


圖3-6　elu函式對應的曲線圖

（3）selu(x, alpha=1.0)。selu函式又被稱為scaled elu函式，其實現和elu函式非常相似，只是新增了一個scale引數，通常設其為1.0507。

程式碼實現：

```
def selu(x):
    scale = 1.0507
    return scale * elu(x)
```

對應的曲線圖如圖3-7所示。

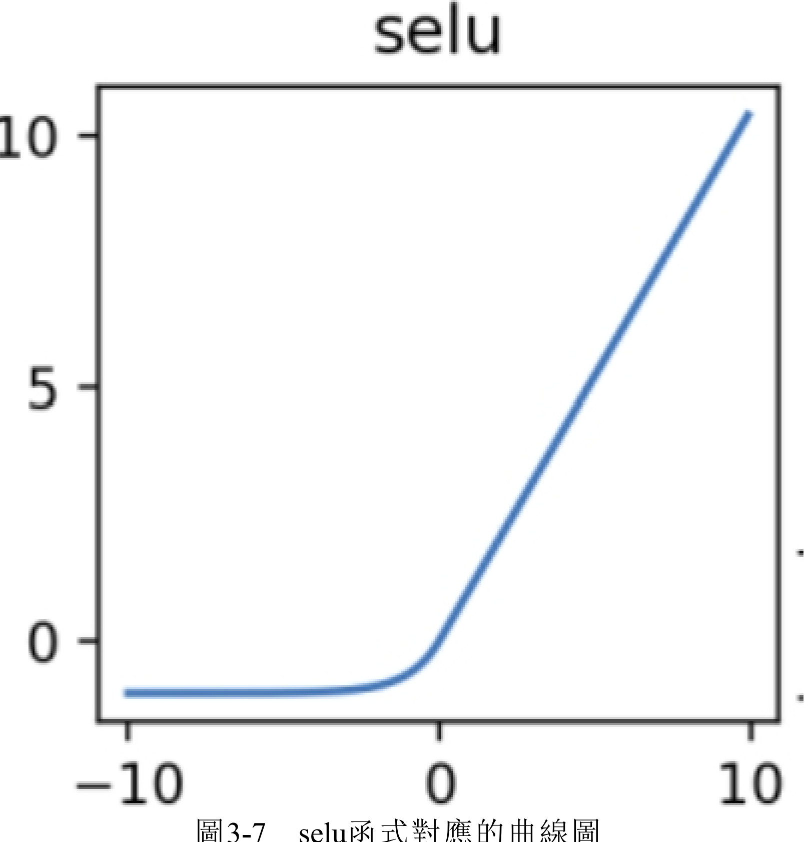


圖3-7　selu函式對應的曲線圖

（4）relu(x, alpha=0.0, max_value=None, threshold=0.0)。relu函式的工作原理非常簡單：relu(x) = max(0, x)。Keras中的relu函式做了一些擴充，額外增加了一些引數，但整體思路未變。

程式碼實現：

```
def relu(x, alpha=0.0, max_value=None, threshold=0.0):
    return max(x, 0)
```

對應的曲線圖如圖3-8所示。

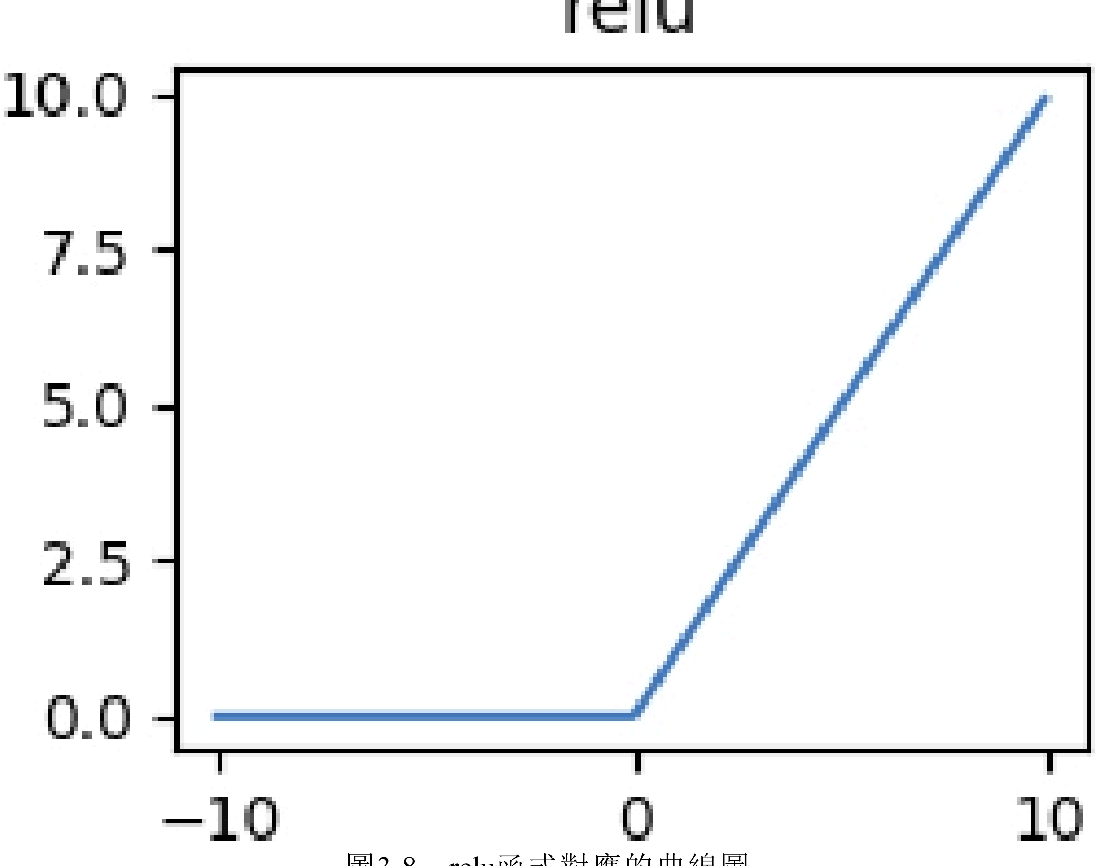


圖3-8　relu函式對應的曲線圖

（5）softplus(x)。

程式碼實現：

```
def softplus(x):
    return math.log(math.exp(x) + 1)
```

對應的曲線圖如圖3-9所示。

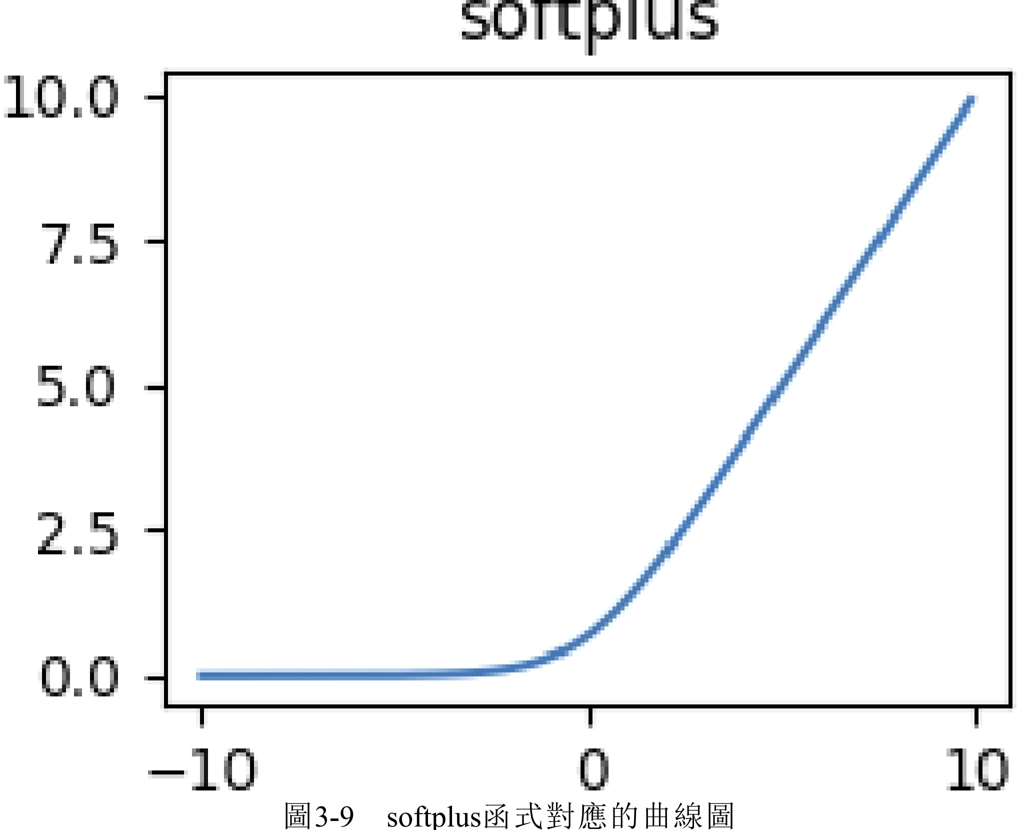


圖3-9　softplus函式對應的曲線圖

（6）softsign(x)。

程式碼實現：

```
def softsign(x):
    return x / (abs(x) + 1)
```

對應的曲線圖如圖3-10所示。

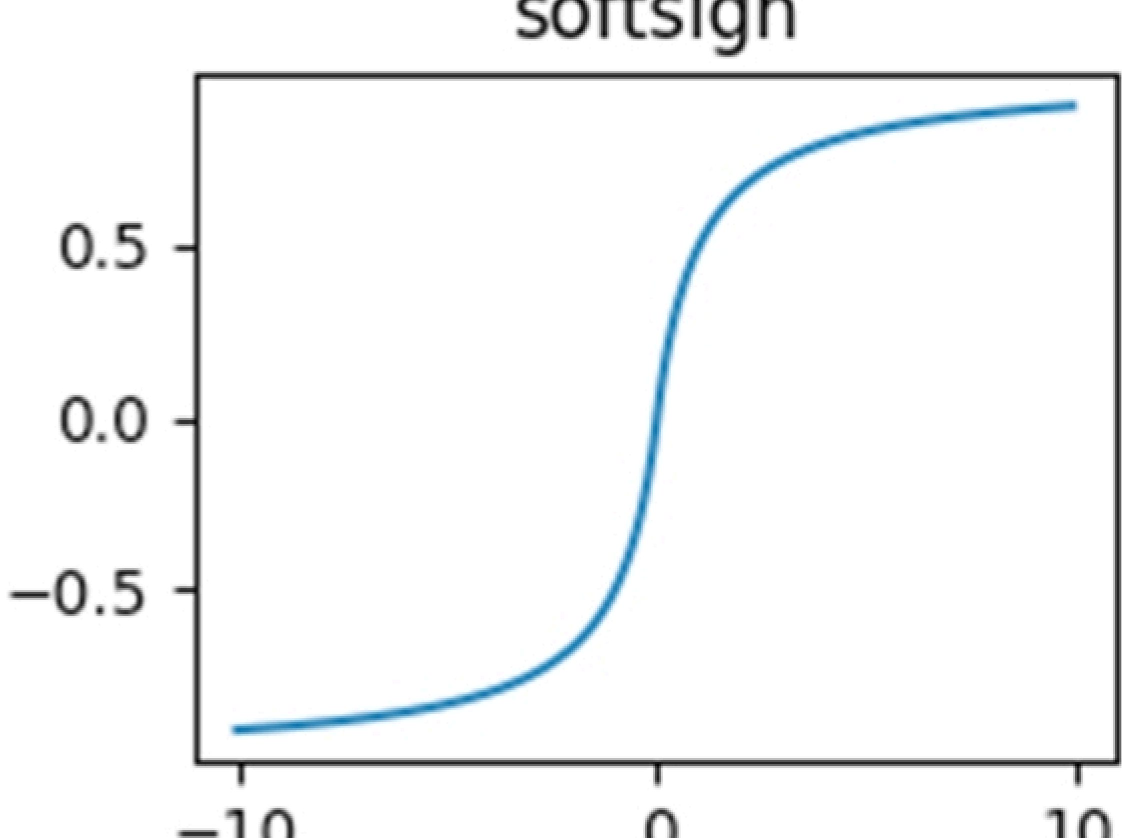


圖3-10　softsign函式對應的曲線圖

（7）tanh(x)。tanh函式的數學表示式為

$$\delta(z)=\frac{e^{z}-e^{-z}}{e^{z}+e^{-z}}$$

函式實現：

```
def tanh(x):
    return (math.exp(x) - math.exp(-x))/ math.exp(x) + math.exp(-x)
```

對應的曲線圖如圖3-11所示。

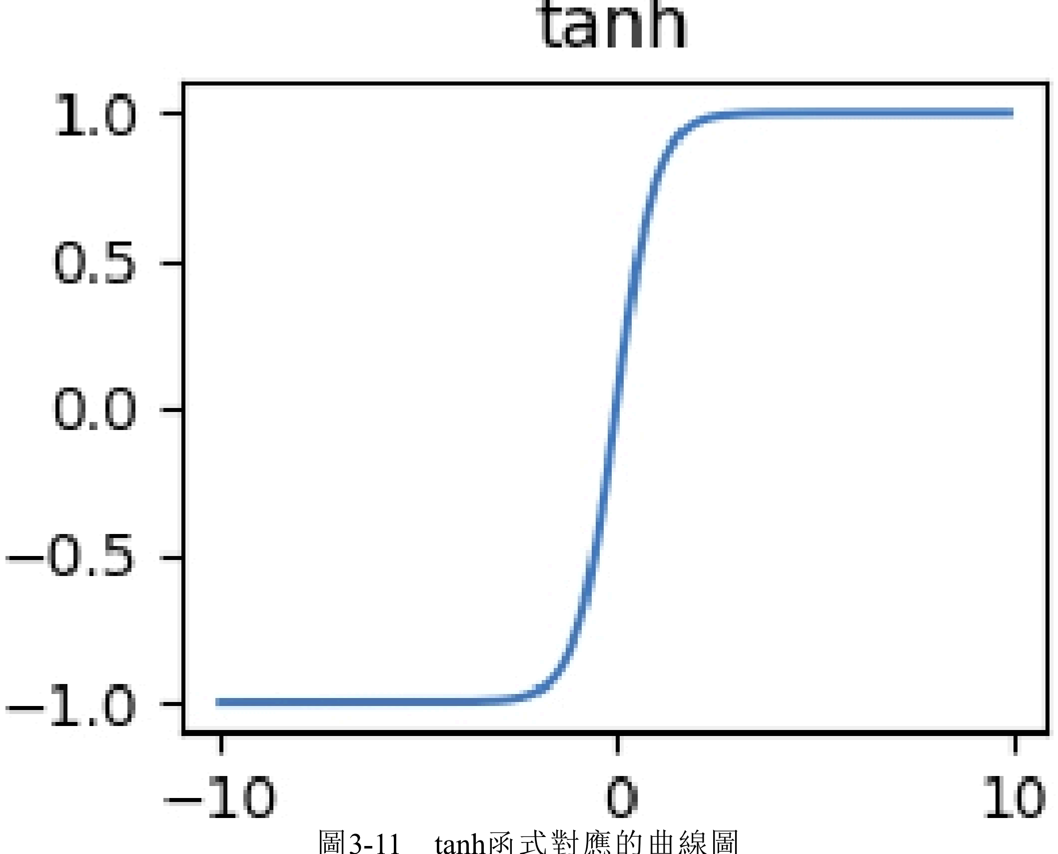


圖3-11　tanh函式對應的曲線圖

（8）sigmoid(x)。這是我們熟悉的sigmoid函式，其數學表示式為

$$\delta(z) = \frac{1}{1 + \mathrm{e}^{-z}}$$

程式碼實現：

```
def sigmoid(x):
    return 1 / (1 + math.exp(-x))
```

對應的曲線圖如圖3-12所示。

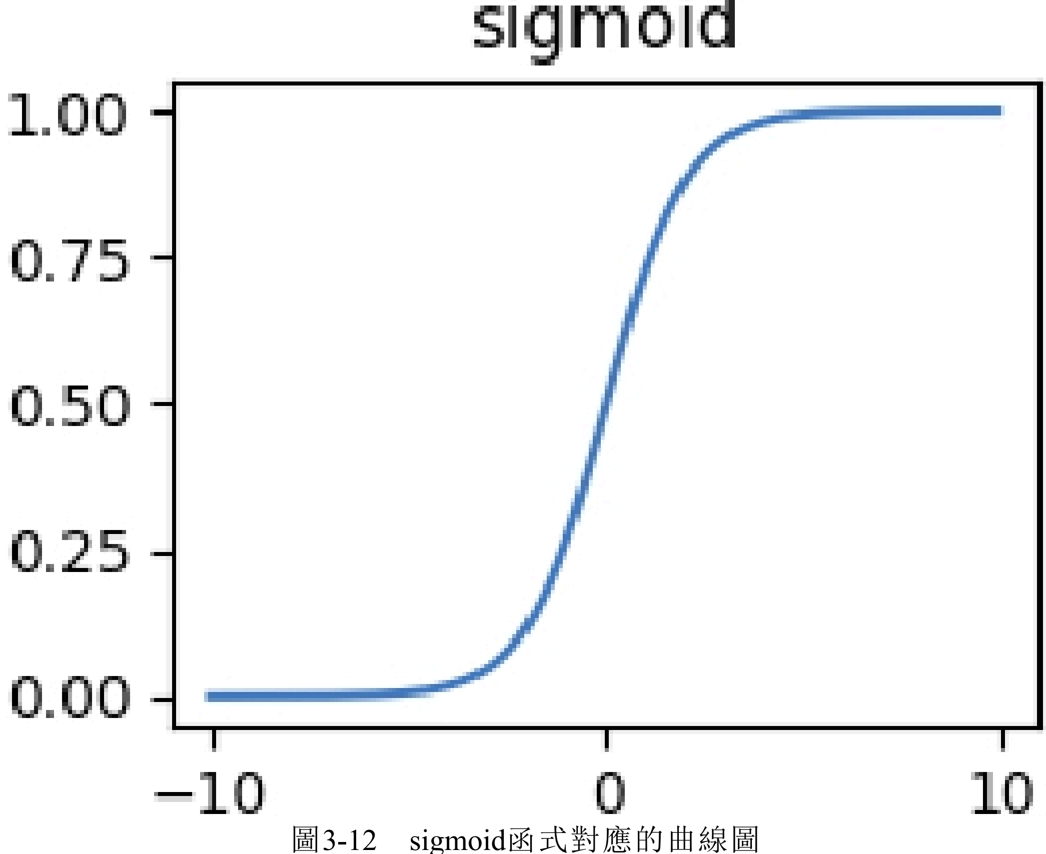


圖3-12　sigmoid函式對應的曲線圖

（9）hard_sigmoid(x)。hard_sigmoid函式是逼近sigmoid函式但是速度快很多的近似實現，其曲線圖和sigmoid函式相當接近。

程式碼實現：

```
def hard_sigmoid(x):
    if x < -2.5:
        return 0
    if x > 2.5:
        return 1
    return 0.2 * x + 0.5
```

對應的曲線圖如圖3-13所示。

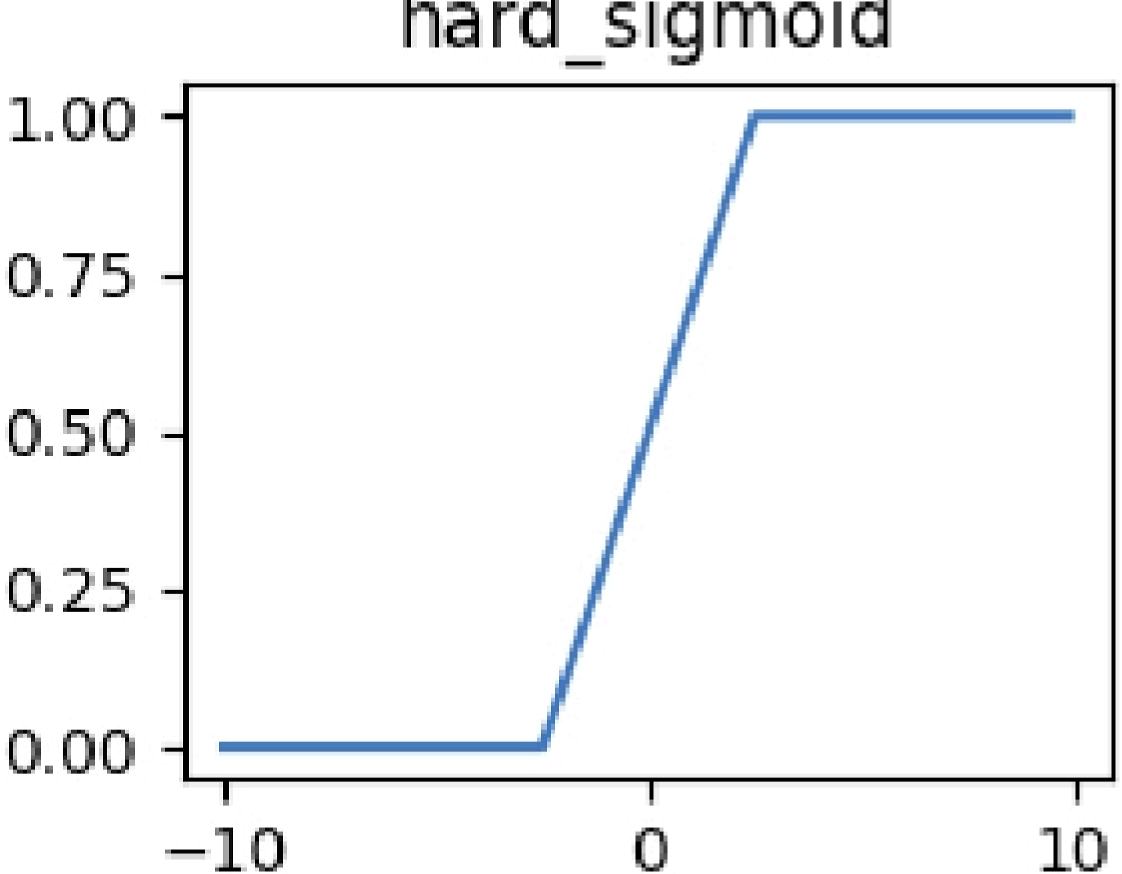


圖3-13　hard_sigmoid函式對應的曲線圖

（10）exponential (x)。它是正常的指數函式。

程式碼實現：

```
def exponential(x):
    return math.exp(x)
```

對應的曲線圖如圖3-14所示。

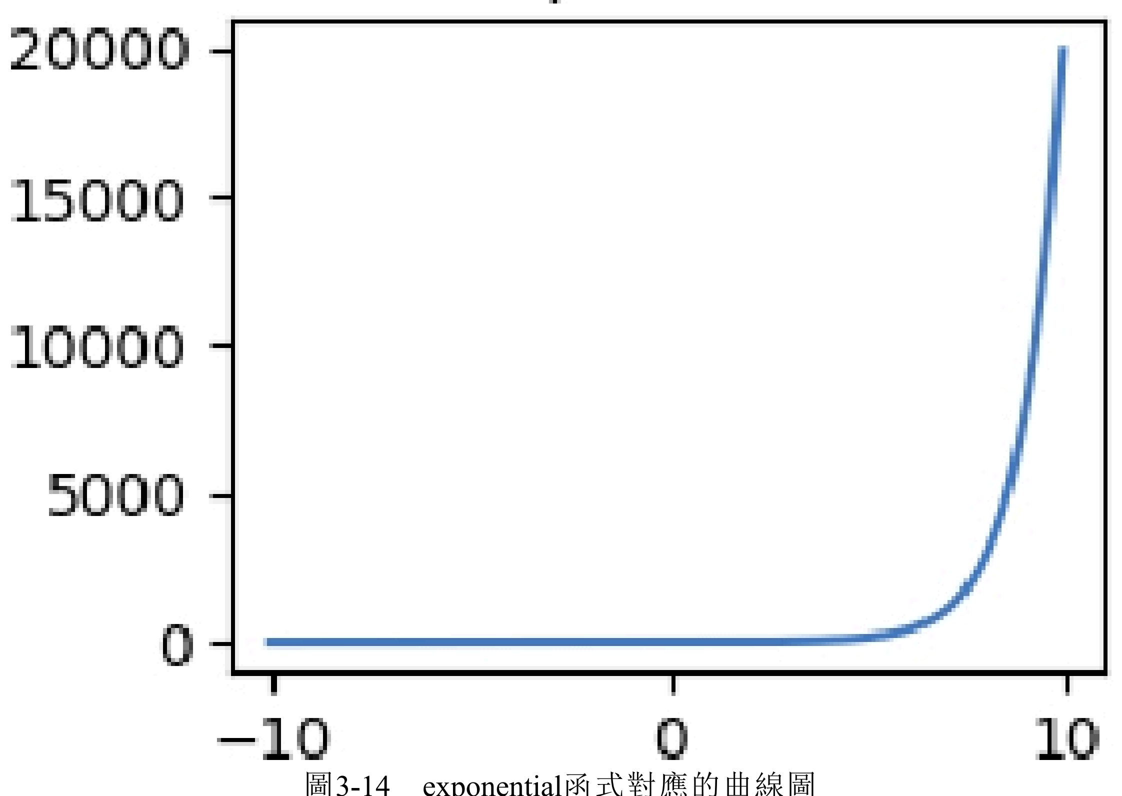


圖3-14 exponential函式對應的曲線圖

（11）linear(*x*)。我們在前面提到，如果在層中不指定啟用函式，那麼會按照線性輸出，也就是直接將輸入作為輸出。

程式碼實現：

```
def linear(x):
    return x
```

對應的曲線圖如圖3-15所示。

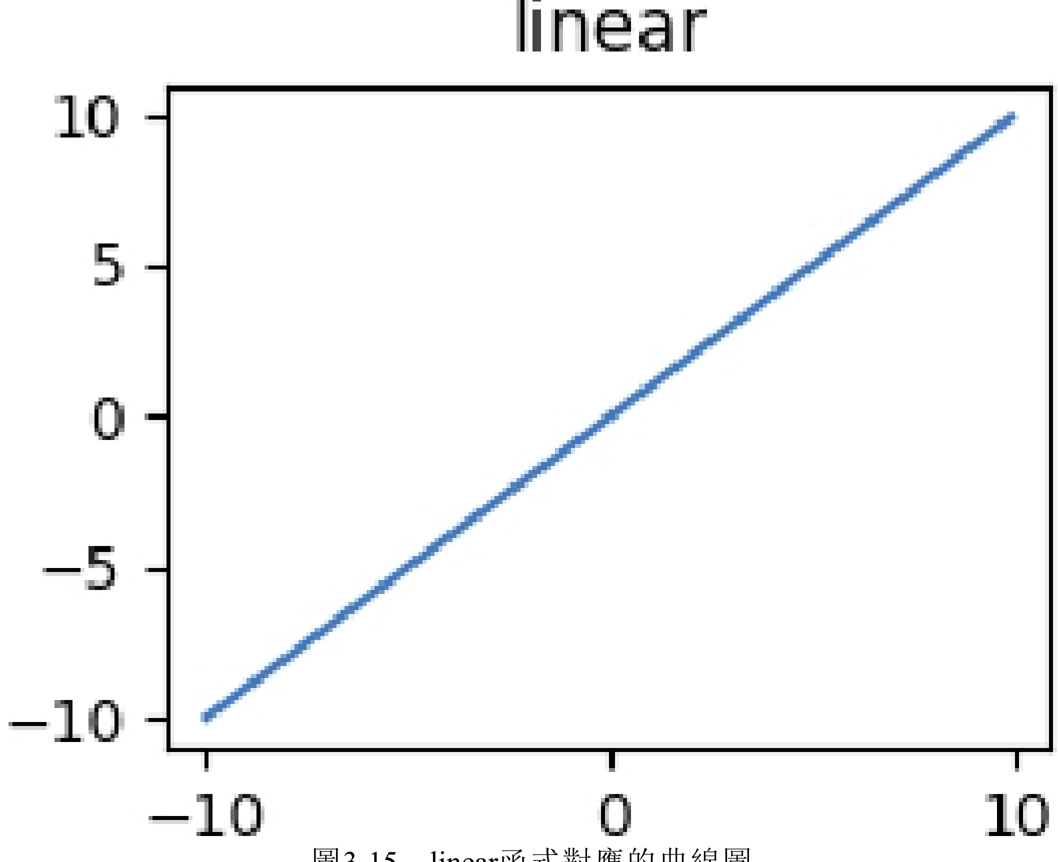


圖3-15　linear函式對應的曲線圖

以上就是Keras中常見的啟用函式，更復雜的如leakyRelu等函式可以參考Keras官方網站。

3．Dropout

Dropout層和全連線層的差別在於：Dropout層在訓練過程中不會選擇所有輸入，而是按照一個比例來隨機選擇輸入，這樣能夠防止過擬合。

在Keras中Dropout的定義很簡單，即Dropout(rate, noise_shape=None, seed=None)。其中，rate為[0,1]區間的浮點數，決定需要忽略的輸入比例；noise_shape是個多維向量，用於控制每次要drop（忽略）的輸入；seed為隨機數種子，為整數。

舉例來說，假如我們的輸入是（batch_size, timesteps, features）：

```
x= [ [[1,2,3,4],[5,6,7,8]], [[11,12,13,14],[15,16,17,18]]]
```

而noise_shape的定義為

```
noise_shape=[[[1,1,0,0]], [[0,0,1,1]]]
```

那麼訓練時Dropout層接收的輸入將是

```
x'= [[[1,2,0,0],[5,6,0,0]], [[0,0,13,14],[0,0,17,18]]]
```

4．Flatten

Flatten層將前面的輸入轉變為一維形式，只接收一個引數data_format，即Flatten(data_format=None)。其中，data_format為string型別，"channels_last"或"channels_first"。其作用是保持輸入中各維度的順序，這樣在進行不同格式的資料轉換時可以方便地處理資料的順序，特別是影像資料的處理。該層預設使用"channels_last"，意味著輸入資料為(batch_size, …, channels)，如果是"channels_first"，就是(batch_size, channels, …)。

下面是Keras官網上的一個小例子：

```
model = Sequential()
model.add(Conv2D(64, (3, 3),
             input_shape=(3, 32, 32), padding='same',))
print(model.output_shape)

model.add(Flatten())
print(model.output_shape)
```

上面的程式碼將輸出：

```
(None, 3, 32, 64)
(None, 6144)
```

我們可以看到，總的輸出數量不變，但在新增Fatten層之前，輸出為一個3×32×64的向量，在新增Flatten之後則成為6144大小的一個一維向量。

5．Input

根據Keras官網的說明，Input方法用於建立Keras中的一個張量（tensor）。我們知道，Keras是執行在其他機器學習框架如TensorFlow、Theano、CNTK等之上的，並在這些框架自身的tensor物件上新增了Keras獨有的屬性如_Keras_shape、_Keras_history等，以便建立Keras自身的模型。

輸入層的引數如下。

◎　shape：這裡舉個例子說明，shape=(32,)意味著我們期待的輸入是一個32維的向量。

◎　batch_shape：一個tuple變數，例如batch_shape=(10,32)表示10個32維的向量，而batch_shape=(None, 32)表示任意數量的32維向量。

◎　name：當前模型中唯一的string命名。

◎　dtype：string格式的資料型別，如"float32""float64""int32"等。

◎　sparse：表示是否是稀疏型別的bool變數。

◎　tensor：如果設定了tensor，則該層不會建立一個臨時的placeholder等待輸入。

該層在前面的例子中已經多次使用，這裡不再贅述。

6．Reshape

顧名思義，Reshape方法的作用是把輸出轉變為給定的目標形狀，其引數只有target_shape，為整數型別的tuple變數。

我們看看它是如何工作的：

```
from tensorflow.Keras.models import Model, Sequential
from tensorflow.Keras.layers import Reshape
import numpy as np
model = Sequential()
model.add(Reshape((3, 4), input_shape=(12,)))

x = np.array([[1,2,3,4,5,6,7,8,9,10,11,12]])
y = model.predict(x)
print(y)
```

執行上面的程式碼，可以看到輸出將一維向量變為一個3×4的矩陣：

```
[[[ 1.  2.  3.  4.]
  [ 5.  6.  7.  8.]
  [ 9. 10. 11. 12.]]]
```

注意，如果給出不正確的目標大小，例如在上面的例子中定義目標大小為6×7，則程式碼會在呼叫model.predict()時報錯。

7．Permute

Permute(dims)中的dims是一個類似(3,2,1)的tuple定義，將輸入資料向量根據dims定義的順序重新進行排序變形。例如：

```
x = np.array([[[1,2,3,4,5,6],[7,8,9,10,11,12]]])
model = Sequential()
model.add(Permute([2,1], input_shape=(2,6)))
y = model.predict(x)
print(y)
```

以上這段程式碼的執行效果如下：

原始数据	变形后
[[[1,2,3,4,5,6],[7,8,9,10,11,12]]]	[[[1. 7.] [2. 8.] [3. 9.] [4. 10.] [5. 11.] [6. 12.]]]

再例如：

```
    x = np.array([[[[1,1],[2,2],[3,3]], [[4,4],[5,5],[6,6]],[[7,7],[8,8],[9,9]],
[[10,10],[11,11],[12,12]]]])

    model = Sequential()
    model.add(Permute([2,3,1], input_shape=(4,3,2)))
    y = model.predict(x)
    print(y)
```

以上這段程式碼的執行效果如下：

原始数据	变形后
[[[[1,1],[2,2],[3,3]], [[4,4],[5,5],[6,6]], [[7,7],[8,8],[9,9]], [[10,10],[11,11],[12,12]]]]	[[[[1. 4. 7. 10.] [1. 4. 7. 10.]] [[2. 5. 8. 11.] [2. 5. 8. 11.]] [[3. 6. 9. 12.] [3. 6. 9. 12.]]]]

8．RepeatVector

重複輸入n次。例如：

```
x = np.array([[1,2,3,4]])
model = Sequential()
model.add(RepeatVector(3, input_shape=(4,)))
print(model.predict(x))
```

以上程式碼將輸出：

```
[[[1. 2. 3. 4.]
  [1. 2. 3. 4.]
  [1. 2. 3. 4.]]]
```

9．Lambda

Lambda層可以直接把一個表示式作為引數傳入，定義該層的功能：

```
Lambda(function, output_shape=None, mask=None, arguments=None)
```

該層的引數如下。

◎　function：接收輸入tensor作為第1個引數並隨後執行。

◎　output_shape：只對Theano框架有用，這裡不做描述。

◎　mask：遮蔽Embedding的輸入。

◎　arguments：需要傳入function的額外引數。

示例如下：

```
x = np.array([[1,2,3,4]])
model = Sequential()
model.add(Lambda(lambda x:x*2, input_shape=(4,)))
print(model.predict(x))
```

以上程式碼將輸出：

```
[[2. 4. 6. 8.]]
```

我們也可以定義較複雜的function傳入，下面是一個略微複雜的例子：

```
def calculation(tensors):
    output1 = tensors[0]-tensors[1]
    output2 = tensors[0]+tensors[1]
    output3 = tensors[0]*tensors[1]
    return [output1, output2, output3]

input1 = Input(shape=[4,])
input2 = Input(shape=[4,])

layer = Lambda(calculation)
out1, out2, out3 = layer([input1, input2])
model = Model(inputs=[input1, input2], outputs=[out1, out2, out3])

x1 = np.array([[1,2,3,4]])
x2 = np.array([[1,1,1,1]])
print(model.predict([x1, x2]))
```

以上程式碼將輸出：

```
[array([[0., 1., 2., 3.]], dtype=float32), array([[2., 3., 4., 5.]],
dtype=float32), array([[1., 2., 3., 4.]], dtype=float32)]
```

在這個例子中，我們首先定義了函式calculation，對輸入的tensor list做了簡單的運算，並返回3個結果。

然後定義了兩個輸入層input1和input2，並定義了Lambda層layer，把上面定義的calculation函式直接作為Lambda function傳入。這裡我們用Keras的function call形式建立了模型，方便定義多個輸入和多個輸出，而不是用Sequential型別（只支援單個輸出）。

最後進行測試，利用x1和x2作為資料輸入，可以看到最後的執行結果符合calculation函式的期望值。

10．ActivityRegularization

其形式為ActivityRegularization(l1=0.0, l2=0.0)，其中的引數如下。

◎ l1：L1正則化引數，浮點正數。

◎ l2：L2正則化引數，浮點正數。

該層在輸出中加上L1或L2正則化引數，不影響輸出資料格式。

11．Masking

其形式為Masking(mask_value=0.0)。

對於其中的引數mask_value，如果是None則忽略，否則，如果在輸入資料中某個timestep的所有feature都等於mask_value，則將該timestep的值全部置0。

舉例如下：

```
model = Sequential()
model.add(Masking(1, input_shape=(4,3)))
x=np.array([[[1,2,3],
          [1,1,1],
               [7,8,9],
          [10,11,12]]])
print(model.predict(x))
```

在這個例子中，我們將輸入視為時間序列資料，在timestep等於1時為[1,2,3]，在timestep等於2時為[1,1,1]，以此類推。我們在建立Masking層時設定mask_value為1，即當某個timestep的所有feature都是1時，忽略該次輸入，因此上面這段程式碼的輸出如下：

```
[[[ 1.  2.  3.]
  [ 0.  0.  0.]
  [ 7.  8.  9.]
  [10. 11. 12.]]]
```

可以看到，feature 為[1,1,1]的部分已經全部被置為[0,0,0]。同樣，我們修改輸入和引數後再看看：

```
model = Sequential()
model.add(Masking(11, input_shape=(4,3)))
x=np.array([[[1,2,3],
             [4,5,6],
             [7,8,9],
             [11,11,11]]])
print(model.predict(x))
```

因為[11,11,11]和這次的mask_value=11匹配，因此輸出如下：

```
[[[1. 2. 3.]
  [4. 5. 6.]
  [7. 8. 9.]
  [0. 0. 0.]]]
```

12. SpatialDropout

Keras的核心層還包括SpatialDropout1D、SpatialDropout2D、SpatialDropout3D，其中，SpatialDropout1D和Dropout的作用是一樣的，而SpatialDropout2D和SpatialDropout3D是面向輸入特徵feature map為2D和3D時的拓展，在此不做過多描述。

13. 其他擴充套件層

前面介紹了Keras的核心層型別，在實際應用中，我們會遇到更多的層型別。在這裡就不一一講解了，只把其他擴充套件型別做一個大致介紹，如表3-2所示，具體細節和使用方式請參考Keras官方網站。

表3-2　對其他擴充套件型別的大致介紹

<table>
<tr><th>类　型</th><th>介　绍</th></tr>
<tr><td>Convolutional Layer</td><td>卷积层，对输入数据进行卷积运算。包括以下类型：
● Conv1D
● Conv2D
● Conv3D
以上类型的变种有 DepthwiseConv2D、Cropping2D、UpSampling3D 等</td></tr>
<tr><td>Pooling Layer</td><td>池化层，也可将其视为采样。包括以下类型：
● MaxPooling1D/2D/3D
● AveragePooling1D/2D/3D
● GlobalMaxPooling1D/2D/3D
● GlobalAveragePooling1D/2D/3D</td></tr>
<tr><td>Locally-Connected Layer</td><td>Locally Connected Layer 和 Convolutional Layer 很相似，差别在于权重 weights 并不共享，包括以下两类：
● LocallyConnected1D
● LocallyConnected2D
由于 Locally Connected Layer 不共享权重，所以它比 Convolutional Layer 需要更多的参数。实际上，它需要对图像上几乎每个点都创建一组 filter</td></tr>
<tr><td>Recurrent Layer</td><td>循环层，多被用于循环神经网络。主要包括：
● RNN
● SimpleRNN</td></tr>
</table>

續表

类　型	介　绍
	● GRU ● LSTM ● ConvLSTM2D ● ConvLSTM2DCell ● SimpleRNNCell ● GRUCell ● LSTMCell ● CuDNNGRU ● CuDNNLSTM
Embedding Layer	Embedding Layer 常见于 NLP 方面的应用，主要用于在训练时对输入词语 token 建立 index（大小由 input_dim 确定）到用户定义的向量空间（大小由 output_dim 确定）的映射表，然后在预测时对每个输入的 token 获取对应的向量
Merge Layer	Merge Layer 包括对网络层之间的多种运算处理，包括 Add、Subtract、Multiply、Average、Maximum、Minimum、Concatenate、Dot 等，我们在前面的部分代码示例中已经见过它
Normalization Layer	主要是 BatchNormalization Layer，其功能是在每一批训练数据中都对前一层的激活函数输出做统一的归一化，保证其平均值接近 0 而标准方差（Standard Deviation）趋近 1
Noise Layer	对数据增加干扰噪声，Noise Layer 只在训练期间起作用

14．自定義層

Keras作為機器學習演算法研究人員最常用的開發框架之一，自然需要能夠讓研究人員自定義自己所需實現的神經網路結構，例如對自定義層的支援。在本書參考文獻[2]中描述了自定義層的必要實現：

```
class MyLayer(Layer):
    def __init__(self, output_dim, **kwargs):
        self.output_dim = output_dim
        super(MyLayer, self).__init__(**kwargs)

    def build(self, input_shape):
        # 为该层创建可以进行训练调节的权重变量
        self.kernel = self.add_weight(name='kernel',
                                      shape=(input_shape[1], self.output_dim),
                                      initializer='uniform',
                                      trainable=True)
        super(MyLayer, self).build(input_shape)  # 确保在最后调用该方法

    def call(self, x):
        return K.dot(x, self.kernel)


    def compute_output_shape(self, input_shape):
        return (input_shape[0], self.output_dim)
```

我們看到，要定製（繼承）Keras層，就需要實現以下幾個方法。

◎　__init__：類初始化，定義輸出。

◎　build(input_shape)：在這個方法中定義權重，如在上面的例子中使用add_weight函式來新增一個可訓練的權重變數。

◎　call(x)：這其實是Forward Pass前向計算的實現。

◎　compute_output_shape(input_shape)：這個實現不是一定被需要的，但是通常會在這裡定義，以便Keras自動推導輸入向量的形狀。

下面簡單實現了一個自定義層，將輸入向量乘以2：

```
class SpecialLayer(Layer):
    def __init__(self, output_dim, **kwargs):
        self.output_dim = output_dim
        super(SpecialLayer, self).__init__(**kwargs)

    def build(self, input_shape):
        super(SpecialLayer, self).build(input_shape)
    def call(self, x):
     return x*2

    def compute_output_shape(self, input_shape):
        return (input_shape[0], self.output_dim)

model = Sequential()
model.add(SpecialLayer(1, input_shape=(2,)))

print(model.predict(np.array([[3,4]])))
```

輸出如下：

```
[[6. 8.]]
```

3.3.3　Loss

我們在前面的例子中主要使用的是MSE作為代價函式（Cost Function）。在很多場合下，我們說的代價函式就是損失函式。更嚴格地說，損失函式指單個訓練樣本的預測誤差，而代價函式指所有訓練樣本的誤差平均值。

需要補充說明的是，損失函式和將要介紹的最佳化函式optimizer是編譯模型時需要指定的必需引數。我們在前面講到如何在模型中新增層之後，只是搭建了模型的基本結構，能夠完成前向傳播（因此我們在前面的例子中可以新增層後直接呼叫predict方法）。但要完成模型訓練，就一定要指定損失函式計算誤差，以及最佳化函式optimizer指定權重最佳化的方法，例如：

```
Model.compile(loss='mean_squared_error', optimizer='sgd')
```

上面這行程式碼就指定了MSE的損失函式和利用sgd（指Stochastic Gradient Descent，隨機梯度下降）做隨機梯度下降最佳化。

我們常常會聽到L1 Loss和L2 Loss的說法。其實這些概念很簡單，L1 Loss是計算真實值和預測值之間的絕對誤差之和，而L2 Loss是計算兩者間誤差的平方和。我們在後面的內容中可以看到這兩類損失函式有各種不同的實現。一般來說，我們傾向於使用L2 Loss型別，但如果在資料集中存在一些較大的誤差，則也會考慮使用L2 Loss。

在Keras中提供了多種損失函式供我們使用。這裡對常見的類別做一些介紹，如表3-3所示。

表3-3　常見的類別介紹

mean_squared_error
* L2 Loss。
* 最常见的损失函数，通用性很强，但准确率不太高。
$loss = \frac{\sum_{i=1}^{n}(y_i - y_i')^2}{n}$

續表

mean_absolute_error

* L1 Loss。

* 适用于不需要在意误差的正负关系，或者较大误差和较小误差的影响相似的情况。

$loss = \frac{\sum_{i=1}^{n} |y_i - y_i'|}{n}$

mean_absolute_percentage_error

$loss = \frac{100\%}{n} \sum_{i=1}^{n} \left| \frac{y_i - y_i'}{y_i} \right|$

mean_squared_logarithmic_error

* 用于希望对较大目标值的误差影响减小时。

$loss = \frac{1}{n} \sum_{i=1}^{n} (\log(y_i + 1) - \log(y_i' + 1))^2$

hinge

* 常见于 SVM 模型中，用于解决二分类问题。

$loss = \max(0, 1 - t \cdot y)$, $t = \pm 1$

squared_hinge

* 对 hinge 结果进行平方运算，有助于平滑误差结果。

$loss = \sum_{i=1}^{n} (\max(0, 1 - y_i \cdot y_i')^2)$

logcosh

* logcosh 其实结果和 MSE 很相似，但不会像 MSE 那样被偶尔较大的误差严重影响。

$loss = \sum_{i=1}^{n} \log(\cosh(y_i^p - y_i))$, where $\cosh(x) = \frac{e^x + e^{-x}}{2}$

categorical_crossentropy

* 在目标为 One-hot encoding 时使用，例如[0,1,0]、[1,1,0]等。

$loss = -\sum_{j=1}^{M} \sum_{i=1}^{N} (y_{ij} \cdot \log(y_{ij}'))$

sparse_categorical_crossentropy

* 在目标为整数时使用，例如[1,2,3]、[11,15,20]等。

續表

binary_crossentropy * 一般被用于解决二分类问题。 $loss = -\sum_{i=1}^{n} y_i' \log y_i + (1 - y')\log(1 - y')$
kullback_leibler_divergence * 评估目标值概率分布 p 和预设概率分布 q 的差异。 $D_{kl}(p\|\|q) = \sum_{i=1}^{n} p(x_i)\log\frac{p(x_i)}{q(x_i)}$
Poisson * 在假设目标值服从于泊松分布时使用。 $loss = \frac{1}{N}\sum_{i=1}^{N}(y_i' - y_i \log y_i')$
cosine_proximity * 用于当我们要比较两个向量的距离（角度差异），而向量本身大小（长度）并不重要时。典型例子包括 NLP 中词语或句子向量的比较。 $loss = \frac{\sum_{i=1}^{n} y_i \cdot y_i'}{\sqrt{\sum_{i=1}^{n} y_i^2} \cdot \sqrt{\sum_{i=1}^{n} {y_i'}^2}}$

我們還可以根據主要用途（分類和迴歸）對損失函式分類，如表3-4所示。

表3-4　根據主要用途對損失函式分類

分类（Classification）	回归（Regression）
KL Divergence	MSE
Hinge	MAE
Crossentropy	Logcosh

1．optimizer

前面提到在模型訓練時一則需要定義損失函式，二則需要定義最佳化函式optimizer。

為什麼我們要說最佳化函式而不是直接說梯度下降呢？因為其實我們要做的是最佳化網路權重並擬合目標值，梯度下降只是其中一種最基本的方法而已，在Keras中有很多其他最佳化方法可供選擇。對於其他最佳化方法，我們在這裡做一個簡單介紹，如表3-5所示。注意，相對於損失函式，optimizer數學公式的含義難以用一兩行文字描述，其中涉及的數學概念也較多，這裡就不再列舉公式了，感興趣的讀者可以自己查閱。

表3-5　其他最佳化方法

SGD(lr=0.01, momentum=0.0, decay=0.0, nesterov=False)

这是前面讲过的随机梯度下降的实现。

- lr：学习率，大于等于 0 的浮点数。
- momentum：大于等于 0 的浮点数，可以加速相关方向的梯度下降并减弱一些摆动的影响。momentum 是梯度下降计算的一种优化处理，可以将其类比为重力加速度，在下降方向上会越来越快地到达底部（最优值），其具体细节在这里不做描述。
- decay：大于等于 0 的浮点数，在每次梯度更新时都减小学习率。
- nesterov：True 或 False，是否采用 Nesterov Momentum

Adagrad(lr=0.01, epsilon=None, decay=0.0)

Adagrad 中的学习率根据参数会自适应调节（Adaptive Learning Rate），对常用参数会进行较小的调节，对不常用的参数调整幅度则偏大。这种方式很适合大型神经网络对稀疏数据的处理。

同样，官方也建议除 learning rate 外，保留 Adagrad 的默认参数值

Adadelta(lr=1.0, rho=0.95, epsilon=None, decay=0.0)

Adadelta 是对 Adagrad 的改良。Adagrad 尽管能自动调节参数，但调节方式比较简单，Adadelta 则对 learning rate 的调节做了更多的控制和优化

RMSprop(lr=0.001, rho=0.9, epsilon=None, decay=0.0)

RMSprop 也是对 Adagrad 的改良，避免 learning rate 快速消失。在此算法中，learning rate 是自动调节的，并对每个参数都设置一个不同的 learning rate。

官方建议保留参数为默认值（除了 learning rate）。

RMSprop 通常被用于循环神经网络

Adam(lr=0.001, beta_1=0.9, beta_2=0.999, epsilon=None, decay=0.0, amsgrad=False)

Adam 也是一个为每个参数都计算一个独立的自适应 learning rate 的方法，以改善 learning rate 快速消失的情况。和 RMSprop 相比，Adam 对稀疏梯度的效果比较好，而 RMSprop 更适合变化较大的在线数据等。另外，Adam 的计算量较小，对内存容量的要求较小，也是最受欢迎的优化算法之一

Adamax

同样是自适应的随机梯度下降算法之一，是 Adam 算法的变种，其优势在于其对类似 learning rate 这样的超参数选择不敏感

Nadam

Nadam 是 Adam 和 NAG（一种 momentum 算法的优化实现，用于加速梯度计算）的混合。Nadam 可被用于噪声较大、不够平滑的梯度计算中

2．Dataset

最後，我們談談Keras中自帶的一些資料集，如表3-6所示。這些資料集被廣泛用於各種研究領域及教學使用，是方便快速驗證演算法的有效工具。在安裝Keras時並不會帶上這些資料集，在引用它們時會自動下載。

表3-6　Keras中自帶的一些資料集

CIFAR10

用于图片分类，共有 5 万张 32×32 大小的图片训练集及 1 万张测试图片、10 个类别

调用方式：

```
from tensorflow.Keras.datasets import cifar10
(x_train, y_train), (x_test, y_test) = cifar10.load_data()
```

CIFAR100

用于图片分类，共有 5 万张 32×32 大小的图片训练集及 1 万张测试图片。和 CIFAR10 不同的是它有 100 个类别。

调用方式：

```
from tensorflow.Keras.datasets import cifar100
(x_train, y_train), (x_test, y_test) = cifar100.load_data(label_mode='fine')
```

IMDB Movie Reviews

包括来自 IMDB 的 25000 条电影评论，以正面或负面作为情感标签，每条评论都已经按照单词频率索引进行了编码。

调用方式：

```
from tensorflow.Keras.datasets import imdb
(x_train, y_train), (x_test, y_test) = imdb.load_data(path="imdb.npz",
                                                      num_words=None,
                                                      skip_top=0,
                                                      maxlen=None,
                                                      seed=113,
                                                      start_char=1,
                                                      oov_char=2,
                                                      index_from=3)
```

Reuters Newswire Topics

路透社新闻标题分类，共有 1 万多条数据、46 个类别。

调用方式：

```
from tensorflow.Keras.datasets import reuters
(x_train, y_train), (x_test, y_test) = reuters.load_data(path="reuters.npz",
```

續表

```
                    num_words=None,
                    skip_top=0,
                    maxlen=None,
                    test_split=0.2,
                    seed=113,
                    start_char=1,
                    oov_char=2,
                    index_from=3)
```

MNIST

手写数字识别，共有 60000 张 28×28 的数字图片、10 个类别（0～9）及 10000 张测试图片。

调用方式：

```
from tensorflow.Keras.datasets import mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()
```

Fashion MNIST

和 MNIST 类似，也有 60000 张 28×28 的图片，但不再是数字，而是不同服装的灰度图，共有 10 个类别，并包括 10000 张测试图片。

调用方式：

```
from tensorflow.Keras.datasets import fashion_mnist
(x_train, y_train), (x_test, y_test) = fashion_mnist.load_data()
```

Boston Housing Price

数据集包括 20 世纪 70 年代后期波士顿地区不同地点的房屋的 13 个属性，以及该地点房屋的中值（Median Value）。

调用方式：

```
from tensorflow.Keras.datasets import boston_housing
(x_train, y_train), (x_test, y_test) = boston_housing.load_data()
```

3.4　再次程式碼實戰

3.4.1　XOR運算

透過前面幾節的介紹，我們應該已經瞭解如何用Keras來構建機器學習模型了。這裡再次以一個簡單的數學問題為例，從設計模型的角度來考慮如何用Keras實現。

問題：如何用機器學習實現對XOR運算的預測？

我們知道，XOR是一個基本的邏輯運算，如表3-7所示。

表3-7　XOR運算示例

X_1	X_2	结果(X_1 XOR X_2)
0	0	0
0	1	1
1	0	1
1	1	0

在二維座標系中如圖3-16所示。

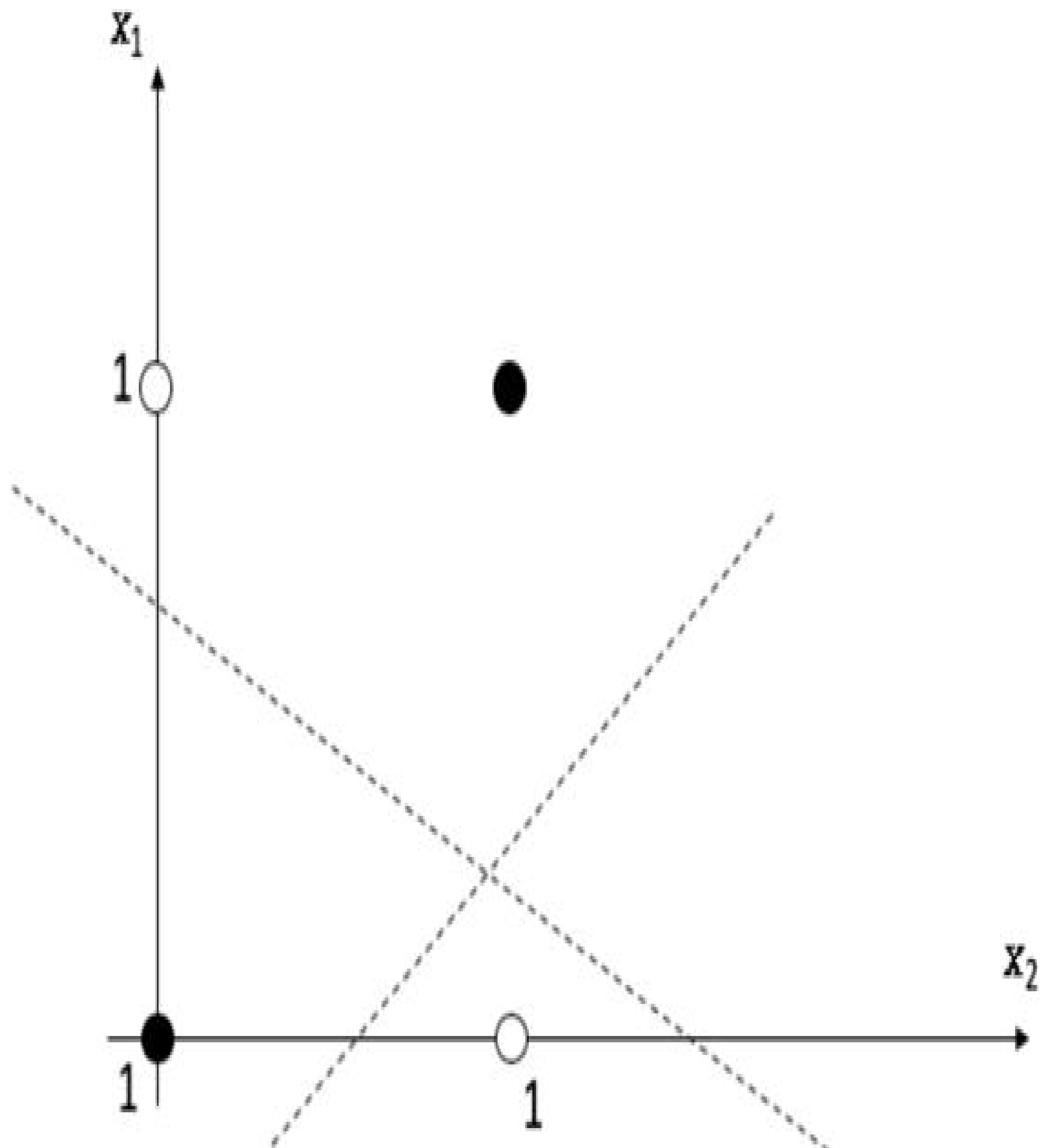


圖3-16　XOR輸入座標圖，黑點表示結果為0，白點表示結果為1

我們首先考慮建立對應的訓練集：

```
X = np.array([[0,0],[0,1],[1,0],[1,1]])
y = np.array([[0],[1],[1],[0]])
```

然後看到我們的輸入為兩個，輸出為一個結果，那麼我們直接建立這樣一個感知器：

```
model = Sequential()
model.add(Dense(1, input_dim=2))
model.add(Activation('sigmoid'))
model.compile(loss='mean_squared_error', optimizer='adam')
```

透過3.3節的學習，我們知道這是個很簡單的感知器網路，實際上並沒有隱藏層，只是一個sigmoid啟用函式的輸出，使用MSE作為損失函式，利用adam進行最佳化。我們來看看它的效果如何：

```
model.fit(X, y, batch_size=1, epochs=10000)
print(model.predict (X))
```

因為訓練資料只有4組，所以我們選擇batch_size為1，同時用了較多的epochs訓練次數。Model.predict_proba可以接收多個輸入並返回預測結果如下：

```
[[0.49851936]
 [0.4997031 ]
 [0.4999745 ]
 [0.50115824]]
```

我們的真實預測值實際上只有兩個，即0和1，並期望預測返回的結果儘量接近真實值。然而可以看到這次預測的結果幾乎是無效的，因為全部位於[0,1]的中間點附近，沒有起到任何預測作用。

實際上這就是線性不可分問題，因為圖3-16中(x_1,x_2)在座標上的兩組點（黑點和白點）並不能用一條直線劃分開，這是圖3-17(a)圖中沒

有隱藏層的感知器網路無法克服的。因此我們決定引入隱藏層，把模型的構建修改一下：

```
model = Sequential()
model.add(Dense(4, input_dim=2))
model.add(Activation('sigmoid'))
model.add(Dense(1))
model.add(Activation('sigmoid'))


model.compile(loss='mean_squared_error', optimizer='adam')
```

現在在模型中第1層不再是直接輸出，而是一個有4個神經元的隱藏層，然後進入輸出層，即如圖3-17(b)圖所示的結構。

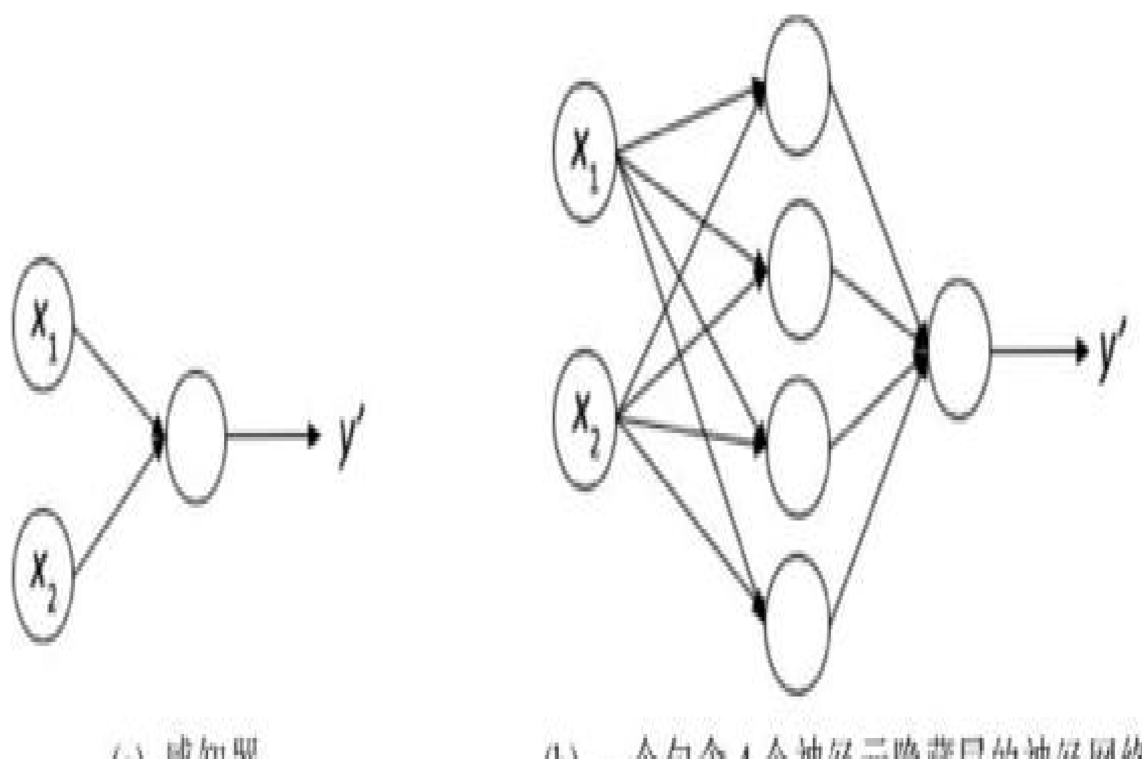


圖3-17　從感知器網路到神經網路

效果如何呢？我們不改其他引數，再跑一次看看：

```
[[0.0084908 ]
 [0.9729198 ]
 [0.97606194]
 [0.03987955]]
```

可以看到，效果有了很大的提升，對[0,0]和[1,1]的輸入，預測結果都相當接近0，而對[0,1]和[1,0]的輸入，預測結果都大於0.97，非常接近真實值1。如果我們再增加神經元的個數（例如增加到16或者32），則效果還有提升，但不再那麼明顯了。

3.4.2　房屋價格預測

前面的所有例子都是使用簡單的數字計算或者分類來體現機器學習的作用，一則方便讀者理解，二則通常透過不多的訓練迭代就可以達到較好的效果，方便讀者體驗程式碼。在本節中，我們將嘗試用真實的資料集來實驗。

在3.3節的最後提到，Keras提供了一些方便、實用的資料集，其中有一個美國波士頓地區在20世紀70年代末的房屋價格中位數的資料集，可以滿足我們用真實資料實驗的需要，這次我們將用它來預測當時該地區其他地點的房屋價格。

首先，我們引入需要的依賴庫和資料集：

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.datasets import boston_housing
```

然後，建立一個函式來構建模型（方便後續調整）：

```
def createModel():
    model = Sequential()
    model.add(Dense(32, input_shape=(13,), activation='relu'))
    model.add(Dense(1))
    model.compile(loss='mean_squared_error', optimizer='adam')
    return model
```

這裡維持了類似前一節的網路結構，只用一個隱藏層，神經元個數為32；然後引入訓練程式碼並利用測試資料完成模型評估。注意，這裡並不能用accuracy這樣的數值去評估，因為我們在做預測而不是分類，並不存在完全匹配的預測值，所以只能去檢視loss的輸出結果：

```
1  (x_train, y_train), (x_test, y_test) = boston_housing.load_data()
2  model = createModel()
3  model.fit(x_train, y_train, batch_size=8, epochs=10000)
4
5  print(model.metrics_names)
6  print(model.evaluate(x_test, y_test))
```

我們看看以上程式碼都做了什麼。

第1行：直接讀入了Keras的boston_housing資料集資料。注意，因為利用了Keras自帶的資料集，所以能方便地匯入訓練資料（x_train、y_train）和測試資料（x_test、y_test），但在實際工作中我們往往需要從.csv檔案或者某個目錄中自行處理原始輸入。

第2～3行：建立模型並訓練。

第5～6行：列印模型評價引數名稱，並利用測試集(x_test,y_test)進行評估，將結果輸出。

其執行結果如下：

```
27.440169689702053
```

因為測試集資料為102條，其MSE為27，所以結果還算可以讓人接受。如果增加層數呢？我們試著增加一個隱藏層：

```
def createModel():
    model = Sequential()
    model.add(Dense(32, input_shape=(13,), activation='relu'))

    model.add(Dense(16, activation='relu'))
    model.add(Dense(1))
    model.compile(loss='mean_squared_error', optimizer='adam')
    return model
```

這次新增了一層，再看看結果：

```
# Epoch 9999/10000
# 404/404 [==============================] - 0s 64us/sample - loss: 1.1444
# Epoch 10000/10000
# 404/404 [==============================] - 0s 66us/sample - loss: 0.9892
# ['loss']
# 102/102 [==============================] - 0s 176us/sample - loss: 20.1909
# 20.19088176652497
```

這次MSE減少到了20.19，效果不錯，我們來看看測試集中前10項的預測結果：

```
for i in range(10):
    y_pred = model.predict([[x_test[i]]])
    print("predict: {}, target: {}".format(y_pred[0][0], y_test[i]))
```

輸出如下：

```
predict: 11.072909355163574, target: 7.2
predict: 20.068275451660156, target: 18.8
predict: 20.760414123535156, target: 19.0
predict: 31.071605682373047, target: 27.0
predict: 21.493732452392578, target: 22.2
predict: 21.203548431396484, target: 24.5
predict: 26.775787353515625, target: 31.2
predict: 21.803157806396484, target: 22.9
predict: 20.411075592041016, target: 20.5
predict: 21.540142059326172, target: 23.2
```

可以看到，預測值儘管和實際值存在差異，但絕大部分的差距都比較小，對於只有13 個特徵來源的模型來說，已經足夠了。如果我們希望再提高準確率，則除了增加網路的複雜度，更重要的是增加資料的特徵和資料量。實際上，人們在工作中越來越發現增加訓練資料（包括特徵和資料總數）比增加網路複雜度（包括深度和廣度）更直接、更準確。

3.5 本章小結

在本章中，我們首先快速介紹了Keras的概念，然後透過3.2節的示例描述了Keras在深度學習中的基本使用流程。在3.3節中，我們針對Keras核心的Model、Layer、Loss和optimizer這4個概念，以圖表、程式碼、公式及概念描述等多種形式進行了介紹。在3.4節中，我們先透過一個普通感知器無法解決的預測問題，快速用Keras建立了對應的神經網路模型並獲得理想結果，隨後使用真實的資料集實現了一個房價預測模型，並透過結果證明瞭其有效程度。

Keras是一個複雜而靈活的機器學習框架，這裡並不期望讀者靠短短幾節就完全熟悉Keras的使用方法，但相信透過本章的實戰和概念講解，讀者對Keras的使用流程已經有了一個大致的理解。實際上，讀完本章後，讀者已經具備了用Keras實現大量的簡單資料分類和預測工作的技術基礎。

從第4章開始，我們將針對機器學習在不同領域的應用分別進行深入講解：第4、5章講解資料處理和推薦系統；第6章講解NLP；第7、8章講解影像分類及識別；第9章講解模型轉換和部署。讀者可以根據自身需要挑選感興趣的章節閱讀，也可以按序通讀。

3.6 本章參考文獻

[1] https://www.tensorflow.org/guide/Keras

[2] https://www.Keras.io

第4章　預測與分類：簡單的機器學習應用

在前面的章節中，我們開始接觸深度神經網路方面的知識，用Keras實現深度神經網路並解決一些簡單的問題。在這個過程中，我們也學習了神經網路的一些主要概念和基本原理，但對其中的一些細節並沒有進行討論，例如分類的評價方式、邏輯迴歸的具體原理、梯度下降的深入分析，等等。要對這些概念和原理進行討論，我們的學習範圍就不單是深度神經網路了，還涉及機器學習的基本概念。

這裡並不想完全採用數學推導形式來解釋機器學習的各種概念，而是採用原理簡述與程式碼描述相結合的形式。

本章的理論及概念較多，讀者不必死記硬背，可以將本章作為對機器學習理論的深入學習，暫時不理解部分概念也不會影響後續的學習。

4.1　機器學習框架之sklearn簡介

工欲善其事，必先利其器。在工程應用中，我們用Python手敲程式碼從頭實現一個演算法的可能性非常低，這樣做不僅耗時耗力，還不一定能夠寫出架構清晰、穩定性強的模型。在更多情況下，我們一般分析採集到的資料，根據資料的特徵選擇合適的演算法，在工具包中呼叫演算法並調整演算法的引數以獲取需要的資訊，從而實現演算法使用的方便程度、執行效率和計算效果之間的平衡。前面介紹過的Keras框架更適合面向複雜問題的深度神經網路開發，並不適合機器學習應用（例如Keras並不包括SVM、Decision Tree等機器學習中的常見演算法）。而scikit-learn（後簡稱sklearn）正是一個可以幫助我們高效實現演算法應用的工具包。

sklearn是一個Python第三方提供的非常強大的機器學習庫，包含了從資料預處理到訓練模型的各個方面。在實際應用中使用sklearn可以幫我們大大減少程式碼編寫時間及程式碼量，讓我們有更多的精力

去分析資料分佈、調整模型和修改模型設定中的各種超參（如batch size、epoch次數、learning rate等）。sklearn還提供了強大的開源資料[1]，我們可以直接下載和使用這些開源資料。

4.1.1　安裝sklearn

確認系統中已安裝Python（2.6或3.3以上版本）、Numpy（1.6.1以上版本）、Scipy（0.9以上版本）。注意，sklearn 0.20是支援Python 2.7和Python 3.4的最後一個版本，sklearn 0.21只支援Python 3.5以上版本。

如果已經安裝了Numpy和Scipy，那麼安裝sklearn的最簡單方法就是使用pip或者conda命令：

```
pip install -U scikit-learn
conda install scikit-learn
```

4.1.2　sklearn中的常用模組

sklearn中的常用模組有分類、迴歸、聚類、降維、模型選擇和預處理。

◎　分類：識別某個物件屬於哪個類別，常用的演算法有SVM（基於支援向量機的分類器）、Nearest Neighbors（最近鄰）和Random Forest（隨機森林），常見的應用有垃圾郵件識別和影像識別。

◎　迴歸：預測與物件相關聯的連續值屬性，常用的演算法有SVR（基於支援向量的迴歸演算法，Support Vector Regressor）、Ridge Regression（嶺迴歸）和Lasso，常見的應用有藥物反應和預測股價。

◎　聚類：將相似的物件自動分組，常用的演算法有k-means、Spectral Clustering和Mean-Shift，常見的應用有客戶細分和分組實驗結

果。

◎　降維：減少要考慮的隨機變數的數量，常用的演算法有PCA（主成分分析）、Feature Selection（特徵選擇）和Non-negative Matrix Factorization（非負矩陣分解），常見的應用有視覺化。

◎　模型選擇：比較、驗證，以及選擇引數和模型，目的是透過調整引數提高精度。常用的模組有Grid Search（網格搜尋）、Cross Validation（交叉驗證）和Metrics（度量）。

◎　預處理：特徵提取和歸一化，常用的模組有Preprocessing和Feature Extraction，常見的應用有將輸入的資料（如文字）轉換為機器學習演算法可用的資料。

4.1.3　對演算法和模型的選擇

sklearn實現了很多演算法，面對這麼多演算法，我們該如何選擇呢？其實，主要考慮需要解決的問題及資料量的大小即可。sklearn官方提供了機器學習演算法引導圖，將其翻譯後如圖4-1所示。

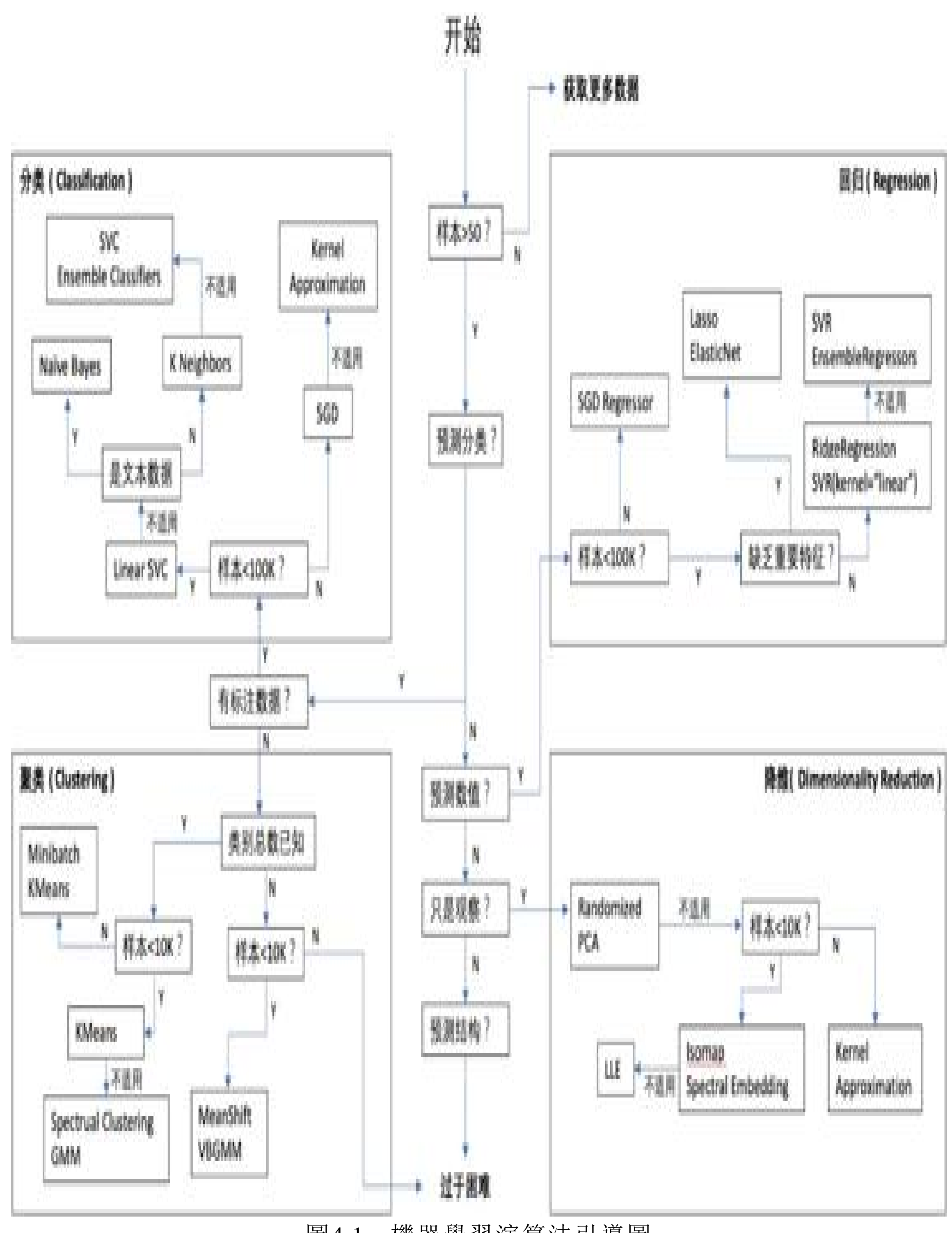


圖4-1　機器學習演算法引導圖

通常來說，我們可以按照圖4-1選擇一個比較合適的解決方法或者模型，但對模型的選擇並不是絕對的。事實上，在很多情況下，我們會實驗很多模型，才能比較出適合該問題的模型。

4.1.4　對資料集的劃分

有時對模型單獨進行一次實驗，其實驗結果會有一定的偶然性，並不能代表模型的平均效能。為此，我們可以使用交叉驗證或劃分資料集的其他方法對資料集進行多次劃分，以得出模型的平均效能。sklearn有很多劃分資料集的方法，在model_selection裡面可以找到，常用的方法如下。

（1）K折交叉驗證。

◎　KFold：普通K折交叉驗證。

◎　StratifiedKFold：保證每一類的比例相等。

（2）留一法。

◎　LeaveOneOut：留一。

◎　LeavePOut：留P驗證，當P為1時變成LeaveOneOut。

（3）隨機劃分法。

◎　ShuffleSplit：隨機打亂後劃分資料集。

◎　StratifiedShuffleSplit：隨機打亂後劃分資料集，每個劃分類的比例與樣本的原始比例一致；同時，StratifiedShuffleSplit和ShuffleSplit不同，它並不保證每組劃分出的資料是不同的。

以上只是對sklearn做了一些簡單介紹，在本章參考文獻[2]中可以找到更詳細的使用方法及引數介紹。

4.2　初識分類演算法

本節將正式結合實際案例，介紹機器學習的知識和程式碼細節。

分類演算法作為常見的機器學習演算法，應用非常廣泛，並且非常適合作為演算法入門選材。根據訓練資料是否擁有標記資訊，學習任務可大致分為監督學習和非監督學習兩類，分類演算法和迴歸演算法是前者的代表，聚類演算法是後者的代表。

詳細地說，監督學習其實就是對輸入的樣本經過模型訓練後有明確的輸出預期，而非監督學習是對輸入的樣本經過模型訓練後得到什麼輸出完全沒有預期。對於分類演算法，輸入的訓練資料有特徵（Feature）和標籤（Label），訓練的過程在本質上就是找到特徵和標籤間的關係。這樣，當有特徵而無標籤的未知資料被輸入時，我們就可以透過已有的關係得到其標籤。

本節主要以分類演算法中常見的幾種模型為切入點，詳細介紹我們在工作中可能會用到的分類模型，並結合具體的例子和程式碼，對機器學習尤其是分類演算法有更加直觀、深刻的理解。

4.2.1　分類演算法的效能度量指標

常見的分類演算法有樸素貝葉斯、決策樹、支援向量機、邏輯迴歸等，這些演算法也會因為自身的特點在不同大小的資料集上有不同的表現。本節主要介紹分類演算法中的一些常見度量指標如正確率、召回率、特意度、ROC曲線和AUC等。有了這些指標，我們就可以對每個演算法都做一個定量的結論，從而比較這些演算法在同一資料集上的不同表現。

1.混淆矩陣

混淆矩陣是資料科學、資料分析和機器學習中總結分類模型預測結果的情形分析表，以矩陣形式將資料集中的記錄按照真實的類別（Ground Truth）與分類結果進行彙總。以二元分類問題為例，資料集將例項分為正類（Positive）和負類（Negative）兩種類別，而對分類模型的預測可能做出陽性判斷（預測屬於正類）或陰性判斷（預測屬於負類）兩種判斷。混淆矩陣是一個2×2的情形分析表，根據實際結果和不同的預測結果總共有4種可能性。

◎　真陽性（True Positive，TP）：實際是正類，預測也是正類。

◎　假陽性（False Positive，FP）：實際是負類，預測是正類。

◎　真陰性（True Negative，TN）：實際是負類，預測也是負類。

◎　假陰性（False Negative，FN）：實際是正類，預測是負類。

表4-1給出了混淆矩陣的結果。

表4-1　混淆矩陣的結果

实际 / 预测		阳　性	阴　性
诊断结果	阳性	真阳性（TP）	假阳性（FP）
	阴性	假阴性（FN）	真阴性（TN）

圍繞混淆矩陣，分類演算法的主要指標如下。

（1）正確率（Precision）：衡量預測是正類，有多少預測結果是準確的。

$$\text{Precision} = \frac{TP}{TP + FP}$$

（2）靈敏度（Sensitivity）或召回率（Recall）：衡量所有正類有多少被準確檢索出來。

$$\text{Sensitivity} = \text{Recall} = \text{TPR} = \frac{TP}{TP + FN}$$

（3）特異度（Specificity）：衡量所有負類有多少被準確檢索出來。

$$\text{Specificity} = \text{TNR} = \frac{\text{TN}}{\text{FP} + \text{TN}}$$

在某些資料集上，Precision和Recall可能會有矛盾，即Precision、Recall呈現一高一低的情況。例如，某個班級有80個男生，20個女生，共計100個人，我們訓練一個模型來找出所有的女生。現在，某人挑選出50個人，其中有20個女生，有30個男生被誤判為女生，作為評估者的我們需要評估（Evaluation）這個人的挑選結果。當然，我們的評估結果是：正確率是40%，即20個女生/(20個女生+30個被誤判為女生的男生)；召回率是100%，即20個女生/(20個女生+ 0個被誤判為男生的女生)。

一般來說，想要覆蓋更多的樣本（Sample），該模型就有可能出錯。在這種情況下，模型會有很高的召回率，但是正確率較低（我們稱之為Overfit，即過擬合）。如果模型很保守，只對它很確定的取樣樣本做出預測，其正確率就會很高，但是召回率會相對低一些（我們稱之為Underfit，即欠擬合）。

為了克服這種問題，F1-Measure應運而生：

F1-Measure：

$$\frac{2}{F_1} = \frac{1}{P} + \frac{1}{R}, P = \text{Precision}, R = \text{Recall}$$

調整一下就是：

$$F_1 = \frac{2PR}{P + R} = \frac{2\text{TP}}{2\text{TP} + \text{FP} + \text{FN}}$$

所以在上一個例子中，挑選女生的F_1分數為

$$F_1 = \frac{2TP}{2TP + FP + FN} = \frac{2 \times 0.4 \times 1}{0.4 + 1} = 57.143\%$$

2.ROC曲線

ROC（Receiver Operating Characteristic，接收者操作特徵）曲線上的每個點都反映了對同一訊號刺激的感受性。簡單地說，ROC是以FPR（False Positive Rate）為橫軸，以True Positive Rate（TPR）為縱軸，透過不同的閾值點繪製而成的。

$$\text{FPR} = 1 - \text{Specificity} = \frac{\text{FP}}{\text{FP} + \text{TN}}$$

假設我們採用邏輯迴歸分類器，並計算出每個例項為正類的機率，設定一個閾值如0.6，使機率大於等於0.6的為正類，小於0.6的為負類，就可以對應地算出一組(FPR,TPR)，在平面中得到對應的座標點。隨著閾值的逐漸減小，越來越多的例項被劃分為正類，但是在這些正類中同樣摻雜著真正的負類，即TPR和FPR會同時增大，閾值最大時，對應的座標點為(0,0)；閾值最小時，對應的座標點為(1,1)。如圖4-2所示的實線為ROC曲線，線上的每個點都對應一個閾值。

◎　橫軸（FPR）：FPR= 1-TNR，FPR越大，預測結果為正類，但實際的負類越多。

◎　縱軸（TPR）：TPR=Sensitivity（正類覆蓋率），TPR越大，預測結果為正類並且實際的正類越多。

◎　理想目標：TPR=1，FPR=0，即圖中的(0,1)點，故ROC曲線越靠近(0,1)點、越偏離45°對角線、Sensitivity及Specificity越大，效果越好。

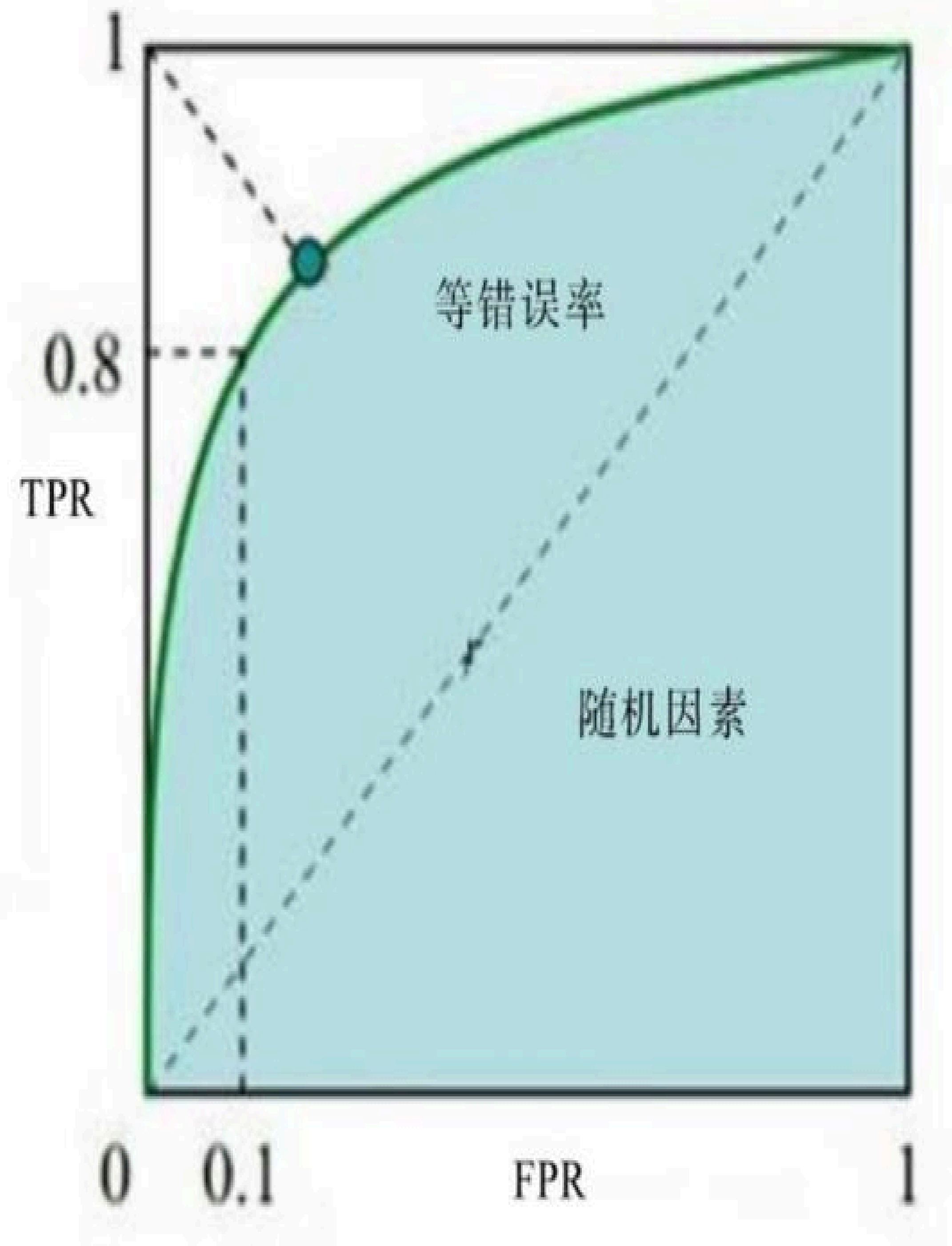


圖4-2　ROC曲線

3．AUC

AUC（Area Under Curve）被定義為ROC曲線下的面積，它的意義是什麼？假設我們有一個分類器，輸出的是樣本為正類的機率，則所有樣本都會有一個相應的機率，這樣就可以得到圖4-3，其中，橫軸表示預測為正類的機率，縱軸表示樣本數。所以，灰色區域表示所有負類的機率分佈，黑色區域表示所有正類的機率分佈。顯然，如果我們希望分類效果最好的話，那麼黑色區域越接近1越好，灰色區域越接近0越好。

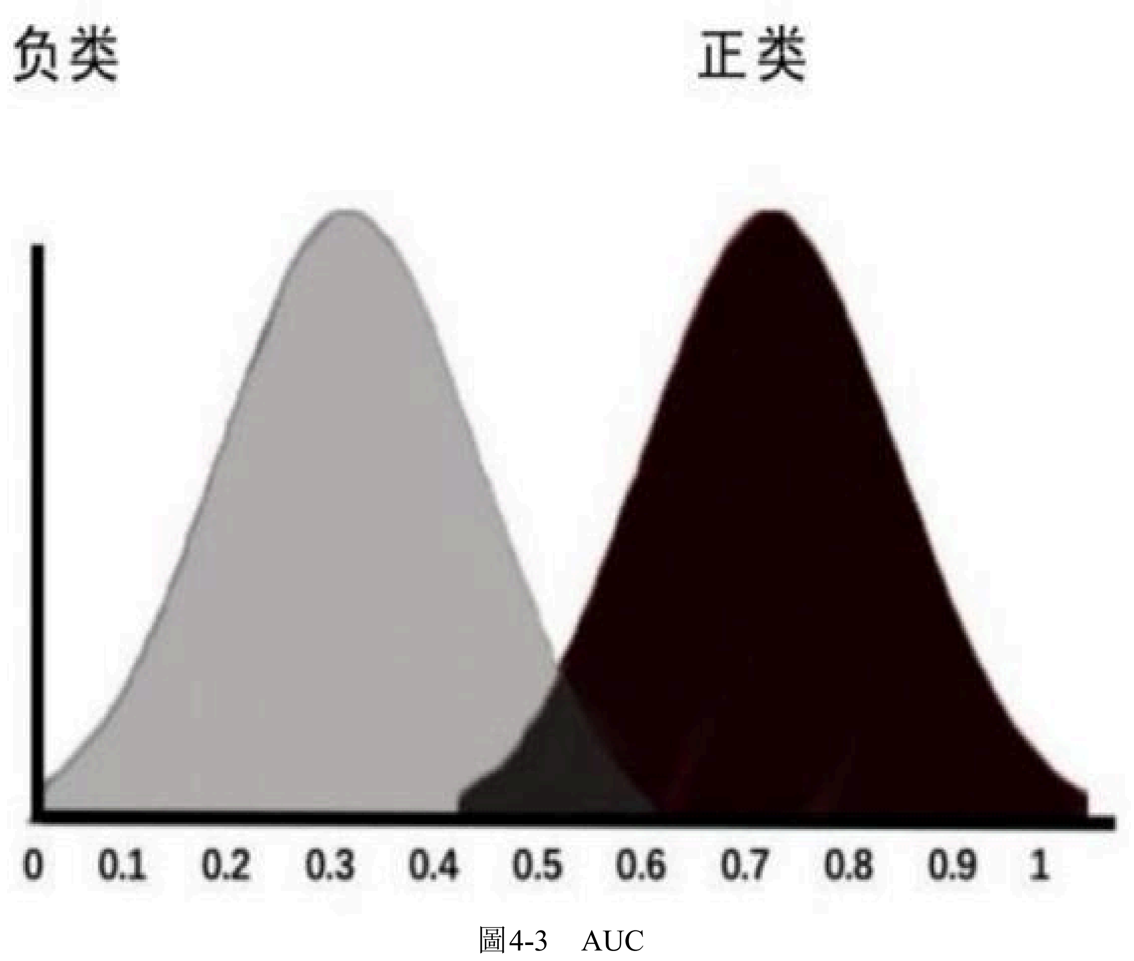


圖4-3　AUC

為了驗證分類器的效果，需要選擇一個閾值，使比這個閾值大的預測為正類，比這個閾值小的預測為負類，如圖4-4所示。

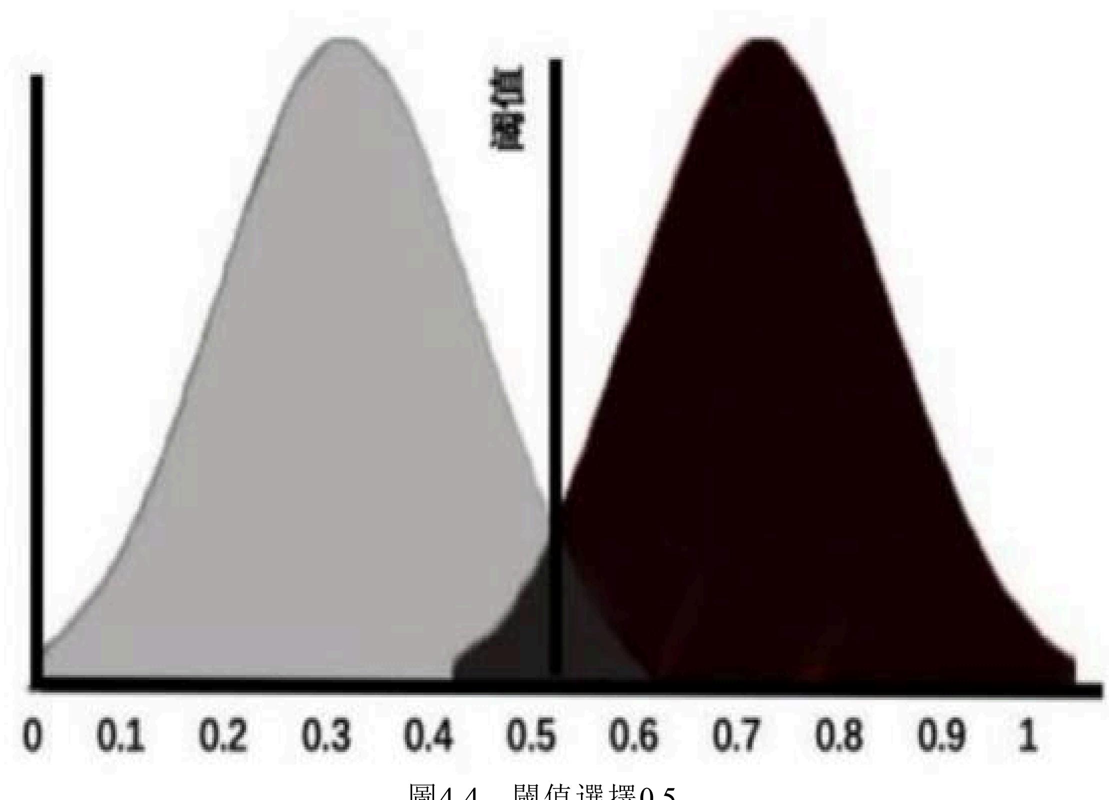


圖4-4　閾值選擇0.5

在圖4-4中閾值為0.5，於是左邊的樣本都被認為是負類，右邊的樣本都被認為是正類。可以看到，灰色區域與黑色區域是有重疊的，所以當閾值為0.5時，我們可以計算出準確率為90%。

現在引入ROC曲線。如圖4-5所示的左上角就是ROC曲線，其中的橫軸就是FPR，縱軸就是TPR。

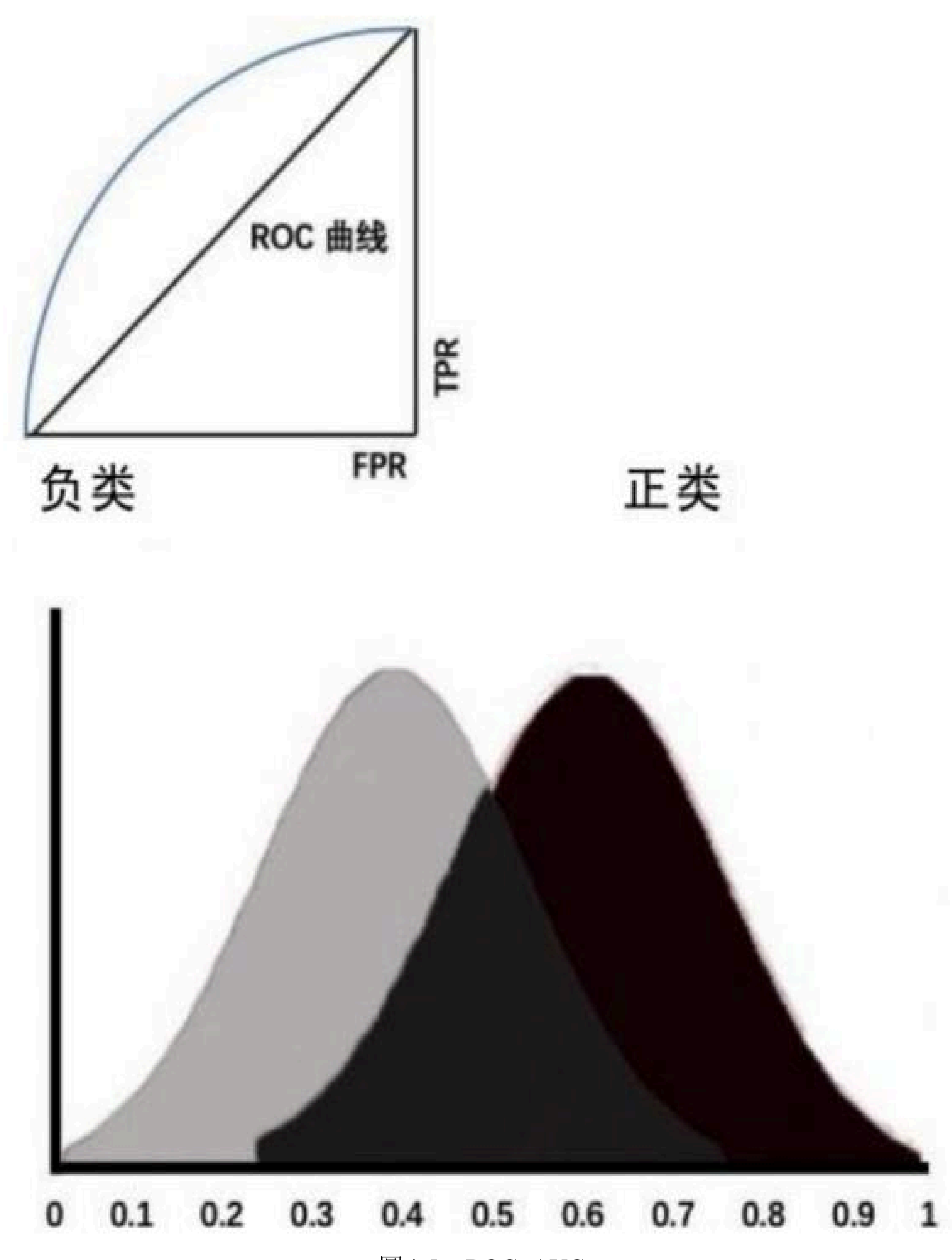


圖4-5　ROC+AUC

我們在AUC座標系中選擇不同的閾值，就可以對應ROC座標系中曲線上的一個點，如圖4-6所示。當閾值為0.8時，對應圖4-6左圖箭頭所指的點；當閾值為0.5時，對應圖4-6右圖箭頭所指的點。這樣，不

同的閾值就對應不同的點，最後，所有的點就可以連成一條曲線，就是ROC曲線。

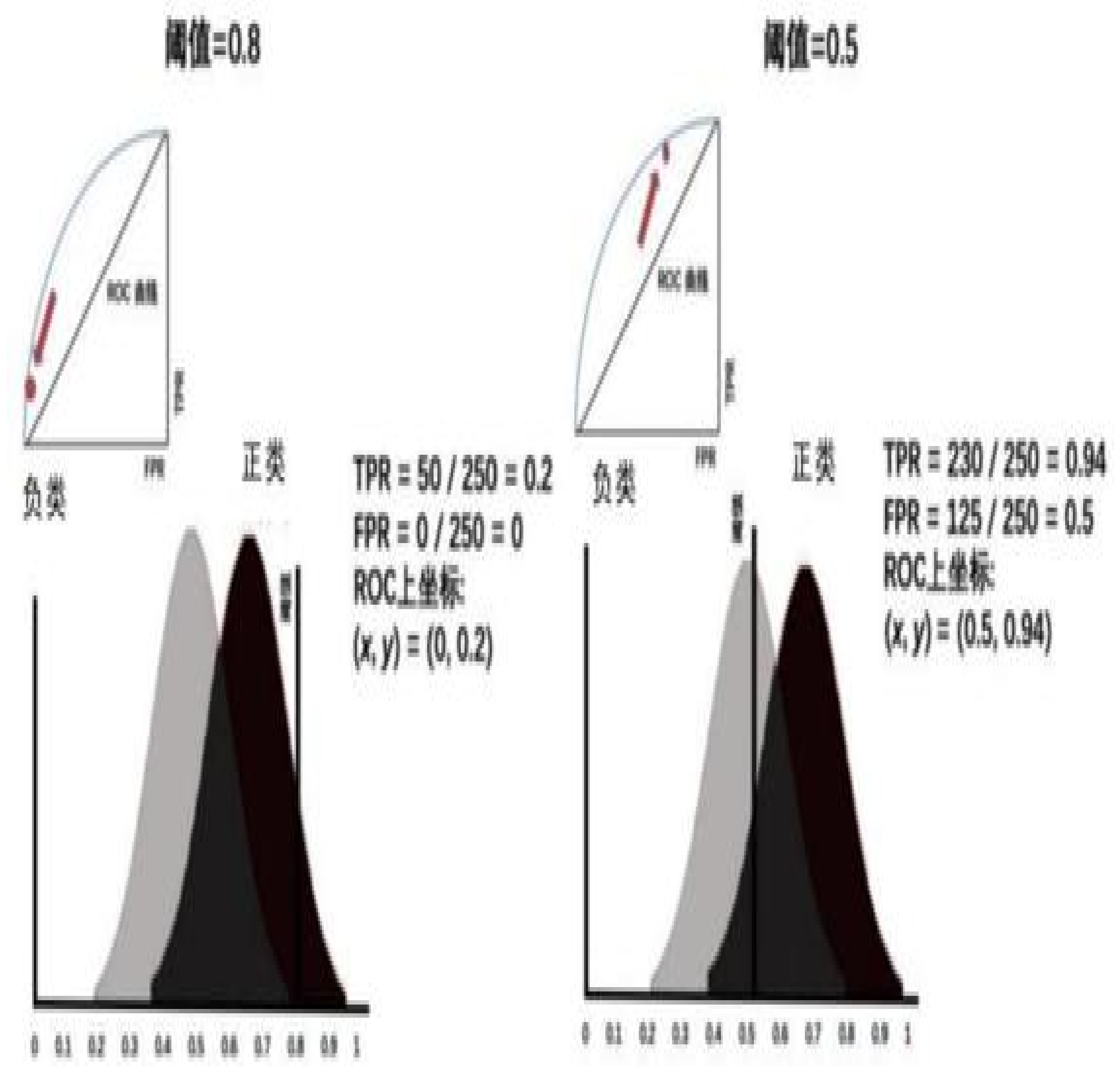


圖4-6　不同閾值的比較

現在我們來看看，如果灰色區域與黑色區域發生變化，那麼ROC曲線會怎麼變化呢？如圖4-7所示，在其左圖中，灰色區域與黑色區域重疊的部分不多，ROC曲線距離該圖左上角很近；在其右圖中，灰色區域與黑色區域基本重疊，ROC曲線就接近$y=x$這條線了。

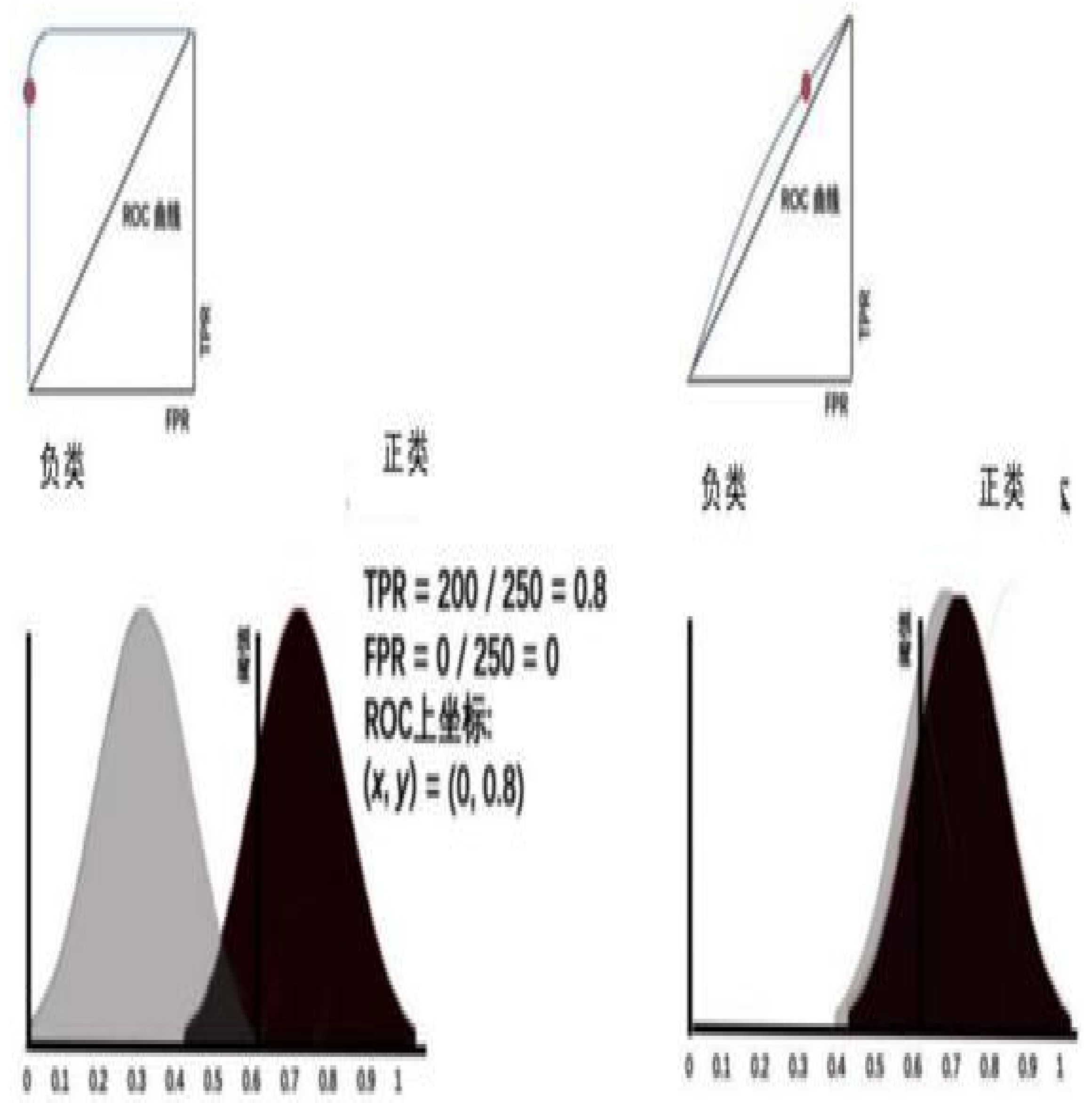


圖4-7　區域重疊後的變化

所以，如果我們想要用ROC曲線來評估分類器的分類質量，就可以透過計算AUC來評估了，這就是AUC的意義所在。其實，AUC表示的是正類排在負類前面的機率。

如圖4-8所示，第1個座標系的AUC值表示所有正類都排在負類前面；第2個AUC值表示有80%的正類排在負類前面；第3個AUC值表示有50%的機率正類排在負類前面。我們知道，閾值可以取不同的值，也就是說，分類的結果會受到閾值的影響，如果使用AUC，則因為考慮到了閾值變動的情況，所以評估效果更好。

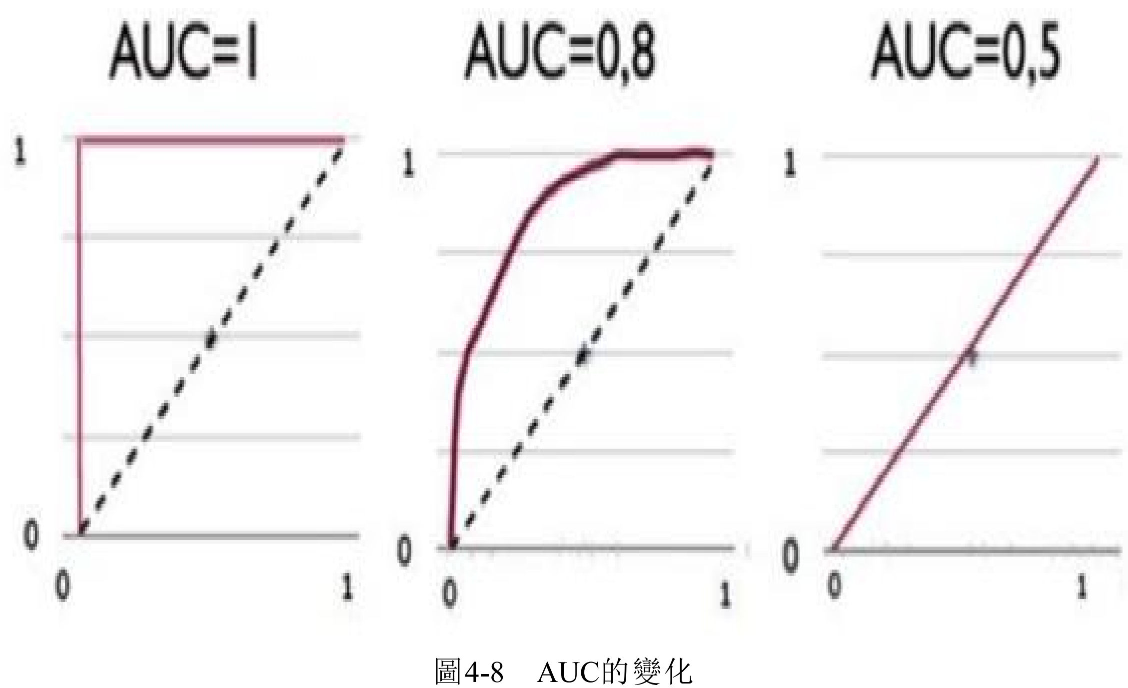


圖4-8　AUC的變化

4.2.2　樸素貝葉斯分類及案例實現

樸素貝葉斯分類是基於貝葉斯定理與特徵條件獨立假設的分類方法，源於古典數學理論，有穩定的數學基礎和分類效率，它是一種十分簡單的分類演算法。當然，簡單並不代表不好用。樸素貝葉斯的思想基礎是這樣的：對於給出的待分類項，求解在此待分類項出現的條件（特徵）下各個類別出現的機率，哪個類別出現的機率最大，就認為此待分類項屬於哪個類別。比如，你在一個屋子裡面看到一個渾身溼漉漉的人走進來，你大機率會猜測現在可能下雨了。當然，也有可能是這個人在外面正好被潑了水或者是因為灑水車經過而被淋溼了。但在沒有其他可用資訊的情況下，我們會選擇條件機率最大的類別，這就是樸素貝葉斯的思想基礎。

樸素貝葉斯分類演算法的實質就是計算條件機率的公式。在事件B發生的條件下，事件A發生的機率用$P(A|B)$來表示：

$$P(A|B) = \frac{P(AB)}{P(B)} = \frac{P(B|A)P(A)}{P(B)}$$

讓我們換個形式來表達：

$$P(\text{类别}|\text{特征}) = \frac{P(\text{特征}|\text{类别})P(\text{类别})}{P(\text{特征})}$$

而正式的樸素貝葉斯演算法的定義步驟如下。

（1）設 $\boldsymbol{X} = \{x_1, x_2, x_3, \cdots, x_n\}$ 為一個未知分類的集合，其中，x_n為集合中每一個訓練資料的一個特徵屬性。

（2）有已知的類別集合 $\boldsymbol{C} = \{y_1, y_2, \cdots, y_n\}$。

（3）計算 $P(y_1|x), P(y_1|x), \ldots, P(y_n|x)$，即不同訓練資料下標籤$y$的分佈機率。

（4）在預測未知標籤的資料時，我們選取機率最大的一個標籤作為這個訓練資料的標籤，即 $x \in y_k$，$P(y_k|x) = \max\{P(y_1|x), P(y_2|x), \cdots, P(y_n|x)\}$。

在上面第3步中各個 $P(y_n|x)$ 的條件機率可以透過下面的步驟得到。

（1）找到一個已知分類的待分類項集合，這個集合叫作訓練樣本集$\boldsymbol{S}$。

（2）透過統計得到各類別下各個特徵屬性的條件機率估計，即

$P(x_1|y_1),P(x_2|y_2),\cdots,P(x_n|y_1);\ P(x_1|y_2),P(x_2|y_2),\cdots,P(x_n|y_2);\cdots;\ P(x_1|y_1),P(x_2|y_2),$

$\cdots,P(x_n|y_m)$。

（3）如果各個特徵屬性是條件獨立的，則根據貝葉斯定理有如下推導：

$$P(y_i|x) = \frac{P(x|y_i)P(y_i)}{P(x)}$$

因為分母對於所有類別為常數，所以只需將分子最大化即可。又因為各特徵的屬性是條件獨立的，所以有：

$$P(x|y_i)P(y_i) = P(x_1|y_1),P(x_2|y_2)\cdots P(x_n|y_i)P(y_i) = P(y_i)\prod_{j=1}^{n}P(x_j|y_i)$$

透過上面這個過程，就可以計算每個x對應不同標籤的機率，然後將x歸屬到機率最大的那個標籤中即可。

下面透過一個具體的例子來更清晰地瞭解整個演算法。假設有以下一組訓練集，其中的天氣和溫度為特徵，而標籤為「是否出去玩」，如表4-2所示。

表4-2　一組訓練集例子

天 气	温 度	是否出去玩
晴天	热	No
晴天	热	No
阴天	热	Yes
雨天	适中	Yes
雨天	冷	Yes
雨天	冷	No
阴天	冷	Yes
晴天	适中	No
晴天	冷	Yes
雨天	适中	Yes
晴天	适中	Yes
阴天	适中	Yes
阴天	热	Yes
雨天	适中	No

這裡以計算當天氣是陰天、氣溫是適中的情況下，「是否出去玩」的值分別為 Yes、No 的機率為例：$P(\text{Play} = \text{Yes} \mid 天气 = 阴天, 气温 = 适中) = P(天气 = 阴天, 气温 = 适中 \mid \text{Play}$ $= \text{Yes})P(\text{Play} = \text{Yes})$。

按照上面的第3步，$P(\text{Play} = \text{Yes} \mid 天气 = 阴天, 气温 = 适中) = P(天气 = 阴天 \mid \text{Play} = \text{Yes})P(气温 = 适中$ $\mid \text{Play} = \text{Yes})P(\text{Play} = \text{Yes})$，因為 $P(\text{Yes}) = \frac{9}{14} = 0.64$，$\text{P}(阴天 \mid \text{Yes}) = \frac{4}{9} = 0.44$，$\text{P}(适中 \mid \text{Yes}) = \frac{4}{9} = 0.44$，所以 $P(\text{Play} = \text{Yes} \mid 天气 = 阴天, 气温 = 适中) = 0.44 \times 0.44 \times 0.64 = 0.124$。

同理，P(Play=No|天氣=陰天,氣溫=適中) =P(天氣=陰天, 氣溫=適中|Play =No)P(Play=No)=P(天氣=陰天, 氣溫=適中)P(Play=No) =P(天氣=陰天|Play=No) P(氣溫=適中|Play=No)P(Play=No)，所以 P(No) =5/14=0.36。

因為 P(天氣=陰天|Play=No) =0/9=0，P(氣溫= 適中|Play=No) =2/5=0.4，所以 P(Play=No|天氣=陰天, 氣溫=適中) =P(天氣=陰天, 氣溫=適中|Play =No)=0×0.4×0.36=0。。

案例實現如下：

```
from sklearn import preprocessing
import pandas as pd
import numpy as np
from sklearn.naive_bayes import GaussianNB
# 声明两列特征数据(weather、temp)和标签数据(play),共14组数据:
weather=['Sunny','Sunny','Overcast','Rainy','Rainy','Rainy','Overcast','Sunn
y','Sunny',
'Rainy','Sunny','Overcast','Overcast','Rainy']
temp=['Hot','Hot','Hot','Mild','Cool','Cool','Cool','Mild','Cool','Mild','Mi
ld','Mild','Hot','Mild']
play=['No','No','Yes','Yes','Yes','No','Yes','No','Yes','Yes','Yes','Yes','Y
es','No']
# 将字符串数据通过label encoding转成数字。如果特征天气对应的值可能有overcast、rainy、
sunny,则通过label encoding转换后分别对应0、1、2。scikit-learn里面的LabelEncoder库提
供了这种方法
le = preprocessing.LabelEncoder()
wheather_encoded=le.fit_transform(weather)
temp_encoded=le.fit_transform(temp)
label=le.fit_transform(play)
# 转换后的特征和标签分别为
# wheather_encoded: [2 2 0 1 1 1 0 2 2 1 2 0 0 1]
# temp_encoded: [1 1 1 2 0 0 0 2 0 2 2 2 1 2]
# label: [0 0 1 1 1 0 1 0 1 1 1 1 1 0]
# 通过pandas的concat方法将两列特征合并
```

```
df1 = pd.DataFrame(wheather_encoded, columns = ['wheather'])
df2 = pd.DataFrame(temp_encoded, columns = ['temp'])
result = pd.concat([df1, df2], axis=1, sort=False)
# 合并后的特征为[(2, 1), (2, 1), (0, 1), (1, 2), (1, 0), (1, 0), (0, 0), (2,
2), (2, 0), (1, 2), (2, 2), (0, 2), (0, 1), (1, 2)]
# 生成朴素贝叶斯分类模型，并将数据代入模型中进行训练
model = GaussianNB()
trainx = np.array(result)
model.fit(trainx, label)

# 用生成的模型预测天气为overcast、温度为mild时的结果
predicted= model.predict([[0,2]]) # 0:Overcast, 2:Mild
print("Predicted Value:", predicted)
```

可以看到預測的結果是1。

透過這個例子，我們可以看到樸素貝葉斯分類的整個演算法計算簡單，並且易於理解和實現，但只能被運用於小資料集，對大資料集則表現欠佳。同時，其演算法認為各特徵之間相互獨立、沒有影響，因此在處理相關性較大的特徵時表現不好。

4.3 決策樹

決策樹（Decision Tree）屬於機器學習有監督學習分類演算法，是根據資料的屬性採用樹狀結構建立的一種決策模型，表示物件屬性和物件值之間的一種對映。樹中的每一個節點都表示物件屬性的判斷

條件，其分支表示符合節點條件的物件，樹的葉子節點表示物件所屬的預測結果。

4.3.1 演算法介紹

決策樹常常用來解決分類和迴歸問題，常見的演算法包括CART（Classification And Regression Tree）、ID（3）、C4.5等。如圖4-9所示是一個簡單的決策樹，用於預測使用者某一天是否出去玩網球。是否出去玩網球主要依據三個屬性：天氣、溼度及是否有風。每一個非葉子節點都表示一個屬性條件判斷，表示使用者這一天是否會出去玩網球。例如：今天天氣是晴天，透過決策樹的根節點判斷，符合左邊分支（天氣為「晴天」）；再判斷溼度情況，今天溼度是60，符合左邊分支（溼度≤70，為『是』）；最後的結果落在「玩」的葉子節點上，所以預測使用者今天出去玩網球。

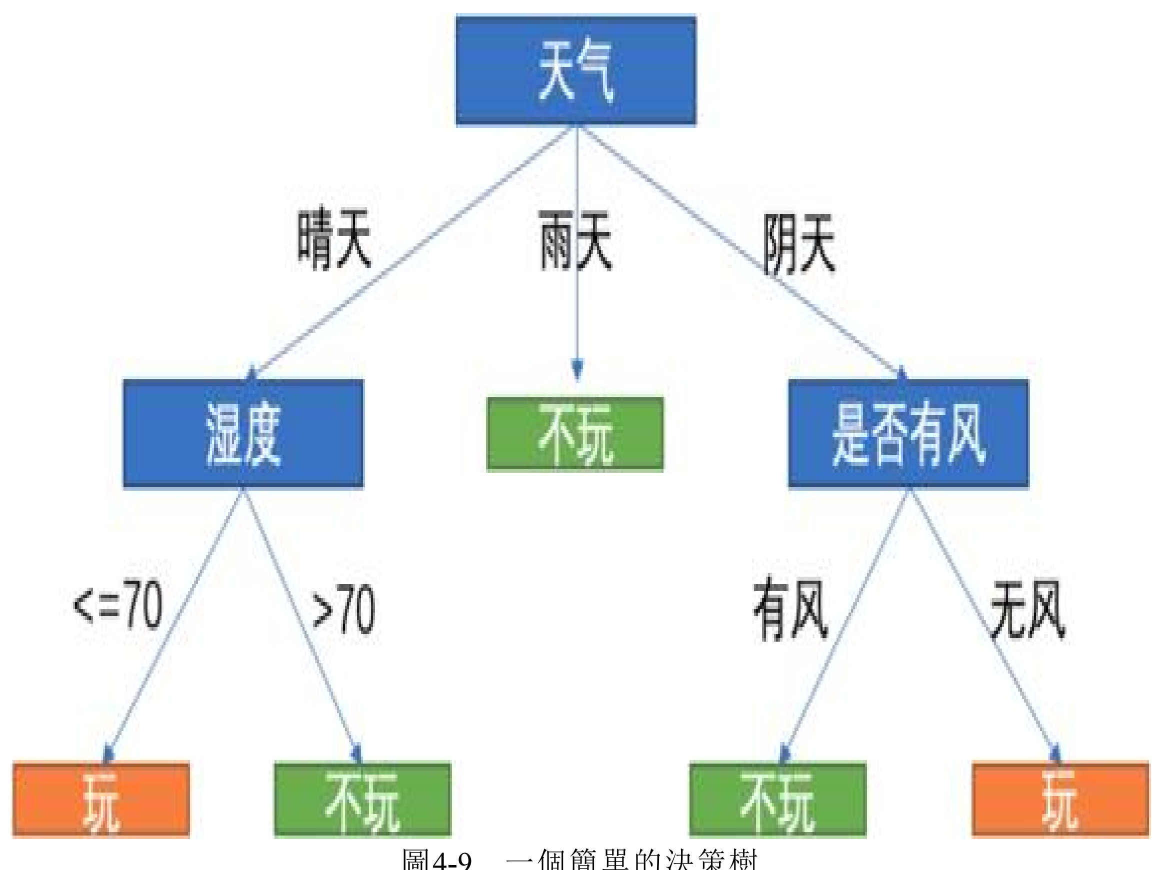


圖4-9　一個簡單的決策樹

4.3.2 決策樹的原理

決策樹是一個樹結構（可以是二叉樹或非二叉樹），每個非葉子節點都表示一個特徵屬性上的測試，每個分支都代表這個特徵屬性在某個值域上的輸出（比如在圖4-9中，溼度左邊的分支代表溼度這一特徵中值不大於70的所有資料，右邊的分支則是溼度大於70的所有資料）。而每個葉子節點都存放了一個類別。使用決策樹進行決策是從根節點開始的，會測試待分類項中相應的特徵屬性，並按照其值選擇輸出分支，直到到達葉子節點，最後將葉子節點存放的類別作為決策結果。決策樹演算法的核心思想是選擇一個合適的特徵作為判斷節點，可以快速地對資料集進行分類，減少決策樹的深度。在上面的例子中，天氣、溼度、是否有風是這個資料集的三個特徵。選擇特徵的目的是使分類後的資料集純度較高。純度其實是用來度量資訊中含有資訊量多少的。常用的三種基本的資訊度量方法有資訊增益、增益比率、基尼指數。在瞭解這三種基本的資訊度量方法之前，先介紹一些基本概念。

1.資訊量

資訊量是對資訊的度量，就跟時間的度量是秒一樣，資訊的多少是透過資訊量來衡量的，也與具體發生的事件有關。發生機率越小的事件發生後產生的資訊量越大，比如買彩票中獎了；發生機率越大的事件發生後產生的資訊量越小，比如在交通高峰期被堵在路上。因此，一個具體事件的資訊量應該隨著其發生機率的增加而遞減，且不能為負。

資訊量的公式如下：

$$h(x) = -\log_2 p(x)$$

$p(x)$為x發生的機率，資訊量的展現形式如圖4-10所示。

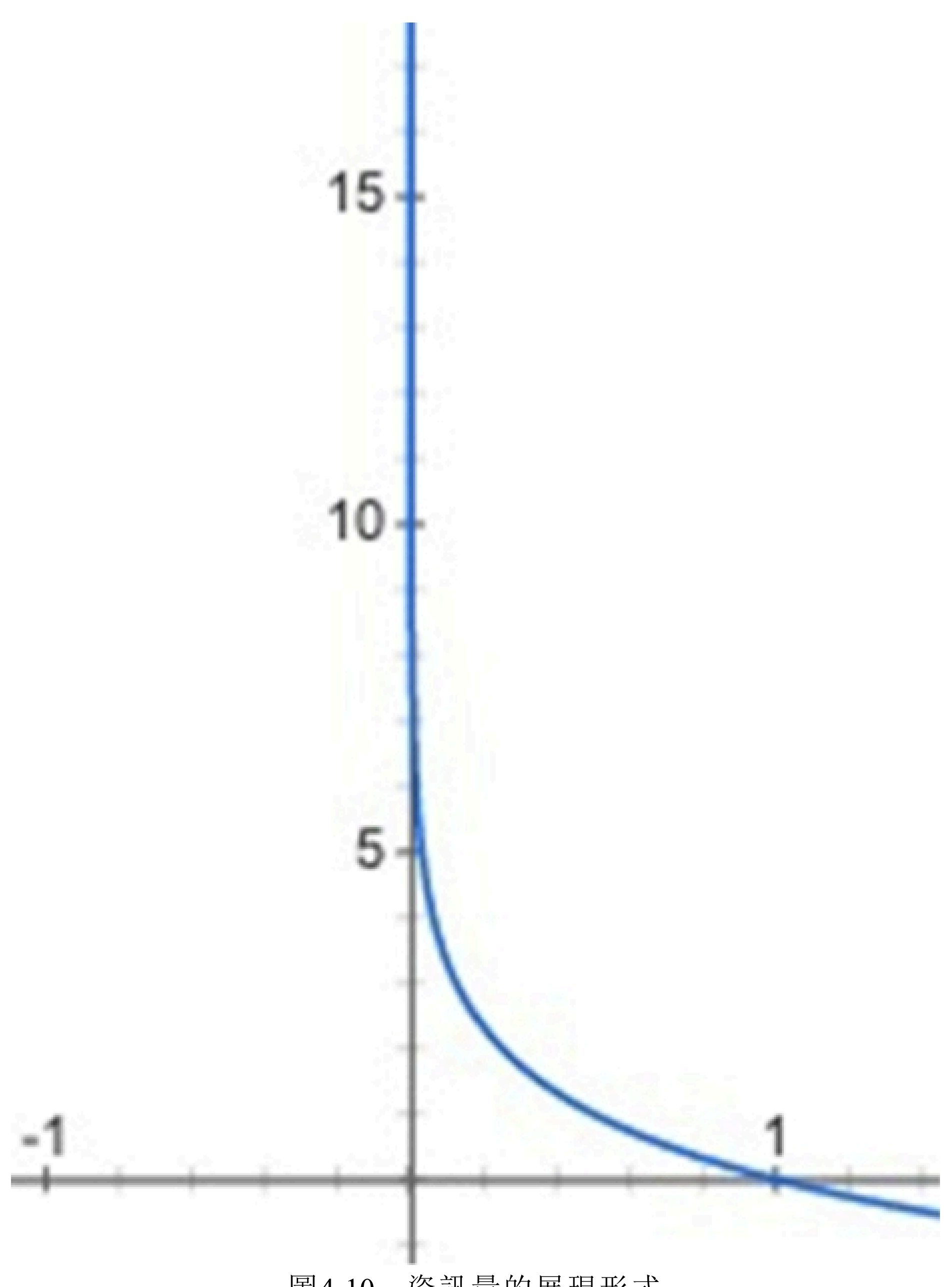


圖4-10　資訊量的展現形式

2.資訊熵

資訊量是一個具體事件發生所帶來的資訊，熵（Entropy）則是在結果出來之前對可能產生的資訊量的期望，它考慮到了該隨機變數所有的可能值，即所有可能發生的事件所帶來的資訊量的期望，公式如下：

$$\mathrm{Ent}(x) = -\mathrm{sum}(p(x)\log_2 p(x))$$

即

$$\mathrm{Ent}(X) = -\sum_{k=1}^{|K|} p_{xi} \log_2 p_{xi}$$

其中，X表示樣本的集合，$|K|$表示該樣本中的類別數量，p_{xi}表示第k種分類發生的機率。Ent(X)的值越小，X的純度越高。比如有A和B兩個同學，A同學的成績非常好，每次都考100分；B同學的成績比較一般，10次考試中會有5次不及格。那麼，A、B同學考試及格的資訊熵分別為

$$\mathrm{Ent}(A) = -1 \times 0 = 0$$

$$\mathrm{Ent}(B) = -\frac{1}{2} \times \log_2 \frac{1}{2} - \frac{1}{2} \times \log_2 \frac{1}{2} = 0.301$$

可以看出，Ent(x)越小，資訊的純度越高。

3.條件熵

設有隨機變數(X,Y)，其聯合機率分佈為

$$p(X = x_i, Y = y_i) \quad i = 1,2,\cdots,n; \quad j = 1,2,\cdots,m$$

條件熵H（Y|X）表示在已知隨機變數X的條件下隨機變數Y的不確定性，其推導公式為

$$H(Y|X) = \sum_{x \in X} p(x) H(Y|X = x)$$

$$= -\sum_{x \in X} p(x) \sum_{y \in Y} p(y|x) \log p(y|x)$$

$$= -\sum_{x \in X} \sum_{y \in Y} p(x, y) \log p(y|x)$$

4.資訊增益

資訊增益＝資訊熵━條件熵，即資訊增益代表某一條件下資訊複雜度減少的程度。換句話說，資訊增益就是在決策樹演算法中，使用某一個屬性a進行劃分後純度提高的程度。如果在選擇一個特徵後資訊增益最大（資訊不確定性減少的程度最大），我們就選擇這個特徵。下面透過如表4-3所示的例子來更好地理解資訊增益。

表4-3　理解資訊增益的例子

收　入	公积金	是否已婚	是否买房子
中	有	是	是
低	无	是	否
低	有	否	否
高	有	是	是
中	有	是	否
高	有	是	是

可以求得隨機變數X（是否買房子）的資訊熵為

$$\mathrm{Ent}(X) = -\frac{1}{2}\log\frac{1}{2} - \frac{1}{2}\log\frac{1}{2} = 0.301$$

假設我們選取收入作為下一個特徵，則收入的可能取值有低、中、高。在資料集中，低收入對應買房的個數為0，不買房的個數為2。中收入對應買房的個數為1，不買房的個數為1。高收入對應不買房的個數為0，買房的個數為2。可以得出條件熵為

$$H\left(Y\middle|X=\text{低}\right)=-\frac{2}{2}\log\frac{2}{2}=0$$

$$H\left(Y\middle|X=\text{中}\right)=-\frac{1}{2}\log\frac{1}{2}-\frac{1}{2}\log\frac{1}{2}=0.301$$

$$H\left(Y\middle|X=\text{高}\right)=-\frac{2}{2}\log\frac{2}{2}=0$$

$$H\left(\text{是否买房}\middle|\text{收入}\right)=\frac{2}{6}\times 0.301+\frac{2}{6}\times 0+\frac{2}{6}\times 0=0.1003$$

最終的資訊增益為

$$\text{Gain}\left(\text{买房},\text{收入}\right)=0.301-0.1003=0.2007$$

資訊增益的定義如下：

$$\text{Gain}(D,a) = \text{Ent}(D) - \sum_{v=1}^{|V|} \frac{|D^v|}{|D|} \text{Ent}(D^v)$$

簡單來說，資訊增益就是指劃分前後資訊熵的變化。

5.資訊增益率

在資訊增益中，Gain越大，劃分的效果越好，因為決策樹演算法在本質上就是找出每一列的最佳劃分及不同列劃分的先後順序。但資訊增益也有其侷限性，通常來說，資訊增益在面對類別較少的離散資料時效果較好，上例中的收入、公積金等資料都是離散資料，而且每個類別都有一定數量的樣本，在這種情況下使用資訊增益與增益率的區別並不大。但如果面對連續的資料（如體重、身高、年齡、距離等），或者每列資料都沒有明顯的類別之分（最極端的例子是該列所有的資料都獨一無二）的情況，資訊增益的效果會怎麼樣呢？我們知道資訊增益的公式為

$$\text{Gain}(D,a) = \text{Ent}(S) - \text{Ent}(A)$$

因為Ent(S)為初始label列的資訊熵，所以Gain(D,a)的大小取決於Ent(A)的大小，Ent(A)越小，Gain(D,a)越大。而資訊增益偏向選擇那些取值較多的特徵。主要原因是當特徵的取值較多時，根據此特徵劃分更容易得到純度更高的子集，因此資訊增益更大，在極端情況下，$\text{Ent}(A) = \sum_{i=1} \frac{1}{n} \log_2(1) = 0$，這樣Ent($A$)最小，Gain($D$,$a$)最大。但事實上，這樣劃分的效果較差。

為瞭解決該問題，這裡引入了資訊增益率（Gain-ratio），首先計算某個行為帶來的資訊：

$$\text{Info} = -\sum_{v \in \text{value}(A)} \frac{\text{num}(S_v)}{\text{num}(S)} \log_2 \frac{\text{num}(S_v)}{\text{num}(S)}$$

接著計算該行為下的資訊增益率：

$$\text{Gain_ratio} = \frac{\text{Gain}(D, a)}{\text{info}}$$

這樣就減小了劃分行為本身的影響。同樣以買房的例子為例，首先計算收入行為帶來的資訊：

$$\text{info}(\text{收入}) = -\sum_{v \in \text{value}(A)} \frac{\text{num}(S_v)}{\text{num}(S)} \log_2 \frac{\text{num}(S_v)}{\text{num}(S)} = -\frac{1}{2} \times \log_2 \frac{1}{2} - \frac{1}{2} \times \log_2 \frac{1}{2} - \frac{1}{2} \times \log_2 \frac{1}{2}$$

$$= 0.1505$$

接著計算買房帶來的資訊增益率：

$$\text{Gain_ratio} = \frac{0.2007}{0.1505} = 1.33$$

6.基尼值

基尼值Gini(D)反映了從資料集中隨機抽取兩個樣本，其類別標記不一致的機率。資料集的純度越高，每次抽到不同類別標記的機率越小。打個比方，在一個袋子裡裝100個乒乓球，其中有99個白球、1個黃球，則隨機從中抽取兩個球時，有很大機率抽到兩個白球。

所以，資料集D的純度可以用基尼值來度量，其定義如下：

$$\mathrm{Gini}(D) = \sum_{k=1}^{|y|} \sum_{k' \neq k} p_k p_{k'} = 1 - \sum_{k=1}^{|y|} p_k^2$$

7.基尼指數

基尼指數是針對屬性定義的，反映的是使用屬性a劃分後，所有分支中（使用基尼值度量的）純度的加權和。

屬性a的基尼指數定義如下：

$$\mathrm{Gini_index}(D, a) = \sum_{v=1}^{V} \frac{|D^v|}{|D|} \mathrm{Gini}(D^v)$$

我們在屬性集合A中選擇劃分屬性時，就選擇使得劃分後基尼指數最小的屬性作為最優劃分屬性。CART就是用基尼指數來選擇劃分屬性的。

4.3.3 例項演練

本節會使用加州大學爾灣分校（University of California at Irvine）提供的隱形眼鏡資料，資料的下載地址見本章參考文獻[3]。這組資料主要根據4個特徵（年齡、近視還是遠視、是否散光、是否經常流淚）將病人分為3類：不適合佩戴隱形眼鏡、適合佩戴軟隱形眼鏡、適合佩戴硬隱形眼鏡。

程式碼部分如下：

```
from collections import defaultdict, namedtuple
from math import log2
from sklearn import tree
import pydot

def split_dataset(dataset, classes, feat_idx):
    ''' 根据某个特征及特征值划分数据集
    :param dataset: 待划分的数据集，由数据向量组成的列表
    :param classes: 数据集对应的类型，与数据集有相同的长度
    :param feat_idx: 特征在特征向量中的索引
    :param splited_dict: 保存分割后数据的字典特征值：[子数据集，子类型列表]
    '''
    splited_dict = {}
    for data_vect, cls in zip(dataset, classes):
        feat_val = data_vect[feat_idx]
        sub_dataset, sub_classes = splited_dict.setdefault(feat_val, [[], []])
        sub_dataset.append(data_vect[: feat_idx] + data_vect[feat_idx + 1:])
        sub_classes.append(cls)
    return splited_dict

def get_majority(classes):
```

```
    ''' 返回类型中占比最多的类型
    '''
    cls_num = defaultdict(lambda: 0)
    for cls in classes:
        cls_num[cls] += 1
    return max(cls_num, key=cls_num.get)

def get_shanno_entropy(values):
    ''' 根据给定列表中的值计算其香农熵
    '''
    uniq_vals = set(values)
    val_nums = {key: values.count(key) for key in uniq_vals}
    probs = [v/len(values) for k, v in val_nums.items()]
    entropy = sum([-prob*log2(prob) for prob in probs])
    return entropy

def choose_best_split_feature(dataset, classes):
    ''' 根据信息增益确定划分数据的最好特征
    :param dataset: 待划分的数据集
    :param classes: 数据集对应的类型
    :return: 划分数据增益最大的属性索引
    '''
    base_entropy = get_shanno_entropy(classes)
    feat_num = len(dataset[0])
    entropy_gains = []
    for i in range(feat_num):
        splited_dict = split_dataset(dataset, classes, i)
        new_entropy = sum([
            len(sub_classes) / len(classes) * get_shanno_entropy(sub_classes)
            for _, (_, sub_classes) in splited_dict.items()
        ])
        entropy_gains.append(base_entropy - new_entropy)
    return entropy_gains.index(max(entropy_gains))

def create_tree(dataset, classes, feat_names):
    ''' 根据当前数据集递归创建决策树
    :param dataset: 数据集
```

```
        :param feat_names: 数据集中数据对应的特征名称
        :param classes: 数据集中数据相应的类型
        :param tree: 以字典形式返回决策树
        '''
        # 如果在数据集中只有一种类型，则停止树分裂
        if len(set(classes)) == 1:
            return classes[0]
        # 如果遍历完所有特征，则返回比例最多的类型
        if len(feat_names) == 0:
            return get_majority(classes)
        # 分裂创建新的子树
        tree = {}
        best_feat_idx = choose_best_split_feature(dataset, classes)
        feature = feat_names[best_feat_idx]
        tree[feature] = {}
        # 创建用于递归创建子树的子数据集
        sub_feat_names = feat_names[:]
        sub_feat_names.pop(best_feat_idx)
        splited_dict = split_dataset(dataset, classes, best_feat_idx)
        for feat_val, (sub_dataset, sub_classes) in splited_dict.items():
            tree[feature][feat_val] = create_tree(sub_dataset, sub_classes,
sub_feat_names)
        tree = tree
        feat_names = feat_names
        return tree

    def build_decisiontree_using_sklearn(X, Y):
        clf = tree.DecisionTreeClassifier()
        clf = clf.fit(X, Y)
        n_nodes = clf.tree_.node_count
        children_left = clf.tree_.children_left
        children_right = clf.tree_.children_right
        feature = clf.tree_.feature
        threshold = clf.tree_.threshold
        dot_data = tree.export_graphviz(clf, out_file=None)
        graph = pydot.graph_from_dot_data(dot_data)
        print(n_nodes)
```

```
    print(children_left)
    print(children_right)
    print(feature)
    print(threshold)
    graph[0].write_dot('iris_simple.dot')
    graph[0].write_png('iris_simple.png')
    return clf


if __name__ == '__main__':
    lense_labels = ['age', 'prescript', 'astigmatic', 'tearRate']
    X = []
    Y = []
    with open('data/decisiontree/lenses_num.txt', 'r', encoding='utf-8-sig')
as f:
        for line in f:
            comps = line.strip().split(',')
            X.append(comps[: -1])
            Y.append(comps[-1])
    dt_model = build_decisiontree_using_sklearn(X, Y)
```

以上程式碼實現了透過ID3演算法選擇最佳特徵的決策樹，並透過Graphviz將決策樹視覺化。但在實際業務中使用該方法生成的決策樹往往發生過擬合，也就是說，將該決策樹運用到訓練資料上可以得

到很小的錯誤率，運用到測試資料上卻得到非常大的錯誤率，其主要原因如下。

◎　在訓練資料中存在噪聲資料，決策樹的某些節點將噪聲資料作為分割標準，導致決策樹無法代表真實資料。

◎　建模樣本抽取錯誤，包括樣本數量太少、抽樣方法錯誤等。

◎　決策樹的生長沒有得到合理限制，導致每個葉子都只包含單純的事件資料。

所以，為了避免過擬合的發生，我們通常會採用決策樹剪枝和隨機森林這兩種最佳化方案，4.3.4節會詳細講解這兩種最佳化方案。

4.3.4　決策樹最佳化

1.決策樹剪枝

在分類模型建立的過程中很容易發生過擬合。對決策樹的過擬合可以透過剪枝（Pruning）進行一定的修復。剪枝分為預先剪枝和後剪枝兩種，如下所述。

（1）預先剪枝指在決策樹生長的過程中使用一定的條件進行限制，使得決策樹在過擬合前就停止生長。預先剪枝的判斷方法也有很多，比如資訊增益在小於一定的閾值時透過剪枝使決策樹停止生長。但如何確定一個合適的閾值也需要一定的依據，閾值太高會導致模型擬合不足，閾值太低又會導致模型過擬合。

（2）後剪枝指在決策樹生長完成之後，按照自底向上的方式修剪決策樹。後剪枝有兩種方式：一種方式是用新的葉子節點替換子樹，該節點的預測類由子樹資料集中的多數類決定；另一種方式是用子樹中最常用到的分支代替子樹。因為預先剪枝可能會因為過早終止決策樹的生長導致模型的擬合能力不足，所以後剪枝對於大多數資料集能夠有更好的效果。

2.隨機森林

隨機森林（Random Forest）顧名思義就是用隨機的方式建立一個由很多決策樹組成的森林。隨機森林的每一棵決策樹之間是沒有關聯

的。在得到隨機森林之後，當有一個新的輸入樣本進入時，就讓隨機森林中的每一棵決策樹分別進行判斷，看看這個樣本應該屬於哪一類（對於分類演算法），然後看看哪一類被選擇得最多，就預測這個樣本為哪一類。隨機森林既可以處理屬性為離散值的量如ID3演算法，也可以處理屬性為連續值的量如C4.5演算法，還可以用來進行無監督學習聚類和異常點檢測。

假設在原始樣本集中共有N個樣本，每個樣本都有M個特徵，則隨機森林的構建過程如下。

（1）從原始的N個樣本集中抽取n個訓練樣本（$n < N$）（在訓練集中，有些樣本可能被多次抽取，有些樣本可能一次都沒被抽取），這n個訓練樣本被作為一個子集訓練一個新的決策樹。

（2）當新的決策樹的每個節點都需要分裂時，則隨機從原始的M個特徵中抽取m個特徵（$m << M$），然後從這m個特徵中採用某種策略（比如說資訊增益）來選擇1個特徵作為該節點的分裂特徵。

（3）重複第2步，一直到新的決策樹不能再分裂為止。

（4）重複第1步到第3步k次（k通常取決於資料量的大小），這樣就構成了隨機森林。

相較於普通的決策樹，隨機森林有以下優點。

◎　由於兩個隨機性（隨機選取n個資料集和m個特徵集）的引入，使得隨機森林不容易過擬合。

◎　在當前的很多資料集上，由於兩個隨機性的引入，使得隨機森林具有很好的抗噪聲能力。

◎　能夠處理很高維度特徵的資料，並且不用進行特徵選擇，對資料集的適應能力強：既能處理離散型資料，也能處理連續型資料，資料集無須規範化。

◎　訓練速度快，可以得到變數的重要性排序。

◎　整個過程容易並行化。

◎　實現簡單。

4.4 線性迴歸

4.4.1 演算法介紹

1.線性模型的基本形式

線性模型形式簡單、易於建模。許多功能強大的非線性模型可線上性模型的基礎上透過引入層級結果或高維度對映得到。

給定由n個特徵描述的集合：$x = (x_1, x_2, \ldots, x_n)$，其中，$x_i$是$x$在第$i$個屬性上的取值，線性模型試圖學習到一個透過特徵的線性組合來預測的函式，即

$$f(x) = w_1x_1 + w_2x_2 + \cdots + w_nx_n + b$$

2.線性迴歸

給定資料集$D = \{(x_1, y_1), (x_2, y_2),\ldots, (x_n, y_n)\}$，其中，$x_i = (x_{i1},x_{i2},\ldots,x_{id})$，$y \in R$，则線性迴歸透過對訓練集中標籤數值的擬合，來儘可能預測測試集中的輸出值。

4.4.2 例項演練

本節從簡單的資料集入手，實現線性迴歸模型。在sklearn的datasets中提供了一些輕量級的訓練資料，我們可以使用這些資料進行分類或者回歸模型的練習。

這裡用到的資料是美國人口普查局收集的馬薩諸塞州波士頓住房價格的相關資訊（詳細介紹見第3章最後的Keras實戰案例）。資料讀取和線性迴歸模型的搭建如下：

```
from __future__ import print_function
from sklearn import datasets
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import ShuffleSplit

if __name__ == '__main__':
   loaded_data = datasets.load_boston()
   feature = loaded_data['feature_names']
   X = loaded_data.data
   y = loaded_data.target
   model = LinearRegression()
   best_model = model
   best_test_mse = 100
   cv = ShuffleSplit(n_splits=3, test_size=.1, random_state=0)
   for train, test in cv.split(X):
      model.fit(X[train], y[train])
      train_pred = model.predict(X[train])
      train_mse = mean_squared_error(y[train], train_pred)
      test_pred = model.predict(X[test])
      test_mse = mean_squared_error(y[test], test_pred)
      print('train mse:' + str(train_mse) + 'test mse:' + str(test_mse))
      if test_mse < best_test_mse:
         best_test_mse = test_mse
         best_model = model
   print('lr best mse score: ' + str(best_test_mse))
```

4.5 邏輯迴歸

邏輯迴歸（Logistic Regression）是一種廣義線性模型（Generalized Linear Model）。線性模型能對連續值的結果進行預測，而在現實生活中還存在常見的分類問題，比如判斷使用者是否會點選或者購買某個商品、判斷比賽的勝負、病人是否生了某種病等。邏輯迴歸是機器學習中的一種分類模型，其演算法簡單、高效，應用非常廣泛。

4.5.1 演算法介紹

我們在工作中可能會遇到這樣的二分類問題：預測一個使用者是否會點選特定的商品、判斷使用者的性別等。要解決這些問題，我們通常會用到一些已有的分類演算法，比如邏輯迴歸或者支援向量機。它們都屬於有監督的學習，因此在使用這些演算法之前，必須先收集已批註好的資料作為訓練集。

假設有一組訓練資料：

$$S = (x_1 y_1 + \ x_2 y_2 + \cdots + \ x_n y_n)$$

其中，x_i是一個m維的向量，$x_i=[x_1, x_2, \ldots, x_m]$，$y$在$\{0, 1\}$中取值。

邏輯迴歸與線性迴歸都是一種廣義線性模型。邏輯迴歸假設因變數y服從伯努利分佈，線性迴歸則假設因變數y服從高斯分佈。因此邏輯迴歸與線性迴歸有很多相同之處，若去除假設函式（Hypothesis Function）sigmoid，則邏輯迴歸就是線性迴歸。可以說，邏輯迴歸是以線性迴歸為理論基礎的，但是邏輯迴歸透過sigmoid啟用函式引入了非線性因素，因此可以輕鬆處理0/1分類問題。

1.假設函式（Hypothesis Function）

設計一個分類模型，首先要給它設定一個學習目標。考慮一個二分類問題，訓練資料是一堆(特徵,標籤)組合：(x_1,y_1), (x_2,y_2), …，

(x_n,y_n)，其中，x_i是特徵向量，y是標籤（y=1表示正類，y=0表示負類）。LR首先定義一個條件機率$p(y|x;w)$表示給定特徵x時標籤y的機率分佈，其中的w是LR的模型引數。有了這個條件機率，就可以在訓練資料上定義一個似然函式，然後透過最大似然來學習w，這是LR模型的基本原理。

接下來的問題是如何定義這個條件機率，這時sigmoid啟用函式就派上用場了。我們知道，對於大多數（或者說所有）線性分類器，響應值小於w，x大於w和x的內積，這代表了資料x屬於正類（y=1）的置信度（Confidence）。$<w, x>$越大，該資料屬於正類的可能性就越大；$<w, x>$越小，該資料屬於負類的可能性就越大。$<w, x>$在整個實數範圍內取值。

現在，我們需要用一個函式把$<w, x>$從實數空間對映到條件機率$p(y=1|x, w)$，並且希望$<w, x>$越大，$p(y=1|x, w)$越大；$<w, x>$越小，$p(y=1|x, w)$越小（等同於$p(y=0|x, w)$越大）。而sigmoid啟用函式恰好能實現這一功能：首先，它的值域是（0,1），滿足機率的要求；然後，它是一個單調上升函式。最終，$p(y=1|x, w)$=sigmoid ($<w, x>$)。sigmoid啟用函式的原型如下：

$$g(z) = \frac{1}{1 + e^{-z}}$$

sigmoid啟用函式的曲線如圖4-11所示。可以看到，sigmoid啟用函式是一個s形的曲線，它的取值為[0, 1]，在遠離0的地方，函式的值會很快接近0或者1。它的這個特性對於解決二分類問題十分重要。

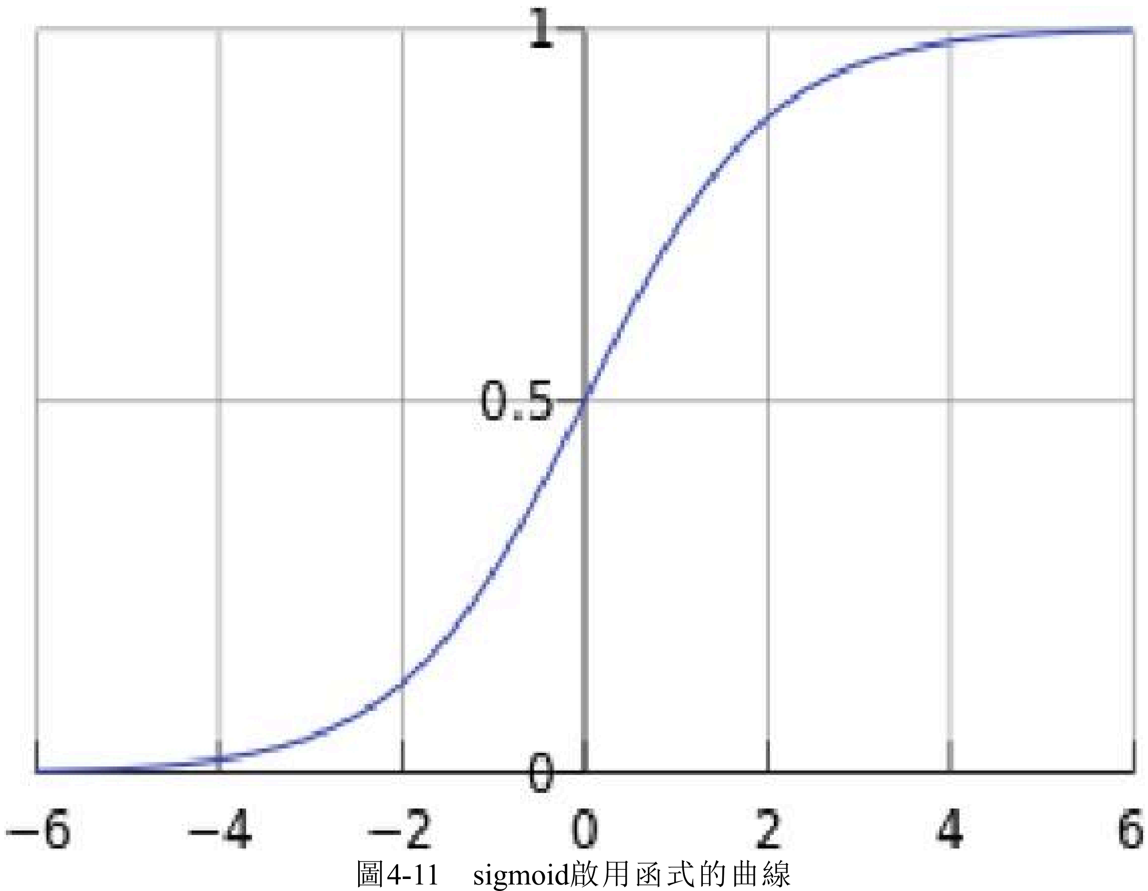


圖4-11　sigmoid啟用函式的曲線

2.決策函式

一個機器學習模型實際上是把決策函式限定在某組限定條件下，這組限定條件決定了模型的假設空間。當然，我們還希望這組限定條件簡單而合理。而邏輯迴歸模型所做的假設是

$$P(y=1|x;\theta)=g(\theta^T x)=\frac{1}{1+\mathrm{e}^{-\theta^T x}}$$

這裡的$g(h)$是前面提到的sigmoid啟用函式，相應的決策函式為

$$\hat{y}=1, \text{ if } P(y=1|x)>0.5$$

選擇0.5作為閾值是通常的做法，在實際應用時對於特定的情況可以選擇不同的閾值，如果對正類的判別準確性要求高，則可以選擇大一些的閾值；如果對正類的召回要求高，則可以選擇小一些的閾值。

3.引數求解

在模型的數學形式確定後，剩下的就是如何去求解模型中的引數。在統計學中常用的一種方法是最大似然估計，即找到一組引數，使資料的似然度（機率）更大。在邏輯迴歸模型中，似然度可表示為

$$L(\theta) = P(D|\theta) = \prod P(y|x;\theta) = \prod g(\theta^T x)^y (1 - g(\theta^T x))^{1-y}$$

取對數可以得到對數似然度：

$$l(\theta) = \sum y\log g(\theta^T x) + (1-y)\log(1 - g(\theta^T x))$$

另一方面，在機器學習領域，我們更經常遇到損失函式的概念，其衡量的是模型預測錯誤的程度。常用的損失函式有0-1損失、log損失、hinge損失等，其中，log損失在單個資料點上的定義為

$$L(y) = -y\log p(y|x) - (1-y)\log 1 - \mathrm{p}(y|x)$$

如果取整個資料集上的平均log損失，則可以得到

$$J(\theta) = -\frac{1}{N} l(\theta)$$

即在邏輯迴歸模型中，最大化似然函式和最小化log損失函式實際上是等價的。對於該最佳化問題存在多種求解方法，這裡以梯度下降為例進行說明。梯度下降又叫作最速梯度下降，是一種迭代求解的方法，透過在每一步選取使目標函式變化最快的一個方向調整引數的值來逼近最優值，基本步驟如下：

（1）選擇下降方向（梯度方向為$J(\theta)$，∇為損失函式對引數θ的求導）；

（2）選擇步長，更新引數 $\theta^i = \theta^{i-1} - \alpha^i \nabla J(\theta^{i-1})$；

（3）重複以上兩步直到滿足終止條件。

其效果如圖4-12所示。

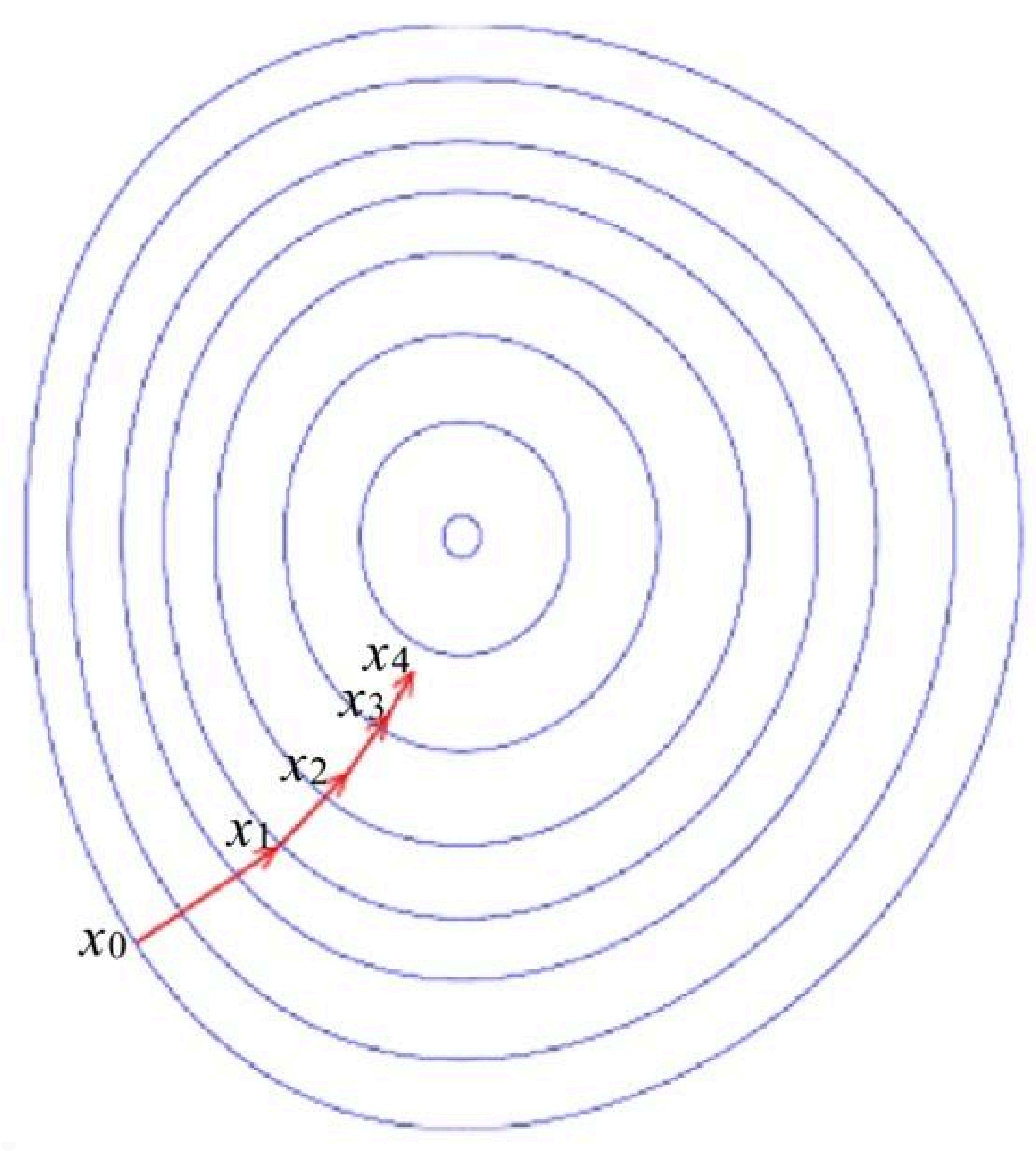


圖4-12　梯度下降

其中，損失函式的梯度計算方法為

$$\frac{\partial J}{\partial \theta} = -\frac{1}{n}\sum_{i}(y_i - y_i^*)x_i + \lambda\theta$$

沿梯度負方向選擇一個較小的步長可以保證損失函式是減小的，另一方面，邏輯迴歸的損失函式是凸函式（加入正則項後是嚴格凸函式），可以保證我們找到的區域性最優值同時是全域性最優值。此外，常用的凸最佳化方法都可以用於求解該問題，例如共軛梯度下降、牛頓法、LBFGS等。

4.分類邊界

在知道如何求解引數後，我們來看看模型得到的最終結果如何。我們可以從sigmoid啟用函式中很容易地看出，當$\theta^T x$ > 0時，y=1，否則y=0。$\theta^T x$ =0是模型隱含的分類平面（在高維空間中是超平面）。所以，邏輯迴歸在本質上是一個線性模型，但這並不意味著只有線性可分的資料能透過LR求解（對於二分類問題的資料集來說，如果存在一條直線，能夠把這兩個分類完美區分，那麼這個資料集就是線性可分的），實際上，我們可以透過特徵變換的方式把低維空間轉換到高維空間，而在低維空間線性不可分的資料在高維空間中線性可分的機率會大一些。如圖4-13所示為線性可分和線性不可分（透過特徵對映）的對比。

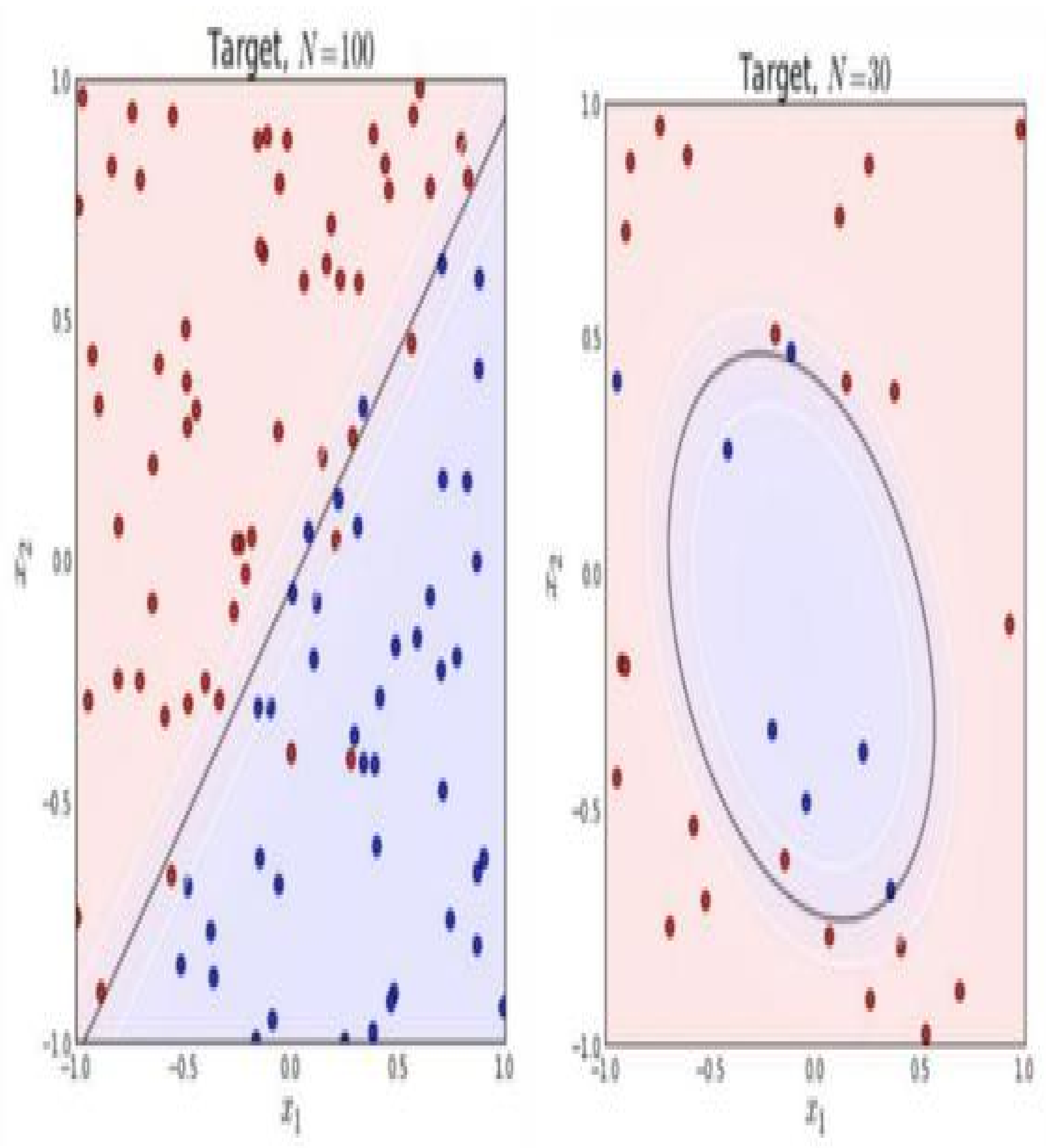


圖4-13　線性可分和線性不可分的對比

如圖4-13左圖所示是一個線性可分的資料集。如圖4-13右圖所示，原始空間中線性不可分，但是在特徵轉換 $[x_1, x_2] => [x_1, x_2, x_1^2, x_2^2, x_1 x_2]$ 後的空間是線性可分的，對應的原始空間中的分類邊界是一條類橢圓曲線。

4.5.2　多分類問題與例項演練

本節採用了經典的鳶尾屬植物資料集，該資料集是由英國統計和生物學家Ronald Fisher在1936年提出的。在這個資料集中包括三類不同的鳶尾屬植物：Iris Setosa、Iris Versicolour、Iris Virginica。

詳細程式碼如下：

```
import numpy as np
import pandas as pd
from sklearn import preprocessing
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import StandardScaler, PolynomialFeatures
from sklearn.pipeline import Pipeline
import matplotlib.pyplot as plt
import matplotlib as mpl
import matplotlib.patches as mpatches


if __name__ == "__main__":
    # 数据文件路径
    path = 'iris.data'
    data = pd.read_csv(path, header=None)
    data[4] = pd.Categorical(data[4]).codes

    x, y = np.split(data.values, (4,), axis=1)

    # 仅使用前两列特征
    x = x[:, :2]
    lr = Pipeline([('sc', StandardScaler()),
                   ('poly', PolynomialFeatures(degree=3)),
                   ('clf', LogisticRegression()) ])
    lr.fit(x, y.ravel())
    y_hat = lr.predict(x)
    y_hat_prob = lr.predict_proba(x)
    np.set_printoptions(suppress=True)
    print('y_hat = \n', y_hat)
    print('y_hat_prob = \n', y_hat_prob)
    print('准确度: %.2f%%' % (100*np.mean(y_hat == y.ravel())))
```

該實驗結果的準確度為80.67%。

4.6　神經網路

4.6.1　神經網路的歷史

自2012年ImageNet大賽技驚四座後，深度學習已經成為近年來機器學習和人工智慧領域中備受人們關注的技術。在深度學習出現之前，人們藉助SIFT、HOG等演算法提取具有良好區分性的特徵，再結合SVM等機器學習演算法進行影像識別。然而SIFT這類演算法提取的特徵是有侷限性的，導致當時比賽的最好結果的錯誤率也在26%以上。卷積神經網路的首次亮相就將錯誤率由26%降低到15%。同樣在2012年，在微軟團隊釋出的論文中顯示，透過深度學習可以將ImageNet 2012資料集的錯誤率降到4.94%。

在隨後幾年裡，深度學習在多個應用領域都取得了令人矚目的進展，例如語音識別、影像識別、自然語言處理等。鑑於深度學習的潛力，各大網際網路公司也紛紛投入資源進行研究與應用。人們意識到，在大資料時代，更加複雜且強大的深度模型能深刻揭示海量資料裡所承載的複雜而豐富的資訊，可對未來或未知的事件做更精準的預測。

筆者所在的公司也在深度學習方面進行了一些探索：在自然語言處理領域，我們將深度學習技術應用於文字分析、語義匹配、搜尋引擎的排序模型等；在計算機視覺領域，我們將深度學習技術應用於文字識別、影像分類、影像質量排序等。

1.神經網路的概念

神經網路是一種模擬人腦的神經網路以期實現類人工智慧的機器學習技術。人腦中的神經網路是一個非常複雜的組織，在成人的大腦中大約有超過1000億個神經元。一個神經元通常具有多個樹突，主要用來接收傳入的資訊；而軸突只有一條，在軸突尾端有許多軸突末梢可以給其他多個神經元傳遞資訊；軸突末梢跟其他神經元的樹突產生

連線，從而傳遞訊號，其連線位置在生物學上叫作「突觸」。人腦中的神經元形狀可以簡單地用圖4-14表示。

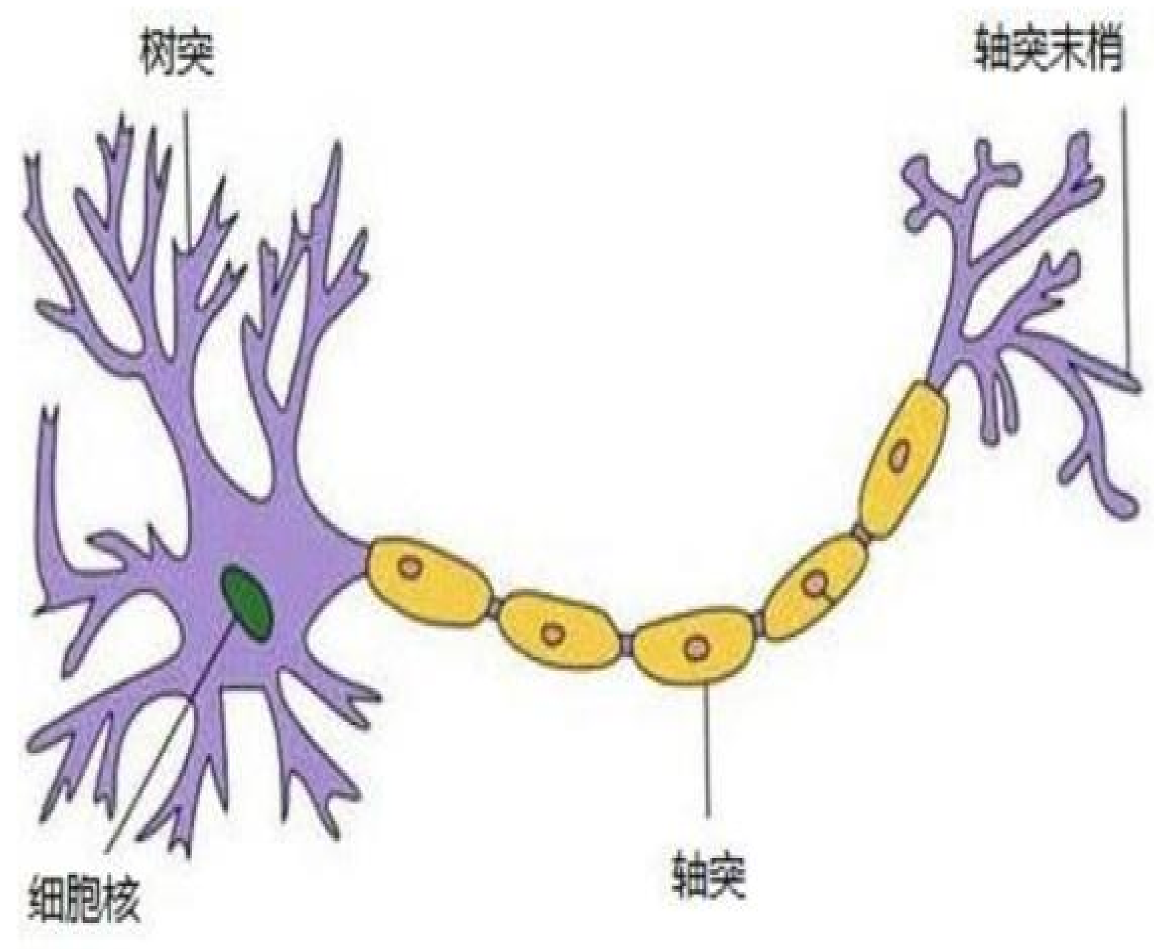


圖4-14　生物學中的神經元

機器學習中的神經網路結構也與人腦中的神經網路結構相符。如圖4-15所示是一個經典的三層神經網路結構，包含輸入層、隱藏層和輸出層。其中，輸入層有兩個單元，隱藏層有3個單元，輸出層有2個單元。每一個單元就是一個神經元。

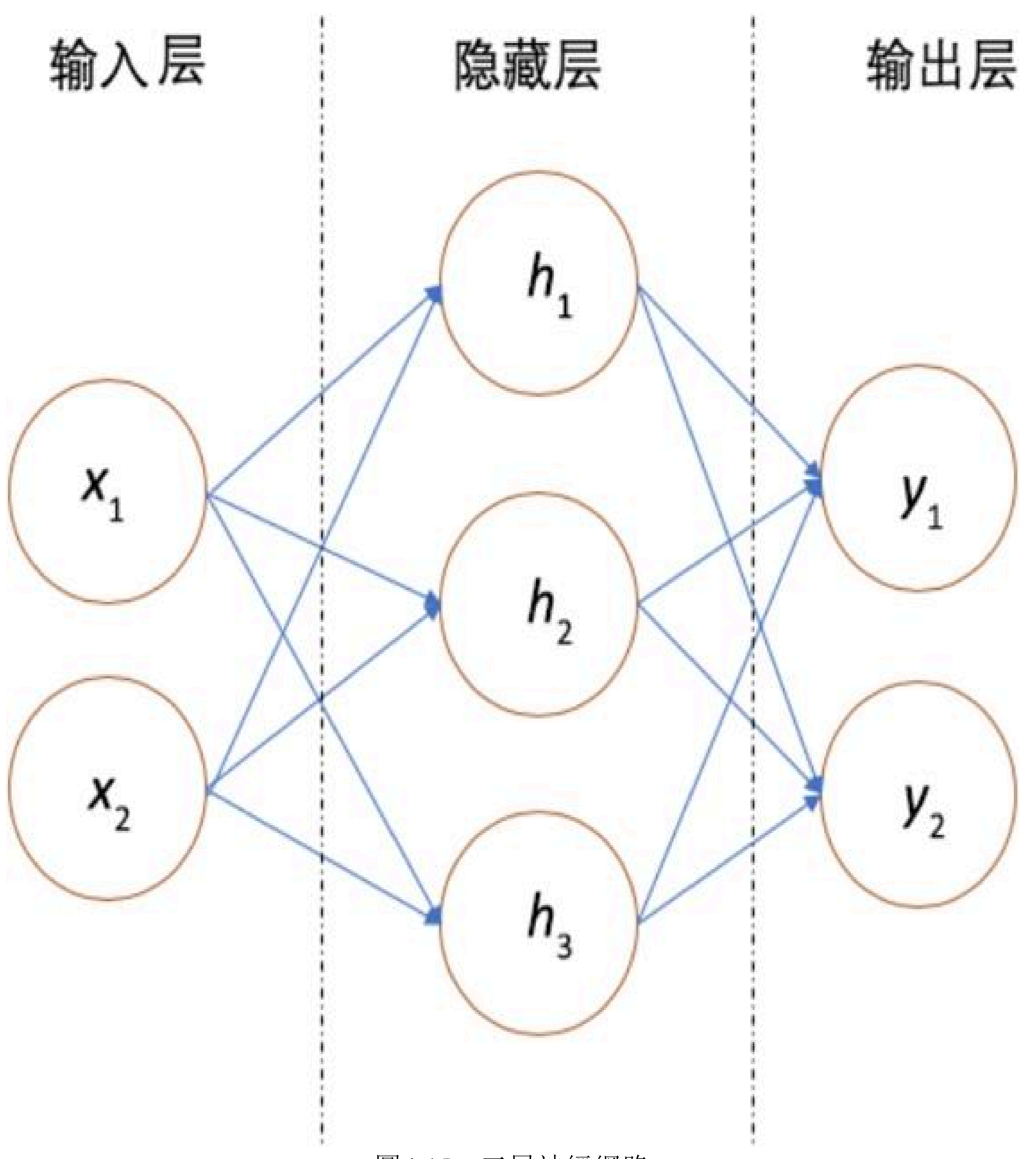


圖4-15　三層神經網路

如圖4-15所示的每個圓圈都是一個神經元，每條線都表示神經元之間的連線。可以看到，上面的神經元被分成了多個層，層與層之間的神經元都有連線，而層內之間的神經元沒有連線。最左邊的層叫作輸入層，負責接收輸入資料；最右邊的層叫作輸出層，我們可以從該層獲取神經網路的輸出資料；輸入層和輸出層之間的層叫作隱藏層。

隱藏層比較多（大於2）的神經網路叫作深度神經網路，而深度學習就是使用深層架構（比如深度神經網路）的機器學習方法。

除了以上特點，神經網路還有其他特點：

◎　同一層的神經元之間沒有連線；

◎　第N層的每個神經元和第N-1層的所有神經元相連（這就是連線的含義），第N-1層神經元的輸出就是第N層神經元的輸入；

◎　每個連線都有一個權值（見圖4-16）。

上面的規則定義了全連線神經網路的結構。事實上還存在很多其他結構的神經網路，比如卷積神經網路、迴圈神經網路（Recurrent Neural Network，RNN），它們都具有不同的連線規則。

那麼，深層網路與淺層網路相比有什麼優勢呢？簡單地說，深層網路的表達力更強。事實上，僅有一個隱藏層的神經網路就能擬合任何一個函式，但是它需要很多神經元，而深層網路用少得多的神經元就能擬合同樣的函式。

但是，簡單地增加神經網路的層數在很多場合下並不能解決問題，其原因主要有以下3個。

（1）在面對大資料或者複雜資料時（例如圖片、語音等），傳統的神經網路需要大量的輸入特徵。比如對於一張1024×768的灰度圖片，第1層就要處理786 432個特徵，會大量提取無用的特徵，並浪費很多計算資源。

（2）想要更精確的近似複雜的函式，就必須增加隱藏層的層數，這就導致了梯度擴散問題和過擬合問題。

（3）多層神經網路不包含時間引數，無法處理時間序列資料（比如音訊）。隨著人工智慧需求的提升，我們想要做複雜的影像識別、自然語言處理、語義分析翻譯等，使用多層神經網路顯然力不從心。

為瞭解決這些問題，人們又在多層神經網路的基礎上創造了深度學習模型。深度學習除了強調了模型結構的深度，還引入了新的結構，明確突出了特徵學習的重要性。透過逐層特徵變換，將樣本在原空間的特徵表示變換到一個新的特徵空間，使分類或預測更加容易。與人工規則構造特徵的方法相比，利用大資料來學習特徵，更能夠刻畫資料的豐富內在資訊。

深度學習克服了之前多層神經網路的缺點，如下所述。

（1）深度學習自動選擇原始資料的特徵。舉一個影像的例子，將畫素值矩陣輸入深度網路（這裡指常用於影像識別的卷積神經網路），網路的第1層表徵物體的位置、邊緣、亮度等初級視覺資訊；第2層會將第1層的邊緣特徵整合成物體的輪廓特徵；之後的層會表徵更加抽象的資訊，如貓或狗這樣的抽象資訊。所有特徵完全在網路中自動呈現，並非出自人工設計。

（2）在深度網路的學習演算法中，一種是改變網路的組織結構，比如用卷積神經網路代替全連線（Full Connected）網路，訓練演算法仍依據反向傳播梯度的基本原理；另一種則是徹底改變訓練演算法，例如Hessian Free Optimization、Recursive Least Squares（RLS）等。

（3）使用帶反饋和時間引數的迴圈神經網路（Recurrent Neural Network，RNN）處理時間序列資料。從某種意義上講，迴圈神經網路可以在時間維度上展開成深度網路，有效處理音訊資訊（語音識別和自然語言處理等），或者用來模擬動力系統。

那麼，為了理解神經網路，我們應該先理解神經網路的組成單元，即神經元。神經元也叫作感知器，感知器演算法在20世紀50～70年代很流行，也成功解決了很多問題。

2.神經元的定義

神經元模型是一個包含輸入、輸出與計算功能的模型。我們可以將輸入類比為神經元的樹突，將輸出類比為神經元的軸突，將計算類比為細胞核。如圖4-16所示是一個典型的神經元模型，包含兩個輸入、1個輸出和1個計算功能。

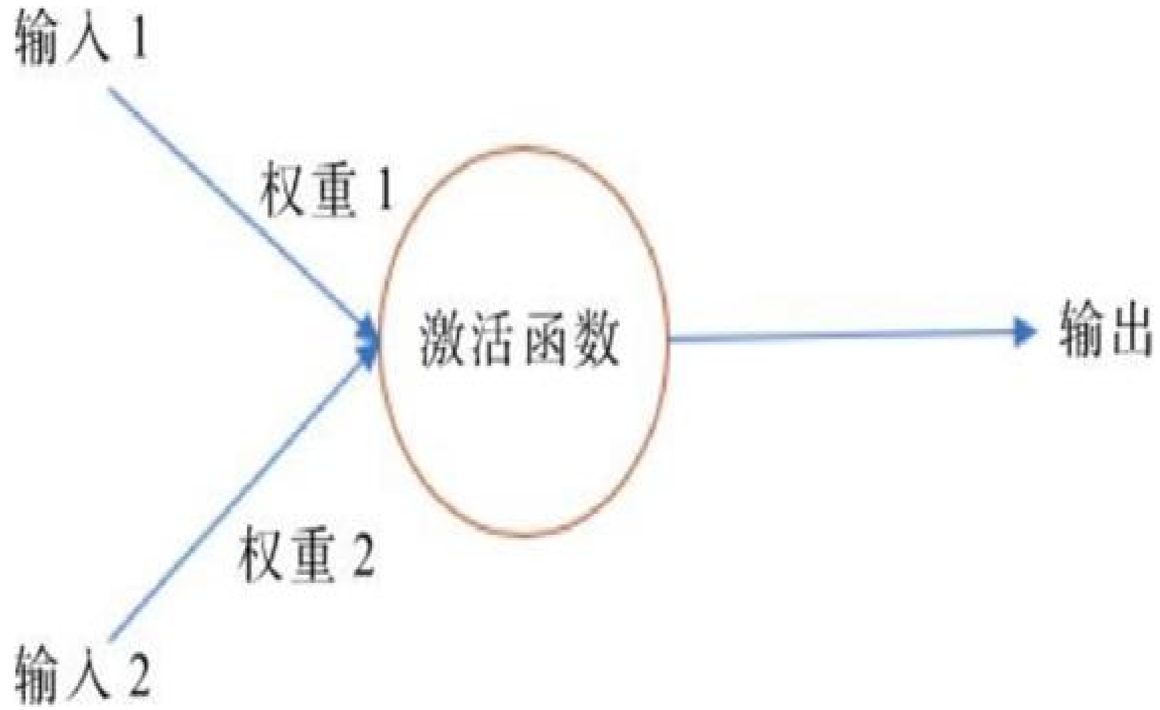


圖4-16　神經元

注意圖4-16中間帶箭頭的線，這些線被稱為「連線」，在每個連線上都有一個權值。一個神經網路的訓練演算法就是透過調整權重的值，使整個網路的預測效果最好。

啟用函式是用來加入非線性因素的，因為線性模型的表達能力不夠。神經網路中常用的啟用函式有relu、sigmoid、tanh等。

下面介紹神經網路的計算流程，如圖4-17所示。

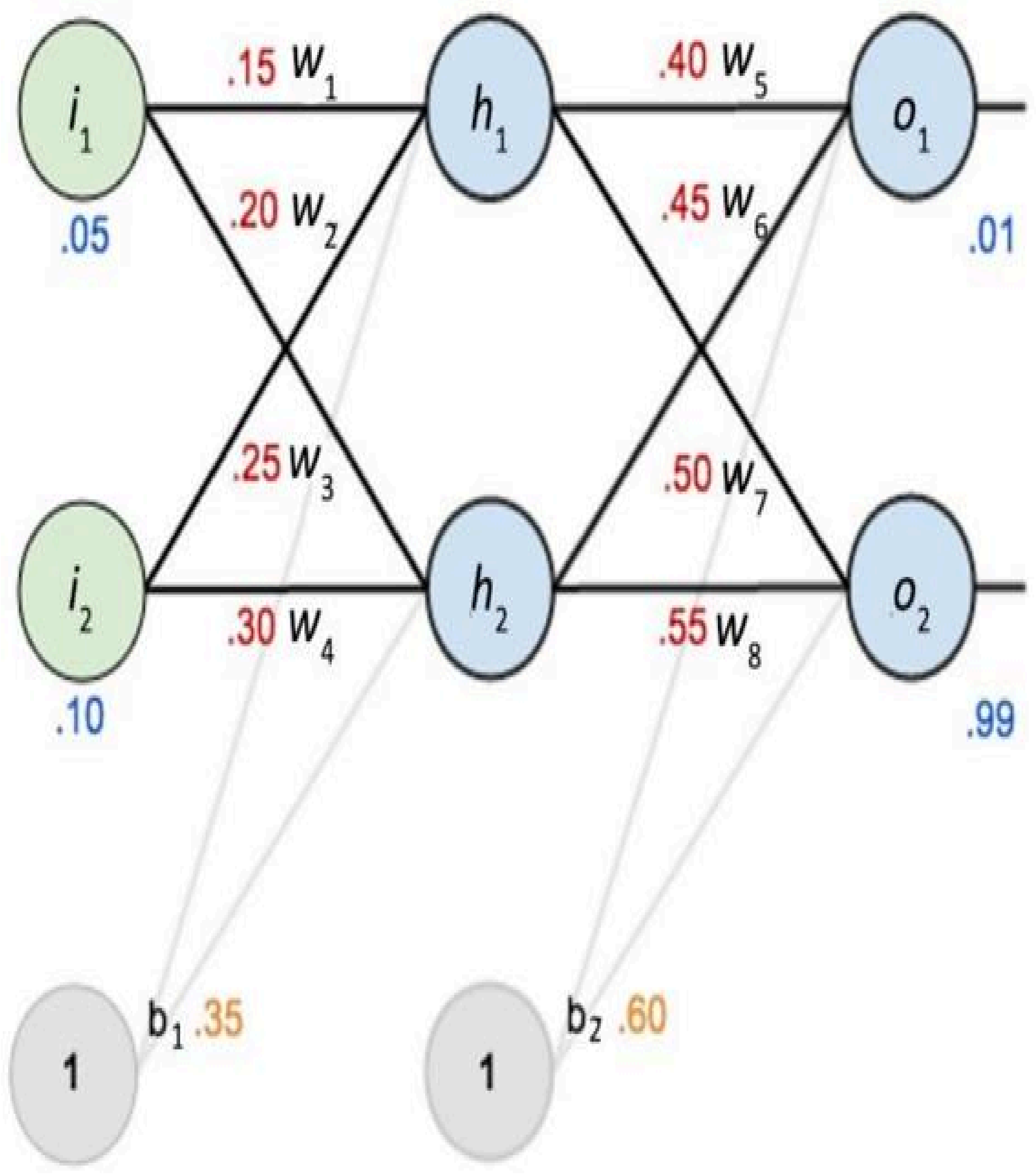


圖4-17　計算神經網路輸出流程圖

在圖4-17中，輸入層有兩個節點，我們將其依次編號為i_1、i_2；隱藏層有兩個節點，我們將其依次編號為h_1、h_2（假設啟用函式為sigmoid）；輸出層有兩個節點，我們將其依次編號為o_1、o_2。因為這個神經網路是全連線網路，所以可以看到每個節點都和上一層的所有節點有連線。所以，隱藏層的節點h_1的值為

$$h_1 = \text{sigmoid}(w_1 i_1 + w_2 i_2)$$

$$\because i_1 = 0.05, i_2 = 0.1,$$

初始權重為

$$w_1 = 0.15, w_2 = 0.20, w_3 = 0.25, w_4 = 0.30, w_5 = 0.40, w_6 = 0.45, w_7 = 0.50, w_8 = 0.55$$

隱藏層h_1的輸出值為

$$\text{out}_{h1} = \frac{1}{1 + e^{w_1 \times i_1 + w_2 \times i_2 + b_1 \times 1}} = 0.5932$$

同理，h_2的輸出值為

$$\text{out}_{h2} = \frac{1}{1 + e^{w_3 \times i_1 + w_4 \times i_2 + b_1 \times 1}} = 0.5968$$

隱藏層到輸出層o_1的計算與之類似，輸出層o_1的輸出值為

$$\text{out}_{h1} = \frac{1}{1 + \text{e}^{w_5 \times h_1 + w_6 \times h_2 + b_2 \times 1}} = 0.7513$$

同理，o_2的輸出值為

$$\text{out}_{h2} = \frac{1}{1 + e^{w_7 \times h_1 + w_8 \times h_2 + b_2 \times 1}} = 0.7729$$

3.神經網路的訓練演算法

接下來介紹神經網路的訓練演算法：反向傳播演算法。

首先根據上一節介紹的神經網路計算流程。這裡用樣本的特徵計算出神經網路中每個隱藏層節點的輸出，以及輸出層每個節點的輸出。對於單個輸出層節點的誤差項計算如下：

$$\delta_i = \sum \frac{1}{2}(\text{target}_i - \text{output}_i)^2$$

其中，δ_i是節點i的誤差項，output是節點的輸出值，target是樣本對應節點的目標值。於是o_1、o_2和總誤差分別為

$$\delta_{o_1} = \sum \frac{1}{2}(\text{target}_{o_1} - \text{output}_{o_2})^2$$

$$\delta_{o_2} = \sum \frac{1}{2}(\text{target}_{o_1} - \text{output}_{o_2})^2$$

$$\delta_{\text{total}} = \delta_{o_1} + \delta_{o_2}$$

下面更新隱藏層到輸出層的權重。

神經網路中權重的更新可以用之前介紹的隨機梯度下降演算法：

$$w_i' = w_i - \mu \frac{\partial \delta_{\text{total}}}{\partial w_i}$$

其中，w_i'為更新後的權重；μ為學習率，這裡將其設為0.5；$\frac{\partial \delta_{\text{total}}}{\partial w_i}$是誤差$\partial \delta_{\text{total}}$對每個權重$\partial w_i$的偏導數。

以權重w_5為例，根據鏈式法則：

$$\frac{\partial \delta_{\text{total}}}{\partial w_5} = \frac{\partial \delta_{\text{total}}}{\partial \text{output}_{o_1}} \times \frac{\partial \text{output}_{o_1}}{\partial \text{net}_{o_1}} \times \frac{\partial \text{net}_{o_1}}{\partial w_5}$$

下面分別計算每個子項的值。

首先計算$\frac{\partial \delta_{\text{total}}}{\partial \text{output}_{o1}}$：

$$\because \delta_{\text{total}} = \frac{1}{2}(\text{target}_{o_1} - \text{out}_{o_1})^2 + \frac{1}{2}(\text{target}_{o_2} - \text{out}_{o_2})^2$$

$$\therefore \frac{\partial \delta_{\text{total}}}{\partial \text{output}_{o_1}} = 2 \times \frac{1}{2}(\text{target}_{o_1} - \text{out}_{o_1})^{2-1} \times -1 + 0$$

$$\frac{\partial \delta_{\text{total}}}{\partial \text{output}_{o_1}} = -(\text{target}_{o_1} - \text{out}_{o_1}) = -(0.01 - 0.7513) = 0.7413$$

然後分別計算$\frac{\partial \text{output}_{o_1}}{\partial \text{net}_{o_1}}$和$\frac{\partial \text{net}_{o_1}}{\partial \text{w}_5}$：

$$\because \text{out}_{o_1} = \frac{1}{1+e^{-\text{net}_{o_1}}}$$

$$\therefore \frac{\partial \text{output}_{o_1}}{\partial \text{net}_{o_1}} = \text{out}_{o_1}(1-\text{out}_{o_1}) = 0.7513(1-0.7513) = 0.1868$$

$$\because \text{net}_{o_1} = w_5 \times \text{out}_{h_1} + w_6 \times \text{out}_{h_2} + b_2 \times 1$$

$$\therefore \frac{\partial \text{net}_{o_1}}{\partial w_5} = 1 \times \text{out}_{h_1} \times w_5^{(1-1)} + 0 + 0 = \text{out}_{h_1} = 0.5932$$

得到∂E_{total}對權重∂w_5的偏導數為

$$\therefore \frac{\partial \delta_{total}}{\partial w_5} = \frac{\partial \delta_{total}}{\partial \text{output}_{o_1}} \times \frac{\partial \text{output}_{o_1}}{\partial \text{net}_{o_1}} \times \frac{\partial \text{net}_{o_1}}{\partial w_5} = 0.7413 \times 0.1868 \times 0.5932 = 0.0821$$

最後，更新w_5的權重：

$$w_5' = w_5 - \mu \frac{\partial E_{total}}{\partial w_5} = 0.4 - 0.5 \times 0.0821 = 0.3589$$

4.6.2 例項演練

神經網路涉及的引數很多，包括神經網路的層數、每一層的隱藏單元數、啟用函式、損失函式等，這裡以一個簡單的例子入手，詳細介紹每個引數及如何選擇引數。

這個例子的資料集來自美國糖尿病、消化和腎臟疾病研究所，用於基於資料集中的診斷結果構建模型，以預測患者是否患有糖尿病。該資料集一共有768個例項、8個特徵。資料的樣例如表4-4所示。

表4-4　資料的樣例

孕期	2小时口服葡萄糖耐量实验中的血浆葡萄糖浓度	血　压	舒张压	胰岛素	体重指数	糖尿病血系功能	年龄	是否患有糖尿病
1	85	66	29	0	26.6	0.351	31	0
8	183	64	0	0	23.3	0.672	21	0
1	89	66	23	94	28.1	0.167	21	0

因此，它是二元分類問題（患者患有糖尿病的標籤為1，反之為0）。在這個例子中使用Keras作為深度學習框架。前幾章已經對Keras做了較詳細的介紹，這裡不再贅述。

具體程式碼如下：

```
1  from tensorflow.keras.models import Sequential
2  from tensorflow.keras.layers import Dense
3  import numpy
4  numpy.random.seed(7)
5  # 读取糖尿病训练数据
6  dataset = numpy.loadtxt("data/dnn/pima-indians-diabetes.data.csv",
7  delimiter=",")
8  # 将数据分为训练数据和标签，前 8 列为特征
9  X = dataset[:,0:8]
10 Y = dataset[:,8]
11 # 创建模型，这边创建 3 层神经网络，分别为输入层、隐藏层、输出层
12 model = Sequential()
13 model.add(Dense(4, input_dim=8, activation='relu'))
14 model.add(Dense(2, activation='relu'))
15 model.add(Dense(1, activation='sigmoid'))
16 # 编译模型
17 model.compile(loss='binary_crossentropy', optimizer='adam',
18 metrics=['accuracy'])
19 # 将特征和标签放入模型中
20 model.fit(X, Y, epochs=10, batch_size=32)
21 # 衡量模型效果
22 scores = model.evaluate(X, Y)
23 print("\n%s: %.2f%%" % (model.metrics_names[1], scores[1]*100))
```

seed函式用於指定隨機數生成時所用演算法一開始的整數值，如果使用相同的seed值，則每次生成的隨機數都相同，如果不設定這個值，則系統根據時間自己選擇這個值，這時每次生成的隨機數都因時間的差異而不同，例如：

```
from numpy import *
num=0
while(num<3):
    random.seed(7)
    print(random.random())
    num+=1
```

將輸出：

```
0.07630828937395717
0.07630828937395717
0.07630828937395717
```

在上面程式碼例項的第12行，我們首先建立了一個Sequential例項，Sequential模型是多個網路層的線性疊加。隨後我們向Sequential中新增了3個層，整個網路結構如圖4-18所示。

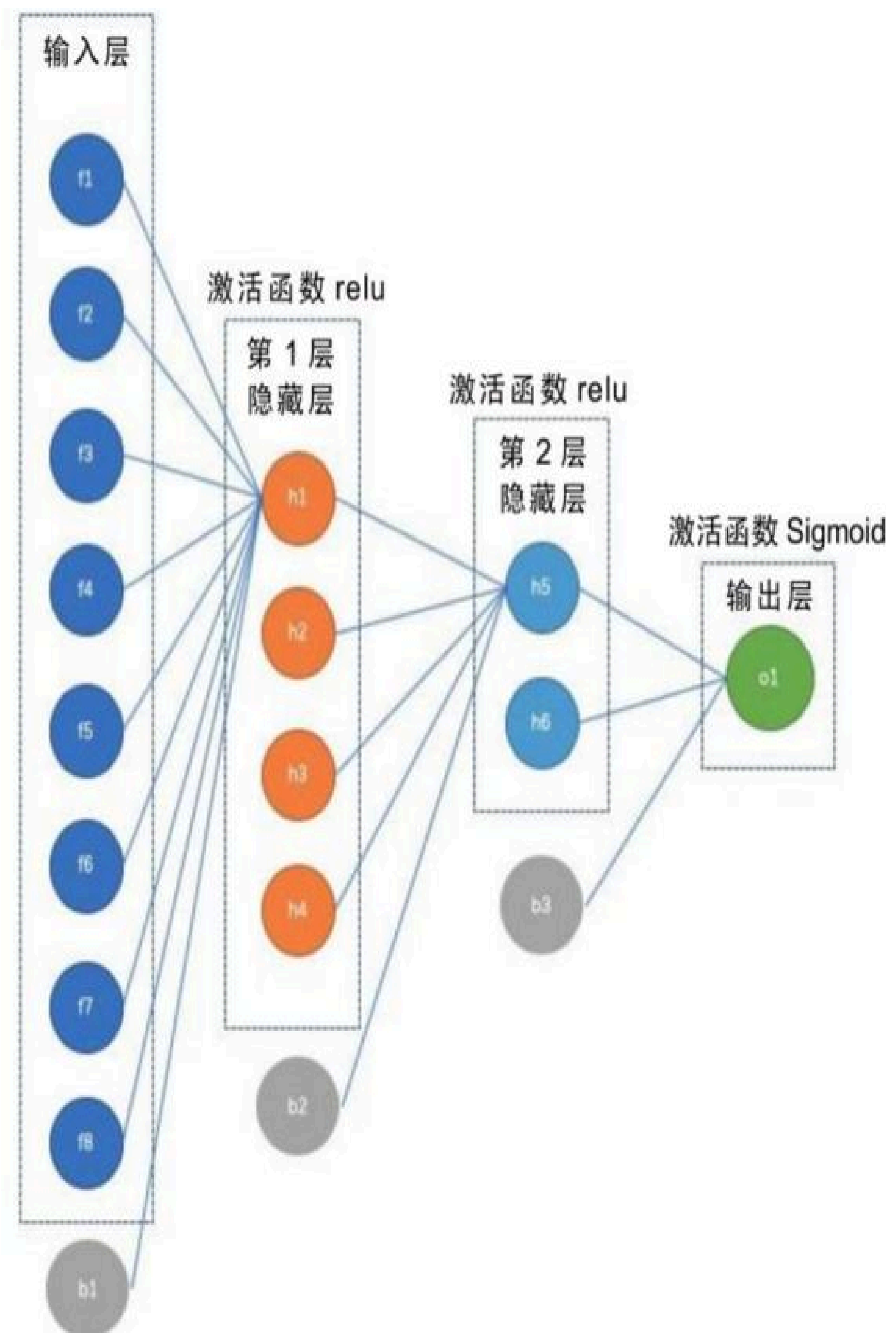

输入层
激活函数 relu
第 1 层
隐藏层
激活函数 relu
第 2 层
隐藏层
激活函数 Sigmoid
输出层

圖4-18　整個網路結構

Dense是常用的全連線層。第1層的dense指定啟用函式、輸入資料的特徵維度、輸出節點的數量；中間的Dense層（隱藏層）只需指定輸出單元的數量即可（從理論上來說，更深的網路可以得到更好的結果。但是透過簡單疊加隱藏層的方式來增加網路深度，可能引來梯度消失或梯度爆炸的問題，感興趣的讀者可以自行閱讀相關文獻）；網路的最後是輸出層，因為這是一個二分類問題，所以最後的啟用函式選擇sigmoid。在搭建整個網路時，要做的第1件事是確保輸入層有正確數量的輸入。Input_dim是第1層神經網路的輸入特徵個數，因為我們的資料有8個特徵，所以input_dim 為8。

在搭建完網路的整體框架後，需要對模型進行編譯：

```
    model.compile(loss='binary_crossentropy', optimizer='adam',
metrics=['accuracy'])
```

compile函式會透過Keras後端（Theano或Tensorflow）根據裝置的硬體條件CPU、GPU或分散式選擇最佳的方式編譯模型。在編譯時需要指定訓練網路時所需的一些引數。

◎　最佳化函式（optimizer）：告訴模型往哪個方向最佳化。常用的方法有梯度下降等，詳細介紹請參考4.6.3節。

◎　損失函式（loss）：模型試圖最小化的目標函式，常見的有交叉熵（categorical_crossentropy）或均方誤差（mean square error）。

◎　評估標準（metrics）：用於評估當前模型的效能。

最後，透過呼叫模型上的fit函式可以載入資料用於訓練模型。如下所示，X、Y分別對應訓練集和標籤：

```
    model.fit(X, Y, epochs=10, batch_size=32)
```

其中，當一個完整的資料集透過神經網路一次並且返回一次時，這個過程就被稱為一個 epoch；在不能將資料一次性透過神經網路

時，就需要將資料集分成幾個 batch。batch-size就是一個 batch 中的樣本總數。

4.6.3　深度學習中的一些演算法細節

很多同學在學習深度學習時，都會盲目嘗試一些引數或者函式，忽視了一些演算法細節。本節重點講解深度學習中比較重要且我們應該掌握的演算法細節。

1.optimizer

4.6.1節介紹了神經網路的計算過程。其實整個網路的訓練過程就是計算合適的w和b的過程，即儘可能減少預測值和真實值的誤差。最佳化函式optimizer就是告訴我們往哪個方向去最佳化。同時，選擇合適的最佳化器會加速整個神經網路的訓練過程，避免在訓練過程中遇到鞍點（區域性最優解）。

1）隨機梯度下降

隨機梯度下降是一種常見的最佳化方法，即每次都迭代計算mini batch的梯度，然後對引數進行更新。其公式為

$$\theta_{t+1} = \theta_t - \mu\nabla_\theta J(\theta)$$

其中，μ是learning rate，控制模型的學習進度；$J(\theta)$是我們定義的損失函式；∇_θ是對損失函式中的變數θ求導。隨機梯度下降的精髓是隻用一個訓練資料近似所有樣本，來調整θ。因而隨機梯度下降會帶來一定的問題，因為計算得到的並不是準確的梯度。對於最最佳化問題，雖然不是每次迭代得到的損失函式都朝著全域性最優方向，但整體是朝著全域性最優解方向的，最終的結果往往在全域性最優解附近。但是相對於其他方法，隨機梯度下降更快，其缺點是對損失方程有比較嚴重的振盪，並且容易收斂到區域性最小值。

2）Momentum

為了克服SGD振盪比較嚴重的問題，Momentum將物理中的動量概念引入SGD中，透過積累之前的動量來替代梯度。即

$$\theta_t = \gamma\theta_{t-1} + \mu\nabla_\theta J(\theta)$$

相較於SGD，Momentum就相當於從山坡上不停地向下走，如果沒有阻力，它的速度就會越來越快，但是如果遇到了阻力，速度就會變慢。也就是說，在訓練時，在梯度方向不變的維度上，訓練速度變快，在梯度方向有所改變的維度上，訓練速度變慢，這樣就可以加快收斂並減小振盪。

3）Adagrad

相較於SGD，Adagrad相當於對學習率多加了一個約束，即

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\mu}{\sqrt{\sum g_{t,i} + \varepsilon}}$$

Adagrad的優點是，在訓練初期，由於$g_{t,i}$較小，所以約束項能夠加速訓練。而在後期，隨著$g_{t,i}$的變大，分母也會不斷變大，最終訓練提前結束。

4）Adam

Adam是Momentum與Adagrad相結合的產物，既考慮到利用動量項來加速訓練過程，又考慮到對學習率的約束，並利用梯度的一階矩估計和二階矩估計動態調整每個引數的學習率。Adam的優點主要在於經過偏置校正後，每一次迭代的學習率都有個確定的範圍，使得引數比較平穩。其公式為

$$\theta_{t+1} = \theta_t - \frac{\mu}{\sqrt{v_t^1} + \varepsilon} m_t^1$$

其中：

$$m_t^1 = \frac{m_t}{1 - \beta_1^t}$$

$$v_t^1 = \frac{v_t}{1 - \beta_2^t}$$

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

實踐證明，Adam結合了Adagrad善於處理稀疏梯度和Momentum善於處理非平穩目標的優點，相較於其他幾種最佳化器效果更好。

2．epoch

當一個完整的資料集透過了神經網路一次並且返回了一次時，這個過程就被稱為一個epoch。然而，當一個epoch對於計算機而言太龐大時，就需要把它分成多個小塊。

為什麼要使用多於一個epoch？這一開始聽起來會讓人覺得很奇怪。在神經網路中傳遞一次完整的資料集是不夠的，而且我們需要將完整的資料集在同樣的神經網路中傳遞多次。但請記住，我們使用的是有限的資料集，並且使用了一個迭代過程即梯度下降，該最佳化學習過程如圖4-19所示，因此僅僅更新權重一次或者說使用一個epoch是不夠的。

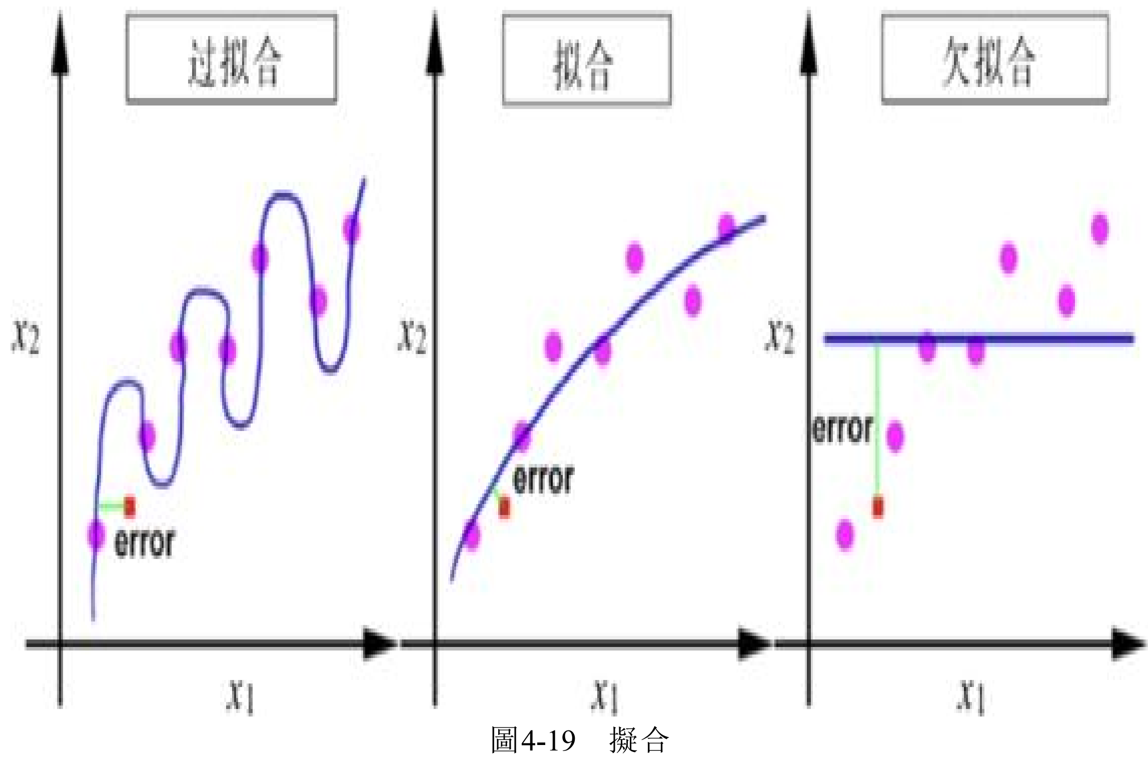


圖4-19　擬合

隨著epoch數量的增加，神經網路中權重的更新次數也增加，曲線從欠擬合變得過擬合。

3.batch_size

深度學習的訓練過程將在epoch中執行固定數量的迭代，我們必須使用epoch引數指定訓練需要的epoch數。但在很多實際應用中，我們很難將所有資料一次性放入神經網路中去訓練。這時就需要將資料集分成幾個batch。每個batch中訓練資料的數量就是batch_size。

在之前的程式碼中，我們對糖尿病資料集設定10輪訓練（epoch=10）。一次訓練使用的訓練資料是32條（batch_size = 32），所以訓練一輪總共需要迭代24次（768/32）。訓練10輪一共需要240次迭代。

選擇一個適合的batch_size有什麼好處呢？如果資料集足夠充分，那麼用部分資料訓練出來的模型與用全部資料訓練出來的模型幾乎是一樣的。

4.7　本章小結

本章對機器學習的主要演算法做了簡要介紹和程式碼說明，對前面沒有深入講解的內容從概念到理論做了分析和實踐。結合本章所講的機器學習相關概念和理論，以及前面章節對深度網路應用開發的介紹，我們現在對機器學習演算法的使用應該不再陌生，能夠開始對現實專案中的部分問題自己開發、解決了。

從第5章開始，我們將進入機器學習、深度學習的不同應用領域，對相關技術在真正業界難題上如何應用建立基礎，並提供技術參考和實現。

4.8 本章參考文獻

[1] https://scikit-learn.org/stable/modules/classes.html#module-sklearn.datasets

[2] https://scikit-learn.org/stable/

[3] https://archive.ics.uci.edu/ml/datasets/lenses

下篇

第5章　推薦系統基礎

推薦系統是機器學習最重要的應用領域之一，Google、Facebook、淘寶、頭條、抖音等無一不將龐大而精準的推薦系統作為基礎。

推薦系統從早期的協同過濾到現在基於深度學習的方案，發生了巨大的變化，這裡集中講解兩種經典的實現方案：協同過濾和邏輯迴歸，希望讀者能對推薦系統的實現有一個基本的認知。

本章內容比較精簡，建議讀者對本章例項親自計算並嘗試，而不要只閱讀、不實踐。

5.1　推薦系統簡介

從業務上來說，推薦系統通常指應用資料分析技術找出使用者最可能喜歡的內容，並將相應的內容推薦給使用者；從技術上來說，推薦系統通常指應用資料分析技術從海量資料中根據一定條件篩選資料，並優先提供最匹配的資料。因此，在推薦系統中包括以下3個關鍵實現步驟：

◎　如何在海量（千萬級別以上）資料中進行快速篩選；

◎　如何對資料的匹配程度進行判斷（打分）；

◎　如何在挑選出來的匹配結果中進行再次處理，讓最匹配的結果優先展示。

瞭解資料庫操作的讀者也許會立刻想到：這不就是資料庫的常見SELECT操作嗎？例如在社交軟體中，在推薦可能有共同話題的其他使用者時，可能只需選擇年齡範圍即可：

```
SELECT * FROM users WHERE age>=20 AND age<30 ORDER BY age ASC LIMIT 10
```

上面這條SQL語句用於在使用者表中選擇年齡為20～30的所有使用者，按照年齡從小到大排序，挑選年齡最小的前10位使用者。這基本上就完成了上面所說的3個步驟：在大量資料（使用者表）中篩選；按照年齡範圍進行匹配；排序後選擇前10位作為結果（優先展示最匹配的結果）。

這樣的實現在大量應用和系統中都可以獲得很好的效果，也是資料庫的基本功能之一，能夠充分滿足資料量較小時的各種業務需求。然而，在資料量達到一個量級，例如user表達到千萬條記錄之上後，同時查詢條件不僅僅侷限於年齡範圍，還包括地理位置、興趣愛好、工作職位、畢業學校及使用者的其他諸多相關屬性時，查詢就變成一個相當耗時且複雜的難題。

首先，這是個非常耗時的大資料處理問題。如果要同時處理千萬甚至上億的使用者搜尋，則給資料庫帶來的壓力是巨大的，也並非是單一資料庫伺服器所能承載的，我們要考慮資料的分片（Sharding）和水平擴容（Horizontal Scaling），以及如何高效地併發處理使用者資料，這些更多地偏向工程問題，這裡不再贅述。

其次，這把使用者搜尋變成了一個沒有固定答案的演算法挑戰。當我們可以利用的使用者屬性越多時，如何決定正確的匹配條件就成了一個開放性問題，假如根據年齡和地理位置匹配，那麼在排序時應該年齡優先還是地理位置優先呢？如果加上使用者的其他資訊如畢業學校、學歷、專業、興趣愛好等，條件就更加複雜。社交軟體中的使用者推薦還可以基於一些特有屬性，或者讓使用者自行設定搜尋條件去完成，不至於給應用的體驗產生太大差異。進入資訊流時代後，我們需要主動為使用者提供最匹配的資訊，主動針對使用者的特點採用不同的匹配演算法，這時不同的實現方案帶來的就是截然不同的體驗，甚至會直接決定一個產品在市場上的存亡，這也是為什麼推薦系統會成為機器學習演算法最典型、最成功的應用領域。

目前應用比較廣泛和成熟的推薦演算法是協同過濾（Collaborative Filtering，CF），該演算法的基本思想是根據使用者之前的喜好及興趣相近的使用者的選擇來向使用者推薦內容。在講解演算法之前，我們先從整體架構上看看推薦系統的工作流程，如圖5-1所示。

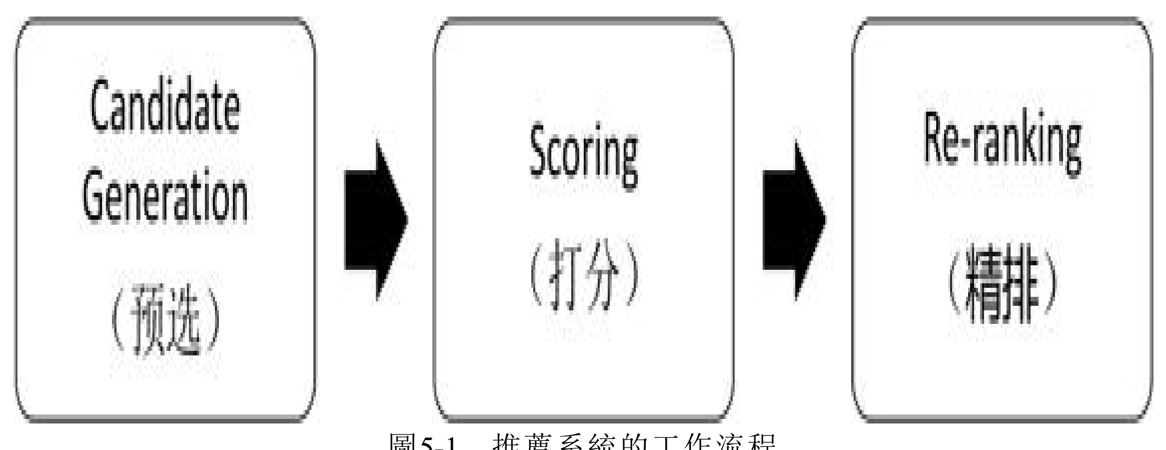


圖5-1　推薦系統的工作流程

圖5-1出自Google開發者網站，它把推薦系統的工作流程分為以下三個階段。

（1）預選階段。在該階段，對應的模型將在大量的資料中快速選出待用資料，例如在淘寶上的所有商品中快速選出幾萬個可用選項。該階段的模型包括多種演算法，每種都針對某個特定型別的選項。

（2）打分階段。在該階段，另一個模型會對第1階段的待用選項進行更精細的打分並排序。因為此時的資料已經減少到我們可接受的階段，所以模型能使用更細緻的方式來處理。

（3）精排階段。在該階段，系統通常會引入一些額外的資訊，例如使用者曾明確註明不想要的選項等，或者強化最新的內容，保證最終結果的多樣性、時效性和公正性。

所有演算法都是針對這3個階段設計的。嚴格地說，打分階段和精排階段是對資料的精細調整，需要結合業務的特點做有針對性的工作。比如，在精排階段，我們很可能要做專業的異常檢測來判斷即將展示給使用者的內容是否包含敏感資訊，抑或最後檢測金融相關的資料是否是虛假資料等。因此這裡只關注預選階段的工作。

在預選階段主要採用了過濾演算法，該演算法主要包括以下兩種方法。

（1）基於內容的過濾（Content-based Filtering）：該方法只將備選項本身的相似度作為推薦依據，例如使用者檢視的兩個物品都是平面電視機，系統就會為該使用者推送大量的平面電視產品。

（2）協同過濾：在協同過濾中決定推薦結果的除了備選項本身的相似度，還包括使用者自身的相關資訊等。例如，使用者*A*和使用者*B*的身份資訊相似（年齡、性別、地區），如果使用者*B*買了某件商品，那麼系統很有可能也會把該商品推薦給使用者*A*。

注意，在上述兩種方法中，「相似」這個詞語被多次提及。實際上，無論是協同過濾還是基於內容的過濾，其根本內容都是如何計算相似度，下面會進行具體講解。

5.2 相似度計算

我們可以把圖5-2看作只有2維的向量空間（儘管在實際應用中一個商品屬性的向量空間遠遠大於2維）。假設*A*、*B*、*C*分別是3個商品的向量，則對於向量搜尋查詢，我們只需找到距離query向量最近的商品即可。

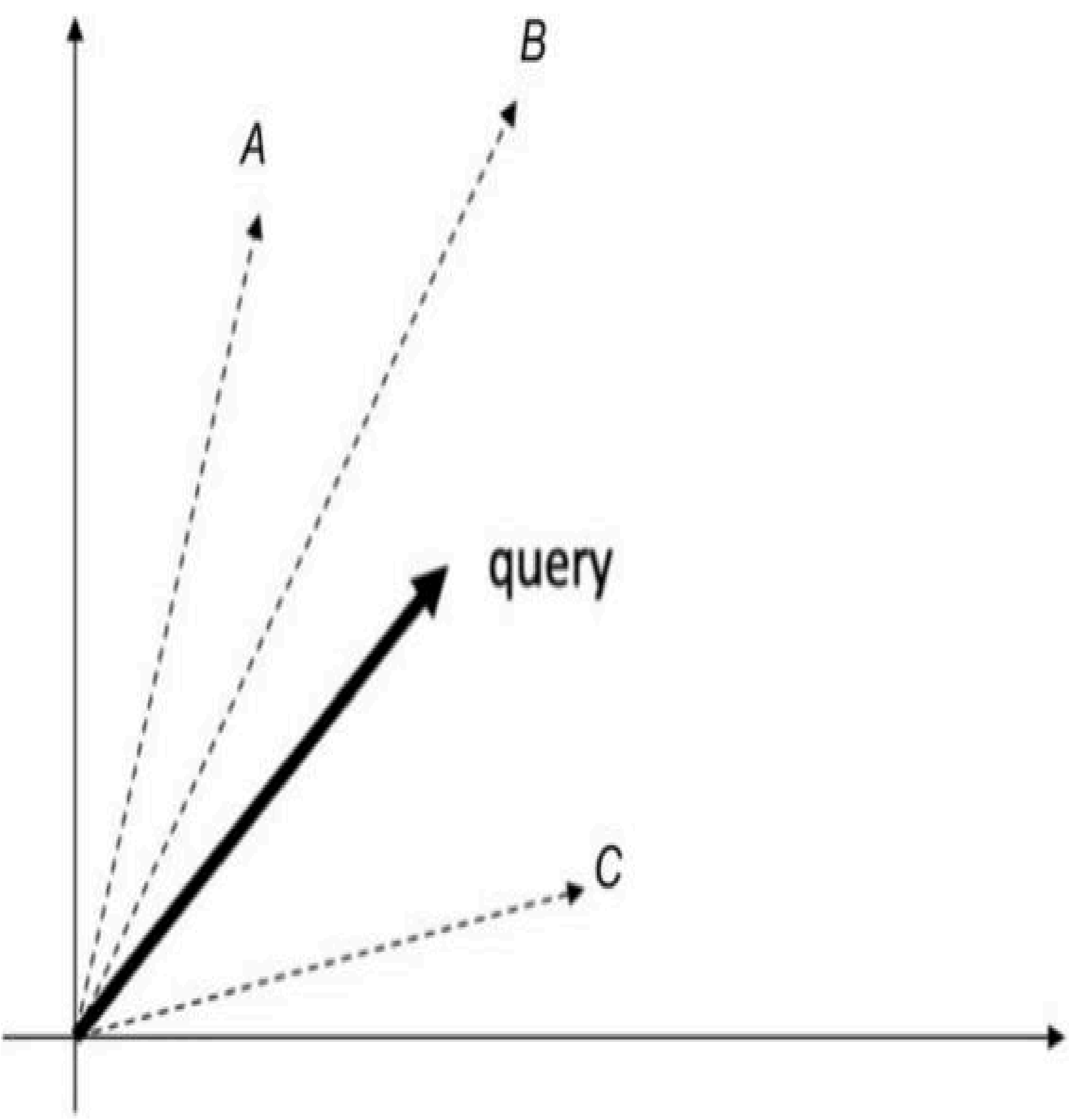


圖5-2　相似度

於是，我們可以使用的相似度度量方法有以下3種。

（1）餘弦相似度（Cosine，也叫作餘弦距離）。直接比較二者的夾角餘弦值：

$$S(q,x)=\cos(q,x)$$

（2）點積距離（Dot）。直接對二者做點積運算：

$$S(q,x) = \sum_{1}^{n} q_i \cdot x_i$$

（3）歐氏距離（Euclidean）。向量空間中的歐氏距離為

$$S(q,x) = \sqrt{\sum_{1}^{n} (q_i - x_i)^2}$$

實際上，我們可以根據不同的情況選擇合適的相似度衡量，一般常用的是餘弦相似度，下面會提到其中的緣由。

5.3　協同過濾

協同過濾是目前應用最廣泛也比較基礎的推薦演算法，可分為以下三類。

（1）基於記憶（Memory-based）的協同過濾：透過使用者打過分的資料計算使用者或商品之間的相似度關係，比較經典的有基於使用者（User-based）的協同過濾和基於物品（Item-based）的協同過濾。

（2）基於模型（Memory-based）的協同過濾：通常會使用資料探勘、機器學習方法搭建模型，並預測使用者對於未使用過的商品的一個可能的打分，比較經典的有貝葉斯網路（Bayesian Network）、聚類模型（Clustering Model）等。

（3）基於混合模型的協同過濾：透過基於記憶和模型的協同過濾，來克服在傳統的協同過濾中資料稀疏與資訊損失的主要問題，同時提高預測的準確度。目前大部分商用推薦系統都用到了該類演算法。

5.3.1　基於使用者的協同過濾

基於使用者的協同過濾演算法先使用相似度計算公式得到與目標使用者有相同喜好的*K*個最相似使用者（Nearest Neighbor），然後根據這些相似使用者的喜好生成對目標使用者的推薦。它在計算上，就是將一個使用者對所有物品的偏好作為一個向量來計算使用者之間的相似度。在找到*K*個最相似鄰居後，根據這些鄰居的相似度權重及其對物品的偏好，預測當前使用者對未接觸過的物品的喜好程度，最後計算得到一個已排序的物品列表作為推薦。例如，我們把電影當作某種物品，如表5-1所示則是使用者*A*、*B*、*C*對5部電影的評分。

表5-1　使用者*A*、*B*、*C*對5部電影的評分

	电影A	电影B	电影C	电影D	电影E	平均值
用户A	4	1	-	4	-	3
用户B	-	4	-	2	3	3
用户C	-	1	-	4	4	3

在這個例子中，對於目標使用者*A*，我們透過計算使用者*A*和*B*及*A*和*C*之間的相似度，得到與使用者*A*最相近的一個使用者*C*，然後將使用者*C*喜歡的電影*A*推薦給使用者*A*。這裡比較一下歐氏距離和餘弦相似度的差別。

（1）對歐氏距離的計算如下：

$$d(x,y) = \sqrt{(\sum (x_i - y_i)^2)} \qquad \mathrm{sim}(x,y) = \frac{1}{1 + d(x,y)}$$

例如，我們要計算使用者*A*和使用者*B*，以及使用者*A*和使用者*C*的相似度，則

$$d(A,B) = \sqrt{(1-4)^2 + (4-2)^2} = 3.6$$

$$d(A,C) = \sqrt{(1-1)^2 + (4-4)^2} = 0$$

可見使用者*A*和使用者*C*更相似。

（2）對餘弦相似度的計算如下：

$$T(x,y) = \frac{x \cdot y}{\|x\|^2 \times \|y\|^2} = \frac{\sum x_i y_i}{\sqrt{\sum x_i{}^2}\sqrt{\sum y_i{}^2}}$$

其中，x_i、y_i分別是不同的使用者向量。

同樣，我們來看使用者*A*、*B*及使用者*A*、*C*的餘弦距離：

$$d(A,B)=\frac{1\times4+4\times2}{\sqrt{1^2+4^2}\times\sqrt{4^2+2^2}}=1.40$$

$$d(A,C)=\frac{1\times1+4\times4}{\sqrt{1^2+4^2}\times\sqrt{1^2+4^2}}=1.01$$

可以看到，仍然是使用者A和C更加相似，而且相似度比用歐氏距離計算更明顯。

圖5-3展示了歐氏距離和餘弦相似度這兩種方法的主要區別：歐氏距離衡量的是空間各點間的絕對距離，與各點所在的位置座標（即個體特徵維度的數值）直接相關；餘弦相似度衡量的是空間向量的夾角，更加體現方向上的差異，而不是位置。

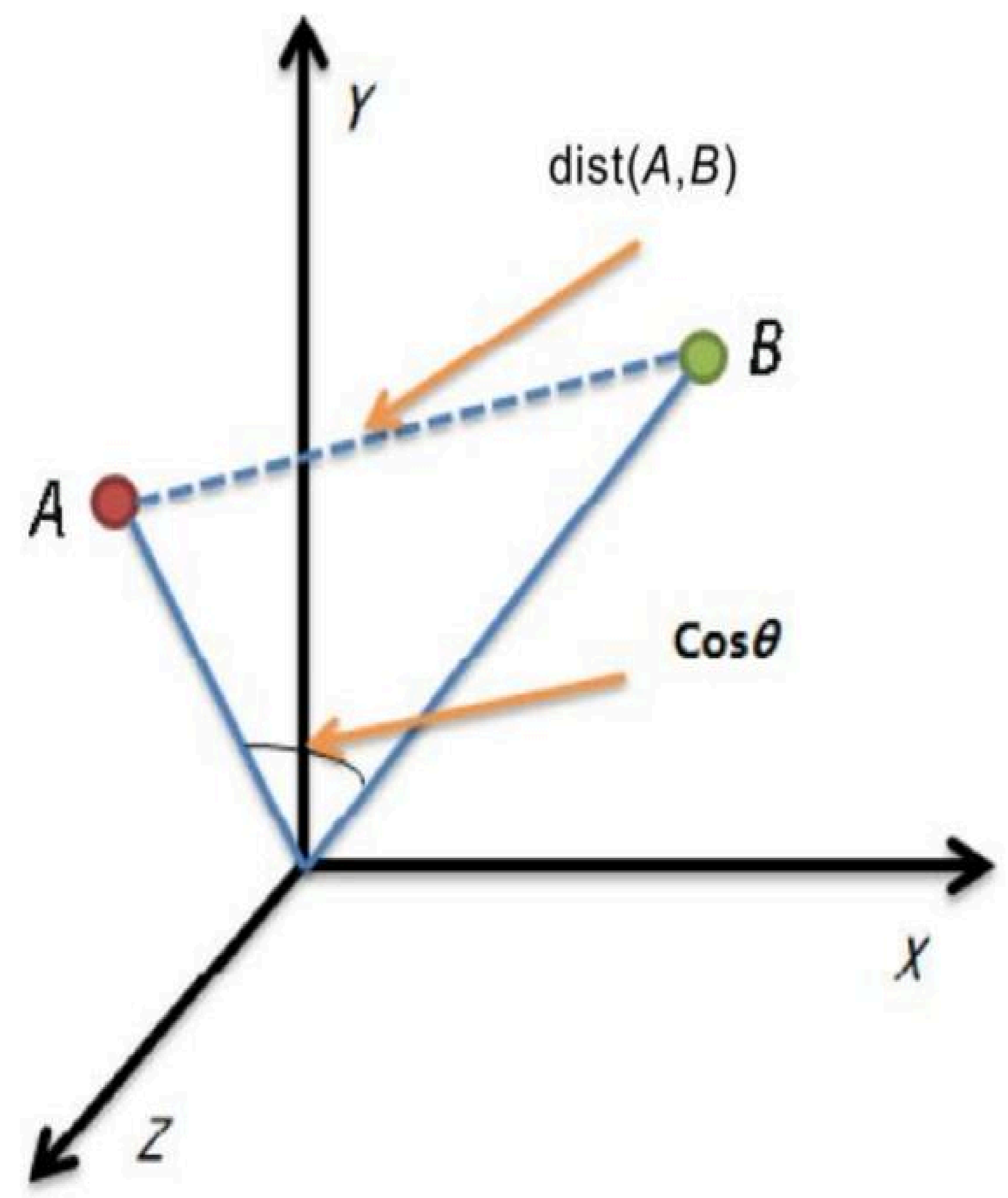


圖5-3　相似度計算

在圖5-3中，如果保持A點的位置不變，將B點朝原方向遠離座標軸原點，那麼這時餘弦相似度Cosθ是保持不變的，因為夾角不變；而A、B兩點的距離顯然在發生改變。所以，歐氏距離被更多地用於從維度的數值大小中體現差異的分析，例如使用使用者的行為指標分析使用者價值的相似度或差異；而餘弦相似度更能從方向上區分差異，而

對具體方向上的絕對數值不敏感，被更多地用於使用使用者對內容的評分來區分使用者興趣的相似度和差異，同時修正了使用者之間可能存在的度量標準不統一的問題（因為餘弦相似度對絕對數值不敏感）。在本章後續的例子和程式碼中會統一使用餘弦相似度。

5.3.2　基於物品的協同過濾

基於使用者的協同過濾隨著使用者數量的增多，計算時間就會變長，所以在2001年，Sarwar等人提出了基於物品的協同過濾。基於物品的協同過濾的原理和基於使用者的協同過濾類似，只是用物品（Item）之間的相似度來代替使用者之間的相似度，即基於使用者對物品的偏好找到相似的物品，然後根據使用者的歷史偏好推薦相似的物品給他（她）。從計算的角度來看，就是將所有使用者對某個物品的偏好作為一個向量來計算物品之間的相似度，在得到物品的相似物品後，根據使用者的歷史偏好預測使用者當前沒有偏好的物品，計算得到一個已排序的物品列表作為推薦。

還是以5.3.1節的例子來說明。首先，計算各電影之間的相似度，在基於物品的協同過濾中，除了可以使用之前介紹的餘弦相似度方法，還可以使用傑卡爾德相似度（Jaccard Similarity）方法，公式如下：

$$w_{i,j} = \frac{|N(i) \cap N(j)|}{|N(i) \cup N(j)|}$$

這裡，分母是分別喜歡物品i和物品j的使用者數，分子是同時喜歡兩個物品的使用者數。根據上面這個公式，我們可以算出5部電影之間的相似度分數（注意，這裡暫時忽略電影分數，只把是否看過作為喜歡和不喜歡的依據），如表5-2所示。

表5-2　5部電影之間的相似度分數

	电影A	电影B	电影C	电影D	电影E
电影A	0	0.33	0	0.33	0
电影B	0.33	0	0	1.0	0.66
电影C	0	0	0	0	0
电影D	0.33	1.0	0	0	0.66
电影E	0	0.66	0	0.66	0

在得到物品之間的相似度後，我們可以根據如下公式計算使用者u對未看過的電影j的興趣（u、j為變數）：

$$P_{AE} = \sum_{i \in N(u) \cap S(j,K)} w_{ji} r_{ui}$$

$$\{p_1, p_2, p_3 \dots\dots\} \subseteq \{M - N\}$$

這裡，$N(u)$是使用者喜歡的電影的集合，$S(j,K)$是和電影j最相似的K個電影的集合，w_{ji}是電影j和i的相似度，r_{ui}是使用者u對電影i的興趣（對於隱反饋資料集，如果使用者u對電影i有過行為，比如看過電影i，即可令$r_{ui}=1$）。和使用者歷史上感興趣的電影越相似的電影，就越有可能在使用者的推薦列表中獲得比較靠前的排名。

例如，使用者B已經看過電影B、D、E，若想向其推薦下一部電影，那麼根據計算，在剩下的電影A、C裡面，只有A的得分較高，所

以向使用者B推薦電影A。

5.3.3 演算法實現與案例演練

下面透過一個實際案例詳細介紹基於使用者的協同過濾及其實現，其中的資料集使用了由GroupLens研究組提供的MovieLens資料集。MovieLens是一個收集了使用者對已看過電影進行評分的資料集合，共有3種不同的資料大小，分別被命名為1M、10M和20M。最大的資料集有約14萬使用者的資料，覆蓋約27000部電影。在本案例中會使用10M的資料集（下載地址見本章參考文獻[1]）。除了評分，MovieLens資料還包含類似「Western」的電影流派資訊和使用者應用的標籤，例如「over the top」和「Arnold Schwarzenegger」。這些流派標記和標籤在構建內容向量方面是有用的。

下面的程式碼主要介紹如何讀取這一資料集，以及如何應用這一資料集去做協同過濾演算法。其中，getRatingInformation(ratings)讀取使用者的打分資料 u.data，並將其載入一個List中，u.data的每一行分別對應使用者ID、電影ID和使用者評分。程式碼如下：

```
def getRatingInformation(ratings):
    rates=[]
    for line in ratings:
        rate=line.split("\t")
        rates.append([int(rate[0]),int(rate[1]),int(rate[2])])
    return rates
#
#   生成用户评分的数据结构
#
#   输入:索引数据 [[2,1,5],[2,4,2]...]
#   输出:①用户打分字典，②电影字典
#   使用字典，key是用户ID，value是用户对电影的评价，
#   rate_dic[2]=[(1,5),(4,2)]... 表示用户2对电影1的评分是5，对电影4的评分是2
def createUserRankDic(rates):
    user_rate_dic={}
    item_to_user={}
    for i in rates:
        user_rank=(i[1],i[2])
        if i[0] in user_rate_dic:
            user_rate_dic[i[0]].append(user_rank)
        else:
            user_rate_dic[i[0]]=[user_rank]
        if i[1] in item_to_user:
```

```
            item_to_user[i[1]].append(i[0])
        else:
            item_to_user[i[1]]=[i[0]]
    return user_rate_dic,item_to_user
```

recommendByUserCF(test_rates)是基於使用者協同過濾的主函式：

```
#
#   使用UserFC进行推荐
#   输入：文件名、用户ID、邻居数量
#   输出：推荐的电影ID、输入用户的电影列表、电影对用户的序列表、邻居列表
#
def recommendByUserCF(file_name,userid,k=5):
    # 读取文件数据
    test_contents=readFile(file_name)
    # 将文件数据格式化成二维数组 List[[用户ID,电影ID,电影评分]...]
    test_rates=getRatingInformation(test_contents)
    # 格式化成字典数据
    #    1.用户字典：dic[用户ID]=[(电影ID,电影评分)...]
    #    2.电影字典：dic[电影ID]=[用户ID1,用户ID2...]
    test_dic,test_item_to_user=createUserRankDic(test_rates)
    # 寻找K个相似用户
    neighbors=calcNearestNeighbor(userid,test_dic,test_item_to_user)[:k]
    recommend_dic={}
    for neighbor in neighbors:
        neighbor_user_id=neighbor[1]
        movies=test_dic[neighbor_user_id]
        for movie in movies:
            if movie[0] not in recommend_dic:
                recommend_dic[movie[0]]=neighbor[0]
            else:
                recommend_dic[movie[0]]+=neighbor[0]
    # 建立推荐列表
    recommend_list=[]
    for key in recommend_dic:
        recommend_list.append([recommend_dic[key],key])
    recommend_list.sort(reverse=True)
    user_movies = [ i[0] for i in test_dic[userid]]
return [i[1] for i in
recommend_list],user_movies,test_item_to_user,neighbors
```

5.4　LR模型在推薦場景下的應用

第4章介紹了邏輯迴歸模型及其在分類場景下的一些應用。同樣，LR模型常常被應用於推薦系統中，作為與其他模型比較的基礎模型。本節依然選用movielens的資料作為訓練資料，透過訓練一個推薦模型，向使用者推薦他沒看過但是可能感興趣的電影。為了得到更好的推薦效果，我們引入了更多的特徵到模型中。這裡考慮將電影型別加入，新資料集的下載地址見本章參考文獻[2]。

在新資料集中主要有兩個檔案：rating.csv和movies.csv。

rating.csv檔案的內容格式如圖5-4所示。

userId	movieId	rating	timestamp
1	1	4	964982703
1	3	4	964981247

圖5-4　rating.csv檔案的內容格式

movies.csv檔案的內容格式如圖5-5所示。

movieId	title	genres
1	Toy Story (1995)	Adventure|Animation|Children|Comedy|Fantasy
2	Jumanji (1995)	Adventure|Children|Fantasy
3	Grumpier Old Men (1995)	Comedy|Romance

圖5-5　movies.csv檔案的內容格式

genres是電影型別，有20種：'Horror'、'Western'、'(no genres listed)'、'Romance'、'Action'、'Thriller'、'War'、'Comedy'、'Musical'、'I

MAX'、'Film-Noir'、'Documentary'、'Fantasy'、'Children'、'Adventure'、'Animation'、'Mystery'、'Crime'、'Drama'和'Sci-Fi'。

接下來需要對資料進行處理，將資料變成訓練資料，主要步驟如下。

（1）將使用者評分rating轉換成label。在這裡規定評分小於3的電影，是使用者不喜歡的電影，即label = 0，反之label = 1。

（2）將genres透過One-hot轉換成0、1特徵，程式碼如下：

```
# 将genres转换成One-hot特征
def convert_2_one_hot(df):
    genres_vals = df['genres'].values.tolist()
    genres_set = set()
    for row in genres_vals:
        genres_set.update(row.split('|'))
    genres_list = list(genres_set)
    row_num = 0
    df_new = pd.DataFrame(columns=genres_list)
    for row in genres_vals:
        init_genres_vals = [0] * len(genres_list)
        genres_names = row.split('|')
        for name in genres_names:
            init_genres_vals[genres_list.index(name)] = 1
        df_new.loc[row_num] = init_genres_vals
        row_num += 1

    df_update = pd.concat([df, df_new], axis=1)
    return df_update
# 将rating转换成0、1分类
def convert_rating_2_labels(ratings):
    label = []
    ratings_list = ratings.values.tolist()
    for rate in ratings_list:
        if rate >= 3.0:
            label.append(1)
        else:
            label.append(0)
    return label
```

（3）將處理後的訓練資料和標籤放到LR模型中訓練，完整的程式碼如下：

```
1  import pandas as pd
2  from sklearn.linear_model import LogisticRegression
3  from sklearn.metrics import roc_auc_score
4  from sklearn.model_selection import train_test_split
5
6  movies_path = './movies.csv'
```

```
7   ratings_path = '. /ratings.csv'
8
9   # 将 rating 转换成 0、1 分类
10  def convert_rating_2_labels(ratings):
11      label = []
12      ratings_list = ratings.values.tolist()
13      for rate in ratings_list:
14          if rate >= 3.0:
15              label.append(1)
16          else:
17              label.append(0)
18      return label
19
20  # 将 genres 转换成 One-hot 特征
21  def convert_2_one_hot(df):
22      genres_vals = df['genres'].values.tolist()
23      genres_set = set()
24      for row in genres_vals:
25          genres_set.update(row.split('|'))
26      genres_list = list(genres_set)
27      row_num = 0
28      df_new = pd.DataFrame(columns=genres_list)
29      for row in genres_vals:
30          init_genres_vals = [0] * len(genres_list)
31          genres_names = row.split('|')
32          for name in genres_names:
33              init_genres_vals[genres_list.index(name)] = 1
34          df_new.loc[row_num] = init_genres_vals
35          row_num += 1
36
37      df_update = pd.concat([df, df_new], axis=1)
38      return df_update
39
40  # 构建逻辑回归模型
41  def training_lr(X, y):
```

```
42     model = LogisticRegression(penalty='l2', C=1, solver='sag', max_iter=500,
43 verbose=1, n_jobs=8)
44     X_train, X_test, y_train, y_test = train_test_split(X, y, test_size =
   0.1,
45     random_state = 42)
46     model.fit(X_train, y_train)
47     train_pred = model.predict_proba(X_train)
48     train_auc = roc_auc_score(y_train, train_pred[:, 1])
49
50     test_pred = model.predict_proba(X_test)
51     test_auc = roc_auc_score(y_test, test_pred[:, 1])
52
53     # print(model.score())
54     print('lr train auc score: ' + str(train_auc))
55     print('lr test auc score: ' + str(test_auc))
56
57 # 读取数据
58 def load_data():
59     movie_df = pd.read_csv(movies_path)
60     rating_df = pd.read_csv(ratings_path)
61     df_update = convert_2_one_hot(movie_df)
62     df_final = pd.merge(rating_df, df_update, on='movieId')
63     ratings = df_final['rating']
64     df_final = df_final.drop(columns=['userId', 'movieId', 'timestamp', 'title',
65 'genres', 'rating'])
66     labels = convert_rating_2_labels(ratings)
67     trainx = df_final.values.tolist()
68     return trainx, labels
69 if __name__ == '__main__':
70     trainx, labels = load_data()
71     training_lr(trainx, labels)
```

我們看看以上程式碼都做了什麼。

第1～38行：對資料中的genres和rating進行轉換和編碼，這在前面已經講過。

第42～54行：核心步驟，建立邏輯迴歸模型，設定引數並開始訓練。我們可以注意到，這裡用的是sklearn中的邏輯迴歸模型，實際上sklearn中的邏輯迴歸模型和前幾章介紹的Keras中的邏輯迴歸模型非常相似。有興趣的讀者可以嘗試用Keras復現。

第57～66行：讀取csv檔案中的資料。前面已經展示過對應的兩個檔案的內容和格式，這裡只是在讀取後呼叫前面的函式進行一些處理，並返回訓練所需的資料及labels。

第70～71行：正式執行程式。在這個案例中，LR模型的訓練資料AUC為0.62，測試資料AUC為0.6。

5.5　多模型融合推薦模型：Wide＆Deep模型

第4章介紹瞭如何利用線性模型搭建一個基礎推薦系統。在一般情況下，具有非線性特徵轉換的廣義線性模型被廣泛用於特徵比較稀疏的大規模迴歸和分類問題。但近幾年隨著資料量和特徵量都呈現爆炸式增長，線性模型的缺點逐漸暴露出來。

線性模型通常能記住歷史資料中那些常見、高頻的資料組合，但缺點是線性模型不能發現歷史資料中未出現過的資料組合，因此線性模型需要做大量的特徵工程，根據人工經驗、業務背景，產生大量的特徵及特徵組合並將其放入線性模型中。

最近幾年，隨著資料量的保障性增長，我們越來越需要推薦系統能夠從歷史資料中發現低頻、長尾的資料組合，從而挖掘使用者的潛在興趣點。本節主要透過介紹2016年Google推出的Wide & Deep模型的原理和實現，讓讀者對推薦系統有進一步的理解。

5.5.1 探索-利用困境的問題

舊時賭場的老虎機有一個綽號叫單臂強盜，因為它即使只有一隻搖桿（胳膊），也會把你的錢拿走。多臂老虎機（或多臂強盜）的名稱就從這個綽號引申而來。假設你進入一個賭場，面對一排老虎機（多臂），而不同老虎機的期望收益和期望損失不同，你採取什麼老虎機選擇策略來保證你的總收益最高呢？這就是經典的多臂老虎機問題。

如圖5-6所示為多臂老虎機示意圖。多臂老虎機由一個盛著金幣的箱子、K個搖臂（圖中為3個搖臂）組成。玩家透過按壓搖臂來獲得金幣（回報）。玩家需要回答按壓哪個搖臂能獲得最大的回報。在這裡按壓搖臂後，獲得的回報服從不同的機率分佈。比如按壓搖臂1，獲得金幣的機率是0.8；按壓搖臂2，獲得金幣的機率是0.3；按壓搖臂3，獲得金幣的機率是0.6。那麼顯而易見，在這裡按壓搖臂1獲得的回報最大。

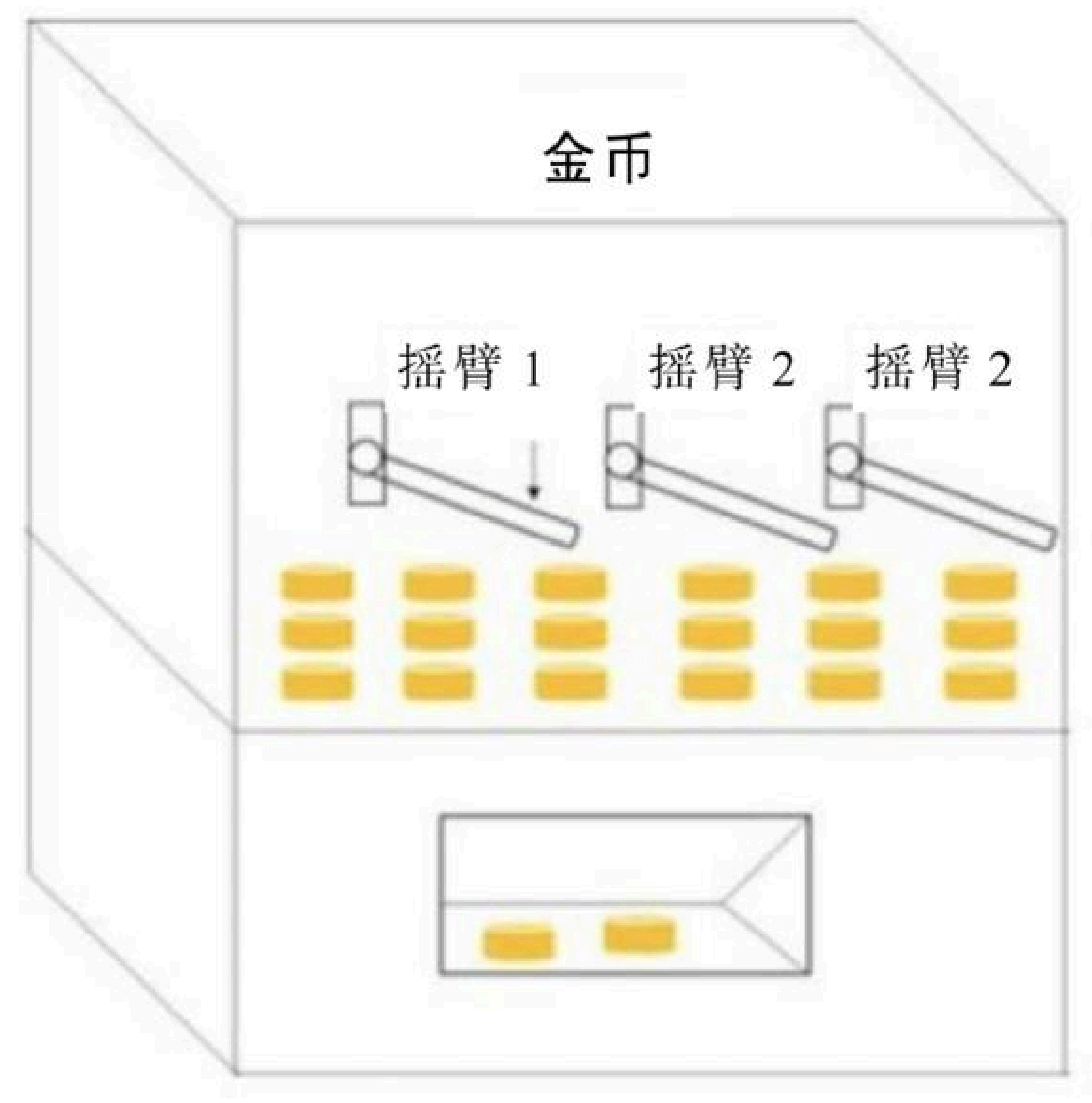


圖5-6　多臂老虎機示意圖

現在把問題複雜化，玩家並不知道按壓每個搖臂獲得回報的機率，這時候應該按壓哪個搖臂呢？

這就引出了多臂老虎機的探索-利用困境的問題：剛開始你並不知道按壓哪個搖臂給出回報的機率大，所以你很可能對每個搖臂都試了試，然後會記住按壓每個搖臂的結果。根據這些結果，你會粗略估計每個搖臂給出回報的機率。假如已經把每個搖臂都按壓了幾次，並觀察、記錄按壓每個搖臂時的回報，那麼下一步你該按壓哪個搖臂來獲得最大的回報呢？有如下兩種方案。

（1）利用（Exploitation）：如果你按壓在前幾輪中獲得回報機率最高的那個搖臂，那麼這就是你在採取「利用」策略。但是，因為回報是隨機的，所以你對每個搖臂的回報機率的估計並不準確，或許回報機率最高的那個搖臂並非當前你用幾輪資料估計的那個搖臂。

（2）探索（Exploration）：你並不去按壓在前幾輪中獲得回報的那個搖臂，而是繼續隨機按壓不同的搖臂，目的是得到每個搖臂給出回報更精確的機率估計，從而可能得到真實的最優的搖臂。

假如你按壓搖臂的次數有限，那麼為了得到最大的回報，對這兩種方案你會選哪個？

其實，折中的策略是大部分時間去利用，但同時以一定的機率去探索，這就是關於探索和利用的平衡。

5.5.2 Wide＆Deep模型

早在2016年，Google就推出了結合線性模型和深度模型分別在「利用」和「探索」上進行優勢組合得到的混合模型：Wide＆Deep模型。在文獻中，Google將線性模型稱為Wide模型，將深度模型稱為Deep模型。Wide模型的優勢是能記住歷史資料中那些常見、高頻的資料組合，然後學習到這些資料之間的權重，做一些資料篩選。比如對於電商網站，<中國人,春節,餃子>、<美國人,感恩節,火雞>、<夏天,冰激凌>都是常見模式，匹配上任何一條都有推薦價值，至於價值多少，在推薦列表中排名如何，就由Wide側的學習權重決定了。

而一個推薦系統不能只向使用者推薦其已經買過的東西或只向使用者推薦其已經閱讀過的文章，而是要替使用者發現其興趣。這就需要推薦系統從歷史資料中發現低頻、長尾的模式，發現使用者潛在的興趣點，即具備良好的「擴充套件」能力。

還是以電商為例，在歷史資料中只有<中國人,春節,餃子>、<美國人,感恩節,火雞>、<夏天,冰激凌>這樣的歷史記錄，如果推薦系統只會「記住」，那麼對於<中國人,感恩節,火雞>的組合，因為該組合和所有歷史記錄都不匹配，所以推薦系統只能打0分，並不會將該組合推薦給中國使用者。

而在Deep側，透過嵌入式詞向量（Embedding Vector）及深層互動能夠學到國籍、節日、食品等各種特徵的最優的向量表示，推薦引擎對<中國人,感恩節,火雞>這種新組合可能會打一個不低的分數（比<美國人,感恩節,火雞>打分低，比<中國人,感恩節,冰激凌>打分高），從而有機會將該組合推薦給中國使用者。簡單來說，Deep側是透過embedding將特徵向量化，將特徵的精確匹配變為特徵向量的模糊查詢，使自己具備良好的「擴充套件」能力。

5.5.3 交叉特徵

在推薦系統中大量運用的是離散類特徵，但是單個離散特徵的表達能力較弱。因此，在原論文中提出透過交叉特徵以增強離散特徵的表達能力。而圍繞如何做特徵交叉又衍生出各種演算法，在原論文中給出的是cross feature的辦法，具體做法為：假設有兩個離散特徵，即國家country和語言language， country=USA 或者 country = China，language = English 或者 language = Chinese。如果對這兩個特徵向量建立了特徵組合country x language，此特徵組合是一個4元素獨熱向量（USA and English, USA and Chinese, China and English, China and Chinese）。該組閤中的單個 1 表示國家與語言的特定連線（比如「USA = 1 and English = 1」代表你是美國人並且說英文）。然後，模型就可以瞭解到有關這種連線的特定關聯性。

但這種方法也有侷限性：如果兩個離散特徵的值很大，那麼交叉後的向量也會非常稀疏（比如特徵A、B各有100個離散值，那麼交叉後的結果就是10000個交叉特徵），這時可以透過因子分解機（Factorization Machine，FM）演算法去解決，這裡不再具體展開。

對於Wide & Deep 模型，有以下兩種思路去實現。

（1）Ensemble Training：在該模式下，Wide模型和Deep模型被單獨訓練，只有在預測時才將這兩部分的預測分數結合。

（2）Joint Training：透過同時在訓練時間中考慮Wide和Deep部分及它們的總和的權重來最佳化所有引數。

同時，對於Ensemble Training，由於訓練是分開的，所以每個單獨的模型大小通常需要更大（例如具有更多的特徵和轉換），以實現Ensemble Training工作的合理精度。相比之下，對於Joint Training訓練而言，Wide部分只需要用少量的叉積特徵變換來補充Deep部分的薄弱環節，而不是全尺寸的寬模型，因此在下面的實現中採用了Joint Training方案，其損失函式是透過計算Wide部分和Deep部分一起得到的，程式碼如下：

```
import tensorflow as tf


# 构建 Wide&Deep 模型
class WideAndDeepModel:
    def __init__(self, wide_length, deep_length, deep_last_layer_len,
softmax_label):
        # 首先定义输入部分，包括 Wide 部分、Deep 部分及标签信息 y
        self.input_wide_part = tf.placeholder(tf.float32, shape=[None,
wide_length], name='input_wide_part')
        self.input_deep_part = tf.placeholder(tf.float32, shape=[None,
deep_length], name='input_deep_part')
        self.input_y = tf.placeholder(tf.float32, shape=[None,
softmax_label], name='input_y')
        # 定义 Deep 部分的网络结构
        with tf.name_scope('deep_part'):
            w_x1 = tf.Variable(tf.random_normal([wide_length, 256],
stddev=0.03), name='w_x1')
```

```
            b_x1 = tf.Variable(tf.random_normal([256]), name='b_x1')

            w_x2 = tf.Variable(tf.random_normal([256, deep_last_layer_len],
stddev=0.03), name='w_x2')
            b_x2 = tf.Variable(tf.random_normal([deep_last_layer_len]),
name='b_x2')

            z1 = tf.add(tf.matmul(self.input_wide_part, w_x1), b_x1)
            a1 = tf.nn.relu(z1)
            self.deep_logits = tf.add(tf.matmul(a1, w_x2), b_x2)

        # 定义Wide部分的网络结构
        with tf.name_scope('wide_part'):
            weights = tf.Variable(tf.truncated_normal([deep_last_layer_len +
wide_length, softmax_label]))
            biases = tf.Variable(tf.zeros([softmax_label]))

            self.wide_and_deep = tf.concat([self.deep_logits,
self.input_wide_part], axis = 1)
            self.wide_and_deep_logits = tf.add(tf.matmul(self.wide_and_deep,
weights), biases)
            self.predictions = tf.argmax(self.wide_and_deep_logits, 1, name=
"prediction")

        # 定义损失函数
        with tf.name_scope('loss'):
            losses =
tf.nn.softmax_cross_entropy_with_logits(logits=self.wide_and_deep_logits,
labels=self.input_y)
            self.loss = tf.reduce_mean(losses)
        # 定义准确率
        with tf.name_scope("accuracy"):
            correct_predictions = tf.equal(self.predictions,
tf.argmax(self.input_y, axis=1))
            self.accuracy = tf.reduce_mean(tf.cast(correct_predictions,
tf.float32), name="accuracy")

    import pandas as pd
```

```
import numpy as np
import csv

# 读取训练数据和标签
def load_data_and_labels(path):
    data = []
    y = []
    total_q = []

    with open(path, 'r') as f:
        rdr = csv.reader(f, delimiter=',', quotechar='"')
        for row in rdr:

            emb_val = row[4].split(';')
            emb_val_f = [float(i) for i in emb_val]

            cate_emb = row[5].split(';')
            cate_emb_val_f = [float(i) for i in cate_emb]

            total_q.append(int(row[3]))
            data.append(emb_val_f + cate_emb_val_f)
            y.append(float(row[1]))
    data = np.asarray(data)
    total_q = np.asarray(total_q)
    y = np.asarray(y)

    bins = pd.qcut(y, 50, retbins=True)

    # 将标签转换为数值区间
    def convert_label_to_interval(y):
        gmv_bins = []
        for i in range(len(y)):
            interval = int(y[i] / 20000)
            if interval < 1000:
                gmv_bins.append(interval)
            elif interval >= 1000:
```

```
            gmv_bins.append(1000)

        gmv_bins = np.asarray(gmv_bins)
        return gmv_bins

    y = convert_label_to_interval(y)

    # 将标签转换为One-hot encoding
    def dense_to_one_hot(labels_dense, num_classes):
        num_labels = labels_dense.shape[0]
        index_offset = np.arange(num_labels) * num_classes
        labels_one_hot = np.zeros((num_labels, num_classes))
        labels_one_hot.flat[index_offset + labels_dense.ravel()] = 1
        return labels_one_hot

    labels_count = 1001
    labels = dense_to_one_hot(y, labels_count)
    labels = labels.astype(np.uint8)
    def dense_to_one_hot2(labels_dense, num_classes):
        num_labels = labels_dense.shape[0]
        index_offset = np.arange(num_labels) * num_classes
        labels_one_hot = np.zeros((num_labels, num_classes))
        labels_one_hot.flat[index_offset + labels_dense.ravel() - 1] = 1
        return labels_one_hot
    total_q_classes = np.unique(total_q).shape[0]
    total_q = dense_to_one_hot2(total_q, total_q_classes)


    data = np.concatenate((data, total_q), axis=1)

    return data, labels


def batch_iter(data, batch_size, num_epochs, shuffle=True):
    # 根据训练数据大小生成batch
    data = np.array(data)
    data_size = len(data)
    num_batches_per_epoch = int((len(data) - 1) / batch_size) + 1
    for epoch in range(num_epochs):
```

```
        # Shuffle the data at each epoch
        if shuffle:
            shuffle_indices = np.random.permutation(np.arange(data_size))
            shuffled_data = data[shuffle_indices]
        else:
            shuffled_data = data
        for batch_num in range(num_batches_per_epoch):
            start_index = batch_num * batch_size
            end_index = min((batch_num + 1) * batch_size, data_size)
            yield shuffled_data[start_index:end_index]


if __name__ == "__main__":
    load_data_and_labels("data/train.csv")
import tensorflow as tf
import data_helpers
import os
import datetime
import time
from WideandDeepModel import WideAndDeepModel

# 模型训练数据路径
tf.flags.DEFINE_string("train_dir", "../data/zutao2.csv", "Path of train data")
tf.flags.DEFINE_integer("wide_length", 133, "Path of train data")
tf.flags.DEFINE_integer("deep_length", 133, "Path of train data")
tf.flags.DEFINE_integer("deep_last_layer_len", 128, "Path of train data")
tf.flags.DEFINE_integer("softmax_label", 1001, "Path of train data")


# 设定模型训练参数
tf.flags.DEFINE_integer("batch_size", 32, "Batch Size")
```

```
    tf.flags.DEFINE_integer("num_epochs", 5, "Number of training epochs")
    tf.flags.DEFINE_integer("display_every", 100, "Number of iterations to
display training info.")
    tf.flags.DEFINE_float("learning_rate", 1e-3, "Which learning rate to start
with.")
    tf.flags.DEFINE_integer("num_checkpoints", 5, "Number of checkpoints to
store")
    tf.flags.DEFINE_integer("checkpoint_every", 500, "Save model after this many
steps")



    # 定义辅助参数
    tf.flags.DEFINE_boolean("allow_soft_placement", True, "Allow device soft
device placement")
    tf.flags.DEFINE_boolean("log_device_placement", False, "Log placement of ops
on devices")

    FLAGS = tf.flags.FLAGS


    def train():
       with tf.device('/cpu:0'):
    # 读取训练数据
          x, y = data_helpers.load_data_and_labels(FLAGS.train_dir)

       print('-' * 120)
       print(x.shape)
       print('-' * 120)

       with tf.Graph().as_default():
          session_conf = tf.ConfigProto(
             allow_soft_placement=FLAGS.allow_soft_placement,
             log_device_placement=FLAGS.log_device_placement)
          sess = tf.Session(config=session_conf)

          with sess.as_default():
```

```
            model = WideAndDeepModel(
                wide_length=FLAGS.wide_length,
                deep_length=FLAGS.deep_length,
                deep_last_layer_len=FLAGS.deep_last_layer_len,
                softmax_label=FLAGS.softmax_label
            )

            global_step = tf.Variable(0, name="global_step", trainable=False)
            train_op =
tf.train.AdamOptimizer(FLAGS.learning_rate).minimize(model.loss,
global_step=global_step)

            # 输出模型文件和临时 checkpoint
            timestamp = str(int(time.time()))
            out_dir = os.path.abspath(os.path.join(os.path.curdir, "runs",
timestamp))

            checkpoint_dir = os.path.abspath(os.path.join(out_dir,
"checkpoints"))
            checkpoint_prefix = os.path.join(checkpoint_dir, "model")
            if not os.path.exists(checkpoint_dir):
                os.makedirs(checkpoint_dir)

            saver = tf.train.Saver(tf.global_variables(),
max_to_keep=FLAGS.num_checkpoints)

            # 初始化所有变量
            sess.run(tf.global_variables_initializer())

            # 为每一次的新训练都生成batch、size
            batches = data_helpers.batch_iter(
                list(zip(x, y)), FLAGS.batch_size, FLAGS.num_epochs)
            for batch in batches:
                x_batch, y_batch = zip(*batch)

                feed_dict = {
```

```
                model.input_wide_part: x_batch,
                model.input_deep_part: x_batch,
                model.input_y: y_batch
            }

            _, step, loss, accuracy = sess.run(
                [train_op, global_step, model.loss, model.accuracy],
feed_dict)

            if step % FLAGS.display_every == 0:
                time_str = datetime.datetime.now().isoformat()
                print("{}: step {}, loss {:g}, acc {:g}".format(time_str,
step, loss, accuracy))

            # 保存check-point
            if step % FLAGS.checkpoint_every == 0:
                path = saver.save(sess, checkpoint_prefix,
global_step=step)
                print("Saved model checkpoint to {}\n".format(path))

        save_path = saver.save(sess, checkpoint_prefix)


def main(_):
    train()


if __name__ == "__main__":
    tf.app.run()
```

5.6 本章小結

本章快速介紹了推薦系統的基本概念，尤其是相似度計算的方式；然後迅速利用經典的電影評分案例討論兩種協同過濾的不同思路及做法，透過具體案例讓讀者基本理解協同過濾，並用程式碼體現了具體實現；之後的邏輯迴歸模型展示瞭如何利用神經網路完成推薦演算法的實現；最後介紹了經典的Wide＆Deep模型，並使用TensorFlow展示了其實現程式碼。

推薦系統是一個龐大的、不斷發展的課題，這裡只是讓讀者瞭解其基本原理。在此基礎之上，更重要的是根據實際業務的特點，有針對性地對推薦結果再次進行排查和篩選。

5.7 本章參考文獻

[1] https://grouplens.org/datasets/movielens/

[2] http//files.grouplens.org/datasets/movielens/ml-latest-small.zip

第6章　專案實戰：聊天機器人

本章將設計並實現一個簡單的自動聊天機器人模型。和其他章節類似，本章首先講解聊天機器人（Chatbot）的發展歷史；然後講解深度學習背景下的Seq2Seq模型及自然語言處理（National Language Processing，NLP）中的核心概念embedding；在此基礎上，會模仿公開的語料庫建立一個迷你訓練集來構造一個AI模型，實現聊天機器人的功能；最後引入Attention的概念和針對一個簡單問題的Attention實現。

本章內容較為深入，建議讀者仔細理解本章的各個例項，將概念瞭解清楚後再去執行程式碼。

6.1　聊天機器人的發展歷史

聊天機器人實際上很早就誕生了，可以說在計算機發展初期，研究人員就開始嘗試和計算機對話。著名的「圖靈測試」就是最好的例子。1950年，在現代計算機誕生幾年後，阿蘭·圖靈便提出了讓人類和計算機透過自然語言對話的方式來測試機器的智慧行為。從那時起，如何透過圖靈測試便成為AI研究的一個關鍵領域。下面列舉幾個有趣的具有代表性的例子。

1.ELIZA（1966年）

最早的一個人機對話程式叫作ELIZA，由MIT（麻省理工學院）的Joseph Weizenbaum教授於1966年開發。這個非常早期的人機對話程式的有趣之處在於，它並不是真正模仿正常人聊天，而是模仿心理治療且以計算機提問為主的形式進行對話，比如：

ELIZA: How do you do! Please tell me your problem.

YOU: Just for fun.

ELIZA: Does talking about this bother you ?

YOU: No. Why do you ask?

ELIZA: Are you saying no just to be negative?

YOU: Not really.

ELIZA: That is interesting. Please continue.

2.ALICE（1996年）

在很長一段時間內，人機對話程式都沒有實質上的進步，始終是基於人為規則和預定義的反饋，其中最具有代表性的是於1996年開發的ALICE。ALICE基於AIML（Artificial Intelligence Markup Language，人工智慧標記語言）能生成數萬種不同的句子，並且能在聊天過程中把使用者的反饋存入自己的系統，豐富自己所給出的結果。AIML能夠靈活地定義對話模板，如下所示：

```
<category>
  <pattern>where are you from</pattern>
  <template>I'm from China</template>
</category>

<category>
  <pattern>My name is *</pattern>
  <template>
    Hello!<think><set name = "username"> <star/></set></think>
  </template>
</category>
```

3.Eugene Goostman（2014年）

歷史上第1個透過圖靈測試的聊天機器人Eugene Goostman是在2014年誕生的。這個聊天機器人在2014年的一次圖靈測試比賽中，讓1/3的評委都相信它是一個真實的人，這在當時引起了一定的轟動。儘管有人認為Eugene Goostman只是用一些技巧性的程式碼規則糊弄了人們，談不上是真正的AI，但這至少是第1個公認透過了圖靈測試的聊天機器人。

4.基於深度學習的聊天機器人

以上實現都基於規則和人工編碼的方式，而我們的重點是討論如何實現基於深度學習的聊天機器人。

一般來說，就機器學習模型而言，我們可以把針對聊天機器人的模型分為如下兩類。

◎　Retrieval-based Model：使用預定義的答案庫，根據問題來選擇合適的預定義答案。這種方式較為簡單，既可以選擇使用硬編碼規

則實現，也可以選擇使用傳統機器學習的分類器實現。不管使用哪種方式，都不會產生全新的回答，只是在答案庫中選擇一個答案而已。

◎ Generative Model：這種方式較為複雜，並不預定義回答，而是自動生成新的答案。這種實現通常基於機器翻譯技術，但並非把一種語言翻譯為另一種語言，而是完成從問題到回答的轉換。

儘管在以上兩種模型中都可以應用深度學習技術，但目前人們更偏向於使用更靈活和更貼近真人的Generative模型。

6.2 迴圈神經網路

聊天機器人開發屬於自然語言處理領域。前面簡單介紹了傳統方式的聊天機器人實現思路。在機器學習尤其是深度學習時代，迴圈神經網路及其各種變種成為幾乎所有處理文字語言的基礎演算法，也是本節要重點講解的內容。

6.2.1 Slot Filling

我們先透過一個簡單的例子來看看迴圈神經網路是怎麼工作的（為了方便演示，這裡使用了英文示例）。

假如文字是：I』ll be at **home**on **8 pm today.**

在這個文字中，重點詞語是「home」「8 pm」「today」，因為這3個位置的詞語變化最為頻繁。

透過使用Slot Filling技術，我們可以定義兩個Slot，即location和time：

```
location: home
time: 8 pm today
```

那麼對大部分類似結構的句子，我們都可以找到Slot所對應的值：

```
I'll be in the company on 7 pm today.
location: company
time: 7 pm today

I'll be in the school for this afternoon.
location: school
time: afternoon
```

我們怎麼使用神經網路預測每個詞到底是不是location或者time呢？可以定義如圖6-1所示的簡單網路。

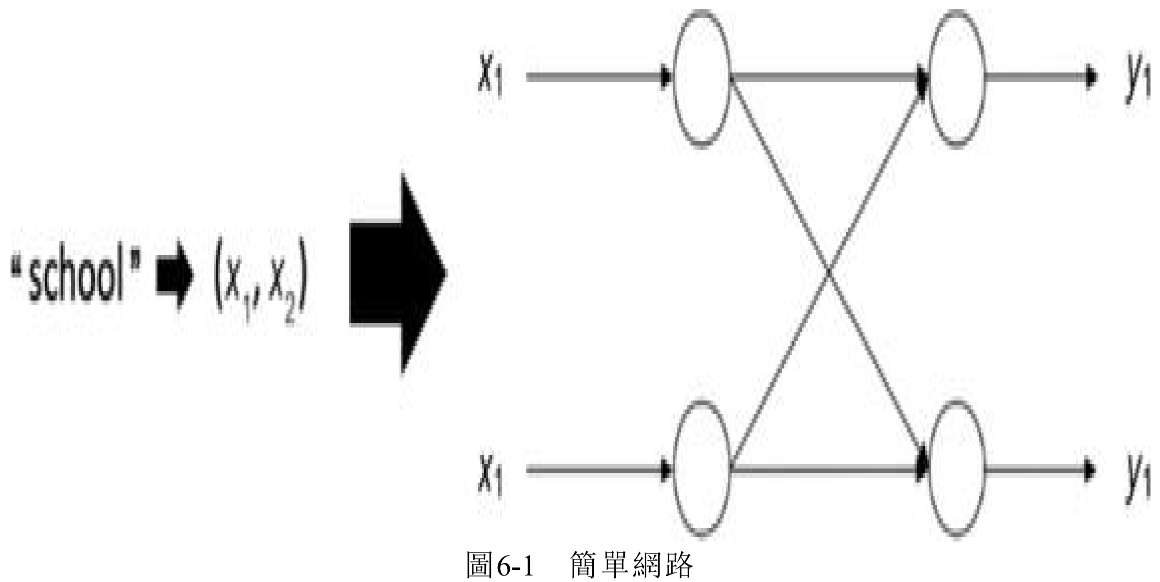


圖6-1　簡單網路

在圖6-1中構建了一個只有一層隱藏層的神經網路，將句子中的每個單詞都作為vector$[x_1,x_2]$輸入，並輸出$[y_1,y_2]$，其中，y_1、y_2代表屬於兩個不同Slot的機率。換句話說，在句子「I』ll be at home tomorrow」中，假設神經網路被定義為f，而$y1$和$y2$分別代表location和time的Slot，則我們將得到：

```
f("I'll") = [0, 0]
f("be") = [0, 0]
f("at") = [0, 0]
f("home") = [1, 0]
f("tomorrow") = [0, 1]
```

那麼，我們是怎麼把一個單詞轉換為[x_1,x_2]這樣的陣列的呢？記得在第3章中提到過的Embedding層嗎？它其實就是在做類似的工作。6.2.2節會講解在NLP中是如何處理不同單詞的。

6.2.2 NLP中的單詞處理

大致上，我們可以將NLP中的單詞處理方式分為以下三種：

◎ One-hot encoding；

◎ n-gram；

◎ word2vec。

其中，One-hot encoding最為簡單。它把詞典中的所有單詞都取出來去重，可以得到一個陣列，比如['dog','cat','animals']。那麼：

```
'dog'=  [1, 0, 0, 0]
'cat' =     [0, 1, 0, 0]
'animals' = [0, 0, 1, 0]
others =    [0, 0, 0, 1]   // 这里指不属于前 3 类的任何词语
```

儘管One-hot encoding的原理很簡單，但是其資料過於稀疏，我們可以透過n-gram方式處理這樣一個句子「dog and cat are animals」：

```
"dog and cat" => [1, 1, 0, 1]
"and cat are" => [0, 1, 0, 1]
"cat are animals" => [0, 1, 1, 1]
```

上面介紹了把文字詞語vector化的兩種方式，我們通常把這種處理方式稱為word embedding。然而以上兩種並不是目前業界流行的方法。目前真正的業界標準是Google在2013年提出的word2vec。

第3章在介紹Keras時提到過softmax函式，我們在理解softmax函式的概念後，就可以來理解word2vec的概念。實際上word2vec包含兩種模型：CBOW模型（Continuous Bag of Words Model，連續詞袋模型）和Skip Gram模型（跳字模型）。

1.CBOW模型

首先，BOW模型（Bag of Words Model，詞袋模型）和前面的One-hot encoding及n-gram類似，都是一種把句子vector化的方法。BOW模型的特點如下：

◎　只包括已知詞語的集合，不關心具體位置；

◎　計算詞語出現的次數或頻率。

從某種意義上來說，我們可以把BOW模型看作句子在頻域上的表現形式。比如，對於上面的例子「dog and cat are animals」，我們就可以定義：

```
"dog and cat are animals" => [1, 1, 1, 2]
```

CBOW模型所要做的，是把一個詞的上下文作為輸入，預測該詞上下文的下一個詞的內容。比如，在前面的例子中如果輸入「cat and dog」，則應該預測下一個詞是「are」。

我們先看一個簡單的CBOW模型，如圖6-2所示，這個模型只接收一個單詞作為輸入。

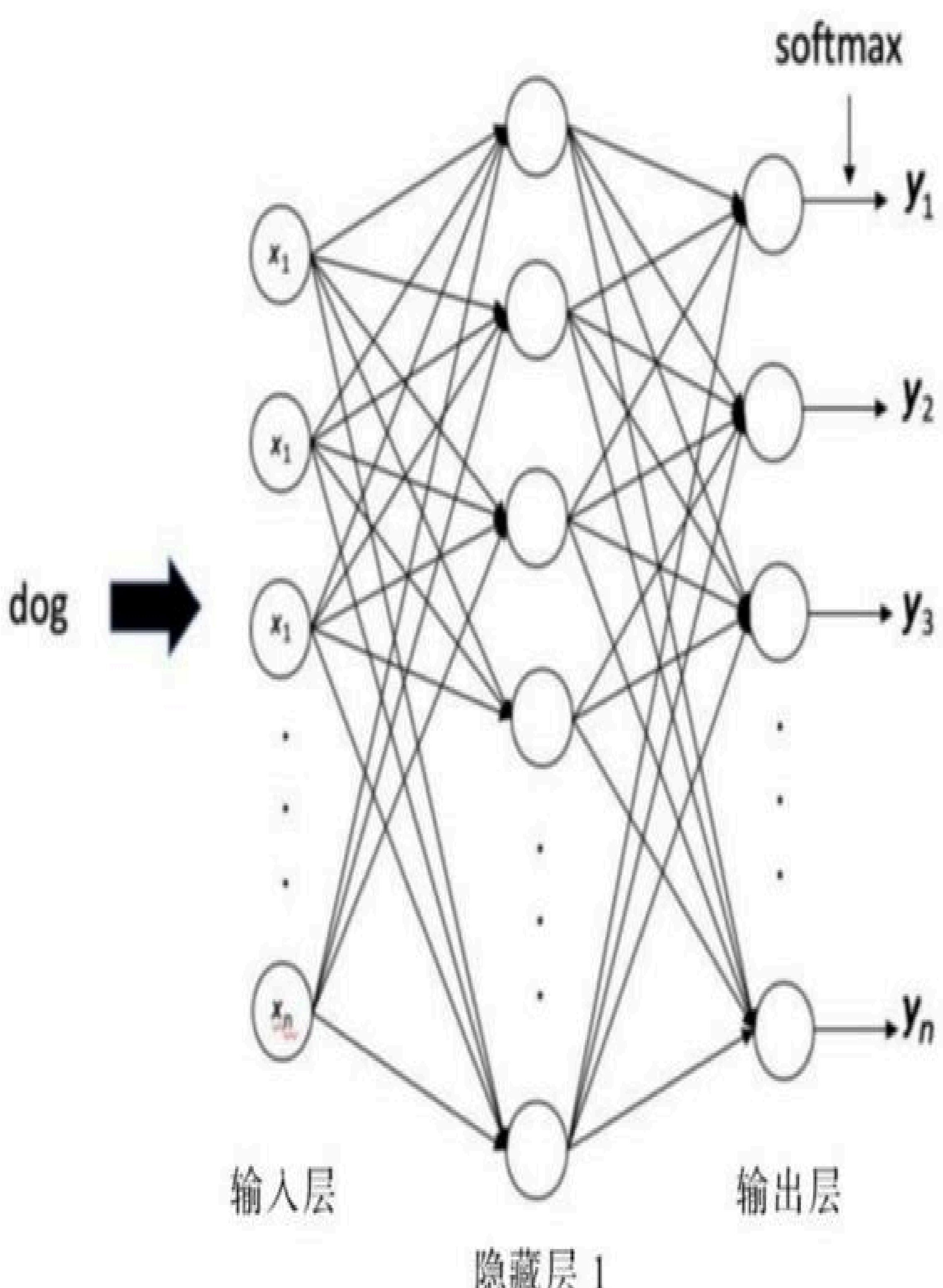


圖6-2　一個簡單的CBOW模型

圖6-2中展示了一個簡單的CBOW模型，它接收一個One-hot encoding的vector輸入，透過一個隱藏層輸出一個同樣長度的vector，再對其應用softmax函式後得到最終的結果。

在實際應用中，我們不會只用一個單詞來預測，而是用多個相鄰詞語來預測。因此，我們可以將一個更接近真實應用的模型設計為如圖6-3所示。

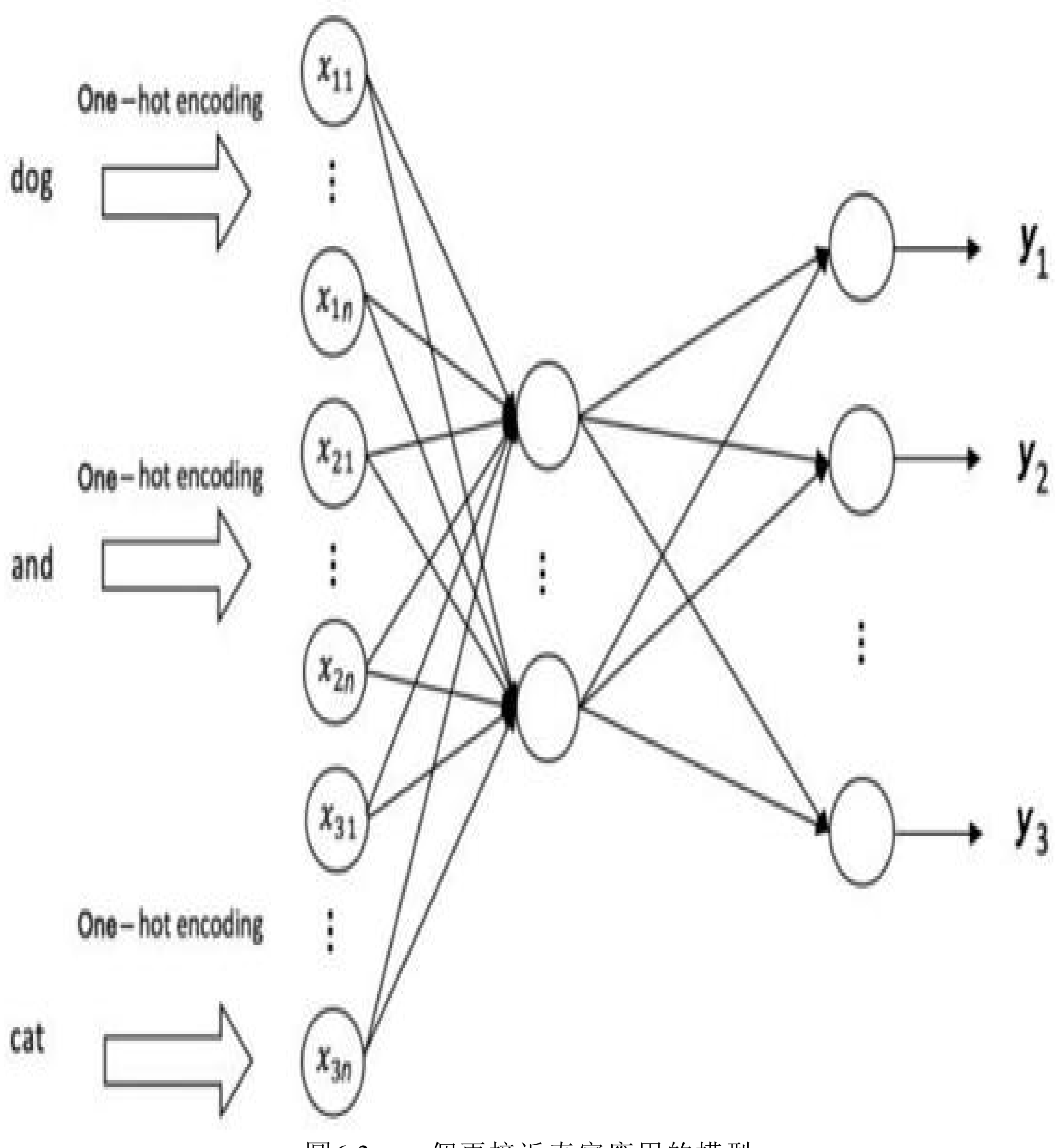


圖6-3　一個更接近真實應用的模型

2.Skip Gram模型

我們可以將Skip Gram模型看作CBOW模型的反轉：CBOW模型是根據上下文的相關詞語推匯出下一個可能的詞，而Skip Gram模型是根據一個給出的詞語推匯出其相鄰位置的詞語的可能性分佈（Probability Distribution）。我們先來看看如圖6-4所示的Skip Gram模型。

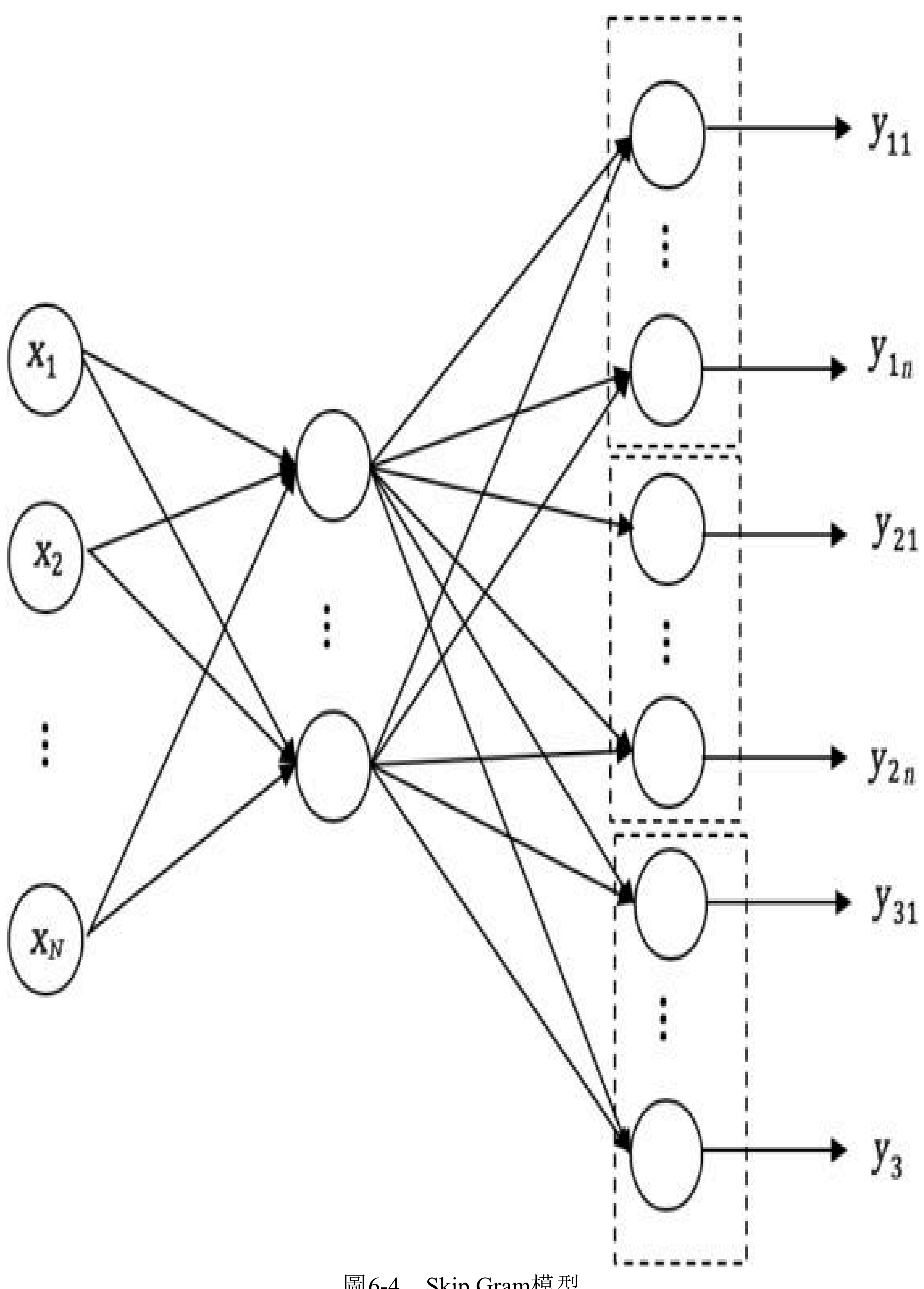


圖6-4　Skip Gram模型

圖6-4幾乎就是圖6-3中模型的反轉，我們輸入一個詞的One-hot encoding向量，透過softmax函式輸出了3個相鄰位置的詞語的可能性分佈。這裡對每個位置的輸出都是一個對應One-hot encoding vector的向量，但該向量中的每個元素不再是One-hot encoding中的整數，而是透過softmax函式計算後的對應分佈機率，例如：

$$y = \begin{bmatrix} [0.2, 0.5, 0.3] \\ [0.6, 0.2, 0.2] \\ [0.1, 0.01, 0.89] \end{bmatrix}$$

而對應的詞典是["dog", "cat", "animals"]，這就意味著第1個詞有50%的可能是cat，第2個詞有60%的可能是dog，第3個詞有89%的可能是animals。

6.2.3　迴圈神經網路簡介

回到先前的例子：

I』ll be at **home** on **8 pm today.**

我們找到了關鍵詞「home」和「8 pm today」，但是如果僅僅靠這兩個詞，我們很可能會被誤導。比如句子變成「I』ll leave home on 8 pm today」，這時意思就全變了。當然，我們也可以對「I』ll」之後的動詞進行另一個Slot處理，但對於這個位置的詞就存在太多的不確定性了，而且對於更復雜的句子更難建立Slot。

因此，我們注意到現在定義的location和time這兩個Slot，其具體意義要依賴前文的相關資訊來理解。換句話說，我們需要一個具有「記憶功能」的神經網路，能夠把之前處理過的資料以某種形式體現在後續的計算上，這也就是迴圈神經網路（後統稱RNN，見圖6-5）被提出的初衷。

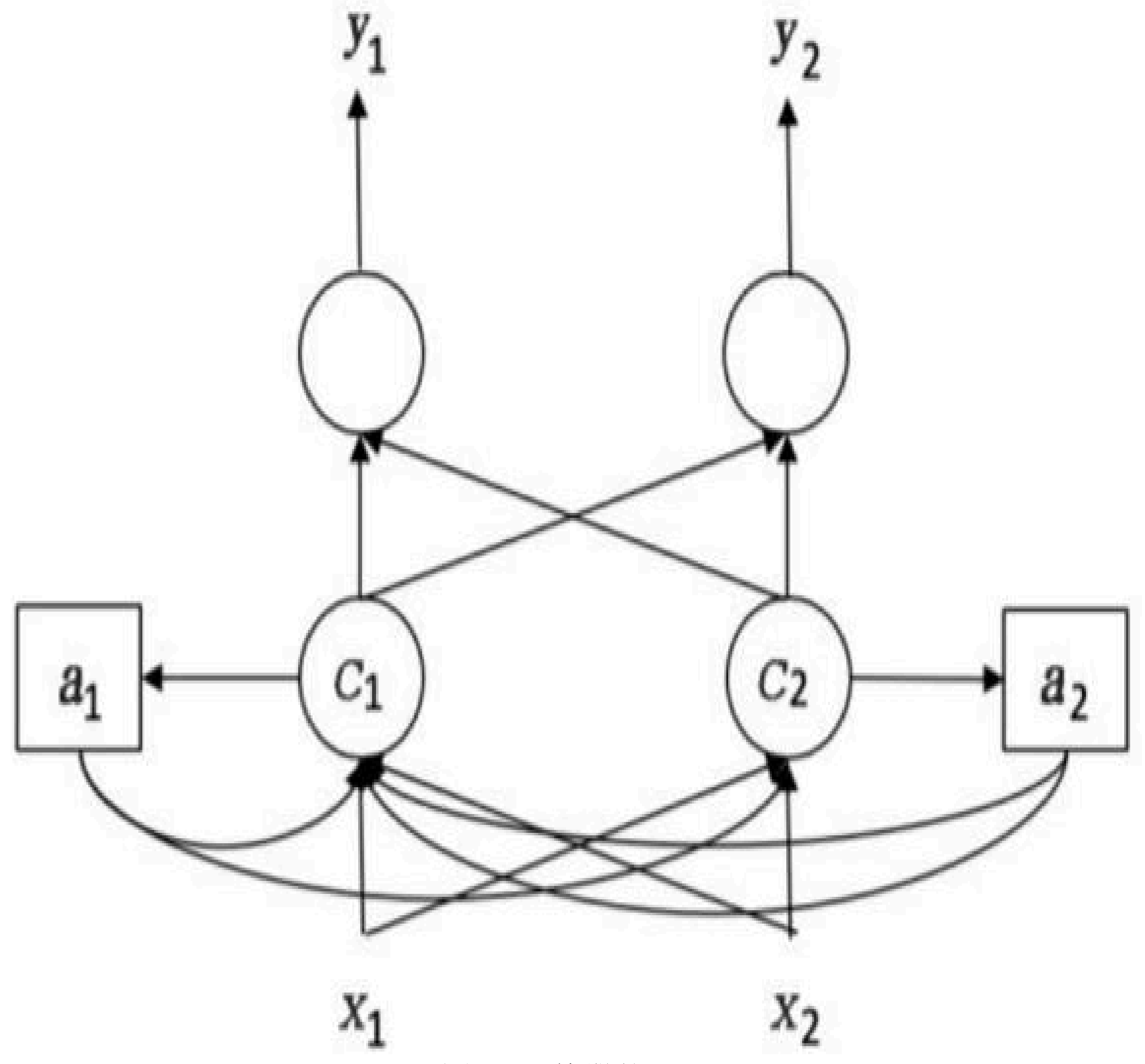


圖6- 5　簡單的RNN

當我們用圖6-5所示的簡單RNN處理句子「I』ll arrive home tomorrow」時，每次的輸出都被作為下一次的輸入，如圖6-6所示。

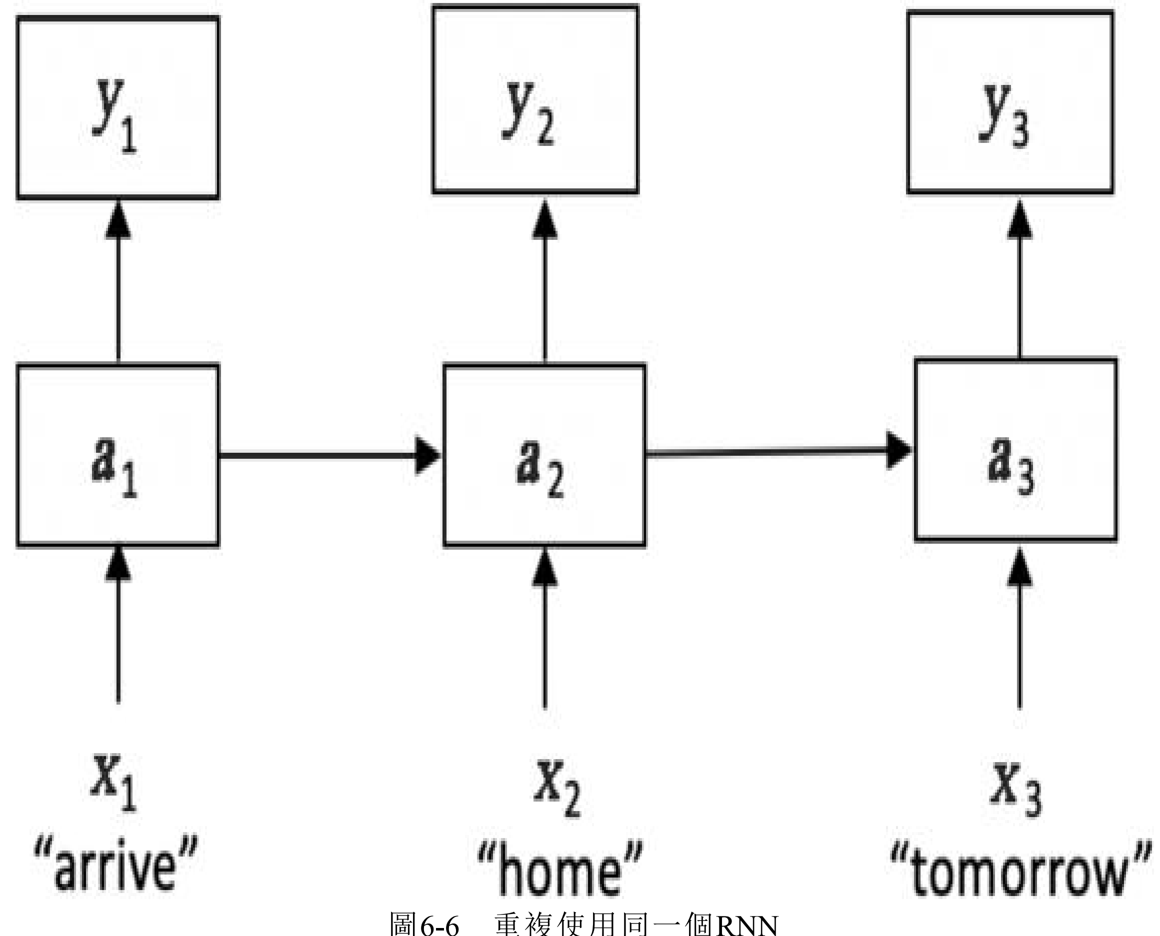
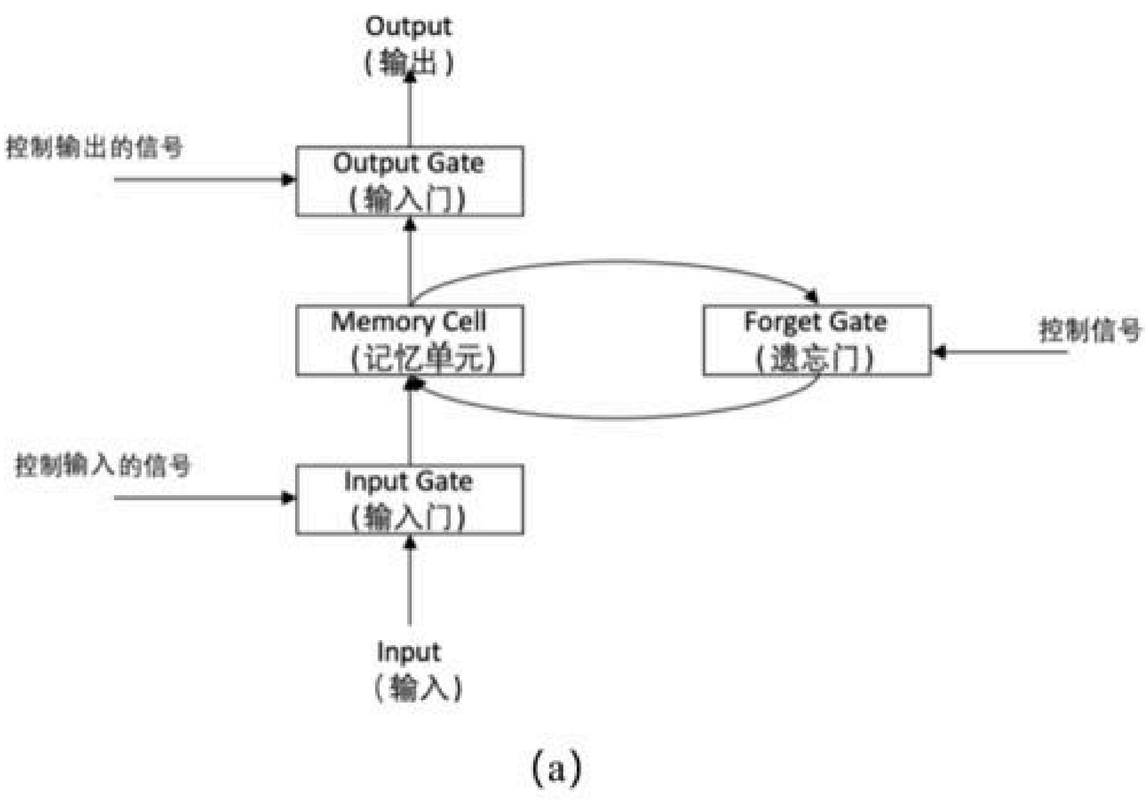


圖6-6　重複使用同一個RNN

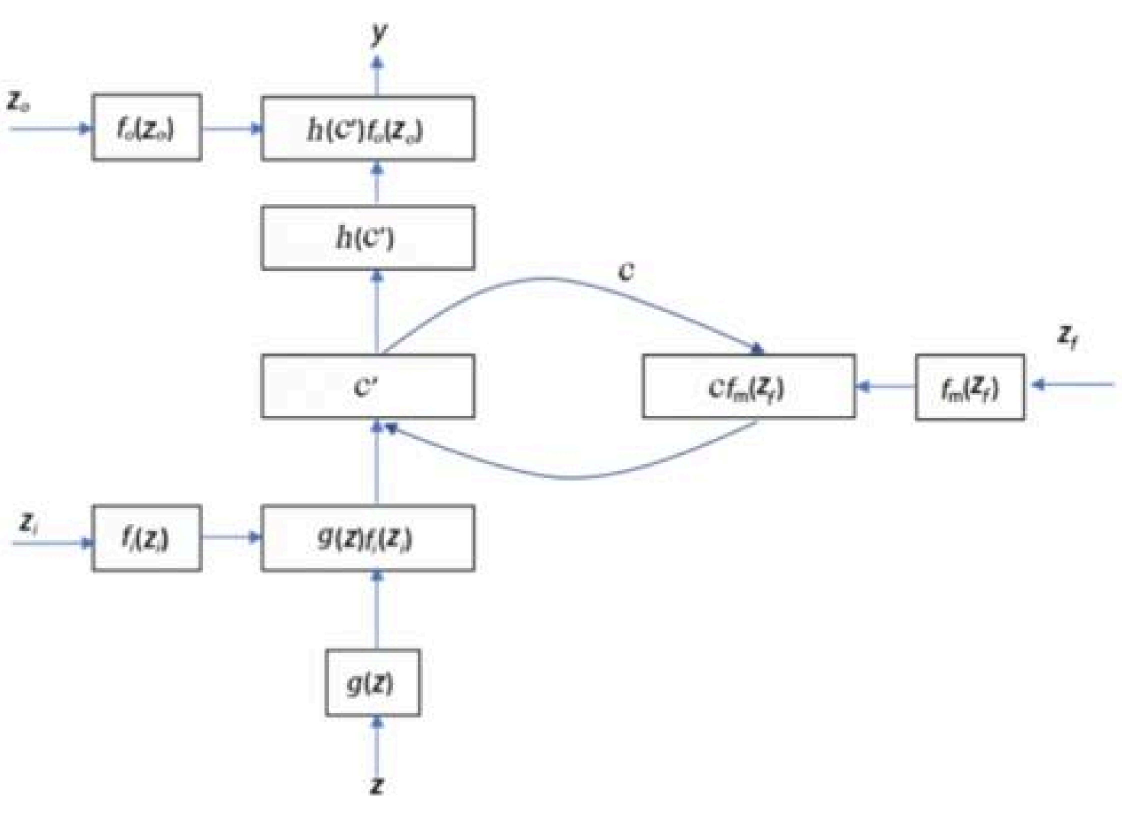
在圖6-5和圖6-6中，為了方便理解，我們只描述了一個單層的簡單網路。當然，我們也可以把圖6-5中的C_1、C_2或者圖6-6中的a_1、a_2根據需要改為多層網路，在此不再贅述。因為在實際使用中，我們更多地使用下面介紹的LSTM網路。

6.2.4　LSTM網路簡介

LSTM（Long Short-Term Memory，長短期記憶）網路顧名思義，指它對資料具有短時間的記憶功能。我們先來看圖6-7中的一個簡單的LSTM網路的結構。



(a)



(b)

圖6-7　一個簡單的LSTM網路的結構

圖6-7展示了LSTM網路的基本結構，首先，我們有以下4個輸入。

◎　Z：當前要處理的資料。

◎　Z_i：控制輸入門的訊號，通常決定是否產生輸出。

◎　Z_o：控制輸出門的訊號，通常決定是否接收輸入。

◎　Z_f：控制前一組資料留在記憶單元中的歷史資料的處理。

我們可以將圖6-7(b)中的C和C』看作記憶單元中的值，其中，C為處理前的舊數值，C』為更新後的新數值。在我們目前的討論中，可以把圖6-7中的$g(Z)$、$h(C$』$)$、$f_i(Z_i)$、$f_o(Z_o)$、$f_m(Z_f)$等啟用函式當作sigmoid函式對待。在瞭解這些定義後，我們透過一個例項來看看LSTM具體是怎麼執行的。

我們定義輸入為$[x_1,x_2, x_3]$，輸出為y，其中：

```
if x2 = 0, C' = C+x1
if x2 = -1, C' = 0
if x3 = 1, y = C'
```

這裡不講解具體的模型訓練細節（仍然基於梯度下降），而是假設已經有訓練好的引數，看看上述LSTM網路是如何工作的。這裡假定訓練好的模型為

```
g(z) = [w11, w12, w13]T ⊗ [x1, x2, x3] + b = 1*x1 + 0*x2 + 0*x3 +1
fi(z) = sigmoid( [w21, w22, w23] T ⊗ [x1, x2, x3] + b = 0*x1 + 100*x2 + 0*x3 -10)
fo(z) = sigmoid( [w31, w32, w33] T ⊗ [x1, x2, x3] + b = 0*x1 + 0*x2 + 100*x3 -10)
fm(z) = sigmoid( [w41, w42, w43] T ⊗ [x1, x2, x3] + b = 0*x1 + 100*x2 + 0*x3 +1)
```

我們實際上會得到如圖6-8所示的LSTM網路。

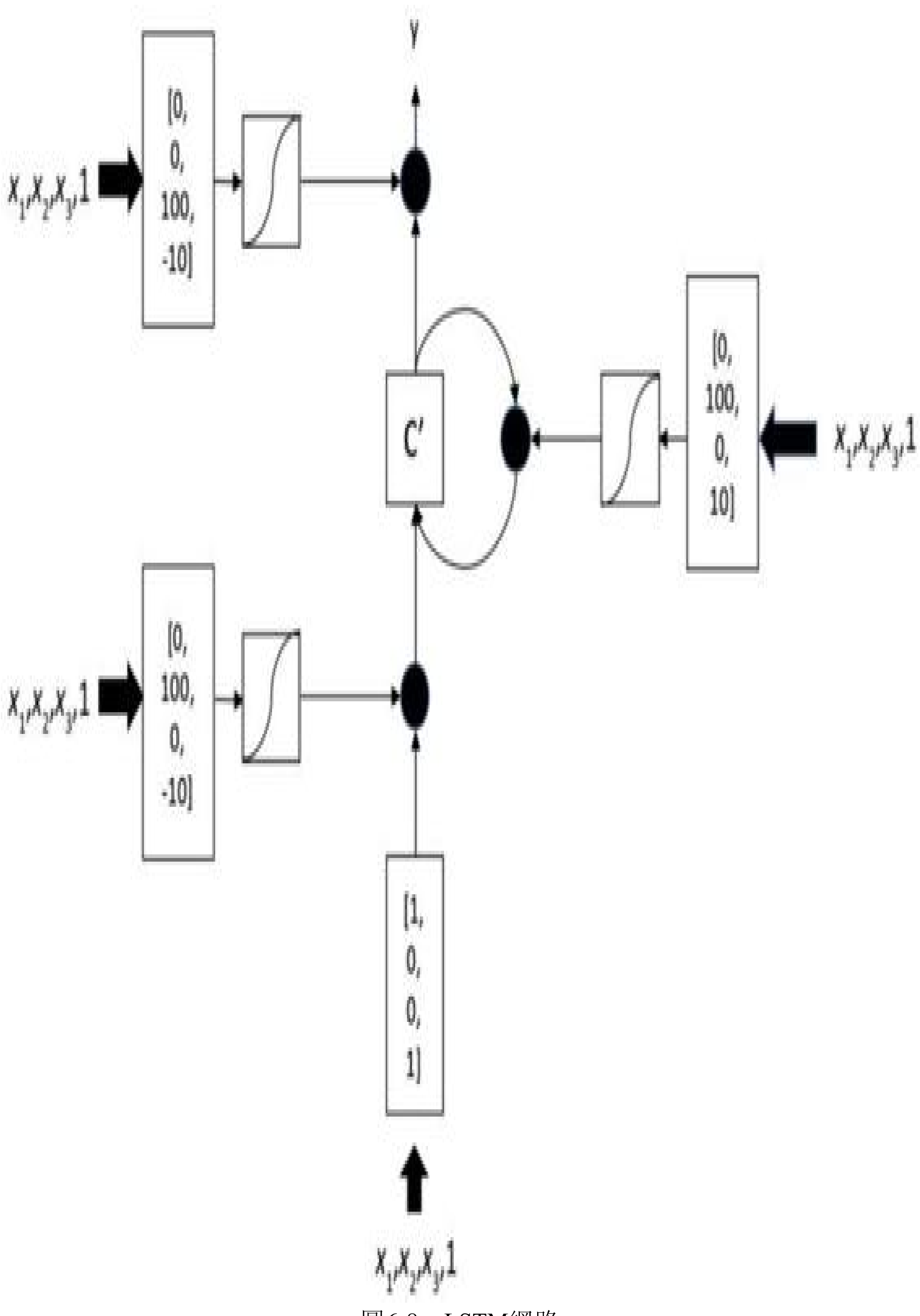


圖6-8　LSTM網路

我們用4組資料來驗證，這4組資料是[1,0,0]、[4, 2, 0]、[1, 0, 1]和[2, -1, 1]。

按照前面給出的ground truth定義，我們可以知道輸出為0, 0, 4, 0。

那麼，把同樣的4組資料輸入圖6-8所示的網路中，根據圖6-7(b)所示的流程進行計算，可得到各變數的值如表6-1所示。

表6-1　各變數的值

输入	f_i	f_m	C	f_o	Y（输出）
1,0,0	0	1	0	0	0
4,2,0	1	1	4	0	0
1,0,1	0	1	4	1	4
2,-1,1	0	0	0	1	0

可以看到，該網路的輸出和ground truth是相符的。

我們在6.2節學習了RNN的重要概念和工作原理。從6.3節開始，我們將學習真實的用於機器翻譯的神經網路Seq2Seq演算法，直到開發出一個聊天機器人。

6.3　Seq2Seq原理介紹及實現

Seq2Seq（Sequence to Sequence）是一個常見的便於理解的基於LSTM網路的模型，在2015年之前被廣泛應用於文字翻譯、聊天機器人等領域。儘管隨著技術的發展，我們有更多、更好的模型去完成相關工作，但Seq2Seq仍不失為一種快速、有效的基於神經網路的NLP處理模型，更重要的是其中涉及的多個概念都是理解後面更複雜模型的基礎。

6.3.1 Seq2Seq原理介紹

Seq2Seq的演算法就是把一種字元序列（Character Sequence）轉換為另一種字元序列，例如：

◎ 「I want to go home」→ seq2seq model →「我想回家」（用於翻譯）；

◎ 「Where are you from?」→ seq2seq model →「I』m from China」（用於對話）。

在前面討論的RNN、LSTM模型中，我們希望透過input→RNN/LSTM→output的簡單流程實現翻譯或者人機對話的轉換。然而在實際應用中，直接用RNN或LSTM網路進行對不同長度的文字翻譯或對話是難以成功的。

在由k.cho等人於2014年發表的論文*Learning phrase representations using RNN encoder-decoder for statistical machine translation*[1]中，提出了使用疊加的RNN作為encoder或decoder來實現機器翻譯的做法，它把輸入序列透過第1個RNN（encoder）轉化為一個固定長度的向量vector（在本章參考文獻[1]中體現為讀完序列後的隱藏狀態h，參考圖6-7(b)圖），然後用另一個RNN（decoder）將其轉換為目標序列。而在另一篇論文*Sequence to Sequence Learning with Neural Networks*[2]中，Google把在本章參考文獻[1]中所用的簡單RNN用不同的兩個Deep LSTM網路進行了替換，降低了訓練的難度和預測的準確性，其大致思路如圖6-9所示。

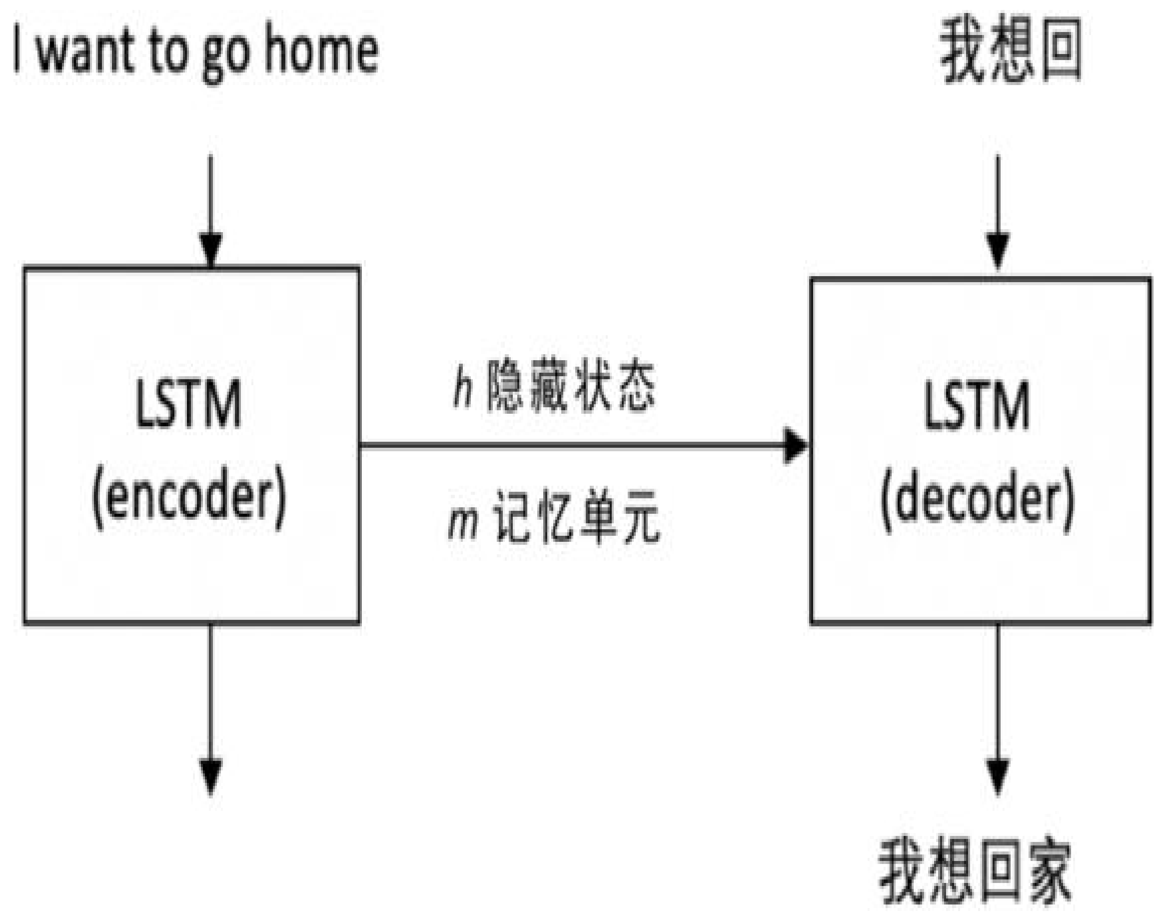


圖6-9　encoder-decoder架構圖

從圖6-9可以看到，encoder並沒有把LSTM網路的輸出匯入decoder中，而是把LSTM網路的隱藏狀態和記憶單元（即圖6-7(b)圖中的*h*和*C*）作為decoder的輸入。其中，encoder輸出的隱藏狀態和記憶單元可以被視為某種上下文（Context）或條件（Condition），來輔助decoder的LSTM網路預測輸出。我們不妨假設並沒有圖6-9左邊的LSTM網路（encoder），也沒有encoder輸出，只有右邊單獨的一個LSTM網路來預測下一個字元的輸出。為了提升decoder的訓練速度和預測效果，我們就在左邊加上利用原文的完整句子訓練的encoder，將它的狀態值作為decoder的初始值，起到更好的訓練效果。

6.3.2　用Keras實現Seq2Seq演算法

實際上在Keras中實現Seq2Seq演算法非常容易，我們只需定義兩個LSTM網路即可，重點在於如何把這兩個網路透過輸入及輸出關聯

起來。

Keras團隊已經把Seq2Seq的實現開源在GitHub[3]上，這裡會在其基礎上做較多改變，以體現在具體開發時如何處理。

首先，我們需要下載訓練資料，在manythings網站的anki頁面[3]可以看到各種語言，包括英語的對照詞庫，我們可以下載fra-en.zip並將其解壓為我們的訓練集。實際上，Seq2Seq演算法並不在意資料集是什麼樣的，因為資料集只是提供一個輸入串到另一個輸入串的參考關係而已。我們可以參考其格式打造一個很小的只有十多行內容的聊天迷你資料集，其中的每一行都包括兩句對話，中間用「\t」隔開，我們將其中的內容存為chat.txt檔案：

Hello.   Hello !

Are you ok?   Yes !

Good morning!   Morning !

Are you hungry?   Yes !

Did you have lunch?   No, I haven't !

Are you at home?   No, I'm not !

Help!   What's happening?

Are you coming with me?   I'd like to.

Cheers!   Cheers!

Have a nice day!   you too!

Have a nice weekend!   you too!

Enjoy the food!   Thanks.

Thank you so much!   No problem.

What's going on?   Nothing.

How old are you?   I'm 30 years old.

See you soon. Bye!

How do you feel?   It's great!

Good night!   Have a nice dream!

This is my father.   Nice to meet you.

The show is over.   Let's go back home!

現在開始編寫我們的程式碼。我們的程式碼分為trainer和inferencer兩部分，其中，trainer負責訓練模型和儲存相關內容到本地檔案中；inferencer負責讀取模型檔案、重建模型及前向運算。整個實現流程如圖6-10所示。

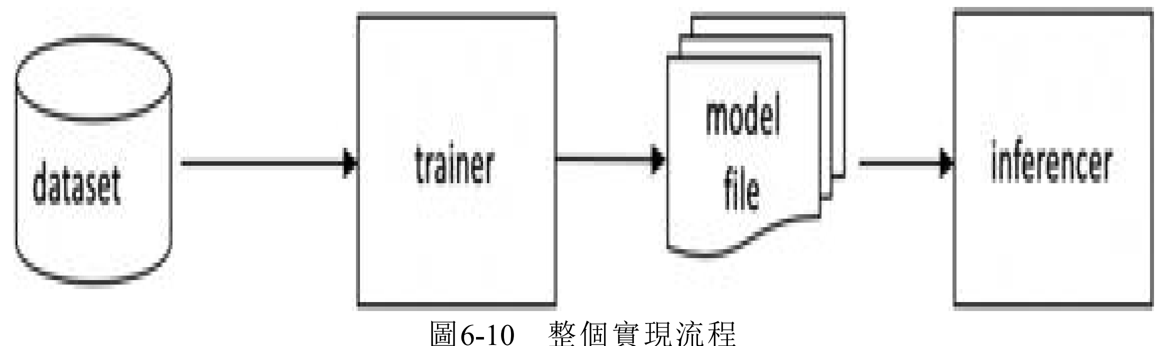


圖6-10　整個實現流程

首先是trainer的實現。我們建立一個新的Python檔案seq2seq_trainer.py，引入相關依賴庫並定義超參（Hyper Parameters），其中，latent_dim代表LSTM的輸出向量的空間大小，因為有256個字元，因此被定義為256：

```
1   import tensorflow as tf
2   import numpy as np
3
4   from tensorflow.keras.models import Model
5   from tensorflow.keras.layers import Input, LSTM, Dense
6   from tensorflow.keras.utils import plot_model
7
8   import json
9
10  batch_size = 64
11  epochs = 100
12  latent_dim = 256
13  max_num_samples = 20
```

然後處理之前定義的迷你對話資料庫chat.txt，我們緊跟著用一個函式init_dataset來處理資料：

```
15 def init_dataset(num_samples):
16     data_path = './chat.txt'
17 
18     # 对数据进行向量化处理
19     input_texts = []
20     target_texts = []
21     input_characters = set()
22     target_characters = set()
23     with open(data_path, 'r', encoding='utf-8') as f:
24         lines = f.read().split('\n')
25     for line in lines[: min(num_samples, len(lines) - 1)]:
26         input_text, target_text = line.split('\t')
27         # 我们用"\t"作为字符串的起始标志
28         # 将"\n"符号作为目标句子结束标志
29         target_text = '\t' + target_text + '\n'
30         input_texts.append(input_text)
31         target_texts.append(target_text)
32         for char in input_text:
33             if char not in input_characters:
34                 input_characters.add(char)
35         for char in target_text:
36             if char not in target_characters:
37                 target_characters.add(char)
38 
```

注意，資料的格式是{input_text}\t{target_text}，因此在以上程式碼中，我們首先獲取每一行，然後將每一行用line.split分割成input_text和target_text（第26行），最後用set()統計input_text和target_text()的樣本空間大小（在這個例子中，我們是按照字元進行處理的）。換句話說，我們在下面的程式碼中建立字元集索引，準備將每個句子向量化：

```
39 input_characters = sorted(list(input_characters))
40 target_characters = sorted(list(target_characters))
41 num_encoder_tokens = len(input_characters)
42 num_decoder_tokens = len(target_characters)
43 max_encoder_seq_length = max([len(txt) for txt in input_texts])
44 max_decoder_seq_length = max([len(txt) for txt in target_texts])
45
46 input_token_index = dict(
47     [(char, i) for i, char in enumerate(input_characters)])
48 target_token_index = dict(
49     [(char, i) for i, char in enumerate(target_characters)])
50
51 encoder_input_data = np.zeros(
52     (len(input_texts), max_encoder_seq_length, num_encoder_tokens),
53     dtype='float32')
54 decoder_input_data = np.zeros(
55     (len(input_texts), max_decoder_seq_length, num_decoder_tokens),
56     dtype='float32')
57 decoder_target_data = np.zeros(
58     (len(input_texts), max_decoder_seq_length, num_decoder_tokens),
59     dtype='float32')
```

其中，比較重要的是第46～49行，這幾行把輸入字元和輸出字元各自做了索引。在第51～59行建立瞭如下3組向量。

◎ encoder_input_data：encoder的輸入句子向量。

◎ decoder_input_data：decoder的輸入句子向量。

◎ decoder_target_data：decoder的目標句子向量。

和如圖6-9所示的encoder-decoder架構圖相比，我們可以看到上面的3組向量正好是圖6-9中的輸入和輸出（有兩個輸入和1個輸出）。所以下面的程式碼是利用前面建立的索引表，對input_texts和target_texts中的每組資料都建立對應的向量，最後把相關資料返回：

```
60      for i, (input_text, target_text) in enumerate(zip(input_texts,
61  target_texts)):
62          for t, char in enumerate(input_text):
63              encoder_input_data[i, t, input_token_index[char]] = 1.
64          for t, char in enumerate(target_text):
65              # decoder_target_data 比 decoder_input_data 要提前一步
66              decoder_input_data[i, t, target_token_index[char]] = 1.
67              if t > 0:
68                  # decoder_target_data 要提前一步，而且不会包括起始字符
69                  decoder_target_data[i, t - 1, target_token_index[char]] = 1.
70
71
72      return {
73          'encoder_input_data': encoder_input_data,
74          'decoder_input_data': decoder_input_data,
75          'decoder_target_data': decoder_target_data,
76          'num_encoder_tokens': num_encoder_tokens,
77          'num_decoder_tokens': num_decoder_tokens,
78          'input_token_index': input_token_index,
79          'target_token_index': target_token_index,
80          'max_encoder_seq_length': max_encoder_seq_length,
81          'max_decoder_seq_length': max_decoder_seq_length,
82          }
```

在前面的資料處理都完成後，我們就可以開始建立Seq2Seq模型。建立Seq2Seq模型的程式碼如下，主要利用了Keras自帶的LSTM模型，這樣我們的工作會簡單很多：

```
87  dataset = init_dataset(max_num_samples)
88  num_encoder_tokens = dataset['num_encoder_tokens']
89  num_decoder_tokens = dataset['num_decoder_tokens']
90  encoder_input_data = dataset['encoder_input_data']
91  decoder_input_data = dataset['decoder_input_data']
92  decoder_target_data = dataset['decoder_target_data']
93
94  # 定义输入数据并处理
95  encoder_inputs = Input(shape=(None, num_encoder_tokens))
96  encoder = LSTM(latent_dim, return_state=True)
97  encoder_outputs, state_h, state_c = encoder(encoder_inputs)
98
99  # 去掉 'encoder_outputs'，只保留状态
100 encoder_states = [state_h, state_c]
101 # 设置decoder， 使用'encoder_states' 作为初始状态
102
103 decoder_inputs = Input(shape=(None, num_decoder_tokens))
104 # 设置decoder返回完整输出序列,
105 # 与此同时，decoder也需要把内部状态返回
106 # 我们并不在训练时使用内部状态，但在做预测时会使用
107 decoder_lstm = LSTM(latent_dim, return_sequences=True, return_state=True)
108 decoder_outputs, _, _ = decoder_lstm(decoder_inputs,
109 initial_state=encoder_states)
110 decoder_dense = Dense(num_decoder_tokens, activation='softmax')
111 decoder_outputs = decoder_dense(decoder_outputs)
112
113 model = Model([encoder_inputs, decoder_inputs], decoder_outputs)
114
115 model.compile(optimizer='rmsprop', loss='categorical_crossentropy')
116 model.fit([encoder_input_data, decoder_input_data], decoder_target_data,
117         batch_size=batch_size, epochs=epochs, validation_split=0.2)
```

我們看看上面的程式碼都做了什麼。

第87～92行：引入相關資料和引數。

第94～100行：建立encoder網路。我們引入一個Input層作為LSTM網路的輸入，但是隻定義LSTM網路返回state，而不需要返回sequence（序列最終結果），如圖6-9中的網路所設計的。然後我們在第97～100行獲得LSTM encoder網路中的隱藏狀態和記憶單元的值。

第103～111行：建立decoder網路。我們同樣定義一個LSTM網路，但這次設定需要返回最終結果（return_sequences=True），最後把最終返回的結果（向量資料）輸入一個全連線網路中，並使用softmax作為啟用函式。

第113～117行：拼接網路、完成模型並進行訓練。注意，我們在搭建從encoder到decoder的網路時，encoder的狀態輸出（隱藏狀態和記憶單元）會被作為decoder的初始值，這意味著encoder在每次處理完一個輸入串後，它的內部狀態就會被用在decoder上進行第1個字元的輸出。這也意味著encoder和decoder的LSTM網路必須保持同樣的單元個數（在這個例子中是256，也就是latent_dim）。

然後，我們可以使用如下程式碼繪製並儲存模型引數：

```
123 plot_model(model, to_file='s2s_1_model.png', show_shapes=True)
124
125 model.save('s2s_1.h5')
126
127 with open('s2s_1.json', 'w', encoding='utf8') as f:
128     f.write(model.to_json(indent=4))
129
130 config = {
131     "latent_dim": 256,
132     "max_num_samples": max_num_samples,
133     'num_encoder_tokens': num_encoder_tokens,
134     'num_decoder_tokens': num_decoder_tokens,
135     'input_token_index': dataset['input_token_index'],
136     'target_token_index': dataset['target_token_index'],
137     'max_encoder_seq_length': dataset['max_encoder_seq_length'],
138     'max_decoder_seq_length': dataset['max_decoder_seq_length'],
139 }
140
141 with open('s2s_1_config.json', 'w', encoding='utf8') as f:
142     f.write(json.dumps(config))
143
```

第 123 ～ 139 行：儲存相關的引數。注意，在第 123 行中用plot_model函式繪製瞭如圖6-11所示的模型架構。

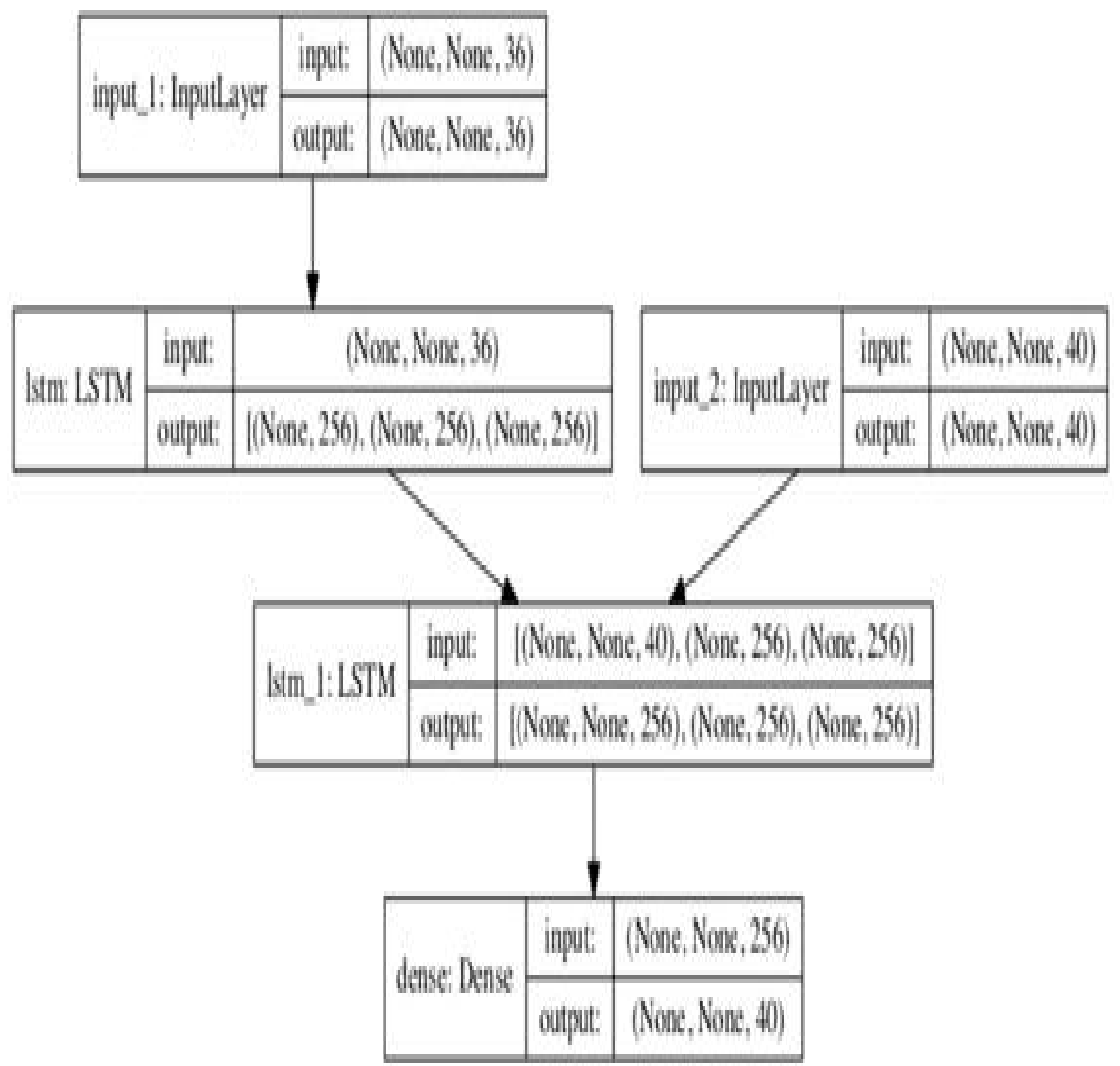


圖6-11　用plot_model函式繪製的模型架構

與在第127～128行中用JSON格式儲存的模型描述檔案相比，可以看到二者是一致的。但用程式碼中描述的方式儲存模型，會對在後面的程式碼中讀入配置和重建網路有很大的便利，在以下inferencer的實現中會詳細講述。

inferencer的實現有如下兩個關鍵問題：

◎　如何重建網路？

◎　如何對兩個連線的LSTM網路實現inference？

我們在trainer的程式碼中最後儲存瞭如下3個檔案。

◎　模型權重（weights）檔案：s2s_1.h5。

◎　模型網路描述：s2s_1.json。

◎　引數配置：s2s_1_config.json。

其中必不可少的是s2s_1.h5檔案，這是trainer訓練出來的網路權重，是重建模型的根本。s2s_1.json實際上並不會在inferencer裡被程式碼讀入，但是對於編寫inferencer時重建網路有重要的參考作用。s2s_1_config.json則僅僅儲存了一些執行引數設定，從演算法角度來說它可有可無，完全可以透過硬編碼寫入程式，但對於工程產品來說需要共享一些共用引數。

現在我們看看inferencer的實現，將相關程式碼存為seq2seq_inferencer.py檔案，如下所示：

```
1  import tensorflow as tf
2  import numpy as np
3
4  from tensorflow.keras.models import Model, load_model
5  from tensorflow.keras.layers import Input, LSTM, Dense
6  from tensorflow.keras.utils import plot_model
7
8  import json
9
10 config_file = './s2s_1_config.json'
11 model_file = './s2s_1.h5'
12
13 # 获取相关配置
14 config = {}
15 with open(config_file) as f:
16     config = json.load(f)
17
18 num_encoder_tokens = config['num_encoder_tokens']
19 num_decoder_tokens = config['num_decoder_tokens']
20 latent_dim = config['latent_dim']
21 max_num_samples = config['max_num_samples']
22
23 model = load_model(model_file)
```

以上程式碼的前兩行，還是先引入依賴庫，然後讀取引數配置檔案和模型檔案。現在問題來了，我們該如何重構模型？

在Keras官方給出的實現 [3]中並沒有涉及這個環節，因為training和inferencing被寫在一個檔案中，所以在實現inferencing時直接引用training的LSTM網路就可以了。但在實際專案中不能這樣操作，需要從訓練出的權重檔案中載入weights，重新建立網路。

如何從權重檔案中載入？或者說怎麼知道在儲存的.h5檔案中包含哪些內容？我們可以參考對應的s2s_1.json檔案。實際上，我們在模型的JSON檔案中只需關注layers部分，因為需要完成的工作其實只有一件事，即對每一層賦值：

```
"layers": [
    {
        "name": "input_1",
        "class_name": "InputLayer",
        // ...
    },
    {
        "name": "input_2",
        "class_name": "InputLayer",
        // ...
    },
    {
        "name": "lstm",
        "class_name": "LSTM",
        // ...
    },
    {
        "name": "lstm_1",
        "class_name": "LSTM",
        // ...
    },
    {
        "name": "dense",
        "class_name": "Dense",
        // ...
    }
],
```

可以看到，上面的資料描述恰好定義了5個層，分別對應如下內容。

◎　input_1：第1個輸入（encoder）。

◎　input_2：第2個輸入（decoder）。

◎　lstm：encoder網路（我們需要其輸出中的states和cells）。

◎　lstm_1：decoder網路。

◎　dense：最後完成softmax的網路。

有了這些資料後，後面的網路構建便順理成章了。我們首先從讀取的模型引數中重構網路：

```
26 encoder_inputs = model.layers[0].input   # input_1
27 encoder_outputs, state_h_enc, state_c_enc = model.layers[2].output
28 encoder_states = [state_h_enc, state_c_enc]
29 encoder_model = Model(encoder_inputs, encoder_states)
30
31 decoder_inputs = model.layers[1].input   # input_2
32 decoder_state_input_h = Input(shape=(latent_dim,), name='input_3')
33 decoder_state_input_c = Input(shape=(latent_dim,), name='input_4')
34 decoder_states_inputs = [decoder_state_input_h, decoder_state_input_c]
35 decoder_lstm = model.layers[3]
36 decoder_outputs, state_h_dec, state_c_dec = decoder_lstm(
37     decoder_inputs, initial_state=decoder_states_inputs)
38 decoder_states = [state_h_dec, state_c_dec]
39 decoder_dense = model.layers[4]
40 decoder_outputs = decoder_dense(decoder_outputs)
41 decoder_model = Model( [decoder_inputs] + decoder_states_inputs,
42                        [decoder_outputs] + decoder_states)
```

我們再來看看以上程式碼都做了什麼。

第26～29行：重構encoder網路。首先，從模型權重的第1個layers[0]中獲得其input向量，並將其存為encoder_inputs；然後，從上面對JSON檔案的瀏覽中發現encoder的LSTM網路被定義在layers[2]中，因此在第27～28行從layers[2]中獲得所需要的輸出（再次注意，這裡只需要encoder中的隱藏狀態和cells）；最後，定義encoder的輸入和輸出。

第31～34行：重構decoder網路，定義inputs。同樣，我們首先從layers[1]中獲得第1個輸入decoder_inputs。然而對decoder來說，除了自身的輸入，還需要引入encoder的hidden states和cells。這裡並不直接引用，而是定義兩個input向量作為decoder_states_inputs。decoder的這3個輸入要在做具體的預測計算時再透過encoder運算後進行賦值，後面再談這個問題。

第35～37行：重構decoder中的LSTM網路。同樣先從layers[3]中匯入LSTM層。和encoder不同，這裡需要LSTM包括運算後的target sequence結果，同時需要decoder LSTM的初始引數與在第31～34行中設定的3個輸入一致，因此在這3行中的實現和上面的第27行不一樣。

第38～42行：重構decoder中的全連線層和完整的decoder的輸入和輸出定義。和前面一樣，我們首先從layers[4]中獲取網路設定，然後將前面的decoder_outputs（decoder LSTM計算後的sequence vector）作為輸入，更新decoder_outputs，最後在第41～42行定義整個模型的輸入和輸出。

這樣，我們完成了Seq2Seq網路在inferencing階段的重構，但是還差一個重要的操作，這是在training階段沒有的操作。

將下面的圖6-12與圖6-9對比，我們可以發現：圖6-12多出一條通道，decoder的每次輸出又成為下一次decoder的輸入之一（另外兩個輸入是encoder的hidden states和cells，即上面程式碼第28行中的[state_h_enc, state_c_enc]）。這是因為在training階段，decoder的順序輸入可以直接從訓練集中獲取，而在inferencing階段只能從上一次的輸出中獲取。

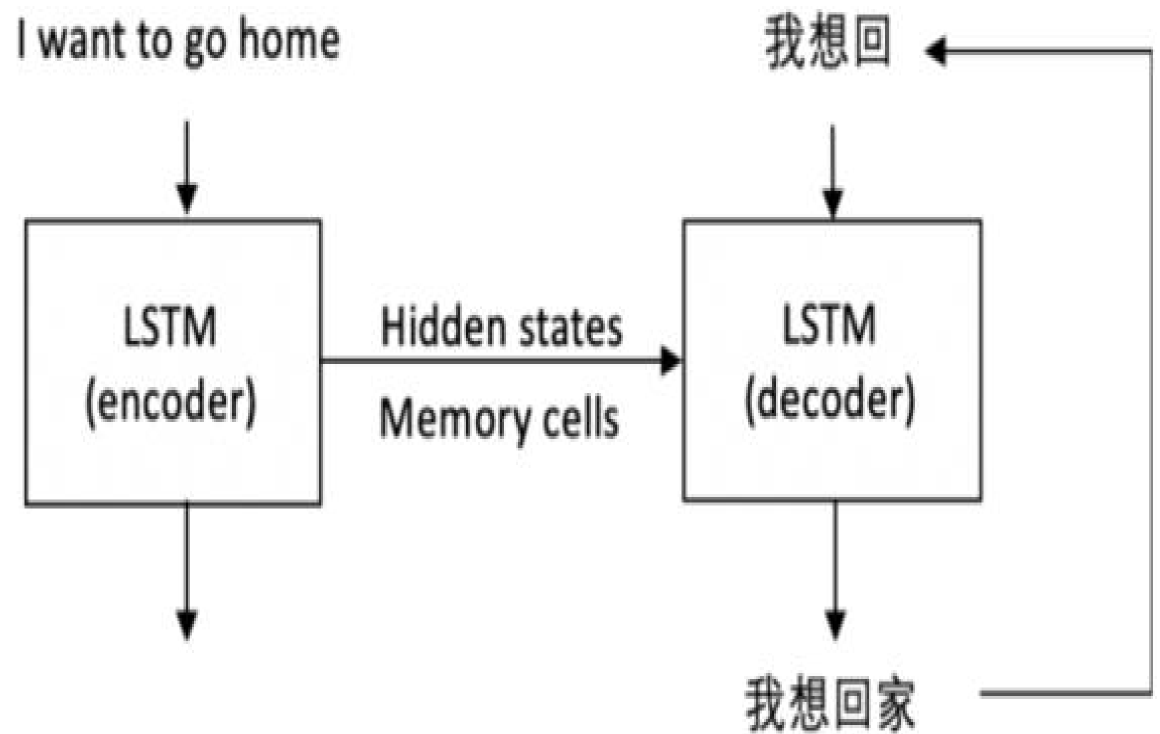


圖6-12　encoder-decoder inferencing

在inference階段，我們需要對輸入的一個句子進行以下處理。

（1）把輸入的句子向量化。

（2）用encoder處理輸入句子向量，並獲取其內部狀態作為decoder的初始值。

（3）使用句子的第1個字元作為decoder的輸入，執行一次，decoder會輸出預測的下一個字元。

（4）在decoder的輸入中增加第3步中的輸出，重複第3步。

我們可以看到，對一個句子而言，encoder只執行一次，在處理完整個句子後把內部狀態對decoder作為初始值賦值，然後按單個字元迴圈執行decoder，並更新decoder的輸入和狀態，直到結束。這也就是預測推斷（inference）的具體過程，其程式碼實現如下：

```
47 input_token_index = config['input_token_index']
48 target_token_index = config['target_token_index']
49 reverse_input_char_index = dict(
50     (i, char) for char, i in input_token_index.items())
51 reverse_target_char_index = dict(
52     (i, char) for char, i in target_token_index.items())
53
54
55 def decode_sequence(input_seq, max_decoder_seq_length):
56     # 首先用encoder模型对输入序列进行预测，获得其内部状态（即该模型输出）
57     states_value = encoder_model.predict(input_seq)
58
59     # 生成长度为1的空目标串
60     target_seq = np.zeros((1, 1, num_decoder_tokens))
61     # 设置第1个开始字符为'\t'的索引值
62     target_seq[0, 0, target_token_index['\t']] = 1.
63
64     stop_condition = False
65     decoded_sentence = ''
66     while not stop_condition:
67         output_tokens, h, c = decoder_model.predict(
68             [target_seq] + states_value)
69
70         # 获取一个token
71         sampled_token_index = np.argmax(output_tokens[0, -1, :])
72         sampled_char = reverse_target_char_index[sampled_token_index]
73         decoded_sentence += sampled_char
74
75         # 退出条件：达到最大长度或发现停止字符标记
76         if (sampled_char == '\n' or
77            len(decoded_sentence) > max_decoder_seq_length):
78             stop_condition = True
79
80         # 更新目标字符串
81         target_seq = np.zeros((1, 1, num_decoder_tokens))
82         target_seq[0, 0, sampled_token_index] = 1.
83
84         # 更新当前状态值
85         states_value = [h, c]
86
87     return decoded_sentence
```

我們看看上面的程式碼都做了什麼。

第47～52行：和在前面的trainer中讀取資料集類似，我們需要建立字元的索引表和反向索引，以便把句子用One-hot encoding向量化。

第55行：這裡定義的函式decode_sequence負責對輸入的句子向量進行處理後產生輸出，其中的第2個引數決定了輸出句子的最大長度（以免無限制生成）。

第56～57行：這裡執行一次encoder，對整個句子向量進行預測，並獲得相關的內部狀態states_value。

第60～62行：定義target_sequence。這裡首先定義有1個字元的target sequence，並用'\t'的索引值作為初始值（因為在資料集中是用『\t』作為開始符號的）。注意，target sequence本身是一個三維陣列，其維度為（1,1,num_decoder_tokens），最後這個number_decode_tokens是One-hot encoding的向量長度。

第64～68行：在定義了中止變數stop_condition和最終字串decoded_sentence後，進入迴圈，不斷用decoder進行下一個字元的預測。

第70～71行：將decoder作為預測模型，將target_seq（初始值為'/t'）加上前面encoder的內部狀態states_value作為輸入，獲得預測結果（下一個字元向量）。

第72～73行：將預測的下一個字元向量透過反向索引錶轉換為正常字元，加入最終的字串結果decoded_sentence中。

第76～78行：判斷是否中止運算。

第80～85行：更新decoder的輸入資料和狀態，然後進入下一次迴圈。

第87行：返回最終結果。

上面仔細講了Seq2Seq模型的實現，下面用一段測試程式碼看看其效果如何：

```
91  data_path = './chat.txt'
92
93  # 建立字符索引表
94  input_texts = []
95  input_characters = set()
96  with open(data_path, 'r', encoding='utf-8') as f:
```

```
97     lines = f.read().split('\n')
98
99  for line in lines[:len(lines) - 1]:
100     input_text, target_text = line.split('\t')
101     # 用'\t'作为开始标记
102     # 用'\n'作为结束标记
103     input_texts.append(input_text)
104     for char in input_text:
105         if char not in input_characters:
106             input_characters.add(char)
107
108 input_characters = sorted(list(input_characters))
109 num_encoder_tokens = len(input_characters)
110 max_encoder_seq_length = max([len(txt) for txt in input_texts])
111
112 input_token_index = dict([(char, i) for i, char in
113 enumerate(input_characters)])
114
115 def test(input_text):
116     input_data = np.zeros(
117         (1, max_encoder_seq_length, num_encoder_tokens), dtype='float32')
118
119     for t, char in enumerate(input_text):
120         input_data[0, t, input_token_index[char]] = 1.
121
122     response = decode_sequence(input_data, config['max_decoder_seq_length'])
123                 print('input:{}, response:{}'.format(input_text, response))
124
125 test_data = [
126             'hello',
127             'hello world',
128             'how are you',
129             'good morning',
130             'cheers',
131             'enjoy',
132             ]
133
134 for _, text in enumerate(test_data):
135     test(text)
```

第91～113行：從原來的訓練資料集中重新讀取文字，目的是建立輸入字元的索引表，以便進行One-hot encoding向量化。

第115～123行：定義一個test函式，輸入一個文字句子，獲取結果並列印。在第116～117行首先定義一個3維向量來代表輸入文字的向量化結果，設定最長的長度為max_encoder_seq_length（由訓練集中最長的句子決定），以及根據訓練集得到的字元總數確定向量長度。在第119～120行根據輸入文字的每個字元對向量賦值。在獲得輸入向量後，在第122行呼叫前面定義的decode_sequence進行運算，獲得最終結果。

第125～135行：定義測試資料並呼叫test函式測試。

最終的測試輸出如下：

```
input:hello, response:Helll

input:hello world, response:No, iikeeeroo.

input:how are you, response:No, 300yeerss dd.

input:good morning, response:No, 300yeerssod.

input:cheers, response:yotttoo!

input:enjoy, response:Cheerss!
```

因為我們只是手工打造了一個只包含20句對話的資料集，並且只訓練了100次，所以最終結果並不理想。即便如此，我們從上面的程式碼中也能看出該網路能生成一些有一定意義的文字反饋。

在產品級別的對話機器人實現中，對於英語語言，我們可以採用康奈爾大學提供的電影對話資料集[4]進行訓練，和前面簡單的一問一答形式的資料集不同，該資料集並沒有固定的問答形式，而是以連續的劇本對話形式存在的。由於該資料集中的原始資料還包括角色名稱和劇本相關資訊，不適合被直接用於機器學習訓練，所以研究人員提供了清晰化後的包含一萬行劇本對話的資料集[5]，部分內容如下：

…

Let me see what I can do.

Gosh, if only we could find Kat a boyfriend...

That's a shame.

Unsolved mystery.　She used to be really popular when she started high school, then it was just like she got sick of it or something.

Why?

Seems like she could get a date easy enough...

…

對於類似上面例子的資料集，我們不妨把頭兩句作為一組「問答」訓練資料，然後把該組的第2句作為第2組的第1句，以此類推，也就是把上面的劇本對話變為如表6-2所示。

表6-2　訓練資料示例

Encoder Input Sentence	Decoder Target Sentence
Let me see what I can do.	Gosh, if only we could find Kat a boyfriend...
Gosh, if only we could find Kat a boyfriend...	That's a shame.
That's a shame.	......

這裡不再重複具體實現，根據不同的資料集結構有不同的處理方式。但正如圖6-9中的架構所示，歸根結底總是從encoder input到decoder target的對映關係，Seq2Seq演算法則是以深度學習的方式來擬合這種對映關係的一種有效方案。

6.4 Attention

前面介紹的Seq2Seq，或者說encoder-decoder的實現，是自然語言處理中在2015年之前被證明非常有效的方案之一，被大量的相關應用採用。然而，儘管Seq2Seq可以達到不錯的效果，但並不能說它是最佳方案。在2015年發表的論文*Neural Machine Translation by Jointly Learning to Align and Translate*[5]中第一次提出了在Seq2Seq中加入Attention機制，並在2017年由Google發表的*Attention Is All You Need*[6]論文中徹底拋開了LSTM，純粹使用基於Attention的encoder-decoder架構（Transformer）並達到了很好的效果。當然，NLP是一個快速發展的領域，2018年年末的論文*BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*[8]進一步使用雙向Transformer再次重新整理業界記錄。對業界的最新研究成果感興趣的讀者，可以自行根據本章參考文獻閱讀相關論文，這裡並不打算對最新的前沿研究進行詳細分析。但是，對其中的核心概念Attention機制進行一定的瞭解，有助於後續進行深入研究。

6.4.1 Seq2Seq的問題

在Seq2Seq的實現中，我們把encoder對整個句子處理後的隱藏狀態作為輸入提供給decoder對每一個詞進行處理，如圖6-13所示。

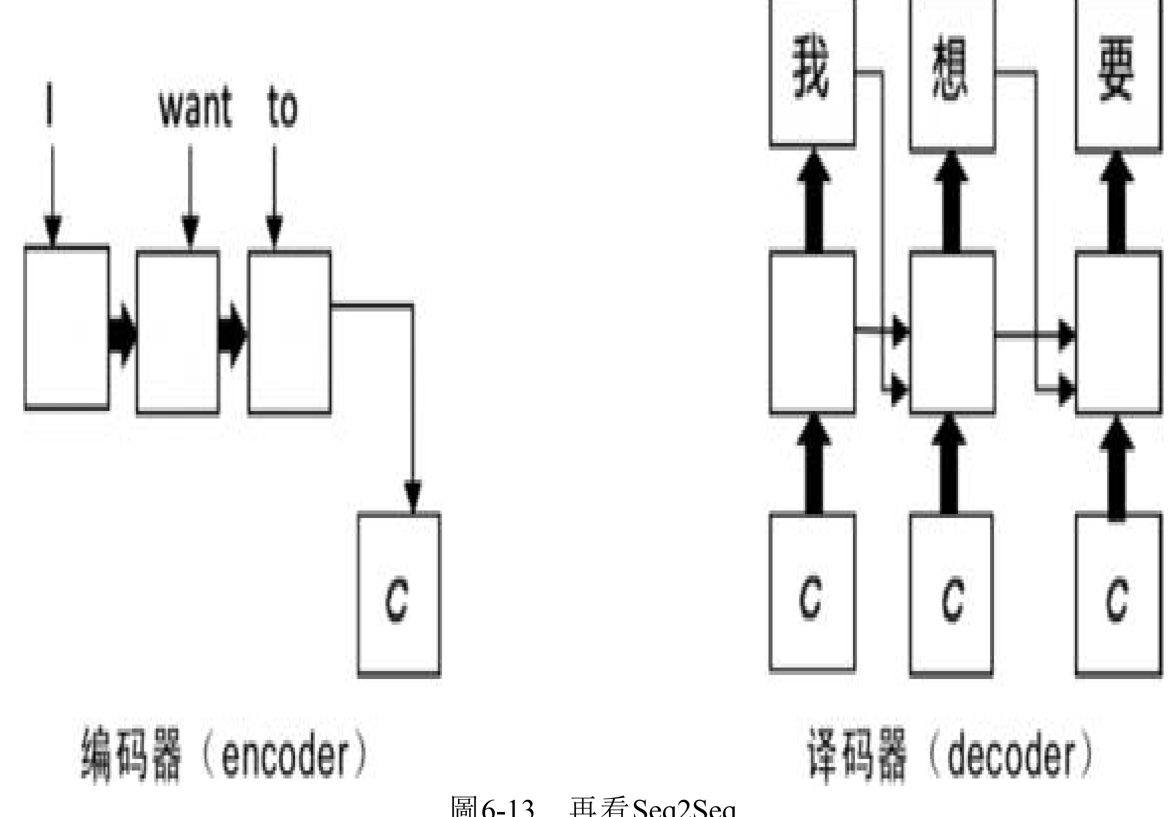


圖6-13　再看Seq2Seq

在圖6-13中可以看到，我們首先從encoder獲得對句子整體處理後的相關資訊，然後把整個句子的相關資訊作為引數，提供給decoder進行每次的處理。這裡的問題是圖6-13中的*C*代表整個句子的資訊，對於短句而言資訊量有限，這樣做是可以的，但對於較長的句子，我們在處理句子開頭的詞語時，並不真的需要關注句子末尾的內容。因此在處理長句時（或者任何長序列資料時），將完整句子的資訊作為引數提供給decoder，並不能達到很好的效果。因此在*Attention Is All You Need* [6]這篇論文中提出了Attention機制，讓encoder提供給decoder的不再是完整句子的隱藏狀態，而是在完整句子的隱藏狀態之上再新增一個步驟：找出每個詞語在當前位置（Timestep）的一個權重關係。下面看看Attention的工作原理。

6.4.2　Attention的工作原理

我們首先從整體講一下Attention的工作原理：對於所處理的每個詞，我們都需要獲取針對當前詞語的句子資訊。也就是說，我們提供給decoder的句子資訊，是根據當前詞語而變化的不同vector，而不是像在Seq2Seq裡面那樣是一個固定的vector。

我們記這個vector為C_i，其中，C代表上下文（Context），i代表句子中的位置（Timestep），那麼如何獲得這個C_i呢？在本章參考文獻[6]中提到「The context vector C_i depends on a sequence of annotation ....Each annotation contains information about the whole input sequence with a strong focus on the parts surrounding the i-th word of the input sequence」。根據這句話，我們的重點是對句子中的詞語獲得一個針對其位置i的annotation，記為A_i。

明白了C_i和A_i的概念，我們來一步一步地看具體怎麼獲得C_i，又怎麼將其提供給decoder。

1.獲得每個字的隱藏狀態

我們假設要翻譯「深度學習」為「Deep Learning」，則我們仍然要有一個encoder，其中的RNN對「深度學習」這4個字處理後分別獲得h_1、h_2、h_3和h_4的隱藏狀態，如圖6-14所示。

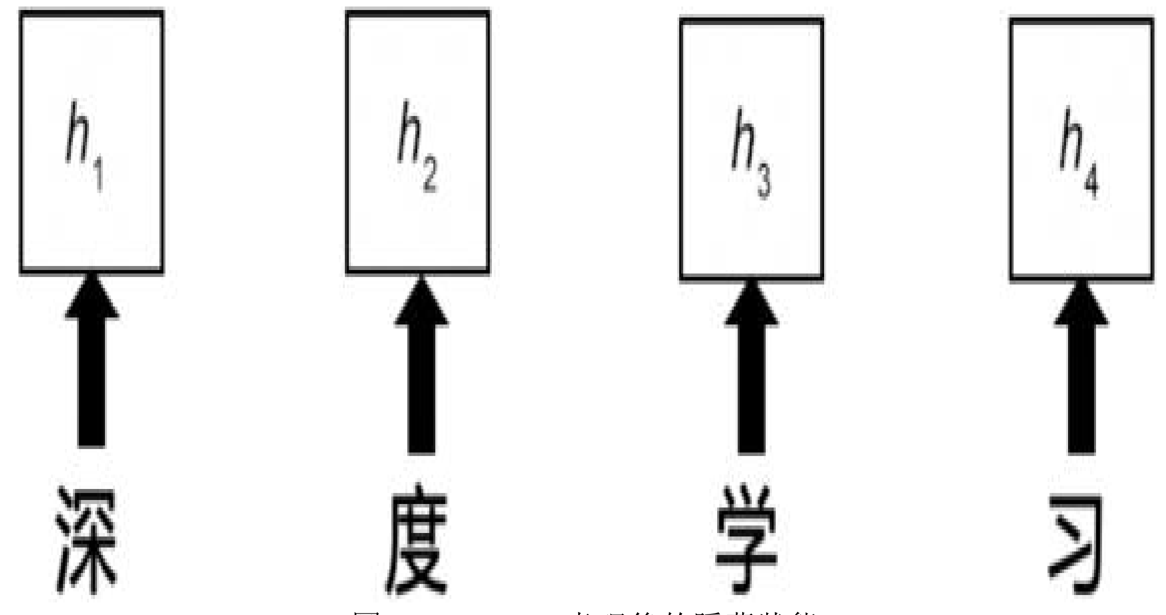


圖6-14　encoder處理後的隱藏狀態

2.獲得第*i*個詞語的相關值a_i

在獲得圖6-14中的隱藏狀態之後，我們需要做如圖6-15所示的操作。

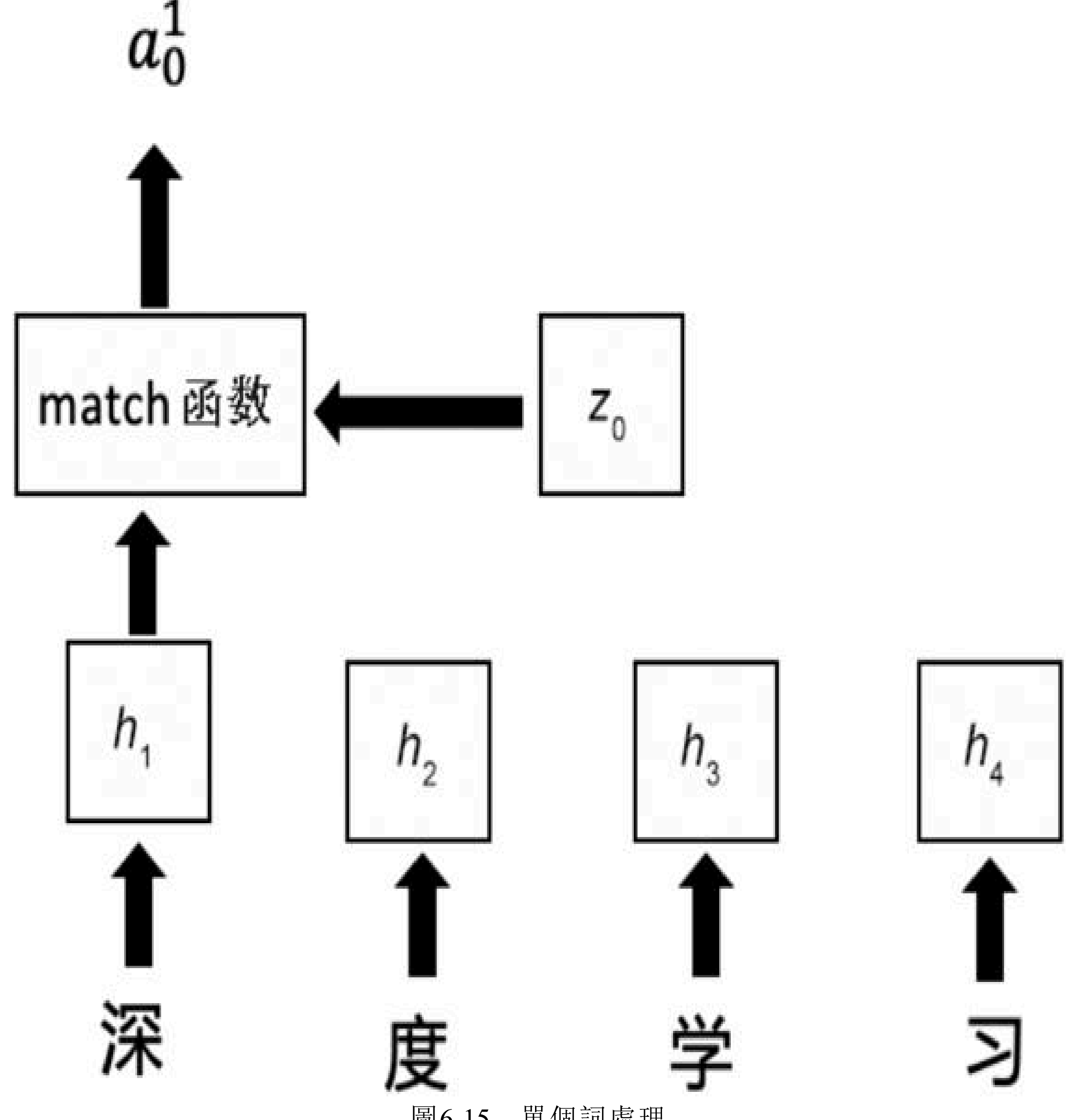


圖6-15　單個詞處理

在圖6-15中，我們可以將(h_1,h_2,h_3,h_4)看作一個初始vector Z_0，其值需要在訓練後確定。而對於每個詞的相關值，則透過match 函式獲得：

$$a_j^i = match(h_i, z_j)$$

其中，match函式的實現現在沒有定論，可以使用多種不同的演算法，比如直接算二者的餘弦距離，設定一個簡單的神經網路，最後輸出單獨的數值或者進行矩陣轉換，只要確保最後輸出一個數值即可[9]。

3.獲取Annotation和Context

我們首先對在上面一步獲取的a_j^i做softmax計算，所得到的數值即上文所介紹的Annotation（在memory j輸入時對應的位置i上）。

$$C_i = A_i^j h_i$$

然後，我們最終需要的C_0向量是所有Annotation與對應的hidden states的乘積之和，即

$$C_i = A_i^j h_i$$

其計算過程如圖6-16所示。

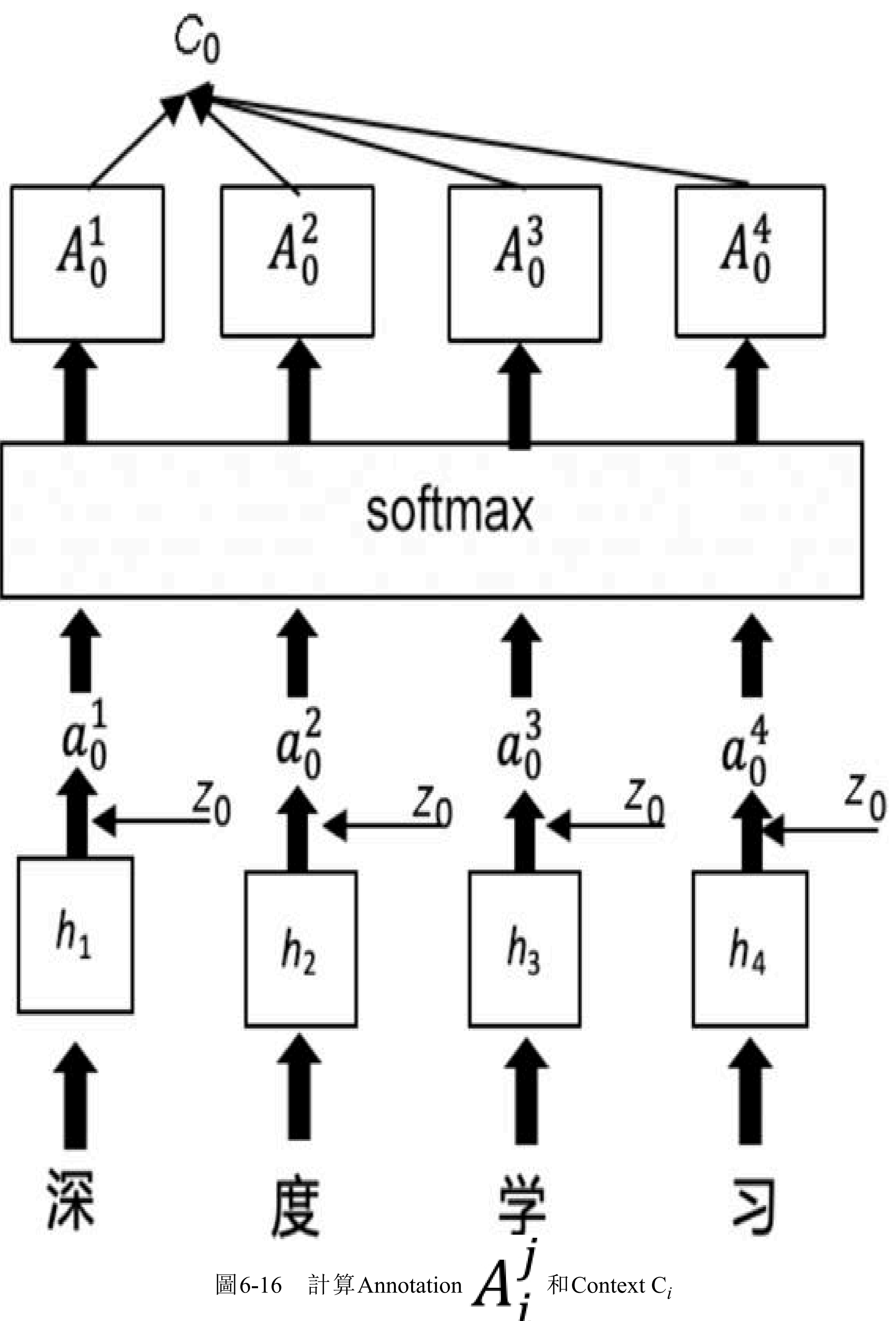


圖6-16　計算Annotation A_i^j 和Context C_i

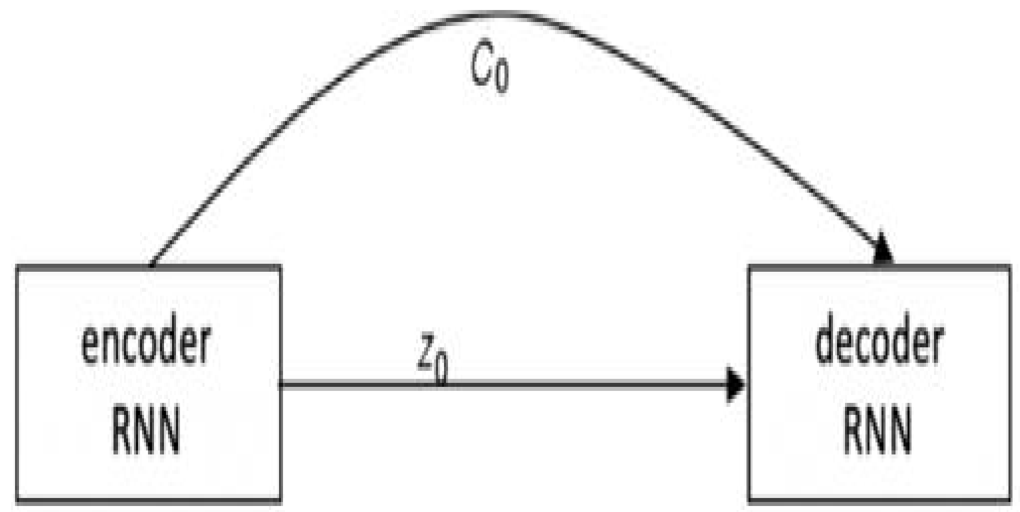
在圖6-16中，我們不妨假設$A_0^1 = 0.5$，$A_0^2 = 0.5$，$A_0^3 = 0.5$，$A_0^4 = 0$，則有

$$C_0 = 0.5 \cdot h_1 + 0.5 \cdot h_2 + 0 \cdot h_3 + 0 \cdot h_4$$

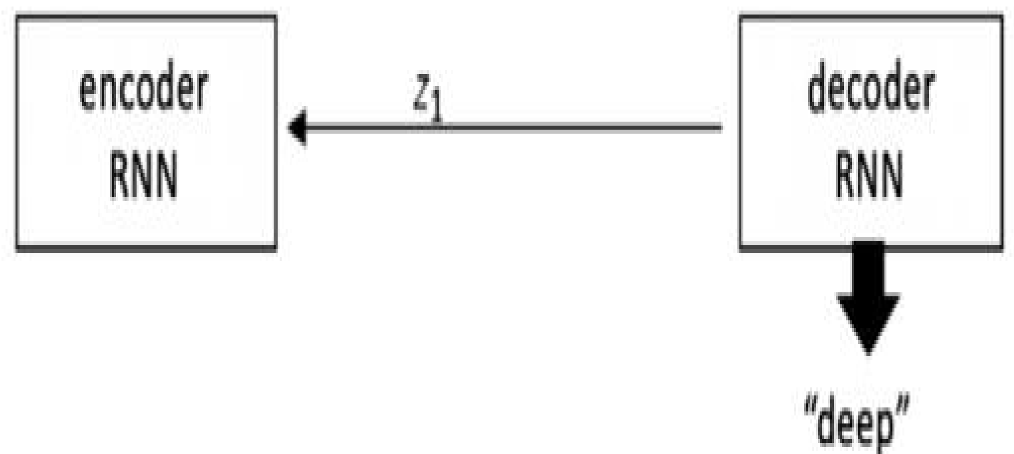
4.輸入decoder，獲取第1個輸出

在圖6-17(a)中，我們對比圖6-12可以看到，實際上Attention機制和Seq2Seq的不同之處在於：Seq2Seq對decoder的輸入是encoder的隱藏狀態和記憶單元；Attention機制則是在隱藏狀態上加了一層做了特殊處理，對decoder的輸入是處理後的上下文C_0和記憶單元z_0。

在圖6-17(b)中，我們把decoder的隱藏狀態返回encoder，並作為z_1向量又重複前面的第2步和第3步，和h_1、h_2、h_3、h_4各自配對並輸入match函式中進行softmax操作，獲得Annotation A_1^1、A_1^2、A_1^3、A_1^4，並得到C_1，再重複第4步，獲得第2個輸出，如圖6-18所示。



(a) 将encoder的记忆单元 z_0 和上下文 C_0 输入decoder中



(b) ecoder将其隐藏状态作为 z_1 输入encoder，进行下一个 C_1 的计算

圖6-17　decoder的工作

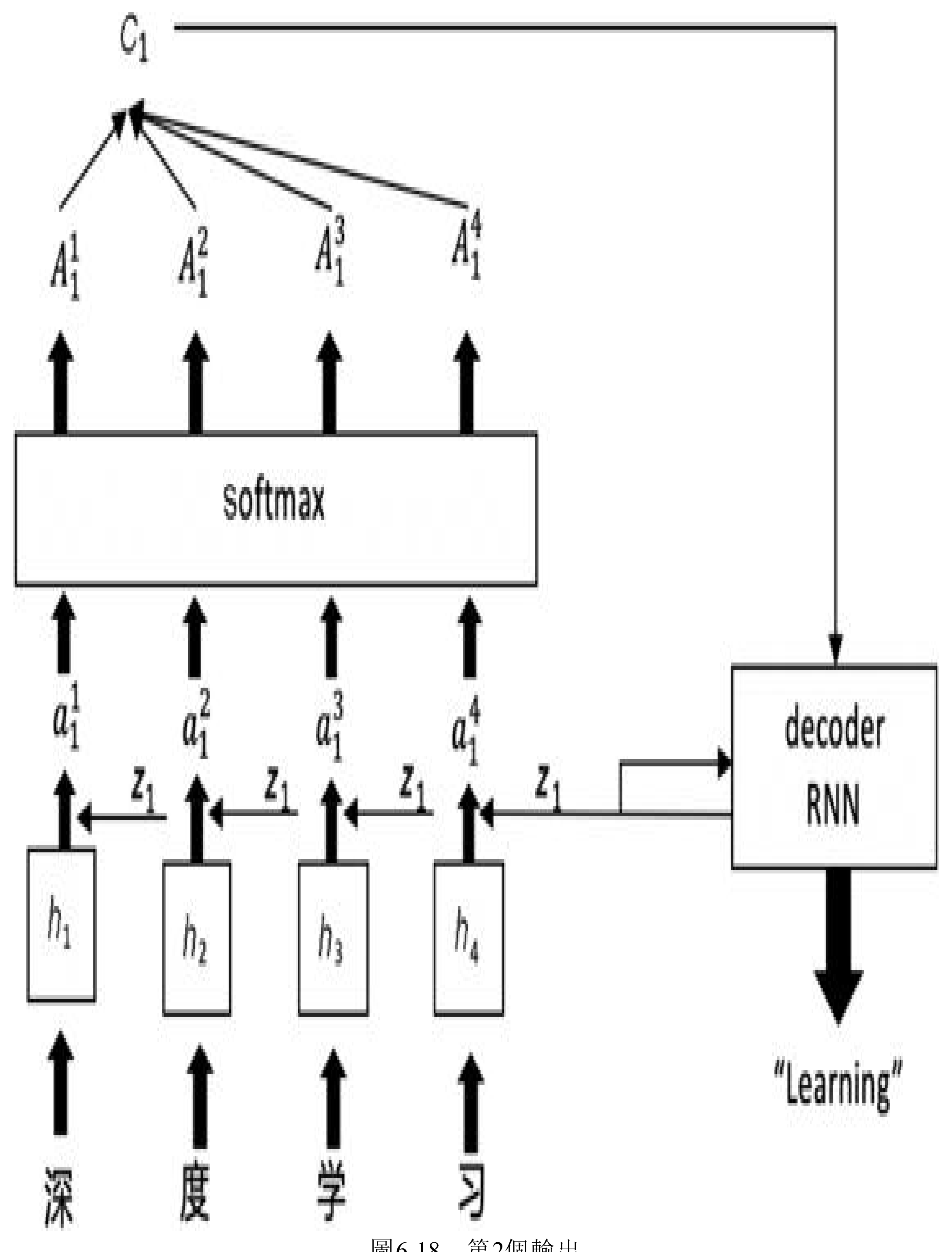


圖6-18　第2個輸出

以上就是Attention機制在自然語言處理中的基本工作原理。實際上，我們可以用同樣的思路進行影像生成（根據周圍畫素生成新的畫

素）等操作，在此不再贅述。

6.4.3　Attention在Keras中的實現

前面講了Attention機制的具體原理，本節將透過程式碼示例看看具體是如何用Keras實現的。

這裡首先把在3.2節講解的步驟用偽程式碼進行描述。假設我們的輸入是$[x_1, x_2, x_3]$，輸出是y_1，那麼根據3.2節中的步驟，我們首先透過encoder RNN獲得隱藏狀態：

$$h_1, h_2, h_3 = \text{encoder}(x_1, x_2, x_3)$$

我們可以把這裡的h_1、h_2、h_3作為LSTM網路的輸出：

```
activations = LSTM(units, return_sequences=True)(embeddings)
```

然後需要透過match函式比較decoder的記憶單元輸出z_0和每個encoder中隱藏狀態的差別。對於decoder的第1個輸出，我們可以將其設為0：

```
e11 = match(0, h1)
e12 = match(0, h2)
e13 = match(0, h3)
```

這裡的重點在於對match函式的選擇。實際上，大多數Attention網路的實現都參考了本章參考文獻[10]中的實現，使用將tanh作為啟用函式的一個全連線網路，在Keras中可以進行如下實現：

```
a = Dense(1, activation='tanh', bias_initializer='zeros')(activations)
```

注意，我們將bias初始化為0，bias即對應預設的decoder輸出。

緊接著進行softmax操作：

```
sum = exp(e11) + exp(e12) + exp(e13)
a11 = exp(e11)/sum
a12 = exp(e12)/sum
a13 = exp(e13)/sum
```

這裡就可以直接使用Keras的softmax函式了：

```
a = Flatten()(a)
attention = Activation('softmax')(a)
attention = RepeatVector(units)(attention)
attention = Permute((2, 1))(attention)
```

以上程式碼中的第3～6行計算了每一個timestep在句子中的重要程度。

最後便可獲得Context vector：

$$c_1 = a_{11} * h_1 + a_{12} * h_2 + a_{13} * h_3$$

這在Keras中用merge函式只需一行程式碼即可實現：

```
Context = merge([activations, attention], mode='mul')
```

綜合上面的程式碼，我們得到一個完整的Attention Keras實現：

```
activations = LSTM(units, return_sequences=True)(embeddings)
a = Dense(1, activation='tanh', bias_initializer='zeros')(activations)
a = Flatten()(a)
attention = Activation('softmax')(a)
attention = RepeatVector(units)(attention)
attention = Permute((2, 1))(attention)
context = merge([activations, attention], mode='mul')
```

Attention機制在目前並沒有一個標準實現，以上程式碼來自GitHub的討論[11]。然而這並不意味著可以直接使用以上程式碼。6.4.4節將結合一個簡單的例子來看看如何在具體的程式碼中使用Attention。

6.4.4 Attention示例

我們先給出一個簡單的例子：對於一個長度為6的整數陣列，只保留前面的3個數字，而將後面的3個數字置0。例如：

```
input: [1,2,3,4,5,6], output: [1,2,3,0,0,0]
input: [20,12,6,12,10,30], output: [20,12,6,0,0,0]
```

該例子源於本章參考文獻[12]所給出的示例，它在本章參考文獻[12]中用於對比encoder-decoder和Attention的效果，但因為其基於較早的Keras版本，所給的Attention實現已經無法在新版本的Keras中執行，所以這裡根據TensorFlow自帶的Keras版本對其進行了調整。

我們需要實現如下輔助函式：

◎ 生成隨機陣列；

◎ One-hot encoding；

◎　One-hot decoding；

◎　生成訓練資料。

因為需要做One-hot encoding，所以為了加快訓練速度，我們設定數字範圍為[0,50]。下面來看具體的程式碼實現。

建立test_attention.py檔案：

```
1  from random import randint
2  from numpy import array
3  from numpy import argmax
4  from numpy import array_equal
5  from tensorflow.keras.models import Sequential, Model
6  from tensorflow.keras.layers import LSTM, Input, Dense, RepeatVector, Flatten
7  from tensorflow.keras.layers import Activation, Permute, multiply
```

可以看到，該檔案的第1～7行引入了相關依賴。注意，第6～7行引入了需要的所有層型別：

```
8  # 生成一组随机整数
9  def generate_sequence(length, n_unique):
10     return [randint(0, n_unique-1) for _ in range(length)]
11
12 # 进行One-hot encoding
13 def one_hot_encode(sequence, n_unique):
14     encoding = list()
15     for value in sequence:
16         vector = [0 for _ in range(n_unique)]
17         vector[value] = 1
18         encoding.append(vector)
19     return array(encoding)
20
21 # 对编码后的字符串进行One-hot decoding
22 def one_hot_decode(encoded_seq):
23     return [argmax(vector) for vector in encoded_seq]
```

我們看看上面這段程式碼都做了什麼。

第9～10行：根據給定的長度length，生成[0, n_unique-1]區間的隨機整數陣列。

第13～19行：對所生成的整數陣列進行One-hot encoding。因為整個數值區間為[0, n_unique-1]，所以每個數的encoding vector長度都為n_unique，根據所對應的數值所在位置，將該vector的對應值設為1。最後生成一個2D陣列，其中的每個元素都是不同的encoding vector。例如，sequence為[1,2,10]，則對應的One-hot encoding結果如下：

```
 [ [0,1,0,0,0,0,0,0,0,0,0,0,0,0,…], [0,0,1,0,0,0,0,0,0,0,0,0,0,0,…],
[0,0,0,0,0,0,0,0,0,0,10,0,0,…]]
```

第22～23行：對encoding vector進行decode（解碼），將其恢復為對應的整數。從第13～19行的程式碼可以看到，一個整數的encoding vector只會有一個位置為1，其他位置為0，所以我們可以用argmax函式找到每個vector的最大值所在的位置，這就是所對應的整數，並作為對應的結果返回：

```
26 def get_pair(n_in, n_out, cardinality):
27     # 生成随机序列
28     sequence_in = generate_sequence(n_in, cardinality)
29
30     sequence_out = sequence_in[:n_out] + [0 for _ in range(n_in-n_out)]
31     # One-hot encoding
32     X = one_hot_encode(sequence_in, cardinality)
33     y = one_hot_encode(sequence_out, cardinality)
34     # 重组序列，变为三维数组
35     X = X.reshape((1, X.shape[0], X.shape[1]))
36     y = y.reshape((1, y.shape[0], y.shape[1]))
37     return X,y
```

上面的get_pair函式建立了訓練資料X和y。其中，X是給定範圍內的隨機整數陣列，y是隻包含前n_out個整數的處理後的結果。

第28行：建立隨機整數陣列作為輸入，其中，n_in指陣列長度，cardinality指數值維度（這裡就是數值的取值空間[0,49]，維度為50）。

第30行：建立輸出陣列，取輸入陣列的前n_out個數字，然後用0填充剩餘位置（保持和輸入陣列最終的長度一樣）。

第32～33行：呼叫前面的one_hot_encoder，將陣列中的每個整數都轉換為vector。

第35～36行：呼叫reshape函式，將X和y轉換為可作為模型訓練輸入的三維陣列（樣本數為sample_number，樣本長度為sample_length，樣本維度為sample_dimension）。這裡只生成了一組資料，因此樣本數為1，樣本長度則為每個樣本的timestep，樣本維度為之前定義的資料取值範圍，因此分別為X.shape[0]、X.shape[1]、y.shape[0]和y.shape[1]。

在定義好輔助資料的相關方法後，我們開始進行模型的具體實現：

```
39 def attention_model(n_timesteps_in, n_features):
40         units = 50
41         inputs = Input(shape=(n_timesteps_in, n_features))
42
43         encoder = LSTM(units, return_sequences=True, return_state=True)
44         encoder_outputs, encoder_states, _ = encoder(inputs)
45
46         a = Dense(1, activation='tanh', bias_initializer='zeros')(encoder_outputs)
47         a = Flatten()(a)
48         annotation = Activation('softmax')(a)
49         annotation = RepeatVector(units)(annotation)
50         annotation = Permute((2, 1))(annotation)
51
52         context = multiply([encoder_outputs, annotation])
53         output = Dense(n_features, activation='softmax', name='final_dense')(context)
54
55         model = Model([inputs], output)
56         model.compile(loss='categorical_crossentropy', optimizer='adam',
57 metrics=['acc'])
58         return model
```

我們看看上面這段程式碼都做了什麼。

第40行：定義LSTM中的units個數。根據Keras的LSTM網路的定義，這是LSTM輸出的空間維度，因為我們輸出的是[0,49]每個數值的機率分佈，所以維度為50。

第41～44行：定義初始的LSTM網路，其中，Input被定義為(n_timesteps, n_features)的二維陣列，這是因為如前所述，每一個樣本都由timestep決定長度，而每個timestep上的資料維度都為n_features。

第46～51行：這是6.4.3節所描述的Attention實現，在此不再贅述。

第52～53行：上一節在講解Attention實現時有一步沒有提到，即當我們將LSTM輸出和Annotation相乘後，其實得到的context還需要做一個softmax轉換計算，其輸出才是我們需要的[0,49]的機率分佈。如果不加上第52行的處理，則雖然模型能執行，但無法得到預期的結果。

第55～58行：建立模型並返回。

我們再定義一種訓練模型的方法：

```
59  def train_evaluate_model(model, n_timesteps_in, n_timesteps_out, _features):
60        for epoch in range(5000):
61      X,y = get_pair(n_timesteps_in, n_timesteps_out, n_features)
62      model.fit(X, y, epochs=1, verbose=0)
63
64        total, correct = 100, 0
65        for _ in range(total):
66      X,y = get_pair(n_timesteps_in, n_timesteps_out, n_features)
67      yhat = model.predict(X, verbose=0)
68      result = one_hot_decode(yhat[0])
69      expected = one_hot_decode(y[0])
70      if array_equal(expected, result):
71            correct += 1
72
73        return float(correct)/float(total)*100.0
```

我們看看上面這段程式碼都做了些什麼。

第60～62行：直接利用前面定義的get_pair方法生成一條訓練資料，並呼叫fit函式對模型訓練一次，一共生成5000次資料進行訓練。

第64～71行：生成100條測試資料進行accuracy檢測。第66行首先再次生成一條測試資料；第67行使用模型的predict方法進行預測；第68～69行將預測結果和所生成資料的真實結果進行decode，將預測結果和真實結果都轉換成類似[1,2,3,10,20,30]的數值陣列，並在第70～71行進行比較，如果一致，則預測成功。最後在第73行返回預測的準確率。

下面我們來執行一下，看看效果：

```
75  n_features = 50
76  n_timesteps_in = 6
77  n_timesteps_out = 3
78  n_repeats = 5
79
80  for _ in range(n_repeats):
81      model = attention_model(n_timesteps_in, n_features)
82      accuracy = train_evaluate_model(model, n_timesteps_in, n_timesteps_out,
83  n_features)
84      print(accuracy)
```

第75～78行：定義全域性引數。n_features是資料維度[0,49]；n_timesteps_in是資料長度；n_timesteps_out是所擷取的資料長度；n_repeats是實驗次數。

第80～84行：執行n_repeats所定義的實驗次數。第81行建立attention模型；第82～83行進行訓練和測試；第84行列印結果。

執行結果如下：

```
86.0
74.0
79.0
51.0
89.0
```

可以看到，因為每次的訓練資料不同，所以最終模型的準確率變化較大，但整體平均在75%以上。如果使用本章參考文獻[12]中單純基於LSTM的encoder-decoder模式，則其準確率不會超過10%。

6.5　本章小結

本章針對自然語言處理領域，以聊天機器人為例子切入，首先介紹瞭如BOW、Embedding、word2vec等關鍵概念，然後仔細講解了RNN和LSTM網路的工作原理。對這方面內容的介紹，重點參考了李宏毅教授的講座[9]，推薦有興趣的讀者自行學習。

在介紹了和NLP相關的基本概念之後，我們以Seq2Seq模型為重點，講解了語言翻譯和聊天機器人的訓練素材（語料庫）組織形式及Seq2Seq代表的encoder-decoder的具體工作流程，最後用Keras實現了一個可用於生產環境的完整Seq2Seq模型，並展示了具體的執行效果。

本章最後引入了近年來NLP領域非常重要的Attention機制，討論了其工作原理和Keras程式碼實現（推薦讀者自行閱讀本章參考文獻[11]中關於Attention模型實現的討論），然後透過一個簡單的整數陣列處理流程的程式碼實現，體現了Attention機制在實際應用中的具體使用方式。當然，因為機制的複雜性，Attention在不同領域中的使用方式都不同，這裡只是讓讀者有一個基本瞭解。若想對Attention有進一步的應用，則可以閱讀本章參考文獻[7][8]等自行學習。

6.6　本章參考文獻

[1]　「Learning phrase representations using RNN encoder-decoder for statistical machine translation」, K.Cho, D.Bahdanau, F.Bougares, H.Schwenk, Y.Bengio, 2014

[2]　「Sequence to Sequence Learning with Neural Networks」, I.Sutskever, O.Vinyals, Q.V.Le, 2014

[3] https://github.com/keras-team/keras/blob/master/examples/lstm_seq2seq.py

[4] https://www.cs.cornell.edu/ ～ cristian/Cornell_Movie-Dialogs_Corpus.html

[5] https://github.com/nicolas-ivanov/debug_seq2seq/blob/master/data/train/movie_ lines_cleaned_10k.txt

[6] 「Neural Machine Translation by Jointly Learning to Align and Translate」, Dzmitry Bahdanau, Kyunghyun Cho, Yoshua Bengio, 2015

[7] 「Attention Is All You Need」, Google, 2017

[8] 「BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding」, Google, 2018

[9] 「Attention-based Model」, http://speech.ee.ntu.edu.tw/～tlkagk/courses_MLDS18.html, Hongyi-Li

[10] 「Attention-Based Bidirectional Long Short-Term Memory Networks for Relation Classification」, P.Zhou, W.Shi, et al, 2016

[11] https://github.com/keras-team/keras/issues/4962

[12] https://machinelearningmastery.com/encoder-decoder-attention-sequence-to-sequence-prediction-keras/

[13] http://www.manythings.org/anki/

第7章　影像分類實戰

本章將介紹深度學習在影像分類中的應用，並以卷積神經網路為重點進行講解。在瞭解其執行原理後，我們將基於Keras框架，使用交通圖示資料集實現一個從模型訓練到線上服務的完整流程。

本章將重點講解卷積神經網路在圖片分類中應用效果突出的原理，讀者應仔細瞭解其實現原理，不必急於執行程式碼。另外，本章程式碼並不複雜，動手實現也有利於在第8章中更好地理解目標識別過程。

7.1　影像分類與卷積神經網路

7.1.1　卷積神經網路的歷史

在過去很長一段時間內，圖片分類一直都是AI研究的難題之一，直到2011年，在IJCNN影像分類比賽中，基於卷積神經網路的演算法在表現上才首次超越了人類[1]。然後在2012年的ImageNet比賽中，基於卷積神經網路的AlexNet在大規模資料集上取得了令人矚目的成績。從那時開始，卷積神經網路就成為影像分類的主流演算法。

基於深度學習的影像分類在原理上並不複雜，一個簡單的例子如圖7-1所示。

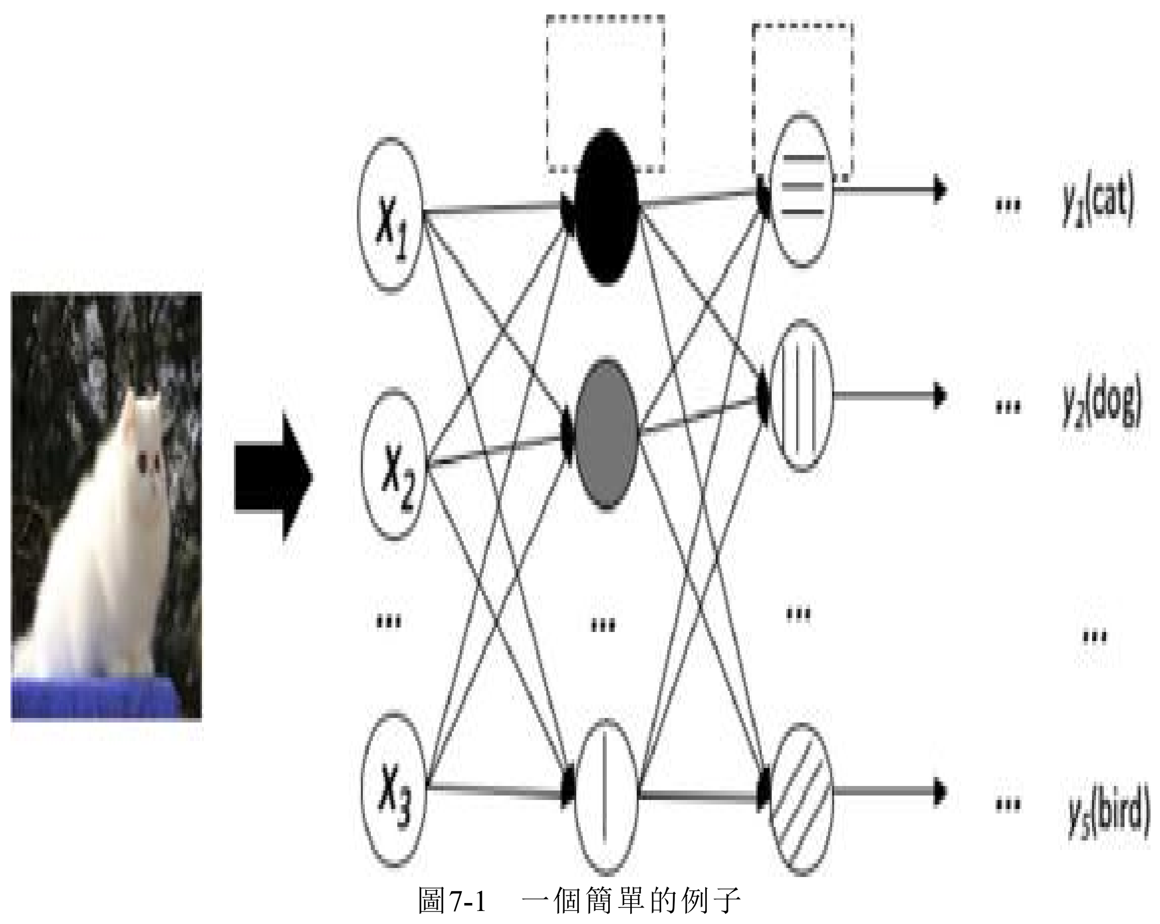


圖7-1　一個簡單的例子

在圖7-1中，我們把圖片的畫素值作為輸入，在每一層都可以學到不同的影像特徵，例如在第1層學到影像的區域性顏色塊和簡單線條，在第2層學到稍微複雜的簡單圖案，以此類推，最後輸出不同類別的機率，例如y_1代表貓的影像，y_2代表狗的影像，等等。

但採用這種方式的問題在於要處理的資料量太大，假設我們要處理一張100×100大小的RGB影像，則輸入的畫素值有30000（3×100×100）個，假設第1層的神經元有1000個，每個a_i的輸入都為$w_{i1}\times x_1 + w_{i2}\times x_2 + \ldots + w_{i30000}\times x_{30000}$，則一共有1000×30000個引數需要訓練，而這只是第1層。儘管在理論上用深度學習來處理影像分類的邏輯是清晰的，但在卷積神經網路出現前並沒有一種切實可行的方式使其得以普及。

7.1.2　影像分類的3個問題

卷積神經網路在影像分類上大放異彩併成為影像分類演算法，這不光利用了一些影像分類的常識，而且和影像分類自身的一些特點分不開。人們在做影像分類時需要解決如下3個問題。

問題一：若某些有代表性的區域性特徵僅僅出現在極小的範圍內，那麼如何發現該類別特徵而不用處理整張圖片？

在圖7-2中，(a)圖為一輛汽車，我們可以發現輪胎是它較為典型的特徵，如(b)圖所示。換句話說，如果我們能發現類似輪胎的特徵，則該圖片屬於汽車相關類別的機率較高。然而，該如何發現區域性區域記憶體在的特徵呢？

(a)

(b)

圖7-2　圖片的區域性特徵

問題二：類似的區域性特徵會出現在圖片的不同區域，該如何有效處理？

如圖7-3所示，(a)圖和(b)圖都有類似的車頭特徵，然而(a)圖的特徵在圖片的左邊，(b)圖的特徵在圖片的右邊。若我們並不想強行設定

對不同位置的區域性特徵分別進行處理，那麼該如何自動發現分佈在不同位置的相似特徵呢？

(a)　(b)

圖7-3　分佈在不同位置的相似特徵

問題三：縮小圖片（均勻取樣）並不影響圖片類別，那麼能否利用這個思路來減少要處理的資料量？

在圖7-4中，我們把圖片縮小後並不會改變圖片中物體的類別。圖片縮小的過程實際上是一個取樣（Sampling）的過程。正如前面利用全連線網路進行影像分類時所提到的，全連線網路的問題是需要處理的引數過多，那麼我們能否利用取樣方式來減少要處理的引數量呢？

(a)

(b)

圖7-4　不同大小的同樣物體

7.2節將基於這3個問題，講解卷積神經網路是如何執行的，並介紹解決這3個問題的方法。

7.2　卷積神經網路的工作原理

從理論上講，卷積神經網路的實現包括3個主要步驟：卷積運算（Convolution）、池化（Pooling）和檢測（Detector）。

其中，檢測的作用是把卷積運算後得到的結果輸入非線性啟用函式如relu[2]中，如圖7-5所示。

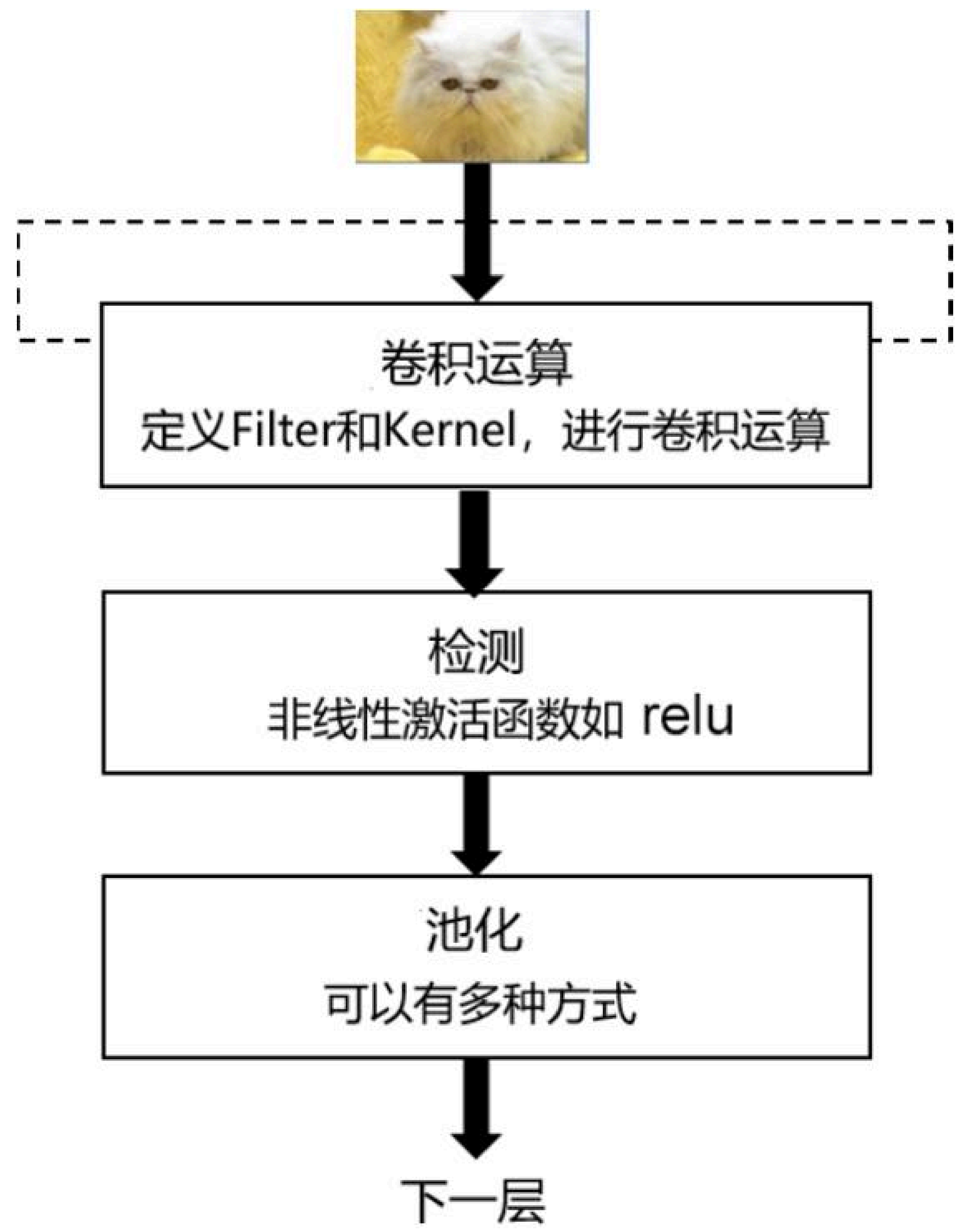


圖7-5　卷積神經網路的基本工作環節

如圖7-5所示的卷積神經網路組成，參考了MIT出版社在*Deep Learning* [2]一書中的描述，但我們在Keras和其他框架的實際開發中，

往往無須把檢測這個啟用函式處理單獨作為一層，因為在定義Conv2D層時可以直接指定所使用的activation函式：

```
model.add(Conv2D(64, (5, 5), activation='relu'))
```

因此在實現中，卷積神經網路是由多個卷積層及池化層疊加而成的。實際上，也可以在最後加入啟用層，如圖7-6所示。

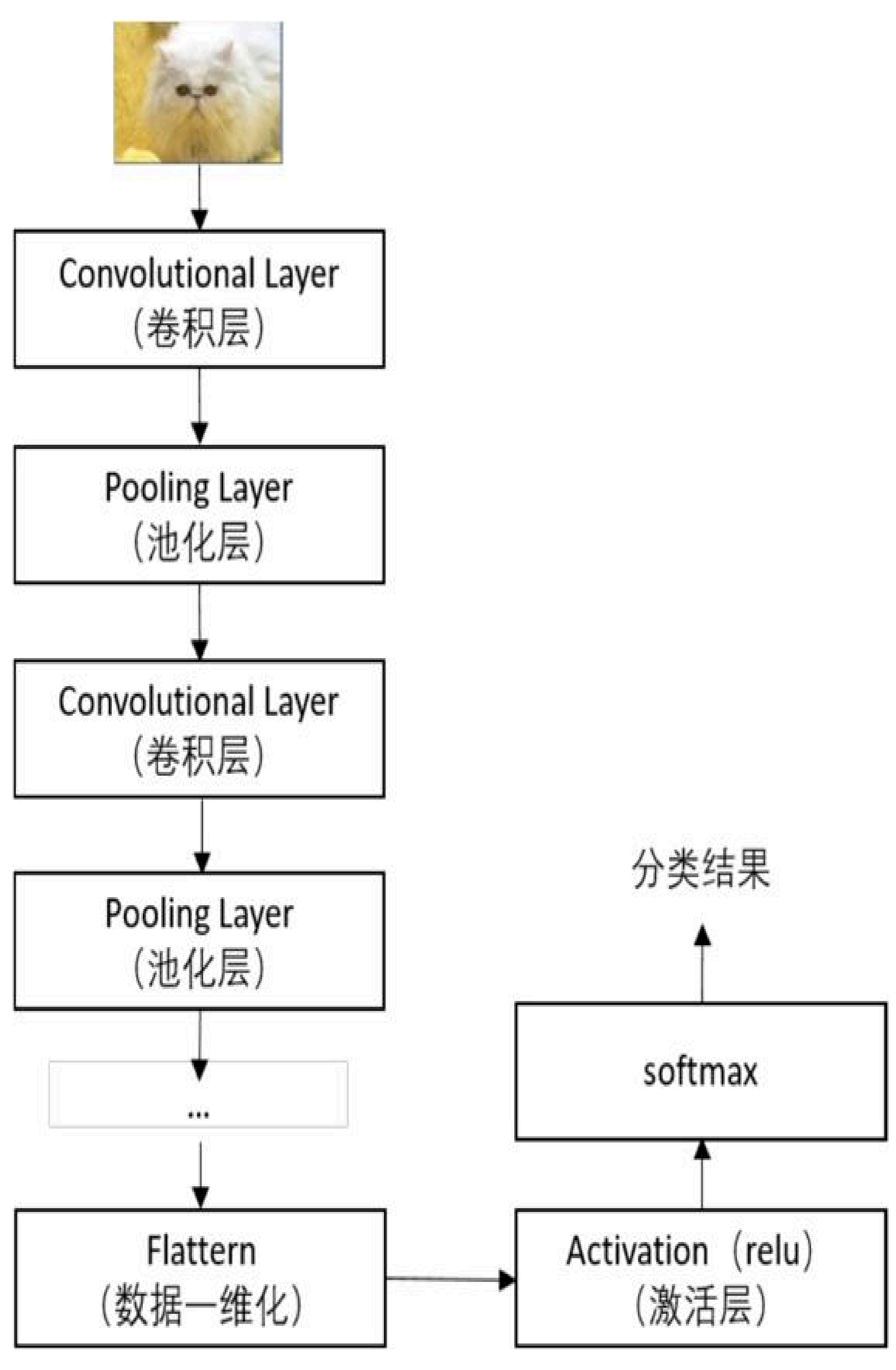

Convolutional Layer
（卷积层）
Pooling Layer
（池化层）
Convolutional Layer
（卷积层）
Pooling Layer
（池化层）
...
Flattern
（数据一维化）
Activation（relu）
（激活层）
softmax
分类结果

圖7-6　卷積神經網路的基本流程

本章的後面會仔細討論程式碼實現，我們先把注意力集中在卷積運算和池化（後簡稱Pooling）這兩個主要概念上，如下所述。

7.2.1　卷積運算

卷積運算是影像處理領域存在已久的一種運算方式。假設有一個3×3矩陣 $\boldsymbol{A}=\begin{bmatrix}0 & 1 & 1\\1 & 0 & 0\\0 & 1 & 0\end{bmatrix}$，以及一個2×2矩陣 $\boldsymbol{B}=\begin{bmatrix}1 & 0\\0 & 1\end{bmatrix}$。

我們從$\boldsymbol{A}$的位置(0,0)開始，以2×2的視窗大小，從左到右、從上到下，將視窗內$\boldsymbol{A}$的子矩陣同$\boldsymbol{B}$做點積運算（Dot Production），如圖7-7所示。

$$\begin{bmatrix}0 & 1 & 1\\ 1 & 0 & 0\\ 0 & 1 & 0\end{bmatrix} \rightarrow \begin{bmatrix}0 & 1\\ 1 & 0\end{bmatrix} \otimes \begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix} = 0\times1+1\times0+1\times0+0\times1=0$$

$$\begin{bmatrix}0 & 1 & 1\\ 1 & 0 & 0\\ 0 & 1 & 0\end{bmatrix} \rightarrow \begin{bmatrix}1 & 1\\ 0 & 0\end{bmatrix} \otimes \begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix} = 1\times1+1\times0+0\times0+0\times1=1$$

$$\begin{bmatrix}0 & 1 & 1\\ 1 & 0 & 0\\ 0 & 1 & 0\end{bmatrix} \rightarrow \begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix} \otimes \begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix} = 1\times1+0\times0+0\times0+1\times1=2$$

$$\begin{bmatrix}0 & 1 & 1\\ 1 & 0 & 0\\ 0 & 1 & 0\end{bmatrix} \rightarrow \begin{bmatrix}0 & 0\\ 1 & 0\end{bmatrix} \otimes \begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix} = 0 \quad 1+0\times0+1\times0+0\times1=0$$

$$= \begin{bmatrix}0 & 1\\ 2 & 0\end{bmatrix}$$

圖7-7　卷積運算示例

在如圖7-7所示的卷積運算中還需要定義幾個術語，方便後面進行講解，如圖7-8所示。

◎　Input：輸入資料。

◎　Kernel：一個矩形區域，以滑動視窗形式在Input上從左到右、從上到下進行點積運算。

◎　Filter：在訊號處理和傳統影像處理中也有Filter的概念，例如Low Pass Filter、High Pass Filter等。在圖7-7中，Kernel所使用的矩陣$\begin{bmatrix}0 & 1\\1 & 0\end{bmatrix}$也就是其使用的Filter。這裡要注意Kernel和Filter的差別。在圖7-7中只有一個通道，其輸入Input實際上是一個維度為[3,3,1]的tensor，Kernel也是一個維度為[2, 2, 1]的tensor。我們可以看到其實輸入Input和Filter都在一個平面上，因此對應tensor的最後一個維度都是1，這時Kernel和tensor實際上是等同的。但如果這時處理彩色圖片，就會有RGB影像，共3個通道（可視為3個平面），那麼Kernel仍然是一個$k \times k \times 1$的tensor，Filter的維度則變成了$k \times k \times 3$，包含3個Kernel。

◎　Stride：在圖7-7中，Kernel的滑動步長為1，例如在第1次點積運算後向右移動一個畫素，再進行運算。這裡就稱Stride為1。為了減少計算量，我們也可以定義Stride為2、3或其他數值，這可以看作對卷積運算輸出的一種下采樣。

◎　feature map：我們通常把Input與Kernel的卷積運算結果稱為feature map，因為我們得到的實際上是由Kernel決定的某些影像特徵。

Stride=1

$$\begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \times \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 2 & 0 \end{bmatrix}$$

Input　Kernel　feature map

圖7-8　卷積運算的相關術語

7.2.2　傳統影像處理中的卷積運算

卷積運算並不是深度學習特有的概念，而是早已被應用在傳統的影像處理中的技巧，例如邊緣檢測、影像模糊等。正因為其在影像處理方面有數十年積累，影像分類才得以成為深度學習第一個真正落地的領域。若想知道卷積神經網路為什麼能成為影像分類的標杆，就需要了解傳統的影像分類方法到底是如何工作的，以及其中存在的問題。如圖7-9所示是一個簡單的傳統邊緣檢測演算法示例[3]。

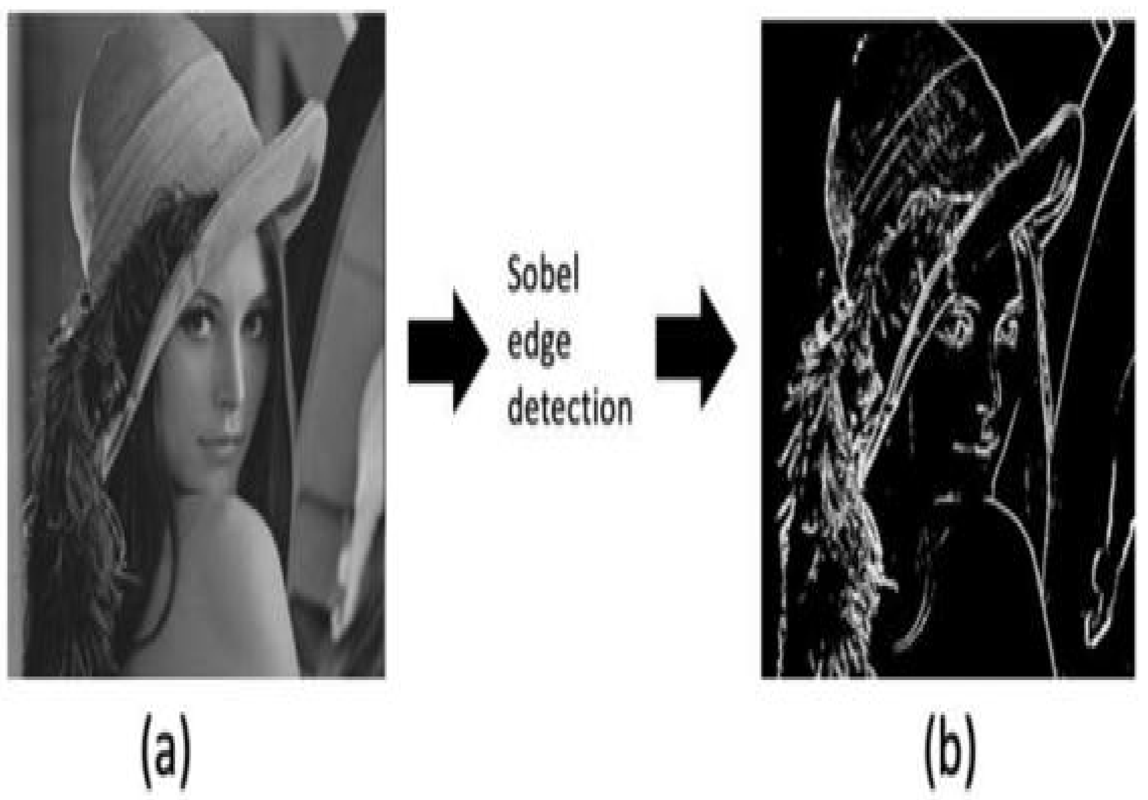


圖7-9　一個簡單的傳統邊緣檢測演算法示例

在圖7-9中所使用的Sobel Operator實際上是兩個3×3的矩陣***A***與***B***：

$$\boldsymbol{A}=\begin{bmatrix}1\\2\\1\end{bmatrix}\begin{bmatrix}-1 & 0 & 1\end{bmatrix}=\begin{bmatrix}-1 & 0 & 1\\-2 & 0 & 2\\-1 & 0 & 1\end{bmatrix}$$

$$\boldsymbol{B}=\begin{bmatrix}-1\\0\\1\end{bmatrix}\begin{bmatrix}1 & 2 & 1\end{bmatrix}=\begin{bmatrix}-1 & -2 & -1\\0 & 0 & 0\\1 & 2 & 1\end{bmatrix}$$

我們定義Input為影像畫素2D矩陣，然後分別與***A***、***B***做前面描述的卷積運算：

$$G_x = \boldsymbol{A} \times \text{Input}$$

$$G_y = \boldsymbol{B} \times \text{Input}$$

我們可以將這裡對Input做的兩個卷積操作視為獲取向右和向下「遞增」的畫素。注意，在***A***的分解矩陣中包含[-1, 0, 1]這樣向右遞增的一維子矩陣，在***B***的分解矩陣中包含$\begin{bmatrix} -1 \\ 0 \\ 1 \end{bmatrix}$這樣向下遞增的一維矩陣。

最後，我們對G_x和G_y中每個對應值的平方和進行運算：

$$G = \sqrt{G_x^2 + G_y^2}$$

這就是最後求出的邊緣值，如圖7-9(b)圖所示。

Sobel運算元的運算過程是依賴了研究人員自身對影像特性、矩陣運算等方面的瞭解，從而設計出的「精巧」計算過程，類似的還有更復雜的Canny運算。在很長一段時間裡，影像領域的研究人員都是憑藉自己的不懈努力，設計出各種精巧的計算方法來解決人臉檢測、影像分類、目標跟蹤等問題的。

然而在實踐中，由於影像本身的複雜性，這些「精巧」的演算法很難在千變萬化的環境下，以及面對大量不同型別的影像時都表現出

較好的效果，而如何把不同的影像處理技巧靈活、高效地「串聯」起來，比如哪裡該做低通（模糊）處理來獲得整體資訊，哪裡該做銳化（邊緣）處理來獲得細節資訊。對這些如果只依靠人們的自身經驗來設計，則也是巨大的難題。

卷積神經網路的誕生，可以使我們透過深度學習方式做到：

◎ 透過使用多個Kernel，解決傳統方案中單一或少數Filter獲取資訊不足的問題；

◎ 透過梯度下降演算法自行學習Kernel的具體數值，從而避免人工設計（例如Sobel Operator的兩個矩陣）。

7.2.3 Pooling

我們在前面把Pooling簡單概括為取樣，這是不夠準確的。Pooling可以透過多種方式實現，在此列舉其中的幾種常見做法。

◎ Max Pooling：取子區域的最大值，子區域通常為一個矩形範圍。

◎ Average Pooling：取子區域的平均值。

◎ Weighted Average Pooling：根據到子區域中心點的距離對該區域內的所有值設定權重並取平均值。

具體採用哪種Pooling方式，要根據情況而定。在實際應用中最常用的是Max Pooling，其原理也非常簡單：

```
for i in range(m):
    for j in range(n):
        max_value = max(max_value, input[i, j])
```

在Keras中則只需要簡單的一行程式碼就可以實現Max Pooling層：

```
MaxPool2D(pool_size=2)(x)
```

這裡沒有必要比較各種不同的Pooling方式，我們更關心的是為什麼要進行Pooling？Pooling層到底解決了什麼問題？

假設我們要識別如圖7-10所示的圖片。在圖7-10中，(a)圖和(b)圖顯然是同一張照片，然而(b)圖相對於(a)圖發生了一些不大的位移。如果我們用傳統的方法去識別圖片，則可能需要獲得貓的眼睛、鼻子、嘴巴等物理特徵，然後搜尋整個圖片，找到單獨的特徵，再對不同特徵的相對位置進行分析。這樣做不但計算耗時非常長，而且無法提高準確率。我們希望能有一種對輕微位移不敏感（Invariant To Small Translation）的影像分類方法。

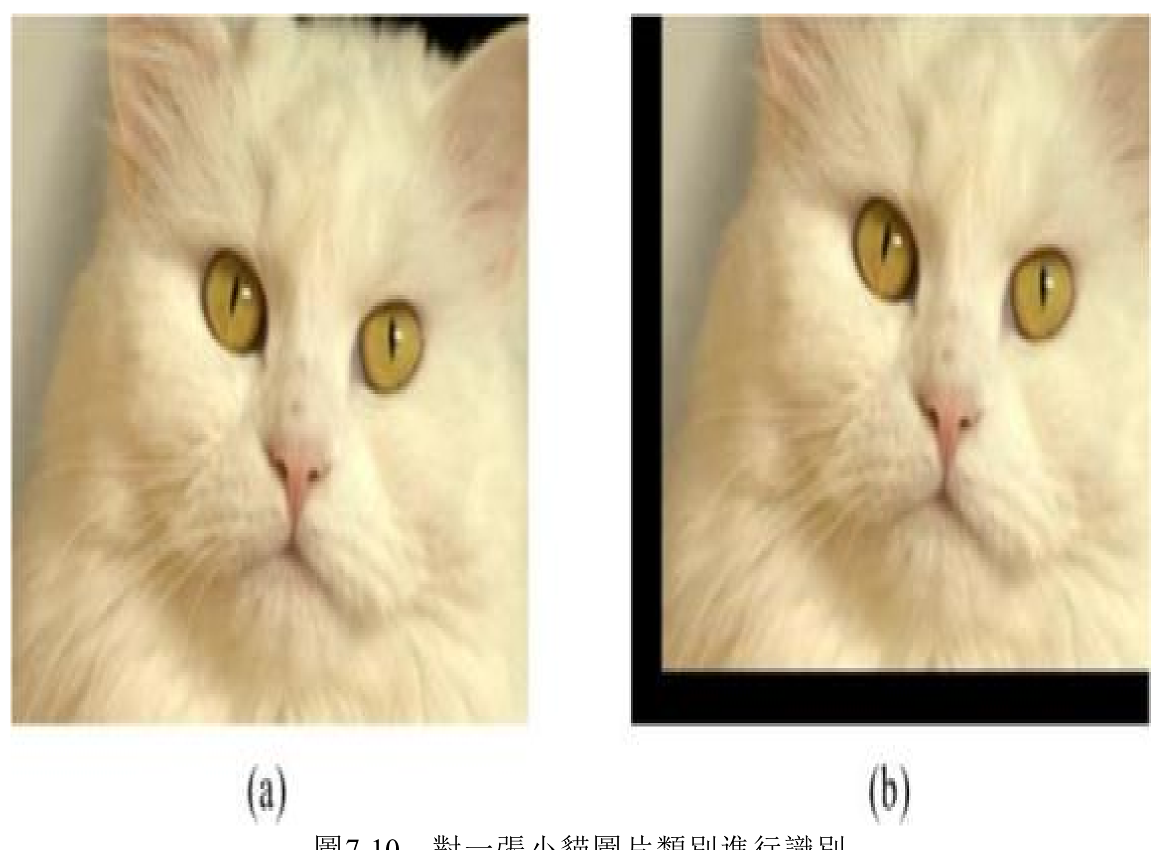
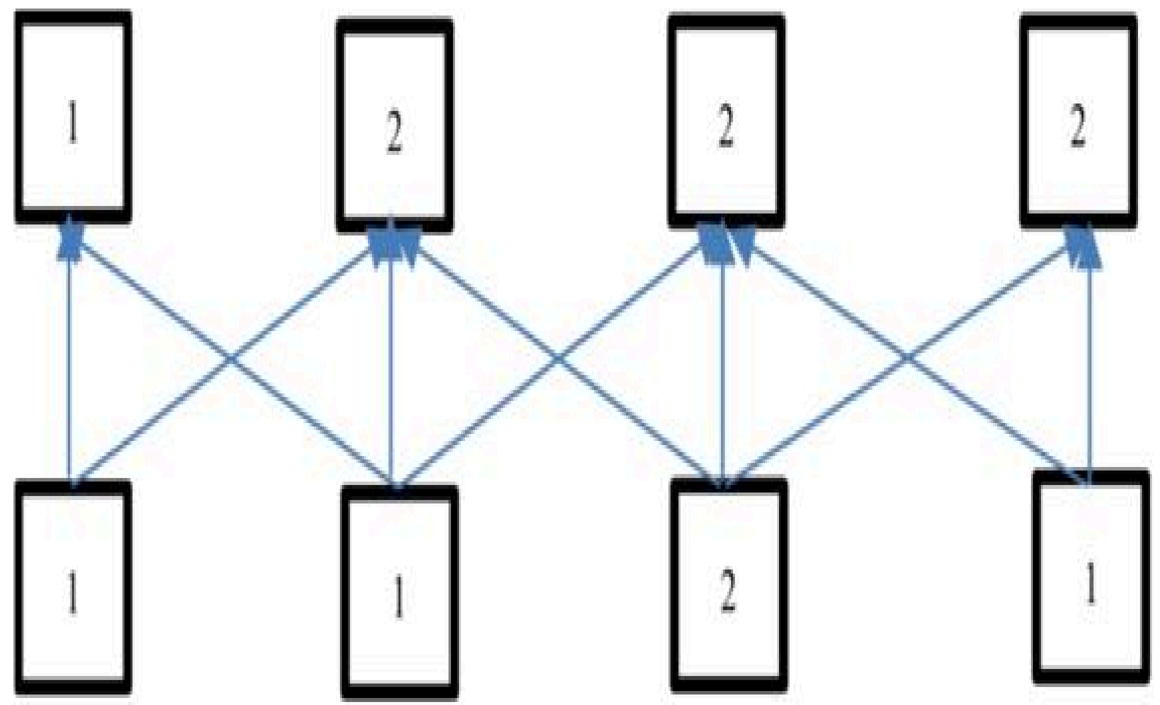


圖7-10　對一張小貓圖片類別進行識別

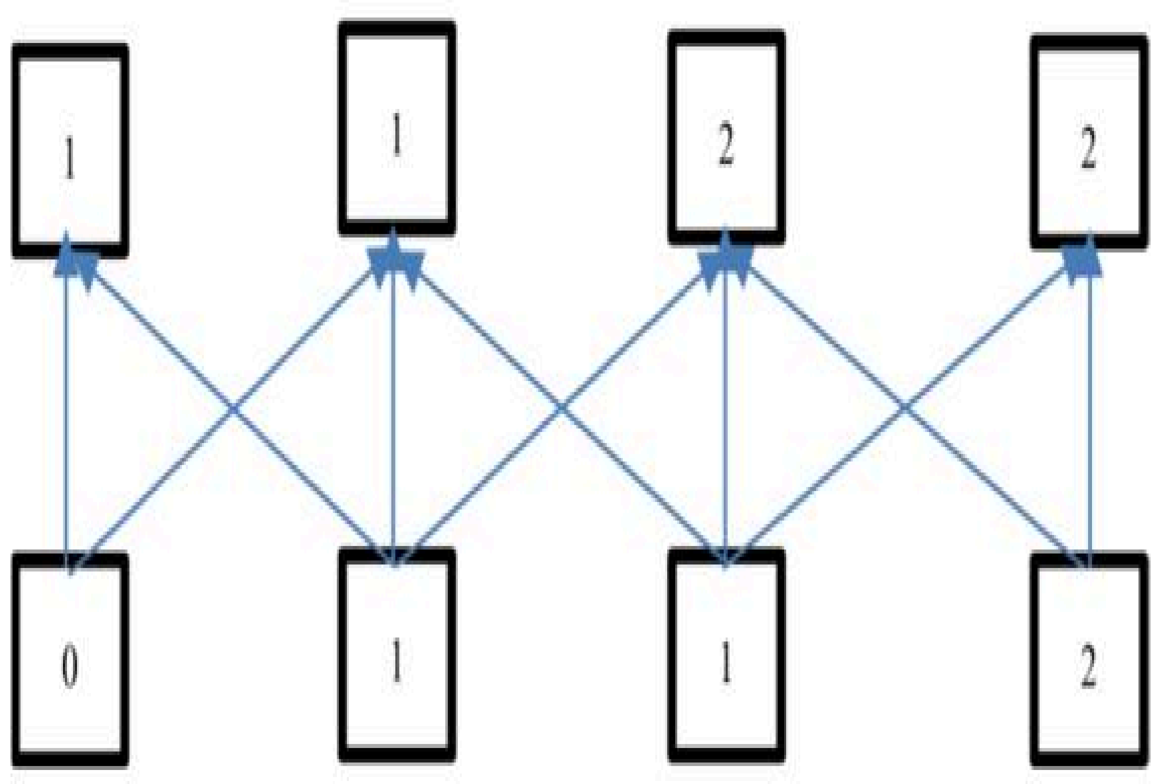
Pooling則能夠達到類似的目的。嚴格地說，Pooling並不是完全對影像特徵的偏移不敏感，而是能保證在輕微偏移的情況下大部分輸出值不變。假設定義一個一維的MaxPooling1D方法：

```
    def MaxPooling1D(input, pooling_size):
        output = []
        for i in range(len(input)):
            max_value = input[i] if i==0 else max(input[i-1], input[i])
            max_value = input[i] if i==len(input)-1 else max(input[i],
input[i+1])
            output.append(max_value)
        return output
```

那麼對於如圖7-11所示的兩個輸入[1,1,2,1]和[0,1,1,2]，我們可以看到輸出雖然會有部分變化，但差異不大，仍具備某種程度的穩定性。



原始输入，MaxPooling1D([1, 1, 2, 1], 3) = [1, 2, 2, 2]



输入向右偏移 1 个像素，MaxPooling1D([0, 1, 1, 2], 3) = [1, 1, 2, 2]

圖7-11　MaxPooling示例

在圖7-11所示的例子中，MaxPooling在輸入的3個相鄰畫素值中選擇了最大的一個。我們可以看到，當原始輸入向右移動1個畫素時，75%的輸出仍然保持一致。

當然，在類似圖7-10這樣的真實圖片中，要做到對影像平移不敏感，並沒有這麼簡單，但原理大體是一致的。在類似圖7-10的圖片中，我們重點關注的是中部區域有貓眼的存在，中下部位置有鼻和嘴的存在，從而達到識別的目的，但我們並不需要獲取它們的精確位置，這正是Pooling所要達到的目標。

實際上，結合卷積運算，Pooling也能對影像的旋轉起到一定作用。在本章參考文獻[2]中給出了一個手寫數字識別的例子，如圖7-12所示。

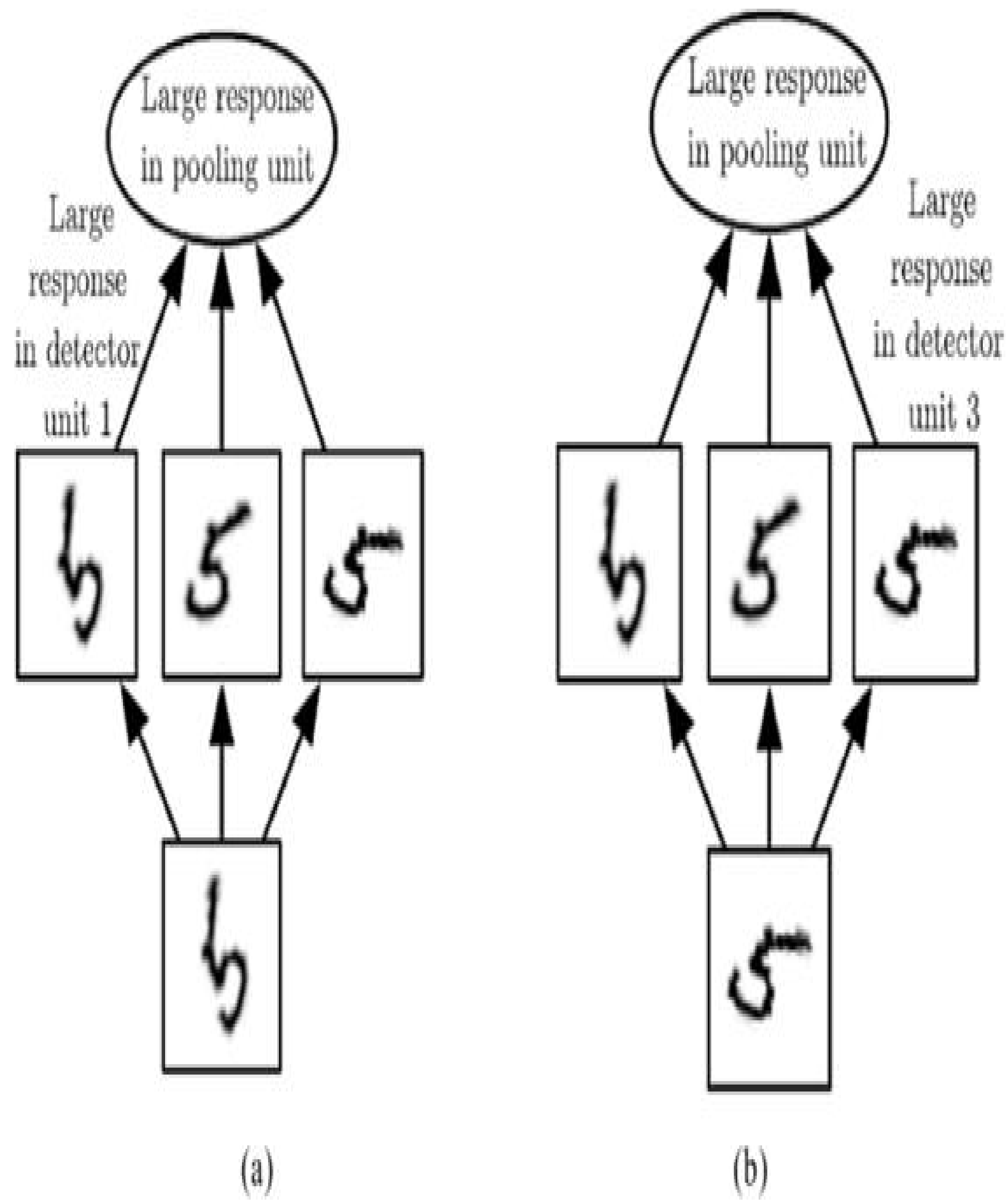


圖7-12　旋轉的手寫數字識別（原圖來自本章參考文獻[2]）

在如圖7-12所示的(a)圖和(b)圖中，我們可以看到透過3個訓練學習到的Filter結合Pooling，能夠做到對旋轉的影像不敏感。在(a)圖和(b)圖中所用的3個Filter都用於匹配手寫數字5旋轉後的不同角度。因此，當輸入手寫數字5時，雖然影像角度不同，但在這3個Filter中總有

一個被啟用（輸出最大）。MaxPooling層並不關心具體哪個Filter被啟用，只需選擇其中最大的一個即可。

7.2.4 為什麼卷積神經網路能達到較好的效果

在7.1.1節和7.1.2節提到了影像分類的幾個問題，例如全連線網路的效能、區域性特徵的識別等，在學習前面的內容後，我們現在可以對這些問題做出回答了。

1.為什麼卷積神經網路能夠比全連線網路更高效?

在回答這個問題時，我們要注意，這裡講的高效並不是簡單地指速度更快或引數更少。的確，使用Kernel函式進行卷積運算後，將運算複雜度從 O(mn)降到了 O(kn)，其中，m為輸出層的節點數量，n為輸入的維度，k為Kernel引數的數量[2]。但是為什麼在引數變少的情況下，它仍然能保持極高的準確率？在本章參考文獻[2]中重點從兩個角度進行了解釋：引數共享（Parameter Sharing）和稀疏連線（Sparse Connectivity）。

首先談談引數共享的意義。在圖7-1中講到了，對於全連線網路，每一層的每個節點的引數都要從上一層的所有輸入中進行處理，其計算量是難以讓人接受的。而透過卷積網路，我們就不需要再對所有輸入都進行處理，需要處理的資料量由Kernel的大小決定。其中的好處在於，因為Kernel是在整個輸入空間中滑動的，所以每一組輸入所使用的Kernel引數都是一樣的，如圖7-13所示。

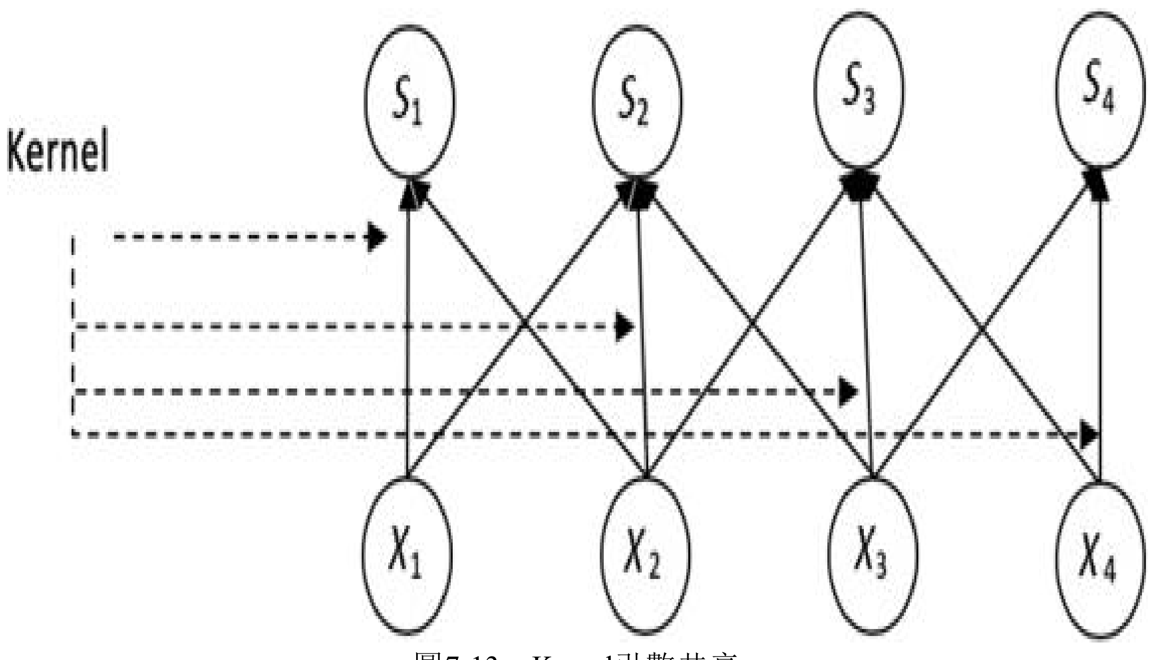
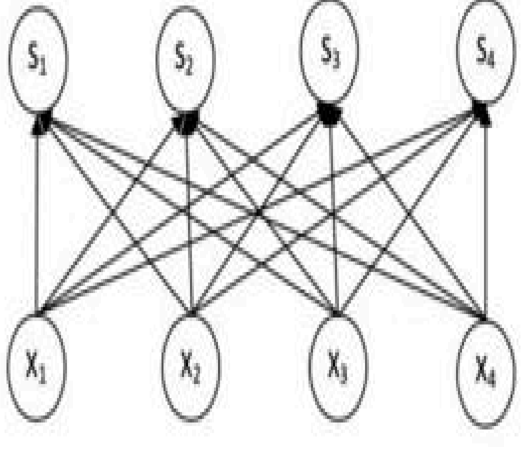


圖7-13　Kernel引數共享

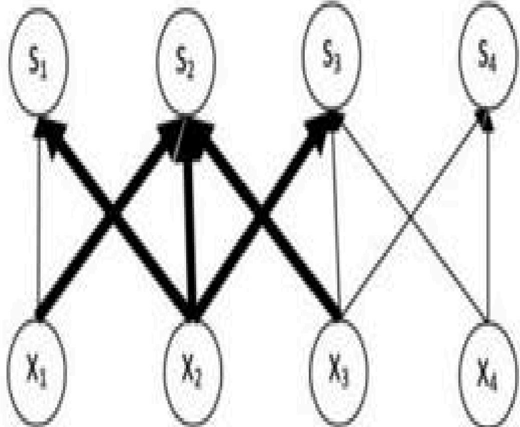
和前面提到的Sobel運算元相比，這裡Sobel運算元所包括的兩個矩陣對所有影像都相同，換句話說，任何應用Sobel運算元的影像都使用同樣的矩陣數值，這和Kernel函式是一樣的。其區別在於，Sobel運算元是依賴人類實驗和經驗設計而得到的，我們在深度學習中則需要透過梯度下降等演算法找到合適的可被多個影像輸入所共享的Kernel引數。

然後，我們來談談稀疏連線。與全連線網路中每一層的每個節點受前一層所有節點的影響不同，在單層卷積神經網路中，第2層的節點只受輸入層和Kernel相關的節點影響，輸入層的節點也只會影響和Kernel大小相關的第2層的節點，如圖7-14所示。

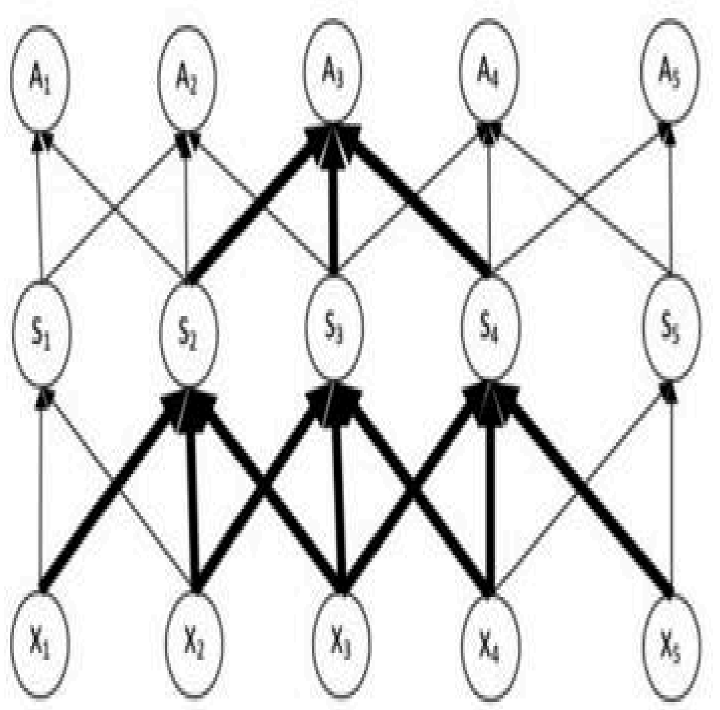
我們在圖7-14(b)圖中看到，對於單層卷積網路，其單個輸出只受區域性輸入的影響，在理論上無法達到較好的效果。然而，當我們把單層卷積網路進行疊加以形成多層卷積神經網路時，如圖7-14(c)圖所示，這時儘管在輸出中A_3仍然只受到上一層的區域性節點S_2、S_3、S_4的影響，但S_2、S_3、S_4又各自受到輸入層X_1、X_2、X_3、X_4、X_5的影響，最終$A3$實際上是由所有輸入節點的共同作用所決定的。同樣，對於每個輸入節點X_i，透過多層網路的傳遞，仍然會影響最終輸出層的每個節點。



(a) 全连接层
第 1 层（输入层）的每个节点都影响下一层的所有节点



(b) 卷积层
S_2只受到X_1,X_2,X_3 3 个输入的影响，同样，X_2的值也只影响S_1,S_2,S_3



(c) 卷积神经网络
S_2只受到X_1,X_2,X_3 3 个输入的影响，同样，X_2的值只影响S_1,S_2,S_3，A_3则受到所有输入的影响

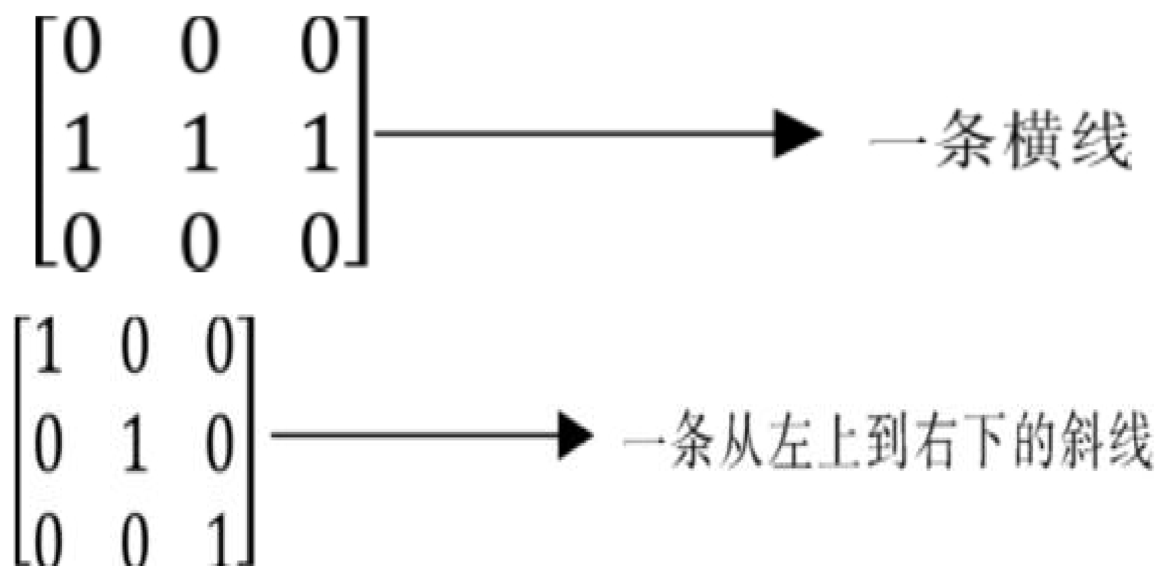
圖7-14　全連線網路和卷積神經網路的對比

因此，透過引數共享和稀疏連線這兩種方式，卷積神經網路才能在大幅度減少運算量和儲存空間的情況下，仍然保持極高的準確率。

2.如何發現區域性特徵？如何處理出現在圖片不同區域的類似特徵？如何處理圖片的縮放？

實際上在前面講解卷積神經網路的工作原理時，已經回答了這3個問題，這裡再歸納一下。

發現區域性特徵，實際上依賴的是Kernel函式，或者說是依賴不同的Filter。我們定義的Kernel函式（Filter），實質上就是對區域性特徵的描述，例如：



我們利用梯度下降來學習這些Kernel函式的引數，實際上就是嘗試自動發現區域性特徵的過程。

如何處理出現在圖片不同區域的類似特徵？卷積運算的整個過程就是用不同的Kernel函式（Filter）對整個影像做滑動視窗的運算，以期發現匹配的部分。配合Max Pooling可更進一步地縮小搜尋範圍，提高匹配的穩定性，從而達到快速處理不同位置的相似特徵的目的。

如何處理圖片的縮放？這在解釋Pooling時已經解釋過。實際上，在產品化的工作中，我們通常規定卷積神經網路的輸入大小必須符合

某個固定的解析度，如果是較大的圖片，則需要先進行縮小處理後再作為輸入。

對卷積神經網路模型的工作原理和優勢就分析到此，下面具體利用Keras實現一個卷積神經網路模型，並用它來真正完成圖片種類識別。

7.3　案例實戰：交通圖示分類

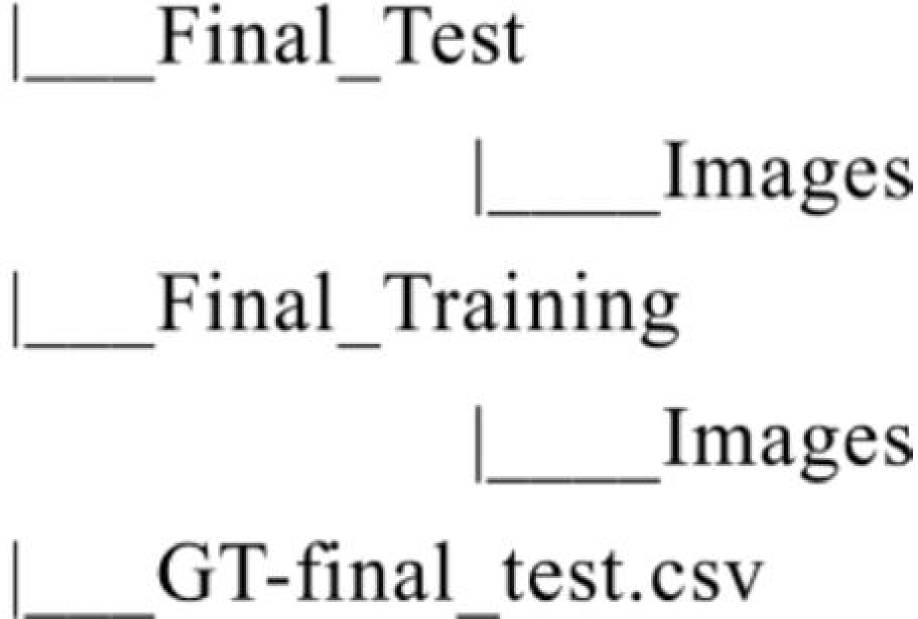
本節將透過一個實際的影像分類示例，來看看如何在具體的資料上基於卷積神經網模型進行程式碼實現，完成訓練及具體的影像類別識別工作。本節將首先介紹相關資料集的準備工作，然後介紹Keras中卷積神經網路模型的具體實現，最後進行訓練和預測程式碼解釋。

7.3.1　交通圖示資料集

我們將使用公開的德國交通圖示資料集 GTSRB[4]，該資料集也是針對影像分類和機器學習研究所使用的公開資料。

從本章參考文獻[4]中下載以下幾組資料：Training Dataset、Images and annotations、Test Dataset、Images and annotations、Extended annotations including class ids。

下載後把檔案按如下目錄組織：



開啟Final_Training/Images，我們看到在其下的每一個子目錄中都帶有大量的相似圖片，例如在子目錄0020中包含右轉道標識的各種圖片，0025包含施工圖識，0030是降雪警告，如圖7-15所示。

圖7-15　代表不同交通標識的訓練樣本

注意，樣本的大小從15×15至250×250不等，同時不保證道路標識就一定處於圖片正中央，這才符合真實場景。

在Final_test目錄下則是用於測試的各種圖片；Gt-final_test.csv是final_test中測試樣本的標籤，開啟後可看到第1行包括各列標題：

```
Filename, width, height, roi.x1, roi.y1, roi.x2, roi.y2, ClassId
```

其中最重要的就是最後一項ClassId，它註明瞭該圖片的類別，也是需要識別的內容。注意，在final_training/images的每一個類別下都有一個同樣格式的CSV檔案，但因為我們已經知道了類別，所以在實際訓練中並不會用到。

7.3.2　卷積神經網路的Keras實現

在Keras中實現卷積神經網路非常簡單，對照圖7-6，我們可以直接構建卷積神經網路模型，如圖7-16所示。

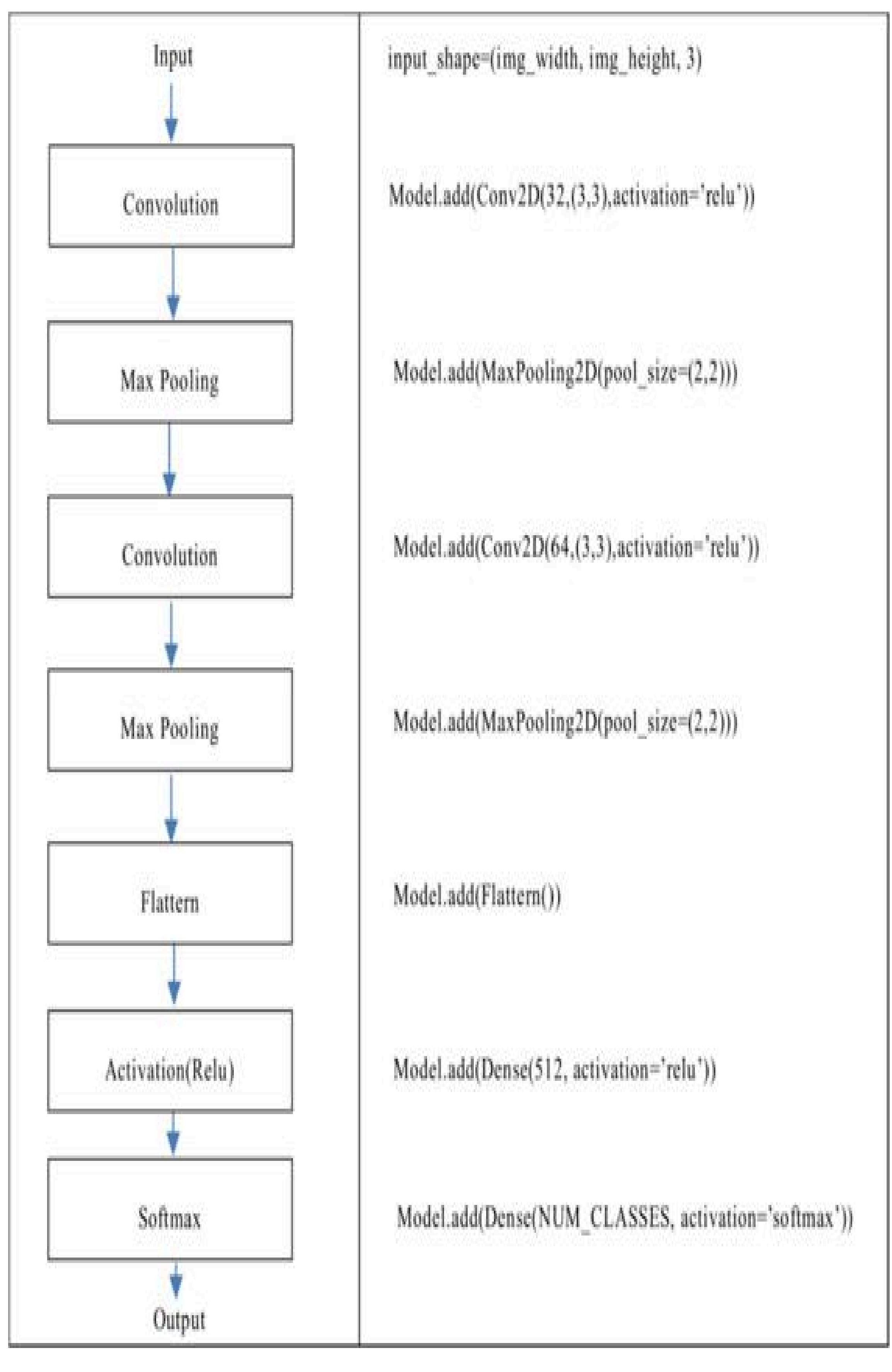

Input
input_shape=(img_width, img_height, 3)
Convolution
Model.add(Conv2D(32,(3,3),activation='relu'))
Max Pooling
Model.add(MaxPooling2D(pool_size=(2,2)))
Convolution
Model.add(Conv2D(64,(3,3),activation='relu'))
Max Pooling
Model.add(MaxPooling2D(pool_size=(2,2)))
Flattern
Model.add(Flattern())
Activation(Relu)
Model.add(Dense(512, activation='relu'))
Softmax
Model.add(Dense(NUM_CLASSES, activation='softmax'))
Output

圖7-16　卷積神經網路模型與Keras對應的程式碼

當然，在如圖7-16所示的程式碼中，只是用Keras部分的程式碼和卷積神經網路模型進行了對照，在實際開發中通常會加入dropout以便提高訓練效率。

我們將用以下3個檔案來實現對交通標誌識別的訓練和預測。

◎　util.py：主要包括卷積神經網路模型的實現。

◎　train.py：卷積神經網路模型的訓練。

◎　predict.py：重建卷積神經網路模型並進行預測。

首先來看看 util.py。為什麼要有一個單獨的檔案來實現卷積神經網路模型？因為在實際生產環境中，訓練程式碼和預測程式碼是不可能放在一起的，那麼AI模型設計人員所建立的模型必然需要同工程部署及後臺開發人員的程式碼共享，因此我們往往把模型的構建單獨作為一個可以分享的模組：

```
1   import numpy as np
2   import os
3
4   from skimage import transform
5
6   from tensorflow.keras.models import Sequential, Model, model_from_json
7   from tensorflow.keras.preprocessing.image import ImageDataGenerator
8   from tensorflow.keras.layers import Dense, Dropout, Activation, Flatten
9   from tensorflow.keras.layers import Conv2D
10  from tensorflow.keras.layers import MaxPooling2D
11
12
13  def preprocess_image(image, size):
14      img = transform.resize(image, (size, size))
15      return img
16
17  def create_model(num_classes, img_size):
18      model = Sequential()
19      model.add(Conv2D(32, (3, 3), padding='same', activation='relu',
20  input_shape=(img_size, img_size, 3)))
21      model.add(Conv2D(32, (3, 3), padding='same', activation='relu'))
22      model.add(MaxPooling2D(pool_size=(2,2)))
23      model.add(Dropout(0.2))
24
25      model.add(Conv2D(64, (3, 3), padding='same', activation='relu'))
26      model.add(Conv2D(64, (3, 3), padding='same', activation='relu'))
```

```
27      model.add(MaxPooling2D(pool_size=(2,2)))
28      model.add(Dropout(0.2))
29
30      model.add(Conv2D(128, (3, 3), padding='same', activation='relu'))
31      model.add(Conv2D(128, (3, 3), padding='same', activation='relu'))
32      model.add(MaxPooling2D(pool_size=(2,2)))
33      model.add(Dropout(0.2))
34
35      model.add(Flatten())
36      model.add(Dense(512, activation='relu'))
37      model.add(Dropout(0.5))
38      model.add(Dense(num_classes, activation='softmax'))
39
40      return model
```

我們看看以上程式碼都做了什麼。

第1～10行：引入相關模組，特別是TensorFlow中的Keras相關模組。

第13～15行：利用skimage將圖片縮放到指定的大小。

第17～40行：構建卷積神經網路模型。與前面的內容相對照，我們看到這裡構建的網路更加複雜。首先，我們有3個大的類似圖7-6的卷積網路，但相對於簡單的convolution→maxpooling，我們實際上對每層實現的是convolution→convolution→maxpooling→dropout。除了對每一個卷積層都應用relu啟用函式，這裡引入的dropout層會在訓練時隨機選取一些節點不進行處理，以提高速度。另外，我們的3個卷積網

路各自使用了不同的Filter個數，從32到64到128，Filter個數成倍增加，目的是獲得更多的圖片細節特徵。

第34～38行：對卷積神經網路進行最後的收尾處理。首先透過Flattern層進行轉換，將其變為一維向量輸入；然後建立了兩個Dense全連線層，第1個層使用relu啟用函式對輸出範圍進行控制，第2個層使用softmax函式獲取每個類別的機率。

然後我們要在train.py中實現具體訓練，具體程式碼如下：

```
1  import numpy as np
2  import glob
3  import os
4
5  from skimage import io
```

```
6   from sklearn.model_selection import train_test_split
7
8   from tensorflow.keras.optimizers import SGD
9
10  from util import preprocess_image, create_model
11
12  def get_label_from_image_path(image_path, data_path):
13      path = image_path.replace(data_path, "");
14      paths = path.split("/")
15      label = int(paths[0])
16      return label
17
18
19  def get_training_data(data_path, num_classes, img_size):
20      images = []
21      labels = []
22
23      all_image_paths = glob.glob(os.path.join(data_path, '*/*.ppm'))
24      np.random.shuffle(all_image_paths)
25      print(data_path)
26      i = 0
27      for image_path in all_image_paths:
28          try:
29              img = preprocess_image(io.imread(image_path), img_size)
30              label = get_label_from_image_path(image_path, data_path)
31              images.append(img)
32              labels.append(label)
33              print("load images: {}".format(i))
34              i = i+1
35          except(IOError, OSError):
36              print("failed to process {}".format(image_path))
37
38
39      X = np.array(images, dtype='float32')
```

```
40      y = np.eye(num_classes, dtype='uint8')[labels]
41
42      return X, y
43
44
45  NUM_CLASSES = 43
46  IMG_SIZE = 48
47
48  TRAINING_DATA_PATH = "./GTSRB/Final_Training/Images/"
49
50  model = create_model(NUM_CLASSES, IMG_SIZE)
51  X, y = get_training_data(TRAINING_DATA_PATH, NUM_CLASSES, IMG_SIZE)
52
53  learning_rate = 0.01
54  sgd = SGD(lr=learning_rate, decay=1e-6, momentum=0.9, nesterov=True)
55
56  model.compile(loss='categorical_crossentropy',
57              optimizer=sgd,
58              metrics=['accuracy'])
59
60  batch_size = 32
61  epochs = 30
62
63  history = model.fit(X, y,  batch_size=batch_size,  epochs=epochs,
64  validation_split=0.2, shuffle=True)
65  model.save('gtsrb_cnn.h5')
```

我們看看以上程式碼都做了什麼。

第1～11行：引入依賴庫，由於我們已經在util.py中完成了卷積神經網路模型構建，所以這裡並不需要引入keras.layers。

第12～42行：讀取訓練資料集。這裡實現了兩個函式get_label_from_image_path和get_training_data。前者從影像路徑名處獲得其對應的Label（類別），這是因為我們的訓練影像檔名類似"./GTSRB/Final_Training/Images/00001/00000_00008.ppm" 的形式，除去前面的路徑，00001就是圖片所屬的類別，所以我們在第12～16行首先去掉了前面的字首路徑，然後把目錄名提取出來轉換為類別編號。第19～42行對訓練目錄下的所有檔案進行遍歷，將圖片進行預處理（縮放到48×48）後存入images陣列，再將對應類別的Label放入labels，最後存入X、y變數中返回。注意，這裡用了Numpy中的eye()函式。numpy.eye(k)本身會生成一個$k \times k$的2維矩陣，其對角線為1，例如，numpy.eye(3) 會生成矩陣$\boldsymbol{A}$：

```
[[ 1. 0. 0.]
 [ 0. 1. 0.]
 [ 0. 0. 1.]]
```

而在第32行中生成的labels是一個一維陣列，假設我們有labels = [1,1,2]，那麼如果使用上面的numpy.eye(3)生成的矩陣$\boldsymbol{A}$，則$\boldsymbol{A}$[labels]則表示根據labels中的每個值從$\boldsymbol{A}$中獲得對應的行，則我們會分別得到3個陣列，每個陣列中的每一列都代表對應的數字（例如第0列代表數字0，第1列代表數字1，第2列代表數字2），因此labels的陣列長度為3，代表[0,2]的整型數值範圍，結果如下：

```
[[ 0. 1. 0.]
 [ 0. 1. 0.]
 [ 0. 0. 1.]]
```

這樣就可以看到，我們獲得的實際上是所有類別的One-hot encoding的值。

第45～48行：定義一些常量，這裡直接設定總類別為43，設定縮放圖片大小為48。

第50行：建立卷積神經網路模型。

第51行：讀入訓練資料，獲得訓練所用的輸入X和Label y。

第53～54行：設定訓練所用的梯度下降最佳化演算法（Optimizer）。注意，梯度下降有多重最佳化實現，這裡不做具體分析。SGD（Stochastic Gradient Descent，隨機梯度下降）是常見的演算法之一（在第3章講過），我們在第54行直接設定該引數即可。

第56～65行：首先透過model.compile對模型的各種引數進行設定，包括損失函式、梯度下降最佳化演算法（設為SGD）及評價標準等；然後設定訓練batch size和epoch次數；最後呼叫model.fit開始訓練（需要一定的時間）。我們會看到類似如下的輸出：

```
Epoch 1/30
31365/31365 [==============================] - 393s 13ms/step - loss: 2.2195
- acc: 0.3499 - val_loss: 0.8014 - val_acc: 0.7436
```

注意，在每個Epoch中，我們會實時看到loss和accuracy值的變化，注意：loss在不斷變小，accuracy在不斷上升。

在執行一段時間後，程式碼會將訓練好的模型儲存到檔案中。

下面看看預測部分的程式碼（predict.py）：

```
1   import numpy as np
2   import os
3
4   import pandas as pd
5   from skimage import io, color, exposure, transform
6   from sklearn.model_selection import train_test_split
7
8   from util import create_model, preprocess_image
9
10  NUM_CLASSES = 43
11  IMG_SIZE = 48
12
13  DATA_PATH = "./GTSRB/Final_Test/Images/"
14
15  def get_test_data(csv_path, data_path):
16      test = pd.read_csv(csv_path, sep=';')
17      X_test = []
18      y_test = []
19
20      i=0
21      for file_name, class_id in
22  zip(list(test['Filename']),list(test['ClassId'])):
23          img_path = os.path.join(data_path,file_name)
24          X_test.append(preprocess_image(io.imread(img_path), IMG_SIZE))
25          y_test.append(class_id)
26          i = i+1
27          print('loaded image {}'.format(i))
28
29      X_test = np.array(X_test)
30      y_test = np.array(y_test)
31
32      return X_test, y_test
33
34  print('start')
35  model = create_model(NUM_CLASSES, IMG_SIZE)
36  model.load_weights('gtsrb_cnn.h5')
```

```
37  test_x, test_y = get_test_data('./GTSRB/GT-final_test.csv', DATA_PATH)
38
39  for i in range(10):
40      x = test_x[i]
41      y = test_y[i]
42      y_pred =  np.argmax(model.predict([[x]]))
43      print("{} : (predict) {}".format(y, y_pred))
44
45  y_pred = model.predict_classes(test_x)
46  acc = np.sum(y_pred==test_y)/np.size(y_pred)
47  print("Test accuracy = {}".format(acc))
```

我們看看以上程式碼都做了什麼。

第1～8行：引入依賴庫。

第10～13行：定義總的類別數量、圖片大小並測試資料路徑。

第15～32行：讀取測試資料，和前面讀取訓練資料的實現幾乎一致，其差別在於類別（Label）資訊不再從圖片路徑中獲取，而是從CSV檔案中獲取。另外，不再使用numpy.eye獲取One-hot encoding型別的標籤。

第34～37行：首先透過前面util.py中的create_model函式建立卷積神經網路模型。和train.py中的程式碼不同的是，這裡不需要重新訓練，只需將在train.py中儲存的權重檔案讀入即可（透過model.load_weights），然後透過前面定義的函式讀入測試資料。

第39～43行：這裡先進行一個小測試，選測試集中前10個樣本看看模型預測效果。注意，在第42行並沒有直接使用模型的預測值model.predict的結果，而是用np.argmax進行了一次轉換。這是因為我們在create_model函式中看到卷積神經網路的最後一層是softmax函

式，也就是說，最後的輸出是一個長度為43（NUM_CLASSES）的一維陣列，其中的每個值都代表一個類別的機率。我們需要在其中選擇最大值，其所代表的類別就是預測結果。執行該部分程式碼，輸出如下：

```
16 : (predict) 16
1 : (predict) 1
38 : (predict) 38
33 : (predict) 33
11 : (predict) 11
38 : (predict) 38
18 : (predict) 18
12 : (predict) 12
25 : (predict) 25
35 : (predict) 35
```

可以看到在10個樣本中，全部資料的預測結果都符合預期。

第45～47行：我們在第45行先直接使用predict_classes來一次性實現對測試集的所有樣本進行預測，然後在第46～47行進行一個完整的準確率統計。執行該程式碼，我們可以看到在12630個測試樣本中預測準確率在97%以上：

```
Test accuracy = 0.9764845605700713
```

7.4　最佳化策略

在上面的簡單卷積神經網路模型的程式碼實現中，我們看到準確率已經超過97%，達到了很好的效果。然而，如果我們還想繼續提升

準確率，那麼怎麼做比較有效呢？

一般來說，可以採用兩種方式提升深度學習模型的效果：資料增強（Data Augmentation）或者模型最佳化。資料增強，指增加更多的有針對性的訓練資料。實際上，深度學習技術之所以在影像識別領域有令人震驚的準確率，很多人都認為這和ImageNet及相關資料集所整理的大規模訓練資料[5]有關。在類似7.3節所使用的交通標識資料的訓練過程中，實際上我們可以透過對所要識別的物體進行平移、縮放、旋轉、亮度、邊緣模糊度等多種手段進行改動，在生成大量的額外資料後再進行訓練，這就是資料增強。

資料增強是工業界和實際專案中最常用也最有效的方式，簡單易行、效果明顯。學術界則追求更有效的演算法模型，希望在較少的資料集上也能有出色的效果。

下面將介紹這兩種方式。首先，引入Keras自帶的資料增強介面，看看在7.3節的基礎上能提升多少效果，然後對近幾年較流行的ResNet做一個簡單介紹，並基於Keras執行和展示。

7.4.1　資料增強

在7.3節的訓練程式碼中，我們可以注意到一共讀入了近4萬張圖片作為訓練樣本。但是，如果我們在create_model()函式的最後加一行model.summary()程式碼，則可以看到整個模型的統計資訊如下：

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 48, 48, 32)        896
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 48, 48, 32)        9248
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 24, 24, 32)        0
_________________________________________________________________
dropout (Dropout)            (None, 24, 24, 32)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 24, 24, 64)        18496
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 24, 24, 64)        36928
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 12, 12, 64)        0
_________________________________________________________________
dropout_1 (Dropout)          (None, 12, 12, 64)        0
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 12, 12, 128)       73856
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 12, 12, 128)       147584
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 6, 6, 128)         0
_________________________________________________________________
dropout_2 (Dropout)          (None, 6, 6, 128)         0
_________________________________________________________________
flatten (Flatten)            (None, 4608)              0
_________________________________________________________________
dense (Dense)                (None, 512)               2359808
```

```
________________________________________________________________
dropout_3 (Dropout)          (None, 512)              0

________________________________________________________________
dense_1 (Dense)              (None, 43)               22059
================================================================
Total params: 2,668,875
Trainable params: 2,668,875
Non-trainable params: 0
```

可以看到，在該模型中共有2 668 875即266萬多個引數需要訓練，相對於如此龐大的引數來說，40000（圖片的數量）就不是什麼太大的數字了。

這裡將使用Keras自帶的ImageDataGenerator來做資料增強。對ImageDataGenerator可以配置多個引數，舉個簡單的例子：

```
X_train, X_val, Y_train, Y_val = train_test_split(X, y, test_size=0.2,
random_state=42)
datagen = ImageDataGenerator(featurewise_center=False,
                             featurewise_std_normalization=False,
                             rotation_range=10.,
                             width_shift_range=0.1,
                             height_shift_range=0.1,
                             shear_range=0.1,
                             zoom_range=0.2,
                             )
datagen.fit(X)
```

在上面的程式碼中，首先透過train_test_split劃分訓練集和測試集，然後在ImageDataGenerator中定義了幾個常用的引數。

◎ featurewise_center：是否讓生成的隨機樣本的每個特性（Feature）均勻分佈，使得mean為0。

◎ featurewise_std_normalization：是否讓隨機樣本的每個特性都符合正態分佈。

◎ rotation_range：生成隨機樣本的旋轉範圍。

◎ width_shift_range：將隨機樣本進行寬度拉伸，可以按比例或畫素值設定。

◎ height_shift_range：將隨機樣本進行高度拉伸。

◎ shear_range：定義隨機樣本相對於原始影像的歪斜程度。

◎ zoom_range：定義隨機樣本的縮放程度。

然後，我們可以用ImageDataGenerator.fit()函式根據輸入X_train自行設定一些內部引數（不一定需要）。

其他程式碼和圖7-16的train.py類似，但要注意：我們現在改成呼叫mode.l.fit_generator()，而不是呼叫 model.fit()來訓練模型。另外，不再直接採用X、y作為訓練資料，而是採用ImageDataGenerator.flow()函式生成增強的訓練資料。調整後的完整程式碼如下：

```
import numpy as np
import glob
import os

from skimage import io
from sklearn.model_selection import train_test_split

from tensorflow.keras.optimizers import SGD
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from util import preprocess_image, create_model

def get_label_from_image_path(image_path, data_path):
    path = image_path.replace(data_path, "");
    paths = path.split("/")
    label = int(paths[0])
    return label


def get_training_data(data_path, num_classes, img_size):
    images = []
    labels = []

    all_image_paths = glob.glob(os.path.join(data_path, '*/*.ppm'))
    np.random.shuffle(all_image_paths)
    print(data_path)
    i = 0
    for image_path in all_image_paths:
        try:
```

```
            img = preprocess_image(io.imread(image_path), img_size)
            label = get_label_from_image_path(image_path, data_path)
            images.append(img)
            labels.append(label)
            print("load images: {}".format(i))
            i = i+1
        except(IOError, OSError):
            print("failed to process {}".format(image_path))


    X = np.array(images, dtype='float32')
    y = np.eye(num_classes, dtype='uint8')[labels]

    return X, y


NUM_CLASSES = 43
IMG_SIZE = 48

TRAINING_DATA_PATH = "./GTSRB/Final_Training/Images/"


model = create_model(NUM_CLASSES, IMG_SIZE)
X, y = get_training_data(TRAINING_DATA_PATH, NUM_CLASSES, IMG_SIZE)

X_train, X_val, Y_train, Y_val = train_test_split(X, y, test_size=0.2,
random_state=42)
datagen = ImageDataGenerator(featurewise_center=False,
                             featurewise_std_normalization=False,
                             rotation_range=10.,
                             width_shift_range=0.1,
                             height_shift_range=0.1,
                             shear_range=0.1,
                             zoom_range=0.2,
                             )
datagen.fit(X)


learning_rate = 0.01
```

```
sgd = SGD(lr=learning_rate, decay=1e-6, momentum=0.9, nesterov=True)

model.compile(loss='categorical_crossentropy',
              optimizer=sgd,
              metrics=['accuracy'])

batch_size = 32
epochs = 30

history = model.fit_generator(datagen.flow(X_train, Y_train,
batch_size=batch_size),
                              steps_per_epoch=X_train.shape[0]/batch_size,
                             epochs=epochs,
                             validation_data=(X_val, Y_val))
model.save('gtsrb_卷积神经网络_augmentation.h5')
```

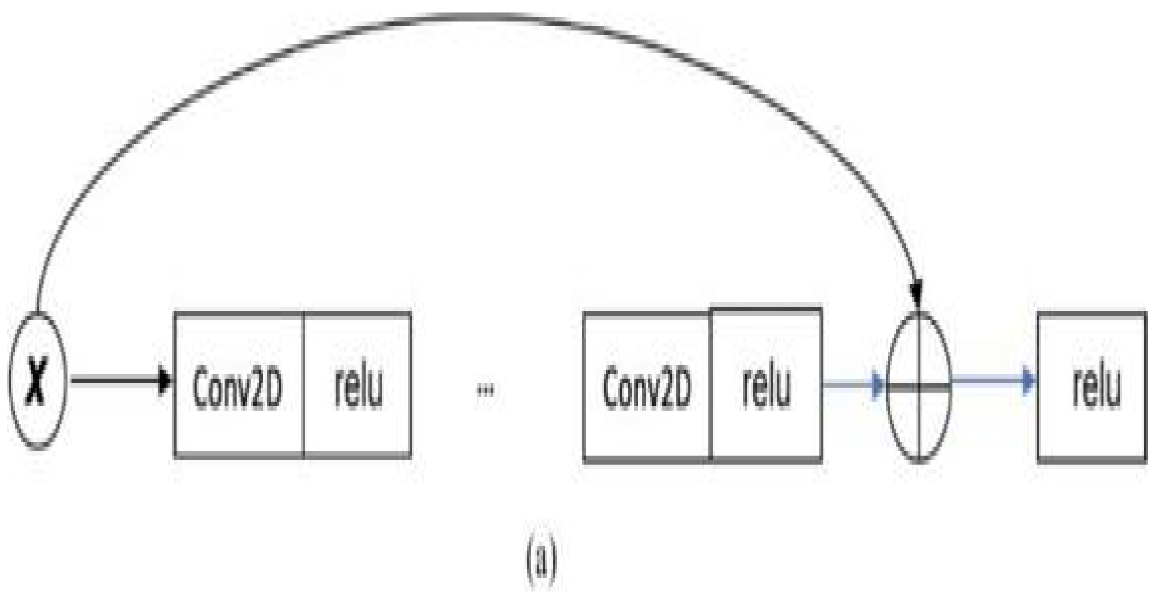
在訓練完成後，修改predict.py中的load_weights()為讀取gtsrb_卷積神經網路_augmentation.h5，執行後可以看到準確率提高了一個百分點，達到98%以上：

```
Test accuracy = 0.9866191607284244
```

7.4.2　ResNet

2015年，微軟研究院提出了ResNet [6]，並獲得當年的ImageNet影像分類冠軍。和標準的卷積神經網路相比，ResNet最大的貢獻在於引入了Skip Connection的概念，使得超深層網路的訓練成為可能。

在7.3節的程式碼中實現了6個卷積層的卷積神經網路，可以看到，僅僅6個卷積層就有266萬以上個引數需要訓練。如果再增加網路層數，則網路的訓練將變得極其困難，同時面臨梯度消失的問題（在第3章中提到，在梯度最佳化中網路層數越大，對誤差求導的結果越小，最終趨於消失）。在本章參考文獻[6]中實現了多達150層網路，其避免梯度消失的問題所依靠的就是Skip Connection的設計，如圖7-17所示。



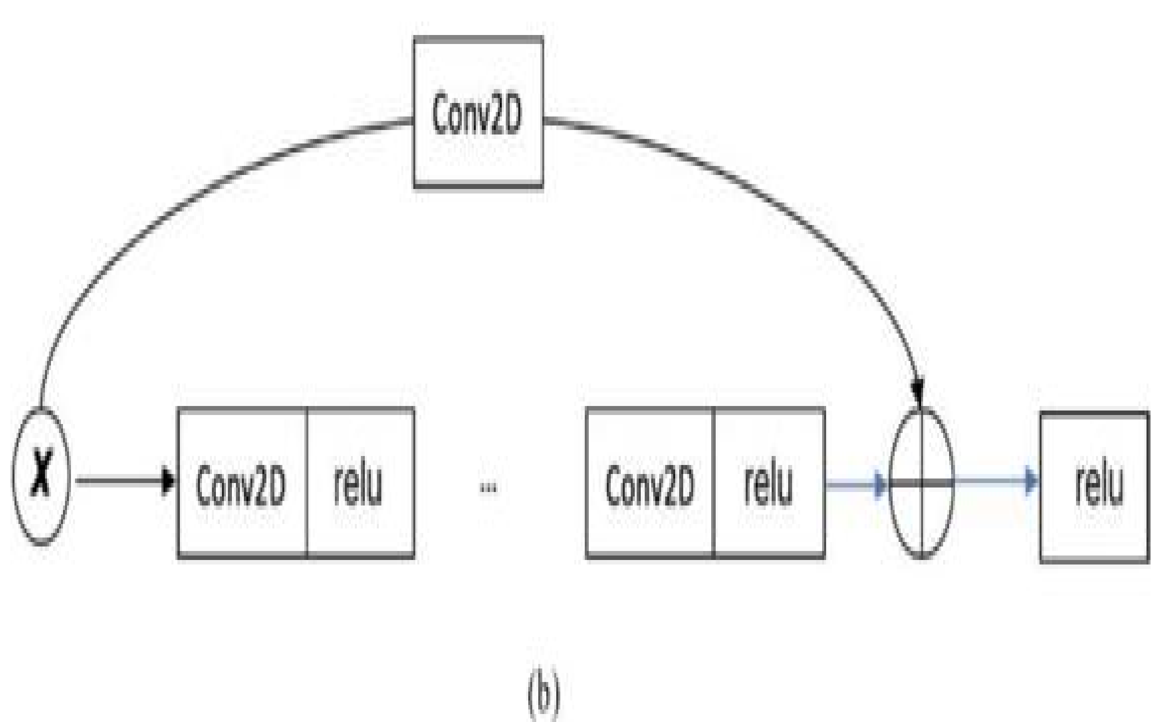


圖7-17　Skip Connection的設計

圖7-17展示了ResNet中Skip Connection的基本概念。我們可以將前面的輸入疊加到後面的輸出中，再把疊加的結果作為下一層的輸入：

```
X_shortcut = X
# 对X进行卷积操作


X=Add()([X, X_shortcut])
```

注意，上面的偽程式碼假設X和X_shortcut是兩個維度相同的矩陣，因此可以直接疊加。如果二者的維度不同，就需要像圖7-17(b)圖中那樣對X_short進行一次卷積操作，使得二者維度相同後再疊加。

這裡不深究Keras中ResNet的實現，在第8章將講到的目標檢測中，我們在最後會講到如何在YOLO模型裡使用Skip Connection實現ResNet。這裡直接使用Keras自帶的ResNet模型看看效果如何。只需在前面的util.py中新增如下幾行程式碼即可實現：

```
41 from tensorflow.keras.layers import GlobalAveragePooling2D
42 from tensorflow.keras.applications import resnet50
43
44 def create_resnet50(num_classes, img_size):
45     base_model = resnet50.ResNet50(weights= None, include_top=False,
46 input_shape= (img_size, img_size, 3))
47     x = base_model.output
48     x = GlobalAveragePooling2D()(x)
49     x = Dropout(0.7)(x)
50     predictions = Dense(num_classes, activation= 'softmax')(x)
51     model = Model(inputs = base_model.input, outputs = predictions)
52     return model
```

我們看看以上程式碼都做了什麼。

第41～42行：引入相關包。

第44行：定義新函式，建立ResNet50。ResNet50是一個較小的ResNet網路。

第45行：建立ResNet基礎模型，我們在這裡選擇weights=0，讓模型從頭開始訓練；也可以設為weights='imagenet'，直接使用ImageNet的訓練權重作為初始值。Include_top指是否在開始加入全連線層。

第47～51行：在基本的ResNet50後加入Global Average Pooling層，這可以看作對全連線層的一個最佳化。這裡不對Global Average Pooling進行解釋，有興趣的讀者可以參考論文*Network in Network*[7]。我們再連線一個dropout層，最後和之前的卷積神經網路一樣，以一個softmax啟用函式得到類別機率。

其他程式碼則和前面的train.py/predict.py差異不大，讀者可自行調整和執行。注意，ResNet在CPU機器上的訓練時間極長，讀者在執行程式碼時需要保持較長時間的耐心。

7.5 本章小結

從本章開始，我們進入計算機視覺領域，並集中在影像分類問題上。本章首先討論了影像分類本身的特點，把焦點集中在為什麼卷積神經網路對影像分類如此有效的問題上，然後圍繞這個問題探討了卷積神經網路模型的實現細節和原理，以及一些重要的相關概念。7.3節使用德國交通標識集基於Keras實現了一個對應的卷積神經網路模型，從資料集中讀取近4萬張圖片進行訓練並驗證，準確率高達97%以上。7.4節從資料增強及模型最佳化兩方面進一步提高識別準確率。在資料增強上，我們引入了Keras自帶的ImageDataGenerator，並在前面程式碼的基礎上進行了修改，以具體的例子講解了ImageDataGenerator的使用方法，其準確率提高到了98%以上，並針對模型最佳化介紹了ResNet模型，包括其原理、核心概念，以及如何在Keras中使用預定義的ResNet50模型。

影像分類是深度學習的重要領域，同時是深度學習爆發的開端。我們將在第9章進入應用最廣泛、也最引人關注的領域：目標識別。

7.6 本章參考文獻

[1] "IJCNN 2011 Competition result table".OFFICIAL IJCNN 2011 COMPETITION.2011

[2] "Deep Learning", MIT Press, 2016

[3] Sobel Operator, https://en.wikipedia.org/wiki/Sobel_operator

[4] GTSRB, http://benchmark.ini.rub.de/?section=gtsrb & subsection=dataset

[5] "ImageNet: Constructing a large-scale image database", Fei fei Li, Jia Deng, Kai Li, 2009

[6] "Deep Residual Learning for Image Recognition", Kaiming He, X.Zhang, S.Ren, J.Sun, Microsoft Research, 2015

[7] "Network in Network", Min Lin, Q.Chen, S.Yan, 2013

第8章　目標識別

在本章中，我們將進入深度學習最引人矚目的領域：目標識別（Object Detection）。

不言而喻，目標識別是當前人工智慧應用最重要的環節之一。從無人機自動跟隨到無人車駕駛，從醫學影像到安全監控，目標識別都是其中的關鍵環節。因為目標識別的應用日趨廣泛，所以各種應用場景和對不同裝置的需求也促使其不斷改進和演化，最新論文的成果往往被快速用於實際產品，這也是學術研究和實際應用結合最緊密的領域之一。

本章將從基本的卷積神經網路（後統稱CNN）講起，引入深度學習目標識別的關鍵概念，然後分別對當前流行的兩種目標識別方式進行介紹並輔以實際應用案例。

讓我們馬上開始吧！

本章可能是全書內容最複雜的一章，因為目標識別本身就是深度學習最具挑戰性的領域。讀者應該主要在流程和概念上理解其中的重點演算法思想，不要陷入程式碼的大量向量處理細節中。本章Faster RCNN部分的核心程式碼可供讀者閱讀和理解，YOLO部分的程式碼可供讀者在本地執行和除錯。

8.1　CNN的演化

8.1.1　CNN和滑動視窗

第7章講到了CNN在圖片分類上的應用，那麼我們能不能直接用CNN來做目標識別呢？

與圖片分類不同，對於目標識別，我們並不是簡單地判斷該圖片包含什麼物體，而是更進一步地輸出該物體的位置及寬高。當在圖片中包含多個物體時，我們要能夠判斷每個物體的位置和寬高，如圖8-1所示。

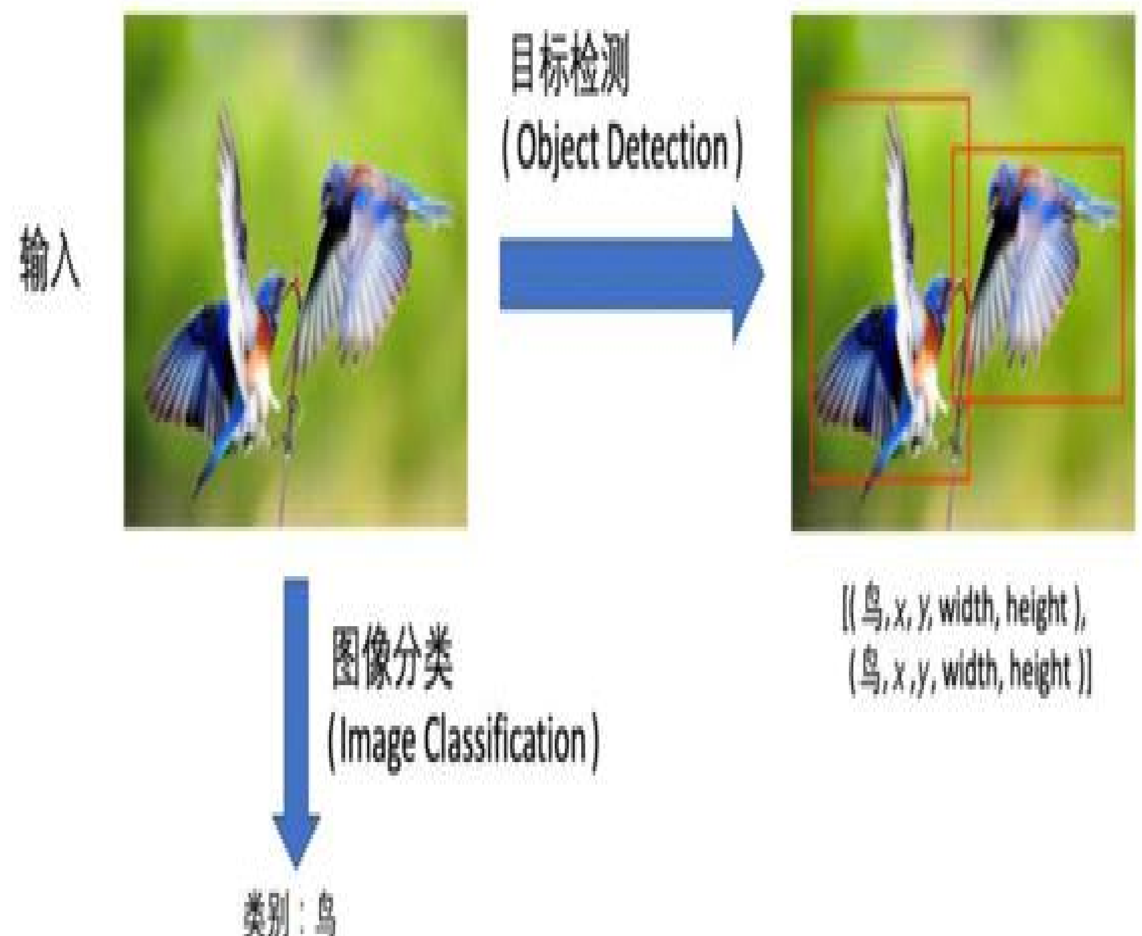


圖8-1　圖片分類vs目標檢測

從圖8-1可以看到，對於影像分類，我們只需要對影像中的主要物體進行分類即可。而對於目標識別，首先要能夠識別圖片中可能存在的所有目標，而所有目標的數量是不確定的。另外，對每個目標不但要找到其類別，還要找到其在圖上的位置和大小。因此，直接使用影像分類的方法如CNN等，對目標識別是不適用的。

但是，我們可以做一點小小的修改，使用滑動視窗將畫面分成無數小區域，對每個區域都用CNN進行分類，從而識別可能存在的不同位置的物體，如圖8-2所示。

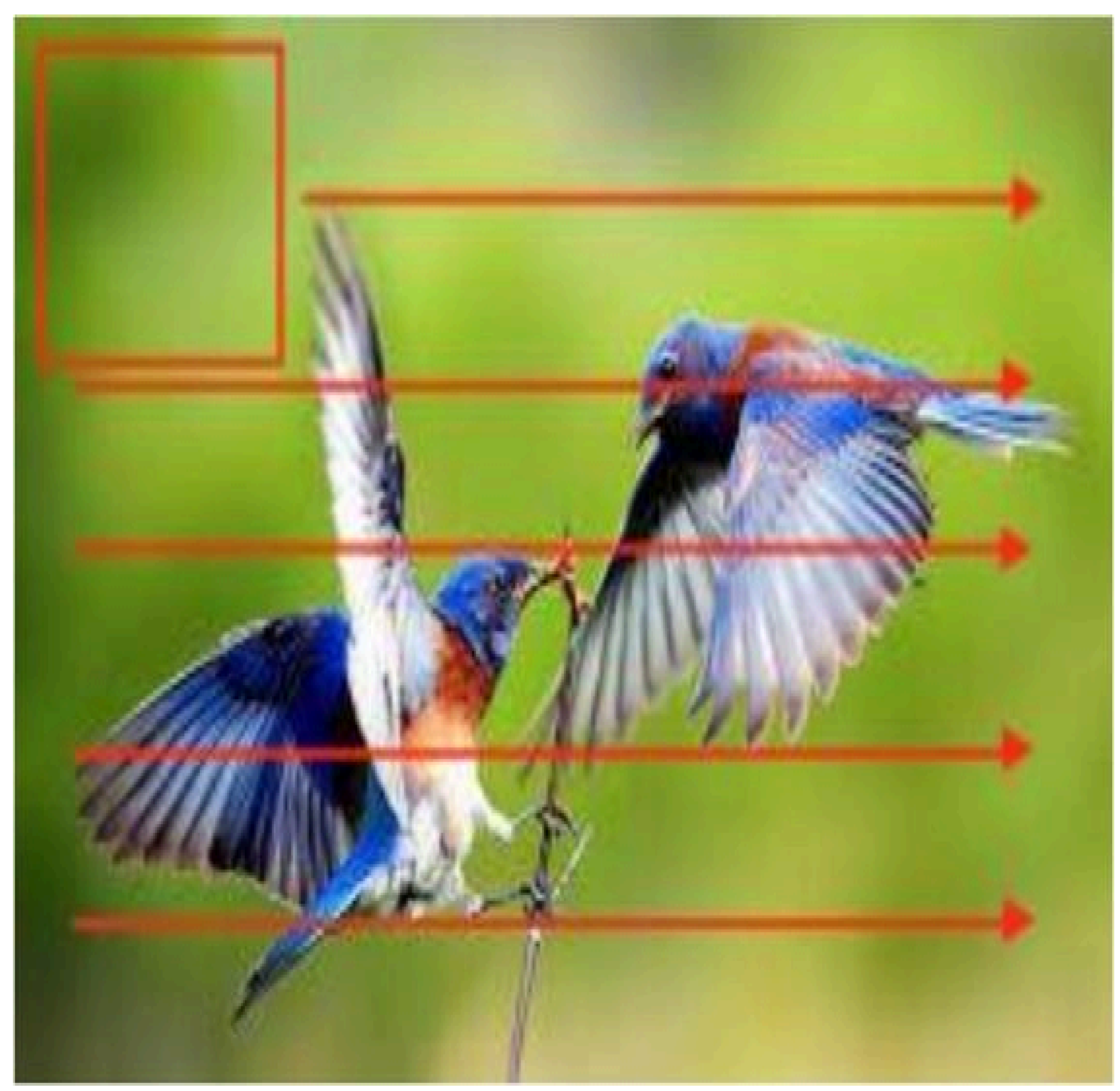


圖8-2　採用滑動視窗（Sliding Window）進行識別

然而，實際畫面上的物體大小會不一樣，我們很難透過一個合適的區域大小來找到準確的目標。當然，我們可以進一步使用不同大小的視窗來處理不同大小的物體，但這樣就會變得極其緩慢，這是無法在實際場景中投入使用的。為瞭解決這個問題，Girshick等人於2014年在*Rich feature hierarchies for accurate object detection and semantic segmentation*[1]一文中提出了RCNN演算法。

8.1.2　RCNN

RCNN的實現流程總體可分為4步，如圖8-3所示。

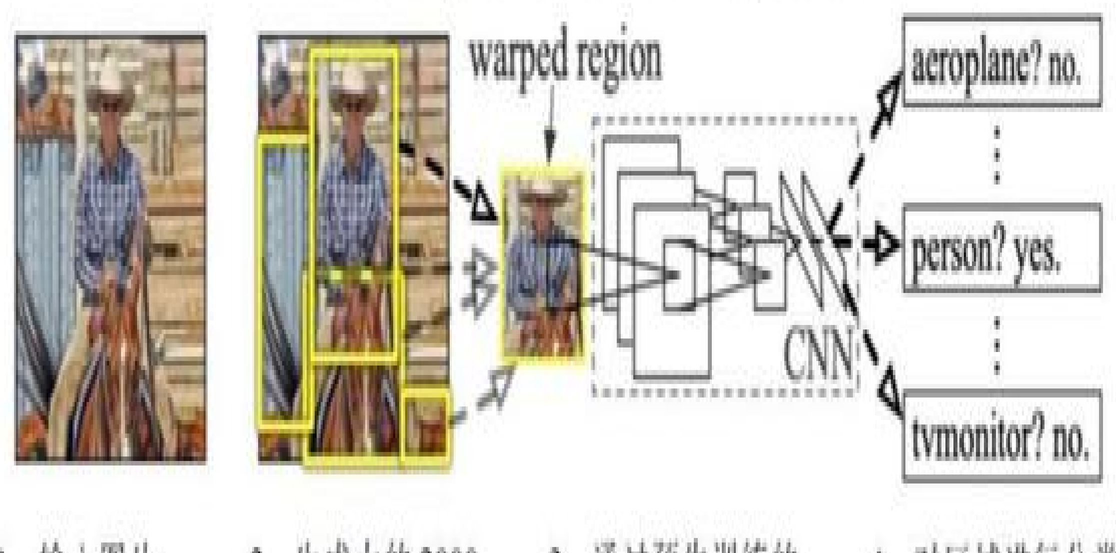


圖8-3　RCNN的實現流程[1]

第1～2步，RCNN對圖片使用在*Selective Search for Object Recognition*[2]中描述的選擇性搜尋（Selective Search）方式生成大約2000個推薦區域（Region Proposal）；然後對每個區域都進行CNN分類。其中的選擇性搜尋指首先根據在*Efficient Graph-Based Image Segmentation*[3]中所描述的演算法將圖片劃分成大量的小區域，然後綜合圖片的色彩、紋理（SIFT資訊）、區域大小、相互覆蓋程度等資訊將小區域合併，最後獲得最合適的2000個左右推薦區域。

在第2步獲取不同的推薦區域後，會對每個區域都進行一個變形，讓它成為一個正方形的影像，然後進行第3步。第3步透過預先訓練的CNN獲得一個feature vector。在第4步的分類中，實際上是用一個SVM（Support Vector Machine，支援向量機）二分類模型來完成的。這個SVM二分類模型需要對每個類別都進行單獨訓練。

在圖8-3的4個主要步驟後實際上還有對邊界框（Bounding Box）的修正過程，因為在第2步中提出的推薦區域未必就是目標的真實位置。這一步是透過邏輯迴歸完成的，下面會進行簡單介紹。

如圖8-4所示，設虛線區域$P = (p_x, p_y, p_w, p_h)$為預測區域（P代表Prediction，也就是上面第2步透過選擇性搜尋給出的推薦區域），其中，(p_x, p_y)是區域中心，p_w和p_h分別是寬和高；實線區域是目標的真實區域（Ground Truth），其區域定義為$G= (g_x, g_y, g_w, g_h)$。

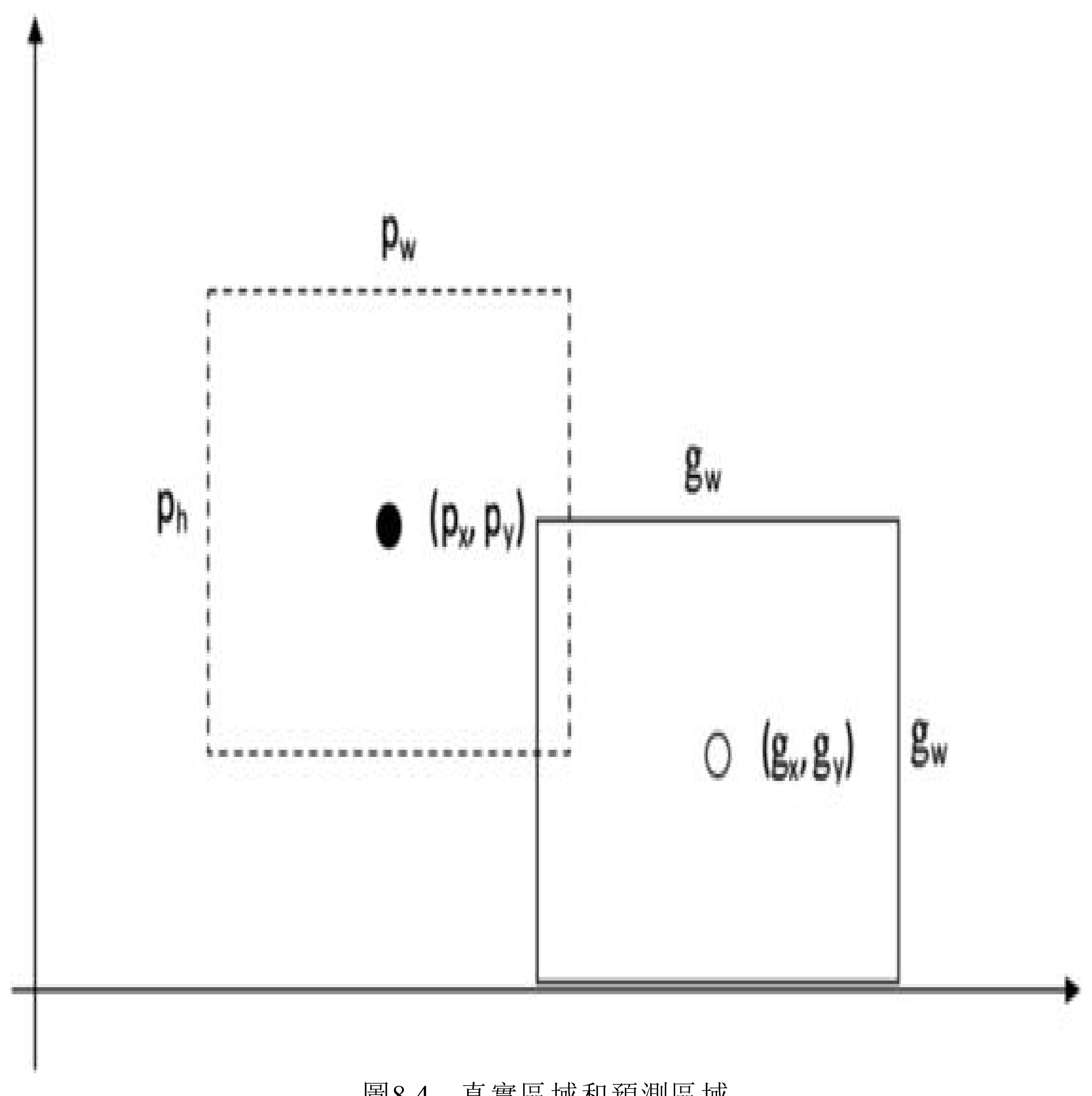


圖8-4　真實區域和預測區域

如果我們把P看作一組長度為4的輸入引數，則可以預設真實區域和預測區域存在以下關係：

```
gx = px + Dx = px + pw * dx(P)
gy = py + Dy = py + ph * dy(P)
gw = pw * Rw = pw * exp(dw(P))
gh = ph * Rh = ph * exp(dh(P))
```

於是可以得到target (t_x, t_y, t_w, t_h)：

```
dx(P) = (gx - px)/pw = tx
dy(P) = (gy - py)/ph = ty
dw(P) = log(gw/pw) = tw
dh(P) = log(gh/ph) = th
```

我們在後面談到Faster RCNN的程式碼實現[8]時，會看到上面的邏輯是如何被具體應用於影像區域預測的，這裡可以看看上面計算t_x、t_y、t_w、t_h的具體程式碼：

```
def apply_regr_np(X, T):
    try:
        x = X[0, :, :]
        y = X[1, :, :]
        w = X[2, :, :]
        h = X[3, :, :]

        tx = T[0, :, :]
        ty = T[1, :, :]
        tw = T[2, :, :]
        th = T[3, :, :]

        cx = x + w/2.
        cy = y + h/2.
        cx1 = tx * w + cx
        cy1 = ty * h + cy

        w1 = np.exp(tw.astype(np.float64)) * w
        h1 = np.exp(th.astype(np.float64)) * h
        x1 = cx1 - w1/2.
        y1 = cy1 - h1/2.

        x1 = np.round(x1)
        y1 = np.round(y1)
        w1 = np.round(w1)
        h1 = np.round(h1)
        return np.stack([x1, y1, w1, h1])
    except Exception as e:
        print(e)
        return X
```

在以上程式碼中，我們看到輸入的X是預測的影像區域資料，T是真實區域，其具體實現和前面討論的內容是一一對應的（注意，在程式碼中，tx、ty、tw、th指的是真實區域的x、y、w、h，最後的輸出是x1、y1、w1、h1，在變數命名上有所差別，但原理一樣），以後不再贅述。

這樣就得到了目標函式（Target Function），我們用t_i（分別對應上面的t_x、t_y、t_w、t_h）來表達，然後可以用簡單的SSE作為損失函式：

$$L = \sum_{i \in \{x,y,w,h\}} (t_i - d_i(P))^2 + \lambda \|w\|^2$$

這裡要注意，並不是所有預測到的影像區域都有對應的真實區域（比如預測錯誤的時候）。一般來說，二者的重合率至少要在0.6以上才能用上面的演算法來訓練。

RCNN無疑是一次巨大的進步，並有相當不錯的效果。但它存在的問題也很明顯：

◎　需要對每張圖片都進行選擇性搜尋，獲得2000個推薦區域；

◎　需要分別對2000個推薦區域執行CNN來獲得特徵向量；

◎　需要訓練額外的3個模型：每個類別的CNN模型、每個類別的SVM二分類模型，以及修正影像區域的邏輯迴歸模型。

在RCNN被提出一年後，針對以上問題，Fast RCNN和Faster RCNN緊接著被提出。

8.1.3　從Fast RCNN到Faster RCNN

二者仍然是由RCNN的作者和其在Facebook的同事一起提出的，分別被髮表在*Faster R-CNN: Towards real-time object detection with region proposal networks*[5]和*Mask R-CNN*[6]兩篇文章中。

首先來談談Fast RCNN。Fast RCNN解決的主要是上面提到的需要分別對2000個區域單獨進行CNN運算的問題。它直接對整張圖片進行CNN運算，然後根據選擇性搜尋的2000個區域，把原圖的「推薦區域」對映到CNN運算結果的對應區域中（這一步被稱為RoI Pooling），這樣便可以直接使用全圖CNN的運算結果，而無須對每個區域都進行重複運算。

從圖8-5可以看到，透過直接對全圖進行一次CNN運算後得到的feature map進行RoI對映（RoI Projection），能夠避免分別對2000個區域都進行CNN運算，從而大幅度提高運算速度。RoI對映實際上指首先定義一個小的視窗區域(r, c, h, w)，其中，(r, c)是左上角座標，(h, w)是高和寬；然後把卷積網路輸出中大小為(H, W)的feature map劃分為$H/h \cdot W/w$個子視窗，並對每個子視窗做Max Pooling處理，最後接入一個全連線網路。

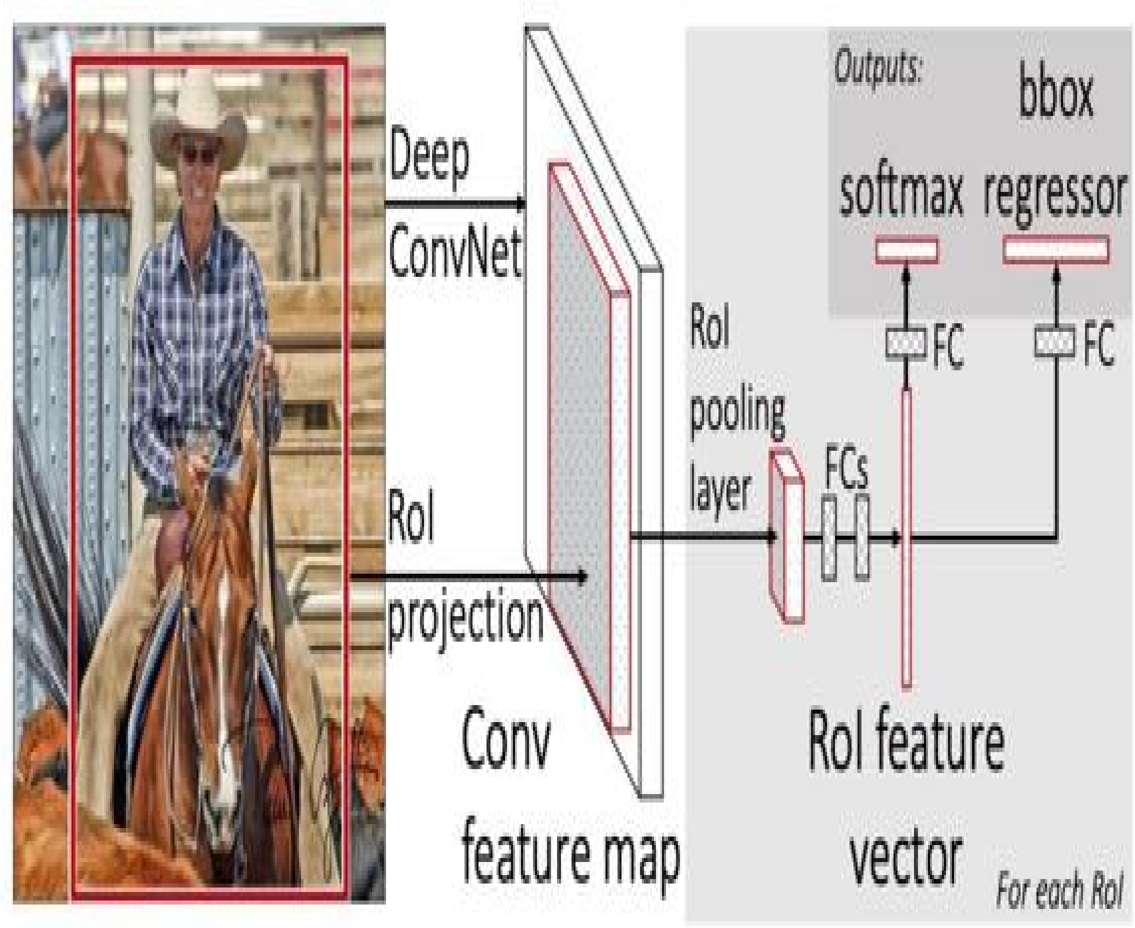


圖8-5　Fast RCNN的工作流程

歸納起來，Fast RCNN的實現步驟如下。

（1）訓練識別影像類別的CNN模型。

（2）使用選擇性搜尋建立2000個左右備選區域。

（3）根據備選區域，將CNN模型最後的Max Pooling層替換為RoI Pooling 層（這是一個非標準層，Keras目前沒有標準介面實現）。

（4）將CNN的softmax層的K個分類改為K+1個分類（因為會有一個不包括任何目標的類別）。

（5）最後會有兩個輸出：類別和影像區域。

我們注意到，在圖8-5的最後輸出中包含兩部分內容：表示分類結果的softmax輸出；影像區域的迴歸模型輸出。實際上，我們在訓練時需要定義的損失函式就是二者的結合：

$$L = L_{\text{class}} + L_{\text{location}}$$

對Fast RCNN損失函式在此不做具體推導，感興趣的讀者可以參考本章參考文獻[5]。

我們再回頭看一看在8.1.2節提到的RCNN存在的問題。既然Fast RCNN解決了重複進行CNN及SVM運算的問題，那麼還剩下一個問題：是否需要進行選擇性搜尋來設定2000個左右「推薦區域」？

我們自然希望用深度學習的方式來處理，最好能融入圖8-5所示的工作流程中，而無須進行額外的運算，實現端到端的模型。因此，Faster RCNN[6]應運而生。

Faster RCNN對Fast RCNN的關鍵改進，就是用被稱為RPN（Region Proposal Network）的網路層取代了RCNN和Fast RCNN所使用的選擇性搜尋，如圖8-6所示。

在圖8-6中，輸入仍然是對圖片本身做卷積運算得到的feature map，這和Fast RCNN一致。然後，我們在它上面應用一個3×3的滑動視窗，將視窗的中央位置稱為錨點（Anchor）。

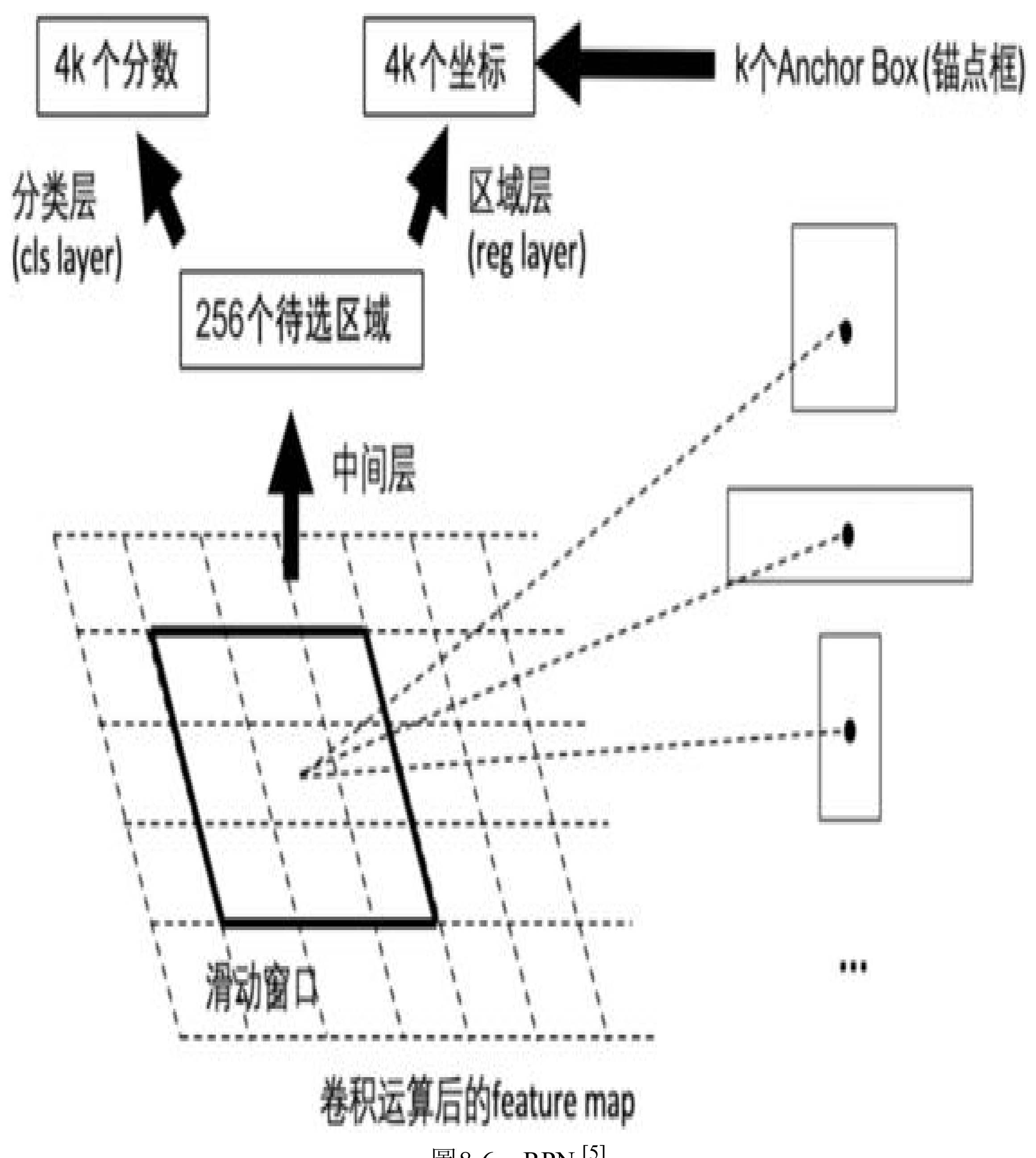


圖8-6　RPN [5]

錨點框（Anchor Box）就是定義了以該點為中心的不同大小和不同比例的長方形視窗。在本章參考文獻[5]中使用了3種比例（1∶1、1∶2、2∶1）和3種尺寸（128、256、512），那麼對每個錨點就有9（3×3）個錨點框，如圖8-7所示。

那麼對於一個40×60的feature map，我們會得到40×60×9=21 600個錨點

框，21 600是個相當大的數字，我們需要做一些處理。在實際應用中，我們會做以下兩步處理：

（1）移除邊緣區域的視窗，例如中心點在$(0, j)$和$(i, 0)$位置的視窗，因為這些滑動視窗包含了空白區域。

（2）應用NMS（Non-Max Suprression），也就是將所有錨點框中IoU（Intersection over Union）不超過0.7的視窗去掉。IoU指的是預測區域和真實區域的交集與二者總面積的比例，即

$$\mathrm{IoU} = \frac{\mathrm{Area}_{\mathrm{predict}} \cap \mathrm{Area}_{\mathrm{ground\ truth}}}{\mathrm{Area}_{\mathrm{predict}} \cup \mathrm{Area}_{\mathrm{ground\ truth}}}$$

一個簡單的例子如圖8-8所示。

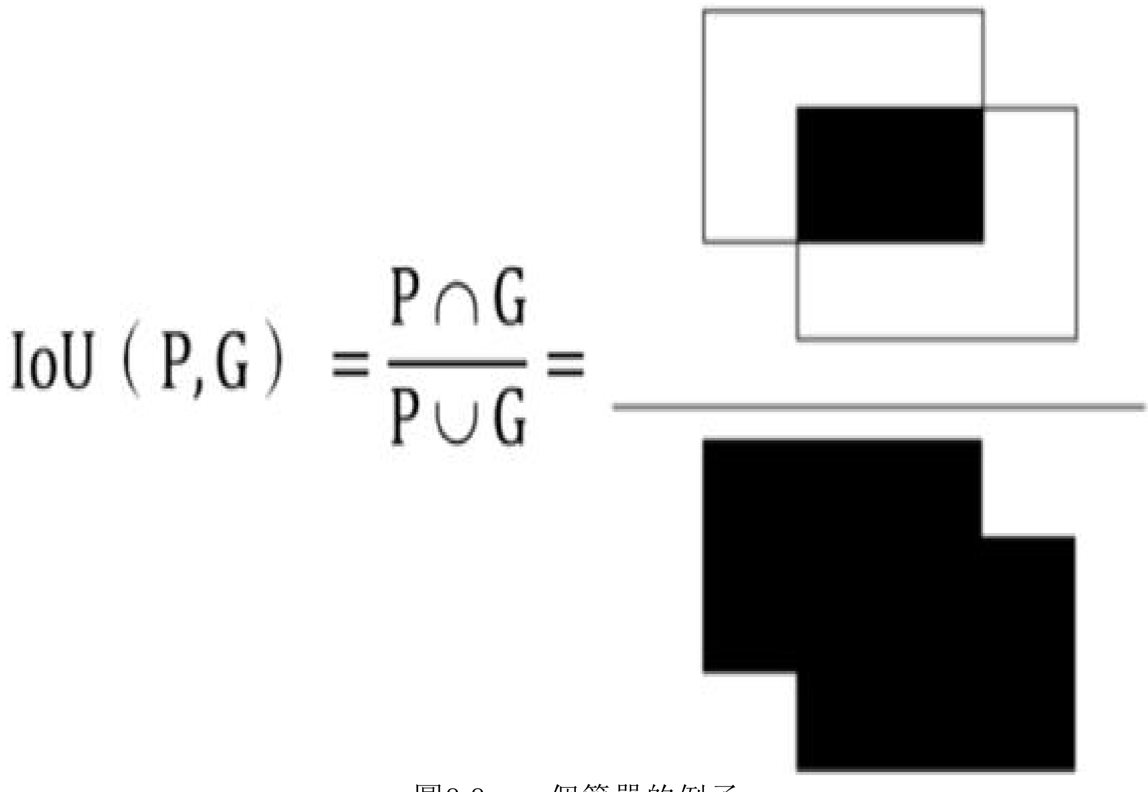


圖8-8　一個簡單的例子

透過上述兩步，我們基本上可以把可用的錨點框控制在2000個左右。

回到圖8-6，在得到錨點框後，我們並不是每次都把所有的視窗都用於下一步運算，而是在每一個epoch中隨機挑選256個正樣本（IoU>0.7）和256個負樣本（IoU<0.3）作為mini batch資料，然後進入下一步。

下一步其實很直接，我們把在上面得到的樣本輸入一個全連線網路中，這個網路包含以下6個輸出（以One-hot encoding vector的形式）。

◎　P_{obj}：包含目標的機率。

◎　$P_{not\text{-}obj}$：不包含目標的機率。

◎　x：預測影像區域的x座標。

◎　y：預測影像區域的y座標。

◎　w：預測影像區域的寬。

◎　h：預測影像區域的高。

我們可以將以上步驟梳理成圖8-9。

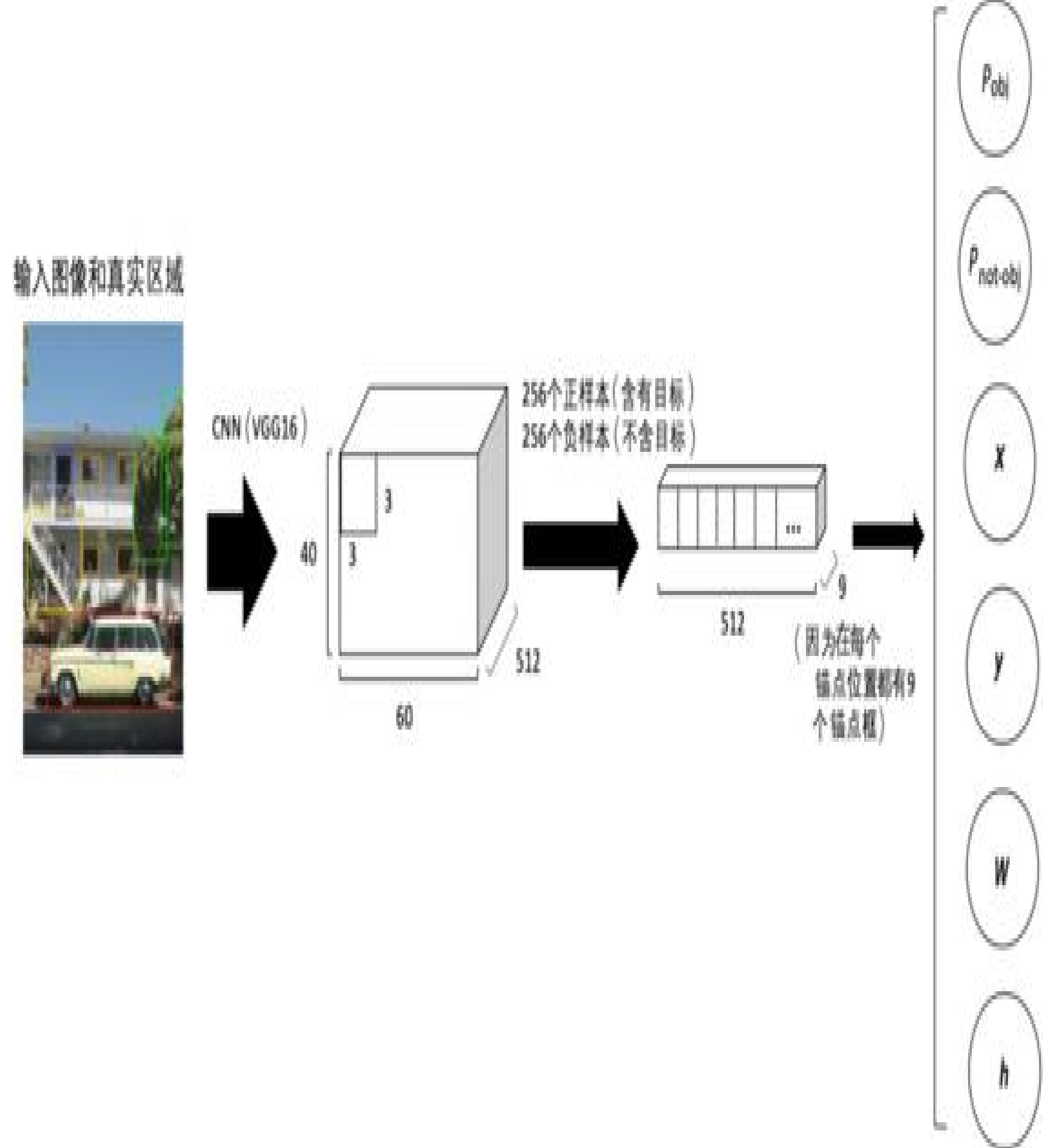
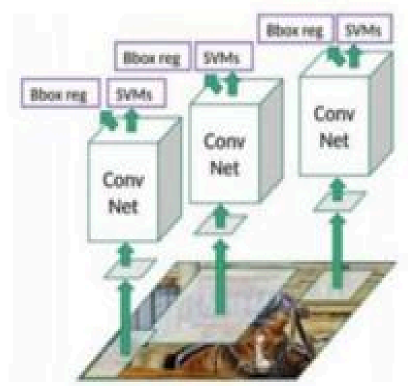


圖8-9　RPN的實現流程

如圖8-9所示就是Faster RCNN中RPN的實現流程，然而這並不是Faster RCNN的完整工作流程。RPN的實際作用是避免Fast RCNN中的選擇性搜尋，用RPN來實現RoI的區域選擇，從而提高運算效率。為了方便對比，我們可以看看在斯坦福大學李飛飛教授的影像識別課程*CS231n: Convolutional Neural Networks for Visual Recognition*[7]中給出的圖例（見圖8-10），該圖例生動地展示了RCNN、Fast RCNN和Faster RCNN的工作流程。

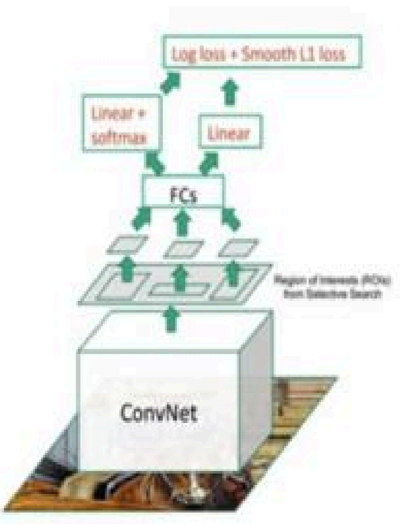


Conv Net: 卷积网络

Bbox Reg: Bounding Box (边界框)做回归预测

SVM: 用SVM算法做分类

(a) RCNN



ConvNet: 卷积网络

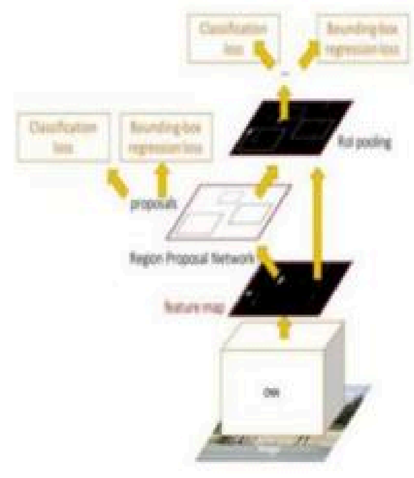
FCs: 全连接层

Linear: 线性回归

Linear + softmax: 线性回归预测+softmax处理

Log Loss + Smooth L1 Loss: 多种Loss函数结合

(b) Fast RCNN



CNN: 卷积神经网络

Feature Map: 特征图

Region Proposal Network: Faster RCNN特有，用于推荐画面可选区域

RoI Pooling: ROI采样，和Fast RCNN类似

Classification Loss: 计算分类误差

Bounding-box Regression Loss: 计算边界框回归预测误差

(c) Faster RCNN

圖8-10　斯坦福大學CS231n課程講義圖例

上面從原理和流程兩方面解釋了基於RCNN的不同目標識別演算法。在Faster RCNN之後，Facebook的研究人員又提出了Mask RCNN[6]，但那是用於更精細的畫素級的影像分割（Image Segmentation）領域，這裡不做解釋。

8.1.4　Faster RCNN核心程式碼解析

要更好地理解Faster RCNN，我們不妨參考一個基於Keras的Faster RCNN開源實現[8]，該實現的程式碼比較集中，方便我們學習。這裡重點講解如何建立模型，而不是花太多時間去看如何處理資料，因此我們從該專案下的frcnn_train_vgg.ipynb程式碼檔案中「build the model」這一段看起：

```
1  input_shape_img = (None, None, 3)
2
3  img_input = Input(shape=input_shape_img)
4  roi_input = Input(shape=(None, 4))
5
6  # 定义基础网络，这里使用了 VGG，但也可以使用 ResNet50 或者 Inception
7  shared_layers = nn_base(img_input, trainable=True)
```

以上程式碼比較簡單，主要是定義兩個輸入層：影像和RoI輸入。nn_base是在該專案下定義的一個輔助函式，用於返回一個CNN模型。

下面看看搭建網路模型的關鍵函式：

```
1  num_anchors = len(C.anchor_box_scales) * len(C.anchor_box_ratios)
2  rpn = rpn_layer(shared_layers, num_anchors)
3
4  classifier = classifier_layer(shared_layers, roi_input, C.num_rois,
5  nb_classes=len(classes_count))
6
7  model_rpn = Model(img_input, rpn[:2])
8  model_classifier = Model([img_input, roi_input], classifier)
9
10 model_all = Model([img_input, roi_input], rpn[:2] + classifier)
```

注意，以上所示的程式碼行數並非真實的程式碼行數，只是為了方便在下文解釋時對照和參考。

讓我們看看以上程式碼都做了什麼。

第1行：num_anchors是每個點上錨點框的數量，這裡anchor_box_scales有3種，anchor_box_ratio也有3種，因此num_anchors為9。

第2行：定義RPC層，後面再專門看這個rpn_layer函式。

第4行：定義最終的分類輸出層。

第5行：獲得目標種類的總數。

第7～10行：定義RPC模型、分類模型及最終完整的Faster RCNN模型。注意，最終模型（model_all）與分類模型（model_classifier）相比只是多了RPC模型的區域位置輸出。

我們看看上面是如何定義rpn_layer和classifier_layer的：

```
1   def rpn_layer(base_layers, num_anchors):
2      x = Conv2D(512, (3, 3), padding='same', activation='relu',
3   kernel_initializer='normal', name='rpn_conv1')(base_layers)
4
5      x_class = Conv2D(num_anchors, (1, 1), activation='sigmoid',
6   kernel_initializer='uniform', name='rpn_out_class')(x)
7      x_regr = Conv2D(num_anchors * 4, (1, 1), activation='linear',
8   kernel_initializer='zero', name='rpn_out_regress')(x)
9
10  return [x_class, x_regr, base_layers]
11
12  def classifier_layer(base_layers, input_rois, num_rois, nb_classes = 4):
13     input_shape = (num_rois,7,7,512)
14
15     pooling_regions = 7
16
17     out_roi_pool = RoiPoolingConv(pooling_regions, num_rois)([base_layers,
18  input_rois])
19
20     out = TimeDistributed(Flatten(name='flatten'))(out_roi_pool)
21     out = TimeDistributed(Dense(4096, activation='relu', name='fc1'))(out)
22     out = TimeDistributed(Dropout(0.5))(out)
23     out = TimeDistributed(Dense(4096, activation='relu', name='fc2'))(out)
24     out = TimeDistributed(Dropout(0.5))(out)
25
26  out_class = TimeDistributed(Dense(nb_classes, activation='softmax',
27  kernel_initializer='zero'), name='dense_class_{}'.format(nb_classes))(out)
28
29  out_regr = TimeDistributed(Dense(4 * (nb_classes-1), activation='linear',
30  kernel_initializer='zero'), name='dense_regress_{}'.format(nb_classes))(out)
31
32     return [out_class, out_regr]
```

以上程式碼定義了RPN網路與Classifier網路。可以看到，其實RPN網路和Classifer網路的結構並不複雜。RPN網路和圖8-10(c)圖一樣，輸出的兩個卷積層表示是否包含目標及目標區域。Classifier網路則先加兩個全連線網路，然後各自用softmax函式輸出類別及用線性迴歸輸出區域。

第1～10行：首先定義RPN，我們看到第2行加入了一個3×3的卷積層；緊接著在第5行定義了一個1×1的卷積層，並用sigmoid作為啟用函式，將是否包括目標作為一個二分類處理；在第7行又定義了一個1×1的卷積層，使用線性迴歸作為啟用函式處理區域位置輸出。注意，第1個分類輸出的filter是錨點的總數，第2個位置輸出的filter是num_anchors×4，因為其包括4個數值（x、y、w、h）。

第12～32行：定義了Classifier網路。其輸入引數如下。

◎ base_layers：其實就是一個CNN×6等。

◎ input_rois：輸入的ROI，為(1, num_rois, 4)的形式，num_rois是ROI的數量，每個ROI都以(x, y, w, h)的形式儲存。

◎ nb_classes：目標類別數量。

第13～18行：定義了input_shape及pooling_regions，pooling_regions其實是pooling_size的意思。對RoiPoolingConv的實現將在後面講解。

第20～24行：定義了兩個全連線網路。注意，在本章參考文獻[9]的程式碼裡使用了TimeDistributed層，該層實際上在新的Keras中是不需要的，可以直接使用Dense處理（可參考第7章CNN中的實現）。

第26行：實現了輸出的分類層，這裡使用softmax作為啟用函式。和前面RPN的輸出分類不同，RPN只關心在區域中是否包含目標，而Classifier網路需要對每個目標類別都進行判定。

第29行：和RPN類似，使用線性迴歸實現對區域位置的輸出。

那麼RoI Pooling到底是怎麼實現的？讓我們看看RoiPoolingConv函式的實現：

```
1   class RoiPoolingConv(Layer):
2       def __init__(self, pool_size, num_rois, **kwargs):
3           self.dim_ordering = K.image_dim_ordering()
4           self.pool_size = pool_size
5           self.num_rois = num_rois
```

```
6          super(RoiPoolingConv, self).__init__(**kwargs)
7
8      def build(self, input_shape):
9          self.nb_channels = input_shape[0][3]
10
11     def compute_output_shape(self, input_shape):
12         return None, self.num_rois, self.pool_size, self.pool_size,
13 self.nb_channels
14
15     def call(self, x, mask=None):
16         assert(len(x) == 2)
17
18         img = x[0]
19
20         rois = x[1]
21
22         input_shape = K.shape(img)
23
24         outputs = []
25
26         for roi_idx in range(self.num_rois):
27             x = rois[0, roi_idx, 0]
28             y = rois[0, roi_idx, 1]
29             w = rois[0, roi_idx, 2]
30             h = rois[0, roi_idx, 3]
31
32             x = K.cast(x, 'int32')
33             y = K.cast(y, 'int32')
34             w = K.cast(w, 'int32')
35             h = K.cast(h, 'int32')
36
37             rs = tf.image.resize_images(img[:, y:y+h, x:x+w, :],
38 (self.pool_size, self.pool_size))
39             outputs.append(rs)
40
41         final_output = K.concatenate(outputs, axis=0)
42         final_output = K.reshape(final_output, (1, self.num_rois,
```

```
43 self.pool_size, self.pool_size, self.nb_channels))
44
45         final_output = K.permute_dimensions(final_output, (0, 1, 2, 3, 4))
46
47         return final_output
48
49     def get_config(self):
50         config = {'pool_size': self.pool_size,
51                   'num_rois': self.num_rois}
52         base_config = super(RoiPoolingConv, self).get_config()
53         return dict(list(base_config.items()) + list(config.items()))
```

我們看到，實際上RoI Pooling是作為Keras Layer的一個子類實現的。我們在第3章中根據Keras官方文件[10]簡單講解了如何實現自定義的Keras Layer，並做了一個簡單實現。這裡看到的關鍵步驟仍然是過載call方法的前向傳播。

第2～6行：定義該層的輸入，其中的重點是pool_size，即要獲取影像區域在Pooling後的大小（比如把100×100的區域轉為7×7，pool_size就是7）。

第8～9行：input_shape實際上是兩個4D tensor（X_img、X_roi），其中，X_img為(1, rows, cols, channels)，這裡channels其實就是RGB顏色通道；X_roi和在Classifier網路中類似，也是(1, num_rois, 4)的結構。

第15～47行：這是RoI Pooling的核心。第18～20行定義了所要用到的具體資料，img和rois分別是兩個4D tensor，格式如上所述。第26～35行的邏輯很清晰，遍歷所有RoI點，獲取每個RoI點的座標和寬高（x、y、w、h），便得到了該RoI區域img[:, y:y+h, x:x+w, :]，然後用

TensorFlow的resize_images函式把該區域的影像根據pool_size縮小（第37～38行），再把縮小後的影像放入outputs中。第41～47行根據在compute_output_shape中定義的輸出，對outputs進行拼接變形後返回輸出。

我們再對照圖8-10(c)圖的Faster RCNN結構，可以看到主要的網路已經定義完成，如圖8-11所示。

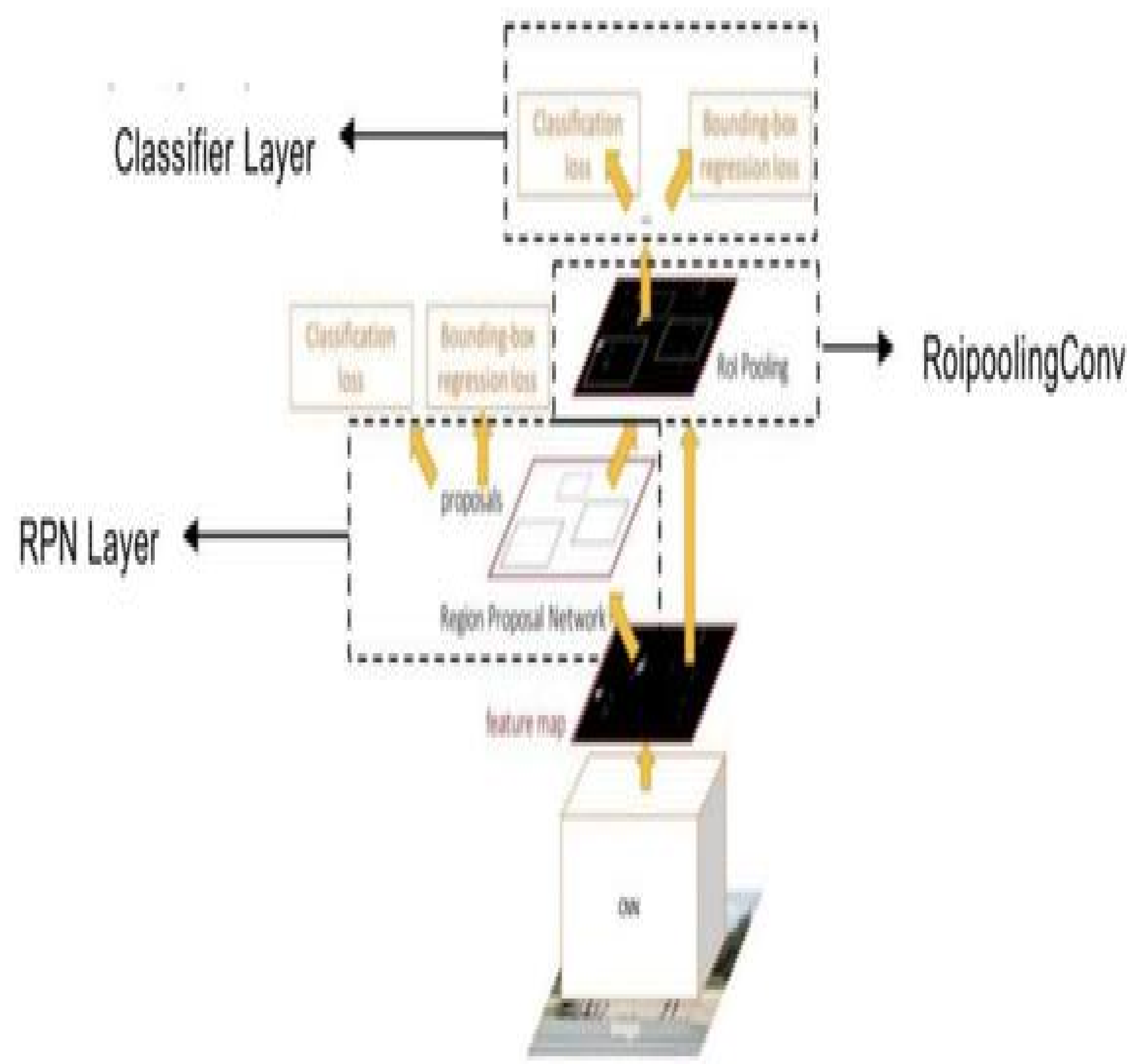


圖8-11　Faster RCNN模型和程式碼的對應關係

剩下的就是如何把這些網路串聯起來完成訓練。完整的具體實現可以閱讀本章參考文獻[8][9]，這裡限於篇幅不再贅述，只講解其中比較重要的幾點。

首先，在圖8-11中，我們從rpn_layer中得到輸出的分類和影像區域資訊後，如何將這些資訊和RoI Pooling關聯起來，或者說，如何將rpn_layer的輸出轉變為RoI Pooling的輸入？可以參考rpn_to_roi函式的實現：

```
1  def rpn_to_roi(rpn_layer, regr_layer, C, dim_ordering, max_boxes, overlap_
2  thresh=0.9):
3      regr_layer = regr_layer / C.std_scaling
4
5      anchor_sizes = C.anchor_box_scales   # (这里的数值实际上是3，表示3种尺寸)
6      anchor_ratios = C.anchor_box_ratios  # (同样，这里也是3，我们有3种比例)
7
8      assert rpn_layer.shape[0] == 1
9      (rows, cols) = rpn_layer.shape[1:3]
10
11     curr_layer = 0
12     A = np.zeros((4, rpn_layer.shape[1], rpn_layer.shape[2], rpn_layer.shape[3]))
13
14     for anchor_size in anchor_sizes:
15         for anchor_ratio in anchor_ratios:
16             anchor_x = (anchor_size * anchor_ratio[0])/C.rpn_stride
17             anchor_y = (anchor_size * anchor_ratio[1])/C.rpn_stride
```

```
18            # 该layer的形状shape 为(18, 25, 4)
19            regr = regr_layer[0,:,:,4*curr_layer:4*curr_layer+4]
20
21            regr = np.transpose(regr, (2, 0, 1)) # 该layer的形状变为(4, 18, 25)
22
23            X, Y = np.meshgrid(np.arange(cols),np. arange(rows))
24
25            A[0, :, :, curr_layer] = X - anchor_x/2 # Top left x coordinate
26            A[1, :, :, curr_layer] = Y - anchor_y/2 # Top left y coordinate
27            A[2, :, :, curr_layer] = anchor_x      # width of current anchor
28            A[3, :, :, curr_layer] = anchor_y      # height of current anchor
29
30            A[:, :, :, curr_layer] = apply_regr_np(A[:, :, :, curr_layer], regr)
31
32            A[2, :, :, curr_layer] = np.maximum(1, A[2, :, :, curr_layer])
33            A[3, :, :, curr_layer] = np.maximum(1, A[3, :, :, curr_layer])
34
35            A[2, :, :, curr_layer] += A[0, :, :, curr_layer]
36            A[3, :, :, curr_layer] += A[1, :, :, curr_layer]
37
38            A[0, :, :, curr_layer] = np.maximum(0, A[0, :, :, curr_layer])
39            A[1, :, :, curr_layer] = np.maximum(0, A[1, :, :, curr_layer])
40            A[2, :, :, curr_layer] = np.minimum(cols-1, A[2, :, :,
   curr_layer])
41            A[3, :, :, curr_layer] = np.minimum(rows-1, A[3, :, :,
   curr_layer])
42
43            curr_layer += 1
44
45      all_boxes = np.reshape(A.transpose((0, 3, 1, 2)), (4, -1)).transpose((1, 0))
46      all_probs = rpn_layer.transpose((0, 3, 1, 2)).reshape((-1))
47
48      x1 = all_boxes[:, 0]
49      y1 = all_boxes[:, 1]
50      x2 = all_boxes[:, 2]
```

```
51      y2 = all_boxes[:, 3]
52
53      idxs = np.where((x1 - x2 >= 0) | (y1 - y2 >= 0))
54
55      all_boxes = np.delete(all_boxes, idxs, 0)
56      all_probs = np.delete(all_probs, idxs, 0)
57
58      result = non_max_suppression_fast(all_boxes, all_probs,
59  overlap_thresh=overlap_thresh, max_boxes=max_boxes)[0]
60
61      return result
```

讓我們看看以上程式碼都做了什麼。

第1～3行：首先看看輸入。在輸入中最重要的就是rpn_layer和regr_layer，它們各自代表RPN網路輸出的分類和影像區域的位置，也就是說，rpn_layer對應前面rpn_layer輸出中的x_class，即每個錨點框的類別機率，形狀為shape(1, feature_map.height, feature_map.width, num_anchors)。注意，num_anchors指的是一個點上錨點框的數量，根據圖 8-7，它為9；regr_layer對應前面rpn_layer輸出中的x_regr，形狀為shape(1, feature_map.height, feature_map.width, num_anchors * 4)；輸入中的C為配置資訊，max_boxes為最多的輸出影像區域數量。對於其他內容，會在下面涉及時解釋。

第5～12行：完成初始化工作。其中最重要的是建立一個tensor A。tensor A實際上用來儲存所有錨點框的資訊。這與輸入的regr_layer中影像區域的位置有所不同，因為影像區域並不包含中心點（即錨點本身），在資料順序上也有所調整。curr_layer的初始值為0，最大值為8，因為一共有9個layer（每個錨點框對應的大小anchor_sizes、比例anchor_ratios各自為一個layer）。

第14～43行：對每個layer的錨點框進行處理。我們抽象理解的話，就是

```
for anchor_size in anchor_sizes:    # [128, 256, 512]
    for anchor_ratio in anchor_ratios: # [1.0, 0.5, 2.0]
        …
        # 转换图像区域到锚点框，并存入tensor A
        curr_layer += 1
```

第16～17行：計算當前的錨點位置。

第19～21行：對輸入的regr_layer進行轉換，首先得到當前layer的所有影像區域資訊，例如對feature map大小為40×60的，先得到一個(40,60,4)的矩陣，然後透過np.transpose將其轉換為(4, 40, 60)的矩陣，方便後續處理。

第23行：透過np.meshgrid建立兩個2D矩陣。Meshgrid的作用可以參考本章參考文獻[11]中的討論。

第25～41行：計算錨點框的位置和寬高，其中用到的apply_regr_np是計算影像區域的regression引數，在講解圖8-4時已經講到。

第45～60行：進行最後的輸出處理。all_boxes對A進行變形後，得到一個(40×60×9, 4)的矩陣，代表所有錨點框的值；all_probs直接對輸入的rpn_layer做一個轉換，得到一個一維陣列(40×60×9, 1)，代表每個錨點框是否包含目標的一個二分類。第53行對錨點框再做一次篩選，篩掉不合理的錨點框，最後做一次NonMax Suppresion，把IoU較小的錨點框全部過濾掉，只保留符合條件的部分，存在max_boxes中。non_max_ suppression_fast的實現可參考本章參考文獻[9]及後面要提到的YOLO演算法*You Only Look Once: Unified, Real-Time Object Detection*[13]，因為其實現比較簡單，所以在此不再贅述。

另外，在Faster RCNN[9]的實現中，另一個在訓練中需要用到的重要步驟就是在mini batch資料中獲取真實區域中的RPN，否則我們沒有

真正的真實區域資料，就無法訓練RPN網路。該實現在本章參考文獻[9]的calc_rpn函式中，如下所示：

```
1   def calc_rpn(C, img_data, width, height, resized_width, resized_height,
2   img_length_calc_function):
3       downscale = float(C.rpn_stride)
4       anchor_sizes = C.anchor_box_scales
5       anchor_ratios = C.anchor_box_ratios
6       num_anchors = len(anchor_sizes) * len(anchor_ratios)
7
8   (output_width, output_height) = img_length_calc_function(resized_width,
9   resized_height)
10
11      n_anchratios = len(anchor_ratios)
12
13      y_rpn_overlap = np.zeros((output_height, output_width, num_anchors))
14      y_is_box_valid = np.zeros((output_height, output_width, num_anchors))
15      y_rpn_regr = np.zeros((output_height, output_width, num_anchors * 4))
16
17      num_bboxes = len(img_data['bboxes'])
18
19      num_anchors_for_bbox = np.zeros(num_bboxes).astype(int)
```

```
20      best_anchor_for_bbox = -1*np.ones((num_bboxes, 4)).astype(int)
21      best_iou_for_bbox = np.zeros(num_bboxes).astype(np.float32)
22      best_x_for_bbox = np.zeros((num_bboxes, 4)).astype(int)
23      best_dx_for_bbox = np.zeros((num_bboxes, 4)).astype(np.float32)
24
25      gta = np.zeros((num_bboxes, 4))
26      for bbox_num, bbox in enumerate(img_data['bboxes']):
27          gta[bbox_num, 0] = bbox['x1'] * (resized_width / float(width))
28          gta[bbox_num, 1] = bbox['x2'] * (resized_width / float(width))
29          gta[bbox_num, 2] = bbox['y1'] * (resized_height / float(height))
30          gta[bbox_num, 3] = bbox['y2'] * (resized_height / float(height))
31
32      for anchor_size_idx in range(len(anchor_sizes)):
33          for anchor_ratio_idx in range(n_anchratios):
34              anchor_x = anchor_sizes[anchor_size_idx] *
35  anchor_ratios[anchor_ratio_idx][0]
36              anchor_y = anchor_sizes[anchor_size_idx] *
37  anchor_ratios[anchor_ratio_idx][1]
38
39              for ix in range(output_width):
40                  x1_anc = downscale * (ix + 0.5) - anchor_x / 2
41                  x2_anc = downscale * (ix + 0.5) + anchor_x / 2
42
43                  if x1_anc < 0 or x2_anc > resized_width:
44                      continue
45
46                  for jy in range(output_height):
47                      y1_anc = downscale * (jy + 0.5) - anchor_y / 2
48                      y2_anc = downscale * (jy + 0.5) + anchor_y / 2
49
50                      if y1_anc < 0 or y2_anc > resized_height:
51                          continue
52
53                      bbox_type = 'neg'
54
55                      best_iou_for_loc = 0.0
56
57                      for bbox_num in range(num_bboxes):
```

```
58                      curr_iou = iou([gta[bbox_num, 0], gta[bbox_num, 2],
59  gta[bbox_num, 1], gta[bbox_num, 3]], [x1_anc, y1_anc, x2_anc, y2_anc])
60                      if curr_iou > best_iou_for_bbox[bbox_num] or curr_iou >
61  C.rpn_max_overlap:
62                          cx = (gta[bbox_num, 0] + gta[bbox_num, 1]) / 2.0
63                          cy = (gta[bbox_num, 2] + gta[bbox_num, 3]) / 2.0
64                          cxa = (x1_anc + x2_anc)/2.0
65                          cya = (y1_anc + y2_anc)/2.0
66
67                          tx = (cx - cxa) / (x2_anc - x1_anc)
68                          ty = (cy - cya) / (y2_anc - y1_anc)
69                          tw = np.log((gta[bbox_num, 1] - gta[bbox_num, 0]) /
70  (x2_anc - x1_anc))
71                          th = np.log((gta[bbox_num, 3] - gta[bbox_num, 2]) /
72  (y2_anc - y1_anc))
73
74                      if img_data['bboxes'][bbox_num]['class'] != 'bg':
75                          if curr_iou > best_iou_for_bbox[bbox_num]:
76                              best_anchor_for_bbox[bbox_num] = [jy, ix,
77  anchor_ratio_idx, anchor_size_idx]
78                              best_iou_for_bbox[bbox_num] = curr_iou
79                              best_x_for_bbox[bbox_num,:] = [x1_anc, x2_anc,
80  y1_anc, y2_anc]
81                              best_dx_for_bbox[bbox_num,:] = [tx, ty, tw, th]
82
83                          if curr_iou > C.rpn_max_overlap:
84                              bbox_type = 'pos'
85                              num_anchors_for_bbox[bbox_num] += 1
86                              # we update the regression layer target if this
87  IoU is the best for the current (x,y) and anchor position
88                              if curr_iou > best_iou_for_loc:
89                                  best_iou_for_loc = curr_iou
90                                  best_regr = (tx, ty, tw, th)
91
92                          if C.rpn_min_overlap < curr_iou < C.rpn_max_overlap:
93                              if bbox_type != 'pos':
```

```
94                                  bbox_type = 'neutral'
95
96                  if bbox_type == 'neg':
97                      y_is_box_valid[jy, ix, anchor_ratio_idx + n_anchratios *
98  anchor_size_idx] = 1
99                      y_rpn_overlap[jy, ix, anchor_ratio_idx + n_anchratios *
100 anchor_size_idx] = 0
101                 elif bbox_type == 'neutral':
102                     y_is_box_valid[jy, ix, anchor_ratio_idx + n_anchratios *
103 anchor_size_idx] = 0
104                     y_rpn_overlap[jy, ix, anchor_ratio_idx + n_anchratios *
105 anchor_size_idx] = 0
106                 elif bbox_type == 'pos':
107                     y_is_box_valid[jy, ix, anchor_ratio_idx + n_anchratios *
108 anchor_size_idx] = 1
109                     y_rpn_overlap[jy, ix, anchor_ratio_idx + n_anchratios *
110 anchor_size_idx] = 1
111                     start = 4 * (anchor_ratio_idx + n_anchratios *
112 anchor_size_idx)
113                     y_rpn_regr[jy, ix, start:start+4] = best_regr
114
115     for idx in range(num_anchors_for_bbox.shape[0]):
116         if num_anchors_for_bbox[idx] == 0:
117             if best_anchor_for_bbox[idx, 0] == -1:
118                 continue
119             y_is_box_valid[
120                 best_anchor_for_bbox[idx,0], best_anchor_for_bbox[idx,1],
121 best_anchor_for_bbox[idx,2] + n_anchratios *
122                 best_anchor_for_bbox[idx,3]] = 1
123             y_rpn_overlap[
124                 best_anchor_for_bbox[idx,0], best_anchor_for_bbox[idx,1],
125 best_anchor_for_bbox[idx,2] + n_anchratios *
126                 best_anchor_for_bbox[idx,3]] = 1
127             start = 4 * (best_anchor_for_bbox[idx,2] + n_anchratios *
128 best_anchor_for_bbox[idx,3])
129             y_rpn_regr[
130                best_anchor_for_bbox[idx,0], best_anchor_for_bbox[idx,1],
```

```
131 start:start+4] = best_dx_for_bbox[idx, :]
132
133     y_rpn_overlap = np.transpose(y_rpn_overlap, (2, 0, 1))
134     y_rpn_overlap = np.expand_dims(y_rpn_overlap, axis=0)
135
136     y_is_box_valid = np.transpose(y_is_box_valid, (2, 0, 1))
137     y_is_box_valid = np.expand_dims(y_is_box_valid, axis=0)
138
139     y_rpn_regr = np.transpose(y_rpn_regr, (2, 0, 1))
140     y_rpn_regr = np.expand_dims(y_rpn_regr, axis=0)
141
142     pos_locs = np.where(np.logical_and(y_rpn_overlap[0, :, :, :] == 1,
143 y_is_box_valid[0, :, :, :] == 1))
144     neg_locs = np.where(np.logical_and(y_rpn_overlap[0, :, :, :] == 0,
145 y_is_box_valid[0, :, :, :] == 1))
146
147     num_pos = len(pos_locs[0])
148
149     num_regions = 256
150
151     if len(pos_locs[0]) > num_regions/2:
152         val_locs = random.sample(range(len(pos_locs[0])), len(pos_locs[0]) -
153 num_regions/2)
154         y_is_box_valid[0, pos_locs[0][val_locs], pos_locs[1][val_locs],
155 pos_locs[2][val_locs]] = 0
156         num_pos = num_regions/2
157
158     if len(neg_locs[0]) + num_pos > num_regions:
159         val_locs = random.sample(range(len(neg_locs[0])), len(neg_locs[0]) -
160 num_pos)
161         y_is_box_valid[0, neg_locs[0][val_locs], neg_locs[1][val_locs],
162 neg_locs[2][val_locs]] = 0
163
164     y_rpn_cls = np.concatenate([y_is_box_valid, y_rpn_overlap], axis=1)
165     y_rpn_regr = np.concatenate([np.repeat(y_rpn_overlap, 4, axis=1),
166 y_rpn_regr], axis=1)
167
168     return np.copy(y_rpn_cls), np.copy(y_rpn_regr), num_pos
```

我們可以將上面的calc_rpn看作rpn_to_roi的逆運算。因為在真實區域的資料中，影像區域其實就是roi，我們需要把它轉變成RPN的形式。

首先看看輸入。在輸入引數中包括影像資料、寬高、縮小後的大小（根據配置C中的設定）。另外，有一個計算輸出feature map的大小的函式，這個函式實際上是把影像的寬高分別除以stride（在本實現中設定為16）後得到的大小。

第2～23行：引數初始化，很多地方都同rpn_to_roi類似。注意，y_rpn_overlap指重疊的RPN區域，因為我們並不區分具體的目標，只考慮RPN是否包含任意目標（是foreground還是background），所以重疊RPN區域只要有一個包含目標，則對應的y_rpn_overlap也為1。y_is_box_valid表示該影像區域是否是有效區域（能夠明確區分foreground和background區域）；y_rpn_regr則是我們多次見到的影像區域的座標和寬高(x, y, w, h)。然後，我們用best_anchor_for_bbox、best_iou_for_bbox、best_x_for_bbox, best_dx_for_bbox來記錄每個區域的最佳值。

第25～28行：使用訓練資料集來獲取真實區域中的真實區域。

從第30行開始：和前面的rpn_to_iou方法類似，對每個layer的資料都進行處理和轉換。

第35～41行：遍歷每個位置，程式碼可以理解如下。

```
for ix in range(output_width):
    for jy in range(output_height):
```

以上程式碼首先獲取x1_anc、x2_anc作為錨點框在*x*軸的位置，然後獲取*y*軸的位置y1_anc和y2_anc，這時實際上已經找到錨點框，後面分別進行以下操作和計算：

◎　忽略落在影像之外的錨點框（第39～40行；第46～47行）；

◎　對當前點的錨點計算它所包括的所有錨點框的相關屬性（從真實區域中獲取），包括位置、寬高。針對預測目標的(tx, ty, tw, th)，

如果是目標foreground物件，則計算並儲存其最佳IoU所對應的錨點位置和相關屬性；

◎ 根據IoU覆蓋率判斷當前影像區域是屬於pos（正樣本，包含目標，IoU>0.7）還是屬於neg（負樣本，不包含任何目標，IoU<0.3），或者屬於neutral（難以判斷）；

◎ 根據bbox_type（當前影像的區域型別，由上一步獲取），設定y_is_box_valid、y_rpn_overlap等對應的值。

第97～110行：遍歷所有錨點，並確保在每個影像區域都包括一個含有目標的RPN區域。

第111～123行：針對輸出形式，對現有的資料做一些格式轉換。

第125～136行：最多輸出256個RPN區域，其中，正樣本有128個，負樣本有128個。num_regions定義了RPN區域的大小。

第137～140行：最後輸出RPN的類別和位置，以及總個數。

透過以上介紹，我們對Faster RCNN的理論和具體實現就有了較為完整的瞭解。Faster RCNN能夠提供較高的精確度，但識別速度略慢，在對實時性要求高的場景下通常不建議使用Faster RCNN。目前，在對識別速度要求高的場景下，常用的演算法是YOLO（You Only Look Once）。

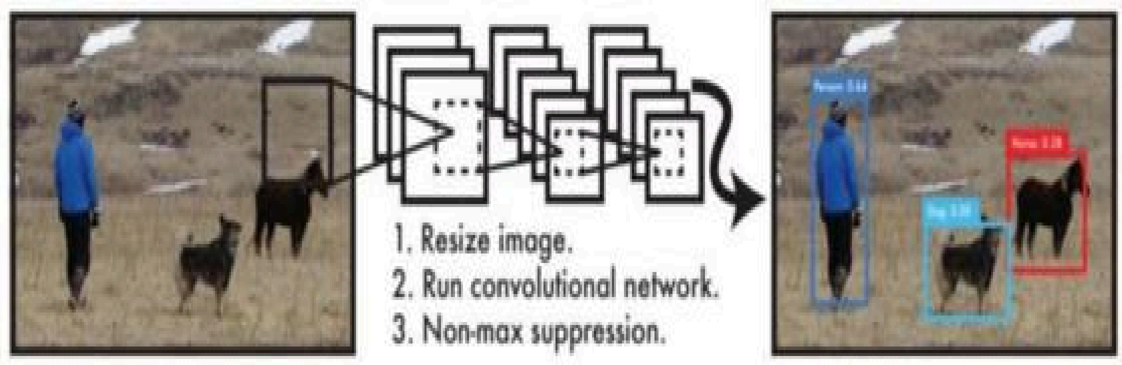
8.2 YOLO

在前面從RCNN到Faster RCNN的介紹中，我們注意到無論是RCNN還是Faster RCNN，始終可以分為兩個階段：找到備選區域（從選擇性搜尋到RPN）；在備選區域的基礎上做分類和影像區域預測。

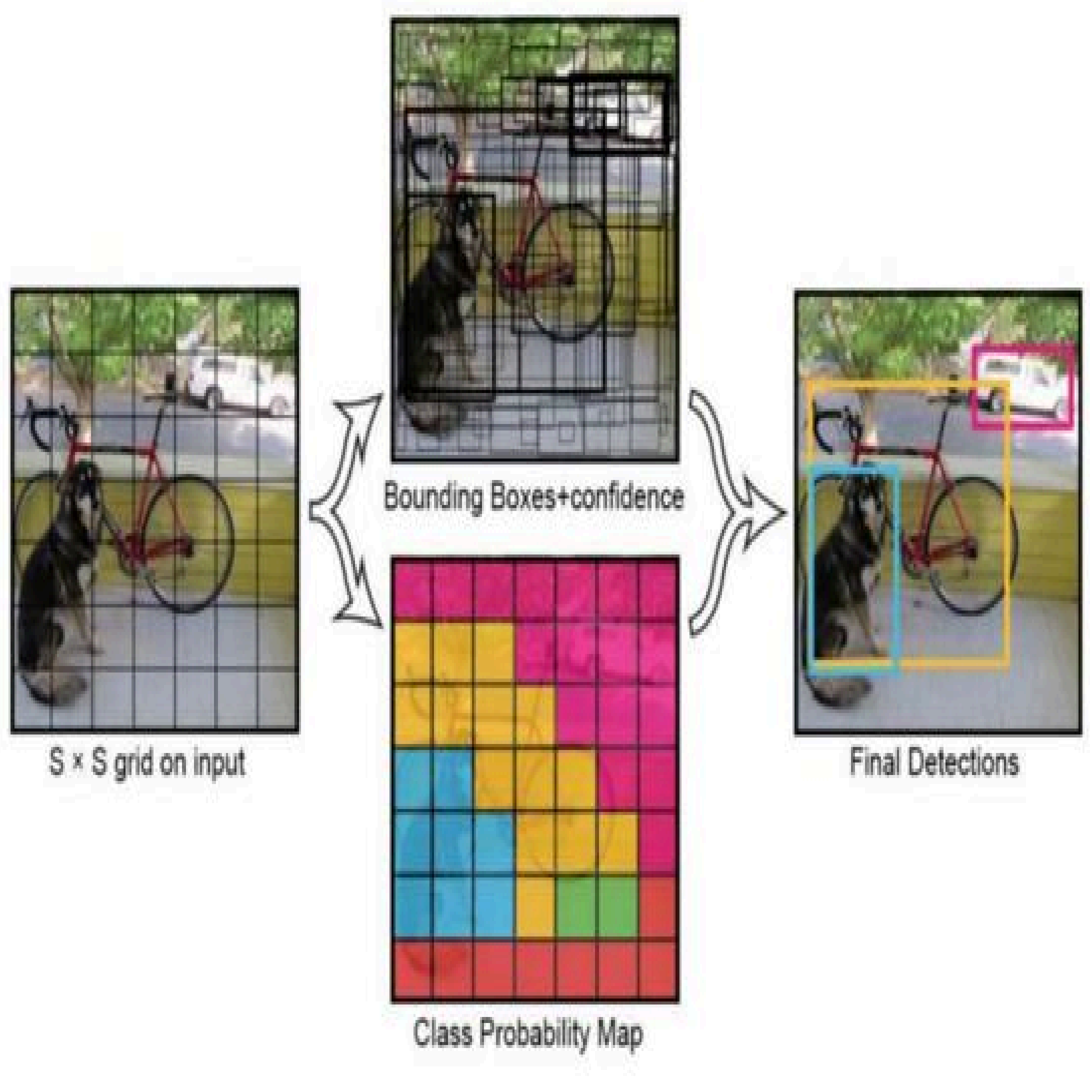
這種方式通常被稱為2-stage detection，我們不由得思考，能否把上面的兩個階段變為一個階段？在本章參考文獻[13]中，YOLO被首次提出，成為目前實時目標檢測中應用最廣泛的演算法之一。從2016年到目前為止，YOLO已經過幾個大版本的迭代，但基本原理未變，這裡先從最早的YOLO v1談起。

8.2.1 YOLO v1

YOLO的整體工作流程和演算法思想如圖8-12所示，該圖很清晰地表達了YOLO的思想：簡單即美。和繁複的Fast RCNN、Faster RCNN不同，YOLO只透過一個卷積網路對全圖進行處理，然後對輸出的feature map結果做直接運算來得到識別結果。



(a) YOLO 的整体工作流程



(b) YOLO 的算法思想

圖8-12　YOLO的整體工作流程和演算法思想[13]

在本章參考文獻[13]中提到，feature map首先被劃分為$S \times S$的網格，網格中的每一格（後簡稱cell）都負責定義B個影像區域及每個影像區域的confidence score。confidence score表示該影像區域中包含目標的機率，可以被定義為

$$\text{confidence} = \Pr(\text{Object}) \cdot \text{IoU}$$

如果在該cell中不包括任何目標，則Pr(Object)為0，否則我們期望Pr(Object)為1。因此，可以理解confidence實質上就是預測影像區域和真實區域之間的IOU。

同時，每個cell也預測C個類別機率，即當cell包含目標，也就是Pr(Object)為1時的$\Pr(\text{Class}_i \mid \text{Object})$機率，和影像區域無關。我們只對每個cell做預測，得到圖8-12(b)圖所示的Class Probability Map。

最後結合二者，在執行模型時對每個影像區域的預測計算如下：

$$\Pr(\text{class} \mid \text{object}) \times \Pr(\text{object}) \times \text{IoU} = \Pr(\text{class}_i) \times \text{IoU}$$

這樣，對每個視窗就都獲得了不同目標的預測值。再回到圖8-12(b)，我們設B=2、C=20、S=7，那麼第1次對全圖做卷積運算後，生成的結果是一個$S \times S \times (5B+C) = 7 \times 7 \times 30$的tensor，如圖8-13所示。

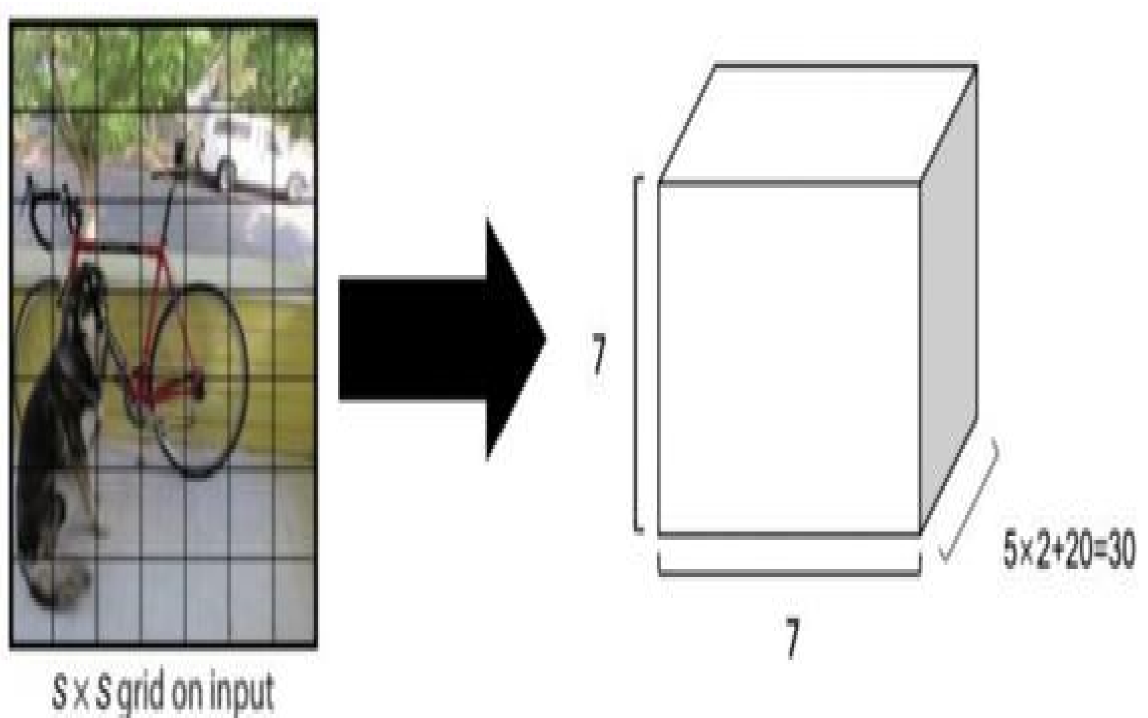


圖8-13　YOLO卷積運算輸出

但圖8-13中存在的問題是：每個cell的輸出都是一個長度為30的vector，這個vector包含什麼？

在上面的例子中，我們設定B為2（表示有兩個邊界框，即Bounding Box），這意味著每個cell都有兩個區域邊界框，而每個邊界框都包括5個屬性（c、x、y、w、h），其中的c代表我們在前面提到的confidence score，因此圖8-12(b)圖中的「Bounding Boxes+confidence」一共有5×2個屬性。緊跟著的20則是對20個不同目標種類的分類機率，如圖8-14所示。

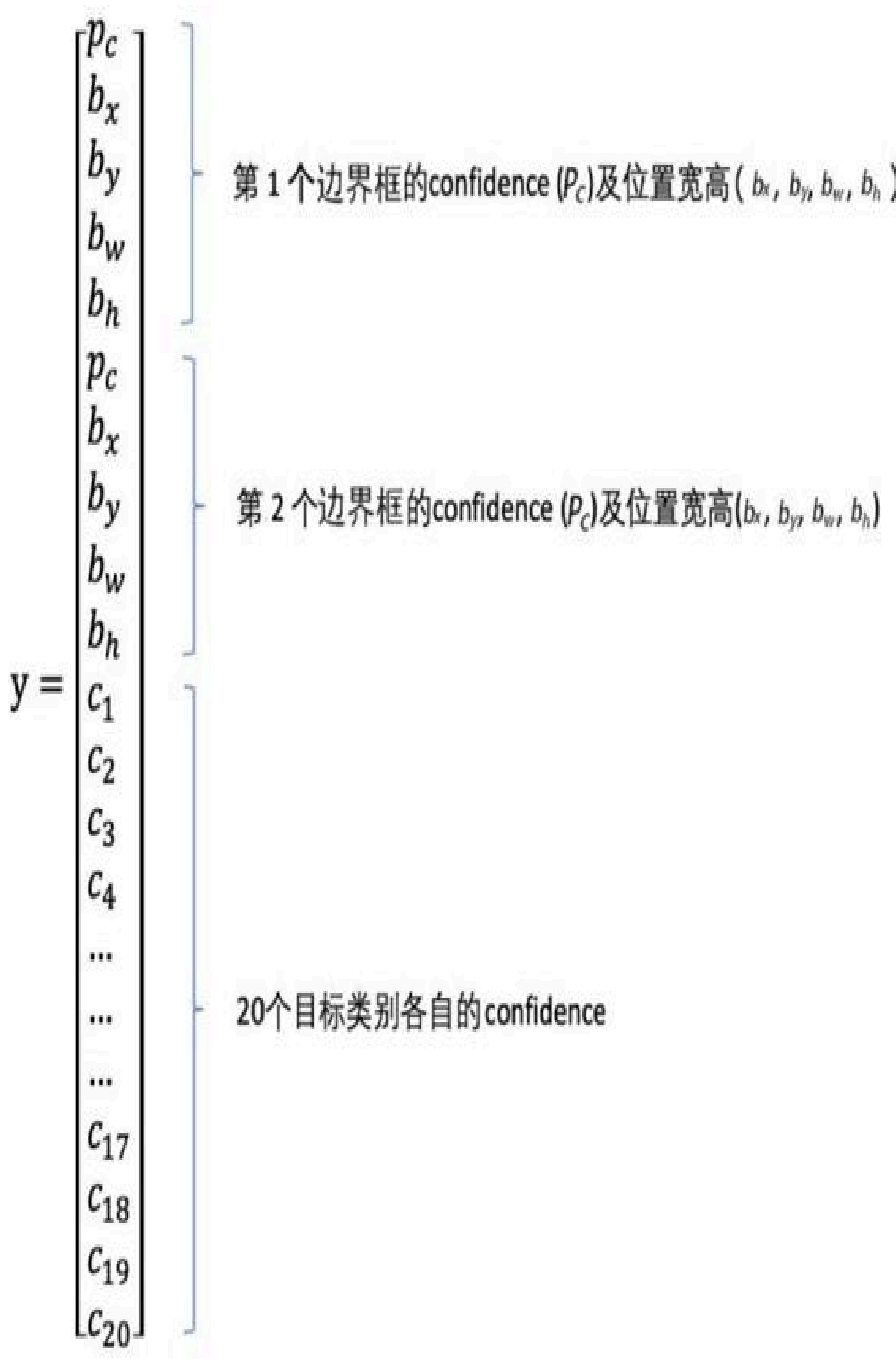

y =
p_c
b_x
b_y
b_w
b_h
p_c
b_x
b_y
b_w
b_h
c_1
c_2
c_3
c_4
…
…
…
c_{17}
c_{18}
c_{19}
c_{20}
第 1 个边界框的confidence (p_c)及位置宽高(b_x, b_y, b_w, b_h)
第 2 个边界框的confidence (p_c)及位置宽高(b_x, b_y, b_w, b_h)
20个目标类别各自的confidence

圖8-14　YOLO中卷積運算後每個cell的輸出

在本章參考文獻[14]的程式碼中包括一個YOLO v1的完整實現，其中的關鍵程式碼如下：

```
    def get_model():
        model = Sequential()
        model.add(Convolution2D(16, 3, 3,input_shape=(3,448,448),
border_mode='same',subsample=(1,1)))
        model.add(LeakyReLU(alpha=0.1))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Dense(4096))
    model.add(LeakyReLU(alpha=0.1))
    model.add(Dense(1470))
    return model
```

首先，以上程式碼的第1步就是建立一個CNN，我們注意到最後一層輸出了一個1470的陣列。而1470=7×7×(5×2+20)，恰好就是我們前面討論的總tensor維度。在這段程式碼中，作者是按照下面的規則來劃分這個總長度為1470的tensor的：

（1）一開始的0～979共980個數值代表7×7個cell對應的20個類別中每一類的機率（7×7×20=980）；

（2）接下來的98個數值代表每個cell對應的影像區域（B=2）的confidence score（7×7×2 = 98）；

（3）餘下的392個值是每個cell對應的影像區域的位置和寬高（7×7×2×4=392）。

然後，我們先跳到下面，看看YOLO模型具體是如何使用的：

```
1  model = get_model()
2  load_weights(model,'yolo-tiny.weights')
3  test_image = mpimg.imread('test_images/test1.jpg')
4  pre_processed = preprocess(test_image)
5  batch = np.expand_dims(pre_processed, axis=0)
6  batch_output = model.predict(batch)
7
8  boxes = yolo_output_to_car_boxes(batch_output[0], threshold=0.25)
9  final = draw_boxes(boxes, test_image, ((500,1280),(300,650)))
```

以上程式碼的第1行首先建立了CNN模型；第2～4行初始化了模型權重並讀入測試資料；第6行使用模型進行了預測。這些都是卷積網路的標準操作。如果我們這時把CNN的輸出列印出來，則會看到一串難以讓人理解的數字，例如：

```
[0.4187, 0.9201, 0.3476, …] # 长度为1470
```

當然，結合前面的講解，我們在理論上也可以手工分析以上資料來構建影像區域，但顯然需要額外的一步來將CNN的輸出轉變為最終的結果（也就是正確的影像區域資訊）。

在第8行，我們看到CNN的輸出傳入了一個函式yolo_output_to_car_boxes，得到了所有預測的影像區域，然後在第9行進行了繪製。實際上，前面圖8-12(b)圖中bounding boxes + confidence和Class Probability Map二者結合得到Final Detection這一步，就是yolo_output_to_car_boxes這個函式所完成的工作。那麼，我們來看具體是如何實現的：

```
1   def yolo_output_to_car_boxes(yolo_output, threshold=0.2, sqrt=1.8, C=20,
2   B=2, S=7):
3       car_class_number = 6
4
5       boxes = []
6       SS = S*S
7       prob_size = SS*C
```

```
8       conf_size = SS*B
9
10      probabilities = yolo_output[0:prob_size]
11      confidence_scores = yolo_output[prob_size: (prob_size + conf_size)]
12      cords = yolo_output[(prob_size + conf_size):]
13      probabilities = probabilities.reshape((SS, C))
14      confs = confidence_scores.reshape((SS, B))
15      cords = cords.reshape((SS, B, 4))
16
17      for grid in range(SS):
18          for b in range(B):
19              bx = Box()
20
21              bx.c = confs[grid, b]
22
23              bx.x = (cords[grid, b, 0] + grid % S) / S
24              bx.y = (cords[grid, b, 1] + grid // S) / S
25              bx.w = cords[grid, b, 2] ** sqrt
26              bx.h = cords[grid, b, 3] ** sqrt
27
28              p = probabilities[grid, :] * bx.c
29
30              if p[car_class_number] >= threshold:
31                  bx.prob = p[car_class_number]
32                  boxes.append(bx)
33
34     boxes.sort(key=lambda b: b.prob, reverse=True)
35
36     for i in range(len(boxes)):
37         boxi = boxes[i]
38         if boxi.prob == 0:
39             continue
40
41         for j in range(i + 1, len(boxes)):
42             boxj = boxes[j]
43
44         if box_iou(boxi, boxj) >= 0.4:
```

```
45            boxes[j].prob = 0
46
47    boxes = [b for b in boxes if b.prob > 0]
48
49    return boxes
```

首先，以上程式碼其實只是要檢測「car」這種型別，其他的都可以忽略，因此在第2行直接指定了car的類別class，這便於我們理解YOLO的邊界框輸出。

第3～15行：初始化變數。我們看到這裡定義了SS，它就是cell的總數（7×7=49）；prob_size相當於Class Probability Map的總數（7×7×20 = 980），conf_size則是所有cell的邊界框的confidence總數（7×7×2 = 98）；然後，我們根據前面討論的CNN輸出的資料格式，把CNN的輸出分為3段：probabilities、confidence_scores、cords。最後，在第12～14行把這3段一維陣列變形，讓後續程式碼能夠根據cell的位置獲取對應的數值。

第17～32行：從以上3段資料中獲取每個邊界框的屬性，注意在第30～31行做了一個過濾，讓只有當類別機率大於閾值時才認為這是可選的邊界框，如果低於閾值則不予考慮。

第34行：所有影像區域根據其中的prob屬性進行遞減排序。

第36～49行：刪除一些不必要的影像區域。在第44行中，如果前面（機率較大的視窗）和後面的視窗之間的IoU大於0.4，則把後面視窗的機率設為0，然後在第47行中過濾掉。最後返回最終的影像區域結果。

這樣我們就把YOLO v1的原理講清楚了。YOLO v1固然達到了很高的檢測速度，但是在準確率上還是比不過Fast RCNN和後來的在*SSD: Single Shot MultiBox Detector*[15]一文中提出的SSD。因此，之後YOLO又進行了兩次較大的迭代，分別是2016年在*YOLO9000: Better,*

Faster, Stronger[16]一文中提出的YOLO v2和2018年在*YOLO v3: An Incremental Improvement* [17]一文中所提出的YOLO v3。

8.2.2 YOLO v2

這裡不打算對YOLO v2和YOLO v3的所有細節都進行解釋，例如在YOLO v2中引入了Batch Normalization，用較高解析度預訓練10次等技巧性的改進，這裡不再贅述，而是把重點放在一些較為重要的改變上。而因為YOLO v3是基於YOLO v2進行改進的，所以這裡也不能跳過YOLO v2直接講YOLO v3。我們先來看看YOLO v2做了什麼改變吧。

YOLO v2主要是針對YOLO v1中存在的以下兩個問題進行改進的。

◎ 錨點框的大小是隨機選擇後透過卷積網路訓練的，如果我們能不「隨機選擇」錨點框的大小（即便在Faster RCNN中，錨點框的大小也不是訓練出來的），則是否可以獲得更好的效果？

◎ 如何準確識別不同尺寸的物體？（YOLO v1在識別小物體和距離鏡頭較近的物體時準確率不高）。

對於第1個問題——如何設定錨點框的大小，我們注意到實際上大部分照片上很多相似物體的比例和尺寸是相近的。如圖8-15所示為YouTube上一個交通目標識別影片的截圖，可以看到，實際上每輛汽車的邊界框在大範圍內都是非常接近的，而邊界框在很大程度上受錨點框範圍的影響。在YOLO v1中並沒有考慮這一點，而是隨意設定錨點框的範圍，然後由訓練時的網路自動調整，這會浪費大量的時間和資源。

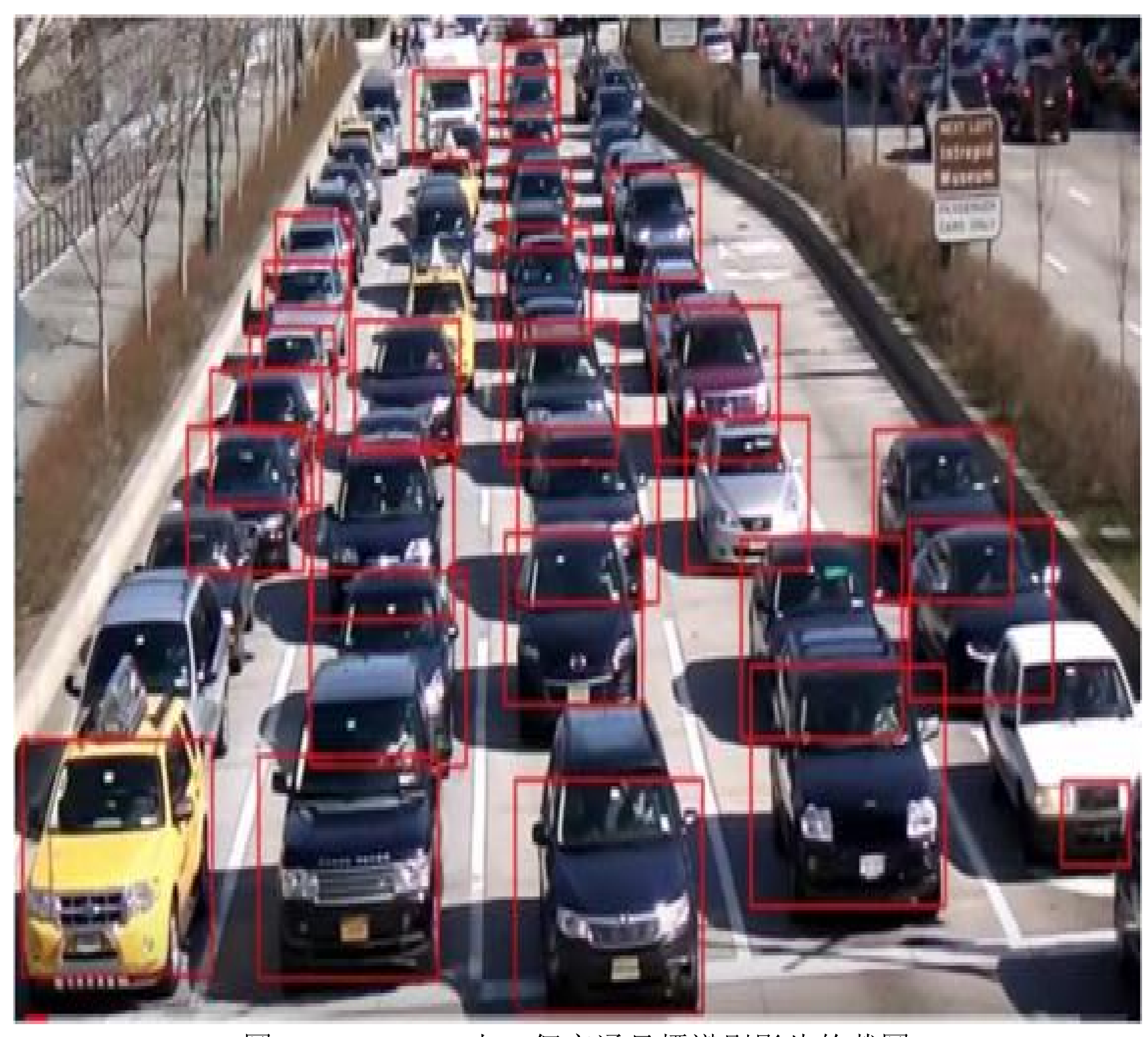


圖8-15　YouTube上一個交通目標識別影片的截圖

在YOLO v2中針對錨點框做了專門的處理。首先對於錨點框，在YOLO v1中很可能得到如圖8-16所示的效果。

圖8-16　不合適的錨點框

在圖8-16中，我們可能對人能得到較合適的錨點框，對車就不一定適用。而實際上，我們並不需要圖8-16中的所有錨點框都很準確地覆蓋目標，只要有一個錨點框有較好的效果就行。

在YOLO v1中，一個錨點對應兩個錨點框。而在YOLO v2中，對每個錨點都設定了5個不同大小比例的錨點框（在後面講解如何設定），然後透過訓練在初始值上進行調整，訓練的不是錨點框的絕對數值，而是相對於對應的cell的偏移。在*YOLO9000: Better, Faster, Stronger*[16]一文中提到使用了邏輯迴歸作為啟用函式，這樣可以把偏移限制在一個cell的範圍內（0～1），不至於出現太大誤差。

這樣做有以下兩個優勢：

（1）初始的錨點框不是隨機大小的，而是根據同類物體計算出來的大致範圍，如圖8-17所示；

（2）迴歸預測不是box的絕對值，而是相對於初始位置和大小的相對值。

這兩個優勢使得YOLO v2可以更快地得到較為準確的效果，而不是像YOLO v1那樣從隨機的初始值中耗費大量時間進行調整和訓練。

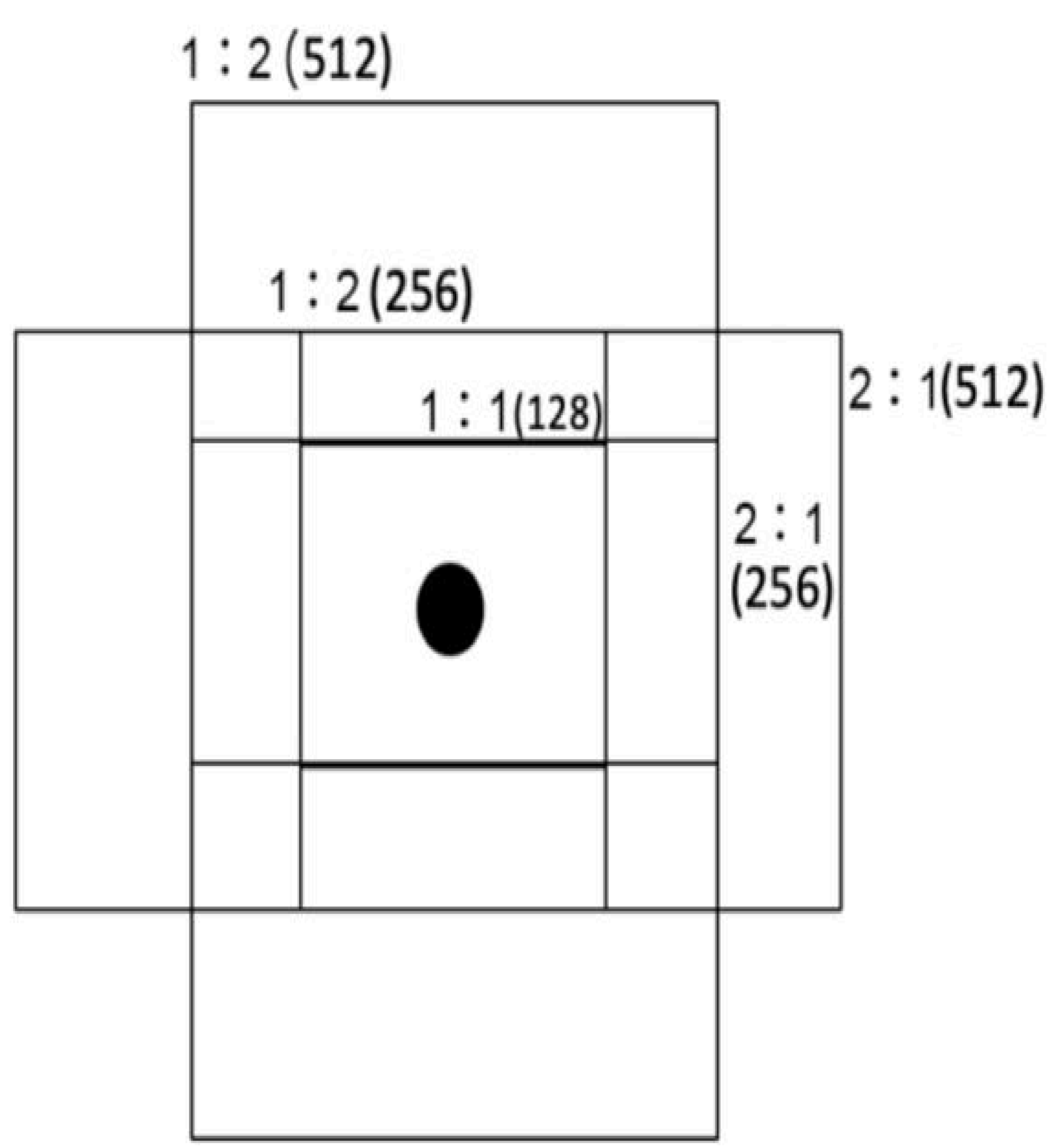


圖8-17　YOLO v2中的初始錨點框，其中，128×128下1：1一個，256×256下1：2、2：1各一個，512×512下1：2、2：1各一個

有一個問題是如何設定這些錨點框的大小？如前所述，一張圖片上相近範圍內的相似物體大小是非常接近的。這讓我們很自然地想到

了k-means聚類，使得我們能根據真實區域的資料預先獲取錨點框可能的大小範圍，而不至於漫無目的地透過訓練得到。

圖8-18描述了聚類後的結果，當然，這裡雖然用k-means做聚類，但並不能直接用錨點框之間的距離來表示，而需要另外思考如何判斷相似性。

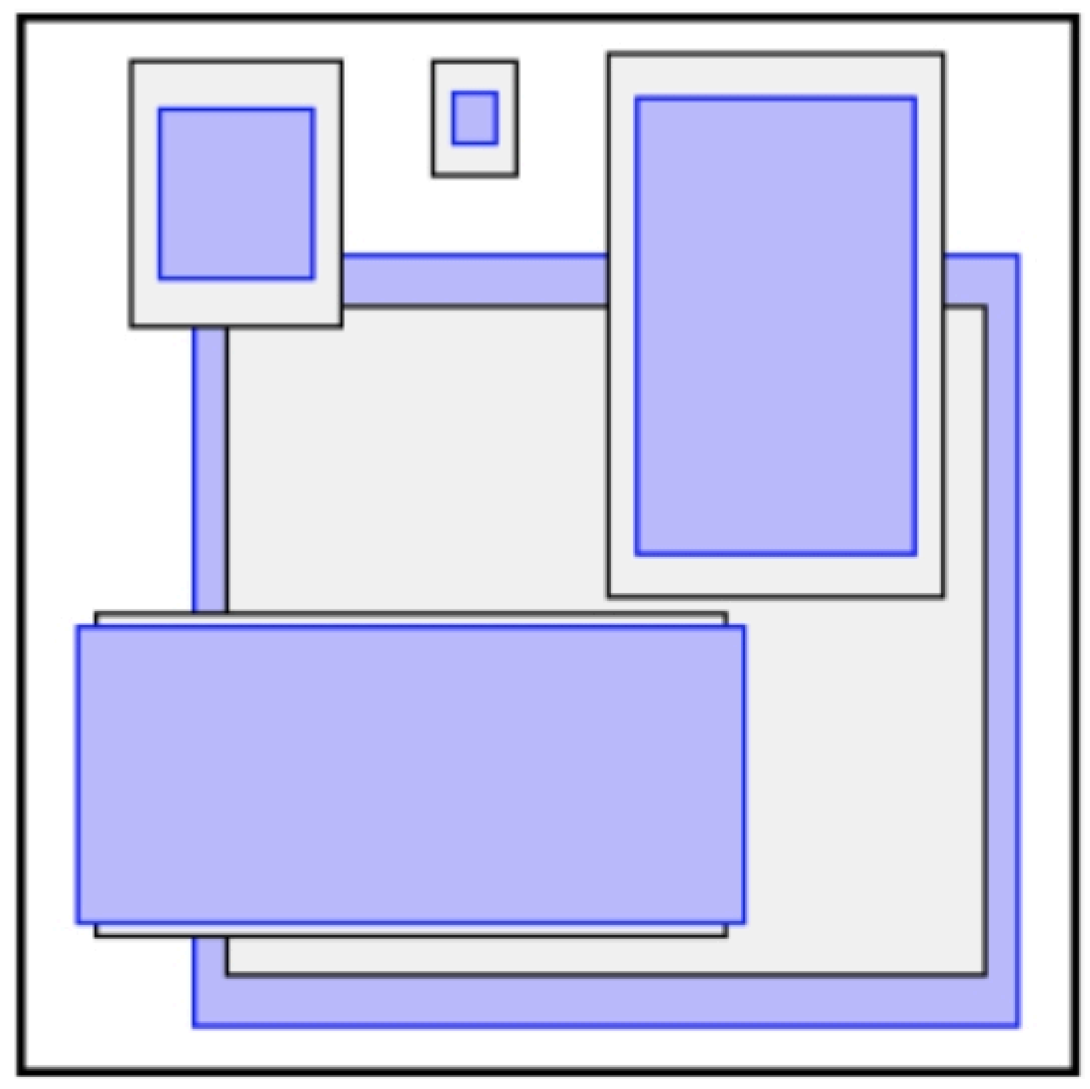


圖8-18　聚類後的結果

我們實際上是要根據二者間的IoU來聚類，IoU越大則越相近，因此在*YOLO9000: Better, Faster, Stronger*[16]一文中提出了距離計算方式：

$$d(\text{box}, \text{centroid}) = 1 - \text{IoU}(\text{box}, \text{centroid})$$

其中，box是待比較的錨點框，centroid是當前選中的基準框。

這樣，在聚類後就能得到大致的影像區域大小，通常在進行訓練之前，我們首先生成相關類別的錨點框配置。筆者在本章參考文獻[19]中有一個較完善的實現，有興趣的讀者可以自行參考。

另一個問題是如何提高對不同尺寸的同一目標的識別準確率。這個問題其實很好解決，在YOLO v2中，只是在訓練時對每次mini batch所用的圖片做了不同尺度的轉換。實際上，對每次mini batch所使用的圖片，除了對尺寸進行隨機縮放，還對圖片銳度、模糊程度、噪聲等都做了一定的處理，起到了資料增強的作用。

以上基本就是YOLO v2最重要的一些改變。下面講解YOLO v3。

8.2.3　YOLO v3

YOLO v3出自2018年CVPR上釋出的*YOLO v3: An Incremental Improvemen*[17]一文，相對於YOLO v2對YOLO v1在錨點框上的較大改進，YOLO v3則更多是對YOLO v2進行最佳化和網路結構上的變動。總的來說，YOLO v3的改進主要集中在以下幾方面。

（1）將錨點框從5個提升到9個。採用類似Faster RCNN的思路，分為3種縮放尺度，每個尺度都包括3個不同比例的錨點框。然而，對於錨點框的設定本身並非像Faster RCNN那樣採用固定比例，仍然和YOLO v2一樣透過k-means聚類得到，透過排序後將最大的3個用於最大的縮放尺度，將接下來的3個用於第2箇中間縮放尺度，將最後3個用於最小的縮放尺度。例如在本章參考文獻[17]中提到對於實驗所用的COCO 資料集[20]，用k-means得到的9個錨點框大小為10×13、16×30、33×23、30×61、62×45、59×119、116×90、156×198、373×326，那麼我們可以按順序劃分如下。

◎　[(10x13), (16x30), (33x23)]：最小縮放尺度的錨點框尺寸。

◎　[(30x61), (62x45), (59x119)]：中間縮放尺度的錨點框尺寸。

◎　[(116x90), (156x198), (373x326)]：最大縮放尺度的錨點框尺寸。

（2）引入ResNet。在網路結構上，在YOLO v2的基礎上引入了ResNet的設計，總層數達到了53層，並大幅度提高了識別準確率。具體的網路結構在本章參考文獻[17]中有闡述。

（3）用邏輯迴歸取代 softmax，從而能夠對同一個區域進行多種類別的標識。在過去的實現中，對同一個區域使用 softmax，只能對該區域選擇機率最大的類別。然而實際上有很多時候在同一個區域可能存在多個目標（例如擁擠的行人），因此在YOLO v3中直接使用Logistical Regression代替了 softmax，從而能夠對一個區域識別多種目標。

（4）最後一點，也是YOLO v3最大的一個貢獻，就是它在3個不同的縮放尺度下分別進行識別，從而提高了對小目標的識別能力。前面已經提到，YOLO v3採用了3種縮放尺度，並透過k-means聚類後獲得3種縮放尺度下不同錨點框的大小，而每種尺度都有3個錨點框。

YOLO最後透過一個1×1的Kernel在3個不同縮放尺度的feature map上進行檢測，這個Kernel本身則是一個$1\times1\times(B\times(5+C))$大小的tensor，其中，$B$為3（每個縮放尺度都有3個錨點框），在YOLO v3所使用的COCO資料集中一共有80類目標，那麼Kernel的大小就是$1\times1\times(3\times(5+80)) = 1\times1\times255$。

圖8-19展示了預測過程中的Kernel結構。Kernel遍歷所有的錨點，其本身是一個1×1×255的tensor。而在每個錨點上，根據Kernel的運算結果來獲得不同影像區域的值$(x, y, w, h, p_o, p_1, p_2, \ldots, p_c)\times B$，其中，$x$、$y$、$w$、$h$為影像區域$d$的位置和大小，$p_o$為包含目標的confidence，$p_1$、$p_2$等為不同目標類別的機率，$B$為影像區域的總數。

圖8-19　預測過程中的Kernel tensor

在YOLO v3中，圖片被分為3個不同尺度的feature map（52×52、26×26、13×13），這樣對不同的縮放尺度分別進行目標預測，52×52負責大目標，26×26負責中等大小目標，13×13負責較小目標，這樣能提升檢測的準確率。另外，在最後stride為32和16的兩層各自進行了一個上取樣操作，從而對較小的目標有更好的檢測效果。

上面簡單講解了YOLO v3的主要改進。下面，我們來看看YOLO v3的具體實現。

8.3　YOLO v3的具體實現

在YOLO作者的網站Darknet[21]上提供了YOLO v3的全套C語言程式碼和使用描述，然而其中的VOC資料集較大（1.9GB），程式碼實現也較為煩瑣，不利於學習。所以，我們在YOLO v3開源實現[22]的基礎上進行簡化，並針對新版TensorFlow進行調整。該程式碼被提供在本章參考文獻[19]中，其中包括一個約200張圖片的浣熊識別資料集和對應的YOLO v3 Keras實現，後面的程式碼討論將基於該版本進行，幫助讀者瞭解具體的實現過程。

整個YOLO v3的訓練和使用可以分為以下3步。

（1）資料集預處理，生成錨點框配置。

（2）訓練模型。

（3）進行預測。

我們看到，和Faster RCNN等演算法的一個差別在於，如果我們要在YOLO v3中自行訓練模型，則需要在模型訓練之前先對真實區域的資料進行預測，然後使用k-means演算法對影像區域進行聚類，獲得合適的錨點框大小。注意，這種方式是在YOLO v2中首次出現的。下面根據這3部分逐一進行講解。

8.3.1　資料預處理

我們需要先準備好訓練和測試所使用的資料，這裡將使用raccoon（浣熊）資料集來訓練一個識別浣熊的YOLO v3模型：

```
git clone https://github.com/likezhang-public/qconbj2019
cd qconbj2019/yolov3
ls data
```

在yolov3/data目錄下包括以下資料。

◎　training_image：訓練所需的圖片資料。

◎　training_annotation：訓練圖片的對應標註。

◎　test_image：測試圖片集。

◎　test_annotation：測試圖片的對應標註。

為了更好地理解資料格式，我們開啟一個annotation標註檔案，例如training_annotation/raccoon-151.xml：

```
<annotation verified="yes">
    <folder>images</folder>
    <filename>raccoon-151.jpg</filename>
    <path>/Users/datitran/Desktop/raccoon/images/raccoon-151.jpg</path>
    <source>
        <database>Unknown</database>
    </source>
    <size>
        <width>225</width>
        <height>225</height>
        <depth>3</depth>
    </size>
    <segmented>0</segmented>
    <object>
        <name>raccoon</name>
        <pose>Unspecified</pose>
        <truncated>0</truncated>
        <difficult>0</difficult>
        <bndbox>
            <xmin>42</xmin>
            <ymin>94</ymin>
            <xmax>108</xmax>
            <ymax>224</ymax>
        </bndbox>
    </object>
</annotation>
```

可以看到，標註檔案是標準的xml格式，其中定義了影像的路徑、大小、色彩通道數量（depth）、目標型別（object）及對應的影像區域位置（bndbox）等。我們真正需要的是其中的filename、width、height及影像區域的位置。注意，因為我們在這個示例程式碼中只識別raccoon型別，因此object.name這個屬性（目標類別）其實沒有什麼影響，但在實際產品中如果有多個類別，name就能發揮作用了。在raccoon-24.xml中包含對多個目標的檢測（儘管是同一類別），其定義如下：

```
<object>
    <name>raccoon</name>
    <pose>Unspecified</pose>
    <truncated>0</truncated>
    <difficult>0</difficult>
    <bndbox>
        <xmin>77</xmin>
        <ymin>48</ymin>
        <xmax>179</xmax>
        <ymax>156</ymax>
    </bndbox>
</object>
<object>
    <name>raccoon</name>
    <pose>Unspecified</pose>
    <truncated>0</truncated>
    <difficult>0</difficult>
    <bndbox>
        <xmin>139</xmin>
        <ymin>77</ymin>
        <xmax>202</xmax>
        <ymax>145</ymax>
    </bndbox>
</object>
```

標註檔案的解析程式碼在voc.py中，僅僅是對xml檔案的解析，這裡不再解釋，讀者可自行檢視。

接著我們開始進行第1步處理：根據圖片找到合理的錨點位置，透過k-means演算法對不同圖片的影像區域聚類，最終獲得9個錨點框的寬高（width和height）。這一步在gen_anchors.py中實現，其程式碼如下：

```
1   import random
2   import argparse
3   import numpy as np
4
5   from voc import parse_voc_annotation
6   import json
7   # 计算IOU区域比例
8   def IOU(ann, centroids):
9       w, h = ann
```

```
10      similarities = []
11
12      for centroid in centroids:
13          c_w, c_h = centroid
14
15          if c_w >= w and c_h >= h:
16              similarity = w*h/(c_w*c_h)
17          elif c_w >= w and c_h <= h:
18              similarity = w*c_h/(w*h + (c_w-w)*c_h)
19          elif c_w <= w and c_h >= h:
20              similarity = c_w*h/(w*h + c_w*(c_h-h))
21          else:
22              similarity = (c_w*c_h)/(w*h)
23          similarities.append(similarity)
24
25      return np.array(similarities)
26  # 计算平均 IOU
27  def avg_IOU(anns, centroids):
28      n,d = anns.shape
29      sum = 0.
30
31      for i in range(anns.shape[0]):
32          sum+= max(IOU(anns[i], centroids))
33
34      return sum/n
35  # 展示中心锚点，并生成对应文件
36  def print_anchors(centroids):
37      anchors = centroids.copy()
38      widths = anchors[:, 0]
39      sorted_indices = np.argsort(widths)
40      sorted_anchors = []
41      for i in sorted_indices:
42          w = int(anchors[i,0]*416)
43          h = int(anchors[i,1]*416)
44          sorted_anchors.append(w)
45          sorted_anchors.append(h)
```

```
46      print(sorted_anchors)
47      with open(anchor_file, 'w') as outfile:
48          json.dump(sorted_anchors, outfile)
49  # 运行 k-means 聚类，动态生成锚点框
50  def run_kmeans(ann_dims, anchor_num):
51      ann_num = ann_dims.shape[0]
52      iterations = 0
53      prev_assignments = np.ones(ann_num)*(-1)
54      iteration = 0
55      old_distances = np.zeros((ann_num, anchor_num))
56
57      indices = [random.randrange(ann_dims.shape[0]) for i in
    range(anchor_num)]
58      centroids = ann_dims[indices]
59      anchor_dim = ann_dims.shape[1]
60
61      while True:
62          distances = []
63          iteration += 1
64          for i in range(ann_num):
65              d = 1 - IOU(ann_dims[i], centroids)
66              distances.append(d)
67          distances = np.array(distances) # distances.shape = (ann_num,
68  anchor_num)
69
70          print("iteration {}: dists = {}".format(iteration,
71  np.sum(np.abs(old_distances-distances))))
72
73          # 将样本根据中心点距离分类
74          assignments = np.argmin(distances,axis=1)
75
76          if (assignments == prev_assignments).all() :
77              return centroids
78
79          # 计算新的中心点
80          centroid_sums=np.zeros((anchor_num, anchor_dim), np.float)
81          for i in range(ann_num):
```

```
82              centroid_sums[assignments[i]]+=ann_dims[i]
83          for j in range(anchor_num):
84              centroids[j] = centroid_sums[j]/(np.sum(assignments==j) + 1e-6)
85
86          prev_assignments = assignments.copy()
87          old_distances = distances.copy()
88
89  def _main_(argv):
90      config_path = args.conf
91      num_anchors = args.anchors
92
93      with open(config_path) as config_buffer:
94          config = json.loads(config_buffer.read())
95
96      train_imgs, train_labels = parse_voc_annotation(
97          config['train']['train_annot_folder'],
98          config['train']['train_image_folder'],
99          config['train']['cache_name'],
100         config['model']['labels']
101     )
102
103     # 运行 k-means 算法找到锚点 anchors
104     annotation_dims = []
105     for image in train_imgs:
106         print(image['filename'])
107         for obj in image['object']:
108             relative_w = (float(obj['xmax']) - float(obj['xmin']))/image['width']
109             relatice_h = (float(obj["ymax"]) - float(obj['ymin']))/image['height']
110             annotation_dims.append(tuple(map(float, (relative_w,relatice_h))))
111
112     annotation_dims = np.array(annotation_dims)
113     centroids = run_kmeans(annotation_dims, num_anchors)
114
115     print('\naverage IOU for', num_anchors, 'anchors:', '%0.2f' %
116 avg_IOU(annotation_dims, centroids))
```

```
117     print_anchors(centroids)
118
119 if __name__ == '__main__':
120     argparser = argparse.ArgumentParser()
121
122     argparser.add_argument(
123         '-c',
124         '--conf',
125         default='config.json',
126         help='path to configuration file')
127     argparser.add_argument(
128         '-a',
129         '--anchors',
130         default=9,
131         help='number of anchors to use')
132
133     args = argparser.parse_args()
134     _main_(args)
```

讓我們看看上面的程式碼都做了什麼。

第1～34行：引入標頭檔案，並定義了IoU的計算方法。其中，IOU(ann, centroids)計算一個目標影像區域與每個得到的錨點框（centroids）的IoU，並作為np.array返回。avg_iou則是每個標註檔案中的影像區域和最終得到的錨點框的最大IoU平均值。

第36～48：顯示所計算的9個錨點框的具體寬高。我們不妨將程式碼中的變數名centroid等同於錨點框（第39行體現了這一點），後面的程式碼則比較容易讓人理解。8.2.3節提到，YOLO v3計算3個不同縮放尺度下3個不同大小的錨點框，因此這裡對所有的錨點框都根據寬度（width）進行遞增排序，然後輸出為json檔案。

第50～87行：對影像區域進行k-means聚類，來獲取最後9個錨點框的具體實現，是核心行。在兩個輸入引數中，ann_dims是所有圖片的標註資訊陣列，anchor_num預設為9（YOLO v3定義使用了9個錨點框）。該實現的基本思路是定義9個錨點框，將所有真實的目標邊界框和這9個錨點框的IoU進行聚類，同時對錨點框進行調整。當所有影像區域所屬的錨點框都不再變化時，則表示聚類完成並返回此刻的錨點框。可以看到，第62～71行使用IOU函式計算每個影像區域和錨點框的IoU，然後選擇其中差異最小的作為對應影像區域的類別（被存在assignments陣列中，assignments包括每一個影像區域所屬的錨點框編號）。在第76～77行中，如果每個影像區域的所屬編號和上一次編號（prev_assignments）沒有發生變化，則說明聚類完成，返回當前的錨點框。

第79～84行：調整錨點框的寬高。這部分程式碼計算每個錨點框的寬高之和（注意ann_dims[i]返回的是兩個值）：

```
for i in range(ann_num):
    centroid_sums[assignments[i]]+=ann_dims[i]
```

以下兩行程式碼則對每個錨點框的寬高均取平均值，得到新的錨點框大小：

```
for j in range(anchor_num):
    centroids[j] = centroid_sums[j]/(np.sum(assignments==j) + 1e-6)
```

第89～117行：main函式為具體的執行流程，首先在第89～91行獲得config配置；然後在第96～101行對標註檔案進行解析；接著在第103～110行獲得每個目標影像區域的寬高資料（注意，是相對於原圖

的比例，不是絕對值）；最後在第112～117行透過上面的k-means演算法獲得錨點框並輸出。

我們執行該程式：

```
python gen_anchors.py -c config.json
```

並檢查輸出檔案anchors.json，可以看到其輸出是一個長度為18的陣列（9個錨點框的寬高），即[112, 134, 160, 231, 186, 338, 235, 380, 280, 294, 305, 385, 338, 226, 359, 310, 378, 392]。注意，以上輸出在每次執行時都會不同，因為k-means聚類本身具有隨機性。

8.3.2　模型訓練

在完成錨點框的聚類後，我們就可以進行模型訓練了。實際上，YOLO v3的訓練過程並不複雜，因為是One Stage Detection/Training，所以並不需要像Faster RCNN那樣同時完成錨點框的預測和影像區域的預測，因此實際上只是對一個Base Net+YoloLayer的整體端到端模型訓練。

在本文所提供的程式碼[19]中，和訓練相關的檔案如下。

◎　train.py：模型訓練的主要流程。

◎　yolo.py：YOLO的網路結構。

◎　generator.py：訓練中的資料生成。

我們首先來看train.py檔案，因為該原始碼檔案過長，所以將其程式碼劃分為不同的模組進行分析。

1.　主流程

首先是主流程的_main函式：

```
1    def _main_(args):
2        global WARMUP_EPOCHS
3        global LEARNING_RATE
4
5        config_path = args.conf
6
7        with open(config_path) as config_buffer:
8            config = json.loads(config_buffer.read())
9
10       anchors = []
11       with open(config['model']['anchors']) as anchors_file:
12           anchors = json.loads(anchors_file.read())
13
14       train_ints, valid_ints, labels, max_box_per_image =
15   create_training_instances(
16           config['train']['train_annot_folder'],
17           config['train']['train_image_folder'],
18           config['train']['cache_name'],
19           config['valid']['valid_annot_folder'],
20           config['valid']['valid_image_folder'],
21           config['valid']['cache_name'],
22           config['model']['labels']
23       )
24       print('\nTraining on: \t' + str(labels) + '\n')
25
26       train_generator = BatchGenerator(
27           instances           = train_ints,
28           anchors             = anchors,
29           labels              = labels,
30           downsample          = 32, # 网络输入和输出的下采样大小比例，对 YOLO v3 来说是 32
31
32           max_box_per_image   = max_box_per_image,
33           batch_size          = config['train']['batch_size'],
34           min_net_size        = config['model']['min_input_size'],
35           max_net_size        = config['model']['max_input_size'],
36           shuffle             = True,
```

```
37          jitter              = 0.3,
38          norm                = normalize
39      )
40
41      valid_generator = BatchGenerator(
42          instances           = valid_ints,
43          anchors             = anchors,
44          labels              = labels,
45          downsample          = 32, # ratio between network input's size and
46  network output's size, 32 for YOLO v3
47          max_box_per_image   = max_box_per_image,
48          batch_size          = config['train']['batch_size'],
49          min_net_size        = config['model']['min_input_size'],
50          max_net_size        = config['model']['max_input_size'],
51          shuffle             = True,
52          jitter              = 0.0,
53          norm                = normalize
54      )
55
56      if os.path.exists(config['train']['saved_weights_name']):
57          config['train']['warmup_epochs'] = 0
58      warmup_batches = config['train']['warmup_epochs'] *
59  (config['train']['train_times']*len(train_generator))
60
61      WARMUP_EPOCHS = config['train']['warmup_epochs']
62      LEARNING_RATE = config['train']['learning_rate']
63
64      multi_gpu = 0
65
66      train_model, infer_model = create_model(
67          nb_class            = len(labels),
68          anchors             = anchors,
69          max_box_per_image   = max_box_per_image,
70          max_grid            = [config['model']['max_input_size'],
71  config['model']['max_input_size']],
72          batch_size          = config['train']['batch_size'],
73          ignore_thresh       = config['train']['ignore_thresh'],
```

```
74         multi_gpu          = multi_gpu,
75         saved_weights_name  = config['train']['saved_weights_name'],
76         lr                = config['train']['learning_rate'],
77         grid_scales        = config['train']['grid_scales'],
78         obj_scale          = config['train']['obj_scale'],
79         noobj_scale        = config['train']['noobj_scale'],
80         xywh_scale         = config['train']['xywh_scale'],
81         class_scale        = config['train']['class_scale'],
82     )
83 
84     callbacks = create_callbacks(config['train']['saved_weights_name'],
85 config['train']['tensorboard_dir'], infer_model)
86 
87     sess = K.backend.get_session()
88     sess.run(tf.global_variables_initializer())
89 
90     train_model.fit_generator(
91         generator        = train_generator,
92         steps_per_epoch  = len(train_generator) * config['train']['train_times'],
93         epochs           = config['train']['nb_epochs'] +
94 config['train']['warmup_epochs'],
95         verbose          = 2 if config['train']['debug'] else 1,
96         callbacks        = callbacks,
97         workers          = 4,
98         max_queue_size    = 8
99     )
100 
101     average_precisions = evaluate(infer_model, valid_generator)
102 
103     for label, average_precision in average_precisions.items():
104         print(labels[label] + ': {:.4f}'.format(average_precision))
105     print('mAP: {:.4f}'.format(sum(average_precisions.values()) /
106 len(average_precisions)))
107 
108 if __name__ == '__main__':
109     argparser = argparse.ArgumentParser(description='train and evaluate
110 YOLO_v3 model on any dataset')
```

```
111    argparser.add_argument('-c', '--conf', help='path to configuration file')
112
113    args = argparser.parse_args()
114    _main_(args)
```

在上面的程式碼中，第101～106行是命令引數處理，這裡略過不提，重點是在_main函式中體現的訓練流程，如下所述。

第2～3行：引入兩個全域性變數WARMUP_EPOCHS和LEARNING_RATE。這兩個全域性變數將被用於訓練時的callback函式中，用於調整learning rate。

第5～12行：讀入config配置和前面建立的錨點框尺寸資料。

第14～23行：呼叫create_training_instances方法讀入訓練資料集，為下一步生成訓練資料做準備。

第26～54行：利用BatchGenerator建立訓練資料和驗證資料生成器。

第56～59行：檢查是否存在權重檔案，如果權重檔案不存在，則需要從頭訓練，增加warmup batches。

第61～62行：設定LEARNING_RATE和WARMUP_EPOCHS。根據Darknet[21]的實現，在WARMUP_EPOCHS之前的訓練批次中，learning rate非常小，直到WARMUP_EPOCHS之後才恢復到指定的learning rate。

第64行：設定GPU個數為0，在本書程式碼中僅以CPU實現為準。GPU的支援並不複雜，在此略過。

第66～82行：建立YOLO模型，其中使用了create_model函式。注意，這裡返回的一個是訓練時使用的模型，另一個是預測時使用的模型。前者需要包括真實區域作為輸入進行訓練，後者則不需要。

第84～85行：針對訓練過程設定回撥函式create_callbacks，方便儲存中間結果。

第87～99行：初始化TensorFlow圖中的全域性變數，並透過Keras中的fit_generator方式訓練模型。

第101～106行：評估模型結果。

我們看到，在上面的程式碼流程中有以下3個關鍵函式。

◎ create_training_instances：將原始訓練資料轉換為程式碼可使用的形式。

◎ BatchGenerator：繼承Keras.utils.Sequence型別，按照YOLO v3的設計生成對應的訓練資料mini batch，作為fit_generator中的generator引數使用。

◎ create_model：建立YOLO v3模型。

另外，create_callbacks也是其中比較重要的一環，在實際訓練中可以看到YOLO v3的訓練非常緩慢，在CPU環境下甚至以天來計算，對中間結果的儲存也變得非常重要。

2．建立資料集

下面看看這幾個關鍵函式的實現，首先是create_training_instances：

```
1   def create_training_instances(
2       train_annot_folder,
3       train_image_folder,
4       train_cache,
5       valid_annot_folder,
6       valid_image_folder,
7       valid_cache,
8       labels,
9   ):
10      # 解析训练集标注信息
11      train_ints, train_labels = parse_voc_annotation(train_annot_folder,
12  train_image_folder, train_cache, labels)
13
14      # 如果验证数据集定义文件目录存在，则对其进行解析
15      # 否则对数据进行分割
16      if os.path.exists(valid_annot_folder):
17          valid_ints, valid_labels = parse_voc_annotation(valid_annot_folder,
18  valid_image_folder, valid_cache, labels)
19      else:
20          print("valid_annot_folder not exists. Spliting the trainining set.")
21
22          train_valid_split = int(0.8*len(train_ints))
23          np.random.seed(0)
24          np.random.shuffle(train_ints)
```

```
25        np.random.seed()
26
27        valid_ints = train_ints[train_valid_split:]
28        train_ints = train_ints[:train_valid_split]
29
30    max_box_per_image = max([len(inst['object']) for inst in (train_ints +
31  valid_ints)])
32
33    return train_ints, valid_ints, sorted(labels), max_box_per_image
```

我們看看以上程式碼都做了什麼。

第 11 ～ 29 行：和前面生成錨點框類似，同樣呼叫parse_voc_annotation來解析訓練資料標註，然後判斷是否已有對應的測試資料目錄，如果沒有的話，就根據20/80的比例來切分資料。

第30～33行：找到每張圖上影像區域的最大可能值，然後將訓練資料、測試資料、類別標籤和最有可能的影像區域值返回。注意，這裡並沒去判斷在影像標籤中是否包含未定義的型別，因為我們只判斷一個型別即raccoon，所以沒必要去做過多檢查。

3．建立訓練回撥函式

再看看create_callbacks，這裡要重點注意：

```
1    def get_current_learning_rate(epoch):
2        global WARMUP_EPOCHS
3        global LEARNING_RATE
4
5        lrate = LEARNING_RATE
6
7        if epoch <= WARMUP_EPOCHS:
8            lrate = LEARNING_RATE * math.pow(epoch/WARMUP_EPOCHS, 4)
9
10       return lrate
11
12   def create_callbacks(saved_weights_name, tensorboard_logs, model_to_save):
13       makedirs(tensorboard_logs)
14
15       early_stop = EarlyStopping(
16           monitor     = 'loss',
17           min_delta   = 0.01,
18           patience    = 5,
19           mode        = 'min',
20           verbose     = 1
21       )
22       checkpoint = CustomModelCheckpoint(
23           model_to_save   = model_to_save,
24           filepath        = saved_weights_name,# + '{epoch:02d}.h5',
25           monitor         = 'loss',
26           verbose         = 1,
27           save_best_only  = True,
28           mode            = 'min',
29           period          = 1
30       )
31
32       lrate = LearningRateScheduler(get_current_learning_rate)
33
34       return [lrate, early_stop, checkpoint]
```

create_callbacks中的early_stop、checkpoint比較容易讓人理解，分別是訓練終止條件和中間結果權重的儲存路徑。

需要注意的是在LearningRateScheduler中使用的get_current_learning_rate函式，在第1～10行中，它根據當前epoch序號返回不同的learning rate，該實現對應了YOLO v3原始程式碼[21]中network.c的相應實現。

4．建立模型

上面介紹了YOLO v3訓練的主要流程和輔助程式碼，但並未提及具體模型的構造。在本節和後面的兩節中，我們將透過3個環節瞭解這個問題：構建模型的過程、YOLO層的實現、完整模型的實現。

首先是create_model函式：

```
def create_model(
    nb_class,
    anchors,
    max_box_per_image,
    max_grid, batch_size,
    warmup_batches,
    ignore_thresh,
    multi_gpu,
    saved_weights_name,
    lr,
    grid_scales,
    obj_scale,
    noobj_scale,
    xywh_scale,
    class_scale
):
    template_model, infer_model = create_yolov3_model(
        nb_class            = nb_class,
        anchors             = anchors,
        max_box_per_image   = max_box_per_image,
        max_grid            = max_grid,
        batch_size          = batch_size,
        warmup_batches      = warmup_batches,
        ignore_thresh       = ignore_thresh,
        grid_scales         = grid_scales,
        obj_scale           = obj_scale,
        noobj_scale         = noobj_scale,
        xywh_scale          = xywh_scale,
        class_scale         = class_scale
    )
# 读取预先训练的权重，如果不存在则使用初始权重
```

```
if os.path.exists(saved_weights_name):
    print("\nLoading pretrained weights.\n")
    template_model.load_weights(saved_weights_name)
else:
    template_model.load_weights("backend.h5", by_name=True)

train_model = template_model

optimizer = Adam(lr=0.0, clipnorm=0.001)
train_model.compile(loss=dummy_loss, optimizer=optimizer)

return train_model, infer_model
```

我們看到，create_model函式的邏輯實際上很簡單。首先，建立兩個YOLO v3模型（後面詳述），然後判斷是否存在預訓練權重並讀入，如果沒有預訓練權重，則讀入單獨的backend.h5作為初始權重，同時設定梯度下降最佳化器並返回。

這裡有個問題：我們是否需要使用backend.h5（實際上是Darnet中YOLO v3的預訓練權重）作為初始值？答案是理論上可行，但實際上對個人開發者來說不現實。YOLO v3的網路相當複雜，即便是YOLO v3本身也使用了Imagenet的權重作為初始值，而不是從隨機權重開始訓練的，那樣對個人開發者來說耗時巨大。因此對於個人開發者來說，使用預訓練的YOLO v3權重進行初始化是正確的選擇。

本節程式碼中所使用的backend.h5權重檔案已被上傳至博文視點官網，可按本書封底所示從博文視點官網下載。

我們再來看看create_model傳入的引數，如下所述。

◎　nb_class：類別數量，這裡只有raccoon一種類別，因此實際上是1。

◎　anchors：我們在gen_anchors.py中生成的anchorbox尺寸陣列。

◎　max_box_per_image：在每張圖片上最多可能具有的影像區域數量，在create_training_instances中獲得。

◎　max_grid：圖片的最大尺寸，這裡在config.json中設定為416×416。

◎　batch_size：訓練時mini batch的大小，這裡設為16。

◎　warmup_batches：熱身訓練的batch上限。

◎　ignore_thresh：IoU的忽略閾值，設定為0.5，即IoU小於0.5的可忽略。

◎　multi_gpu：在本節程式碼中忽略GPU的設定。

◎　saved_weights_name：訓練中的權重檔名。

◎　lr：learning rate。

◎　grid_scales：YOLO v3模型定義了3個不同縮放尺度的輸出層，對每一層的loss都可以透過grid_scale中對應的數值進行控制，這裡將grid_scales設為[1, 1, 1]，即不同輸出層的loss保持原值不變。

◎　obj_scale, noobj_scale, xywh_scale, class_scale：用於YOLO輸出時的loss計算，其中，除了obj_scale被設定為5，其餘皆為1。

這些引數也是建立YOLO v3模型的Hyper Parameter（超參），以上就是train.py中最重要的函式解析，但這只是基本流程。下面看看在yolo.py檔案中對YOLO v3模型的具體實現。

5．YOLO層

YOLO層並非標準的Keras層，主要是對輸出做最後的處理，計算邊界框屬性、confidence及類別機率的loss。在yolo.py檔案中包括了對YOLO層的描述：

```
1   class YoloLayer(Layer):
2       def_init_(self, anchors, max_grid, batch_size, warmup_batches, ignore_thresh,
3                       grid_scale, obj_scale, noobj_scale, xywh_scale, class_scale,
4                       **kwargs):
5           # 使模型设置持久化
6           self.ignore_thresh  = ignore_thresh
7           self.warmup_batches = warmup_batches
8           self.anchors       = tf.constant(anchors, dtype='float', shape=[1,1,1,3,2])
9           self.grid_scale     = grid_scale
10          self.obj_scale      = obj_scale
11          self.noobj_scale    = noobj_scale
12          self.xywh_scale     = xywh_scale
13          self.class_scale    = class_scale
14
15          # 获得网格宽高
16          max_grid_h, max_grid_w = max_grid
17
18          cell_x = tf.cast(tf.reshape(tf.tile(tf.range(max_grid_w),
```

```
19 [max_grid_h]), (1, max_grid_h, max_grid_w, 1, 1)), tf.float32)
20         cell_y = tf.transpose(cell_x, (0,2,1,3,4))
21         self.cell_grid = tf.tile(tf.concat([cell_x,cell_y],-1), [batch_size,
22 1, 1, 3, 1])
23
24         super(YoloLayer, self).__init__(**kwargs)
25
26     def build(self, input_shape):
27         super(YoloLayer, self).build(input_shape)
28
29
30     def call(self, x):
31         input_image, y_pred, y_true, true_boxes = x
32
33         y_pred = tf.reshape(y_pred, tf.concat([tf.shape(y_pred)[:3],
34 tf.constant([3, -1])], axis=0))
35
36         object_mask    = tf.expand_dims(y_true[..., 4], 4)
37
38         batch_seen = tf.Variable(0.)
39
40         grid_h     = tf.shape(y_true)[1]
41         grid_w     = tf.shape(y_true)[2]
42         grid_factor = tf.reshape(tf.cast([grid_w, grid_h], tf.float32),
43 [1,1,1,1,2])
44
45         net_h      = tf.shape(input_image)[1]
46         net_w      = tf.shape(input_image)[2]
47         net_factor  = tf.reshape(tf.cast([net_w, net_h], tf.float32),
48 [1,1,1,1,2])
49
50         """
51         调整预测结果
52         """
53         pred_box_xy    = (self.cell_grid[:,:grid_h,:grid_w,:,:] +
54 tf.sigmoid(y_pred[..., :2]))
```

```
55          pred_box_wh    = y_pred[..., 2:4]
56          pred_box_conf  = tf.expand_dims(tf.sigmoid(y_pred[..., 4]), 4)
57          pred_box_class = y_pred[..., 5:]
58
59          """
60          Adjust ground truth
61          """
62          true_box_xy    = y_true[..., 0:2]
63          true_box_wh    = y_true[..., 2:4]
64          true_box_conf  = tf.expand_dims(y_true[..., 4], 4)
65          true_box_class = tf.argmax(y_true[..., 5:], -1)
66
67          """
68          比较所有预测目标框和真实目标
69          """
70          # 初始化时将所有目标框的 objectiveness（是否包含目标）置为 0
71          conf_delta  = pred_box_conf - 0
72
73          # 然后忽略一些和真实目标重合度过高的对象
74          true_xy = true_boxes[..., 0:2] / grid_factor
75          true_wh = true_boxes[..., 2:4] / net_factor
76
77          true_wh_half = true_wh / 2.
78          true_mins    = true_xy - true_wh_half
79          true_maxes   = true_xy + true_wh_half
80
81          pred_xy = tf.expand_dims(pred_box_xy / grid_factor, 4)
82          pred_wh = tf.expand_dims(tf.exp(pred_box_wh) * self.anchors /
83  net_factor, 4)
84
85          pred_wh_half = pred_wh / 2.
86          pred_mins    = pred_xy - pred_wh_half
87          pred_maxes   = pred_xy + pred_wh_half
88
89          intersect_mins  = tf.maximum(pred_mins,  true_mins)
```

```
90          intersect_maxes = tf.minimum(pred_maxes, true_maxes)
91
92          intersect_wh    = tf.maximum(intersect_maxes - intersect_mins, 0.)
93          intersect_areas = intersect_wh[..., 0] * intersect_wh[..., 1]
94
95          true_areas = true_wh[..., 0] * true_wh[..., 1]
96          pred_areas = pred_wh[..., 0] * pred_wh[..., 1]
97
98          union_areas = pred_areas + true_areas - intersect_areas
99          iou_scores  = tf.truediv(intersect_areas, union_areas)
100
101         best_ious   = tf.reduce_max(iou_scores, axis=4)
102         conf_delta *= tf.expand_dims(tf.cast(best_ious < self.ignore_thresh,
103 tf.float32), 4)
104
105         wh_scale = tf.exp(true_box_wh) * self.anchors / net_factor
106         wh_scale = tf.expand_dims(2 - wh_scale[..., 0] * wh_scale[..., 1],
107 axis=4)
108         xy_delta    = object_mask * (pred_box_xy-true_box_xy) * wh_scale *
109 object_mask
110         wh_delta    = object_mask * (pred_box_wh-true_box_wh) * wh_scale *
111 object_mask
112         conf_delta  = object_mask * (pred_box_conf-true_box_conf) *
113 self.obj_scale + (1-object_mask) * conf_delta * self.noobj_scale
114         class_delta = object_mask * \
115                       tf.expand_dims(tf.nn.sparse_softmax_cross_entropy_with_
116 logits(labels=true_box_class, logits=pred_box_class), 4) * \ self.class_scale
117
118         loss_xy    = tf.reduce_sum(tf.square(xy_delta), list(range(1,5)))
119         loss_wh    = tf.reduce_sum(tf.square(wh_delta), list(range(1,5)))
120         loss_conf  = tf.reduce_sum(tf.square(conf_delta), list(range(1,5)))
121         loss_class = tf.reduce_sum(class_delta, list(range(1,5)))
122
123         loss = loss_xy + loss_wh + loss_conf + loss_class
124
125         return loss*self.grid_scale
126
127     def compute_output_shape(self, input_shape):
128         return [(None, 1)]
```

YoloLayer是基於Keras Layer的自定義層，第3章已講解了如何自定義Keras層，這裡不再重複。

首先是__init__方法，在第2～22行中實現。和前面train.py中的create_model類似，這裡主要儲存傳入的引數。注意，我們在這裡開始使用TensorFlow的原生資料型別和函式，不再是純粹的Keras程式碼。例如在第8行，將傳入的anchors陣列轉為一個[1,1,1,3,2]的向量（每一層的錨點框一共有3個，每個都有寬、高兩個屬性）：

```
    self.anchors      = tf.constant(anchors, dtype='float', shape=[1,1,1,3,2])
```

另外，對YOLO所要處理的grid資訊進行轉換處理：

```
    cell_x = tf.cast(tf.reshape(tf.tile(tf.range(max_grid_w), [max_grid_h]), (1,
max_grid_h, max_grid_w, 1, 1)), tf.float32)
    cell_y = tf.transpose(cell_x, (0,2,1,3,4))
    self.cell_grid = tf.tile(tf.concat([cell_x,cell_y],-1), [batch_size, 1, 1, 3, 1])
```

上面TensorFlow所進行的矩陣轉換看似複雜，實質上也就是建立一個多維tensor，用於存放每個尺寸下不同的錨點框設定時，每個cell所對應的影像區域的x、y位置。實際上，假設max_grid_w和max_grid_h為7，batch_size為64，這時如果我們檢視這3個變數，則會看到cell_x和cell_y都是shape為(1, 7, 7, 1, 1)的tensor，而cell_grid是一個shape為(64, 7, 7, 3, 2)的tensor，其中，64為mini batch的大小，grid為7×7，一共有3個錨點框，每個都包括x、y兩個屬性。

對於上面程式碼中所涉及的TensorFlow運算tf.range、tf.tile、tf.reshape、tf.cast、tf.transpose等，讀者可自行查閱TensorFlow資料[25]學習，在此不做過多解釋。

第24～29行：只是呼叫了基本的基類方法。

第28行：開始的 call(x)實現是關鍵方法，用於計算loss，我們來仔細看看完成了哪些工作。

首先在第29行解析了輸入資料，我們看到輸入包括如下內容。

◎　input_image：輸入影像。

◎　y_pred：預測結果。

◎　y_true：真實類別的預測結果。

◎　true_boxes：真實類別的影像區域。

實際上，我們在後面構建完整的YOLO模型時會看到如下呼叫方式：

```
loss_yolo_1 = YoloLayer(anchors[12:],
                        [1*num for num in max_grid],
                        batch_size,
                        warmup_batches,
                        ignore_thresh,
                        grid_scales[0],
                        obj_scale,
                        noobj_scale,
                        xywh_scale,
                        class_scale)([input_image, pred_yolo_1, true_yolo_1,
true_boxes])
```

後面的[input_image, pred_yolo_1, true_yolo_1, true_boxes]就是call()方法的輸入。

第31～43行：對以上部分變數如y_pred進行了轉換，變為[batch, grid_h, grid_w, 3, 4+1+nb_classes]的tensor。程式碼如下：

```
    input_image, y_pred, y_true, true_boxes = x
    y_pred = tf.reshape(y_pred, tf.concat([tf.shape(y_pred)[:3], tf.constant([3,
-1])], axis=0))
```

實際上y_pred這個tensor代表著：[batch的輸入,grid第幾行,grid第幾列,第幾個anchor] 所對應的屬性（x、y、w、h、object、class機率），我們可以在後面的第54～60行對照y_pred理解轉換後的資料內容。

同時，這部分定義了一些需要用到的新變數，例如：batch_seen，用於記錄當前的batch編號；object_mask，增加一個維度代表是否有目標；grid_h、grid_w、grid_factor、net_h、net_w、net_factor，這些是從真實區域的資料和輸入影像中獲得的grid寬高及輸入影像的寬高。

第45～69行：得到預測影像區域和真實影像區域的座標、寬高、confidence和類別機率。注意，這裡和YOLO v3論文[17]中的一段對應：

Predict location coordinates relative to the location of the grid cell.This bounds the ground truth to fall between 0 and 1.We use a logistic activation to constrain the network』s predictions to fall in this range.

因此，我們看到儘管對預測影像區域的寬高是直接從輸入中獲取的，但對於座標x、y加入了sigmoid函式進行轉化：

```
    pred_box_xy    = (self.cell_grid[:,:grid_h,:grid_w,:,:] +
tf.sigmoid(y_pred[..., :2]))
```

注意，對真實區域的影像區域直接採用輸入即可，不必再轉換。

第71～95行：計算一些變數，其中最重要的是下面這一步（第95行），這一步把IoU大於閾值的部分影像區域的confidence差異全部置零。

```
    conf_delta *= tf.expand_dims(tf.cast(best_ious < self.ignore_thresh,
tf.float32), 4)
```

第97～98行：計算wh_scale，是實際物件相對於輸入影像的大小。它是一個負相關的關係，為了識別小面積的物件，實際的邊框面積越小，相應的縮放尺度越大。注意，第98行的wh_scale[..., 0] * wh_scale[..., 1] 為邊框面積。

第99～103行：計算影像區域的x、y座標差異、寬高差異、confidence和類別機率的差異。一般來說，如果object_mask為1，則表示檢測到目標時，對位置、寬高、confidence和機率都要計算。如果object_mask為0，則表示沒有檢測到目標，這時只有confidence有計算loss的意義，因此我們看到在第103行中根據1-object_mask的結果做了不同的選擇。

第103～113行：應用簡單的平方和計算loss，最後乘以scale係數返回。

這樣對YoloLayer的分析就完成了。但YoloLayer只是整個YOLO模型的最後一層，完整的YOLO模型是一個類似ResNet的網路加上YoloLayer。

6．完整的YOLO v3模型

首先，定義一個卷積操作：

```
1   def _conv_block(inp, convs, do_skip=True):
2       x = inp
3       count = 0
4   
5       for conv in convs:
6           if count == (len(convs) - 2) and do_skip:
7               skip_connection = x
8           count += 1
9   
10          if conv['stride'] > 1: x = ZeroPadding2D(((1,0),(1,0)))(x)
11          x = Conv2D(conv['filter'],
12                     conv['kernel'],
13                     strides=conv['stride'],
14                     padding='valid' if conv['stride'] > 1 else 'same',
15                     name='conv_' + str(conv['layer_idx']),
16                     use_bias=False if conv['bnorm'] else True)(x)
17  
```

```
18        if conv['bnorm']: x = BatchNormalization(epsilon=0.001, name='bnorm_' +
19    str(conv['layer_idx']))(x)
20
21        if conv['leaky']: x = LeakyReLU(alpha=0.1, name='leaky_' +
22    str(conv['layer_idx']))(x)
23
24    return add([skip_connection, x]) if do_skip else x
```

_conv_block函式用於實現對輸入建立連續的卷積層，並根據情況決定是否使用Batch Normalization或LeakyRelu，最後決定是否加上skip connection（設為-2層，即倒數第2層）。

為瞭解主要流程和目的，我們再來看看以上程式碼都做了什麼。

第6～8行：判斷當前是否是-2層，如果是，則將skip_connection設為x（上一層的輸出）。

第10行：對左上角進行zero padding。

第11～16行：是常見的卷積層。

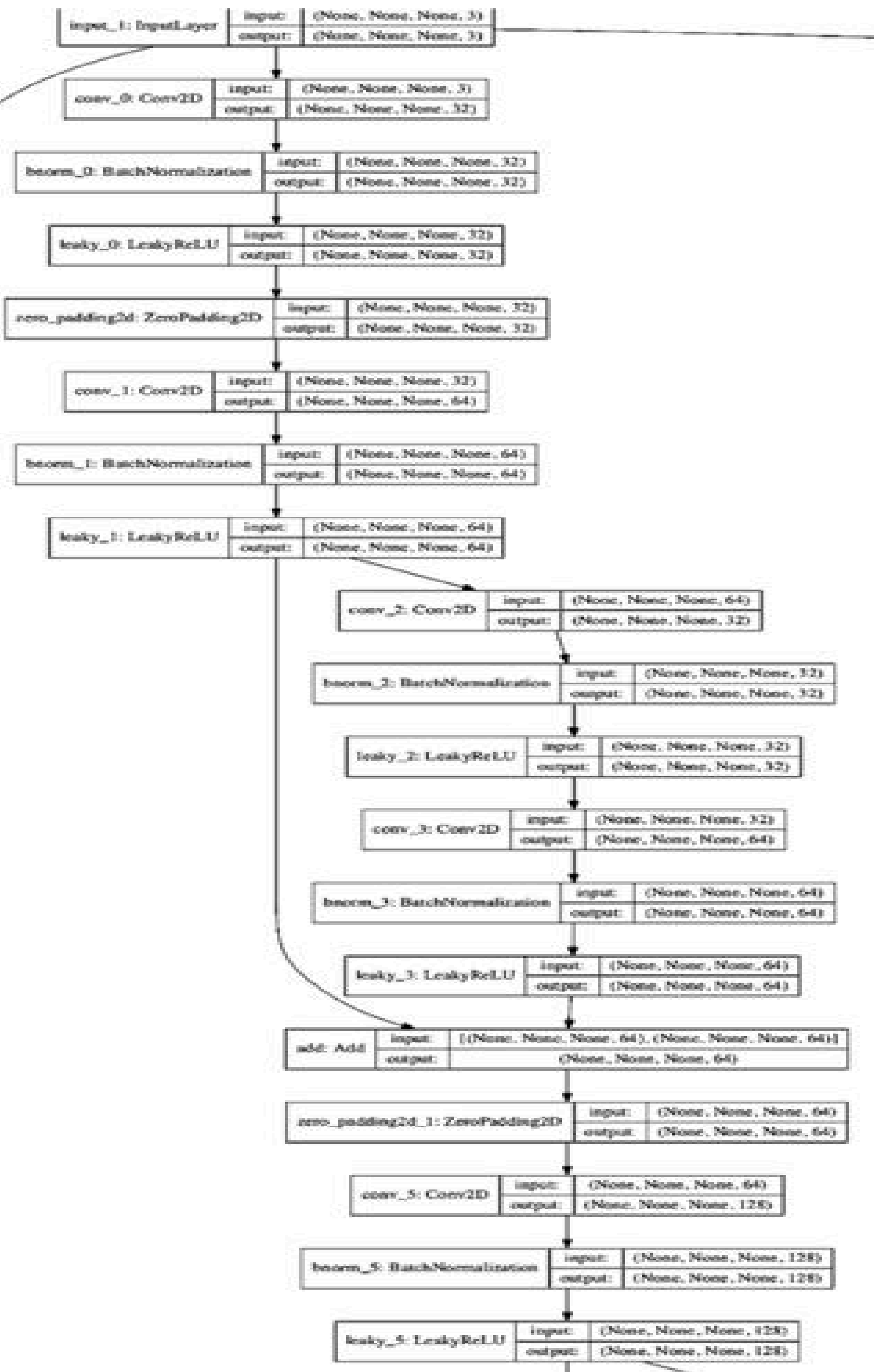
第18行：根據引數判斷是否應用BatchNormalization。

第21行：判斷是否連線LeakyReLU。

第24行：根據引數判斷是否增加skip connection。

在進入程式碼之前，我們先來看看YOLO v3訓練模型的大致結構，如圖8-20所示。

如圖8-20所示是YOLO模型一開始的幾層。我們看到，輸入除了被傳送到卷積層，還被傳送到其他地方。值得注意的是，leaky_1同樣被傳送到後續層，和後面的輸出leaky_3疊加在一起（add），然後作為新的輸出進行處理，這就是第7章講解ResNet時涉及的Skip Connection概念。我們來看看在程式碼中利用前面的_conv_block()方法是怎麼實現的：



input_1: InputLayer
input: (None, None, None, 3)
output: (None, None, None, 3)
conv_0: Conv2D
input: (None, None, None, 3)
output: (None, None, None, 32)
bnorm_0: BatchNormalization
input: (None, None, None, 32)
output: (None, None, None, 32)
leaky_0: LeakyReLU
input: (None, None, None, 32)
output: (None, None, None, 32)
zero_padding2d: ZeroPadding2D
input: (None, None, None, 32)
output: (None, None, None, 32)
conv_1: Conv2D
input: (None, None, None, 32)
output: (None, None, None, 64)
bnorm_1: BatchNormalization
input: (None, None, None, 64)
output: (None, None, None, 64)
leaky_1: LeakyReLU
input: (None, None, None, 64)
output: (None, None, None, 64)
conv_2: Conv2D
input: (None, None, None, 64)
output: (None, None, None, 32)
bnorm_2: BatchNormalization
input: (None, None, None, 32)
output: (None, None, None, 32)
leaky_2: LeakyReLU
input: (None, None, None, 32)
output: (None, None, None, 32)
conv_3: Conv2D
input: (None, None, None, 32)
output: (None, None, None, 64)
bnorm_3: BatchNormalization
input: (None, None, None, 64)
output: (None, None, None, 64)
leaky_3: LeakyReLU
input: (None, None, None, 64)
output: (None, None, None, 64)
add: Add
input: [(None, None, None, 64), (None, None, None, 64)]
output: (None, None, None, 64)
zero_padding2d_1: ZeroPadding2D
input: (None, None, None, 64)
output: (None, None, None, 64)
conv_5: Conv2D
input: (None, None, None, 64)
output: (None, None, None, 128)
bnorm_5: BatchNormalization
input: (None, None, None, 128)
output: (None, None, None, 128)
leaky_5: LeakyReLU
input: (None, None, None, 128)
output: (None, None, None, 128)

圖8-20　YOLO v3模型：起始部分

```
    x = _conv_block(input_image,
                    [{'filter': 32, 'kernel': 3, 'stride': 1, 'bnorm': True,
'leaky': True, 'layer_idx': 0},
                    {'filter': 64, 'kernel': 3, 'stride': 2, 'bnorm': True,
'leaky': True, 'layer_idx': 1},
                    {'filter': 32, 'kernel': 1, 'stride': 1, 'bnorm': True,
'leaky': True, 'layer_idx': 2},
                    {'filter': 64, 'kernel': 3, 'stride': 1, 'bnorm': True,
'leaky': True, 'layer_idx': 3}])
```

在以上程式碼中定義了4組layer，下面根據_conv_block中的程式碼來模擬對每一層的處理：

```
for conv in convs:
        if count == (len(convs) - 2) and do_skip:
            skip_connection = x
        count += 1

        if conv['stride'] > 1: x = ZeroPadding2D(((1,0),(1,0)))(x)
        x = Conv2D(…)(x)
        if conv['bnorm']: x = BatchNormalization(…)(x)
        if conv['leaky']: x = LeakyReLU(…)(x)

    return add([skip_connection, x]) if do_skip else x
```

對第1組（layer_idx=0），會建立conv_0、bnorm_0和leaky_0。

對第2組（layer_idx=1），stride=2，因此會先建立一個ZeroPadding層，然後建立conv_1、bnorm_1和leaky_1。

對第3組（layer_idx=2），一開始會先檢查count，此時count=2，len(convs)=4，因此skip_connection前面第2層的輸出（leaky_1）被設為skip_connection，然後同樣是conv_2、bnorm_2、leaky_2。

對第4組 (layer_idx=3)，同樣建立conv_3、bnorm_3、leaky_3。在這之後，第1批layer設定完畢，退出for conv in convs迴圈，然後執行add(skip_connection, x)，此刻skip_connection為前面的leaky_2，而x為當前的leaky_3。

對照圖8-20，上述過程則是從conv_0到zeropadding2d_1輸入的全過程。

在知道了skip connection的實現後，我們來看3個YoloLayer的結構（如前所述，YOLO v3一共定義了3個不同縮放尺度的YoloLayer），如圖8-21～圖8-23所示。

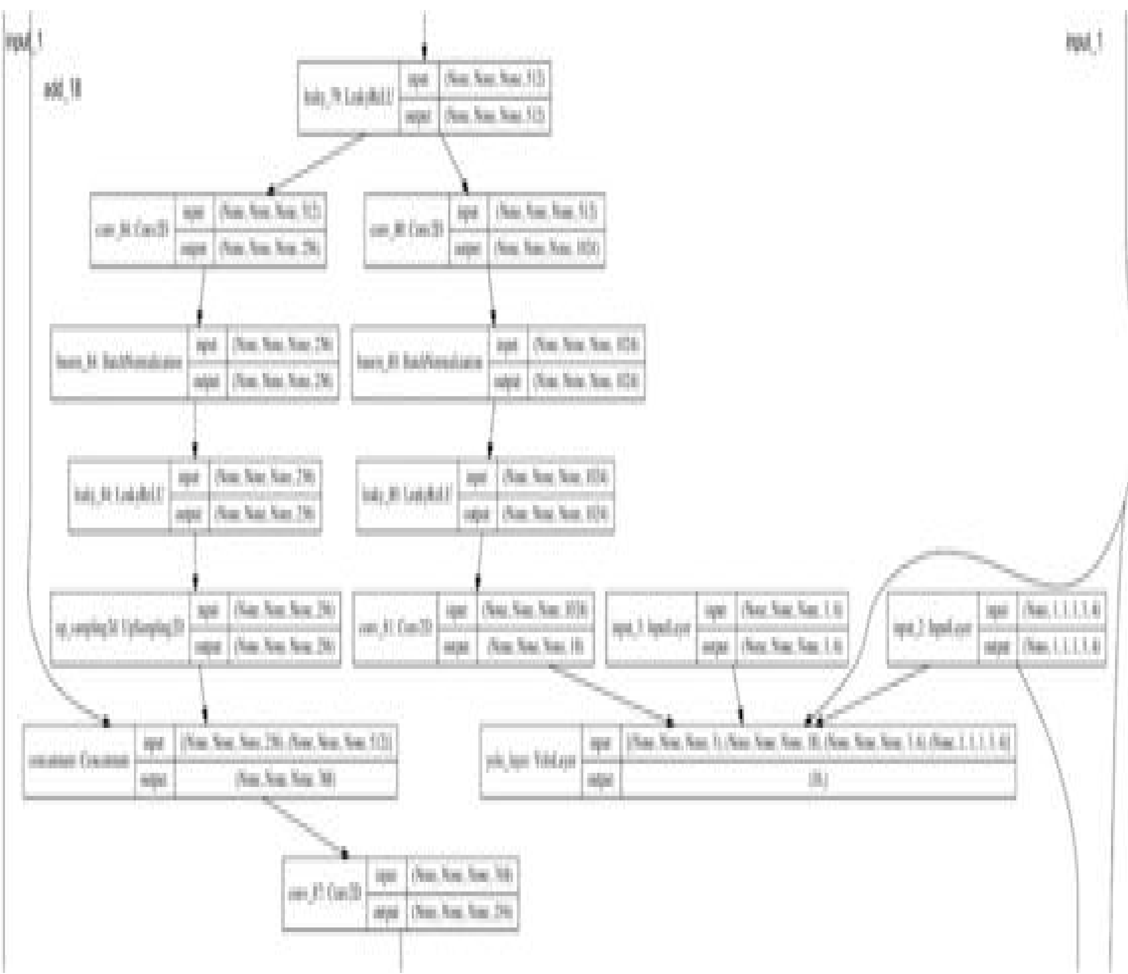


圖8-21　YOLO v3模型：第1個YoloLayer包括3個input及第81層（卷積層）的輸出

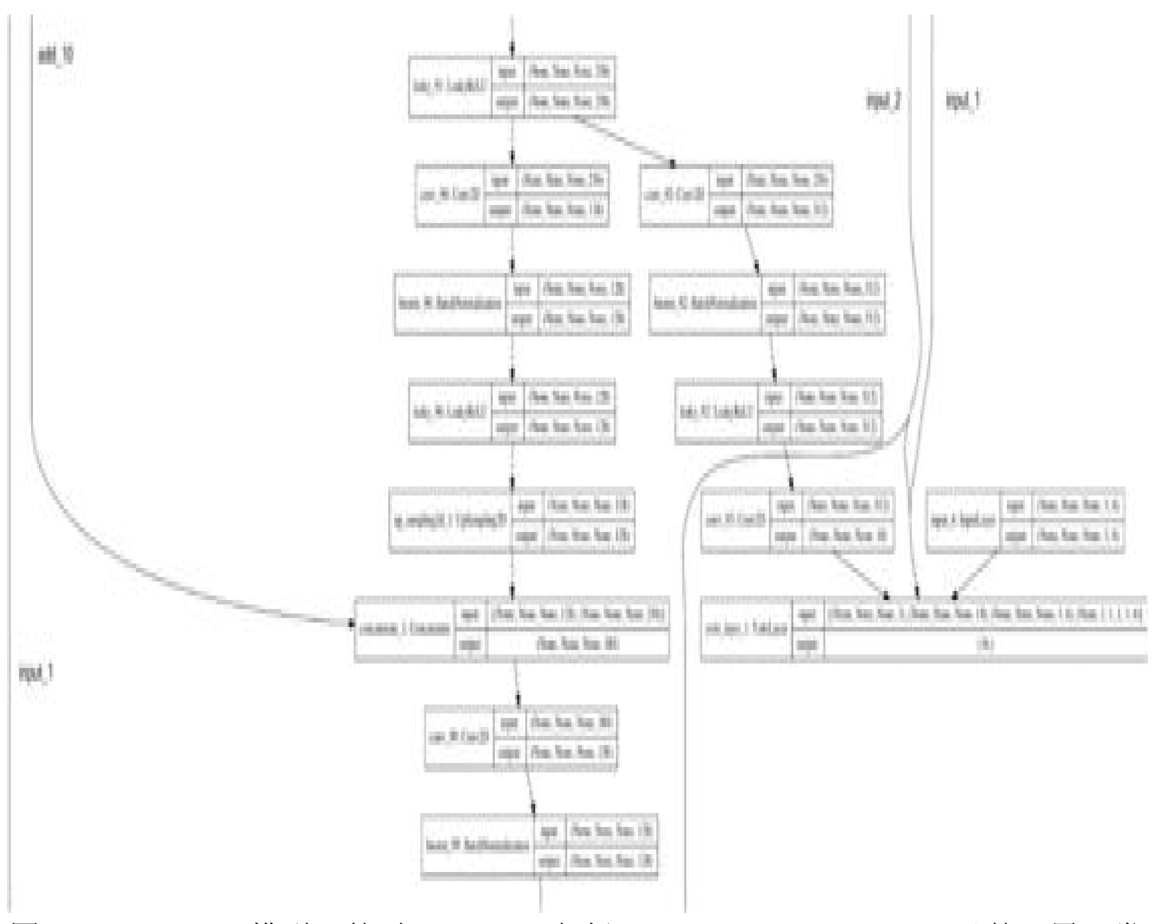


圖8-22　YOLO v3模型：第2個YoloLayer包括input_1、input_2、input_4及第93層（卷積層）的輸出

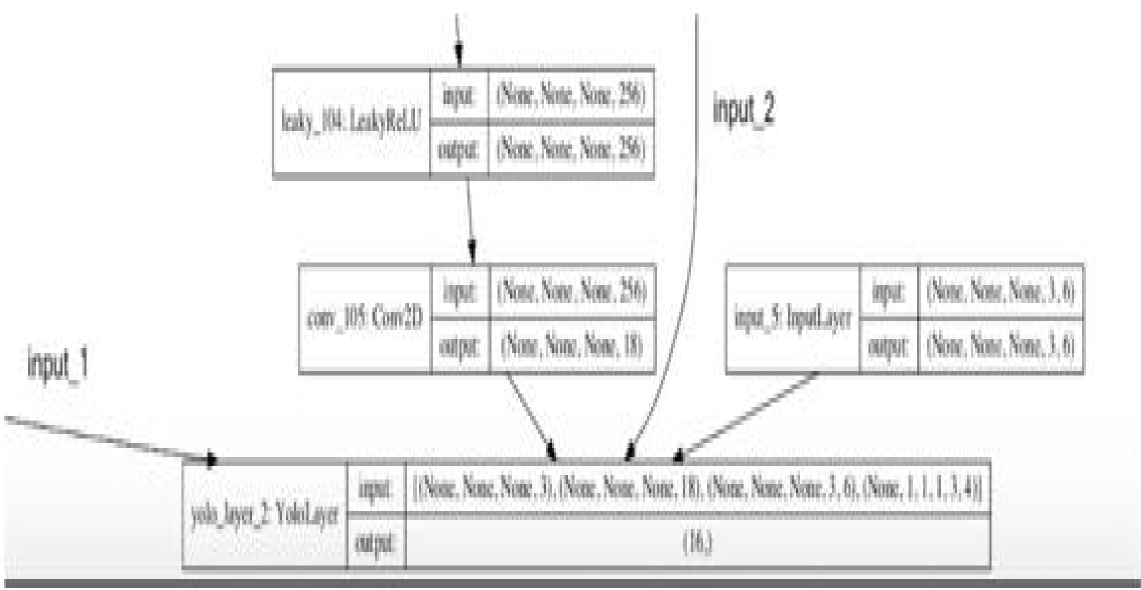


圖8-23　YOLO v3模型：第3個YoloLayer包括input_1、input_2、input_5及最後第105層的卷積層輸出

另外，注意在圖8-21中，skip_connection add_18用於concatenate的輸入，與前面的up_sampling2D輸出拼接。同樣，如圖8-22所示的add_10與up_sampling2D_1拼接。其程式碼實現類似於：

```
    for i in range(7):
        x = _conv_block(x, [{'filter': 256, 'kernel': 1, 'stride': 1, 'bnorm':
True, 'leaky': True, 'layer_idx': 41+i*3},
        {'filter': 512, 'kernel': 3, 'stride': 1, 'bnorm': True, 'leaky': True,
'layer_idx': 42+i*3}])
    skip_61 = x
    …
    x = _conv_block(x,
                        [{'filter': 256, 'kernel': 1, 'stride': 1, 'bnorm':
True, 'leaky': True, 'layer_idx': 84}], do_skip=False)
    x = UpSampling2D(2)(x)
    x = concatenate([x, skip_61])
```

這樣，我們就可以看看完整的YOLO v3模型的形態了。經過前面的程式碼解析，相信讀者能夠順利理解下面的程式碼，這裡不再贅述：

```
def create_yolov3_model(
    nb_class,
    anchors,
    max_box_per_image,
    max_grid,
    batch_size,
    warmup_batches,
    ignore_thresh,
    grid_scales,
    obj_scale,
    noobj_scale,
    xywh_scale,
    class_scale
):
    input_image = Input(shape=(None, None, 3)) # net_h, net_w, 3
    true_boxes  = Input(shape=(1, 1, 1, max_box_per_image, 4))
    true_yolo_1 = Input(shape=(None, None, len(anchors)//6, 4+1+nb_class)) #
grid_h, grid_w, nb_anchor, 5+nb_class
    true_yolo_2 = Input(shape=(None, None, len(anchors)//6, 4+1+nb_class)) #
grid_h, grid_w, nb_anchor, 5+nb_class
    true_yolo_3 = Input(shape=(None, None, len(anchors)//6, 4+1+nb_class)) #
grid_h, grid_w, nb_anchor, 5+nb_class

    # Layer  0 => 4 （定义第0~4层）
    x = _conv_block(input_image, [{'filter': 32, 'kernel': 3, 'stride': 1,
'bnorm': True, 'leaky': True, 'layer_idx': 0},
```

```
                               {'filter': 64, 'kernel': 3, 'stride': 2,
'bnorm': True, 'leaky': True, 'layer_idx': 1},
                               {'filter': 32, 'kernel': 1, 'stride': 1,
'bnorm': True, 'leaky': True, 'layer_idx': 2},
                               {'filter': 64, 'kernel': 3, 'stride': 1,
'bnorm': True, 'leaky': True, 'layer_idx': 3}])

       # Layer  5 => 8 （定义第5-8层）
       x = _conv_block(x, [{'filter': 128, 'kernel': 3, 'stride': 2, 'bnorm':
True, 'leaky': True, 'layer_idx': 5},
                       {'filter':  64, 'kernel': 1, 'stride': 1, 'bnorm': True,
'leaky': True, 'layer_idx': 6},
                       {'filter': 128, 'kernel': 3, 'stride': 1, 'bnorm': True,
'leaky': True, 'layer_idx': 7}])



       # Layer  9 => 11 （第9~11层类似，不重复）
       # …
       # Layer 16 => 36 （定义第16~36层）
       for i in range(7):
          x = _conv_block(x, [{'filter': 128, 'kernel': 1, 'stride': 1,
'bnorm': True, 'leaky': True, 'layer_idx': 16+i*3},
                          {'filter': 256, 'kernel': 3, 'stride': 1, 'bnorm':
True, 'leaky': True, 'layer_idx': 17+i*3}])

       skip_36 = x

       # ...
       # Layer 41 => 61（定义第41~64层）
       for i in range(7):
          x = _conv_block(x, [{'filter': 256, 'kernel': 1, 'stride': 1,
'bnorm': True, 'leaky': True, 'layer_idx': 41+i*3},
                          {'filter': 512, 'kernel': 3, 'stride': 1, 'bnorm':
True, 'leaky': True, 'layer_idx': 42+i*3}])

       skip_61 = x

    # Layer 62 => 65
    # …
```

```
        # Layer 80 => 82
        pred_yolo_1 = _conv_block(x, [{'filter': 1024, 'kernel': 3, 'stride': 1,
'bnorm': True,  'leaky': True,  'layer_idx': 80},
                             {'filter': (3*(5+nb_class)), 'kernel': 1, 'stride':
1, 'bnorm': False, 'leaky': False, 'layer_idx': 81}], do_skip=False)
        loss_yolo_1 = YoloLayer(anchors[12:],
                             [1*num for num in max_grid],
                             batch_size,
                             warmup_batches,
                             ignore_thresh,
                             grid_scales[0],
                             obj_scale,
                             noobj_scale,
                             xywh_scale,
                             class_scale)([input_image, pred_yolo_1, true_yolo_1,
true_boxes])

        # Layer 83 => 86
        x = _conv_block(x, [{'filter': 256, 'kernel': 1, 'stride': 1, 'bnorm':
True, 'leaky': True, 'layer_idx': 84}], do_skip=False)
        x = UpSampling2D(2)(x)
        x = concatenate([x, skip_61])

        # Layer 87 => 91
        # …
        # Layer 92 => 94
        pred_yolo_2 = _conv_block(x, [{'filter': 512, 'kernel': 3, 'stride': 1,
'bnorm': True,  'leaky': True,  'layer_idx': 92},
                             {'filter': (3*(5+nb_class)), 'kernel': 1, 'stride':
1, 'bnorm': False, 'leaky': False, 'layer_idx': 93}], do_skip=False)
        loss_yolo_2 = YoloLayer(anchors[6:12],
                             [2*num for num in max_grid],
                             batch_size,
                             warmup_batches,
                             ignore_thresh,
                             grid_scales[1],
                             obj_scale,
                             noobj_scale,
                             xywh_scale,
```

```
                             class_scale)([input_image, pred_yolo_2, true_yolo_2,
true_boxes])

        # Layer 95 => 98
        x = _conv_block(x, [{'filter': 128, 'kernel': 1, 'stride': 1, 'bnorm':
True, 'leaky': True,   'layer_idx': 96}], do_skip=False)
        x = UpSampling2D(2)(x)
        x = concatenate([x, skip_36])

        # Layer 99 => 106
        pred_yolo_3 = _conv_block(x, [{'filter': 128, 'kernel': 1, 'stride': 1,
'bnorm': True,  'leaky': True,  'layer_idx': 99},
                              {'filter': 256, 'kernel': 3, 'stride': 1, 'bnorm':
True,  'leaky': True,  'layer_idx': 100},
                              {'filter': 128, 'kernel': 1, 'stride': 1, 'bnorm':
True,  'leaky': True,  'layer_idx': 101},
                              {'filter': 256, 'kernel': 3, 'stride': 1, 'bnorm':
True,  'leaky': True,  'layer_idx': 102},
                              {'filter': 128, 'kernel': 1, 'stride': 1, 'bnorm':
True,  'leaky': True,  'layer_idx': 103},
                              {'filter': 256, 'kernel': 3, 'stride': 1, 'bnorm':
True,  'leaky': True,  'layer_idx': 104},
                              {'filter': (3*(5+nb_class)), 'kernel': 1, 'stride':
1, 'bnorm': False, 'leaky': False, 'layer_idx': 105}], do_skip=False)
        loss_yolo_3 = YoloLayer(anchors[:6],
                             [4*num for num in max_grid],
                             batch_size,
                             warmup_batches,
                             ignore_thresh,
                             grid_scales[2],
                             obj_scale,
                             noobj_scale,
                             xywh_scale,
                             class_scale)([input_image, pred_yolo_3, true_yolo_3,
true_boxes])

        train_model = Model([input_image, true_boxes, true_yolo_1, true_yolo_2,
true_yolo_3], [loss_yolo_1, loss_yolo_2, loss_yolo_3])
        infer_model = Model(input_image, [pred_yolo_1, pred_yolo_2, pred_yolo_3])

        return [train_model, infer_model]
```

注意，在倒數第5～3行建立了兩個模型，一個用於訓練，另一個用於預測。可以看到，預測模型只是少了真實區域的對應輸入，輸出從loss_yolo變為pred_yolo。若想了解loss_yolo和pred_yolo的區別，則可以參考第1個YoloLayer的輸出：

```
    pred_yolo_1 = _conv_block(x, [{'filter': 1024, 'kernel': 3, 'stride': 1,
'bnorm': True,  'leaky': True,  'layer_idx': 80},
   {'filter': (3*(5+nb_class)), 'kernel': 1, 'stride': 1, 'bnorm': False,
'leaky': False, 'layer_idx': 81}], do_skip=False)

    loss_yolo_1 = YoloLayer(anchors[12:],
                    [1*num for num in max_grid],
                    batch_size,
                    warmup_batches,
                    ignore_thresh,
                    grid_scales[0],
                    obj_scale,
                    noobj_scale,
                    xywh_scale,
                    class_scale)([input_image, pred_yolo_1, true_yolo_1,
true_boxes])
```

可以看到，pred_yolo_1的輸出就是用標準4+1+nb_class大小的tensor作為filter的卷積層輸出，這是我們預測時需要的內容；而loss_yolo_1是前面仔細分析過的YoloLayer輸出，輸出則為loss值，在訓練期間使用。對於其他部分，二者完全一致。

7. 資料增強

前面仔細講解了YOLO v3的訓練流程和模型構建。我們也提到過，YOLO v3還有一個改進，即利用資料增強在訓練過程中構造不同解析度的圖片，從而提高對不同縮放尺度的圖片的識別精度。

在第7章介紹CNN網路時使用了Keras自帶的ImageDataGenerator來實現資料增強，這裡在generator.py中透過繼承keras.utils.Sequence實現自定義的資料增強類BatchGenerator：

```
    def __init__(self,
            instances,
            anchors,
            labels,
            downsample=32, # 网络输入和输出的下采样比例，在Yolo v3中被设为32
            max_box_per_image=30,
            batch_size=1,
            min_net_size=320,
            max_net_size=608,
            shuffle=True,
            jitter=True,
            norm=None
        ):
            self.instances          = instances
            self.batch_size         = batch_size
            self.labels             = labels
            self.downsample         = downsample
            self.max_box_per_image  = max_box_per_image
            self.min_net_size       = (min_net_size//self.downsample)*self.downsample
            self.max_net_size       = (max_net_size//self.downsample)*self.downsample
            self.shuffle            = shuffle
            self.jitter             = jitter
            self.norm               = norm
            self.anchors            = [BoundBox(0, 0, anchors[2*i], anchors[2*i+1])
for i in range(len(anchors)//2)]
            self.net_h              = 416
            self.net_w              = 416

            if shuffle: np.random.shuffle(self.instances)

    def __len__(self):
            return int(np.ceil(float(len(self.instances))/self.batch_size))
```

在BatchGenerator的初始化中，我們定義了以下變數。

◎　instances：資料例項（圖片集）。

◎　anchors：利用gen_anchors.py生成的錨點資訊構建邊界框（第25～26行）。

◎　labels：資料標籤。

◎　downsample：模型輸入和輸出大小的轉換比例，在YOLO v3中為32。

◎　max_box_per_image：每張圖片上最多的目標個數。

◎　batch_size：mini batch的總次數。

◎　min_net_size：模型最小的輸入圖片尺寸。

◎　max_net_size：模型最大的輸入圖片尺寸。

◎　shuffle：是否打亂圖片順序。

◎　jitter：圖片縮放的範圍引數。

◎　norm：normalize方法，在程式碼的utils.py中定義，實際上就是把圖片畫素值normalize到[0, 1]區間後放入生成的訓練集中。

在__len__()中透過instances長度和batch_size返回每個mini batch中的資料個數。

BatchGenerator的核心函式是__getitem__(idx)，根據batch的批次編號返回訓練所需的資料。但在分析__getitem__之前，我們先要看一些輔助函式：

```
    def _get_net_size(self, idx):
        if idx%10 == 0:
            net_size =
self.downsample*np.random.randint(self.min_net_size/self.downsample, \

self.max_net_size/self.downsample+1)
            print("resizing: ", net_size, net_size)
            self.net_h, self.net_w = net_size, net_size
        return self.net_h, self.net_w

    def _aug_image(self, instance, net_h, net_w):
        image_name = instance['filename']
        image = cv2.imread(image_name) # 这里的图片为RGB 格式

        if image is None: print('Cannot find ', image_name)
        image = image[:,:,::-1]

        image_h, image_w, _ = image.shape

        # 确定缩放和裁剪范围
        dw = self.jitter * image_w;
```

```
        dh = self.jitter * image_h;

        new_ar = (image_w + np.random.uniform(-dw, dw)) / (image_h +
np.random.uniform(-dh, dh));
        scale = np.random.uniform(0.25, 2);

        if (new_ar < 1):
            new_h = int(scale * net_h);
            new_w = int(net_h * new_ar);
        else:
            new_w = int(scale * net_w);
            new_h = int(net_w / new_ar);

        dx = int(np.random.uniform(0, net_w - new_w));
        dy = int(np.random.uniform(0, net_h - new_h));

        # 对图像 image 进行缩放和裁剪
        im_sized = apply_random_scale_and_crop(image, new_w, new_h, net_w,
net_h, dx, dy)

        # 对 HSV 色彩空间进行随机扭曲
        im_sized = random_distort_image(im_sized)

        # 随机对图片进行翻转
        flip = np.random.randint(2)
        im_sized = random_flip(im_sized, flip)

        # 修正边界框的位置和大小，代码实现见后面的示例
        all_objs = correct_bounding_boxes(instance['object'], new_w, new_h,
net_w, net_h, dx, dy, flip, image_w, image_h)

        return im_sized, all_objs

    def on_epoch_end(self):
        if self.shuffle: np.random.shuffle(self.instances)

    def num_classes(self):
        return len(self.labels)

    def size(self):
```

```
        return len(self.instances)

    def get_anchors(self):
        anchors = []

        for anchor in self.anchors:
            anchors += [anchor.xmax, anchor.ymax]

        return anchors

    def load_annotation(self, i):
        annots = []

        for obj in self.instances[i]['object']:
            annot = [obj['xmin'], obj['ymin'], obj['xmax'], obj['ymax'],
self.labels.index(obj['name'])]
            annots += [annot]

        if len(annots) == 0: annots = [[]]

        return np.array(annots)

    def load_image(self, i):
        return cv2.imread(self.instances[i]['filename'])
```

其中：

◎　_get_net_size(idx)用於每隔10組隨機設定當前影像的寬高。

◎　_aug_image(instance, net_h, net_w) 用於對所給的圖片進行增強（Augmentation）處理。首先，透過傳入的jitter引數控制圖片拉伸比例，並計算裁剪區域所需的dx和dy；然後，利用utils.image中的函式對影像進行變形、扭曲和翻轉等操作。這裡不對這些影像操作函式的具體實現做過多解釋，因為程式碼本身比較容易理解，對影像的變形、拉伸、縮放等實現也可以有多種方式，沒有必要一定根據示例去做，讀者完全可以憑自己的經驗對影像進行更豐富的操作，比如加入噪聲、加入色塊、半透明等效果進行影像特徵的改變。

值得一提的是其中的correct_bounding_boxes()，該函式在影像改變後是如何把原始影像所包括的影像區域對映到改變後的影像上的呢？在utils/image.py中是這麼實現的：

```
1   def correct_bounding_boxes(boxes, new_w, new_h, net_w, net_h, dx, dy, flip,
2   image_w, image_h):
3       boxes = copy.deepcopy(boxes)
4
5       # 打乱边界框bounding box的顺序
6       np.random.shuffle(boxes)
7
8       # 调整相关尺寸和位置
9       sx, sy = float(new_w)/image_w, float(new_h)/image_h
10      zero_boxes = []
11
12      for i in range(len(boxes)):
13          boxes[i]['xmin'] = int(_constrain(0, net_w, boxes[i]['xmin']*sx +dx))
14          boxes[i]['xmax'] = int(_constrain(0, net_w, boxes[i]['xmax']*sx +dx))
15          boxes[i]['ymin'] = int(_constrain(0, net_h, boxes[i]['ymin']*sy +dy))
16          boxes[i]['ymax'] = int(_constrain(0, net_h, boxes[i]['ymax']*sy +dy))
17
18          if boxes[i]['xmax'] <= boxes[i]['xmin'] or boxes[i]['ymax'] <=
19  boxes[i]['ymin']:
20              zero_boxes += [i]
21              continue
22
23          if flip == 1:
24              swap = boxes[i]['xmin'];
25              boxes[i]['xmin'] = net_w - boxes[i]['xmax']
26              boxes[i]['xmax'] = net_w - swap
27
28      boxes = [boxes[i] for i in range(len(boxes)) if i not in zero_boxes]
29
30      return boxes
```

首先，在輸入中包括新的影像寬高（new_w、new_h）、當前輸入圖片的寬高（net_w、net_h）、裁剪區域和邊緣的距離（dx、dy）、是否翻轉（flip）、原始影像的寬高（image_w、image_h）。

然後，在第9行計算相對於原圖的縮放比例sx、sy。

其次，在第13～27行對每一個原始影像對應的影像區域，透過縮放和裁剪距離計算出新的影像區域的左上座標和右下座標，如果是翻轉的影像，則需要進行第24～27行中的調整。

最後，把正確的影像區域陣列返回。

on_epoch_end()、num_classes()、size()等是Keras自定義data generator時所需的方法，這裡不做過多解釋。get_anchors()、load_annotations()、load_image()等方法並非是在訓練時使用的，而是在訓練完成後進行評價和測試時使用的，可以參考utils/evaluate.py。

這樣就完整解釋了YOLO v3的訓練、模型實現和資料生成三大核心環節，下面具體執行該程式碼。

8．預測和測試

首先，我們按照前面所說的，生成對應的anchors檔案：

```
python gen_anchors.py -c config.json
```

執行後會發現多了一個anchors.json檔案，內容為包括9對anchors寬高的陣列：

```
[112, 134, 160, 231, 186, 338, 235, 380, 280, 294, 305, 385, 338, 226, 359,
310, 378, 392]
```

緊接著便可以執行訓練程式碼：

```
python train.py -c config.json
```

我們將會看到如下輸出，且可以看到loss在不斷減少（注意，CPU環境下的訓練時間非常長）：

```
    Epoch 00001: loss improved from inf to 325.90974, saving model to raccoon.h5
     - 15101s - loss: 325.9097 - yolo_layer_loss: 43.1135 - yolo_layer_1_loss:
87.0226 - yolo_layer_2_loss: 195.7736


    …
    Epoch 00002: loss improved from 325.90974 to 265.72863, saving model to
raccoon.h5
     - 14250s - loss: 237.6354 - yolo_layer_loss: 24.7856 - yolo_layer_1_loss:
67.7918 - yolo_layer_2_loss: 142.6411
    ..
```

為了提高速度，我們在config.json中設定nb_epochs=10，只訓練10次。在訓練完成後根據config.json中的配置生成raccoon.h5。

那麼，訓練好的模型在predict.py中是怎麼使用的呢？前面提到過，在示例程式碼中包括一個訓練用的模型和一個預測用的模型。二者的輸出不一樣：前者輸出的是訓練所用的loss值，後者輸出的是卷積層輸出的4+1+nb_classes結構的預測值。predict.py用的就是後者，其中的關鍵函式是utils.py中的get_yolo_boxes及decode_netout：

```
1  def get_yolo_boxes(model, images, net_h, net_w, anchors, obj_thresh, nms_thresh):
2      image_h, image_w, _ = images[0].shape
3      nb_images          = len(images)
4      batch_input        = np.zeros((nb_images, net_h, net_w, 3))
5
6      for i in range(nb_images):
7          batch_input[i] = preprocess_input(images[i], net_h, net_w)
8
9      batch_output = model.predict_on_batch(batch_input)
10     batch_boxes  = [None]*nb_images
11
12     for i in range(nb_images):
13         yolos = [batch_output[0][i], batch_output[1][i], batch_output[2][i]]
14         boxes = []
15
16         for j in range(len(yolos)):
17             yolo_anchors = anchors[(2-j)*6:(3-j)*6]
18             boxes += decode_netout(yolos[j], yolo_anchors, obj_thresh, net_h,
19  net_w)
20
21         correct_yolo_boxes(boxes, image_h, image_w, net_h, net_w)
22         do_nms(boxes, nms_thresh)
23
24         batch_boxes[i] = boxes
25
26     return batch_boxes
```

首先看看上面的get_yolo_boxes，其輸入包括一批圖片、輸入圖片的寬高、anchors、目標檢測閾值、nms(No-Max Suppression)的閾值。

程式碼並不複雜：第6～7行透過utils.py中的一個preprocess_input函式將圖片轉換為適合模型處理的tensor格式；第9行直接用model.predict_on_batch獲得模型的輸出，但這個輸出還不能被直接使用；第12～24行對每一張圖片都將yolos設定為模型輸出中該圖片的相應檢測結果（3個不同尺寸的結果），然後對每個尺寸的輸出（第17行）獲得其對應的anchors；第18行呼叫decode_netout獲得該尺寸下的邊界框；第22行用前面提到過的NMS（No-Max Compression）去掉IoU過小的影像區域，然後把剩下的加入檢測結果中。

decode_netout函式如下：

```
1   def decode_netout(netout, anchors, obj_thresh, net_h, net_w):
2       grid_h, grid_w = netout.shape[:2]
3       nb_box = 3
4       netout = netout.reshape((grid_h, grid_w, nb_box, -1))
5       nb_class = netout.shape[-1] - 5
6   
7       boxes = []
8   
9       netout[..., :2]  = _sigmoid(netout[..., :2])
10      netout[..., 4]   = _sigmoid(netout[..., 4])
11      netout[..., 5:]  = netout[..., 4][..., np.newaxis] * _softmax(netout[...,
12   5:])
13      netout[..., 5:] *= netout[..., 5:] > obj_thresh
14  
15      for i in range(grid_h*grid_w):
16          row = i // grid_w
17          col = i % grid_w
18  
19          for b in range(nb_box):
20              # 第4个值代表是否包含目标（objectiveness score）
21              objectness = netout[row, col, b, 4]
22  
23              if(objectness <= obj_thresh): continue
24  
25              x, y, w, h = netout[row,col,b,:4]
26  
27              x = (col + x) / grid_w
28              y = (row + y) / grid_h
29              w = anchors[2 * b + 0] * np.exp(w) / net_w
30              h = anchors[2 * b + 1] * np.exp(h) / net_h
31  
32              classes = netout[row,col,b,5:]
33  
34              box = BoundBox(x-w/2, y-h/2, x+w/2, y+h/2, objectness, classes)
35  
36              boxes.append(box)
37  
38      return boxes
```

在以上程式碼中，netout是某一尺寸下YOLO預測模型的輸出，我們首先獲得YOLO劃分的grid大小，設定nb_box為3（因為單一尺度下每個錨點都對應3個錨點框），然後對netout做變形，變形後可以得到類似這樣維度的tensor：(13, 13, 3, 85)、(26, 26, 3, 85)、(52, 52, 3, 85)。

另外，13×13、26×26、52×52 是不同尺寸的grid大小，每個尺寸都有3個錨點框，然後85=4+1+80，因為標準的YOLO v3支援80種目標識別，當然這裡實際上只有一種目標。第5行做了個簡單計算來獲得類別數量，但在這裡用不上。

第9～14行做了一些有趣的轉換，我們看到首先對netout的頭兩個輸出netout[…, :2]做了sigmoid運算，也就是把x、y轉換到了(0,1)區間，同樣對confidence做了類似的操作，這在*YOLO9000: Better, Faster, Stronger*[16]一文中做過解釋，在前面介紹的YoloLayer訓練中對預測的x、y和confidence也做了類似的運算。對輸出的類別機率做了一個softmax處理。在第14行透過閾值設定，拋掉了類別機率過小的部分。

第15～36行對每個cell都進行處理；第19行對每個影像區域都進行處理。

我們再次重複YOLO每個影像區域的輸出：

[x, y, w, h, confidence, class_probabilities]

因此第21行的objectiveness實際上是confidence，表示該影像區域包含目標的可能性，若太小，則略過。第27～32行的程式碼都是從影像區域的輸出中獲得相關屬性的。這裡對寬高的計算同樣可以參閱*YOLO9000: Better, Faster, Stronger*[16]一文。

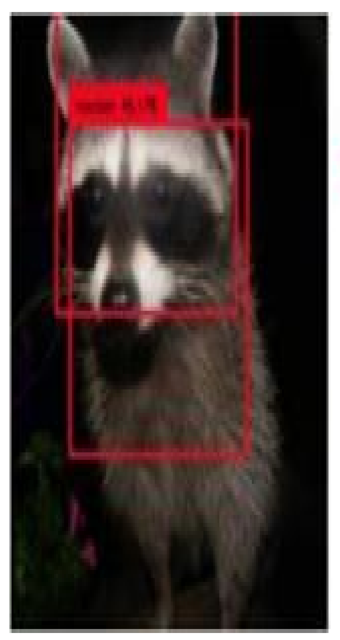
在獲得了影像區域屬性後將其加入返回值中。

最後執行：

```
python predict.py -c config.json
```

我們會看到生成了一個output目錄，裡面對在config.json中定義的valid_image_folder的所有影像都會生成一個對應的識別結果。因為只

訓練了10次，所以檢測效果並不是很好，但仍然可以看到能夠對部分浣熊的面部進行識別，有一定的效果，如圖8-24所示。



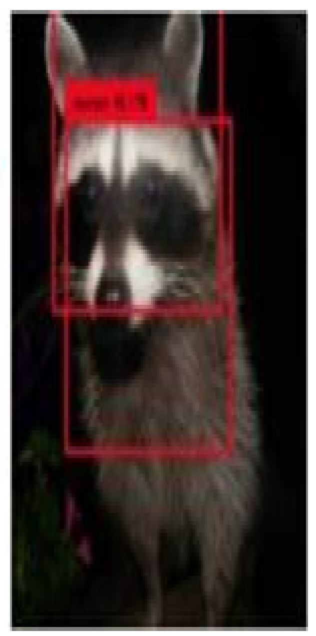


圖8-24　測試效果

8.4　本章小結

本章從目標識別的根源問題和難點入手，完整解釋了目標識別的兩大方式：RCNN系列和YOLO系列，前者為Two Stage的代表，後者為One Stage的代表。

對於RCNN系列，本章從最初的RCNN，講到改進的Fast RCNN，然後解釋了引入RPN的Faster RCNN，並輔以Faster RCNN重點實現的Keras程式碼講解了其模型的核心部分實現。

對於YOLO系列，本章循序漸進地講解了從YOLO v1到YOLO v3的變化。對於YOLO v1，透過Keras程式碼解釋了其核心模型的實現，最後透過一個完整的YOLO v3 Keras實現，深入剖析了訓練過程、模型實現、資料生成、預測流程、YOLO輸出處理等關鍵環節。

讀者在讀完本章後應對目標識別、具體實現和應用有較為完整的理解，並能根據需要對模型的設計和使用進行調整、修改，以滿足不同場景的需要。

8.5 本章參考文獻

[1] 「Rich feature hierarchies for accurate object detection and semantic segmentation」, R.Girshick, et.al, CVPR 2014

[2] 「Selective Search for Object Recognition」, IJCV 2013

[3] 「Efficient Graph-Based Image Segmentation」, IJCV 2004

[4] 「Fast R-CNN」, Ross Girshick, Proc.IEEE Conf.on Computer Vision, 2015

[5] 「Faster R-CNN: Towards real-time object detection with region proposal networks.」, Shaoqing Ren, Kaiming He, R.Girshick, NIPS, 2015

[6] 「Mask R-CNN」, Kaiming He, G.Gkioxari, P.Dollar, R.Girshick, 2017

[7] CS231n: Convolutional Neural Networks for Visual Recognition, http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture11.pdf, Feifei Li

[8] https://github.com/RockyXu66/Faster_RCNN_for_Open_Images_Dataset_Keras

[9] https://github.com/RockyXu66/Faster_RCNN_for_Open_Images_Dataset_Keras/ blob/master/frcnn_train_vgg.ipynb

[10] https://keras.io/layers/writing-your-own-keras-layers/

[11] https://stackoverflow.com/questions/36013063/what-is-the-purpose-of-meshgrid-in-python-numpy

[12] http://www.pyimagesearch.com/2015/02/16/faster-non-maximum-suppression-python

[13] 「You Only Look Once: Unified, Real-Time Object Detection」, J.Redmon, S.Divvala, R.Girshick, A.Farhadi, 2016

[14] https://github.com/subodh-malgonde/vehicle-detection

[15] 「SSD: Single Shot MultiBox Detector」, ECCV 2016

[16] 「YOLO9000: Better, Faster, Stronger」, CVPR 2016

[17] 「YOLO v3: An Incremental Improvement」, CVPR 2018

[18] https://www.youtube.com/watch?v=xVwsr9p3irA

[19] https://github.com/likezhang-public/qconbj2019/blob/master/yolov3/

[20] http://cocodataset.org/

[21] https://pjreddie.com/darknet/yolo/

[22] https://github.com/experiencor/keras-yolo3

[23] https://www.tensorflow.org/api_docs/python/tf

第9章　模型部署與服務

在前面的章節中，我們從基本的機器學習簡單模型談起，逐步過渡到較為複雜的RNN、CNN及目標識別相關模型等，並引用程式碼例項講解了如何將這些模型應用於不同的應用場景。

但是到目前為止，直接基於Keras的程式碼實現仍然不能被稱為產品程式碼，只能用於個人研究，距離產品級服務仍然差了一個重要的環節：模型的轉換和部署。下面將基於TensorFlow Serving來探討機器學習模型在實際生產環境中的部署及服務。

本章要求讀者對Docker環境的使用有基本的瞭解。

9.1　生產環境中的模型服務

對於機器學習研究人員來說，他們重點關注的是模型本身的準確率（Accuracy）、召回率（Recall）等指標。然而對於生產環境（Production Environment）而言，這些是遠遠不夠的。實際上，對於機器學習工程開發人員來說，模型的準確率、召回率預設已經達到可用標準，他們真正關注的實際上是傳統後臺開發人員所關注的併發支援和響應速度。我們可以用表9-1來體現機器學習研究人員和機器學習工程開發人員在技術關注點上的差異。

表9-1　機器學習研究人員和機器學習工程開發人員在技術關注點上的差異

技术关注点	算法研究	工程开发
参数设置	重点	不关注
模型结构	重点	不关注

續表

技术关注点	算法研究	工程开发
数据集	标准数据集	通常需要自行建立
数据标注	标准数据，稳定可靠	自行设计，误差较大
部署过程	不关注	重点
模型大小	不关注	重点
版本控制	不关注	重点
并发性能	不关注	重点
负载均衡	不关注	重点

正因為以上的種種差異，機器學習研究人員在需要將研究成果落地到實際的工程業務中時，往往會遇到諸多問題，例如：

◎　單一問題的模型過大，生產環境的伺服器難以載入多個模型；

◎　模型響應時間過慢；

◎　模型版本控制困難，無法進行A/B測試；

◎　介面引數難以定義，實際應用中的資料轉換複雜。

我們可以用一個簡單的Keras例子來看為什麼會有這些問題。在第7章的影像分類中，我們要建立一個簡單的CNN並進行分類預測，程式碼如下：

```
1    model = create_model(NUM_CLASSES, IMG_SIZE)
2    weight_file = 'gtsrb_cnn_augmentation.h5'
3    model.load_weights(weight_file)
4    y_pred = model.predict([[x]])
```

在以上程式碼中，我們建立了一個CNN讀入網路權重，並對輸入的資料x進行了分類預測。當然，在實際應用中，我們也可以把model作為一個全域性變數，然後對輸入的資料進行處理，從而避免了每次都要重新建立模型的問題。對於一個基於Flask的Web服務，其程式碼可以類似下面的示例：

```
model = create_model(NUM_CLASSES, IMG_SIZE)
weight_file = 'gtsrb_cnn_augmentation.h5'
model.load_weights(weight_file)

@proxy.route('/cnn', methods=["POST"])
def cnn_action():
   payload = request.get_json()
   # 处理 post 参数
   # 将 post 参数中的图片信息（例如以 base64 编码）转换为模型输入格式存入 image_data
   y_pred =  np.argmax(model.predict([[image_data]]))

   return jsonify({'result': y_predict})
```

在以上程式碼中，我們看到所有Web請求都可以呼叫作為全域性變數的model來完成預測分類，因為這是一個無狀態（Stateless）的請求，看上去可以簡單地用Nginx或Haproxy做負載均衡，即可實現水平

擴充套件（Horizontal Scaling）。然而真的這麼簡單嗎？事實上是有問題的。

首先，在上面的程式碼中，在呼叫作為全域性變數的model時，實際上意味著要共享對應的TensorFlow graph和session，這在併發請求較多時會導致模型出錯，這一點在stackoverflow網站的討論中已提到[1]。儘管有多種相應的解決方式，但在實際應用中往往會因為處理這些問題而浪費大量的時間。

其次，基於Python的Keras實現，對於要求處理速度較快的線上服務來說，其處理速度實際上是較慢的，一則Python作為解釋性語言，其本身的執行速度非常慢；二則Python缺乏真正的多執行緒支援[2]，在高速處理資料和業務邏輯的實現上都有很大的限制。

因此，在很多公司早期乃至現在的機器學習服務中，都需要實現專門的機器學習模型服務，並針對機器學習模型的特點和用法實現專門的線上服務架構，以便將機器學習研究人員設計的模型轉變為高效的專用模型檔案，透過自定義的框架對外服務。我們通常把這樣的工作稱為模型服務（Model Serving）。圖9-1展示了一個早期的模型服務架構。

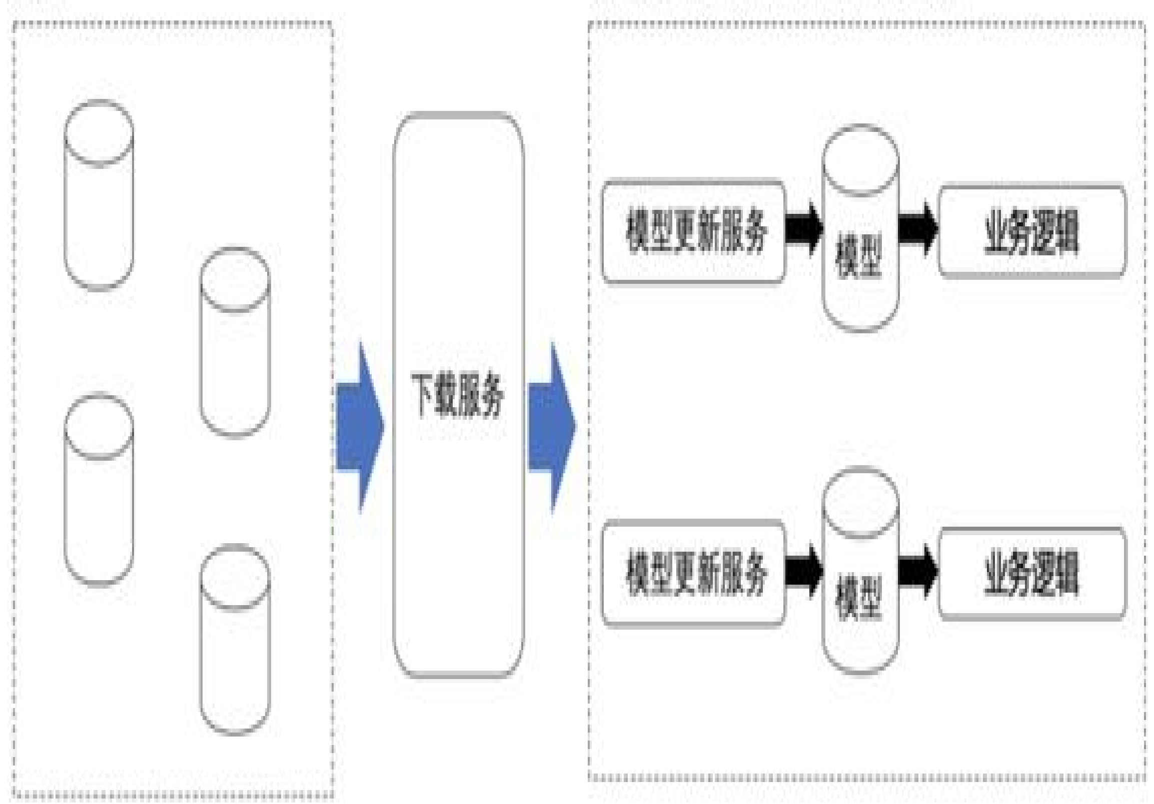


圖9-1　一個簡單的早期模型服務架構

在圖9-1中可以看到，Web應用伺服器在模型伺服器上檢查版本（通常定時檢查）時，如果有新的版本，則進行更新和下載。在模型伺服器上透過上傳管理介面，可以讓模型開發人員上傳更新的機器學習模型並儲存。這裡所指的模型檔案及在Web應用上的呼叫，當然可以直接用Keras的.h5模型檔案並在Web應用框架中加入Keras環境呼叫。然而，由於前面提到的在生產環境中使用Keras的種種問題，在大量的早期機器學習服務中通常用C++或者Java重寫各種需要部署在生產環境中的模型，然後用專門的自定義框架去執行。

出於種種原因，圖9-1所展示的模型服務架構仍然在當前眾多公司內執行，通常作為其自研產品出現。截至2019年的工業界，真正可被廣泛用於實際AI模型服務的通用開發框架只有TensorFlow。更嚴格地說，是TensorFlow Extension（TFX）包中的TensorFlow Serving提供了支援AI模型多版本部署和高併發支援的功能。我們在這裡不對TensorFlow Serving的架構和設計做深入講解，而是直接介紹其實際使用方法。

9.2 TensorFlow Serving的應用

TensorFlow Serving是Google推出的針對機器學習模型線上部署和服務的高效能執行框架，它是第1個真正針對現實生產環境的機器模型執行框架，並與TensorFlow訓練模型無縫銜接（對於Keras模型需要進行一定的轉換，後面會進行講解）。

TensorFlow Serving基於C++開發而成，除了效能上的優勢，它也提供了針對生產環境所需要的版本控制、多種型別介面支援（HTTP/gRPC）、請求打包及對擴充套件性的支援。TensorFlow Serving已在GitHub上開源[3]，不少公司已基於其原始碼根據自己的需求進行定製，比如支援特定型別的模型、載入方式的修改，等等。本章將基於原生的TensorFlow Serving對Keras模型的支援進行講解。注意，本書編寫完成時，TensorFlow 2.0已經於2019年9月正式釋出。本書程式碼主要基於TensorFlow 1.*x*，但同樣適用於2.0。

9.2.1 轉換Keras模型

對於Keras模型，我們可以透過TensorFlow特定的API進行轉換。根據不同的Keras模型，有兩個API可以使用：tf.contrib.saved_model.save_keras_model[4]、tf.saved_model.builder.SavedModelBuilder.save[5]。

其中，save_keras_model的呼叫最為直接，但是對TensorFlow 1.*x*和Keras模型尤其是tf.Keras模型的支援，在我們的實驗中並不友好。因此我們在本章中使用第2個介面，以便更清晰地反映從Keras模型到TensorFlow PB的格式轉換所需的工作。但需要注意在TensorFlow 2.0以後，第1種介面方式可能會得到更好的支援。

要完成從Keras模型到TensorFlow Serving支援的格式轉換，我們需要經過以下幾個關鍵步驟。

（1）讀取Keras模型。

（2）獲得Keras模型的輸入和輸出（在Keras模型中已自帶定義）。

（3）定義TensorFlow模型的輸入/輸出signature（即輸入資料和輸出結果的形式和定義）和方法名稱（method name）。

（4）構建ModelBuilder並儲存。

在以上關鍵步驟的第1～2步中，我們主要是從已經生成的Keras模型中獲得對應的輸入和輸出定義。我們在第6章中已經分析過如何將Keras模型各層的輸入和輸出存為JSON檔案（model.to_json），所以在Keras的model.save()方法中所儲存的模型檔案裡實際上包括了該模型的權重和layer的輸入和輸出定義。這些定義都是可以再次獲取的。然後在第3～4步，既然我們已經獲取了模型定義，所以剩下的就是按照TensorFlow Serving的要求定義模型的輸入和輸出，最後儲存。具體實現可以參考以下程式碼（以下程式碼以第8章的YOLO v3模型為基礎，關於YOLO v3的實現請參考第8章）：

```
1   import os
2   import argparse, json
3   from tensorflow.keras.models import load_model
4   import tensorflow as tf
5
6   def save_with_signature(input_path, output_path):
7       model = load_model(input_path)
8
9       model_input = tf.saved_model.utils.build_tensor_info(model.inputs[0])
10      model_output1 = tf.saved_model.utils.build_tensor_info(model.outputs[0])
11      model_output2 = tf.saved_model.utils.build_tensor_info(model.outputs[1])
12      model_output3 = tf.saved_model.utils.build_tensor_info(model.outputs[2])
13
14      prediction_signature = (
15          tf.saved_model.signature_def_utils.build_signature_def(
16              inputs={'inputs': model_input},
17              outputs={'output1': model_output1, 'output2':model_output2,
18  'output3':model_output3},
19              method_name=tf.saved_model.signature_constants.PREDICT_METHOD_NAME))
20
21      builder = tf.saved_model.builder.SavedModelBuilder(output_path)
22
23      with tf.keras.backend.get_session() as sess:
24          builder.add_meta_graph_and_variables(
25              sess=sess, tags=[tf.saved_model.tag_constants.SERVING],
26              signature_def_map={
27                  'predict':
28                      prediction_signature,
29              })
30
31          builder.save()
```

我們看看以上程式碼都做了什麼。

第1～5行：引入依賴庫。

第7～12行：讀入模型，然後生成對應的tensor定義。注意，我們在這裡已經知道YOLO v3模型包括1個輸入和2個輸出，因此直接硬編碼了對應的變數。在實際專案中，如果我們期望實現一個通用的模型轉換服務，則需要對model.inputs和outputs做一定解析和處理。

第14～18行：這裡定義模型預測的signature。我們看到這裡主要使用了tf.saved_ model.signature_def_utils.build_signature_def函式，並將上面獲取的輸入和輸出tensor資訊作為引數。注意，這裡一定要定義method_name，並使用預定義的PREDICT_METHOD_NAME賦值，確保該模型支援預測（Predict）操作。

第22～32行：首先在第22行建立SavedModelBuilder物件，然後在當前的TensorFlow session中（一定要有）使用上面建立的prediction signature和預定義引數來構建TensorFlow模型的Graph及相關變數，最後儲存。

將模型的檔案路徑和輸出路徑作為引數呼叫以上程式碼，會得到轉換後的檔案：

```
|__variables/*

|__saved_model.pb
```

我們需要把以上檔案放入一個專用目錄中，如：

```
--models
  |_yolo3
  |_1
|_variables
|_saved_model.pb
```

這樣就把模型放到models/yolo3/1下面。在variables目錄中包括TensorFlow網路圖中各個變數的序列化結果；saved_model.pb則是tensorflow.SavedModel的序列化結果，包括模型的完整定義及相關signature等資訊。注意，models/yolo3/1中的「1」代表版本號，在不加限制時，TensorFlow Serving會使用最新的版本號進行服務。

在模型轉換完成後，我們現在看看怎麼部署TensorFlow Serving。

9.2.2 TensorFlow Serving部署

首先，我們要下載TensorFlow Serving的Docker映象。如前所述，我們並不想修改並編譯TensorFlow Serving的程式碼，直接使用其Docker映象即可：

```
docker pull tensorflow/serving
```

執行以上命令拉取TensorFlow Serving映象。在執行以上命令前需要安裝Docker [6]。

我們在啟動該映象時只需掛載模型檔案的路徑。注意，TensorFlow Serving在啟動時可以指定需要載入的模型路徑，也可以把所有要載入的模型寫到一個config配置檔案中，如：

```
model_config_list {
config {
   name: 'yolo3',
   base_path: '/models/yolo3/',
   model_platform: "tensorflow"
}}
```

在上面的配置檔案中給出了要載入的模型名稱、路徑和模型型別，將其存為tfs_model.config，然後執行命令啟動Docker：

```
docker run
-p 8500:8500
-p 8501:8501
--mount type=bind,source="$(pwd)"/serving/yolo3,target=/models/yolo3
--mount
type=bind,source="$(pwd)"/tfs_model.config,target=/models/models.config
-t tensorflow/serving
--model_config_file=/models/models.config
```

我們首先暴露8500和8501兩個埠以便使用TensorFlow Serving的服務介面（8501是Web介面，8500是GRPC介面）；然後使用mount引數指定要掛載的模型路徑和模型配置檔案路徑；最後透過設定Docker容器的額外引數指定要載入的model_config_file。

9.2.3 介面驗證

我們來看看怎麼呼叫TensorFlow Serving的介面，並且驗證其執行流程和本地的Keras模型一致。

首先看看如何檢查模型已經成功部署，執行以下命令：

```
curl -X GET http://localhost:8501/v1/models/yolo3/metadata
```

可以獲得模型資訊：

```
{
  "model_spec": {
  "name":"yolo3",
  "signature_name": "",
  "version": "1"
  }
}
```

以及介面定義資訊：

```
"metadata":{"signature_def":{
"signature_def": {
  "predict":{
  "inputs":{
  //...
  },
  "outputs":{
  "output1":{
  "dtype": "DT_FLOAT",
  "tensor_shape": {
  //...
  }
  "name":"conv_81_BiasAdd:0"
  },
  "output2": {
  //...
  }
  "output3":{
  //...
  }
  },
  "method_name":"tensorflow/serving/predict"
  }
 }
}
```

為了避免資訊重複，在以上程式碼中沒有顯示所有介面定義的內容。可以看到，我們透過呼叫TensorFlow的metadata介面，能夠獲取模型的詳細定義。注意，我們在9.2.1節提到如何定義模型的signature，也就是輸入、輸出的資料格式。這裡透過返回結果中signature_def屬性的inputs和outputs引數就可以檢驗其是否和9.2.1節中轉換程式碼所定義的一致。

然後看看具體怎麼檢查TensorFlow Serving的模型輸出。第8章分析過如何編寫predict.py函式來使用本地的Keras模型進行預測，這裡將其稍做修改，存為predict_tfs.py，將本地的Keras模型呼叫改為對線上TensorFlow Serving的HTTP介面呼叫：

```
import os, sys
import requests, json
import cv2
from utils.utils import get_yolo_box_tfs, makedirs
from utils.bbox import draw_boxes
import numpy as np

anchors=[]
with open('anchors.json') as anchors_str:
   anchors = json.load(anchors_str)

net_h, net_w = 416, 416
obj_thresh, nms_thresh = 0.4, 0.45

TFS_URL="http://localhost:8501/v1/models/yolo3:predict"
img_path = sys.argv[1]
img_data = cv2.imread(img_path)

boxes = get_yolo_box_tfs(TFS_URL, img_data, net_h, net_w, anchors,
obj_thresh, nms_thresh)

draw_boxes(img_data, boxes, ["raccoon"], 0)
cv2.imwrite('./output/' + img_path.split('/')[-1], np.uint8(img_data))
```

以上程式碼和在第8章中所講到的predict.py的主要區別在第15～19行。我們建立了新的函式get_yolo_box_tfs，將圖片資料和TensorFlow Serving的URL地址傳入並獲得返回。該函式的實現如下：

```
1    def get_yolo_box_tfs(tfs_url, image_data, net_h, net_w, anchors, obj_thresh,
2    nms_thresh):
3        image_h, image_w, _ = image_data.shape
4
5        input_data = np.expand_dims(image_data, axis=0)
6        input_data = mobilenet.preprocess_input(input_data)
7
8        input_data = {
9        'signature_name': 'predict',
10       'instances':input_data.tolist()
11       }
12
13       response = requests.post(tfs_url, json=input_data)
14       response_data = json.loads(response.text)
15       output1 = response_data['predictions'][0]['output1']
16       output2 = response_data['predictions'][0]['output2']
17       output3 = response_data['predictions'][0]['output3']
18
19       outputs = [output1, output2, output3]
20       boxes = []
21
22       for i in range(len(outputs)):
23           yolo_output = np.array(outputs[i])
24           bboxes = decode_netout(yolo_output, anchors, obj_thresh, net_h,net_w)
25           boxes += bboxes
26
27       correct_yolo_boxes(boxes, image_h, image_w, net_h, net_w)
28       do_nms(boxes, nms_thresh)
29
30       return boxes
```

我們看到，首先在第3～6行，我們獲取了圖片的寬和高，並利用tensorflow.keras.applications.mobilenet.preprocess_input函式對圖片進行了預處理。當然，我們也可以像第8章那樣自己實現preprocess方法，實際上這只是對影像資料進行了縮放，但這裡可以直接採用mobilenet的預定義方式。

第8～11行定義了輸入資料。注意，TensorFlow Serving的輸入有多種形式，其中最通用的就是定義signature_name和instances這兩個屬性。

然後，我們就可以如第13行那樣，利用Python的requests包中的post方法呼叫TensorFlow Serving的伺服器介面獲得返回。這裡要注意TensorFlow Serving的URL結構，目前我們呼叫過兩個URL的地址：http://localhost:8501/v1/models/yolo3/metadata 和 http://localhost:8501/v1/models/yolo3:predict。其中，v1代表TensorFlow的1.*x*版本，models/yolo3則與我們在config配置中對指定的模型定義的名字「yolo3」相對應。程式碼如下：

```
{
  name: 'yolo3',
  base_path: '/models/yolo3/',
  model_platform: "tensorflow"
}}
```

然後我們可以用/metadata路徑獲取模型的介面屬性，或呼叫:predict進行預測。

第14～17行則是對返回值的處理，我們看到跟之前的設定一樣，獲取了3個返回結果output1/output2/output3，分別對應YOLO v3中3種不同縮放尺度的識別結果。其他步驟則和第8章中的predict.py類似，這裡不再贅述。

最後，我們需要驗證線下線上預測的一致性。我們先用predict.py透過本地模型對本地的一張測試圖片進行一次預測：

```
python predict.py -c config.json -i ./data/test1.jpg
```

獲得輸出：

```
Output tensor shape:(13, 13, 18)
0.5938352
0.5915714
0.59009105
Output tensor shape:(26, 26, 18)
0.5744155
0.5738328
0.5718676
Output tensor shape:(52, 52, 18)
0.5602994
0.5489948
0.54830253
```

然後用predict_tfs.py呼叫TensorFlow Serving進行預測：

```
python predict_tfs.py ./data/test1.jpg
```

獲得輸出：

```
Output tensor shape:(13, 13, 18)
0.5938352358851674
0.591571334046146
0.5900911019659046
Output tensor shape:(26, 26, 18)
0.5744156261772013
0.5738327248846318
0.5718675993838211
Output tensor shape:(52, 52, 18)
0.5602993401426218
0.5489947751588031
0.5483026401920875
```

對比兩組輸出，我們可以看到其結果幾乎一致，這也驗證了模型轉換後線上線下的一致性。

9.3 本章小結

本章首先簡單介紹了機器模型線上服務中的難點和問題，以及TensorFlow Serving的意義；然後以第8章中YOLO v3模型的線上部署為例，覆蓋了模型轉換、TensorFlow Serving配置與啟動、介面驗證和線上線下一致性驗證的各個部署流程。我們必須注意，在現實專案的開發流程中，模型的轉換和部署是很多機器學習研究人員進行大規模部署時所遇到的障礙之一，希望本章能為進行實際機器學習工程開發的讀者帶來一些啟發和幫助。

9.4　本章參考文獻

[1] https://stackoverflow.com/questions/56137254/python-flask-app-with-keras-tensorflow-backend-unable-to-load-model-at-run

[2] https://wiki.python.org/moin/GlobalInterpreterLock

[3] https://github.com/tensorflow/serving

[4] https://tensorflow.google.cn/api_docs/python/tf/keras/experimental/export_saved_model

[5] https://www.tensorflow.org/api_docs/python/tf/saved_model/Builder

[6] https://www.docker.com/

Broadview® www.broadview.com.cn **博文视点 · IT出版旗舰品牌**

技术凝聚实力 · 专业创新出版

专家好评

本书对于AI初学者来说是一本很好的入门书，对于AI大咖来说是一本很好的理论联系代码的参考书。作者在书中介绍AI概念、动手写代码及测试代码时下了一番苦功，在阐述算法的背景和内容时既有深度又有直观形象的介绍，对数学公式的引用对于有程序员背景的读者来说恰到好处，并且在实际代码的解释上紧扣主题、讲解清晰。

——**龙门博士**，Broadcom首席工程师

机器学习已经成为计算机视觉、自然语言处理和AI领域的重要基石，但是其抽象的理论让很多初学者望而却步。本书将机器学习中的经典理论与实践相结合，由浅入深地介绍了每种理论的原理、代码实现和应用，让初学者即刻体验和实践算法，在实践中深入理解和熟练掌握机器学习理论，为今后进行机器学习应用打下扎实的基础。

——**卢亦娟**，微软Cloud AI首席科学家、美国德克萨斯州立大学计算机系教授

这是一本干货满满且附带详细实例的深入浅出机器学习的优秀参考工具书。

——**蒋良骏**，Walmart电商平台高级架构师、蚂蚁金服硅谷中心前技术专家

本书通俗易懂，并且覆盖了AI在多个领域的应用场景，是非常好的AI程序员入门教材。

——**耿秀波**，微软高级应用科学家

读者服务

微信扫码回复：38270

· 获取博文视点学院20元付费内容抵扣券

· 获取免费增值资源

· 加入本书读者交流群，与作者共同交流

· 获取精选书单推荐

责任编辑：张国霞

封面设计：吴海燕　马　俊

上架建议：计算机与互联网>AI技术

ISBN 978-7-121-38270-3

定价：109.00元