

深度学习在图像识别中的应用

戚锦秀

院(系): 航天学院控制科学与工程系 专 业: 自动化

学 号: 1110410427 指导教师: 杨旭东

2015年6月29日

哈爾濱工業大學

毕业设计（论文）

题 目：深度学习在图像识别中的应用

专 业 自动化

学 号 1110410427

学 生 戚锦秀

指导教师 杨旭东

答辩日期 2015年6月29日

摘要

深度学习是近年来人工智能研究领域的一个热点，这个机器学习的分支以全连接的深度置信网络和局部连接的卷积神经网络为代表，融合概率图模型、马尔可夫链蒙特卡罗方法等多个技术，其基于大规模数据而训练出的模型在声音识别、图像识别等众多领域中获得了突破性成果。本文主要研究了深度置信网络及卷积神经网络的工作原理，并在MNIST与CIFAR数据集上进行模型的训练，利用GPU与CPU的异构计算在MNIST数据集上取得了98.9%的识别正确率，在CIFAR-10数据集上取得了62%的识别正确率。在本文中，我们还讨论了包括降维在内的多个神经网络设计技巧以及实验中发现的一些现象。

关键词： 深度学习；受限玻尔兹曼机；深度置信网络；卷积神经网络；MCMC；GPU计算

Abstract

Deep learning as a branch of machine learning which is represented by a full connected neural network named Deep Belief Networks as well as a local connected network named Convolutional Neural Networks has drawn lots of attention in the field of artificial intelligence. Multiple technologies like probabilistic graphical models and Markov Chain Monte Carlo methods have integrated into deep learning nowadays and they help deep learning make a great breakthrough in many AI tasks such as speech and image recognition by training models based on large-scale data. This paper focus on the principles of Deep Belief Networks and Convolution Neural Network s, and we trained some models on the MNIST and CIFAR-10 dataset using DBNs or CNNs. As a result, we achieved 98.9% recognition accuracy on MNIST dataset and 62% recognition accuracy on CIFAR-10 dataset with the help of GPU&CPU-based heterogeneous computing. A number of phenomena found in our experiments and tricks of designing the neural networks, including dimensionality reduction, will also be discussed in this paper.

Keywords: deep learning, restricted boltzmann machines, deep belief networks, convolutional neural networks, Markov chain Monte Carlo, GPU computation

目 录

| | |
|---------------------------------|----|
| 摘要..... | I |
| ABSTRACT | II |
| 第1章 绪论 | 1 |
| 1.1 课题来源及研究的目的和意义 | 1 |
| 1.2 国内外在该方向的研究现状及分析..... | 3 |
| 1.2.1 神经网络与深度学习的发展状况..... | 3 |
| 1.2.2 感知器时期..... | 3 |
| 1.2.3 反向传播时期 | 5 |
| 1.2.4 深度学习时期 | 5 |
| 1.3 深度学习在人工智能上的应用 | 6 |
| 1.3.1 语音识别 | 6 |
| 1.3.2 图像识别 | 6 |
| 1.3.3 自然语言处理 | 7 |
| 第2章 控制论与机器学习 | 8 |
| 2.1 白盒模型与经典控制论 | 8 |
| 2.2 灰盒模型与系统辨识..... | 9 |
| 2.3 黑盒模型与统计机器学习 | 9 |
| 2.4 本章小结 | 11 |
| 第3章 受限玻尔兹曼机 | 12 |
| 3.1 伊辛模型 | 12 |
| 3.2 玻尔兹曼机..... | 13 |
| 3.3 受限玻尔兹曼机 | 14 |
| 3.4 本章小结 | 19 |
| 第4章 马尔可夫链蒙特卡罗方法 | 20 |
| 4.1 蒙塔卡罗方法核心思想 | 20 |
| 4.2 舍弃采样 | 21 |
| 4.3 重要性采样 | 25 |
| 4.4 马尔可夫链 | 27 |
| 4.5 Metropolis-Hastings算法 | 30 |

| | |
|----------------------------|-----------|
| 4.6 Gibbs采样 | 33 |
| 4.7 对比离差 | 35 |
| 4.8 本章小结 | 36 |
| 第5章 深度置信网络 | 37 |
| 5.1 神经网络组成及表达能力 | 37 |
| 5.1.1 神经元 | 37 |
| 5.1.2 逻辑表达 | 39 |
| 5.2 神经网络的前馈 | 40 |
| 5.2.1 神经激活 | 41 |
| 5.3 分类器 | 45 |
| 5.3.1 平方误差分类器 | 45 |
| 5.3.2 softmax分类器 | 46 |
| 5.4 神经网络的反馈 | 48 |
| 5.4.1 分类器参数校正 | 49 |
| 5.4.2 误差传播 | 51 |
| 5.5 深度置信网络 | 53 |
| 5.6 本章小结 | 56 |
| 第6章 卷积神经网络 | 57 |
| 6.1 卷积神经网络综述 | 57 |
| 6.2 卷积神经网络的前馈 | 59 |
| 6.2.1 卷积 | 59 |
| 6.2.2 采样 | 62 |
| 6.2.3 分类器 | 63 |
| 6.3 卷积神经网络的反馈 | 63 |
| 6.3.1 分类器误差传播 | 63 |
| 6.3.2 采样层误差传播 | 64 |
| 6.3.3 卷积层误差传播 | 64 |
| 6.4 本章小结 | 65 |
| 第7章 神经网络的设计技巧 | 66 |
| 7.1 数据预处理 | 66 |
| 7.1.1 降维 | 67 |
| 7.1.2 扩容 | 74 |

| | |
|-----------------------------------|------------|
| 7.2 训练技巧 | 75 |
| 7.2.1 学习率 | 75 |
| 7.2.2 动量项 | 77 |
| 7.2.3 权衰减 | 78 |
| 7.3 编码技巧 | 79 |
| 7.3.1 面向对象技术 | 79 |
| 7.3.2 梯度校验 | 80 |
| 7.4 本章小结 | 81 |
| 第 8 章 GPU计算 | 82 |
| 8.1 GPU体系结构..... | 82 |
| 8.2 CUDA | 85 |
| 8.3 Cudamat..... | 87 |
| 8.4 Gnumpy | 87 |
| 8.5 PyCUDA..... | 88 |
| 8.6 Caffe | 89 |
| 8.7 本章小结 | 90 |
| 第 9 章 实验现象及讨论 | 91 |
| 9.1 数据集简介 | 91 |
| 9.1.1 MNIST | 91 |
| 9.1.2 CIFAR-10 | 92 |
| 9.2 深度置信网络在MNIST数据集上的性能..... | 93 |
| 9.3 卷积神经网络在MNIST数据集上的性能..... | 96 |
| 9.4 卷积神经网络在CIFAR-10数据集上的性能 | 98 |
| 9.5 使用Caffe实现的CIFAR-10数据集训练 | 101 |
| 9.6 本章小结 | 103 |
| 结 论 | 104 |
| 参考文献 | 105 |
| 原创性声明 | 109 |
| 致 谢 | 110 |
| 附录 A 深度置信网络源代码 | 111 |
| 附录 B 卷积神经网络源代码 | 127 |

第1章 绪论

1.1 课题来源及研究的目的和意义

机器学习以及模式识别均属于人工智能范畴，机器学习源自于计算机科学，模式识别源自于工程学，尽管这两者源自于不同的背景，但这两者可以认为是同一个领域下的不同描述^[1]。自计算机出生以来，人们一直都在探索一个问题：机器能否实现智能？早期的一部分研究人员试图从事物的机理出发，寻找模式背后的规律，比如在自然语言处理领域，最初人们从分析语法语义开始，企图让机器理解自然语言，并能实现一些类似于翻译之类的工作，但失败了，一个重要的原因是自然语言的规则错综复杂，要摸清其运作机理哪怕以目前的学科水平来看也几乎是难以实现的。

在随后的二三十年里，人类乐观地认为可以找到自然语言背后的运作机理，但实验结果都不尽人意，机理建模这条道路看起来像是一条死胡同。到了上世纪九十年代，统计机器学习开始成为主流，经过二十多年的高速发展，目前基于统计的建模方法成为了主流，而统计机器学习取得的成就远大于上世纪六十年到到九十年代取得的成果，当然这也不排除计算机运算能力的影响。

相比于机理建模，统计机器学习的出发点是从数据出发，建立一个可以刻画已有数据的概率分布，如果将机理建模比作是牛顿力学，那么统计学习就相当于统计力学。事实上，从统计的角度上去解决人工智能问题并不是一个新的想法，早在上世纪七十年代统计学习就已经开始出现。统计机器学习使我们不再需要研究模式背后的运作机理，并且通过它可以实现通用学习。如果说一个事物背后有一个函数决定了它的特性，那么我们不再需要研究这个函数究竟是什么形式，只要有足够多的数据，通过统计方法，选取恰当的模型，我们便可以在一定程度上拟合出这个函数，尽管我们拟合出的函数在大多数情况下都不等价于事物的本质函数，但如果它能在可允许的误差范围内正常工作，这就足够了。

统计机器学习方法的出现，使得工作重点转为寻找合适的模型上，事实上这也是统计机器学习其本质：经验风险最小化^[2]。也就是说，我们应该如何寻找一个合适的模型，使得测试误差最小化？由于模型的选取取决于设计者的意愿，我们既可以选择一个简单的模型，比如线性函数，也可以选择一个复杂的函数，比如高次多项式函数，但是实际问题中，大多数情况下，如果使用简单的线性函数，

数据在低维空间中是不可分的，为了使数据变得线性可分，我们可以将数据由低维空间映射到高维空间中，但这种方案引入了庞大的运算以至于计算机难以计算，我们称这种现象为“维数灾难”，即随着维数的线性增加，计算量指数地增大。又或者，我们一开始就不使用线性函数而使用较为复杂的函数比如高斯函数实现非线性分割，但由于我们不知道数据的分布形式，也难以将实际问题中的高维数据可视化，所以除非我们有关于数据分布的先验知识，否则并不知道我们选取的这个复杂的函数是否是合理的。

为了解决维数灾难，我们可以使用核函数，在低维空间中直接计算高维空间返回的结果，避开了高维空间的运算，这种方法也被称之为核方法，比如上世纪九十年代提出的支撑向量机（kernel-SVM）就是一种使用了核方法的分类器。而如果为了避免复杂函数的选取问题，我们可以使用神经网络。神经网络基于一种假说——智能源自于单一的算法，在神经网络中，我们只需选取激活函数，即所谓的“单一的算法”，然后训练网络的连接权值，从而跳过函数选取步骤。

关于神经网络的表达能力方面，在理论上，柯尔莫格洛夫（Kolmogorov）证明了：只要给予足够多的神经元、合适的激活函数以及恰当的权值，任何从输入到输出的连续映射函数都可以用三层神经网络实现。如果从傅里叶理论上看，则相当于：任何连续函数都可以用足够多的谐波来逼近^[3]。

柯尔莫格洛夫的定理说明了神经网络的表达能力与神经网络的层数无关，只与神经元的数目有关，似乎我们没有必要加深网络的深度，但Hastad与Goldmann在1991年提出了以下定理^[4]

定理 1.1 为了计算一个函数 $f_k \in \mathcal{F}_{k,N}$ ，深度为 $k - 1$ 的单调加权门限电路所需的规模至少为 2^{cN} ，其中常数 $c > 0$ ， $N > N_0$

由于神经网络与电路的联系十分密切，如果将定理1.1 转化为神经网络的术语，则为：深度为 k 的神经网络所能表达的函数，深度为 $k - 1$ 的神经网络为了达到同样的效果至少需要引入指数级规模的节点。

倘若从大脑科学的角度思考，我们已经知道，哺乳动物的大脑是一个深度结构，它可以将输入的模式在多个层次上进行抽象再表达，而每个层次对应着皮层的不同区域。以大脑在视觉系统上的工作机理为例，其工作过程依次经过如下几个步骤：边界检测、基本外形建立、逐渐地完善更为详尽外形^[5]。

在定理1.1以及大脑科学的启发下，我们对深度神经网络充满信心，因此神经网络研究人员花费数十年致力于深度神经网络的研究，然而直到2006年之前^[6, 7]，我们都难以找到一个较好的算法来训练深度神经网络，其原因一方面是深度神经网络局部最优解繁多，导致深度神经网络比浅层神经网络更容易陷入局部最小值，

另一方面则是受计算机运算能力的限制^[8]。

2006年Hinton提出了受限玻尔兹曼机以及深度置信网络的训练方法，通过贪婪逐层地训练^[9]，成功地实现了第一个深度神经网络，随后深度神经网络又发展处深度卷积网络、稀疏自动编码机^[10]等方法，这些方法被统称为深度学习。目前，深度学习已应用到分类、回归、聚类、降维^[11]、自然语言处理、信息检索、机器学习等任务中，且在某些领域中其应用效果已远远超越以前的机器学习方法。

1.2 国内外在该方向的研究现状及分析

2015年5月，Nature为纪念人工智能60周年开辟了“人工智能与机器人”专题，回顾了过去的研究历程与当今的研究热点，其中深度学习作为一个热点被加以介绍^[12]。目前深度学习在国内外都受到广泛关注，新的深度学习算法源源不断地被提出，在不久的将来，深度学习将会取得更多的成功。

1.2.1 神经网络与深度学习的发展状况

数十年来，神经网络研究人员在神经科学上付出了巨大心血，其研究之深入，范围之广，内容之繁杂是别的机器学习方法所难以企及的。我们今天看到的深度学习只是神经计算科学中的冰山一角，在它之下，是几十年探索积累起来的失败经验，这些模型中不乏观点新颖却无法工作的模型。这么多年来研究人员试图将神经网络理论化，然而到目前为止这项工作依然没有实现。

1.2.2 感知器时期

关于神经网络的想法由来已久，远在1800人们已经开始研究神经网络，1943年心理学家Warren McCulloch与数理逻辑学Walter Pitts在总结前人关于神经工作原理的基础上将神经元工作数理化，随后冯·诺依曼在领导计算机设计时试图使用神经网络的形式而不是目前的指令形式^[13]，限于当时科技水平限制最后并没有采用神经网络的方案。随着二十世纪六十年代控制论兴起以及计算机的出现，联接主义学者迫切地期待建立一个可用的神经网络数学模型，在这个环境中，受Warren McCulloch与Walter Pitts两人工作的启发下，Frank Rosenblatt提出了第一个神经元模型——感知器^[14]，这标志着神经网络进入第一次浪潮。

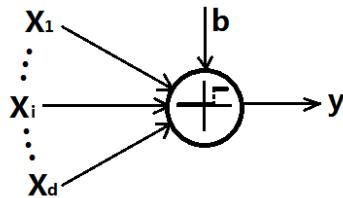


图 1-1 感知器模型

如图1-1 所示，在感知器中，含有 d 个输入和1个输出，其中非线性激活函数为阶跃函数，即

$$y = \begin{cases} 1 & \text{若 } \theta^T x - b \geq 0 \\ 0 & \text{其他} \end{cases} \quad (1-1)$$

式中 b 为偏置项，有时也称之为阈值。如果从神经科学的角度上解释，在感知器中，向量 x 相当于神经元接收到的刺激，而 $\theta^T x$ 相当于刺激的叠加。当总的刺激量到达一定阈值 b 时，神经元被激活，即 $y = 1$ ，否则神经元对刺激不作反应，即 $y = 0$ 。

伴随感知器一同被提出的还包括感知器训练算法，其方法借鉴了生理学中关于反馈方面的思想，通过引入奖励与惩罚的概念，当某个训练样本被正确分类时，参数维持不变，若该样本被错误分类，则通过惩罚项对参数进行更新^[2]。

得益于非线性的激活函数，感知器网络可以实现逻辑功能以及非线性分类任务。例如，在输入维度为2的感知器网络中，若网络参数 $\theta = [-2, -2]^T$ ，偏置 $b = 3$ ，此时，这组参数构成的感知器即为门电路中的与非门。由于与非门可以构成与、或、非三种基本逻辑门，而这三者又可以构成所有的逻辑函数，因此我们有理由相信，恰当的感知器网络可以实现逻辑功能。事实上，六十年代感知器网络的提出，也确实解决了一些简单的人工智能任务。

神经网络第一次浪潮只持续了大约十年左右，究其原因，一方面，六十年代是电子计算机刚刚兴起的年代，人们更多地关注于计算机，导致感知器网络得不到重视，另一方面，限于电子管及早期晶体管的工艺，耗费大量的资源来建立一个感知器网络似乎也不太现实。此外，人们一直没有找到一种能较好地训练多层感知器网络的算法，导致了大批神经网络学者对其失去信心，因此，在六十年代末期，神经网络的研究进入第一个低谷。

1.2.3 反向传播时期

神经网络研究在八十年代进入第二次浪潮，最重要的原因是1986年Hinton、Remelhart和Williams等人提出了反向传播算法^[15]，首次在手写数字识别任务上使用反向传播训练的神经网络，取得了较好的效果。此外，大规模集成电路的发展以及sigmoid等新的激活函数的应用也很大程度程度上推动了神经网络的发展。

反向传播算法通过链式求导，将偏导数从高层传送到低层，使得多层感知器网络中可以使用简单的梯度下降方法进行训练，尤其是在三层感知器网络中效果显著。随后，感知器网络也改名为神经网络，尽管如此，神经网络只是人为主观地对神经元建模，其本质与生物的神经系统是没有太大联系的，例如神经具有局部稀疏性而神经网络是非局部的，因此神经网络在生物学上被批评为缺乏真实性。

在这个时期，新的模型不断地被提出，比如使用Hebb规则全连接的反馈网络Hopfield网络^[16]、将Hopfield网络随机化后的玻尔兹曼机^[17]、具备模式变换不变性的卷积网络等^[3]，此外，一些新的技术例如快速传播、共轭梯度等也被应用到神经网络中。

九十年代人们开始尝试建立深度神经网络，但随着网络深度的加大，局部最优解增多，在参数没有被恰当地初始化的情况下，使用反向传播算法进行简单的梯度下降容易陷入局部极小值，人们难以找到一种较好的方法来训练深度网络。在硬件方面，九十年代计算机的计算能力仍不足以满足深度神经网络中大规模的运算。此外，同时期被提出的SVM方法可以在较少运算的前提下实现高性能的分类，因此，在九十年代中期，神经网络进入第二次低谷。

1.2.4 深度学习时期

深度神经网络的僵局一直持续到2006年，Hinton与Osindero在深度置信网络中使用一种无监督的逐层贪婪预训练受限玻尔兹曼机，使得权值被初始化到一个合适的位置，随后对整个网络使用全局的反向传播进行有监督的微调，在MNIST手写数字识别任务上超越了已有的记录^[6]，随后，在受限玻尔兹曼机的基础上又发展出了稀疏编码与自动编码机等方法。深度置信网络是一种无监督学习，对应的，由LeCun等人提出的深度卷积网络实现了在深度结构下的有监督学习^[18, 19]，成为第一个真正意义上的深度结构。

自2006年以来，深度学习的研究持续升温^[20, 21]，Google、Facebook、Microsoft等公司以及麻省理工、斯坦福大学、多伦多大学等高校投入大量资

源到深度学习的相关研究中，麻省理工大学在2013年将深度学习评为十大突破技术的榜首^[22]。由于深度学习需要训练样本容量足够大，而近年来大数据时代的到来，人们难以处理规模如此庞大的数据，所以某种程度上，深度学习与大数据是相辅相成的。

2012年前后，国内各大公司逐渐投入到深度学习相关的研究中，百度于2013年开展百度大脑计划，成立“深度学习研究院”，这是百度首次成立研究院，并于2014年邀请了深度学习的领军人物吴恩达作为首位院长。腾讯公司于2014年开发的Mariana平台将致力于研究深度学习在语音识别、图像识别和广告推荐上的应用。2015年京东将开展“JIMI”机器人项目，企图将深度学习应用到人工客服上。

1.3 深度学习在人工智能上的应用

深度学习在人工智能领域正在取得重大进展，利用这套技术解决了历史遗留的大量问题，目前已被应用到科学、商业、政策上。除了在传统的声、图、文领域深度学习掀起巨浪外，深度学习逐步开始渗入到生物学等其他领域。

1.3.1 语音识别

传统的语音识别一般使用混合高斯模型（GMM）或隐马尔可夫模型（HMM）^[3]，这些模型一般都比较简单，难以实现较高的正确率。Microsoft于2009年与Hinton在语音识别方面展开合作，并于2011年实现了深度学习在语音识别上的应用，该应用已经投入到WindowsPhone的语音识别“小娜”中。Google、百度也开展语音识别框架的改革，Google使用4~5层的神经网络，而百度使用9层的神经网络^[22]。使用深度神经网络实现的语音识别在错误率上要比传统方法的错误率相对少20% ~ 30%左右。

1.3.2 图像识别

在声、图、文三个领域中，深度学习在图像识别上对识别效果的提升最显著。在MNIST手写数字识别任务中，使用传统识别方法的正确率一般为98%左右，使用SVM能达到98.6%的正确率，而使用深度学习方法在最好的情况下能达到99.79%的正确率。在ImageNet1000任务中，对于1000个类别的识别，随着深度学习的研究进展，其识别效果分别经历了72%、85%、89%、93%等阶段，截止

到2015年1月，由微软亚太研究院实现的最好效果为96.06%^[23]。在SVHN街景门牌号的识别任务中，Google通过11层的神经网络对门牌号实现了97.84%的正确率，这个系统已经帮助Google从街景中分析出全球近1亿个门牌号。

1.3.3 自然语言处理

Microsoft于2012年在天津展示了使用深度学习训练的同声翻译系统，其英中翻译较传统方法更为流畅。然而，目前在自然语言处理领域，深度学习取得的效果并没有显著超越传统方法。相比于图像识别以及语音识别，自然语言处理一个难点在于上下文，尽管深度学习被认为是一种特征学习，但这种特征学习似乎并没有实现可以联系上下文的功能。自然语言处理在近期广受关注，深度学习研究人员认为下一个突破口将会出现在自然语言处理上。

第2章 控制论与机器学习

控制论其本质，在于利用已有的信息抑制系统熵的增加^[24]，需要注意的是，这里的熵并不是指热力学中的热力熵或信息论中的信息熵，而应理解为熵的原始定义，即对混乱的度量。既然是度量，便需要一个准则，正如测量一个物体的长度需要一把刻度尺，但不同的刻度尺将得到不同的测量结果，同理，对混乱的不同定义，也将得到不同的控制效果。例如，在温度控制系统中，让温度保持在一个较高的温度或较低的温度均是抑制熵增的行为，因为两者对熵的定义不同。对于孤立系统，熵总是朝着增加的方向移动，但也有例外，此时系统必须是非孤立的，能够收到外界的控制，比如有机体具有熵减趋势，因为DNA是有序的（尽管我们目前仍无法完全理解它），其中起到控制作用的外力是自然选择。控制论所要做的，便是引入外力作用，即反馈，或者说控制环节，以之降低系统的混乱程度，得到我们的期望输出。

2.1 白盒模型与经典控制论

经典控制论中，为了实现对一个系统的控制，我们往往需要先由系统的结构开始分析，利用牛顿力学、电磁原理、热力学等物理原理对系统特性建立微分方程，将其转换为复域的传递函数后，再利用根轨迹或频域分析设计控制环节。如图2.1 所示的简单RLC网络中，假定我们以 $u_i(t)$ 作为输入， $u_o(t)$ 作为输出，那么此时我们所拥有的信息是整个系统的结构，所需要抑制的混乱就是控制 $u_o(t)$ 朝着我们的期望输出迈进。

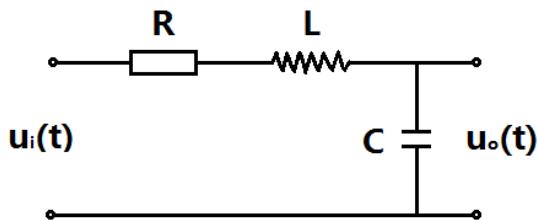


图 2-1 简单的RCL网络

由于这个系统的结构是已知的，整个系统对于我们而言相当一个白盒。为此，我们可以很容易地根据基尔霍夫回路电压定律建立方程

$$u_L(t) + u_R(t) + u_o(t) = u_i(t) \quad (2-1)$$

进一步利用电感特性、电容特性、欧姆定律，式(2-1)可以进一步推导为

$$LC \frac{d^2 u_o(t)}{dt^2} + RC \frac{du_o(t)}{dt} + u_o(t) = u_i(t) \quad (2-2)$$

此时，输入输出的微分方程已建立，下一步便是经典控制论的内容，我们并不打算继续展开细说。在这个例子中，我们可以看出，白盒模型对整个系统是了如指掌的，是可以对其建模的。此时，“利用已有信息抑制系统的熵增”相当于，对系统建立传递函数（利用已有信息），设计控制环节使我们能得到期望输出（抑制熵增）。

2.2 灰盒模型与系统辨识

牛顿一生的工作对自然科学的贡献是无以衡量的，但牛顿生活的时代忽视了一些重要的东西—统计的概念。在牛顿力学中，有一个前提，系统的状态是可测量的，因此，牛顿力学基于一个已给定的精确初始状态之上对系统分析。然而，物理的测量从来都不是精确的，我们对世界的观察也总是片面而不确定的，世界对我们而言是未知的，我们无法完全确定事物的运作机理。例如图2.1 中的电阻R， $R=50\Omega$ 并不是一个严谨的说法，我们不知道物体细分到最小粒子（如果存在的話）后事物是否变得精确，但就目前人类所拥有的知识而言，阻值为 50Ω 的电阻是不存在的。我们平常所说的 50Ω 电阻只是一个统计概念，即电阻在 50Ω 左右的统计结果。延伸到整个系统，图2.1 的网络也变得不确定，R、L、C均是不精确的值，此时系统变为一个灰盒模型。所谓灰盒模型，即系统的一部分内容是未知的，另一部分内容是已知的，例如这里，尽管R、L、C的精确值是未知的，但我们依然可以知道其数值的大致范围。

倘若我们再推广一步，R、L、C的大致范围我们也不知道，如何设计该系统的控制环节便是系统辨识所要研究的内容。系统辨识领域中的一个重要的话题是如何解析出一个含有未知参数的系统结构，如果我们可以获取系统的输入输出样本，利用这些样本，配合上含有未知参数的系统结构方程，采取恰当的拟合方法，如最小二乘以及极大似然等，最后可以获取未知参数的近似解，使得这个灰盒模型的灰色褪去（但不会褪为白盒模型），进一步便可设计系统的控制环节。

2.3 黑盒模型与统计机器学习

白盒模型与灰盒模型的讨论均是基于一个前提：我们知道系统整体模型框架，只是某些参数有可能是未知的。但这个前提在实际生活中往往是不成立的，很多

时候，我们非但不知道系统的参数，甚至连系统的模型也知之甚少，此时的系统相当于一个黑盒，我们无法了解其内部结构。例如，判断一封100字的邮件是否为垃圾邮件，假设这个行为可以用一个函数来描述，即这个决策背后存在一个真理（或者说函数），通过它，我们输入100个字符，函数的输出告诉我们这封邮件是否为垃圾邮件，那么我们便可以通过这个函数来描述这个系统。可以肯定的值，这个函数在邮件完全可分（即不存在一些可能是或可能不是垃圾邮件的情况）的前提下是存在的，因为字符编码是有限的，只需穷举即可得到这个函数的决策面。但这个方法并不现实，假使所有的汉字只有2000个，那么100字的中文邮件将有 2000^{100} 种可能，要穷举是不可能的。另一种做法是寻求语法结构，先对这100字进行分词，再进行语义分析，这个过程也可以描述为一个函数映射过程：假设函数 $f(x)$ 代表分词， $g(x)$ 代表语义分析， y 代表系统输出，那么系统的模型可以表述为 $y = g[f(x)]$ 。这种想法早在二十年前就被抛弃，因为自然语言含有强烈的上下文气息。例如在“冬天能穿多少穿多少，夏天能穿多少穿多少”这个例子中，同一句“能穿多少穿多少”在不同的上下文中含有不同的意义。尽管我个人认为这种上下文背后也必然存在一个因果关系，但这种因果关系是在是太难寻找了，所以这种方法也不是一个可行的方案。

统计机器学习的做法是，假定一个模型（这个模型与系统本质的模型关联并不大），利用大量样本训练假设的模型，最后将训练完毕的模型作为本质模型的逼近。与之前讨论的系统辨识相比，两者在训练阶段是类似的，均是利用样本进行参数整定，不同点在于，系统辨识是训练带有强烈先验的模型（比如由具体物理原理推导出的传递函数）的参数，而统计学习训练的是假设模型（比如高斯模型、隐式马尔可夫模型、神经网络、支持向量机等）的参数。

回到垃圾邮件分类的例子中，统计机器学习的一种解决方案是利用朴素贝叶斯分类，这种方法中，所假设的模型是朴素贝叶斯模型。我们首先建立一个含有 d 个元素的垃圾邮件特征字典，比如{购买、大促销、店庆…}，此时，任何一封邮件都可以用一个 d 维列向量表示。例如，某封邮件在字典中的元素只包含“购买”，而其他的元素均不包含，那么这封邮件便可以表示为

$$x = [1, 0, 0, \dots, 0]^T \quad (2-3)$$

如果我们有大量垃圾邮件样本，我们不难统计出字典中各个元素在垃圾邮件中的出现概率 $p(x_i|y = 1)$ ，其中， y 代表样本的标签，若该样本是垃圾邮件，则 $y = 1$ ，反之 $y = 0$ ， x_i 代表字典中的第 i 个元素，例如，我们的字典中， $x_1 = “购买”$ 。建立样本的描述方式后，我们采用朴素贝叶斯公式，计算样本为垃圾邮件

时的概率分布

$$\begin{aligned} p(x|y=1) &= p(x_1, x_2, \dots, x_d|y=1) \\ &= p(x_1|y=1)p(x_2|y=1) \cdots p(x_d|y=1) \end{aligned} \quad (2-4)$$

显然，式(2-4)在数学上只有在 x_i 均独立时才成立，而本质上， x_i 并不是独立的（因为存在上下文），这里本应使用全概率公式，而我们之所以假设 x_i 独立的原因是这样做可以降低模型的复杂度。所以说，在统计学习中，假设模型与本质模型的关联往往不大。

模型训练完毕后（即计算 $p(x_i|y=1)$ ），对于一封新收到的邮件，我们只需使用贝叶斯公式即可计算该邮件是垃圾邮件的概率

$$\begin{aligned} p(y=1|x) &= \frac{p(x|y=1)p(y=1)}{p(x)} \\ &= \frac{\prod_{i=1}^d p(x_i|y=1)}{\prod_{i=1}^d p(x_i|y=1) + \prod_{i=1}^d p(x_i|y=0)} \end{aligned} \quad (2-5)$$

统计学习更像是一种假设检验，本例子中的朴素贝叶斯，完全抛弃了语义分析，这并不符合自然语言的原理。但实际上，尽管这个模型并不是本质模型，但逼近效果足以让人接受，能工作得很好，因此被很多邮件厂商所采用。

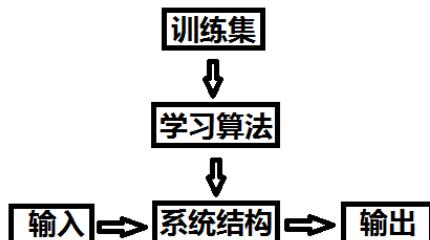


图 2-2 机器学习系统框图

如果一定要对机器学习下一个定义，我认为Tom.Mitchell的描述比较恰当：对于某类任务T和性能度量P，如果一个计算机程序在T上以P衡量的性能随着经验E而自我完善，那么我们称这个计算机程序从经验E中学习^[25]。如果将这段话转化成系统框图，则如图2-2 所示。

2.4 本章小结

精确性只存在于数学中，现实世界充满了不确定性，难以寻找事物背后的机理。统计学习绕过这套机理，根据设计者的意愿，假设一个模型，利用这个模型逼近事物的机理，相当于把世界看成一个黑盒，并不打算去探索黑盒的内部结构，而是仿造一个黑盒来模拟其工作原理。

第3章 受限玻尔兹曼机

统计学习并不在乎事物的本质是什么，我们更关心的是数据的分布是怎样的，为了描述数据的分布，我们往往引入各种各样的模型刻画这种分布，比如，贝叶斯决策论中我们引入多维高斯分布，支持向量机中我们引入最大间隔分离面等。对于同一个任务，采用不同的模型得到的结果是不一样的，机器学习与模式识别的一个不同点在于，机器学习更注重统计，模式识别更注重于模型。例如，对于同一个任务，机器学习学者可能会假设出多个模型，再根据模型选择理论选取一个最优的模型，而模式识别学者更倾向于先大致分析这个任务更适合使用哪个模型，选取后仔细优化这个模型。尽管两者有一定的差异性，但很多方面两者是共通的，比如神经网络。神经网络并不依赖于具体的任务，也就是说，语音识别可以用神经网络处理，图像识别也可以用神经网络处理，之所以能这样做的一个原因是，神经网络的内部对于我们而言是透明的，如果我们只看重结果，希望有一个模型可以在给定一个输出的时候输出一个决策结果，不需要关心这个结果是怎么得到的，那么神经网络是一个很好的选择。神经网络的种类众多，一个分支是玻尔兹曼机，这种网络灵感源自统计物理中的玻尔兹曼分布与伊辛模型，随后，在这种网络的基础上又发展出一套受限玻尔兹曼机，通过这些网络，都可以建立数据的概率分布描述。

3.1 伊辛模型

伊辛模型（Ising model）是统计物理中描述物质相变的一种模型^[16]，是一个相互磁耦合的自旋阵列。比如在铁这种物质中，当温度降到某个程度，微观原子的自旋会表现出一定的倾向性，从而在宏观上产生磁矩，而当温度升高到一定程度时，其自旋就变得随机。

假设某个伊辛模型中有 N 个自旋的原子，对于每个原子，其状态只能取+1或-1，我们用向量 s 来表示所有原子的状态，亦即 s 代表这个伊辛模型的状态，那么我们定义该伊辛模型处于状态 s 下的能量函数为^[16]

$$E(s; W, H) = - \left[\frac{1}{2} \sum_{i,j=1}^N w_{ij} s_i s_j + \sum_j H s_j \right] \quad (3-1)$$

式中， w_{ij} 代表原子 i 和原子 j 之间的耦合，如果 i 和 j 是相邻的，那么 $w_{ij} = C$ ，否则 $w_{ij} = 0$ ，如果常数 $C > 0$ ，那么这个模型是铁磁性的，否则是反铁磁性的。常

数 H 代表作用场， $E(s; w, H)$ 代表能量函数，对于一般化的能量函数，我们也称之为Lyapunov函数。如果将这个能量函数推广到 H 和 W 不是常数的情况，即^[16]

$$E(s; W, H) = - \left[\frac{1}{2} \sum_{i,j=1}^N w_{ij} s_i s_j + \sum_j h_j s_j \right] \quad (3-2)$$

此时我们将得到一个物理学家称之为“自旋玻璃”的模型，这也是神经网络中的“Hopfield网络”。

3.2 玻尔兹曼机

在统计物理中，对于一个具有一定自由度的物理系统，其系统的状态是具有随机性的而不是固定的（比如房间中的氧气分子分布），假设系统处于某个状态*i*的概率为 p_i ，那么当系统与外界达到热平衡时，其概率分布为

$$p_i = \frac{1}{Z_T} e^{-E_i/T} \quad (3-3)$$

式中， T 代表系统所处的温度， E_i 代表系统处在*i*状态下的能量， Z_T 是在 T 温度下为了使得概率满足柯尔莫果洛夫第二公理的归一化常数。这个分布也称之为正则分布或吉布斯分布。

在机器学习中，我们往往自定义一个能量函数，然后通过正则分布建立模型，通过这种基于能量的模型来对数据进行分析。所以正则分布可以看做是机器学习与统计物理间的桥梁。

如果将式(3-2)中的作用场去掉，并改写成矩阵形式，则得到玻尔兹曼机中的能量函数

$$E(s; W) = - \frac{1}{2} s^T W s \quad (3-4)$$

事实上，在机器学习中我们并不关心常数 T ，则玻尔兹曼分布为

$$p_i = \frac{1}{Z} e^{-E_i} = \frac{1}{Z} \exp \left[\frac{1}{2} s^T W s \right] \quad (3-5)$$

对于一个数据对象，假设我们能观察到 n 维特征，但不仅仅代表这个数据只含 n 维特征，因此我们往往引入隐含特征的概念，假设对于一个对象，我们能观察到 v 个特征，那么我们用 v 个节点来表示这些特征，又假设我们自定义隐含的特征为 h 个，那么我们用 h 个节点来代表这个特征。比如图3-1 代表了含有4个可见节点和2个隐含节点的玻尔兹曼网络。

由图3-1 可见，在玻尔兹曼机中，可见节点与可见节点之间、隐含节点与隐含节点之间是可以有连接的，因此这是一个反馈网络。层内节点有连接可以大大地

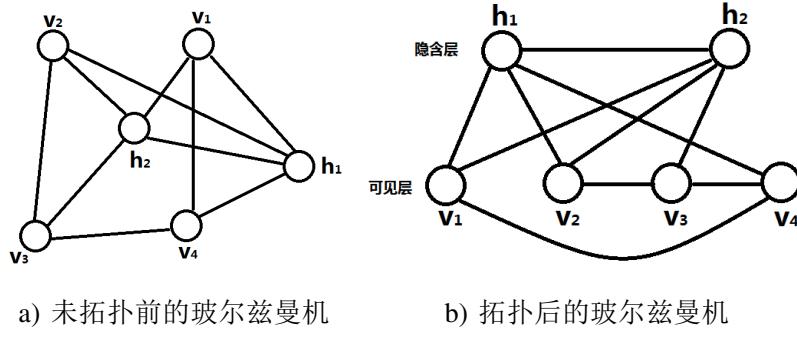


图 3-1 玻尔兹曼机网络构型

增强网络的表达能力，但是也大大地增加了网络的训练难度，因此玻尔兹曼机并没有非常成功地解决人工智能的任务。

3.3 受限玻尔兹曼机

在受限玻尔兹曼机（Restricted Boltzmann Machine, RBM）中，我们取消层间连接，从而出发后经过若干次“移动”也无法回到原点，因此RBM也可以认为是一个有向无环图。其网络结构图3-2所示。

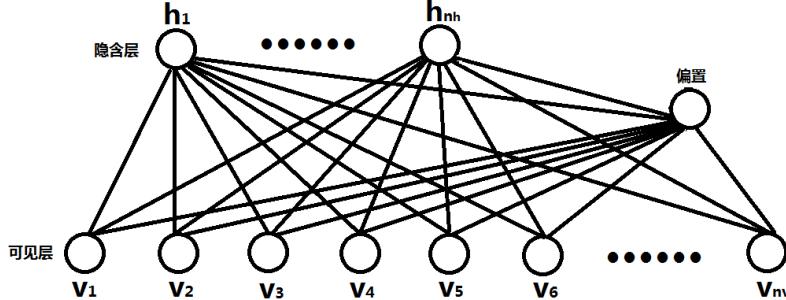


图 3-2 RBM网络构型

为了方便往后的讨论，我们约定网络参数的数学符号如下

$$W_{n_h \times n_v} = \begin{bmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,n_v} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,n_v} \\ \vdots & \vdots & \ddots & \vdots \\ w_{n_h,1} & w_{n_h,2} & \cdots & w_{n_h,n_v} \end{bmatrix} \quad v = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_{n_v} \end{bmatrix} \quad h = \begin{bmatrix} h_1 \\ h_2 \\ \vdots \\ h_{n_h} \end{bmatrix} \quad b_v = \begin{bmatrix} b_{v_1} \\ b_{v_2} \\ \vdots \\ b_{v_{n_v}} \end{bmatrix} \quad b_h = \begin{bmatrix} b_{h_1} \\ b_{h_2} \\ \vdots \\ b_{h_{n_h}} \end{bmatrix} \quad (3-6)$$

式中， $W_{n_h \times n_v}$ 代表网络的权值参数， w_{ij} 代表第*i*个隐含节点到第*j*个可见节点间的连接权值，因此 $W_{n_h \times n_v}$ 的第*i*行代表了通向 h_i 的所有连接的权值，第*j*列代表了通

往 v_j 的所有连接的权值。 v 代表可见节点的状态， h 代表隐含节点的状态， b_v 代表隐含层到可见层的偏置， b_h 代表可见层到隐含层的偏置。

在RBM中，我们定义其Lyapunov函数如下^[26]：

$$E(v, h) = - \sum_{i=1}^{n_v} b_{v_i} v_i - \sum_{j=1}^{n_h} b_{h_j} h_j - \sum_{i=1}^{n_v} \sum_{j=1}^{n_h} h_j w_{j,i} v_i \quad (3-7)$$

若将(3-7)写成矩阵形式，则为：

$$E(v, h) = -b_v^T v - b_h^T h - h^T W v \quad (3-8)$$

由正则分布的定义(3-3)，我们可以得到 v, h 的联合分布为

$$p(v, h) = \frac{1}{Z} e^{-E(v, h)} \quad (3-9)$$

其中配分函数 Z 为

$$Z = \sum_{v, h} e^{-E(v, h)} \quad (3-10)$$

由于实际中我们往往只能观察到可见节点，因此对联合分布(3-9)边缘化得

$$p(v) = \sum_h \frac{1}{Z} e^{-E(v, h)} = \frac{1}{Z} \sum_h e^{-E(v, h)} \quad (3-11)$$

这个 $p(v)$ 总可以写成如下形式：

$$p(v) = \frac{e^{-F(v)}}{Z} \quad (3-12)$$

式中，我们称 $F(v)$ 为自由能函数^[4]，由式(3-11)与(3-12)，我们不难推出 $F(v)$ 为

$$F(v) = -\ln \sum_h e^{-E(v, h)} \quad (3-13)$$

为了继续我们往下的讨论，我们不加证明地引入关于受限玻尔兹曼机的一个定理：

定理 3.1 在RBM中，在给定可见元状态时，隐含元的激活条件独立；反之，在给定隐含元状态时，可见元的激活条件独立。

此外，我们还需引入一些记号

$$h_{-k} = (h_1, h_2, \dots, h_{k-1}, h_{k+1}, \dots, h_{n_h})^T \quad (3-14)$$

$$\alpha(k) = b_{h_k} + \sum_{i=1}^{n_v} w_{k,i} v_i \quad (3-15)$$

$$\beta(v, h_{-k}) = \sum_{i=1}^{n_v} b_{v_i} v_i + \sum_{\substack{j=1 \\ j \neq k}}^{n_h} b_{h_j} h_j + \sum_{i=1}^{n_v} \sum_{\substack{j=1 \\ j \neq k}}^{n_h} h_j w_{j,i} v_i \quad (3-16)$$

即 h_{-k} 代表除 k 外的所有隐含元状态， $\beta(v, h_{-k})$ 代表除 h_k 外所有节点构成的能量函数。因此，总的能量函数(3-7)可以写为

$$E(v, h) = -\beta(v, h_{-k}) - h_k \alpha(k) \quad (3-17)$$

下面我们来推导隐含层的激活函数，在给定可见节点状态时，对于节点 h_k ，其激活概率为

$$P(h_k = 1|v) \quad (3-18)$$

由定理3.1，式(3-18)等价于

$$\begin{aligned} P(h_k = 1|h_{-k}, v) &= \frac{P(h_k = 1, h_{-k}, v)}{P(h_{-k}, v)} \\ &= \frac{P(h_k = 1, h_{-k}, v)}{P(h_k = 0, h_{-k}, v) + P(h_k = 1, h_{-k}, v)} \\ &= \frac{1}{1 + \exp(-E(h_k = 0, h_{-k}, v) + E(h_k = 1, h_{-k}, v))} \\ &= \frac{1}{1 + \exp[(\beta(v, h_{-k}) + \alpha(k) \cdot 0) + (-\beta(v, h_{-k}) - \alpha(k) \cdot 1)]} \\ &= \frac{1}{1 + e^{-\alpha(k)}} \\ &= \text{sigmoid}(\alpha(k)) \end{aligned} \quad (3-19)$$

因此，给定可见层状态时，隐含元 k 的激活概率为

$$P(h_k = 1|v) = \text{sigmoid}(b_{h_k} + \sum_{j=1}^{n_v} w_{k,j} v_j) \quad (3-20)$$

同理可推导在给定隐含层状态时，可见元 k 的激活概率为

$$P(v_k = 1|h) = \text{sigmoid}(b_{v_k} + \sum_{i=1}^{n_h} w_{i,k} h_i) \quad (3-21)$$

由定理3.1的独立性可知

$$P(h|v) = \prod_{j=1}^{n_h} P(h_j|v) \quad (3-22)$$

$$P(v|h) = \prod_{i=1}^{n_v} P(v_i|h) \quad (3-23)$$

在RBM中，我们的目标是训练参数 W ， b_v ， b_h 使得模型能成功地刻画出数据分布。为了方便起见，我们记 $\theta = (W, b_v, b_h)$ ，注意，这并不是矩阵合并，而是将所有的参数用 θ 来代替。

假定我们有一个训练集 S

$$S = \{v^{(1)}, v^{(2)}, \dots, v^{(i)}, \dots, v^{(n_s)}\} \quad (3-24)$$

其中 n_s 为训练样本数， $v^{(i)}$ 为第 i 个样本且

$$v^{(i)} = [v_1^{(i)}, v_2^{(i)}, \dots, v_{n_v}^{(i)}]^T \quad (3-25)$$

由于样本是独立的，因此似然函数为

$$\mathcal{L}(\theta) = \prod_{i=1}^{n_s} P(v^{(i)}) \quad (3-26)$$

对应的对数似然为

$$\ell(\theta) = \ln \prod_{i=1}^{n_s} P(v^{(i)}) = \sum_{i=1}^{n_s} \ln P(v^{(i)}) \quad (3-27)$$

对于整个训练集，我们要优化参数，使得似然最大化，假设我们使用梯度上升方法，则针对某个样本 \hat{v} ，参数的更新规则为

$$\theta = \theta + \eta \frac{\partial \ln \mathcal{L}_{\hat{v}}}{\partial \theta} \quad (3-28)$$

其中

$$\begin{aligned} \ln \mathcal{L}_{\hat{v}} &= \ln P(\hat{v}) \\ &= \ln \left[\frac{1}{Z} \sum_h e^{-E(\hat{v}, h)} \right] \\ &= \ln \sum_h e^{-E(\hat{v}, h)} - \ln Z \end{aligned} \quad (3-29)$$

从而

$$\frac{\partial \ln \mathcal{L}_{\hat{v}}}{\partial \theta} = \frac{\partial}{\partial \theta} \left[\ln \sum_h e^{-E(\hat{v}, h)} \right] - \frac{\partial}{\partial \theta} \ln Z \quad (3-30)$$

由式(3-10)、式(3-11)、式(3-12)，得

$$\begin{aligned} \frac{\partial \ln \mathcal{L}_{\hat{v}}}{\partial \theta} &= -\frac{\partial}{\partial \theta} F(\hat{v}) + \frac{1}{Z} \sum_v e^{-F(v)} \frac{\partial}{\partial \theta} F(v) \\ &= -\frac{\partial}{\partial \theta} F(\hat{v}) + \sum_v p(v) \frac{\partial}{\partial \theta} F(v) \end{aligned} \quad (3-31)$$

通过自由能函数 $F(v)$ 对参数 θ 求偏导，我们有

$$\begin{aligned} \frac{\partial}{\partial \theta} F(v) &= \frac{\sum_h e^{-E(v, h)} \cdot \frac{\partial E(v, h)}{\partial \theta}}{\sum_h e^{-E(v, h)}} \\ &= \sum_h \frac{e^{-E(v, h)}/Z}{\sum_h e^{-E(v, h)}/Z} \cdot \frac{\partial E(v, h)}{\partial \theta} \\ &= \sum_h \frac{p(v, h)}{p(v)} \cdot \frac{\partial E(v, h)}{\partial \theta} \end{aligned} \quad (3-32)$$

从而，式(3-31)等价于

$$\begin{aligned}
\frac{\partial \ln \mathcal{L}_{\hat{v}}}{\partial \theta} &= - \sum_h \frac{p(\hat{v}, h)}{p(\hat{v})} \cdot \frac{\partial E(\hat{v}, h)}{\partial \theta} + \sum_v p(v) \sum_h \frac{p(v, h)}{p(v)} \cdot \frac{\partial E(v, h)}{\partial \theta} \\
&= - \sum_h p(h|\hat{v}) \cdot \frac{\partial E(\hat{v}, h)}{\partial \theta} + \sum_{v,h} p(v, h) \frac{\partial E(v, h)}{\partial \theta} \\
&= - \mathbb{E}_{p(h|\hat{v})} \left[\frac{\partial E(\hat{v}, h)}{\partial \theta} \right] + \mathbb{E}_{p(v, h)} \left[\frac{\partial E(v, h)}{\partial \theta} \right]
\end{aligned} \tag{3-33}$$

式中， $\mathbb{E}_{p(h|\hat{v})}$ 为 $p(h|\hat{v})$ 分布下的期望， $\mathbb{E}_{p(v, h)}$ 为 $p(v, h)$ 分布下的期望。

通过式(3-33)，我们就可以计算相对于某个样本 \hat{v} 的对数似然梯度。式(3-33)中的第一项是容易计算的，因为我们已经推导出了 $p(h|v)$ 的分布形式，即式(3-20)和式(3-22)。由于我们简记 $\theta = (W, b_v, b_h)$ ，因此我们需要对 W ， b_v ， b_h 三个参数分别推导梯度公式。

$$\begin{aligned}
\sum_h p(h|v) \frac{\partial E(v, h)}{\partial w_{ij}} &= - \sum_h \prod_{k=1}^{n_h} p(h_k|v) h_i v_j \\
&= - \sum_{h_i} \sum_{h_{-i}} p(h_i|v) p(h_{-i}|v) h_i v_j \\
&= - \sum_{h_i} p(h_i|v) h_i v_j \sum_{h_{-i}} p(h_{-i}|v) \\
&= - \sum_{h_i} p(h_i|v) h_i v_j \\
&= - \sum_{h_i} p(h_i|v) v_j
\end{aligned} \tag{3-34}$$

$$\begin{aligned}
\sum_h p(h|v) \frac{\partial E(v, h)}{\partial b_{h_i}} &= - \sum_h \prod_{k=1}^{n_h} p(h_k|v) h_i \\
&= - \sum_{h_i} \sum_{h_{-i}} p(h_i|v) p(h_{-i}|v) h_i \\
&= - \sum_{h_i} p(h_i|v) h_i \sum_{h_{-i}} p(h_{-i}|v) \\
&= - \sum_{h_i} p(h_i|v) h_i \\
&= - \sum_{h_i} p(h_i|v)
\end{aligned} \tag{3-35}$$

$$\begin{aligned}
\sum_h p(h|v) \frac{\partial E(v, h)}{\partial b_{v_i}} &= - \sum_h p(h|v) v_i \\
&= - v_i
\end{aligned} \tag{3-36}$$

尽管我们可以很容易地通过式(3-34)、式(3-35)、式(3-36)计算式(3-33)中的第一项，但是第二项却无法计算，因为第二项涉及到归一化因子Z，这将是 $O(2^{n_v+n_h})$ 复杂度的项，因此我们需要使用马尔可夫链蒙特卡罗方法（MCMC）进行处理。

3.4 本章小结

本章中，我们从伊辛模型出发，引入玻尔兹曼机，进一步推广到受限玻尔兹曼机。在受限玻尔兹曼机中，我们建立了对数据描述的概率分布即正则分布，随后，我们推导出受限玻尔兹曼机中由可见节点激活隐含节点以及由隐含节点激活可见节点所使用的公式，此时，如果整个网络的参数是确定的，那么，利用这个网络可以刻画数据的特征。但由于未经训练的网络参数是未知的，我们需要使用样本来求取参数的近似值，求取的方法我们使用极大似然，为了使用梯度下降方法最大化似然函数，我们对似然函数求导，进而得到每一步迭代所使用的梯度。但由于这个梯度有一项不容易求取，我们需要利用马尔可夫链蒙特卡罗方法来处理，但本章没有详细讨论，因此，本章节并不是一个独立的章节，考虑到马尔可夫链蒙特卡罗方法所蕴含的内容较多，所以我们将其独立成为一章。

第4章 马尔可夫链蒙特卡罗方法

蒙特卡罗方法被评为“20世纪十大算法”之一，自20世纪50年代该方法被提出后的几十年里已被学术界与工业界广泛应用。这套方法最初源自于Stan Ulam对纸牌游戏的思考，他试图计算52张卡牌的组合可能，在尝试穷举失败后，他意识到需要一种随机方法去求取近似解而不是花费大量时间求取一个精确解。随后，他找到冯·诺依曼，两人共同完善了蒙特卡罗方法的一些理论，比如重要性采样和舍弃采样。由于蒙特卡罗方法是一整套解决方案，算法众多，其理论贡献不能仅仅归功于这两人，还应包括提出MH算法的Metropolis和Hastings、将蒙特卡罗方法应用到中子扩散问题的Fermi等^[27]。20世纪80年代，蒙特卡罗方法被引入机器视觉与人工智能领域，同时，在此之上加入马尔可夫链形成一套新的理论体系。这套理论常用于贝叶斯推断的归一化、边缘化与期望问题，概率论的配分函数问题，最优化问题以及机器学习中的模型选择问题。

4.1 蒙塔卡罗方法核心思想

蒙特卡罗方法的一个应用是计算一个分布下某个函数的期望，现在假设我们要计算一个积分

$$I_f = \int_{-\infty}^{+\infty} (2x^2 + x + 1) \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{1}{2}x^2\right) dx \quad (4-1)$$

为了简化，我们记

$$f(x) = 2x^2 + x + 1 \quad (4-2)$$

$$p(x) = \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{1}{2}x^2\right) \quad (4-3)$$

那么，式(4-1)可以简写为

$$I_f = \int_{-\infty}^{+\infty} f(x) \cdot p(x) dx \quad (4-4)$$

由于 $p(x)$ 恰好是一个标准高斯分布，倘若我们能从 $p(x)$ 中采样得到 N 个独立同分布(i.i.d) 样本 $\{x_i\}_{i=1}^N$ ，则积分 I_f 可以近似为^[27]

$$I_f \approx \hat{I}_f = \frac{1}{N} \sum_{i=1}^N f(x_i) \quad (4-5)$$

此时，若 $N \rightarrow \infty$ ，根据大数定律，可知

$$I_f = \lim_{N \rightarrow \infty} \hat{I}_f = \lim_{N \rightarrow \infty} \frac{1}{N} \sum_{i=1}^N f(x_i) \quad (4-6)$$

倘若从数学期望的角度上来解释上面的问题，那么积分(4-1)可以理解为：我们现在有一个函数 $f(x) = 2x^2 + x + 1$ ，其自变量 x 符合标准高斯分布，即 $x \sim N(0, 1)$ ，那么为了计算 $f(x)$ 在分布 $N(0, 1)$ 下的数学期望，我们可以在分布 $N(0, 1)$ 中采样出 N 个独立的样本 $\{x_i\}_{i=1}^N$ ，将这些样本代入 $f(x)$ ，得到 N 个函数值 $\{f(x_i)\}_{i=1}^N$ ，对这些函数值求和取平均后得到的结果即为期望 I_f 的近似值 \hat{I}_f 。显然，其近似程度与 N 有关， N 越大，采样的样本越多，近似的程度也便越高。

但积分(4-1)未免过于特殊，首先， $p(x)$ 是一个标准高斯分布，这使得我们可以利用一些很成熟的方法来从 $N(0, 1)$ 中采样出 N 个样本，但如果 $p(x)$ 不是一个高斯分布，也不是一个均匀分布、伽马分布等一些我们常见的概率分布，它只是一个普通得不能再普通的分布，此时又该如何解决？其次，积分(4-1)只是一维形式，而实际生活中的概率分布往往是高维的，那么高维情况又该如何推广？接下来的几个小节我们将致力于解决以上几个问题。

4.2 舍弃采样

我们先来解决之前提到的第一个问题，如果说 $p(x)$ 不是一个常见的概率分布应该如何解决？例如

$$p(x) = 0.3 \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{(x-2)^2}{2}\right) + 0.7 \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{(x+2)^2}{2}\right) \quad (4-7)$$

当 $f(x)$ 依然设定为(4-2)时，概率密度曲线 $p(x)$ 以及函数 $f(x) \cdot p(x)$ 的图像如图4-1 所示

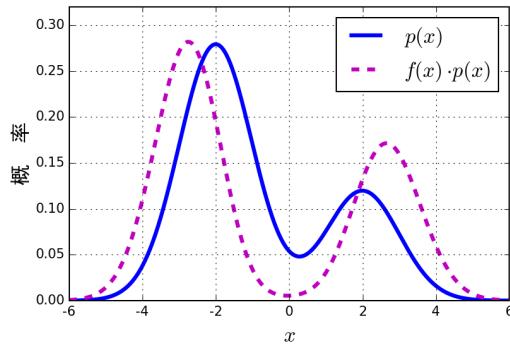


图 4-1 实际分布 $p(x)$ 与 $f(x) \cdot p(x)$ 的函数图像

此时，由于 $p(x)$ 并不是一个常见的概率分布，而我们所拥有的一些简单的采样方案基本都是针对于某一类特定的分布而提出的，基于这个原因，这里要想在 $p(x)$ 中采样出 N 个样本并不是一件简单的工作。为此，我们引入舍弃采样来解决在任意分布上采样的问题。

舍弃采样基于这样一个思想：既然我们无法从一个随意的分布 $p(x)$ 上采样，但是可以在一个特殊的分布 $q(x)$ 上采样，比如从高斯分布中采样，那么我们为何不用 $q(x)$ 来逼近 $p(x)$ 呢？为此，我们引入一个分布 $q(x)$ ，称之为提议分布，这个分布需要满足以下条件^[27]

$$p(x) < Mq(x), \quad M < \infty \quad (4-8)$$

式中， M 是一个定常数，上述约束条件相当于，提议分布 $q(x)$ 与实际分布 $p(x)$ 的比值 $p(x)/q(x)$ 需要在变量 x 的空间 X 中存在下界 M 。从图像的角度看，提议分布扩大 M 倍后，应该“覆盖”，或者说“包着”实际分布 $p(x)$ ，例如，对于式(4-7)的分布 $p(x)$ ，当 $M = 2.3$ 且提议分布 $q(x)$ 为

$$q(x) = \frac{1}{3\sqrt{2\pi}} \exp\left(-\frac{(x+1.3)^2}{2 \times 3^2}\right) \quad (4-9)$$

即 $q(x) \sim \mathcal{N}(-1.3, 3)$ 时，实际分布 $p(x)$ 与提议分布 $q(x)$ 的图像如图4.2 所示

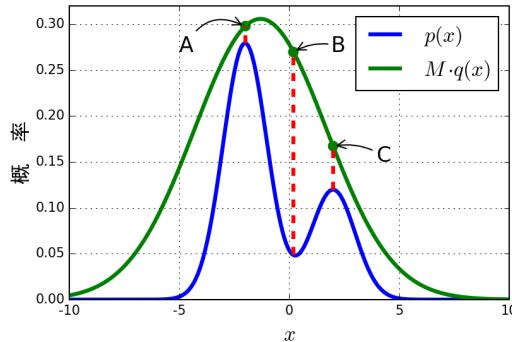


图 4-2 实际分布 $p(x)$ 与提议分布 $q(x)$ 的函数图像

由于提议分布是一个常规的高斯分布，我们有一系列的成熟方法可以在其之上采样出多个样本。假设我们采样得到一个样本后，我们有两种选择：要么接受这个样本，并将这个样本看做是从 $p(x)$ 上采样得到的，要么舍弃这个样本，认为这个样本与 $p(x)$ 采样的样本差距太大，不能看做是 $p(x)$ 的采样样本。

然而，我们什么时候应该接受 $q(x)$ 的样本作为 $p(x)$ 的样本，什么时候又应该拒绝呢？为了刻画这个事件，我们引入了接受概率的概念。假定我们现在已从 $q(x)$ 中采样得到一个样本 $x^{(i)}$ ，则其接受概率 A 我们定义为^[16]

$$A = \frac{p(x^{(i)})}{M \cdot q(x^{(i)})} \quad (4-10)$$

计算得到接受概率后，我们以 A 作为接受样本的概率，但在计算机中，没有一种方法直接地描述“以概率 A 接受样本”这个行为，为了仿真这个行为，我们可以在区间为 $[0, 1]$ 的均匀分布 $U(0, 1)$ 上随机生成一个数 u ，若 $u < A$ 则接受样本，否则拒绝。通过这样的方式，我们便可以模拟“以 A 为概率接受样本”。我们很容易将一个样本推广到 N 个样本的情况，其具体描述如算法4-1所示。

```

Input: 真实分布 $p(x)$ ; 提议分布 $q(x)$ ; 采样量 $N$ 
Output:  $N$ 个采样样本 $\{x^{(i)}\}_{i=1}^N$ 

1  $i = 1;$ 
2 repeat
3   采样 $x^{(i)} \sim q(x)$ , 获取随机数 $u \sim U(0, 1)$ ,  $A = \frac{p(x^{(i)})}{M \cdot q(x^{(i)})}$ ;
4   if  $u < A$  then
5     | 接受样本 $x^{(i)}$ 作为 $p(x)$ 的样本,  $i += 1$ ;
6   end
7   else
8     | 舍弃样本 $x^{(i)}$ ,  $i$ 保持不变;
9   end
10 until  $i = N$ ;
```

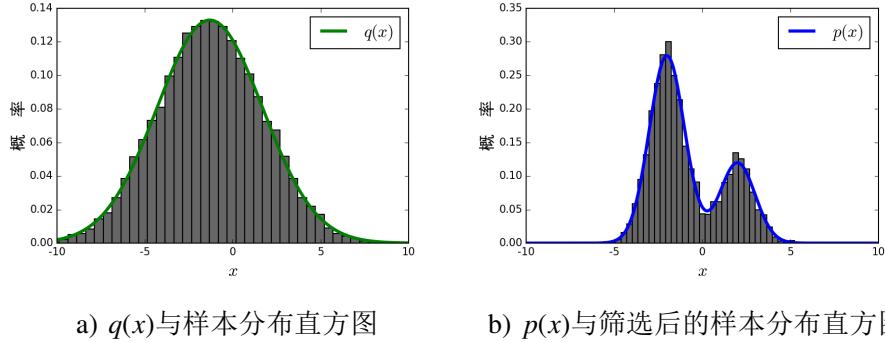
算法 4-1 舍弃采样算法

对于接受概率其定义，一种较为直观的理解是：接受概率 $A(x^{(i)})$ 刻画了 $p(x^{(i)})$ 与 $Mq(x^{(i)})$ 的相似程度。如图4.2中的A、B、C三点，假设我们可以分别从两个分布中采样得到多个样本，对于 $q(x)$ 而言，采样样本出现在A点附近的概率最大，其次是B点附近，再次是C点附近。然而，对于 $p(x)$ 而言，采样样本出现在B点附近的概率要比出现在C点附近的概率要小，因此，把从 $q(x)$ 中采样得到的样本直接作为 $p(x)$ 的采样样本是不合适的。但由于

$$\frac{p(x_B \text{附近})}{M \cdot q(x_B \text{附近})} < \frac{p(x_C \text{附近})}{M \cdot q(x_C \text{附近})} \quad (4-11)$$

B点较小的接受概率使得我们舍弃了 $q(x)$ 采样样本中B点附近大量的样本，C点较大的接受概率使得我们保留了 $q(x)$ 采样样本中C点附近的大量样本，经过舍弃阶段后，我们便可以从 $q(x)$ 的采样样本中筛选出可以刻画 $p(x)$ 性质的样本，因此，舍弃操作可以看做是对样本的纠正。

在 $q(x)$ 定义为式(4-9), $p(x)$ 定义为式(4-7)且 $M = 2.3$ 的情况下, 图4-3 a)为 $q(x)$ 及采样样本的概率分布直方图, 将这些样本经过舍弃后, 其分布直方图如图4-3 b)所示。不难看出, 尽管样本是从提议分布 $q(x)$ 中采样得到的, 但舍弃采样方法可以很好地逼近原始分布, 因此我们可以用筛选后的样本间接地作为 $p(x)$ 的采样样本而不再需要从 $p(x)$ 中直接采样。



a) $q(x)$ 与样本分布直方图 b) $p(x)$ 与筛选后的样本分布直方图

图 4-3 舍弃采样的逼近效果

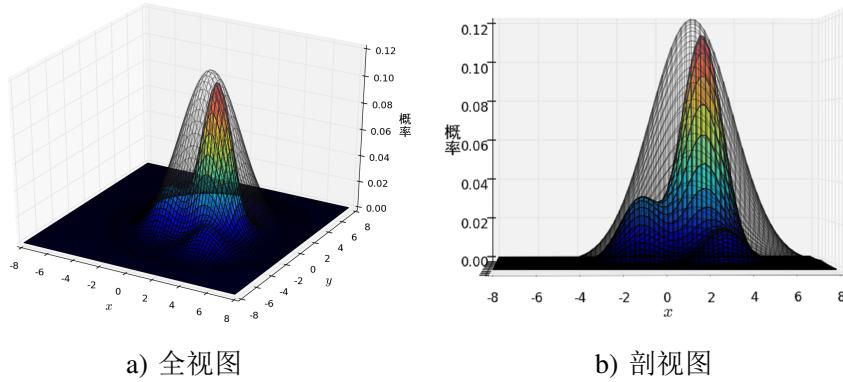
事实上, 式(4-7)依然过于特殊, 其概率密度不过是两个一维高斯分布的线性组合, 如果我们将其扩展到二维情形, 例如, 当真实分布 $p(x)$ 与提议分布 $q(x)$ 分别定义为

$$\begin{aligned} p(x) = & 0.2 \cdot \frac{1}{2\pi} \exp\left(-\frac{(x+1)^2 + (y+1)^2}{2}\right) + \\ & 0.1 \cdot \frac{1}{2\pi} \exp\left(-\frac{(x-3)^2 + (y+3)^2}{2}\right) + \\ & 0.7 \cdot \frac{1}{2\pi} \exp\left(-\frac{(x-2)^2 + y^2}{2}\right) \end{aligned} \quad (4-12)$$

$$q(x) = \frac{1}{2 \times 2^2 \cdot \pi} \exp\left(-\frac{(x-1.5)^2 + y^2}{2 \times 2^2}\right) \quad (4-13)$$

在 $M = 3$ 时, 真实分布与提议分布的图像如图4-4 所示

以上例子都过于简单, 实际中我们遇到的一般都是高维的情形, 概率密度也更为复杂。这将会导致一个结果, 提议分布 $q(x)$ 为了“包裹”真实分布 $p(x)$, M 需要取一个很大的值使得 $q(x)$ 可以覆盖掉最凸出的维度。想象这样一种极端的情形, 在一个高维度的 $p(x)$ 中, 有一个维度是类似于脉冲的尖峰, 此时 M 需要取很大才足以覆盖它, 但对于剩余的维度而言, M 可能只需要一个较小的值便可以覆盖掉它们, 为了使约束 $p(x) < M \cdot q(x)$ 恒成立, M 需要取最大的值, 过大的 M 使得接受概率 $A = p(x)/Mq(x)$ 过小, 大规模地舍弃样本将使采样速度变慢甚至无法再可以接受的时间内完成采样。究其原因, 其最根本的弊端在于 $q(x)$ 必须满足约



a) 全视图

b) 剖视图

图 4-4 二维真实分布（彩色）与提议分布（灰色）

束 $p(x) < M \cdot q(x)$ 。为了解决这个问题，我们将引入重要性采样，这种方法可以使我们不受约束地选取任意的 $q(x)$ 。

4.3 重要性采样

回顾章节4.1中的讨论，我们为什么要使用舍弃采样？因为蒙特卡罗方法中积分式(4-4)在 $p(x)$ 不是常规的分布形式时难以采样，导致我们无法利用式(4-5)来计算积分。而舍弃采样通过构建一个容易采样的分布 $q(x)$ ，对其采样样本筛选后作为 $p(x)$ 的样本，式(4-5)从而得以进行下去。

式(4-5)之所以能成立，是因为我们利用了点质量函数来逼近概率密度，即^[27]

$$p_N(x) = \frac{1}{N} \sum_{i=1}^N \delta_{x^{(i)}}(x) \quad (4-14)$$

式中， $\delta_{x^{(i)}}(x)$ 为 $x^{(i)}$ 处的脉冲函数。下面我们将简单证明式(4-14)的正确性¹

为了得到概率密度 $p(x)$ ，我们可以在 x 附近以 x 为中心建立一个小区域 A ，其对应的体积为 V_A ，此时我们从 $p(x)$ 中随机采样 N 个样本 x_1, x_2, x_N ，那么这些样本中落入区域 A 中的概率为

$$P(x \in A) \approx \frac{1}{N} \sum_{i=1}^N 1(x_i) \quad (4-15)$$

式中， $1(x_i)$ 为示性函数，其定义为

$$1(x) \begin{cases} 0, & x \in A \\ 1, & x \notin A \end{cases} \quad (4-16)$$

¹感谢刘家锋老师的指点

实际上，式(4-15)的作用类似于窗函数的作用，即根据离散样本估计点密度，但同时，根据连续性我们也可以得到点 x 的概率密度，即

$$P(x \in A) = \int_A p(x)dx \quad (4-17)$$

当区域 A 为无穷小区域时，即 $V_A \rightarrow 0$ ，区域 A 内的每一点概率密度相等，此时 $p(x)$ 是常数，因此有

$$P(x \in A) = \int_A p(x)dx = p(x) \int_A dx = p(x)V_A \quad (4-18)$$

联合式(4-15)与式(4-18)，有

$$p(x)V_A \approx \frac{1}{N} \sum_{i=1}^N 1(x_i) \quad (4-19)$$

则

$$\begin{aligned} p(x) &\approx \frac{1}{V_A} \frac{1}{N} \sum_{i=1}^N 1(x_i) \approx \frac{1}{N} \sum_{i=1}^N \frac{1(x_i)}{V_A} \\ &\approx \frac{1}{N} \sum_{i=1}^N \delta(x_i) \end{aligned} \quad (4-20)$$

因此式(4-14)成立，证明完毕。

倘若我们不采用这种逼近方式，而采用另外一种，在介绍这种方式之前，我们先引入所谓的重要性权值 $w(x)$ ，即

$$w(x) = \frac{p(x)}{q(x)} \quad (4-21)$$

式中， $q(x)$ 为任意一个分布。此时，新的逼近方式可以陈述为^[27]

$$\hat{p}_N(x) = \sum_{i=1}^N w(x^{(i)}) \delta_{x^{(i)}}(x) \quad (4-22)$$

对比式(4-14)，我们可以理解为 $p_N(x)$ 中的 $1/N$ 相当于重要性权值恒为 $1/N$ 。引入重要性权值后，积分式(4-4)可以改写为

$$I_f = \int_{-\infty}^{+\infty} f(x)w(x)q(x)dx \quad (4-23)$$

对应的，式(4-5)改写为

$$\hat{I}_f \approx \sum_{i=1}^N f(x^{(i)})w(x^{(i)}) \quad (4-24)$$

如果从另一个角度思考，以上讨论相当于，我们现在有一个目标函数 $f(x)$ 及概率分布 $p(x)$ ，要计算 $f(x)$ 在分布 $p(x)$ 下的期望，由于 $p(x)$ 不是一个常见的分布形式，导致难以利用式(4-5)计算积分，那么我们构建一个常见的分布形式 $q(x)$ ，将目标函数改写为

$$\hat{f}(x) = \frac{f(x)p(x)}{q(x)} = f(x)w(x) \quad (4-25)$$

则积分式(4-4)变为

$$\hat{I}_f = \int_{-\infty}^{+\infty} \hat{f}(x)q(x)dx \quad (4-26)$$

此时我们又可以使用类似于式(4-5)的方式来计算积分了。

对比与之前讨论的舍弃采样而言，重要性采样的优点在于 $q(x)$ 不受约束，也不存在舍弃行为，这使得算法效率相对与舍弃采样有所提高。但重要性采样也有其内在缺点，即 $q(x)$ 选取的好坏程度影响着积分的精度，若 $q(x)$ 选取得不好，则需要大量的样本来提高积分精度。刻画 $q(x)$ 好坏的一种准则是最小化 $\hat{I}_N f(x)$ 的方差，即

$$var_{q(x)}[f(x)w(x)] = \mathbb{E}[f^2(x)w^2(x)] - I^2(f) \quad (4-27)$$

由于 $I^2(f)$ 于 $q(x)$ 无关可以摄取，再利用Jensen不等式，有

$$\mathbb{E}[f^2(x)w^2(x)] \geq \mathbb{E}^2[|f(x)|w(x)] = \left(\int |f(x)|p(x)dx \right)^2 \quad (4-28)$$

因此提议分布可以选取为

$$q(x) = \frac{|f(x)|p(x)}{\int |f(x)|p(x)dx} \quad (4-29)$$

由于分母只是一个归一化常数可以不管，我们现在要做的是在 $|f(x)|p(x)$ 中采样，这看起来似乎没有什么用，实际上也确实没有什么用，毕竟 $|f(x)|p(x)$ 并不似一个容易采样的函数。但它启示我们一点，如果 $p(x)$ 的样本落在的重要域可以使得 $|f(x)|p(x)$ 取值较大，那么采样效率将会非常高，这也是重要性采样的命名来源。

无论是舍弃采样还是重要性采样都是独立采样，即样本之间是独立的，从而采样效率较低。为了提高采样效率，我们可以使用关联采样，即样本之间存在关联，构建样本之间的关联性所要用到的便是著名的马尔可夫链。

4.4 马尔可夫链

正式引入马尔可夫链之前，我们打算先从一个随机游走的例子入手。想象一个正方形区域内，在某一点放入N个粒子，这些例子随机地朝着各个方向游走。对于每一个粒子而言，每一次游走的步长是相等的，只是角度随机选择。当粒子移动到正方形的边界时，它将被反弹回正方形区域内。直觉上的想象，经过漫长的时间，粒子将漫布在整个正方形区域内。如图4-5 所示是两个随机游走的例子，

其中蓝色的初始状态在中心附近，绿色的初始状态在左下角。

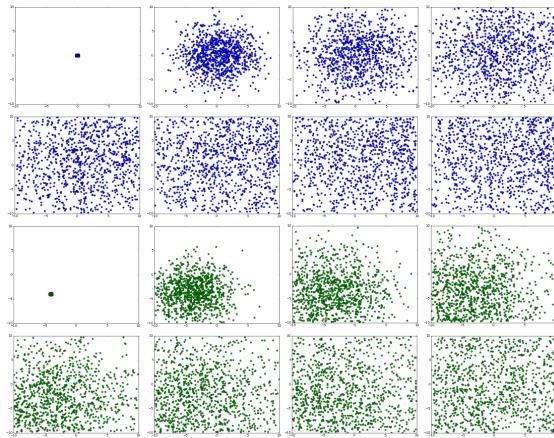


图 4-5 两个随机游走的例子

这个例子类似于布朗运动，有趣的现象是，起始位置的选取并不会影响最终结果—粒子终将呈均匀分布。正如一锅未加盐的汤，无论盐从哪个位置撒下，最终整锅汤的咸淡是均匀的。对于某个特定的粒子而言，追踪它的轨迹是无意义的，从宏观上看，粒子下一步处于哪个位置与之前的位置无关，只与它当前的位置有关，因为它是经过怎样的路径到达当前的状态对下一步移动到哪里起不到任何作用。

马尔可夫链也是基于同一个原理，假设我们的变量 x 可取得状态有 s 个，那么集合 $S = \{x_1, x_2, x_s\}$ 被称为状态空间，马尔可夫链刻画的是变量 x 在状态空间 S 中各个状态之间迁移的轨迹。类似于随机游走的例子，马尔可夫链的下一个状态与之前的状态无关，只与当前的状态有关，即

$$p(x^{(i+1)}|x^{(i)} \dots x^{(1)}) = p(x^{(i+1)}|x^{(i)}) \quad (4-30)$$

这个性质也被称为马尔可夫性质。对于给定的系统，我们定义某个状态转移到另一个状态的概率为转移概率为

$$p(x^{(i+1)}|x^{(i)}) = T(x^{(i)} \rightarrow x^{(i+1)}) \quad (4-31)$$

如图4-6 中的系统，状态空间为 $\{A, B, C\}$ ，由图中的参数我们容易算得转移矩阵 T

$$T = \begin{bmatrix} T(A \rightarrow A) & T(A \rightarrow B) & T(A \rightarrow C) \\ T(B \rightarrow A) & T(B \rightarrow B) & T(B \rightarrow C) \\ T(C \rightarrow A) & T(C \rightarrow B) & T(C \rightarrow C) \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 \\ 0.6 & 0 & 0.4 \\ 0 & 0.2 & 0.8 \end{bmatrix} \quad (4-32)$$

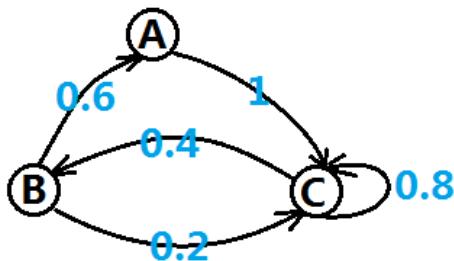


图 4-6 马尔科夫链

倘若我们为A、B、C赋予一些实际意义，我们令A代表下雨，B代表多云，C代表晴天，又假设今天的天气为下雨，现在我们要估计10天后的天气状况。首先，我们可以很容易地将今天的天气描述为一个向量

$$x = [1 \ 0 \ 0] \quad (4-33)$$

为了计算10天后的天气，用 x 乘以10次转移矩阵后将得到10天后的天气状况，即

$$[1 \ 0 \ 0] \times \begin{bmatrix} 0 & 0 & 1 \\ 0.6 & 0 & 0.4 \\ 0 & 0.2 & 0.8 \end{bmatrix}^{10} = [0.091 \ 0.151 \ 0.758] \quad (4-34)$$

也就是说，10天后下雨的概率为0.091，多云的概率为0.151，晴天的概率为0.758。

倘若我们假设今天的天气为多云，利用同样的方法，我们可以计算得10天后的天气状况为

$$[0 \ 1 \ 0] \times \begin{bmatrix} 0 & 0 & 1 \\ 0.6 & 0 & 0.4 \\ 0 & 0.2 & 0.8 \end{bmatrix}^{10} = [0.091 \ 0.151 \ 0.758] \quad (4-35)$$

我们发现，不管今天是雨天开始多云，计算得到10天后的天气情况是十分接近的（这里完全一样是因为我们舍去了尾部小数），如果读者有兴趣的话可以验证初始状态为晴天是得到的结果也是一样的。也就是说，不管今天的天气如何，对10天后的天气均无影响²。

之所以出现这种现象是因为马尔可夫链在第十个周期时已经进入一个我们称之为平稳的状态。类似于之前的随机游走，当经过足够长的时间后，系统的状态与初始状态再无关联，只与系统的结构（也就是转移矩阵）有关。但并不是所有的转移矩阵都能达到平稳，在这里，我们并不打算深入讨论，只给出结论定理

²需要解释一下，这个天气系统是我们假设的，实际中的天气系统并不会如此简单

定理 4.1 如果一个马尔科夫链是各态遍历的，那么存在一个时间 t_s ，当 $t > t_s$ 时，马尔可夫链到达一个平稳分布 x^* ，其中 x^* 满足

$$x^* = x^* \mathcal{T} \quad (4-36)$$

所谓各态遍历，即要求马尔可夫链是不可约且非周期的，所谓不可约，即所有状态都是有关联的，从某个状态出发，不存在无法到达的状态，所谓非周期，即马尔可夫链不会陷入某几个状态间循环，满足上述两个条件的马尔可夫链我们称它是各态遍历的。

利用马尔可夫链的这个性质，我们就可以实现关联采样。由于初始状态与稳态无关，我们可以随机设置，经过多次随机游走，进入稳态后得到的状态便可以作为一个样本。以上讨论均基于一个前提，即转移矩阵是已知的。然而，实际中，我们并不知道转移矩阵的数值。例如，我们并不知道由晴天转移到多云的概率是 0.2，这个数值是我们捏造的。但是一旦转移矩阵知道了，采样问题便迎刃而解，目前成熟的马尔可夫链蒙特卡罗方法整体框架都是类似的，不同的地方往往在于转移矩阵的构造上。

4.5 Metropolis-Hastings 算法

与舍弃采样类似，Metropolis-Hastings 算法（以下简称 MH 算法）也存在一个提议分布，这个提议分布定义为 $q(x^*|x)$ ，不同的是，舍弃采样中的提议分布受一个强约束，即 $p(x) < M q(x)$ ，而 MH 算法中的提议分布只需要满足 $q(x^*|x) > 0$ 即可，显然这个约束相当于没有约束，因为概率论三大公理的第一条就使这个约束成立了。另一个不同点在于，MH 算法中的提议分布其意义为：在当前状态 x 下，由提议分布 $q(x^*|x)$ 产生一个试探性状态 x^* ，随后根据接受概率决定是否转移到新状态 x^* 上，这里，接受概率定义为

$$A = \frac{p(x^*) \cdot q(x|x^*)}{p(x) \cdot q(x^*|x)} \quad (4-37)$$

此时，若接受概率 $A > 1$ ，则接受这个新状态，否则，以概率 A 接受新状态。若接受了这个新状态 x^* ，则状态从 x 转移到 x^* 处，并在 x^* 处的提议分布 $q(x^{**}|x^*)$ 产生一个新的试探状态 x^{**} ，如此反复循环，若不接受这个状态 x^* ，则状态仍然停留 x 处。

需要注意的一点是，在 MH 采样中，提议分布 $q(x^*|x)$ 是对着状态 x 的变动而变动的，例如，如果我们将提议分布 $q(x^*|x)$ 设计成一个在以 x 为中心，以 2 为方差的高斯分布，即 $q(x^*|x) \sim \mathcal{N}(x, 2)$ ，那么如图 4-7 所示，在 $x^{(1)}$ 状态和 $x^{(2)}$ 状态下的提议分布是不同的。

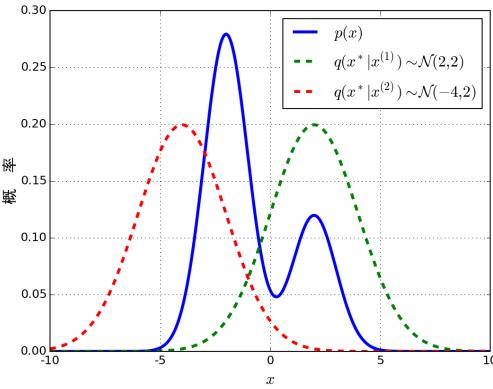


图 4-7 不同状态下的提议分布

综合以上的讨论，MH算法可以描述为算法4-2 中的过程。

Input: 真实分布 $p(x)$; 提议分布 $q(x^*|x)$; 游走次数 N

Output: 1个采样样本 $x^{(N)}$

```

1 设置初始状态 $x^{(0)}$ ,  $i = 0$ ;
2 repeat
3   采样 $x^* \sim q(x^*|x^{(i)})$ , 获取随机数 $u \sim U(0, 1)$ ;
4   计算接受概率 $A = \frac{p(x^*) \cdot q(x|x^*)}{p(x)q(x^*|x)}$ 
5   if  $u < A$  then
6      $x^{(i+1)} = x^*$ ;
7   end
8   else
9      $x^{(i+1)} = x^{(i)}$ 
10  end
11 until  $i = N$ ;
12 返回 $x^{(N)}$ 

```

算法 4-2 Metropolis-Hastings 算法

相比于舍弃采样，舍弃采样的拒绝是直接抛弃样本，而MH采样的拒绝是让样本停留在当前状态。另一个不同点在于，算法4-2描述的是采样出一个样本的过程，是一个随机游走的过程，而算法4-1描述的是采样出 N 个样本的过程。尽管算法4-2只能采样出一个样本，但是我们也可以很容易将其扩展成为采样 N 个样本的算法。另外，由于各个样本的随机游走是独立的，不存在线程安全问题，因此算

法4-2可以很容易设计成为并行算法。

正如我们提到的，如果我们使用一个形式为 $q(x^*|x) \sim N(x, 2)$ 的提议分布区采样1000个式(4-7)中实际分布 $p(x)$ 的样本，对于不同的游走次数 N ，其结果如图4-8所示

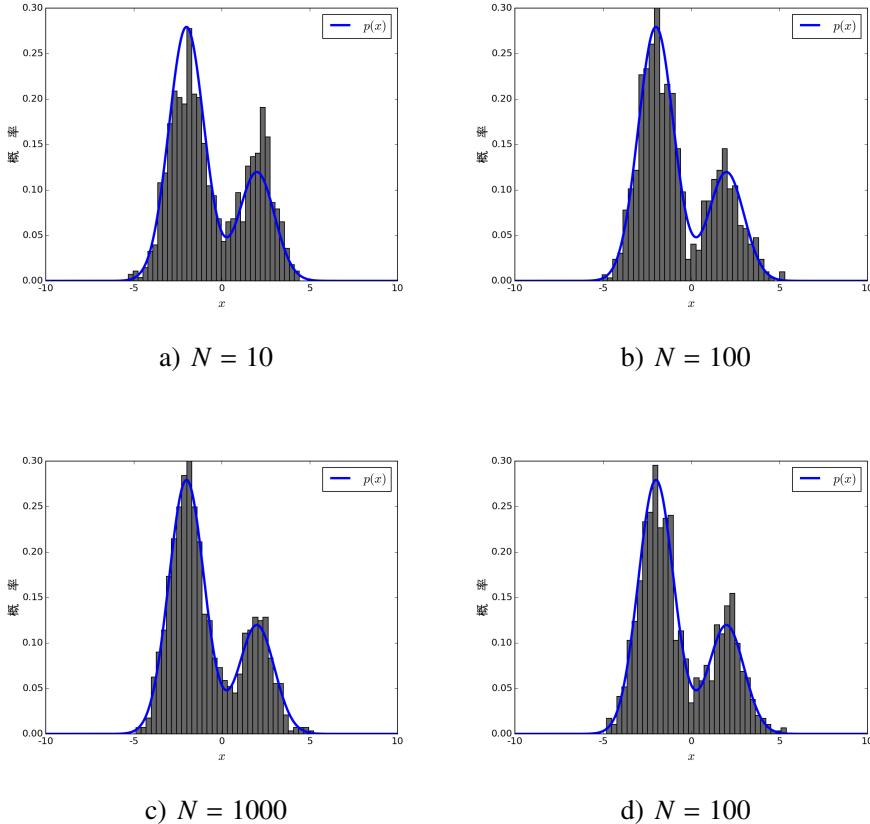


图 4-8 不同游走次数下的样本分布直方图

从图中我们不难发现，在游走次数 $N = 10$ 时，逼近效果并不完美，随着 N 的增大，当 $N = 100$ 时，采样结果已经能很好地刻画实际分布了。此时， N 再增大（比如1000或5000）已经差别不大。从随机游走的角度上看，可以理解为， $N = 100$ 时就已经进入平稳分布，再继续游走下去并没有太大意义。

MH采样可以认为是很多MCMC方法的模板，其他的MCMC方法大多都是MH算法的特例。之所以会从MH算法中衍生出如此多的分支是因为MH有其自身的局限性。首先，我们难以估计状态是否进入平稳状态，即游走次数 N 难以确定。其次，MH算法在高维问题中的效果并不好，因为高维空间的地形复杂，MH算法一不小心就会落入一个周围状态的接受概率都很小的区域，这将导致试探状态不断地被否决，从而长时间滞留在该点附近。接受概率的存在，使得游走

效率低下，而Gibbs采样是MH采样的一个特例，这种方法不存在接受概率，或者说接受概率为1，因此每一次游走都将得到一个新的状态，这个特性将大大地加快收敛速度。

4.6 Gibbs采样

Gibbs采样又称为热浴方法或“Glauber动力学”^[16]，常用于解决高维分布中的采样问题。假设我们有一个 d 维向量 x ，并且知道任一分量的条件概率分布形式

$$p(x_j|x_{-j}) \triangleq p(x_j|x_1, \dots, x_{j-1}, x_{j+1}, \dots, x_d) \quad (4-38)$$

则我们构建提议分布^[27]

$$q(x^*|x) = \begin{cases} p(x_j^*|x_{-j}) & \text{若 } x_j^* = x_j \\ 0 & \text{其他} \end{cases} \quad (4-39)$$

由于Gibbs采样是MH采样的特例，而MH采样的接受概率公式(4-37)又可以描述为

$$A = \min \left\{ 1, \frac{p(x^*) \cdot q(x|x^*)}{p(x) \cdot q(x^*|x)} \right\} \quad (4-40)$$

将式(4-39)代入式(4-40)，并利用 $x_j^* = x_j$ 性质以及贝叶斯公式，有

$$\begin{aligned} A &= \min \left\{ 1, \frac{p(x^*) \cdot p(x_j|x_{-j}^*)}{p(x) \cdot p(x_j^*|x_{-j})} \right\} \\ &= \min \left\{ 1, \frac{p(x^*) \cdot p(x_j|x_{-j})}{p(x) \cdot p(x_j^*|x_{-j}^*)} \right\} \\ &= \min \left\{ 1, \frac{p(x_{-j}^*)}{p(x_{-j})} \right\} \\ &= 1 \end{aligned} \quad (4-41)$$

由式(4-41)我们不难得出结论，Gibbs的接受概率恒为1，亦即在Gibbs采样中，我们不存在丢弃样本或状态停滞的行为，每一次都会转移到一个新的状态上，这显然会加快算法的收敛速度。

以上内容未免过于晦涩，为了方便大家理解，我们将再次阐述Gibbs采样的原理。首先，Gibbs采样要基于一个大前提—真实分布 $p(x)$ 在各个维度的条件概率分布是已知的，即 $p(x_j|x_1, \dots, x_{j-1}, x_{j+1}, \dots, x_d)$ 已知。如果这个条件概率分布是未知的，则使用Gibbs采样不是一个恰当的策略，因为Gibbs采样的提议分布是基于这个条件概率分布之上构建出来的。式(4-39)定义的提议分布，其含义为：固定 $x_1, \dots, x_{j-1}, x_{j+1}, \dots, x_d$ 这 $d - 1$ 个维度的状态不变，单独处理 d 个维度中的一个，

即 x_j 。如果将Gibbs采样试想成为一只章鱼，那么每次它都只迈出一只脚，当这只脚固定后，再迈出剩下的另一只脚，因此，Gibbs采样的算法描述如算法4-3所示。

Input: 各个维度的条件分布 $p(x_j|x_{-j})$; 游走次数 N

Output: 1个采样样本 $x^{(N)}$

- 1 设置初始状态 $x^{(0)}$, $i = 0$;
- 2 **repeat**
- 3 采样出第1个维度 $x_1^{(i+1)} \sim p(x_1|x_2^{(i)}, x_3^{(i)}, \dots, x_d^{(i)})$
- 4 采样出第2个维度 $x_2^{(i+1)} \sim p(x_2|x_1^{(i+1)}, x_3^{(i)}, \dots, x_d^{(i)})$
- 5 ⋮
- 6 采样出第 j 个维度 $x_j^{(i+1)} \sim p(x_j|x_1^{(i+1)}, \dots, x_{j-1}^{(i+1)}, x_{j+1}^{(i)}, x_d^{(i)})$
- 7 ⋮
- 8 采样出第 d 个维度 $x_d^{(i+1)} \sim p(x_d|x_1^{(i+1)}, x_2^{(i+1)}, \dots, x_{d-1}^{(i+1)})$
- 9 **until** $i = N$;
- 10 返回 $x^{(N)}$

算法 4-3 Gibbs采样算法

至此，MCMC的大体内容已讨论完毕，让我们回到第三章遗留的问题。
第三章中，式(3-33)，即

$$\frac{\partial \ln \mathcal{L}_{\hat{v}}}{\partial \theta} = -\mathbb{E}_{p(h|\hat{v})}\left[\frac{\partial E(\hat{v}, h)}{\partial \theta}\right] + \mathbb{E}_{p(v, h)}\left[\frac{\partial E(v, h)}{\partial \theta}\right] \quad (4-42)$$

我们说，式中的第一项期望是容易处理的，而第二项却难以处理，因为如果我们采用穷举的方法解决这个问题，这将会是一个 $O(2^{n_v+n_h})$ 复杂度的运算，但采用MCMC的方法就很好处理。由于目前我们需要解决的问题是 $\frac{\partial E(v, h)}{\partial \theta}$ 在分布 $p(v, h)$ 下的期望，根据式(3-7)中对能量函数 $E(v, h)$ 的定义，偏导数非常容易求取。如果我们能在 $p(v, h)$ 中采样出多个样本，那么这个问题便迎刃而解。但采样 $p(v, h)$ 是一件困难的事，幸运的是，我们知道 $p(v, h)$ 的条件概率 $p(v|h)$ 以及 $p(h|v)$ ，即式(3-22)和式(3-23)，因此我们完全可以通过Gibbs采样，经过多次状态转移后采样出一个样本，再以同样的方法采样出多个样本，利用这多个样本，加以式(4-5)，便可以算出期望值，整个问题便解决了。

4.7 对比离差

尽管第三章遗留的问题可以通过Gibbs采样解决，但事实上，正如我们前面提到的，马尔可夫链进入平稳分布的时间难以确定。我们知道，马尔可夫链的初始状态对平稳分布在性质上是没有影响的，但是不同的初始状态会影响进入平稳分布的时间。就如同往汤里加盐，在不同的位置撒下并不会影响最后的咸淡，但会影响盐在水中的扩散速度，Gibbs采样也同理，不同的初始值对收敛速度有影响。如果初始值随机设置，则模型的训练速度十分缓慢。Hinton提出了一种名为对比离差^[28]（Contrastive Divergence, CD）的方法，其中心思想是：既然数据出现了，那么说明这个数据是接近于平稳分布的，因此我们令初始值为该数据样本，进行吉布斯采样，所以，对比离差算法如算法4-4所示。

Input: 条件分布 $p(h|v)$ 和 $p(v|h)$; 迭代次数 k

Output: 1个采样样本 $v^{(k)}$

- 1 设置初始状态 $x^{(0)} = data, i = 0;$
- 2 **repeat**
- 3 采样出 $h^{(i)} \sim P(h|v^{(i)})$
- 4 采样出 $v^{(i+1)} \sim P(v|h^{(i)})$
- 5 **until** $i = k;$
- 6 返回 $v^{(k)}$

算法 4-4 CD-k 算法

算法4-4也被称为CD-k算法，随着k的增大，采样得到的样本越接近于模型的平稳分布，但在实验中， $k = 1$ 的时候已经能获得很不错的结果，因此我们往往令 $k = 1$ 。而得到最终的采样样本 $v^{(k)}$ 后，对数似然的梯度近似为

$$\frac{\partial \ln P(v)}{\partial w_{i,j}} \approx P(h_i = 1|v^{(0)})v_j^{(0)} - P(h_i = 1|v^{(k)})v_j^{(k)} \quad (4-43)$$

$$\frac{\partial \ln P(v)}{\partial b_{vi}} \approx v_j^{(0)} - v_j^{(k)} \quad (4-44)$$

$$\frac{\partial \ln P(v)}{\partial b_i} \approx P(h_i = 1|v^{(0)}) - P(h_i = 1|v^{(k)}) \quad (4-45)$$

CD-k算法可以认为是利用

$$CD_k = - \sum_h P(h|v^{(0)}) \frac{\partial E(v^{(0)}, h)}{\partial \theta} + \sum_h P(h|v^{(k)}) \frac{\partial E(v^{(k)}, h)}{\partial \theta} \quad (4-46)$$

来近似^[29]

$$\frac{\partial P(v)}{\partial \theta} = - \sum_h P(h|v^{(0)}) \frac{\partial E(v^{(0)}, h)}{\partial \theta} + \sum_{v,h} P(v, h) \frac{\partial E(v, h)}{\partial \theta} \quad (4-47)$$

利用CD-k，可以很高效地求得参数的增量 ΔW 、 Δb_v 以及 Δb_h ，从而训练受限玻尔兹曼机来刻画数据的分布。

4.8 本章小结

本章从蒙特卡罗的核心思想说起，引入舍弃采样，由于舍弃采样的提议分布受到一个约束，为此我们又引入重要性采样。无论是重要性采样或是舍弃采样，都属于非关联采样，因此我们引入马尔可夫链，得到一个关联采样方法，即MH采样。MH采样在高维空间中效果并不好，且存在接受概率使算法效率较低，为此我们引入用于解决高维问题且不存在接受概率的Gibbs采样，通过Gibbs采样可以第三章的期望问题，但Gibbs的收敛速度依赖于初始值的设定，为了选取一个较好的初始值，也为了加快算法的收敛速度，我们引入了对比离差方法。通过对比离差，可以高效的解决第三章中的期望问题。事实上，马尔可夫链蒙特卡罗方法其潜力并不仅局限于此，这套方法在机器学习中还有更为广泛的应用，然而我们并没有涉及。至此，受限玻尔兹曼机的内容讨论完毕，通过一个训练好的RBM，就可以刻画数据的分布。但目前的讨论并没有进入到核心内容，我们仍没有讨论识别，或者说分类问题。在第五章中，我们将介绍如何用多个RBM作为砖块垒出一个深度置信网络，并用这个网络进行图像识别。

第5章 深度置信网络

深度置信网络是由Hinton于2006年提出的一种深度全连接神经网络，全连接的深度神经网络的训练是及其困难的，然而在深度置信网络中，我们采用网络的逐层训练，随后对网络参数的微调，这种策略使得我们可以容易地训练多层神经网络。由于深度置信网络本质是一种传统神经网络的推广，在本章中，我们将从传统神经网络说起，介绍其训练方法，逐步将其推广到深度置信网络。

5.1 神经网络组成及表达能力

一个神经网络往往由多层神经元组成，神经元作为网络的基本单元，在给定恰当的激活函数和权值的前提下，只要这个网络足够庞大，足以容纳较多的神经元，那么这个网络可以表达任何一个连续函数^[30]，当然这只是一个理论上成立的理想情况，实际工程中，我们没有办法利用神经网络来描述所有的函数，但是这个定理能使我们对神经网络的表达能力充满信心。

5.1.1 神经元

一个神经元如图5-1所示，它包含 d 个输入 $x = [x_1, x_2 \dots, x_d]^T$ 以及对应的 d 个权值 $w = [w_1, w_2 \dots, w_d]^T$ ，还有一个偏置 b 。此外，它还应包括一个执行非线性映射的激活函数 $f(\cdot)$

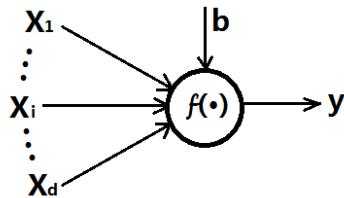


图 5-1 神经元

与 d 个输入 $x = [x_1, x_2 \dots, x_d]^T$ 直接相连的是数据的输入，例如，一张 28×28 像素的灰度图像作为网络输入，由于该图像可以展开成 $28 \times 28 = 784$ 的列向量，因此这个例子中的神经元就应该有784个输入，即 $x = [x_1, x_2 \dots, x_{784}]^T$ 。 d 个权值 $w = [w_1, w_2 \dots, w_d]^T$ ，刻画了 d 个输入 $x = [x_1, x_2 \dots, x_d]^T$ 的重要性， w 中的某个分量 w_i 越大，说明对应的分量 x_i 对最后决策结果的影响越大。从生物学的角度上看，

x 相当于给予生物各种刺激， w 相当于该生物对各种刺激的敏感度。例如，我们利用神经元设计一个医学诊断系统来判定一个人是否需要住院观察，对于该神经元，假设输入为 $x = [\text{心脏疼痛}, \text{头疼}, \text{腰疼}]$ ，如果有某个症状，则对应位置1，否则置0。显然，若一个人心脏疼痛，我们更倾向于让他住院观察，因此我们为心脏疼痛对应的权值设定一个较大的权值 $w_1 = 0.9$ ，头疼次之，我们设定为 $w_2 = 0.5$ ，当一个人腰疼时，不太可能需要住院治疗，我们为其设定一个较小的偏置 $w_3 = 0.1$ 。因此，这个神经元的权值为

$$w = [0.9, 0.5, 0.1]^T \quad (5-1)$$

假设现在有一位患者来到医院，他既有头疼又有心脏疼痛的症状，那么这位患者可以表示为向量

$$x = [1, 0, 1]^T \quad (5-2)$$

此时，病情积累的严重性为

$$w^T x = 0.1 + 0.9 = 1 \quad (5-3)$$

那么这位患者是否需要住院呢？为此，我们利用前面提及到的偏置 b 来刻画这件事。假设 $b = 0.7$ ，如果 $w^T x > b$ ，则说明这位患者患有严重的疾病，需要住院观察，网络输出 $y = 1$ ，否则这位患者不需要住院，网络输出 $y = 0$ 。如果我们定义为神经元的净激活为

$$net = w^T x + b \quad (5-4)$$

净激活可以理解为排除偏置干扰后神经元接收到的刺激总和。那么上述判别过程相当于用一个阶跃函数对净激活作一个非线性映射，即

$$y = f(net) = \begin{cases} 1 & \text{若 } w^T x - b > 0 \\ 0 & \text{其他} \end{cases} \quad (5-5)$$

其中，由于 b 是一个常数，因此 $w^T x + b$ 与 $w^T x - b$ 并没有区别。但实际上， $b = 0.7$ 时的神经元做出的决策，其结果只由心脏疼痛这个因素决定，因为如果一个患者没有心脏疼痛，那么最大的净激活是在他即患有头疼又患有腰疼的情况下取得，此时 $w^T x = 0.6$ 。而 $0.6 < 0.7$ ，并不会让他住院。因此，上面的神经元，转换成等价的逻辑命题便是：如果一个人的症状中含有心脏疼痛，则需要住院，否则不需要住院。如果我们把偏置调低，另 $b = 0.55$ ，则此时等价的逻辑命题为：如果一个人的症状中含有心脏疼痛，或者症状中既含有头疼又含有腰疼，则需要住院，否则不需要住院。综合以上讨论，偏置起到的是阈值的作用，但是这个阈值取多少就是设计者的意愿了，不同的阈值一般情况下会等价于不同的逻辑命题，

这取决于系统的期望输出是什么。

神经元总能转换成一个对应的逻辑表达，需要提出的一个因果关系是，我们并不是从逻辑表达中推出神经元的参数，而是从神经元的参数中可以得到一个逻辑表达。神经元的优点在于，通过一个神经元，便可以实现一个简单的决策，如果拥有更多的神经元，将他们组合起来形成一个神经网络，便可以实现更为复杂的决策行为。这个决策的因果关系是由参数决定的，后面我们将会看到，参数是通过统计学习学到的，因此在神经网络中，我们不需要对事物的因果关系进行分析，只需要使用大量的数据让网络学习其中的因果关系，这些因果关系对于我们而言是透明的。

5.1.2 逻辑表达

在本小节，我们将再一次讨论网络的逻辑表达能力，如图5-2所示的一个接收两维输入的神经元，其偏置 $w = [-2, -2]^T$ ，偏置 $b = 3$

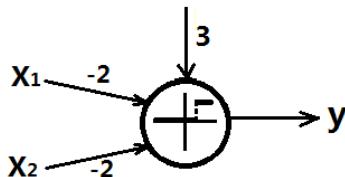


图 5-2 与非门的神经元表达

如果我们的输入至少有一个为假，即 $x = [0, 0]^T$ 或 $x = [1, 0]^T$ 或 $x = [0, 1]^T$ ，则网络的输出为真，即 $y = 1$ ，若两个输入都为真，即 $x = [1, 1]^T$ ，则网络的输出为假，即 $y = 0$ 。事实上，图5-2等价于逻辑电路中的与非门，即

$$y = \overline{x_1 \cdot x_2} \quad (5-6)$$

由于与非门是通用门，利用多个与非门可以构建出三大逻辑门：与门、或门、非门。因此，利用图5-2中的神经元可以表达任意一个逻辑函数。例如，在半加器中，其逻辑函数为

$$y_1 = x_1 \oplus x_2 \quad (5-7)$$

$$y_2 = x_1 \cdot x_2 \quad (5-8)$$

利用与非门搭建的电路图如图5-3 a)所示，而将其转换成对应的神经网络如图5-3

b)所示¹。

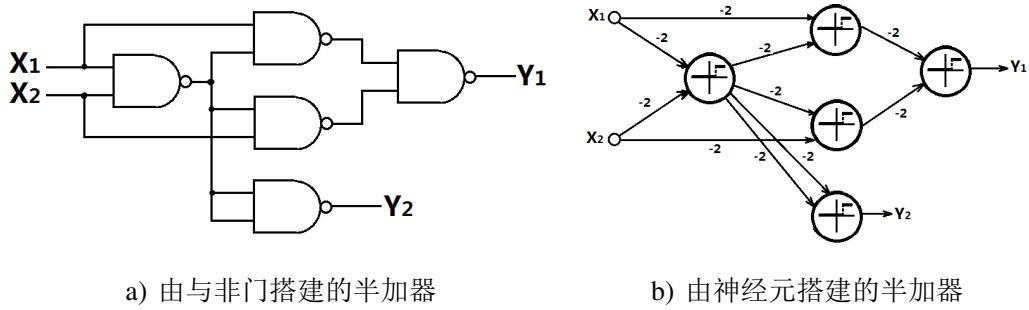


图 5-3 半加器的两种不同描述

我们之所以花费篇幅去介绍神经元的表达能力，是为了说明神经网络确实能实现一些逻辑决策，但并不意味着网络的权值需要我们手工设定的。这里的权值之所以要手工设定，是因为我们精心筛选出来用以阐述神经网络的表达能力，但实际应用中，权值是通过误差传播自动学习的，学习到的权值，其对应的逻辑命题对于人类而言是难以理解的，但这并不重要，因为我们需要的是仅仅决策结果，而不是这个决策是如何产生的因果关系。

5.2 神经网络的前馈

一般而言，神经网络是一个多层结构组织，网络的每一层都由多个神经元组成，后一层的神经元由前一层的神经元激活。同一层的神经元之间没有连接，因此同一层节点的激活是独立的。神经网络接收到一个数据后，逐层地激活各层的神经元，前一层的输出作为后一层的输入，最后得到一个输出结果。这个过程称为神经网络的前馈或前向传播。

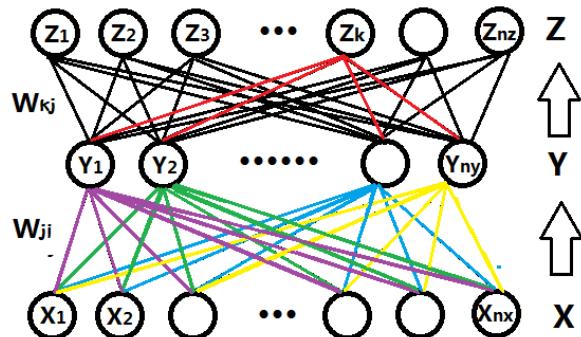


图 5-4 神经网络的前馈

¹图中，为了美观，我们并没有将偏置画出来

如图5-4所示是一个三层神经网络，由于神经激活是自底向上传播的，如果各层的激活函数都相同，为 $f(\cdot)$ ，则网络的第k个节点的输出为

$$z_k = f \left[\sum_{j=1}^{n_y} w_{kj} f \left(\sum_{i=1}^{n_x} w_{ji} \right) + b_j \right] + b_k \quad (5-9)$$

式中， w_{kj} 为y层的第j个节点到z层的第k个节点之间的连接权值， w_{ji} 为x层的第i个节点到y层的第j个节点之间的连接权值， b_j 为y层第j个节点的偏置， b_k 为z层第k的节点的偏置。

神经网络的这种自底向上的激活，有时也被解释为特征提取或重新编码。当接收到一个数据时，神经网络将这个数据逐层地进行特征抽取，过滤掉一些冗余信息，将剩余的重新编码，最后利用一个分类器将抽取到的最抽象（即最顶层）特征进行分类，从而完成整个识别过程。

5.2.1 神经激活

到目前为止，我们只介绍了一种激活函数，即阶跃函数，但实际上我们并不会使用阶跃函数作为激活函数。首先，阶跃函数是不连续的，其次，阶跃函数是不可导的²。激活函数不可导，将导致网络无法利用反向传播进行训练，关于这点我们将在反向传播章节讨论。作为激活函数，它应该满足以下几个条件

1. 它必须是非线性的。由式(5-9)可以看出，若激活函数是线性的，则多层神经网络实际上只相当于两层网络，因为多个矩阵相乘的结果仍为一个矩阵。网络一旦失去多层结构，则表达能力迅速下降。
2. 它应该具有饱和特性，即至少存在一个上界或下界。饱和特性的存在，使得神经元的输出不至于过高或过低，整个网络的编码维持在一定的范围内。但最近的研究表明，一些近似饱和的激活函数也能工作得很好，如此看来，饱和似乎并不是必要的。
3. 它应该连续且可导。由于反向传播中需要求取激活函数相对于净激活的偏导数，如果激活函数是不可导的，那么反向传播算法将无法执行。

实际工程中，我们常用的激活函数是sigmoid函数³或双曲正切函数。其中，sigmoid函数我们定义为

$$f(net) = \frac{1}{1 + e^{-net}} \quad (5-10)$$

其对应的图像如图5-5所示

²事实上阶跃函数是可导的，其导函数为脉冲函数，这是一个广义函数，我们不深入讨论

³也被称为logistic函数

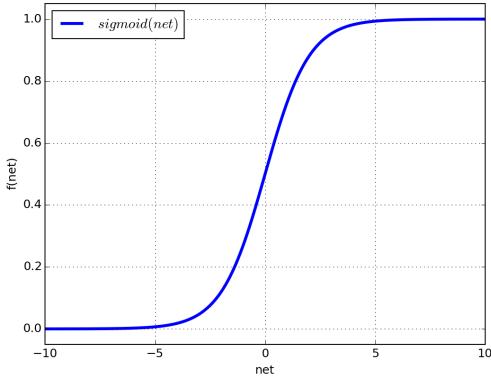


图 5-5 sigmoid 激活

可以看到，sigmoid 函数是一个值域为(0, 1)的连续可导函数，其导函数有一个重要特性，即

$$\begin{aligned} f'(net) &= \frac{e^{-net}}{(1 + e^{-net})^2} = \frac{1}{1 + e^{-net}} - \frac{1}{(1 + e^{-net})^2} \\ &= f(net)[1 - f(net)] \end{aligned} \quad (5-11)$$

这个特性之所以重要，是因为利用式(5-11)可以直接通过网络的输出直接计算激活函数相对于净激活的偏导数而不必引入额外的计算。我们看到，sigmoid 函数的输出范围为(0, 1)，有时候，我们希望网络的输出为(-1, 1)，此时可以使用双曲正切函数，即

$$f(net) = \tanh(net) = \frac{e^{net} - e^{-net}}{e^{net} + e^{-net}} \quad (5-12)$$

其对应的图像如图5-6所示

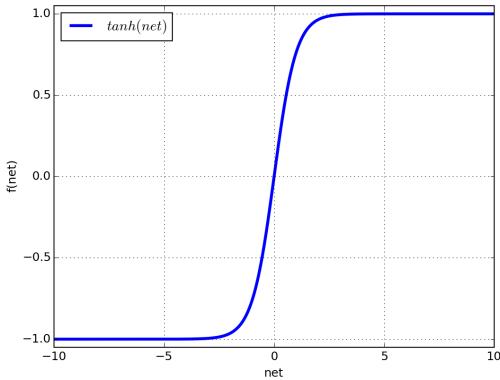


图 5-6 tanh 激活

近年来，在神经网络的研究上又提出了一些新的激活函数，其中最重要的莫

过ReLU激活函数，其定义为^[31]

$$f(net) = \begin{cases} net & \text{若 } net > 0 \\ 0 & \text{其他} \end{cases} \quad (5-13)$$

式(5-13)也可以简写为

$$f(net) = \max(0, net) \quad (5-14)$$

其对应的图像如图5-7所示

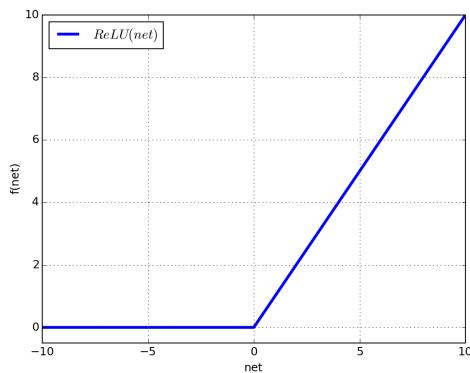


图 5-7 ReLU激活

这种激活函数相比于sigmoid函数有较大的改进，关于其优点我们将留到反向传播的章节讨论。但ReLU有一个缺点：在 $net < 0$ 时导数为0，从而导致反向传播无法更新参数。因此，在ReLU的基础上又产生了一些变体，例如，softplus中，激活函数定义为^[31]

$$f(net) = \log(1 + e^{net}) \quad (5-15)$$

其对应的图像如图5-8所示

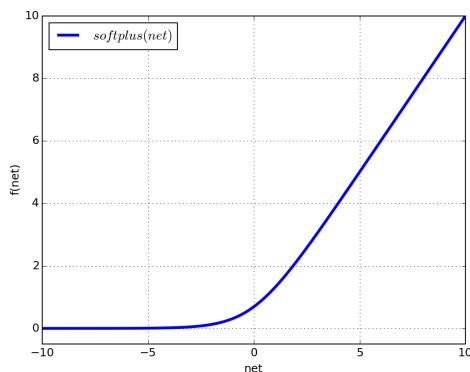


图 5-8 softplus激活

从图中不难看出，softplus是ReLU的圆滑版，关于softplus，一个有意思的结果是，其激活函数的导数为logistics函数，即

$$f'(net) = \frac{e^{net}}{1 + e^{net}} = \frac{1}{1 + e^{-net}} \quad (5-16)$$

然而这个结论并没有什么用处，因为它并不能减轻程序的计算量。

ReLU的另一种变体是由He Kaiming等人提出的PReLU，其激活函数定义为^[23]

$$f(net) = \begin{cases} net & \text{若 } net > 0 \\ \alpha net & \text{若 } net \leq 0 \end{cases} \quad (5-17)$$

式中， α 为一个较小的参数，例如当 $\alpha = 0.1$ 时，其图像如图5-9所示

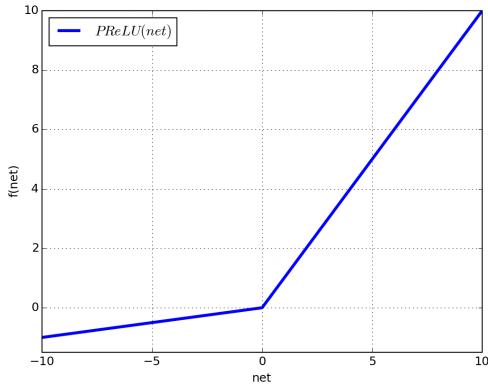


图 5-9 PReLU激活

显然，PReLU在负半平面不再会出现导数为0的情况。尽管参数 α 需要学习得到，但这种方法在ImageNet 2012数据集上获得了非常好的识别效果，首次在图像识别任务上超越了人的识别效果，错误率仅为4.94%。在该文中，作者提到了一个有意思的现象：在较低层的网络中，学到的参数 α 较大，而在高层的网络中，参数 α 较小。作者的猜测是较低层网络需要尽可能多地保留数据的信息，因此激活函数更倾向于类线性，而高层网络更倾向于抽象数据的结构，从而做出决策，因此高层的激活函数更倾向于非线性。

在同一个任务中，激活函数的不同选取会有不同的实验现象，目前并没有证明哪种激活函数更好。习惯上，在全连接神经网络中，我们更喜欢sigmoid派的激活函数，而在卷积网络中，我们更喜欢ReLU派的激活函数，尽管这些都不是强制的。

5.3 分类器

神经网络的逐层激活，其本质在于将一种编码转化成另一种编码。编码之间的转换，可以过滤掉一些原始编码中无用的噪声或信息，这个过程一般都伴随着熵的减小（但并不绝对）。然而，一个神经网络在最顶层的激活，其编码是人类所无法理解的，因此，在最顶层节点之上还需要一个分类器对神经网络提取到的特征或者说编码进行分类。分类器其作用，一方面在于将编码转换为人类所能理解的编码，这在有监督的分类任务以及无监督的聚类任务中是必要的，因为我们最后希望网络能给我们一个输出标签。但对于降维任务而言有时候并不是必要的，因为降维并不需要输出标签。另一方面，分类器都带有一个准则函数，这个准则函数定义了熵，即定义了什么是混乱，什么是有序。从控制论的角度上看，控制工程中，误差是系统校正的核心，没有误差便没有反馈，没有反馈，则系统难以控制。机器学习也是同样的道理，没有误差，则无法对参数进行校正，机器便无法从样本中学习。误差的来源，源自于准则的定义，即预先预定什么是对的，什么是错的，对与错之间就会存在一个误差，如果机器产生了一个误差，利用这个误差便可以对参数进行校正。神经网络中，常用的分类器有三种：平方误差、softmax、支撑向量机，尽管支撑向量机是一种重要的分类器，但是鉴于篇幅有限，我们并不打算在本文中介绍支撑向量机，关于其原理读者可参考文献[32, 33]。

5.3.1 平方误差分类器

严格来说，平方误差并不能算作一个分类器，它充其量只能算作一个准则，这个准则除了在神经网络中有应用之外，在很多领域也有广泛的应用。平方误差，或者说二范数，其准则函数定义为

$$J = \sum_{i=1}^d (x_i - t_i)^2 \quad (5-18)$$

式中， x_i 为网络的输出，这个输出由输入数据与网络参数共同决定， t_i 代表标签值，也成为教师信号，由数据的标签值决定， d 则是 x 的维数。假设一个神经网络的最顶层有4个节点，网络的输出为 $[-0.1 \ 0.1 \ 0.8 \ 0.1]$ ，而我们的期望输出是 $[0 \ 0 \ 1 \ 0]$ （再一次强调，这个期望输出由标签值决定），那么实际输出与期望输出之间就会存在一个误差向量 $[-0.1 \ 0.1 \ 0.2 \ 0.1]$ ，将其转化成平方误差后为 $[0.01 \ 0.01 \ 0.04 \ 0.01]$ ，利用式(5-18)可以很容易算的此时 $J = 0.07$ 。事实上， $J = 0.07$ 对我们而言没有太大用处，它最多只能刻画总的误差量有多大，真正对我

们有用的是平方误差向量，利用这个向量，可以将误差反向传播回神经网络的底层，从而自顶向下就网络参数进行校正，关于这个过程更多的细节，我们将会留到反向传播章节介绍。

5.3.2 softmax分类器

实际中，为了方便分类，我们往往希望网络的输出是“k中取1”的形式，例如，假设我们有四个类别，如果使用二进制编码的话，我们只需两个节点即可，即00代表第一类，01代表第二类，以此类推。如果我们使用所谓的“k中取1”形式，我们便需要四个节点，即0001代表第一类，0010代表第二类，0100代表第三类，1000代表第四类。为了实现“k中取1”的形式，我们需要引入softmax分类器，softmax分类器可以认为是logistics回归的推广，在使用logistics回归进行分类时，只能实现两类分类，而softmax分类器可以实现多类分类功能。

在softmax分类器中，假设容量为n的训练集为 $S = \{(x^{(1)}, y^{(1)}), \dots, (x^{(n)}, y^{(n)})\}$ ，由于标签y的取值可以有c种，因此 $y^{(i)} \in \{1, 2, \dots, c\}$ 。我们引入记号 $(\phi_1, \phi_2, \dots, \phi_c)$ 来表示各个类别的输出概率，由于 $\sum \phi_i = 1$ ，因此记号存在冗余，只需要 $c - 1$ 个记号 $(\phi_1, \phi_2, \dots, \phi_{c-1})$ 即可，对于 ϕ_i ，有

$$\phi_i = P(y = i; \phi) \quad (5-19)$$

以及

$$\phi_c = P(y = c; \phi) = 1 - \sum_{i=1}^{c-1} \phi_i \quad (5-20)$$

注意，正如我们之前提到的， ϕ 中参数存在冗余，所以 ϕ_c 并不是参数，而是一个为了方便后面讨论所引入的一个记号。

由于softmax的概率分布也属于指数分布族，而在指数分布族中，对于概率分布而言，都可以写成如下形式

$$P(y; \eta) = b(y) \exp \left[\eta^T T(y) - a(\eta) \right] \quad (5-21)$$

在softmax中，我们记 $T(y) \in R^{c-1}$ 为

$$T(1) = \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \quad T(2) = \begin{bmatrix} 0 \\ 1 \\ \vdots \\ 0 \end{bmatrix} \quad \dots \quad T(c-1) = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 1 \end{bmatrix} \quad T(c) = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \quad (5-22)$$

记 $[T(y)]_i$ 为 $T(y)$ 的第*i*个元素，记 $1\{\cdot\}$ 为示性函数，即

$$1\{A\} = \begin{cases} 1, & \text{若命题 } A \text{ 为真} \\ 0, & \text{若命题 } A \text{ 为假} \end{cases} \quad (5-23)$$

因为 $[T(y)]_i = 1\{y = i\}$ ，因此

$$\begin{aligned} \mathbb{E}\left[[T(y)]_i\right] &= [T(y)]_i \cdot P(y = i) \\ &= P(y = i) \\ &= \phi_i \end{aligned} \quad (5-24)$$

而

$$\begin{aligned} P(y; \phi) &= \phi_1^{1\{y=1\}} \phi_2^{1\{y=2\}} \cdots \phi_c^{1\{y=c\}} \\ &= \phi_1^{1\{y=1\}} \phi_2^{1\{y=2\}} \cdots \phi_c^{1-\sum_{i=1}^{c-1} 1\{y=i\}} \\ &= \phi_1^{[T(y)]_1} \phi_2^{[T(y)]_2} \cdots \phi_c^{1-\sum_{i=1}^{c-1} [T(y)]_i} \\ &= \exp \left\{ \ln \left[\phi_1^{[T(y)]_1} \phi_2^{[T(y)]_2} \cdots \phi_c^{1-\sum_{i=1}^{c-1} [T(y)]_i} \right] \right\} \\ &= \exp \left\{ [T(y)]_1 \ln \phi_1 + [T(y)]_2 \ln \phi_2 + \cdots + \left[1 - \sum_{i=1}^{c-1} [T(y)]_i \right] \ln \phi_c \right\} \\ &= \exp \left\{ [T(y)]_1 \ln \frac{\phi_1}{\phi_c} + [T(y)]_2 \ln \frac{\phi_2}{\phi_c} + \cdots + \ln \phi_c \right\} \end{aligned} \quad (5-25)$$

对比式(5-21)，得

$$b(y) = 1 \quad (5-26)$$

$$\eta = \left[\begin{array}{cccc} \ln \frac{\phi_1}{\phi_c} & \ln \frac{\phi_2}{\phi_c} & \cdots & \ln \frac{\phi_{c-1}}{\phi_c} \end{array} \right]^T \quad (5-27)$$

$$a(\eta) = -\ln \phi_c \quad (5-28)$$

定义

$$\eta_i = \ln \frac{\phi_i}{\phi_c} \quad \eta_c = \ln \frac{\phi_c}{\phi_c} = 0 \quad (5-29)$$

由于

$$e^{\eta_i} = \frac{\phi_i}{\phi_c} \quad (5-30)$$

则

$$\phi_c \cdot e^{\eta_i} = \phi_i \quad (5-31)$$

且

$$\phi_c \sum_{i=1}^c e^{\eta_i} = \sum_{i=1}^c \phi_c e^{\eta_i} = \sum_{i=1}^k \phi_i = 1 \quad (5-32)$$

从而

$$\phi_i = \phi_c e^{\eta_i} = \frac{\exp(\eta_i)}{\sum_{i=1}^c \exp(\eta_i)} \quad (5-33)$$

即

$$P(y = i; \phi) = \phi_i = \frac{\exp(\eta_i)}{\sum_{i=1}^c \exp(\eta_i)} \quad (5-34)$$

以及

$$P(y = i|x; \theta) = \frac{\exp(\theta_i^T x)}{\sum_{j=1}^c \exp(\theta_j^T x)} \quad (5-35)$$

本质上，softmax分类器都可以看做是一种两层神经网络，与常规的神经网络中使用sigmoid函数作为激活函数不同，在softmax分类器中，激活函数 $f(net_k) \propto e^{net_k}$ ，对应的激活概率定义为^[3]

$$z_k = \frac{e^{net_k}}{\sum_{i=1}^c e^{net_i}} \quad (5-36)$$

式中， z_k 代表第 k 个节点激活的概率， net_k 代表第 k 的节点的净激活，取值为输入特征 x 的线性函数，即 $net_k = \theta_k x^T$ ， c 为类别的总数。在式(5-36)中分母的作用下，我们对每个类别的输出都进行了归一化，所以总的激活概率为1。在识别中，计算各个类别的激活概率后，只需选取最大激活概率所对应的类别作为输出类别即可。

5.4 神经网络的反馈

神经网络的反馈是基于准则函数的反馈，准则函数作为误差的度量，刻画了网络的输出与训练集的标签两者之间的差异。整个反馈过程中，实质上就是准则函数最优化的过程，即对网络的参数进行更新，使得误差不断减小。假设一个以参数 θ 为自变量的误差函数 $f(\theta)$ ，反馈过程就相当于不断寻求 θ 使得 $f(\theta)$ 最小的过程。如果 $f(\theta)$ 是凸的，即Hessian矩阵是正定的，那么我们称这个问题为凸优化问题，我们可以有很多策略解决凸优化问题，但实际中遇到的问题基本都是非凸问题，解决非凸问题，一个简单有效的方法是利用梯度下降。想象我们在一座山上，为了更快速的下山，我们可以每一步都朝着当前最陡峭的方向（即梯度方向）前进，梯度下降也是基于这个原理，每一次迭代，我们都在当前的 θ 下计算 $f(\theta)$ 的梯度 $\Delta_\theta f(\theta)$ ，让 θ 加上这个梯度分量，也就是让它朝着梯度下降的方向迭代。由于梯

度下降的每一步都是基于当前的参数 θ ，所以这是一个贪婪算法，并不能保证收敛到全局最优。如果 $f(\theta)$ 是凸的，那么梯度下降在大部分情况下它会收敛到全局最优（但并不绝对，受学习率影响），然而，实际任务由于其非凸性，我们基本上不可能收敛到全局最小值，我们得到的往往是局部极小值。另外，值得一提的是，收敛到全局最小值是没有意义的，因为这个全局最小值是基于训练集的最小值，一旦收敛到此处，意味着网络可以很好地刻画训练集，但并不意味它同样可以很好地泛化到测试集上，这个时候往往伴随着较高的过拟合风险，我们不应一味地最求全局最小，而应在训练集与测试集两者之间进行一个权衡。梯度下降，因其简单有效的特点，称为优化问题中一个很常用的策略。

5.4.1 分类器参数校正

20世纪80年代以前，人们并没有找到一个较好的方法训练神经网络，这种状况一直维持到1986年，在这一年里由Rumelhart与McClelland为首的科学家小组⁴提出了反向传播算法，首次在神经网络中利用梯度下降逐层地训练网络参数，每训练完一层后将当前层的误差传递回前一层，依次迭代，最后完成整个网络的训练。在这套方法中，核心点有两个：如何利用最顶层（即分类器）的误差来更新最顶层的参数以及如何在某一层网络更新完参数后将该层的误差传播回前一层，我们将会在本小节中讨论最顶层的参数校正，并在下一小节讨论误差传播。

5.4.1.1 平方误差分类器参数校正

由于在平方误差分类器中，准则函数定义为

$$J(\theta) = \frac{1}{2} \sum_{i=1}^c (t_i - z_i)^2 = \frac{1}{2} \|t - z\|^2 \quad (5-37)$$

式中， z_i 为训练集的标签值，而网络的输出 t_i 又是由前一层的净激活经过非线性映射后得到的，即

$$t_i = f(net_i) \quad (5-38)$$

式中， net_i 为输出层的前一层，即倒数第二层的净激活，而这个净激活又被定义为

$$net_i = wx + b \quad (5-39)$$

式中， w 为输出层与倒数第二层两者之间的权值， b 为偏置，因此，我们不难通过链式求导得到误差相对于参数的梯度，即 $J(\theta)$ 相对于 θ （这里的 θ 即为 w 和 b ）的导数

⁴事实上反向传播是由很多人在同一时代同时独立提出的

$$\frac{\partial J(\theta)}{\partial w} = \frac{\partial J(\theta)}{\partial net} \frac{\partial net}{\partial w} \quad (5-40)$$

以及

$$\frac{\partial J(\theta)}{\partial b} = \frac{\partial J(\theta)}{\partial net} \frac{\partial net}{\partial b} \quad (5-41)$$

如果此时的激活函数是sigmoid函数，那么容易知道

$$\frac{\partial J(\theta)}{\partial net} = net(1 - net) \quad (5-42)$$

如果激活函数是ReLU函数，也容易得到

$$\frac{\partial J(\theta)}{\partial net} = \begin{cases} 1 & \text{若 } net > 0 \\ 0 & \text{若 } net \leq 0 \end{cases} \quad (5-43)$$

而 $\partial net / \partial \theta$ 只是一个线性函数，同样容易求得

$$\frac{\partial net}{\partial w} = x \quad \frac{\partial net}{\partial b} = 1 \quad (5-44)$$

利用(5-42)或(5-43)⁵以及(5-44)便可以求取式(5-40)以及(5-41)中参数应该修改的幅度，即梯度。

5.4.1.2 softmax分类器参数校正

softmax的训练中，由于网络的输出为

$$z = \mathbb{E}[T(y)|x; \theta] = \mathbb{E} \left[\begin{array}{c} 1\{y=1\} \\ 1\{y=2\} \\ \vdots \\ 1\{y=c-1\} \end{array} \middle| x; \theta \right] = \begin{bmatrix} \phi_1 \\ \phi_2 \\ \vdots \\ \phi_{c-1} \end{bmatrix} = \begin{bmatrix} \frac{\exp(\theta_1^T x)}{\sum_{j=1}^c \exp(\theta_j^T x)} \\ \frac{\exp(\theta_2^T x)}{\sum_{j=1}^c \exp(\theta_j^T x)} \\ \vdots \\ \frac{\exp(\theta_{c-1}^T x)}{\sum_{j=1}^c \exp(\theta_j^T x)} \end{bmatrix} \quad (5-45)$$

对应的似然函数为

$$\mathcal{L}(\theta) = \prod_{i=1}^n P(y^{(i)}|x^{(i)}; \theta) \quad (5-46)$$

对应的对数似然函数为

$$\begin{aligned} \ell(\theta) &= \sum_{i=1}^n \ln P(y^{(i)}|x^{(i)}; \theta) \\ &= \sum_{i=1}^n \sum_{j=1}^c \ln \left(\frac{\exp(\theta_j^T x^{(i)})}{\sum_{t=1}^c \exp(\theta_t^T x^{(i)})} \right)^{1\{y^{(i)}=j\}} \\ &= \sum_{i=1}^n \sum_{j=1}^c 1\{y^{(i)}=j\} \ln \left(\frac{\exp(\theta_j^T x^{(i)})}{\sum_{t=1}^c \exp(\theta_t^T x^{(i)})} \right) \end{aligned} \quad (5-47)$$

⁵这取决于你的激活函数选取

式(5-47)的相反数也被称为交叉熵，在训练过程中充当准则函数，我们对其求导，有

$$\begin{aligned}\frac{\partial \ell(\theta)}{\partial \theta_j} &= \sum_{i=1}^n \sum_{j=1}^c 1\{y^{(i)} = j\} \left[x^{(i)} - \frac{\partial}{\partial \theta_j} \ln \sum_{t=1}^c \exp(\theta_t^T x^{(i)}) \right] \\ &= \sum_{i=1}^n \sum_{j=1}^c 1\{y^{(i)} = j\} \left[x^{(i)} - \frac{\exp(\theta_j^T x^{(i)})}{\sum_{t=1}^c \exp(\theta_t^T x^{(i)})} x^{(i)} \right] \\ &= \sum_{i=1}^n \sum_{j=1}^c 1\{y^{(i)} = j\} x^{(i)} \left[1 - P(y^{(i)} = j | x^{(i)}; \theta) \right]\end{aligned}\quad (5-48)$$

由于 $1\{y^{(i)} = j\} \in \{0, 1\}$ ，所以写成向量形式为

$$\frac{\partial \ell(\theta)}{\partial \theta_j} = \sum_{i=1}^n x^{(i)} \left[1\{y^{(i)} = j\} - P(y^{(i)} = j | x^{(i)}; \theta) \right] \quad (5-49)$$

通过使用式(5-49)进行迭代，我们便可以使用梯度上升（下降）进行搜索，从而最大化似然（或最小化交叉熵）来训练softmax分类器。

5.4.2 误差传播

为了简单起见，我们使用一个三层神经网络对误差传播进行讨论，但通过这里的讨论，我们可以很容易将神经网络扩展到多层结构。

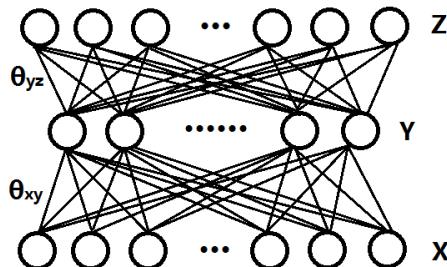


图 5-10 三层神经网络的网路构型

如图5-10中的三层神经网络， y 层到 z 层为分类器，根据之前的讨论，我们可以利用式(5-40)和式(5-41)求取 $\partial J(\theta)/\partial \theta_{yz}$ 的值，如果我们要求取 $\partial J(\theta)/\partial \theta_{xy}$ 的值，那么通过链式求导，我们有

$$\frac{\partial J(\theta)}{\partial \theta_{xy}} = \frac{\partial J(\theta)}{\partial y} \frac{\partial y}{\partial net_y} \frac{\partial net_y}{\partial \theta_{xy}} \quad (5-50)$$

而我们又可以求得

$$\frac{\partial J(\theta)}{\partial y} = \frac{\partial J(\theta)}{\partial net_z} \frac{\partial net_z}{\partial y} \quad (5-51)$$

如果我们定义式(5-40)以及式(5-41)中的误差为

$$\delta_z = \frac{\partial J(\theta)}{\partial net_z} \quad (5-52)$$

那么我们可以将式(5-51)写为

$$\frac{\partial J(\theta)}{\partial y} = \delta_z \frac{\partial net_z}{\partial y} = \delta_z \theta_{yz}^T \quad (5-53)$$

式中， $\partial J(\theta)/\partial y$ 就是分类器输出层（Z层）的误差传播回其上一层（Y层）的误差，如果我们定义Y层的误差为

$$\delta_y = \frac{\partial J(\theta)}{\partial y} \frac{\partial y}{\partial net_y} \quad (5-54)$$

则我们可以利用后一层传播回来的误差 δ_z 计算该层的误差

$$\delta_y = \delta_z \theta_{yz}^T \cdot \frac{\partial y}{\partial net_y} \quad (5-55)$$

从而我们可以利用该层的误差 δ_y 计算该层参数 w_{xy} 的增量，即梯度

$$\frac{\partial J(\theta)}{\partial \theta_{xy}} = \delta_y \cdot \frac{\partial net_y}{\partial \theta_{xy}} \quad (5-56)$$

通过式(5-56)对参数 θ_{xy} 进行校正，校正完毕后，如果网络不止三层，而是四层，假设X层的下一层为P层，那么为了计算连接X层与P层参数 θ_{px} 的梯度，利用链式求导，同样有

$$\frac{\partial J(\theta)}{\partial \theta_{px}} = \frac{\partial J(\theta)}{\partial x} \frac{\partial x}{\partial net_x} \frac{\partial net_x}{\partial \theta_{px}} \quad (5-57)$$

与之前的做法类似，我们有

$$\begin{aligned} \frac{\partial J(\theta)}{\partial x} &= \frac{\partial J(\theta)}{\partial y} \frac{\partial y}{\partial net_y} \frac{\partial net_y}{\partial x} \\ &= \delta_y \cdot \theta_{xy}^T \end{aligned} \quad (5-58)$$

同样的道理，X层的误差 δ_x 定义为

$$\delta_x = \frac{\partial J(\theta)}{\partial x} \frac{\partial x}{\partial net_x} = \delta_y \theta_{xy}^T \cdot \frac{\partial x}{\partial net_x} \quad (5-59)$$

则参数 θ_{px} 的梯度便可以利用 δ_x 计算

$$\frac{\partial J(\theta)}{\partial \theta_{px}} = \delta_x \cdot \frac{\partial net_x}{\partial \theta_{px}} \quad (5-60)$$

如果还有更多的层，其原理雷同，我们不再详细叙说。现在让我们抛开以上的数学内容从网络的结构上解释这个算法为什么叫做误差反向传播，首先我们观察各层的误差，即

$$[\delta_z \ \delta_y \ \delta_x] = \left[\frac{\partial J(\theta)}{\partial net_z} \ \delta_z \theta_{yz}^T \cdot \frac{\partial y}{\partial net_y} \ \delta_y \theta_{xy}^T \cdot \frac{\partial x}{\partial net_x} \right] \quad (5-61)$$

式中， θ^T 起到的作用是将误差 δ 反向注入的行为，这个行为如图5-11所示

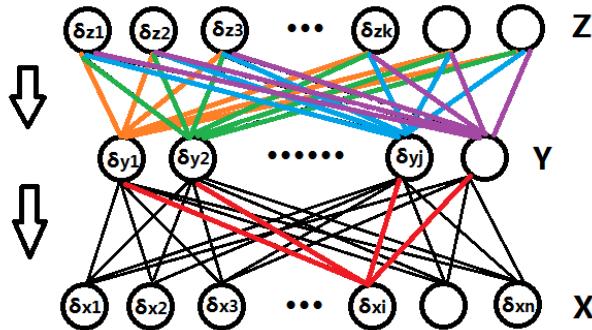


图 5-11 误差反向注入

观察式(5-61)，我们不难看出，除了最顶层外，剩余层的误差形式都是类似的，如果用一句话概括反向传播算法，便是：将 ℓ 层的误差注入到 $\ell - 1$ 层中，再乘上 $\ell - 1$ 层激活函数的导数，得到 $\ell - 1$ 层的误差，利用这个得到的误差，对 $\ell - 1$ 层的参数进行更新，更新完毕后，将 $\ell - 1$ 层的误差重新注入到 $\ell - 2$ 层，重复上述步骤直至误差传遍整个网络。但是这里有个例外，最顶层的误差定义与剩余层的不一样，之所以会这样是因为最顶层的误差并不是通过注入方式得到的，而是人为定义得到的，因为这个误差源自于准则函数。

我们可以看到，反向传播这种策略实际上与前向传播是类似的，反向传播也是一种贪婪算法，这将会导致一些问题。因为每一次误差传播都是更新参数后再将误差注入回前一层，这并不能保证计算得到的梯度就是真实的梯度，一旦网络的层数过深，将会导致前面层的真实导数与利用反向传播计算得到的导数相差过大。另外，如果使用sigmoid函数作为激活函数，它将很容易进入饱和状态，前面层的梯度接近于0，从而参数无法更新，这种现象我们称之为梯度消失。一种对抗梯度消失的方法是将sigmoid函数换成ReLU激活函数，关于ReLU为什么可以抵抗梯度消失的原理目前尚未研究出来，但是实验现象表明它确实能抑制梯度消失。

5.5 深度置信网络

深度置信网路（Deep Belief Networks，简写为DBN）本质就是一种传统的神经网络，但是它在传统神经网络的基础上加入一些变动。首先，这是一种深层的神经网络，而传统的神经网络一般只设计为三层结构。另外，深度置信网络是一个以受限玻尔兹曼机为基础，多个受限玻尔兹曼机堆出来的多层神经网络。图5-12展示了一个深度为4的深度置信网络。

我们之所以采用深度结构，是因为在神经元总数不变的前提下，深度结构的表达能力比浅结构的表达能力更强。之所以要在这个结构中引入受限玻尔兹曼机，

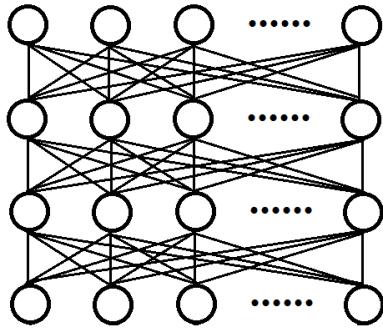


图 5-12 DBN网络构型

是因为传统的深度结构无法训练。正如我们前面提及到的，深度结构会在网络的较浅层出现梯度消失现象，导致浅层无法对参数进行校正。由于数据必须从浅层神经元逐层地传播到深层神经元，一旦浅层的参数无法校正，将会导致深层的网络也无法进行参数校正，因为浅层参数无法校正意味着浅层无法对数据进行合理地重编码，数据经过浅层后得到的是混乱的数据，尽管深层不会产生太大的梯度消失现象，但数据经过浅层的打乱后，数据已经混乱了，训练也便没有意义。另一方面，传统的神经网络初始值设定为随机值，这些随机值一般设定为一个均值为0，方差较小的高斯分布，神经网络是一个非凸问题，局部最小值众多，网络最后收敛到的最小值很大程度上取决于初始值的选取，随机选取初始值虽然也是一个可行的方法，但是如果能让参数的初始值设定在一个较合理的初始值，将会很大程度地改善网络的收敛性能，而受限玻尔兹曼机一个重要的贡献在于，将深度神经网络的参数初始化到一个较好的值。

深度置信网络的训练分为两个阶段，分别是预训练阶段和参数微调阶段。在DBN的预训练阶段中，将相邻两层看做一个受限玻尔兹曼机，采用受限玻尔兹曼机的训练方法，将原始数据作为最底层的输入，每层RBM隐含层的输出作为后一层的输入，然后进行逐层贪婪的无监督训练。对于每层RBM，其训练过程描述如算法5-1所示

Input: 由前一层网络提供的含有 n 个样本的数据集 $S = \{x_i\}_{i=1}^n$; 网络参数 w, b_h, b_v ; 动量项系数 p ; 学习率 η ; 权衰减系数 e ; CD-k中的参数 k

Output: 训练好的网络参数;

由RBM提取到的数据 $S' = \{y_i\}_{i=1}^n$

```

1 for  $i = 1; i \leq 1; i++ \text{ do}$ 
2    $v_0 = v_t = x_i$ 
3    $h_0 = \text{sampling h given v } (v_0)$ 
4   for  $j = 0; j < k; j++ \text{ do}$ 
5      $h_t = \text{sampling h given v } (v_t)$ 
6      $v_t = \text{sampling v given h } (h_t)$ 
7   end
8    $\Delta_w = v_t^T * h_t - v_0^T * h_0$ 
9    $\Delta_{b_v} = v_t - v_0$ 
10   $\Delta_{b_h} = h_t - h_0$ 
11   $w = e * (p * w + \eta * \Delta_w)$ 
12   $b_v = b_v + \eta * \Delta_{b_v}$ 
13   $b_h = b_h + \eta * \Delta_{b_h}$ 
14   $y_i = \text{sampling h given v } (x_i)$ 
15 end

```

算法 5-1 受限玻尔兹曼机训练算法

算法5-1中，学习率 η 、动量项系数 p 以及权衰减系数 e 我们将会留到网络设计技巧中讨论，这些技巧在数学推导的过程中是不必要的，然而在实际工程中是必要的，有时候缺了它们网络训练会失败。当逐层训练完毕后，网络的参数已经初始化到一个较好的位置^[34]。在参数微调阶段，接着执行全局的反向传播算法进行有监督的权值微调。通过这样的方法，可以避免单纯地使用反向传播方法中会出现的陷入局部最优问题，由于识别的过程中，数据是逐层地进行维度变化，所以DBN也可以认为是一种特征提取方法，对应的，深度学习有时候也称之为“特征学习”^[35]。

5.6 本章小结

本章中，我们从传统的神经网络说起，介绍了神经元的工作原理以及神经网络的前向传播。当数据经过神经网络的逐层编码后，得到的编码并不是人类所能理解的编码，因此我们介绍了两种分类器将这些编码转换成自然语言所能描述的编码。随后我们介绍了神经网络的反向传播算法，最后引出深度置信网络。尽管这章名为“深度置信网络”，我们却对“深度置信网络”的介绍篇幅很短，但由于深度置信网络中并没有太多额外的东西，很多思想都是与传统神经网络是相同的，所以我们花了较大的篇幅来介绍传统神经网络。在本章中，我们依然留下一些问题没有解决，即训练过程中的一些额外参数，例如学习率、动量项等，关于这些内容，由于其更偏向于工程内容，我们将其集中到“神经网络设计技巧”章节中讨论。

第6章 卷积神经网络

卷积神经网络灵感源自哺乳动物的视觉系统，它是唯一一个不需要预训练便能直接训练的深度网络。LeCun在1989年设计了第一个卷积神经网络，并将这个网络运用到邮编数字识别中取得了很好的效果，随后，文献[19]将卷积神经网络应用到文档识别中，实现了可理解数字串的网络，这是一个重大的突破，因为这涉及到数字间的上下文。但卷积神经网络在提出的二十多年里一直默默无闻，直到最近几年，人们发现在图像识别中卷积神经网络相比于其他的模型能更好地进行特征提取才被重视。目前，几乎所有最好的识别系统都是基于卷积神经网络的，从某种意义上而言，卷积神经网络相当于深度学习的代言人。

6.1 卷积神经网络综述

卷积神经网络可以看做是一种特殊的神经网络，在深度置信网络中，网络节点是全连接的，而卷积神经网络中连接是局部的。此外，卷积网络强制权值共享，对比于深度置信网络，卷积网络的这些特性都体现着非常强的正则。

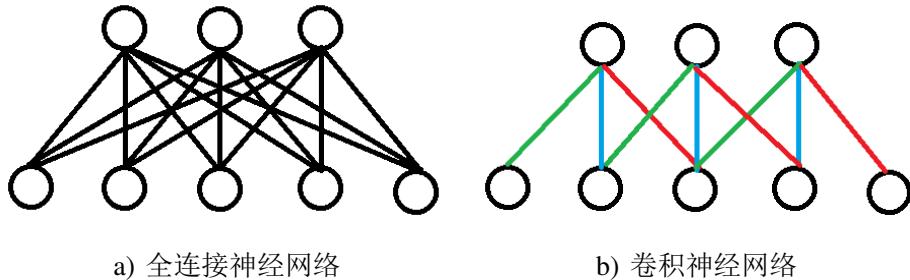


图 6-1 二维视觉下的全连接神经网络与卷积神经网络

如图6-1 a)所示为传统的全连接神经网络，我们可以看到，每一个上层节点与下层节点都含有连接，而所有的连接都是独立无关联的，即每个连接权值都不相等。对应的，图6-1 b)所示为卷积神经网络，在这种网络构型中，每个上层节点只与部分的下层节点连接，并且，这些连接的权值是共享的，即相同颜色的连接代表其权值相等。局部连接将会大大减少网络的连接数量，而权值共享又会大大减少网络的参数数量。例如在图6-1 a)中，连接数量为 $3 \times 5 = 15$ ，由于权值不共享，其参数数量也是15。而图6-1 b)的连接数量为 $3 \times 3 = 9$ ，权值共享使得网络的参数只有3个。卷积网络的设计目的是让网络拥有更多的连接，而拥有更少的权值。尽管

这里连接数量上卷积网络要比全连接网络少，但是我们后面将会看到，卷积网络将通过多张特征图构造出更多的连接。

卷积神经网络，一般用于图像识别与声音识别两个领域，因为这两个领域带着明显的二维特性。例如在图像识别中，图像即可以看做是高维的，也可以看做是二维的，如果将它看做高维的，那么就是将像素点展开成为一个高维列向量，展开后的图像将不再具有原始的面貌。如果将它看做是二维的，就是保留图像的原貌，利用二维的平面直角坐标系来描述。声音之所以可以看做二维的，是因为它带有时间这一维度，关于语音识别，我们不过多讨论，更多的细节请参考文献[36–38]。

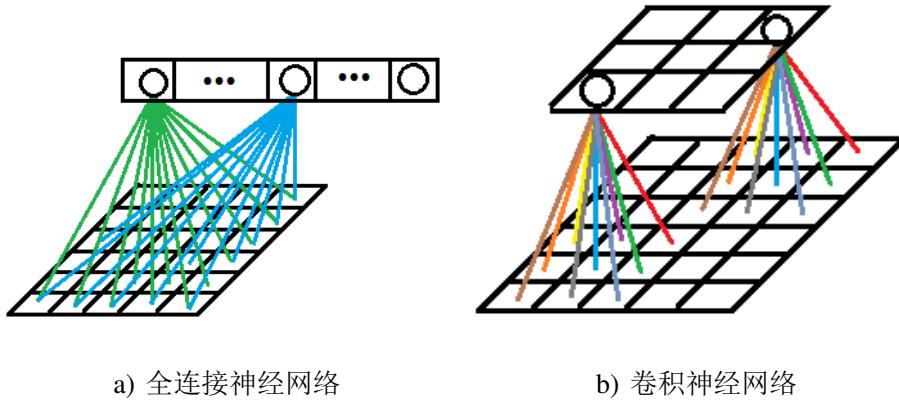


图 6-2 三维视觉下的全连接神经网络与卷积神经网络

由于卷积网络保留了图像的二维面貌，因此相比于全连接神经网络而言，其网络构型是一个三维构型，如图6-2 a)所示为三维视觉下全连接神经网络。由于上层节点是一个高维列向量，我们可以认为这个向量是一维的，并且这个向量里的每一个元素都与输入图像上的每一个像素点连接，不同元素之间的连接权值是不同的。而图6-2 b)所示是三维视觉下的卷积神经网络，我们可以看到，上层节点是一个二维矩阵，因此它的特征我们可以认为是二维的，这些上层节点组成的二维矩阵我们称为特征图（feature map），特征图里的每一个元素，都只与输入图像上的一小块区域有连接，并且，特征图里的不同元素的连接权值是相等，这些相同的连接权值我们称之为卷积核（convolution kernel）。图6-2 b)仅仅是一张特征图，实际中的卷积神经网络，往往通过多个不同的卷积核，卷积出多张特征图。

6.2 卷积神经网络的前馈

一个完整的卷积神经网络应包含三个内容：卷积层、采样层以及全连接层，关于各层的细节以及作用我们将在后面的小节中叙述，现在让我们先大致了解卷积神经网络的结构。如图6-3所示是卷积神经网络其网络构型

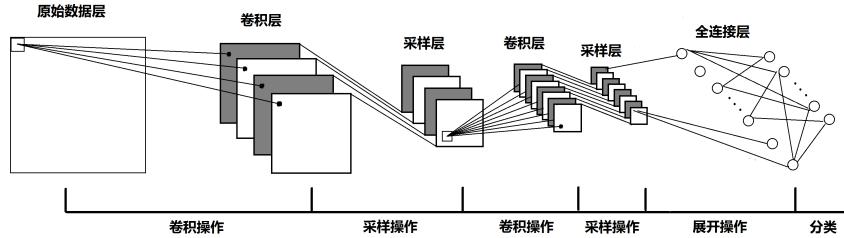


图 6-3 卷积神经网络网络构型

在这种结构中，卷积层与采样层交错出现，在网络顶端采用全连接神经网络或其他分类器。图中卷积层里的每一张特征图，其执行过程都如图6-2 b)所示，这些不同的特征图对应着不同的卷积核，多张特征图，增加了网络的连接数目，但由于每张特征图只对应一个卷积核，所以网络的参数（即卷积核）的数目相比与全连接神经网络参数的数目而言是很小的。

6.2.1 卷积

卷积神经网络所使用的卷积即二维离散卷积，其定义为

$$y[s, t] = \sum_{i=1}^{m_1+m_2-1} \sum_{j=1}^{n_1+n_2-1} x[i, j] \cdot k[s - i + 1, t - j + 1]$$

$$1 \leq s \leq m_1 + m_2 - 1$$

$$1 \leq t \leq n_1 + n_2 - 1$$
(6-1)

式中， x 为 $m_1 \times n_1$ 的矩阵，称为原始数据， $x[i, j]$ 代表 x 中的第*i*行第*j*列元素。 k 为 $m_2 \times n_2$ 的矩阵，称为卷积核， $k[i, j]$ 代表 k 中的第*i*行第*j*列元素。 y 为 $(m_1 + m_2 - 1) \times (n_1 + n_2 - 1)$ 的矩阵，称为卷积结果， $y[i, j]$ 代表 y 中的第*i*行第*j*列元素。如果从图像上来解释二维离散卷积，卷积层的操作相当于图6-4所描述的过程。图中，原始图像是一张 5×5 像素的数字“4”的二值图像，如果我们定义卷积核为

$$kernel = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$
(6-2)

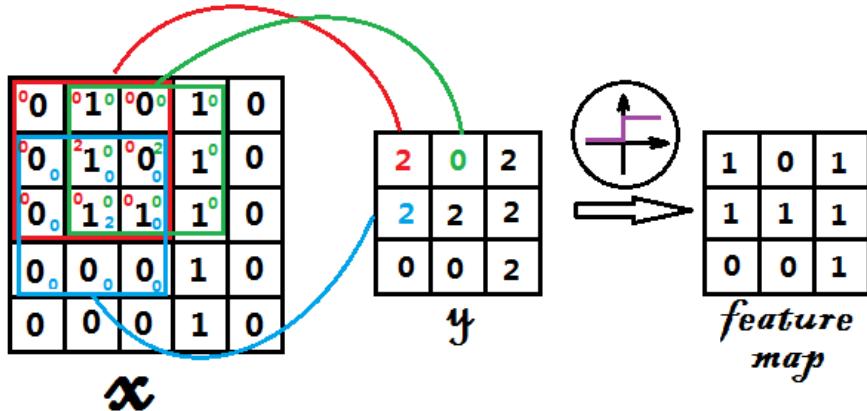


图 6-4 卷积神经网络网络构型

那么经过二维离散卷积后，其得到的卷积结果为

$$y = x * \text{kernel} = \begin{bmatrix} 2 & 0 & 2 \\ 2 & 2 & 2 \\ 0 & 0 & 2 \end{bmatrix} \quad (6-3)$$

将卷积结果再经过一个阶跃函数进行非线性变换后，我们可以看到，卷积层通过式(6-2)中定义的卷积核对原始图像（即数字4）进行卷积后，其非线性化后的结果依然保留着数字4的特征。由于卷积核是固定的，因此卷积操作经常被认为是一种滤波，即用卷积核去检测特征，将这些特征提取出来。

图6-4只是利用一个卷积核对原始图片进行卷积得到一张特征图，事实上，正如图6-3所示，如果我们利用多个卷积核对原始图像进行卷积，那么便可以得到多张特征图。多张特征图，意味着可以提取到原始图像的多个特征，例如在手写数字识别中，我们提取了三张特征图，其中一张特征图表明左上角有一点，一张特征图表明右上角有一个折线，剩下的一张特征图表明右下角有一个点，那么通过这三张特征图，我们就可以判定这个数字是“7”。特征图的作用，在于降低数据的冗余信息，例如“7”这个数字中，横线事实上只需要两个点就可以确定一条直线，竖线也同理只需要两点便可确定一条直线，然而横线与竖线之间存在一个相对位移，因此折线的作用便是刻画相对位移的。

图6-4中的例子只针对与原始输入数据是一张图像的情况，实际上，卷积操作应该泛化到输入图像为多张的情况。例如，图6-3中的第二个卷积层，其输入图像便是多张图像。实际中，输入图像也不太可能是一张图像，一张输入图像的情况往往只出现于灰度图像中。然而在彩色图像中，我们知道，彩色图像是具有三张矩阵的，分别代表红(R)、绿(G)、蓝(B)。对于输入图像为多张的情况，解决

的方法有两种。一种是利用同一个卷积核去卷积所有的 n 输入图像，将得到的 n 张卷积结果进行相加合并成为一张，在将它进行非线性映射，从而得到一张特征图，这个过程如图6-5所示。

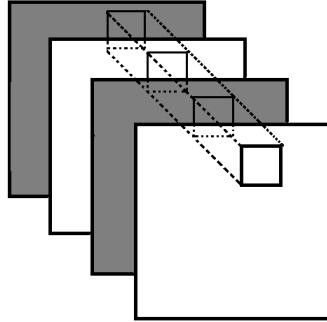


图 6-5 使用同一卷积核卷积多张图像

另外一种解决方法是，假设我们有 m 张输入特征图，而我们想要卷积后得到 n 张输出特征图，那么我们使用 $m \times n$ 个卷积核，令 $k_{i,j}$ 代表从第 i 张输入特征图映射到第 j 张输出特征图这个过程中所需要的卷积核， b_j 代表卷积完成后第 j 张特征图所要加上的偏置，那么卷积操作就可以描述为：

$$\left[\mathcal{M}_1, \mathcal{M}_2, \dots, \mathcal{M}_m \right] * \begin{bmatrix} k_{1,1} & k_{1,2}, \dots, k_{1,n} \\ k_{2,1} & k_{2,2}, \dots, k_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ k_{m,1} & k_{m,2}, \dots, k_{m,n} \end{bmatrix} \boxplus \left[b_1, b_2, \dots, b_n \right] = \left[\mathbf{M}_1, \mathbf{M}_2, \dots, \mathbf{M}_n \right] \quad (6-4)$$

式中， $\{\mathcal{M}_i\}_{i=1}^m$ 为 m 张输入特征图， $\{\mathbf{M}_j\}_{j=1}^n$ 为 n 张输出卷积结果， \boxplus 为面向元素的加法， $*$ 是我们提出的一种运算，其运算与矩阵乘法类似，唯一不同是在元素操作时，矩阵乘法执行的是乘法操作，而 $*$ 执行的是卷积操作。卷积操作完成后，再进行非线性映射，即可得到多张特征图。式(6-4)实际上类似于全连接神经网络的前向传播，但是不同的是，全连接神经网络中， $m \times n$ 矩阵里的每一个元素代表连接权值，是一个常数，而式(6-4)中， $m \times n$ 矩阵里的每一个元素代表一个卷积核，是一个矩阵。另外一个不同点在于，全连接神经网络执行的是乘法，而式(6-4)中执行的是卷积。

事实上，第一种解决方案只是第二种解决方案的特例，在第一种解决方案中，实际上相当于 $K_{m \times n}$ 中的每一列都强制相等，即

$$k_{1,1} = k_{2,1} = \dots = k_{m,1} \quad (6-5)$$

因此，第一种解决方案意味着更强的正则，或者说更强的惩罚，因为它强制每一列的卷积核都相等。但是我认为，尽管这种方法确实能工作得很好，但这是

不太合理的。例如在图像中，三个输入图像，也就是RGB三个矩阵，采用第一种方案意味着对三个颜色都同等对待，因为作用在三个矩阵上的卷积核是相同的。但我们直觉上可以感觉到，这三种颜色不应该同等对待，而应区分开来，因此在往后的讨论中，我们只讨论第二种解决方案。

6.2.2 采样

卷积得到的特征图，需要经过一个采样层，采样层是针对每一张特征图的，即各张特征图的采样时独立互不干扰的，因此采样得到的特征图与原始的特征图之间是一一对应关系。假设我们使用一个均值采样，对于一个 4×4 的特征图，我们对其每 2×2 区域内取平均，即

$$\begin{bmatrix} 1 & 1 & 2 & 2 \\ 1 & 1 & 2 & 2 \\ 3 & 3 & 4 & 4 \\ 3 & 3 & 4 & 4 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \quad (6-6)$$

采样层的作用在于压缩数据，使得维度不至于快速增长。想象一个原本为 6×6 像素的原始图片，在经过一个 3×3 的卷积核进行卷积后，得到的特征图大小为 4×4 ，但这只是一张特征图，由于在卷积层中我们往往使用多个卷积核进行卷积，假设我们设定卷积层的输出特征图总量为50，那么将会有 $50 \times 4 \times 4 = 800$ 个节点，而原来的节点只有 $5 \times 5 = 25$ 个节点，这将会使节点增加了32倍。采样层一般都是对卷积得到的特征图进行2倍的缩小，仅此经过采样层后，特征图的大小为 2×2 ，此时的节点仅为 $50 \times 2 \times 2 = 200$ 个。

另一方面，采样层可以抑制位移的变化，卷积得到的特征图，在经过小区域内的平均后，弱化了其绝对位置，而保留了其相对位置，需要注意的是，这里的位移并不仅仅只欧式距离里的位移，还应包括图像的伸缩，旋转等，所以这个位移应理解为广义的位移。

采样的方法除了上面提到的平均采样方案，还有一些别的方案，例如xxx文献中介绍了一种自学习的采样，即小区域内并不是简单的求和取平均，而是类似于加权求和取平均，这些权值便是所需要学习的参数。这两种方案并没有孰劣孰优的说法，实际上两者都能很好的工作，但自学习的方法确实是会比简单的平均采样好一些，但由于平均采样更简单，所以我们往后的讨论只使用平均采样。

6.2.3 分类器

原始输入数据经过多次卷积采样后，特征图的尺寸不断缩小，最后将使得卷积无法再进行，此时，我们将这些非常小的特征图展开成为一个列向量。例如，一个原始图片为 32×32 大小的图像，经过一个 5×5 大小的卷积核后得到 28×28 的特征图，对其进行均值采样，将变为 14×14 大小的特征图，再用 5×5 大小的卷积核卷积，将得到 10×10 大小的特征图，采样后为 5×5 大小，此时再执行一次卷积后，特征图大小为 1×1 ，这时候再也无法进行采样了，假设现在有100个 1×1 大小的特征图，那么它便可以展开成为一个100维的列向量。又例如，一个原始图片为 28×28 大小的图像，如果卷积核都设定为 5×5 ，那么经过卷积、采样、卷积、采样后，将得到 4×4 的特征图，此时卷积核尺寸大于特征图的尺寸，卷积无法执行，应将这些特征图展开，假设有10张 4×4 的特征图，那么展开后将得到 $10 \times 4 \times 4 = 160$ 维的列向量。

对于最后的列向量，即可以直接使用分类器，也可以在这些列向量的基础上搭建几层隐含层后再使用分类器，这个分类器可以是全连接神经网络或支撑向量机等，选取哪个取决于设计者的意愿。关于分类器如何使用，读者可以参考前面的章节，在卷积网络最顶层的分类器中，与传统神经网络是相同的。

6.3 卷积神经网络的反馈

卷积神经网络的训练方法与传统神经网络的训练方法类似，都是采用反向传播算法，但由于卷积网络特殊的构型，需要对其进行一些改动，但两者的核心都是相同的，即通过后一层的误差注入前一层中，乘上该层激活函数相对于净激活的偏导数后得到该层的误差，利用这个误差乘上净激活相对于输入的偏导数即可得到参数更新的增量。唯一的不同点在于注入的方式不同。

6.3.1 分类器误差传播

由于卷积神经网络中最顶层与传统的神经网络相同，因此参数校正以及反向传播方式是相同的。有一点需要注意的是，如果网络最后一个卷积层（或采样层）中将特征图拉成一个列向量，那么反向传播的时候需要将列向量还原回特征图形式，例如，如果网络中最后阶段得到10张 4×4 大小的特征图，前向传播过程中会将它们合并成为一个160维的列向量，那么在反向传播过程中，需要将160维的误差列向量还原成10张 4×4 的特征图形式，此时得到的特征图可以看做是误差特征

图。

6.3.2 采样层误差传播

在采样层中，如果我们使用平均采样，由于平均采样并没有额外的参数需要学习，因此只需要将后一层传播回来的误差继续传播回前一层即可。由于采样层是对前一层局部区域的平均，所以在将误差传播回前一层时，只需要将其尺寸放大到相同的比例，对应的局部区域中每一个元素均取采样层中的元素即可。例如，一个缩小比例为2的采样层，假设它接收到后一层传播回来的误差特征图尺寸为 2×2 ，那么这个误差传播回采样层的前一层将得到 4×4 尺寸的特征图，这个过程如式(6-7)所示

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 1 & 2 & 2 \\ 1 & 1 & 2 & 2 \\ 3 & 3 & 4 & 4 \\ 3 & 3 & 4 & 4 \end{bmatrix} \quad (6-7)$$

如果读者在此之前了解过Kronecker积（也称直积），上述过程便是一个Kronecker积过程，即

$$\delta_{conv} = \delta_{sampling} \otimes 1_{n \times n} \quad (6-8)$$

式中需要注意的是， \otimes 是直积符号而不是逻辑电路里的异或符号。 δ_{conv} 代表传播回前一层（即卷积层）的误差， $\delta_{sampling}$ 代表后一层网络传播到采样层的误差， $1_{n \times n}$ 是一个 $n \times n$ 的单位向量， n 的取值等于采样层的缩小比例，例如，如果采样层的缩小比例是2，则 $n = 2$ 。

6.3.3 卷积层误差传播

与传统的全连接神经网络类似，卷积层接收到后一层传播回来的误差后，需要乘以当前层激活函数相对于净激活的偏导数，得到当前层的误差，即

$$\delta_i^\ell = \delta_i^{\ell+1} \cdot \frac{\partial f(net_i)}{\partial net_i} \quad (6-9)$$

由于卷积网络中有多张特征图，所以 δ_i^ℓ 代表当前层（ ℓ 层）的第*i*张误差特征图， $\delta_i^{\ell+1}$ 代表第 $\ell + 1$ 层传播回来的第*i*张误差特征图， net_i 代表第*i*张特征图的净激活。

计算得到当前层的误差后，通过卷积的自相关性，利用这个误差可以计算卷积核以及偏置的迭代增量，即^[39]

$$\frac{\partial J}{\partial k_{i,j}^{\ell}} = \text{rot180}\left(\text{conv}\left(x_i^{\ell-1}, \text{rot180}(\delta_j^{\ell}), 'valid'\right)\right) \quad (6-10)$$

式中， $\text{rot180}(\cdot)$ 执行将一个矩阵旋转180°操作，其目的在于利用卷积的自相关性质， conv 为卷积操作，‘valid’代表卷积操作范围限制在可用区域内。对于偏置，其梯度为^[39]

$$\frac{\partial J}{\partial b_j} = \sum_{u,v} (\delta_j^{\ell})_{uv} \quad (6-11)$$

获取参数的梯度后，对卷积层的参数进行更新，随后将卷积层的误差反向传播到采样层，其公式为^[39]

$$\delta_j^{\ell-1} = f'(u_j^{\ell}) \boxdot \text{conv}\left(\delta_j^{\ell}, \text{rot180}\left(\sum_{i \in M_j^{\ell}} k_{ij}^{\ell}\right), 'full'\right) \quad (6-12)$$

式中， \boxdot 为面向元素的乘法，‘full’代表卷积的操作范围是全图卷积。通过以上策略，对网络进行类似的反向传播，即可对整个网络进行反馈校正。

6.4 本章小结

本章中我们介绍了卷积神经网络中卷积层以及采样层的原理，并介绍了一种卷积神经网络的结构。卷积网络的结构变体有很多，1989年LeCun设计的第一个卷积神经网络仍带有轻微的全连接网络气息^[18]，而1998年LeCun与Bengio等人设计的卷积神经网络在卷积时并不是全特征图卷积，而是选取一部分的特征图进行卷积^[19]。Lee Honglak与Andrew Ng等人将受限玻尔兹曼机与卷积神经网络相结合，形成一种名为Convolutional deep belief networks的神经网络^[40]。文献[41]中介绍了一种使用三维卷积核的卷积网络，因此这种网络可以看做是四维网络，Krizhevsky等人利用它在2012年的ImageNet LSVRC-2010数据集上获得了世界第一的正确率，这种卷积网络成为当前的主流结构。尽管卷积网络的变体众多，但其大体思路是类似的，都是基于卷积核对局部特征进行提取。

第7章 神经网络的设计技巧

神经网络的设计技巧众多，这些技巧大致可分为两大类，一类是训练前对数据的预处理，另一类是训练过程中采用的一些特殊的训练手段。数据预处理的作用在于将原始数据处理成容易训练的数据，加快准则函数的收敛速度，另一方面，原始数据的降维可以在牺牲较小精度的前提下减少运算量，提高程序的执行效率。训练过程中采用的一些特殊手段在数学上是没有太大意义，甚至有时候是违反直觉的，这些技巧是多年来科研人员积累出来的经验，其灵感来源一般靠蒙。尽管这些经验有时候反而会降低网络的性能，但大部分情况下确实会提高性能的，某些情况下，不适用这些技巧，网络的训练初始阶段准则函数无法收敛。具体设计过程中，对于这些技巧，我们应该抱着试一试的心态，如果在网络中使用某个技巧，使得网络性能变好了，那么我们便保留这个技巧，否则如果网络性能变差了，那么就舍弃这个技巧¹。

7.1 数据预处理

数据的预处理大致可分为几类：白化、降维、扩容、归一化。白化的作用在于去除数据耦合的相关性，降维的作用在于压缩数据，加快算法执行效率，由于主成分分析是一种去除相关性的降维方法，因此我们将白化合并到主成分分析的章节中而不另起章节。扩容目的在于利用一些手段，人工地伪造一些训练数据，将训练样本的容量扩大，使得网络的泛化能力变强。归一化的目的在于让数据保持在 $[0, 1]$ 区间内，之所以要在这个区间内，一方面，在一些激活函数，比如Sigmoid函数中，如果数据尺度过大，则使得激活函数迅速饱和，从而梯度接近于0，权值更新缓慢。而数据归一化到较小的数值上时，我们知道，Sigmoid函数在原点附近是一个近似于线性的函数，在训练的初始阶段便可让权值获得较大的梯度。另一方面，如果数据尺度过大，有可能会导致计算机的数值溢出，尤其是激活函数为Sigmoid的时候，因为这个激活函数带有一个指数项，对于某些不自带溢出检测的编程语言如C++而言，这是一场灾难，因为它会导致你的代码莫名其妙地失败。即使是一些自带溢出检测的语言，如Python，当数值溢出时，往往使用Inf或NaN代替，此时进行的不再是数值运算，而是符号运算，这将极大地降低运算速度。

¹但前提是这个技巧被正确使用，即编码正确

7.1.1 降维

人是一种受限于三维空间的动物，我们只能理解三维的而无法想象更高维度的事物。事实上，人同样无法理解二维的或一维的世界，尽管我们可以在脑海中想象二维曲面或一维曲线，但人脑海里的这些事物这些都不是二维，他们只是嵌在三维空间中的二维（或一维）流形²。一张一百万像素的图片，它总能对应着一百万维空间中的某一点，而很多张某个类比的图片，比如狗的图片，将会对应着一百万维空间中的多个点。然而并不是一百万维空间中的任何一点都是狗的图片，它也有可能是猫的图片，但是同样是狗的图片，它会在总会有某些共通性，这些共通性，或者说特征，使得狗这一类图片在一百万维的空间中构成一个流形，如果我们能找到这个流形，那么我们就可以避开一百万维空间，直接在低维流形中进行分类。

由于我们无法理解高维空间，因此我们通过一个三维空间中的二维流形简单叙述这件事，如果我们通过式(7-2)求取三维空间中任何一点的(x, y, z)坐标

$$\alpha = 1.5\pi * (1 + 2 * \text{random}) \quad (7-1)$$

$$x = \alpha * \cos \alpha \quad y = 21 * \text{random} \quad z = \alpha * \sin \alpha \quad (7-2)$$

式中， random 是 $[0, 1]$ 区间内均匀分布的随机数，那么我们就可以得到一个称为“swiss roll”的散点图，如图7-1所示。如果这些点分别代表一个三维的数据样本，那么我们可以发现，这些样本坐落于一个二维流形中，即图7-1中卷筒蛋糕的形状。这里我们用了错误的措辞，因为流形实际上是一个空间而不是一个形状，但为了通俗地解释这件事，我们认为采用这种措辞是可取的。观察式(7-2)，我们可以发现，事实上三维空间中的点(x, y, z)可以只用二维空间的(α, y)描述，因为 x 和 z 都是 α 的函数，从图像上我们也可以直观的感受到，三维数据是多余的，因为数据的坐落位置十分有规律。

降维，也是同样的思想，高维的数据是带有冗余的，如果我们能找到这些数据的低维流形，那么就可以在低维空间中进行数据处理。这衍生出了机器学习的一个流派—流形学习。关于流形学习我们不会做过多的介绍，我们只在本章中介绍两种降维方法，即主成分分析和多维尺度分析。与统计学习类似，实际中我们也无法找到数据真实的低维流形，不同的降维方法相当于对低维流形的不同逼近方法。如果说数据背后确实存在一个低维流形，而我们的降维方法又恰好是这个

²更多有意思的观点可以观看麻省理工学院的《量子力学》公开课

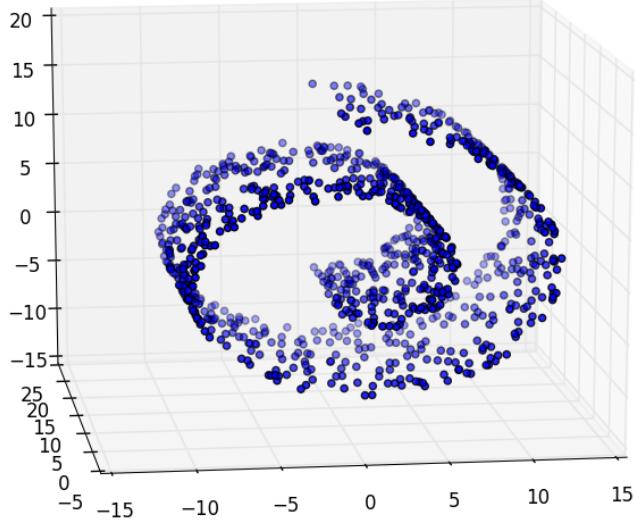


图 7-1 Swiss Roll

流形的描述，那么降维不会引入信息的损失，例如，三维坐标降到球坐标并不会引入误差，但实际上我们并不知道具体流形，我们的降维方法也不是真实的维度变换方法，所以实际中的降维一定会引入信息的损失，导致分类性能的下降，相比于降维带来的巨大好处——运算效率的提高，这个性能的下降是可接受的。图像识别，由于生活图像其维度都以千万记，降维是必须的，否则以目前的计算机性能无法完成准则函数的收敛。

7.1.1.1 主成分分析

为了简单起见，我们只在这里讨论如何使用主成分分析将二维空间中的数据降到一维空间，但二维空间中的结论很容易被推广到更高维度的情况，因为更高维度空间中的结论与二维空间中的结论是相同的。

假设我们有一个含有 n 个2维样本的数据集 $X = [x^{(1)}, x^{(2)}, \dots, x^{(n)}]^T$ ，如图7-2所示是一个含有100个样本的数据集在平面直角坐标系上的分布。

如果我们通过计算得到这些样本点的均值 \bar{x} ，即

$$\bar{x} = \frac{1}{n} \sum_{i=1}^n x^{(i)} \quad (7-3)$$

正如图7-2中的红点，那么我们总能将所有的点用一组基的线性组合来代替，即

$$x^{(i)} = \bar{x} + \alpha_1 e_1 + \alpha_2 e_2 \quad (7-4)$$

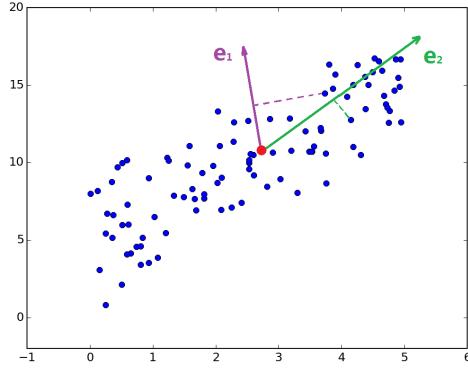


图 7-2 二维空间中的数据样本点

式中， e_1 和 e_2 是任意两个不平行的单位向量，正如图7-2中的紫色方向和绿色方向。但此时我们只是将样本点从一个原始的正交二维坐标系转换到另一个以 e_1 和 e_2 为基的二维坐标系，为了降维，我们必然需要舍弃一个维度，这个维度可能是 e_1 或 e_2 ，然而选取哪个维度更好呢？直觉上我们会认为选取 e_2 会更好，因为当所有的数据点都投影到这个基上时，相比于 e_1 能更好地保留数据的信息。

然而， e_1 和 e_2 是我们任意选取的两个基向量，事实上有可能有另外的基向量要比 e_1 更好的保留原始数据的特征，假设最优的基向量为 e ，那么为了刻画数据降维后原始数据与降维后数据的差异性，我们引入一个准则函数

$$J(\alpha_1, \alpha_2, \dots, \alpha_n) = \sum_{i=1}^n ((\bar{x} + \alpha_i e) - x^{(i)})^T ((\bar{x} + \alpha_i e) - x^{(i)}) \quad (7-5)$$

式中， $\bar{x} + \alpha_i e$ 实质上代表的是原始数据降维到一维空间后在二维空间中的坐标，即原始数据在基向量 e 的投影点的二维坐标。准则函数使用的依然是我们非常熟悉的二范数，这个准则函数事实上刻画的是原始坐标到投影坐标距离的平方，也就是两者究竟相差了多远。为了最小化这个准则函数，我们先对式(7-5)进行展开，这将得到

$$J(\alpha_1, \alpha_2, \dots, \alpha_n, e) = \sum_{i=1}^n \alpha_i^2 \|e\|^2 - 2 \sum_{i=1}^n \alpha_i e^T (x_i - \bar{x}) + \sum_{i=1}^n \|x_i - \bar{x}\|^2 \quad (7-6)$$

由于 $\sum_{i=1}^n \|x_i - \bar{x}\|^2$ 只与数据相关而与参数无关可以舍去，此外， e 是单位向量，所以 $\|e\|^2$ 等于1，因此准则函数可以简写为

$$J(\alpha_1, \alpha_2, \dots, \alpha_n, e) = \sum_{i=1}^n \alpha_i^2 - 2 \sum_{i=1}^n \alpha_i e^T (x_i - \bar{x}) \quad (7-7)$$

为了最小化这个准则函数，我们对其求偏导，并另导数为0，则

$$\frac{\partial}{\partial \alpha_i} = 2\alpha_i - 2e^T(x_i - \bar{x}) = 0 \quad (7-8)$$

因此当 $\alpha_i = e^T(x_i - \hat{x})$ 准则函数取得最小。这实质上说明，当数据点投影到 e 时使得准则函数最小，与我们前面的猜测符合。将(7-8)代回(7-7)，我们便可以消去准则函数中的 n 个变量 $\alpha_1, \dots, \alpha_n$ ，只剩余一个变量 e ，即

$$J(e) = -\sum_{i=1}^n e^T(x_i - \hat{x})(x_i - \hat{x})^T e \quad (7-9)$$

如果我们记号散布矩阵为

$$S = \sum_{i=1}^n (x_i - \hat{x})(x_i - \hat{x})^T \quad (7-10)$$

则准则函数可以写为

$$J(e) = -e^T S e \quad (7-11)$$

此时是一个约束优化问题，约束条件为等式约束 $e^T e = 1$ ，为此引入拉格朗日乘子 λ ，构造拉格朗日函数为

$$\mathcal{L}(e) = -e^T S e + \lambda(e^T e - 1) \quad (7-12)$$

对拉格朗日求偏导，并另偏导为0

$$\frac{\partial}{\partial e} \mathcal{L}(e) = -2S e + 2\lambda e = 0 \quad (7-13)$$

此时说明，当 $S e = \lambda e$ 时，准则函数最小，而事实上这是矩阵对特征根的定义，因此为了最小化准则函数，基向量 e 应当选取为散布矩阵 S 的特征向量。这个结论可以很容易推广到高维情况，假设我们有 n 个 d 维的数据样本，那么我们总能构造他的散布矩阵 S ，为了将它降到 k 维而同时又要使得准则函数最小化，我们应该选取 S 最大的 k 个特征值所对应的特征向量作为投影基，从而新的 k 维坐标为

$$\alpha_i = e^T(x^{(i)} - \hat{x}) \quad (7-14)$$

由于特征向量之间是正交的，因此降维后的数据在各个维度是独立的，这也是白化的目的，因此主成分分析也是一种白化方法。有时候我们需要将降维后的数据还原回原始的维度以检测降维后的数据究竟损失了什么信息，那么利用特征向量的正交性，有

$$e\alpha_i = ee^T(x^{(i)} - \hat{x}) = x^{(i)} - \hat{x} \quad (7-15)$$

此时等号两边同时加上均值 \hat{x} 即可还原出 d 维数据。如图7-3所示是 28×28 像素的图像，它共有784维，其原始数据为黑底的图像所示。如果我们使用主成分分析将其降到20维后再将其还原成784维数据，其图像如灰底图像所示，我们可以看到，经

过降维后再还原的数据已经变得模糊化，而这些模糊化并不会影响我们对数字的识别。

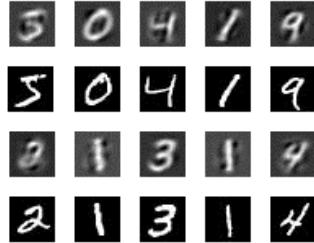
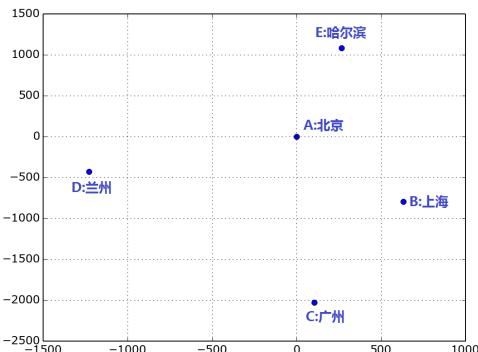


图 7-3 经过主成分分析降维后还原的数据

7.1.1.2 多维尺度分析

在介绍多维尺度分析^[42]之前，我们先从地图重构的问题讲起。假设我们在中国地图中选取五个城市，分别为北京、上海、广州、兰州、哈尔滨，依次用A、B、C、D、E代替。如果我们以北京为坐标平面的原点，那么剩下的城市坐标分别为：上海(631.6, -796.4)、广州(103.8, -2029.2)、兰州(-1228, -428.1)、哈尔滨(265, 1082)，将这些城市的坐标描绘在平面直角坐标系上的结果如图7-4 a)所示。另外，我们可以通过计算得到这五个城市的两两距离如图7-4 b)所示。



a) 城市分布图

| | 北京 | 上海 | 广州 | 兰州 | 哈尔滨 |
|-----|----|--------|--------|--------|--------|
| 北京 | 0 | 1016.4 | 2031.9 | 1300.5 | 1114.1 |
| 上海 | | 0 | 1341.1 | 1895.7 | 1914.0 |
| 广州 | | | 0 | 2082.6 | 3115.6 |
| 兰州 | | | | 0 | 2127.7 |
| 哈尔滨 | | | | | 0 |

b) 城市间两两距离

图 7-4 五个城市的信息

如果现在我们不再知道各个城市的具体坐标，我们只知道图7-4 b)中两两城市之间的距离，那么如何通过图7-4 b)中的信息进行地图的重建，或者说根据距离推导出各个城市的坐标呢？这就是多维尺度分析所要完成的工作。

首先，我们假设这n个城市的坐标构成一个矩阵 $X = [x_1, x_2, \dots, x_n]^T$ ，那么城市*i*和城市*j*之间的欧式距离的平方为

$$s_{ij} = (x_i - x_j)^T(x_i, x_j) \quad (7-16)$$

由 s_{ij} 可以构成一个 $n \times n$ 的矩阵 $S_{n \times n}$, 这个矩阵我们称为距离矩阵, 如果我们再定义Gram矩阵为

$$G_{n \times n} = H X X^T H \quad (7-17)$$

其中, H 为去中心化矩阵, 定义为

$$H = I_n - ee^T \quad (7-18)$$

其中, I_n 为 n 阶单位矩阵, $e = \frac{1}{\sqrt{n}}(1, \dots, 1)^T$, 那么Gram矩阵可以推导为

$$G_{n \times n} = H X X^T H = \left[(x - \bar{x})^T (x - \bar{x}) \right] \quad (7-19)$$

式中, \bar{x} 为 X 的均值, 通过式(7-19), 进一步可以推导得

$$G_{n \times n} = -\frac{1}{2} H S H \quad (7-20)$$

由于欧几里得距离是正定的, 因此Gram矩阵也是正定的, 如果我们将Gram矩阵特征分解, 则有

$$G_{n \times n} = P \Sigma P^T \quad (7-21)$$

式中, Σ 为Gram矩阵特征值构成的对角矩阵

$$\Sigma = \begin{bmatrix} \lambda_1 & & \\ & \ddots & \\ & & \lambda_n \end{bmatrix} \quad (7-22)$$

P 为Gram的特征向量构成的矩阵, 即 $P = [p_1, \dots, p_n]$ 。如果我们提取出Gram矩阵最大的 k 个特征值以及其对应的特征向量, 那么通过

$$\hat{X} = \begin{bmatrix} \sqrt{\lambda_1} & & \\ & \ddots & \\ & & \sqrt{\lambda_k} \end{bmatrix} [p_1, \dots, p_k] \quad (7-23)$$

我们将得到含有 n 个 k 维向量的矩阵 \hat{X} , 这个矩阵 \hat{X} 就是 X 在 k 维空间的一个映射。例如, 在我们地图重构的问题中, 通过图7-4 b)中的距离信息, 当 $k = 2$ 时, 我们求得

$$\hat{X} = \begin{bmatrix} -45.6 & 434 \\ -677.1 & -362.1 \\ -149.3 & -1594.9 \\ 1182.5 & 6.2 \\ -310.5 & 1516.5 \end{bmatrix} \quad (7-24)$$

如果将上述求得的坐标描绘在平面直角坐标系上，其结果如图7-5 a)所示。我们可以看到，这些点与真实分布十分相似，只相差一个水平镜像，我们将其镜像后的结果如图7-5 b)所示。镜像结果显示，重构的地图中，各个城市所在的相对位置与真实地图中城市的相对位置相同，但是不同的是，在重构的地图中，绝对位置改变了，例如我们可以看到，原来处于原点位置的北京现在不在处于原点。

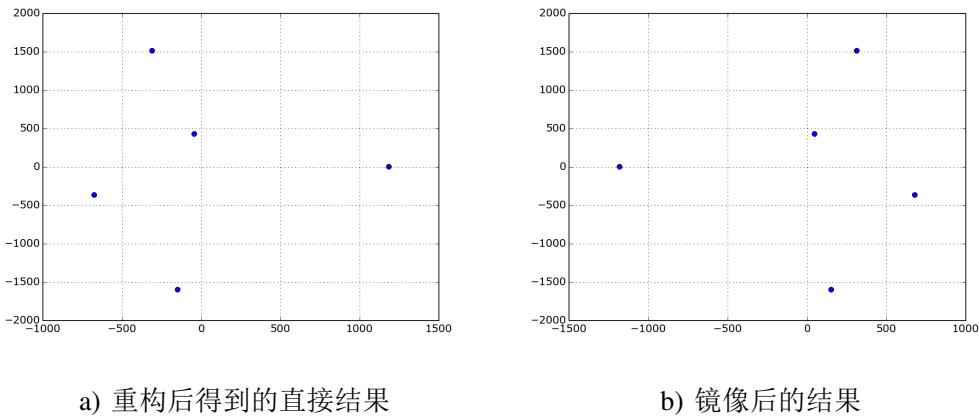


图 7-5 多维尺度分析对地图的重构

地图重构的例子并没有展现出多维尺度分析是如何降维的，因为这里只是从一个二维坐标转换到另一个二维坐标。但由于这个转换的过程中使用的欧几里得距离对于高维空间是可复用的，因此对于多维尺度分析的降维，我们先通过高维空间中的数据点两两求距离，得到平方距离矩阵，在通过这个平方距离矩阵求取Gram矩阵，将Gram矩阵特征分解，选取最大的 k 个特征值以及对应的特征向量，利用他们可以将原始的高维数据降到 k 维数据。例如，图7-6描述了三维空间中的数据降到二维空间的过程。

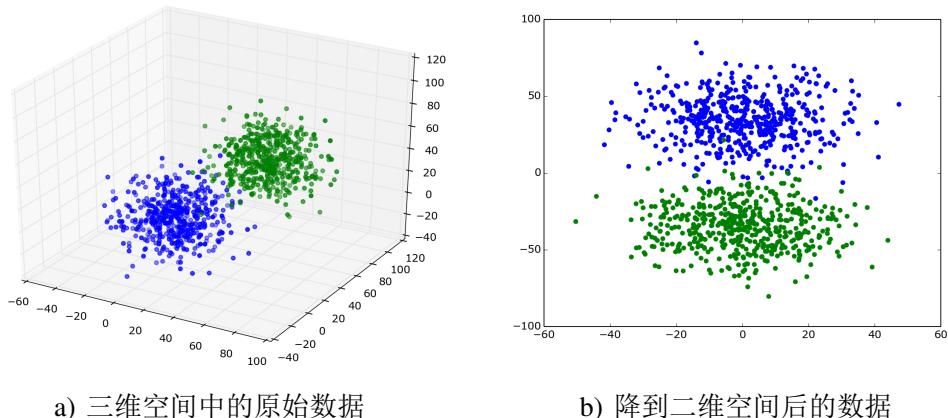


图 7-6 多维尺度分析的降维

多维尺度分析并不是一个应用非常广泛的降维方法，其主要原因在于欧几里得距离，如图7-7三维空间中AB两点，如果使用欧几里得距离（图中的绿线），那么我们认为这两点是十分接近的，因此降到二维空间后这两点将会靠得很近。但事实上AB两点相距很远，它们两者之间经过很长一段流形（图中的红线），而欧几里得距离忽视了这种距离。

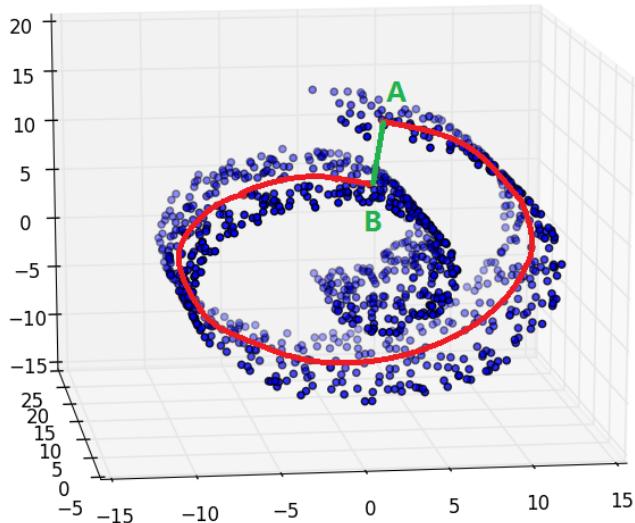


图 7-7 Swiss Roll

尽管多维尺度分析并不常用，但是通过它可以引入流形学习中的一些经典算法，例如Isomap等，用以解决图7-7中类似的情况，更多的讨论可参考文献xxxx

7.1.2 扩容

机器学习业内有一个共识，很多时候别人的分类性能比你的好并不意味着他的算法比你的好，更大的可能是他拥有比你更多的数据。数据是统计机器学习的命脉，没有数据，再好的模型也难以奏效。如果把机器学习中的算法比作是航天器的引擎，那么数据就是航天器的燃料。这种现象在深度学习中尤为明显，特别是卷积神经网络。深度学习需要大量的数据样本，这个需求量随着识别任务复杂性的增加而增加。尽管互联网的大数据时代数据容易获取，但有些时候数据是有限的，对于有限的数据集，我们可以通过一定的手段将数据集的容量扩大。例如在字符识别任务中，我们知道，字符微量的扭曲、旋转、放大、缩小对人而言是

不会影响字符的识别，因此，我们可以对原始数据进行随机地选取上述一个或多个变换，使得数据的容量增大^[43]。又例如，现实生活中的图片，水平镜像、对比度微调、饱和度微调等操作对人而言不影响最终的图像识别，我们利用这个特性对图像采取一定的变换便可很大程度地增大训练集的容量，例如，仅仅是图像的水平镜像就可以使得数据集扩大一倍。

数据的扩容，实质上是一种贝叶斯先验，因为我们对数据采取的变换策略不会影响最终的分类，例如，我们不会对字符进行镜像处理。这种先验，使得模型可以免疫一定的干扰，例如采取了水平位移变换策略的数据集受到水平位移的影响就减小了，这就提高了模型的泛化能力。

7.2 训练技巧

网络的训练过程中，我们会使用到一些技巧，这些技巧大部分的目的都在于抑制网络的过拟合以及防止准则函数陷入局部最优解。由于这些技巧众多，我们无法在这里涵盖所有，因此我们只挑选了几个常用且重要的技巧，分别是学习率的设定、动量项的使用以及权衰减。

7.2.1 学习率

在梯度下降算法中，每次求得准则函数相对于参数的梯度 $\partial J / \partial \theta$ 后，我们并不是直接以这个值作为参数的增量，工程中，我们还应对梯度乘上一个学习率 η ，即参数更新的公式为

$$\theta = \theta - \eta \frac{\partial J}{\partial \theta} \quad (7-25)$$

学习率的作用类似于下山时步伐的长度，步伐过长，则有可能导致准则函数震荡，影响收敛速度，如果步伐十分大，则会导致发散现象，哪怕准则函数是凸函数，存在唯一极值点，过大的学习率也会使其发散。例如，假设准则函数设定为二次函数，即

$$J(\theta) = \theta^2 \quad (7-26)$$

显然这是一个凸函数，存在唯一极值点，即 $\theta = 0$ 处。如果我们对这个准则函数求偏导，则为

$$\frac{\partial J}{\partial \theta} = 2\theta \quad (7-27)$$

即准则函数 $J(\theta)$ 上的每一点 θ ，其梯度都等于 2θ 。假设现在 $\theta = 2$ ，那么我们可以很容易计算得到其梯度为4，此时，最优的学习率为 $\eta^* = 0.5$ ，因为 $2 - 0.5 \times 4$ 恰

好使得 θ 落入到最优解即 $\theta = 0$ 处，如图7-8 a)所示。如果我们的学习率选取得比最优学习率大，假设我们选取为 $\eta = 0.7$ ，那么参数 θ 就会沿着 $2 \rightarrow -0.8 \rightarrow 0.32 \rightarrow -0.128 \dots$ 的轨迹下降，如图7-8 b)所示。倘若我们的学习率再大一些，为两倍的 η 时，即当 $\eta = 1$ 时， θ 就会在+2和-2两者之间变换而不会进入别的位置，如图7-8 c)所示。当学习率大于两倍的最优学习率，例如 $\eta = 1.5$ ，那么 θ 就会沿着 $2 \rightarrow -4 \rightarrow 8 \rightarrow -16$ 的轨迹移动，此时准则函数便发散，这个过程如图7-8 d)所示。

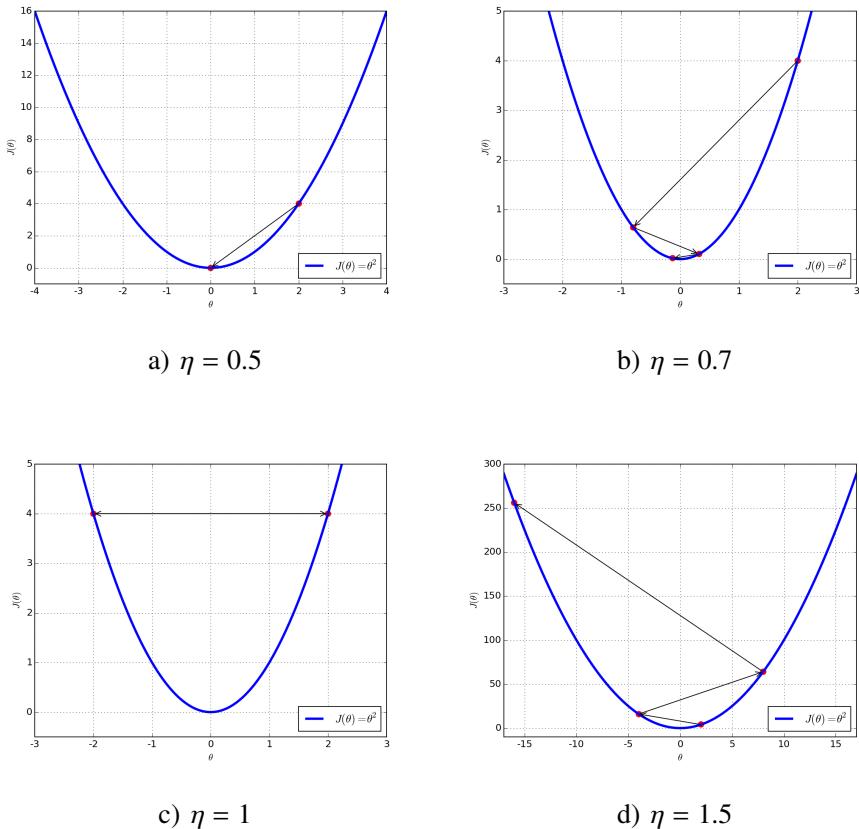


图 7-8 不同的学习率对收敛的影响

但是较大的学习率并不意味着一无是处，学习率稍微大一些，可以加快收敛速度，而且可以越过一些局部极小值（但是对于高尔夫球场形状的局部极小值无能为力）。在工程中，学习率的选取我们的建议是，先将学习率初始化为一个较大的数值，如果准则函数发散了，那么我们就将其缩小2倍，直到准则函数能开始下降为止。另外一种做法是，先初步计算梯度的数值大小，将学习率设定为一个比梯度小3个数量级的常数，例如，我们上面的计算得到的梯度是4，那么学习率可以设定为0.001。

事实上学习率固定为一个常数并不是一个较好的策略，因为初始时设定的学习率相对于训练的后期而言是过大的，这将会导致准则函数在极小值附近震荡。一种解决方法是将学习率也设定为一个学习参数，这个参数随着网络的训练而不断地更新，另外一种虽然愚蠢但行之有效的方法是，不断地关注准则函数的下降情况，一旦感觉下降开始出现震荡或平谷现象就将学习率调小1个数量级。

7.2.2 动量项

很多时候，在参数校正的过程中，仅仅使用学习率是不行的，一般还应使用动量项。动量项，即在参数的更新过程中，引入一个动量因子，更新规则描述为

$$\theta_{t+1} = \theta_t - \eta(1-p) \frac{\partial J(\theta)}{\partial \theta_t} - p * (\theta_{t-1} - \theta_{t-2}) \quad (7-28)$$

式中， t 代表第 t 次参数更新， η 为学习率， p 为动量项因子，这个因子一般选取为 $p = 0.9$ 。式(7-28)实际上讲述了这样一件事：参数更新的时候，增量不仅仅依赖于当前的梯度，还依赖于上一次权值变化量的 p 倍，因此参数的更新，带着类似于物理中动量的项，即 $p * (\theta_{t-1} - \theta_{t-2})$ ，它也被称之为动量项。动量项可以使得准则函数快速地下降，并且可以越过一些较为平坦的局部极值。如果读者有数字信号处理的背景，式(7-28)实际上是一个脉冲响应低通滤波器，目的在于平滑参数的更新过程^[3]。

但由于(7-28)需要记录上一次的参数增量，这在实际编码中需要一定的内存，而且实现起来也不方便，另外一种关于动量项的简单用法是

$$\theta_{t+1} = p\theta_t - \eta \frac{\partial J(\theta)}{\partial \theta_t} \quad (7-29)$$

式中， p 为动量项系数，一般取0.9。动量项的使用，确实可以在一定程度上避免局部最优解，例如，假设准则函数为

$$J(\theta) = -5\theta^3 - 80\theta^2 - 400\theta - 700 \quad (7-30)$$

我们可以很容易地计算准则函数在任意一点的梯度为

$$\frac{\partial J(\theta)}{\partial \theta} = -15\theta^2 - 160\theta - 400 \quad (7-31)$$

假设学习率设置为常数0.001，如果不采用动量项，即参数更新的规律为

$$\theta = \theta - 0.001 \frac{\partial J(\theta)}{\partial \theta} \quad (7-32)$$

那么准则函数的收敛过程如图7-9 a)所示，我们可以看到，在收敛的后半阶段，参数 θ 会收敛到局部极值，但如果我们使用一个系数为0.9的动量项，即参数更新规则为

$$\theta = 0.9 * \theta - 0.001 \frac{\partial J(\theta)}{\partial \theta} \quad (7-33)$$

此时的准则函数收敛过程如图7-9 b)所示，我们可以看到，动量项的引入，使得参数 θ 在平坦的局部极值区域内依然带有冲劲，这个性质使得它可以跨越一些较小的局部极值而不至于陷入其中。

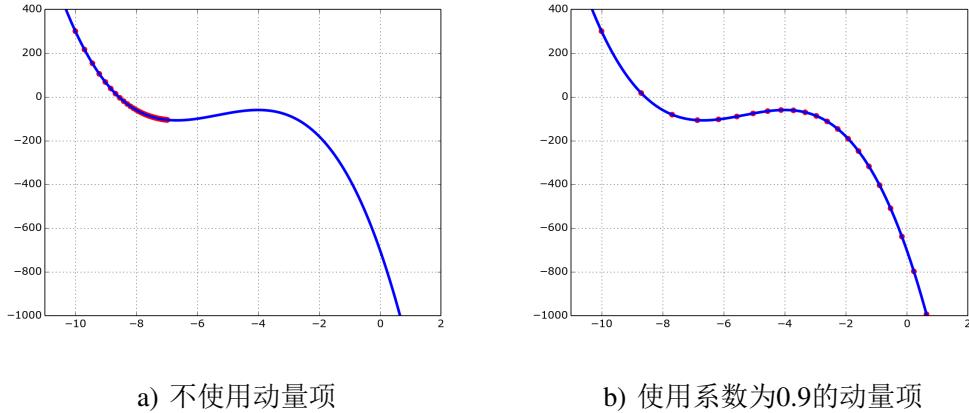


图 7-9 动量项对收敛的影响

7.2.3 权衰减

谓权衰减，即在权值更新完毕后对所有的权值乘以一个小于1的常数 e ， e 一般取0.999左右。之所以要这样做的原因，是因为我们希望权值保持在较小的范围内，因为更小的权值使得净激活停留在激活函数的非饱和区域内。使用权衰减的参数更新规则为

$$\theta = e * \theta \quad (7-34)$$

关于权衰减的另一种解释是，在准则函数中，我们加入一个二范数惩罚，即新的准则函数定义为

$$J_{new}(\theta) = J(\theta) + \frac{\lambda}{2} \theta^T \theta \quad (7-35)$$

(7-35)中的第二项也被称为正则项。如果我们对式(7-35)求导，则为

$$\frac{\partial J_{new}(\theta)}{\partial \theta} = \frac{\partial J(\theta)}{\partial \theta} + \lambda \theta \quad (7-36)$$

因此权值的更新规则为

$$\theta = \theta - \frac{\partial J_{new}(\theta)}{\partial \theta} = \left(\theta - \frac{\partial J(\theta)}{\partial \theta} \right) - \lambda \theta \quad (7-37)$$

这事实上等价于是式7-34，即在权值更新后乘上一个衰减系数。我们之所以使用权衰减，或者说加入二范数正则（这两个命题等价），实际上它代表着一种贝

叶斯先验，即我们对于权值的期待（希望它是较小的值）。

需要注意的是，在神经网络中，我们一般只针对权值进行权衰减，而不针对偏置进行权衰减，因为我们希望获得较大的偏置。至于为什么，这实际上并没有为什么。

7.3 编码技巧

尽管我们讨论了大量的理论内容，这些理论内容具体实现在代码上并不是一件简单的事。代码如何设计以及如何测试是我们遇到的最大问题。与传统的软件工程所不同的是，机器学习中的测试无法使用单元测试，因为在单元测试中我们需要知道代码的期望输出，机器学习自学习的特点注定了我们无法知道其正确输出是什么。代码的测试，更多时候是凭感觉，尽管如此，我们还是有办法进行一些简单测试的，在本小节中，我们将会介绍面向对象的设计思路以及用于测试的梯度校验。

7.3.1 面向对象技术

神经网络的设计中，不同层可以采用不同的激活函数，也可以采用不同的网络结构，例如卷积层或全连接层。这些不同层之间都有两个共同的特性。在前向传播过程中，无论是什么形式的层都接收前一层传播的激活，而在反向传播过程中，所有的层都接收后一层传播回来的误差，并利用该层的输入与输出对参数进行校正。尽管不同结构的网络层其校正方式不同，但这个过程可以使用面向对象的技术实现。为了方便讨论，我们将用JAVA的语法来描述这个过程³。我们定义一个接口Layer，它含有两个方法，即fprop和bprop，分别对应着前向传播和反向传播。fprop接收一个参数，即前向传播得到的输入数据，bprop接收三个参数，分别是反向传播回来的误差、该层的输入以及该层的输出，事实上，bprop也可以设定为只接收一个参数，即反向传播回来的误差，而该层的输入以及该层的输出设定为成员变量。具体的网络结构，例如ConvolutionLayer、FullConnectLayer等实现这个接口，并实现这个接口中的两个方法，利用多态性，就可以容易地实现不同层执行不同的更新方法。如果网络的所有层都采用同一个激活函数，那么这个激活函数可以直接写到接口中（尽管JAVA要JDK9中才能在接口中提供方法的实现，但我们可以利用继承让具体的层继承另一个类）。如果网络的所有层采用的是不同的激活函数，那么我们可以通过策略模式将激活函数抽象成一个接口，具体的激活

³然而JAVA并不是一门适合数值运算的语言

函数实现这个接口，在具体网络层实现类的构造方法中通过依赖注入将具体的激活函数实现注入到网络层的类中，利用多态性，可以让不同的层执行不同的激活函数，而这些不同的激活函数又可以实现代码复用。最后将这些不同的网络层组合到一个神经网络实现类中，例如CNN或DBN，即可完成整个网络的编码工作，整个工程代码的UML图如图7-10所示。

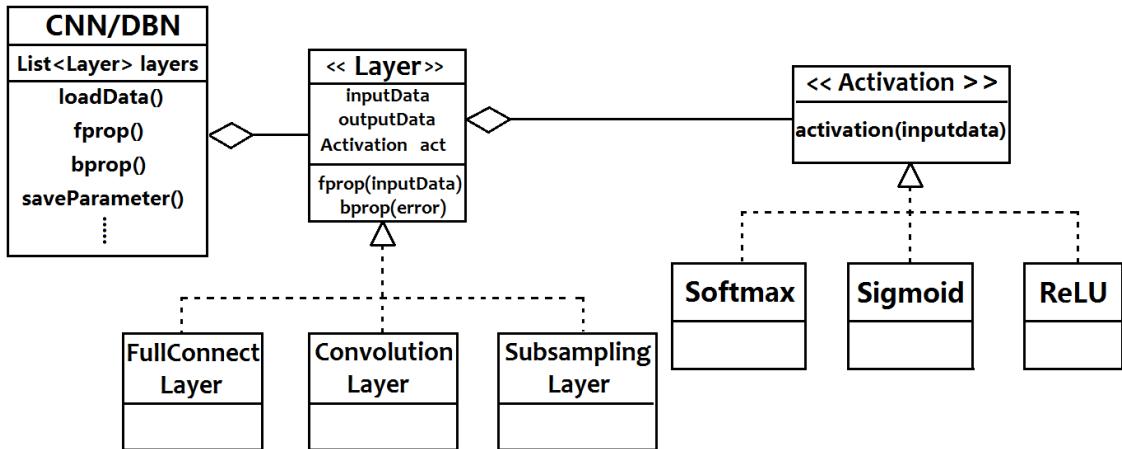


图 7-10 神经网络类图设计

7.3.2 梯度校验

在神经网络的编码中，一个令人头疼的问题是反向传播是否已经被正确编码。很多时候，错误的编码并不会导致严重的错误，但它会影响最终的分类效果。在debug过程中，一个十分有效的用来检测编码是否正确的方法是梯度校验。假设我们要优化以 θ 为参数的准则函数，在梯度下降中，我们的更新规则如式(7-25)所示。假设我们通过算法计算得到准则函数相对于参数的梯度，例如，在全连接神经网路中，这个梯度一般为

$$\frac{\partial J(\theta)}{\partial \theta} = \delta \cdot \frac{\partial f(net)}{\partial net} \frac{\partial net}{\partial \theta} \quad (7-38)$$

式(7-38)本质上是利用导数公式取计算梯度，正如我们要算 $J(\theta) = \theta^2$ 的导数，那么我们可以通过其导数即 $J' = 2\theta$ 直接计算在某点处的导数，但我们还可以通过另一种方式计算导数。回想高等数学中关于导数最原始的定义，导数即增量的极限，定义为

$$\frac{\partial J(\theta)}{\partial \theta} = \lim_{\Delta\theta \rightarrow 0} \frac{J(\theta + \Delta\theta) - J(\theta)}{\Delta\theta} \quad (7-39)$$

也就是说，如果我们给某个参数 θ_i 加上一个非常小的增量 $\Delta\theta$ ，假设为0.001，那么利用式(7-39)的原理，我们对比 θ_i 没有加上这个增量之前准则 $J(\theta)$ 的值与加上增量

之后准则的值，这两个值作差，再除以增量，将会得到准则函数相对于 θ_i 这个变量的梯度。这种方法求得的梯度 $\Delta_{\theta}^*J(\theta)$ 与利用式(7-38) $\Delta_{\theta}J(\theta)$ 满足如下关系

$$\Delta_{\theta}^*J(\theta) \approx \Delta_{\theta}J(\theta) \quad (7-40)$$

如果 $\Delta\theta$ 取得足够小，那么这两者近似程度非常高，但是实际编码中 $\Delta\theta$ 不应取得过小，因为计算机的浮点运算是会带来误差，一般而言， $\Delta\theta$ 取0.001左右就足以用来验证梯度了。另外需要注意的是，这种近似性只会在较深的层中发生，在较浅的层中误差会较大，因为反向传播是贪婪的，所以随着误差越传越远，梯度的偏离也会越大。利用梯度校验在可以方便检验代码正确性的同时，还可以在梯度校验的过程总发现，如果激活函数是Sigmoid函数，而网络有是深度网络，那么不经过预训练的网络会出现梯度消失现象。

7.4 本章小结

本章中我们讨论了神经网络训练过程中一些会使用到的技巧，我们只介绍了一些通用的技巧，更多的特殊技巧如dropout、maxout等我们并没有讨论，更多的讨论可参考文献[44]和[45]。本章介绍的技巧中，降维自身就是一个很大的话题，并不仅仅作为技巧而存在，它是机器学习的一个分支，我们只介绍了两种降维方式，更多的降维方法如Isomap、局部线性嵌入、基于核的主成分分析等可参考文献[46–48]。最后还需要说明一点，网络的设计者不应迷信这些技巧，它们只是起到指导作用，具体是否可行要具体任务具体分析。

第8章 GPU计算

上世纪六十年代提出的摩尔定律到目前为止已经持续了半个多世纪，硅技术似乎已经难以再有重大突破，摩尔定律在未来几年或将成为过去。然而当前计算机的计算能力仍不足以满足人们的需求，并行技术的发展在一定程度上缓解了这种压力，某种程度上而言，计算并行化可以看做是摩尔定律的一种延伸。

GPU（Graphic Processing Unit）又称“图形处理器”，是相对于CPU的一个概念，其最初目的是满足计算机图形界面中人们对于实时、高清的三维图形的需求。在图形开发中，由于图形渲染及变换等操作是带有可并行特征的，因此对于GPU的设计者而言，他们更倾向于添加更多的核心而不是提高核心的运行效率，通过多个核心以及线程技术，使得同一条指令可在每个元素上执行。此外，他们还取消了CPU中的缓存与逻辑控制部件，因此GPU适用于计算密集的并行计算，而CPU更适用于含有复杂流控的计算。

但是最初的GPU是为图形开发者而开发的平台，对于非图形开发者而言，使用GPU进行通用计算并不是一件容易的事。为此，ATI公司（已被AMD收购）于2005年提出的ATI Stream方案，NVIDIA公司在2007年推出的CUDA平台，都是为了解决GPU通用计算问题而做出的努力。在GPU通用计算不断发展的同时，新的规范也被不断地提出，Apple公司在2008年提出OpenCL（Open Computing Language）规范，微软提出的DirectCompute规范，其目的都在于制定了一套API，并在此基础上开发GPU通用计算软件。

在这些GPGPU（General Purpose GPU）方案中，由于CUDA为开发者提供了一个类似于C的编程环境，使用者可以快速学习，所以受到了广泛的使用。

8.1 GPU体系结构

相比于CPU而言，GPU由于其设计原则就是针对密集运算的，所以在浮点运算上占着先天优势。近几年CPU在运算效率上的提升缓慢，与此同时，GPU计算却飞速发展，其运算性能已远远超过CPU。如图所示是Intel的CPU与NVIDIA的GPU在各个型号上每秒浮点运算量的对比。相比于CPU而言，GPU由于其设计原则就是针对密集运算的，所以在浮点运算上占着先天优势。近几年CPU在运算效率上的提升缓慢，与此同时，GPU计算却飞速发展，其运算性能已远远超过CPU。如图8-1所示是Intel的CPU与NVIDIA的GPU在各

个型号上每秒浮点运算量的对比¹。

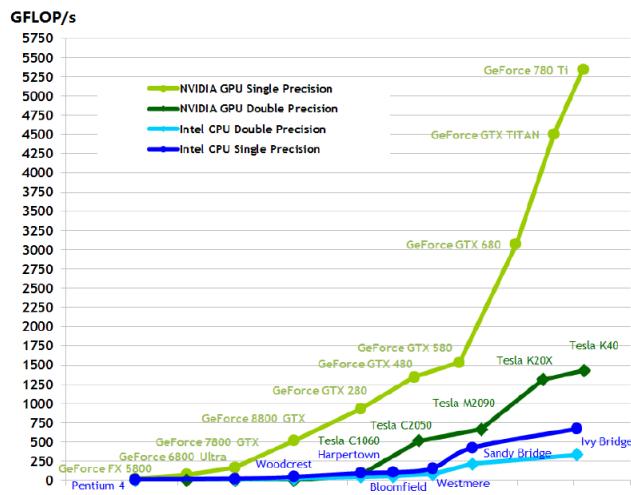


图 8-1 GPU 与 CPU 在每秒浮点运算量上的差异

由于GPU中取消了逻辑控制单元，相比于CPU其体系结构有着很大的不同。典型的CPU结构带有三级缓存，数据由内存经过三层缓存后才进入到处理器核心，这个结构如图所示。GPU同样带有缓存，这个缓存一般只有两级，与CPU不同的是，CPU中只有一个处理器，而GPU中并没有处理器核心的概念，取而代之的是流处理器簇（Streaming Multiprocessor），在一个GPU中往往含有多个流处理器簇，尽管它们与CPU处理器核心有着很大的差异，但某种程度上也可以看做是GPU的核心，整个GPU的结构如图所示。

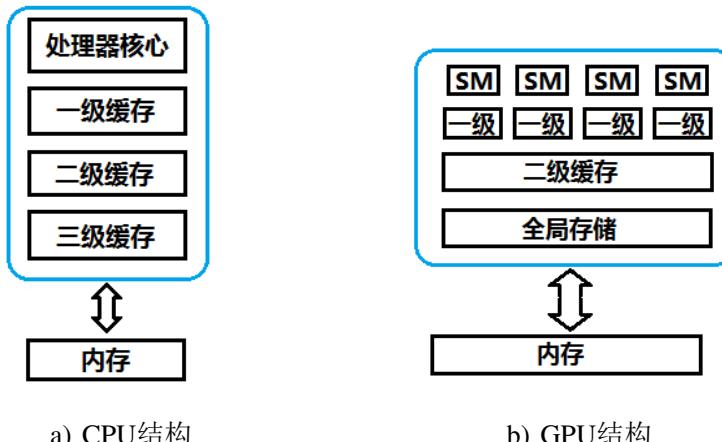


图 8-2 CPU 与 GPU 结构差异

CPU为了实现多核技术，往往在一个芯片中集成多个CPU核心，例如在Intel

¹ 图片出自于NVIDIA的官方技术文档《CUDA C Programming Guide》

i7处理器中含有4个CPU核心，多核心使得CPU可以并行执行计算，但相比于GPU而言，CPU的并行粒度是非常大的。例如要将一个含有100个元素的向量的每一个元素乘上一个常数2，那么对于4核的CPU，我们可以指派1号CPU执行第1~25个元素的操作，2号CPU执行第26~50个元素的操作，但对于GPU而言，它可以直接开辟100个线程，每个线程针对每个元素进行操作，这些线程是并行线程而不是并发线程，因此GPU可以实现小粒度的并行。

GPU之所以能开辟大规模线程是因为一个GPU包含了一个流处理器簇阵列，这个阵列由多个流处理器簇构成，例如费米架构的GPU的阵列大小为16。而每个流处理器簇又包含了几十个到几百个的CUDA核心，例如，采用开普勒架构的GPU每个流处理器簇包含48个CUDA核心，而采用费米架构的GPU每个流处理器簇包含192个CUDA核心。这些核心并不类似于CPU的核心，CPU的每个核心只能执行一个线程，而CUDA核心可以并行执行多个线程。

GPU的线程由线程网格管理，线程网格可以看做是一个二维网格，这个结构如图8-3所示。在这个网格中，一个维度是线程块，另一个是线程束。每个GPU核心最多可包含65536个线程块，每个线程块最多可包含512个线程束，这意味着我们可以一次开辟最多大约3300万个线程，目前的费米架构已经实现每个线程块可包含1024个线程束，因此它最多可开辟6600万个线程。但这只是说我们可以在代码中开辟线程，并不意味着执行线程也是这个数量级。事实上，每个流处理器簇可执行的线程数量是有限制的，这个数量级大约是一千左右。

| | | 线程1 | 线程 2 | ... | 线程 n |
|------|--|-----|------|-----|------|
| | | 线程1 | 线程 2 | ... | 线程 n |
| 线程块1 | | ⋮ | ⋮ | ⋮ | ⋮ |
| 线程块2 | | ⋮ | ⋮ | ⋮ | ⋮ |
| ⋮ | | ⋮ | ⋮ | ⋮ | ⋮ |
| 线程块m | | ⋮ | ⋮ | ⋮ | ⋮ |
| | | 线程1 | 线程 2 | ... | 线程 n |

图 8-3 线程网格

实际中在跑的线程数大约为几万，然而我们却可以在代码中开辟几千万的线程，这得益于GPU对硬件的隐藏机制，类似于操作系统的虚拟地址，GPU会自动地将这几千万的虚拟线程在几万实际线程中调度，而程序员不需要参与到这个过

程中，只需要在代码中开辟线程即可。这个机制使得硬件升级扩展变得简单，因为虚拟的线程与实际调度的线程是分离的，GPU为我们自动调度，我们可以随意的更换未来更多核心的GPU而不需要改动我们的代码。

为了更好地理解线程块与线程的概念，我们举一个GPU中实现两个向量相加的例子，如代码8.1所示

代码 8.1 向量加法内核函数

```

1  __global__ void add(int *c, int *a, int *b){
2      int tid = blockIdx.x;
3      c[tid] = a[tid] + b[tid];
4 }
```

代码8.1中的`__kernel__`代表这是一个在GPU中执行的代码，我们称其为内核函数。由`__kernel__`标记的代码将由nvcc编译器编译，而没有这个标记的函数将由C++的编译器编译²。第2行代表获取当前的线程号，这里我们假定线程号就是线程块的编号，即默认每个线程块只有一个线程在运行。如果每个线程块有多个线程在执行，那么第2行代码应改为

```
int tid = threadIdx.x + blockIdx.x + blockDim.x;
```

观察图8-3，这实质上类似于二维索引空间转换为线性空间的代码。代码8.1中的第3行代表每个线程执行一个操作，即将向量a中的第tid个元素与向量b中的第tid个元素相加，并存储在向量c的第tid个元素中。这可以理解为，代码8.1会做为一个副本分发给各个线程，各个线程拿到这段代码后，根据自己的线程ID（即tid）执行向量a和向量b中的第tid的元素的操作。

8.2 CUDA

CUDA（Compute Unified Device Architecture）是NVIDIA公司在2007年推出的高性能运算平台，它可以让GPU在执行常规的图形渲染的基础上额外地实现高性能的通用并行计算。CUDA包含了CUDA指令集架构以及GPU内部的并行计算引擎，通过利用GPU的处理能力，可大幅提升计算性能。支持CUDA平台的GPU包括NVIDIA Tesla、NVIDIA Quadro以及NVIDIA GeForce多个系列，其价格也涵盖了高端到底端多个市场，使得CUDA能够满足从消费级到专业级等多个方面的

²例如Windows操作系统下的msvc或Linux操作系统下的gcc

需求。目前为止，CUDA已经应用到图像与视频处理、计算生物学、流体力学模拟、金融风险分析、地震分析等多个领域，全球五百强企业以安装700多个基于CUDA的GPU集群，这些公司包括了能源领域的斯伦贝谢与雪佛龙以及银行业的法国巴黎银行等。

在CUDA中，程序员需要手工将数据从内存中迁移到GPU中，还需要负责GPU内存的回收，例如，为了执行两个向量的加法，其片段如代码8.2所示

代码 8.2 GPU中向量加法的调用过程

```

1 int a[N], b[N], c[N];
2 int *dev_a, *dev_b, *dev_c;
3 // init a[N], b[N].....
4 cudaMalloc((void**)&dev_a, N*sizeof(int));
5 cudaMalloc((void**)&dev_b, N*sizeof(int));
6 cudaMalloc((void**)&dev_c, N*sizeof(int));
7 cudaMemcpy(dev_a, a, N*sizeof(int), cudaMemcpyHostToDevice);
8 cudaMemcpy(dev_b, b, N*sizeof(int), cudaMemcpyHostToDevice);
9 add<<<N, 1>>>(dev_c, dev_a, dev_b);
10 cudaMemcpy(c, dev_c, N*sizeof(int), cudaMemcpyDeviceToHost);
11 cudaFree(dev_a);
12 cudaFree(dev_b);
13 cudaFree(dev_c);

```

在代码8.2中，我们先在GPU中申请内存，即对应的4~6行，随后在7~8行中将内存中的数据迁移到之前申请的GPU内存中。第9行执行代码8.1实现的加法内核函数，在GPU中开辟N个线程块，每个线程块含有1个线程，GPU完成计算后，我们在第10行将计算结果从GPU中迁移回内存，最后第11~13行释放之前申请的内存。释放内存是重要的，如果申请了内存不释放便会导致内存泄露，当内存消耗完毕后程序崩溃。

我们不由感叹，为了执行一个简单的向量加法就要编写如此多的代码，我们需要手工申请内存，手工将数据迁移到GPU中，手工调用内核函数，调用完毕后需要手工将计算结果从GPU中迁移回来，最后还需要手工释放申请的内存。如果用户不是专业的程序员，那么代码编写的过程十分困难。因此，在CUDA之上有多个团队为机器学习研究人员开发了专门的第三方库，我们将会介绍几个常见的包。

8.3 Cudamat

Cudamat是一个由多伦多大学开发的一套针对于Python的开源第三方库^[49]，它基于CUDA，在实现GPU高性能计算的同时保留了Python“语法优雅”的特性，所以使用者可以很方便地在Python语法层次上调用矩阵运算库。开发Cudamat的目的是为了方便机器学习建模，使科研人员从繁琐的CUDA编程中解放出来，由于GPU在浮点并行运算上巨大的优势，所以在计算密集的矩阵运算任务中Cudamat相对于numpy或MATLAB，其运算速度大约提升了50倍左右。

在Cudamat中，开发者已为我们重载了一些运算符，这使得矩阵的四则运算运算以及面向元素的四则运算，以及矩阵的切片、转置等操作不再需要编写大量的代码或调用对应的函数。此外，开发者还提供了一些常用的函数，例如面向元素的exp()、log()、sqrt()、pow()，矩阵的乘法以及面向轴的求和、随机矩阵的生成等。例如，为了实现一个logistic函数，只需要在Python中只书写代码8.3中的语句

代码 8.3 Cudamat中logistic的实现

```
def logistic(A):
    expTerm = cudamat.CUDAMatrix(numpy.random.randn(A.shape))
    cm.exp(-A, target=expTerm)
    return 1 / (1 + expTerm)
```

尽管Cudamat并没有将CUDA的所有功能都囊括其中，但在机器学习中常用的功能基本都实现了，所有我们可以在不需要了解底层CUDA的前提下高效地建立数学模型的代码，减轻了我们的工作量。

8.4 Gnumpy

在Python科学计算中，numpy与scipy两个第三方包充当着重要角色，其中numpy凭借其优美的语法实现以及高效地执行速率深受人们喜爱。在numpy中，有着许多巧妙的设计，比如广播特性、切片、方便的元素存取、丰富的函数等。其语法特性与MATLAB接近，几乎MATLAB上能实现的功能在numpy中都能找到对应的实现，此外还加入一些MATLAB所不支持的功能，例如方便地嵌入C++代码、方便地存储图像、网络IO等功能。尽管numpy语法简单，运行效率接近于C，但由于其本质上还是基于CPU的运算，所以在大规模的计算中运行效率显得有些难尽人意。

在Cudamat上编写代码要比直接地使用CUDA编写代码要方便得多，我们不再需要了解底层，不再需要担心GPU的内存释放，不再需要编写复杂的内核函数，然而，Cudamat代码要比numpy代码逊色不少，例如代码8.3中的exp函数需要在参数列表中带上返回变量，而这个返回变量又必须要先声明，无法实现变量的动态使用，所以Cudamat带着明显的C语言风格，这违背了Python关于“简单就是美”的设计原则。鉴于以上原因，多伦多大学在cudamat的基础上开发了新一代的开源第三方包—Gnumpy^[50]。Gnumpy其计算本质是GPU计算，但其接口特性接近于numpy。尽管Gnumpy是基于Cudamat的，但在Gnumpy中你将看不到Cudamat的影子，开发者已经将其隐藏了，你看到的只有类似于numpy那样便捷的接口。例如，同样是实现logistic函数，在Gnumpy中只需写成代码8.4中的形式

代码 8.4 Gnumpy中logistic的实现

```
def logistic(A):
    return 1 / (1 + numpy.exp(-A))
```

事实上，numpy已经为我们实现了logistic函数，因此我们只需要一条语句即可完成该功能

A. logistic()

使用Gnumpy更容易编写程序，代码相对于Cudamat而言更简短，也更容易阅读与调试，对于拥有numpy使用经验的开发者而言可以很容易地上手。尽管Gnumpy是基于Cudamat的基础上进行的二次开发，但其运行效率接近于Cudamat，因此使用者无需过于担心效率问题。

8.5 PyCUDA

无论是Gnumpy或是Cudamat都只提供了一些矩阵的基本操作，我们无法实现一些这两个库没有的矩阵操作，例如二维离散卷积。如果我们因为内存管理的原因不想直接写繁琐的CUDA C代码，那么PyCUDA是一个可以选择的库。PyCUDA是由Andreas Klockner与Nicolas Pinto等人开发的一个Python第三方库^[51]，其目的在于允许我们在Python中内嵌CUDA C代码。在PyCUDA中，我们可以只书写内核函数，而不写内存管理的代码。在程序第一次执行时，Python会调用CUDA的编译器将CUDA C的代码编译成动态链接文件，编译完成后可以直接在Python中调用这个函数。例如，为了实现一个矩阵相乘的函数，我们的代码如代码8.5所示

代码8.5 Pycuda中实现向量相加

```

1 import pycuda.driver as GPU
2 from pycuda.compiler import SourceModule
3 def add(A, B, N):
4     code = SourceModule('''
5         __global__ void add(int *c, int *a, int *b){
6             int tid = blockIdx.x;
7             c[tid] = a[tid] + b[tid];
8         }
9     ''')
10    addByGPU = code.get_function("add")
11    C = numpy.zeros_like(A)
12    addByGPU(GPU.Out(C), GPU.In(A), GPU.In(B),
13              block=(N, 1, 1), grid=(1, 1))
14    return C

```

代码8.5的第4~9行通过字符串的形式嵌入CUDA C的内核函数，第10行试图去获取名为“add”的内核函数，如果这行语句是第一次执行，并且在我们已经通过字符串形式嵌入了“add”这个函数，那么Python就会调用nvcc编译器对代码进行编译，并保存为动态链接文件，当下一次再试图获取这个内核函数时不需要再编译一次而直接调用。第12行执行GPU中的计算，以A、B作为输入，C作为输出，执行在N个线程块上。最后返回运算结果C。

PyCUDA使得我们的代码自由度变得更高，我们可以实现任何一个我们想要的内核函数。但与此同时它也使得我们的代码变得复杂。天下没有免费的午餐，选择一个自由度高的库或是选择一个代码简单的库完全取决于你的权衡。

8.6 Caffe

以上的几个Python第三方库都是一些机器学习中的通用库，这些库只提供基础的功能，利用他们可以实现很多算法。如果一个科研人员没有太多程序设计的经验，而又希望将他设计的神经网络在计算机上实现，那么Caffe是一个选择。Caffe是由加州大学伯克利分校的Jia Yangqing等人的领导下开发的一套基于CUDA的深度学习工具箱^[52]，其源代码由C++/CUDA实现，在此之上提供了Python、MATLAB接口，并且可以通过命令行执行。在Caffe中，开发者已经编

写好各种各样的网络层的代码，如全连接层、卷积层等。使用者无需编写具体的代码，Caffe让深度学习的研究人员从具体的代码中解放出来，我们只需要将自己设计的网络模型写成配置文件即可。如代码8.6是一个神经网络的配置文件中的一个片段

代码 8.6 神经网络配置文件片段

```

1 layers {
2     name: "pool1"
3     type: POOLING
4     bottom: "conv1"
5     top: "pool1"
6     pooling_param {
7         pool: MAX
8         kernel_size: 3
9         stride: 2
10    }
11 }
```

在代码8.6的配置文件片段中，定义了一个池化层。配置文件由两部分组成，2~5行为属性定义，6~10行为参数定义。其中第2行定义了这层的名字，第3行定义了层的类型，第4行定义了这层网络的前一层网络的名字，第5行定义了后一层网络的名字（这里就是它自身）。第7行定义了这层池化层使用的是最大池采样，第8行定义了卷积核的尺寸为 3×3 ，第9行定义了卷积间隔为2。

8.7 本章小结

本章中，我们介绍了GPU计算，由于本文并不是一个GPU编程指南，因此我们仅限于简单地介绍，更多关于CUDA C的编程指南可参考文献[53]和[54]。在本章的后半阶段，我们介绍了几个基于CUDA开发的库，关于这几个库如何选取，如果需要编写的代码只含有简单的矩阵操作，那么我们推荐使用Gnumpy库，如果代码中含有一些很复杂的矩阵操作，而这些操作又是可并行的，那么我们推荐使用PyCUDA。这两个库除了可以用在神经网络中外还可以用在别的机器学习算法上，因为它们提供的功能是如此的基础以至于你可以任意组合出你希望的代码。如果工作的内容只涉及神经网络，而又不想编写代码，那么我们推荐使用Caffe。

第9章 实验现象及讨论

我们在MNIST数据集上分别实现了深度置信网络与卷积神经网络，使用深度置信网络训练得到的模型实现了98.72% 的识别正确率，使用卷积神经网络实现了98.9%的识别正确率。我们在CIFAR-10数据集上实现一个卷积神经网络，实现了62%的正确率，此外，通过Caffe测试了由xxx提出的网络构型的学习效果，并将其与我们的网络学习效果进行对比。

9.1 数据集简介

MNIST与CIFAR-10是学术界两个重要的数据集，这两个数据集一般作为标准数据集而存在，每当人们提出一种新的算法，都会用这两个数据集做验证，下面我们将简单介绍这两个数据集。

9.1.1 MNIST

MNIST数据集是一个真实世界中采集的手写数字图像数据集^[55]，它由NIST会议收集并持有，读者可到MNIST主页免费获取该数据集。这个数据集一共含有4个文件，分别存储训练数据、训练标签、测试数据、测试标签。文件以二进制文件形式存储，不过我们可以很容易编写一段小代码将其转换成图像。训练集共含有60000个样本，测试集含有10000个样本，这些样本收集自500位不同的人的手写字体。

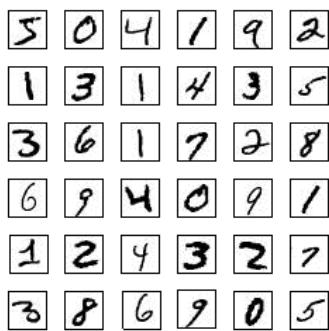


图 9-1 MNIST数据集部分数据样本

每个数据样本是 28×28 像素的灰度图像，由于引入了抗锯齿效果，所以图像数值范围是0 ~ 255而不是二值图像。图像已经经过预处理，因此图像会集中在中

心 20×20 的区域内，此外，图像的中心点与像素点的重心重合，所以如果要使用模板匹配的方法（比如k近邻，SVM等）进行分类的话对图像再进行一些预处理使得数字的几何中心与图像中心重合会改善你的算法性能。

如图9-1是MNIST数据集中的一小部分样本的展示，原始的数据应该是黑底白字的，为了美观，我们将其颜色反转并加上周围的边框。

9.1.2 CIFAR-10

CIFAR是一个由Alex Krizhevsky, Vinod Nair以及Geoffrey Hinton收集的一个含有8千万张图片的数据集^[56]，这些图片并没有经过手工标注。而CIFAR-10是这个数据集的一个子集，含有50000个训练样本和10000个测试样本，这些样本经过人手工标注为10个类别，分别是飞机、小汽车、鸟、猫等^[57]。读者可以从CIFAR-10的官网免费获取这个数据集，它包含7个文件，其中有5个文件是训练集，每个文件包含10000个训练样本，有1个文件存储测试集，包含10000个样本，这些样本都被随机打散，所以不用担心类别的出现顺序会导致算法性能上的差异。剩余的一个文件是标签值与类别名字的键值对。CIFAR-10为我们提供了三种存储形式，分别对应Python、MATLAB与二进制形式的数据存储格式，读者可根据自己的语言背景选取其中一种进行下载。

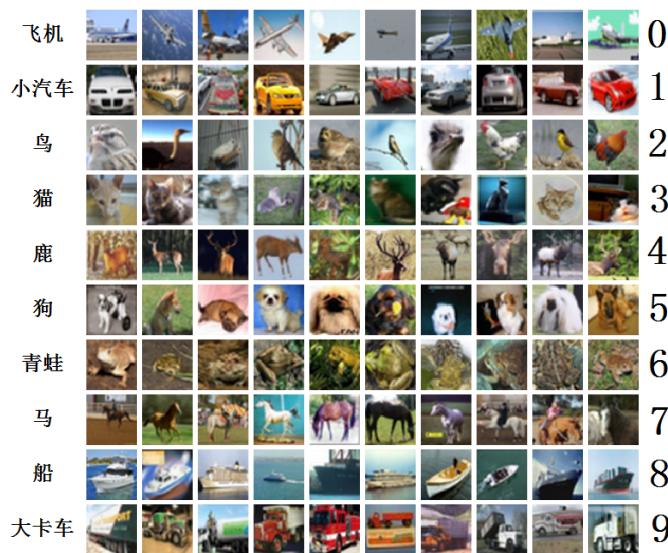


图 9-2 CIFAR-10数据集部分数据样本

每个数据样本都是大小为 32×32 的彩色图像，因此每张图像应包含三张 32×32 大小的矩阵，分别代表R、G、B三个原色通道。如图9-2所示是这个数据集的一部分样本，我们可以看到，这些图像更接近于现实生活中的图像，相比

于MNIST而言，每个类别个体的图像差异较大，而不像MNIST中每个类别的个体差异较小，所以在CIFAR-10数据集中使用模板匹配的方法进行分类是几乎不可能的。

9.2 深度置信网络在MNIST数据集上的性能

在MNIST数据集上，我们设计了一个7层神经网络，每层所含的节点分别是784、621、982、600、410、569、10，最底层的节点数是根据原始数据输入维度决定的，即 $28 \times 28 = 784$ ，最顶层的节点数是根据最终的类别决定的，即10个类别。中间的隐含层节点我们随意选取，这些节点是如此的随意以至于源自我的银行卡号。在深度置信网络中，对隐含节点并没有过多的要求，大致合理即可。

整个网络可以看做5个受限玻尔兹曼机叠加组成，分别是784~621，621~982，…，410~569，最后一层569~10是softmax分类器。在整个网络的训练过程中，我们先依次对其中的5个受限玻尔兹曼机做贪婪训练，即先训练784~621的受限玻尔兹曼机，训练完毕后将所有的样本（60000个）通过这个训练完毕的受限玻尔兹曼机前向传播，得到60000个621维的数据样本，用这些维度变换后的样本训练下一个，即621~982的受限玻尔兹曼机，以此类推。最后的softmax分类器其预训练是将其当做一个两层softmax网络进行预训练。

观察受限玻尔兹曼机的权值更新公式(4-43)、(4-44)以及(4-45)，为了方便大家观察，我们将这三个公式再书写一次

$$\frac{\partial \ln P(v)}{\partial w_{i,j}} \approx P(h_i = 1|v^{(0)})v_j^{(0)} - P(h_i = 1|v^{(k)})v_j^{(k)} \quad (9-1)$$

$$\frac{\partial \ln P(v)}{\partial b_{vi}} \approx v_j^{(0)} - v_j^{(k)} \quad (9-2)$$

$$\frac{\partial \ln P(v)}{\partial b_i} \approx P(h_i = 1|v^{(0)}) - P(h_i = 1|v^{(k)}) \quad (9-3)$$

这三个公式背后隐藏着一层重构的含义，即对于一个两层的受限玻尔兹曼机，在第一层的数据通过前向传播得到第二层的数据，第二层的数据反向注入得到第一层的数据，数据经过这样一个迁移，相当于利用提取的特征重构原始样本，因此在受限玻尔兹曼机的训练过程中，一个刻画其训练情况的方法是跟踪其重构误差，如图9-3所示是某一层受限玻尔兹曼机的重构误差下降曲线。

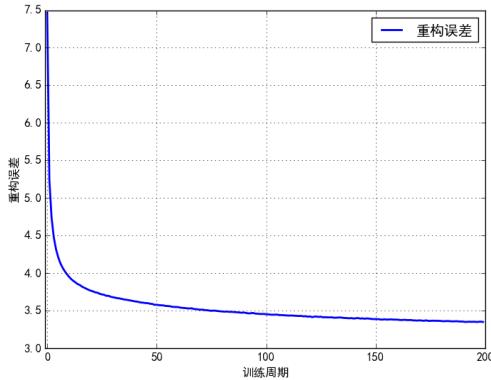


图 9-3 受限玻尔兹曼机重构误差下降曲线

从图9-3中我们不难发现，训练的前10个周期重构误差迅速下降，随后的周期下降速度缓慢，因此如果你的实验时间有限，那么可以只训练较少的周期即可，但训练更多的周期会有助于获取更好的实验结果。需要提到的一点是，在受限玻尔兹曼机的训练中，动量项是必须的，我们在实验中发现，不使用动量项时，在训练的初始阶段重构误差无法下降。

我们还追踪了受限玻尔兹曼机对图像的具体重构，如图9-4所示是一些样本在784~621的受限玻尔兹曼机中的重构情况，原始样本与重构样本我们已在图中标注。需要提醒的是，图9-4中的原始图像与图9-1中的原始图像不一致，这是因为图9-1中的样本每个像素点取值是0 ~ 255，而我们将这些像素点01化，所使用的方法是，对每个像素点除以255得到一个[0,1]区间的小数 p ，以数值 p 为概率将其置1，以 $1 - p$ 为概率将其置0。另一种01化的方法是，直接保留这个小数而不将其离散化。

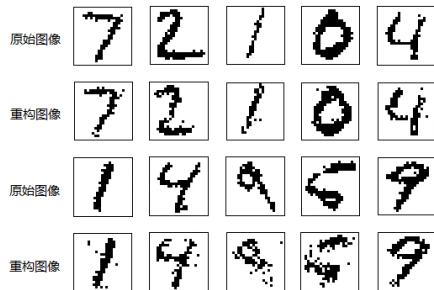


图 9-4 受限玻尔兹曼机对数字的重构

最顶层的softmax分类器的训练也是贪婪的，即它只训练569~10两层网络之间的参数，在这里，训练周期我们不建议太长，一般训练5~10个周期即可，否则在

随后的方向传播过程中，如果softmax预训练过久，则网络的输出误差较小，没有误差就难以进行反向传播，全局微调容易失败，这会导致网络陷入局部最优解，这个局部最优由最顶层的softmax决定而不是整个网络决定。

当整个网络预训练完毕后，我们执行全局的反向传播算法对参数进行微调，这个过程与传统的神经网络相同，我们不进行过多的叙述，如图9-5所示是整个网络进行反向传播时的误差下降曲线

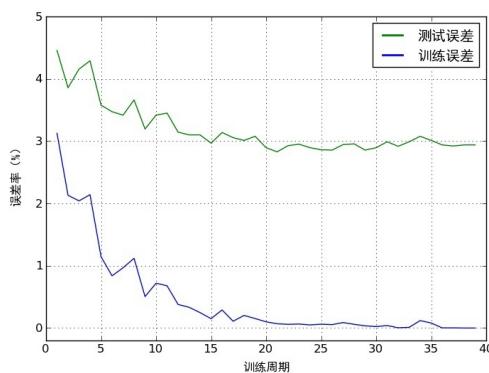


图 9-5 深度置信网络误差下降曲线

细心的读者会发现，图9-5的下降曲线表明最终这个网络并没有收敛到我们号称的正确率98.72%，确实如此，因为我们在数据迁移时遗失了一部分的数据，图9-5中的数据是前期研究工作中没有遗失的数据，因此这并不是最终的误差下降曲线。但是我们的实验是可重现的，只需按照我们的结构便可再现98.72% 的正确率，由于我们的时间有限，并没有将这个实验重新做一遍，感兴趣的读者可以尝试。

网络训练完毕后，我们选取了一部分网络识别错误的数字，将其展示在图9-6中，每个样本下边的黑色数字代表测试集给定的标签，而红色数字代表网络的预测标签。我们可以发现，这些错误的样本中，有一部分自身就带有二义性，人也难以区分究竟是哪个数字，而有一部分样本人可以很容易识别，网络却识别错误，还有一部分样本数据本来就是错误的，根本无法识别它是哪个数字，这时候我们不能舍弃机器也能将其识别出来。

在整个网络的训练中，我们使用了动量项、权衰减技术，而没有使用降维技术，因为数据维度并不是特别高，计算机的处理能力足以应付。如果原始数据的维度非常高，比如几百万维，那么就需要采用一些降维技术将原始数据进行压缩。

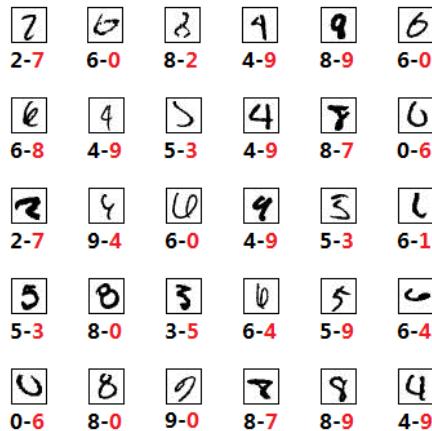


图 9-6 被网络误分类的样本

深度置信网络在一定程度上仍然带有模板匹配的气息，因为我们如果只使用2000个样本作为训练集而不是全部的60000个样本，网络也能实现90多的正确率，后面的实验中我们会看到，这种策略在卷积神经网络中是行不通的。

9.3 卷积神经网络在MNIST数据集上的性能

在MNIST数据集，我们设计了一个6层卷积网络，其网络构型描述如下

- 1张 28×28 的原始图像
- 卷积原始图像得到6张 24×24 特征图
- 对6张特征图进行采样得到6张 12×12 特征图
- 对6张特征图进行卷积得到12张 8×8 特征图
- 对12张特征图进行采样得到12张 4×4 特征图
- 此时12张 4×4 特征图无法再进行卷积，将其展开得到192个节点
- 192个节点与10个输出节点做全连接网络，与传统神经网络一样

其中，激活函数我们选取的是sigmoid函数，当然这个函数也可以换成ReLU函数，全连接网络我们采用平方误差作为准则，而不是深度置信网络中的softmax分类器。在这个网络中，我们不采用预训练而是直接进行全局的反向传播。以上的网络构型的选取方案都是随意的，并没有说一定要采用这个方案。

实验获取的训练误差及测试误差曲线如图9-7所示，训练完毕后，我们在MNIST上取得了98.9%的正确率。我们可以看到，在网络训练的前20个周期，训练误差与测试误差迅速下降，随后的周期中误差缓慢下降，到了后期，收敛十分缓慢，但依然会有下降的趋势。有意思的是，对网络训练一个很长的周期并不

会产生明显的过学习现象，所以如果你需要实现一个高识别率的网络，那么你可以放心地等待一段很长的时间。

值得注意的是，误差下降过程带有明显的波动性，在深度置信网络的训练过程中，一般到了训练的后期，识别率不会有太大的波动，比如，深度置信网络在到达98.70%的正确率后，其波动范围就在98.70%附近波动，可能是98.73%、98.74%。然而，在卷积网络中，到达98.70%后，它可能一下子跌到98.40%，然后又升到98.80%，这种较大范围的波动，应该不是随机梯度造成的，因为深度置信网络中我们采取随机梯度更新时也没有这么大的波动，我们推测这是因为卷积网络代表着一种非常强的惩罚（强迫权值共享），惩罚过大导致波动变大。

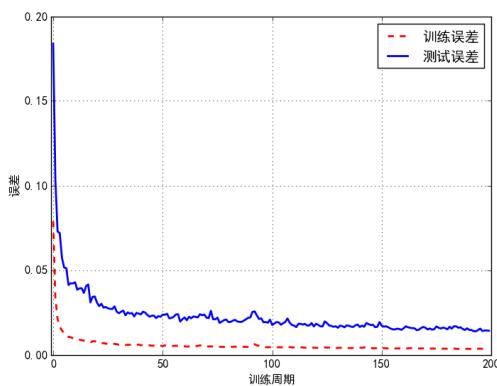
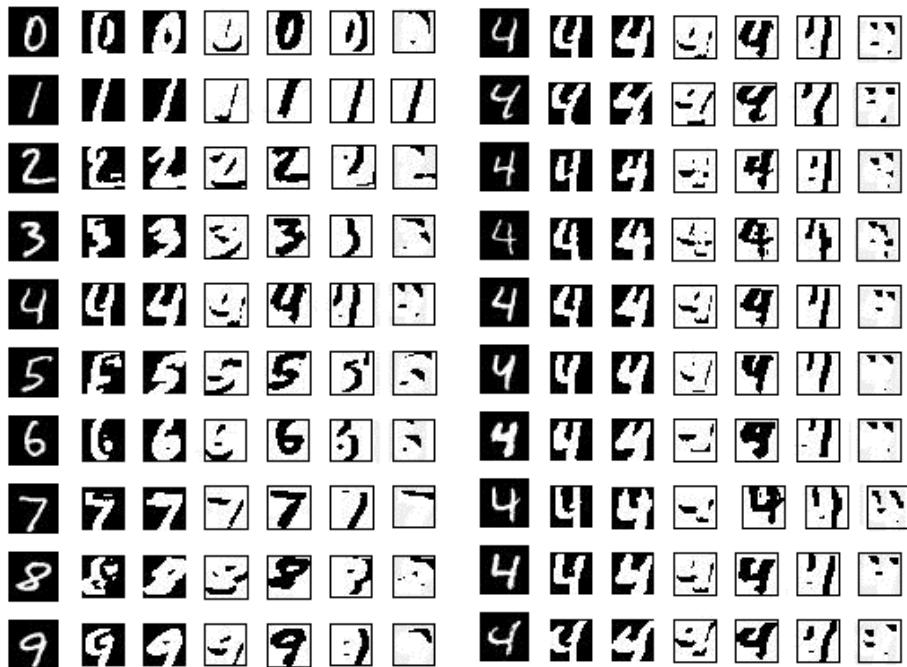


图 9-7 卷积神经网络训练误差及测试误差

在实验中，我们并没有采用权衰减，因为我们发现使用权衰减后网络的性能变差。此外，我们发现当改变网络中的一些参数，比如特征图的张数，学习率时，对网络的收敛影响较大，但对最终结果并没有太大影响。卷积网络对样本的需求量非常大，实验中，我们选取2000个样本作为训练集对网络进行训练，网络完全不会收敛，这与深度置信网络是不一样的，只有我们将所有的60000个样本作为训练集对网络进行训练时，网络才开始收敛。我们估计这是因为卷积网络的权值共享导致它只有通过大量的样本才能学习特征，这与模板匹配方法有着很大的区别。

网络训练完毕后，我们对测试集中的一部分数据进行观察，第一层卷积层提取出的特征如图9-8 a)所示。我们分别选取了0 ~ 9一共10个样本，每一行代表一个样本。其中，每一行的第一张是原始的 28×28 图像，随后六张是卷积出来的六张 24×24 大小的特征图。观察图9-8 a)，我们可以发现一些有意思的现象。例如，原始图像是黑底白字的，而有一些特征图反转成为白底黑字，又比如，第六张特征图是对图像进行边界检测，为了验证我们这个想法，我们随机选取了“4”这个

数字的几个样本进行特征抽取，其结果如图9-8 b) 显示，通过观察，我们不难发现，对于不同的写法，其提取到的特征都是近似的，它会检测“4”的左边竖线的上方一点以及对下来的一个折线，右边竖线的上方一点以及下方的一点。另一个有意思的现象是，第三张特征图看起来是一个3D图像，想象图像的左上角有一束阳光洒下，当我们伸出右手，掌心贴着当前的纸张页面，大拇指朝纸张左方，握拳，数字经过我们四个手指的方向旋转过一定角度后，那么阳光洒下的阴影如同第三张特征图所示。这个现象在第五张特征图中也出现了，只是第五张特征图的阳光处于左侧而不是左上角。同样伸出我们的右手，掌心贴着当前的纸张页面，大拇指朝纸张下方，握拳，我们可以看到数字经过我们四个手指的方向旋转过一定角度后，阳光洒下的投影正如第五张特征图所示。



a) 针对0-9不同数字提取到的特征图 b) 针对数字“4”提取到的特征像

图 9-8 第一层卷积层抽取得到的特征图

9.4 卷积神经网络在CIFAR-10数据集上的性能

相比于MNIST数据集，CIFAR-10数据集的识别更为困难。由于计算资源的限制，我们只在CIFAR-10上设计了一个较小的卷积网络，与MNIST手写数字的卷积网络构型类似，在CIFAR-10数据集上，我们同样设计了一个4层卷积网络，其网

结构型描述如下

- 3张 32×32 的原始图像
- 卷积原始图像得到9张 28×28 特征图
- 对9张特征图进行采样得到9张 14×14 特征图
- 尽管9张 5×5 特征图仍然可以被卷积，但我们依然将其展开为1764个节点
- 1764个节点与10个输出节点做全连接网络，与传统神经网络一样

网络的属性与MNIST实验中的相同，激活函数我们依然选取sigmoid函数，全连接网络依然采用平方误差作为准则，对网络进行训练得到训练集误差下降如图9-9 a)所示，测试集误差量下降如图9-9 b)所示。需要注意的是，看起来在图9-9 a)与图9-9 b)中误差下降速度非常快，对比与图9-7，其斜率更大，然而这是一个假象，因为我们对纵轴进行了尺度缩放，事实上在CIFAR中误差下降的非常慢，而且在训练集与测试集中仍然很大的误差可以下降。

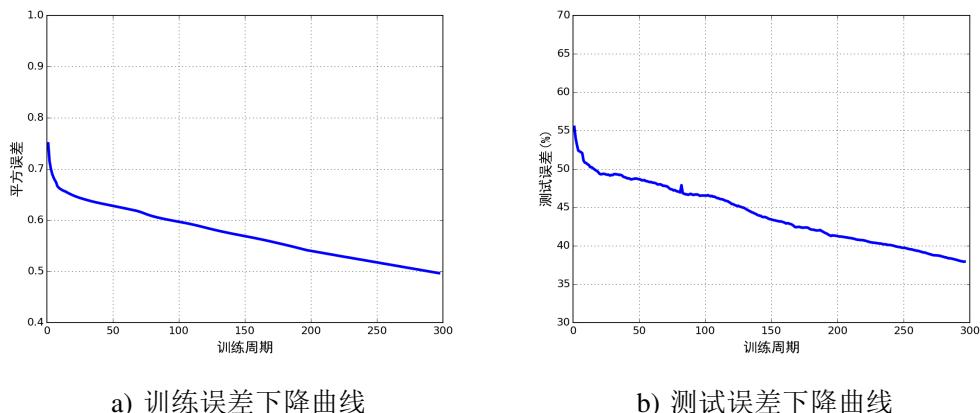


图 9-9 CIFAR的训练误差与测试误差

因为CIFAR-10数据集的复杂性，本应该建立一个庞大的网络进行训练，但由于我们的计算资源有限，时间紧迫，所以无法实现一个更大的卷积网络，此外，对网络的训练我们也仅仅训练了400个周期左右，因此，整个网络训练完毕后我们只得到62%的识别正确率。

如图9-10所示是一部分被网络正确认别的样本，其中每一行代表一个类别，图中总共包含了10个类别100个样本。观察这些样本我们会发现，网络能识别的图像对位移、旋转等性质具有不敏感性。例如，在飞机这个类别中，网络可以识别头部向左、向右等多个角度的飞机，还可以识别俯仰角不同的图片，由于这些图片不具备模板性，所以卷积网络基本没有了模板匹配的缺点。



图 9-10 被网络正确识别的CIFAR-10部分样本

我们跟踪了网络无法识别的样本，将其一部分绘制如图9-11所示。图中每个样本下方的黑色数字代表由训练集提供的标签值，而红色数字代表网络的输出标签值，标签值于类别名字的键值对可参照图9-1。我们发现，在一些样本中，网络会将大卡车错误地识别成小汽车或将小汽车错误地识别成大卡车，这似乎可以原谅，但有一些图像的错误识别是我们无法原谅的，例如将马识别成大卡车。这些错误的样本中很多样本人是可识别的，而机器不可识别，我们推测这是因为我们的网络设计得不够庞大，训练周期也步长，导致网络无法进行更好的特征提取，从而影响最终的识别效果。

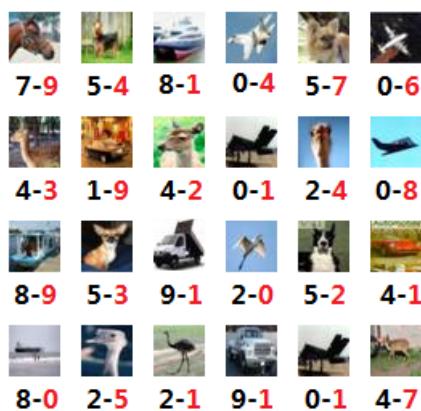


图 9-11 被网络错误识别的CIFAR-10部分样本

为了描述网络的特征提取性能，如同在MNIST上卷积网络的工作，我们同样将第一层卷积层提取到的特征绘制如图9-12所示。我们选取了“飞机”类别下的9个样本，每一行代表一个样本。其中，每一行的前三张黑白图像是原始 32×32 大小的RGB通道，这三张图像合成第四张 32×32 彩色图像。随后的九张是第一层卷积层提取到的九张 28×28 大小的特征图。对比这些原始数据与特征图，我们发现，原始数据中图像是含有冗余的，大部分的特征图过滤掉这些冗余信息，将飞机的边界提取出来形成特征图。这些特征图中，有一些特征例如第2张，第8张并没有提取到一些人可理解的特征，我们猜测随着训练周期的增加，这些特征将会逐渐显露出来，由于时间有限，我们并没有验证我们的猜想。

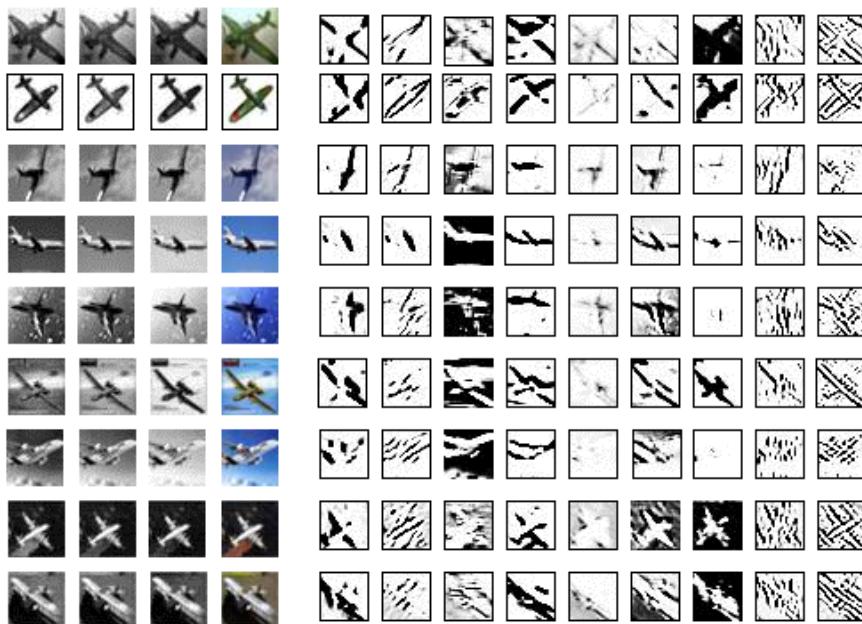


图 9-12 针对“飞机”类别提取的特征

这些特征图应该可以通过选取某几张组成彩色的特征图，但我们并不知道选取哪几张合适，所以我们并没有做这一步工作。此外，这些特征图表明，在他们之上应该再添加一些卷积层对特征继续提取，而不是我们设计的全连接网络，由于我们的计算资源十分匮乏，所以我们也没有进一步展开这项工作。

9.5 使用Caffe实现的CIFAR-10数据集训练

目前学术界认为CIFAR-10数据集已被解决，因为最好的识别效果由xxx实现，正确率为91%。另一识别率较高的网络由Alex Krizhevsky等人实现，其识别率为89%，这个网络稍微有点复杂所以我们不会在本文中提及，更多的讨论可阅读

其论文[41]。在Caffe中的卷积网络构型采取的是Alex Krizhevsky的方案，为了实现对比，我们使用Caffe验证了Alex Krizhevsky等人的网络构型在CIFAR-10数据集上的训练效果，其训练误差与测试误差曲线如图9-13所示

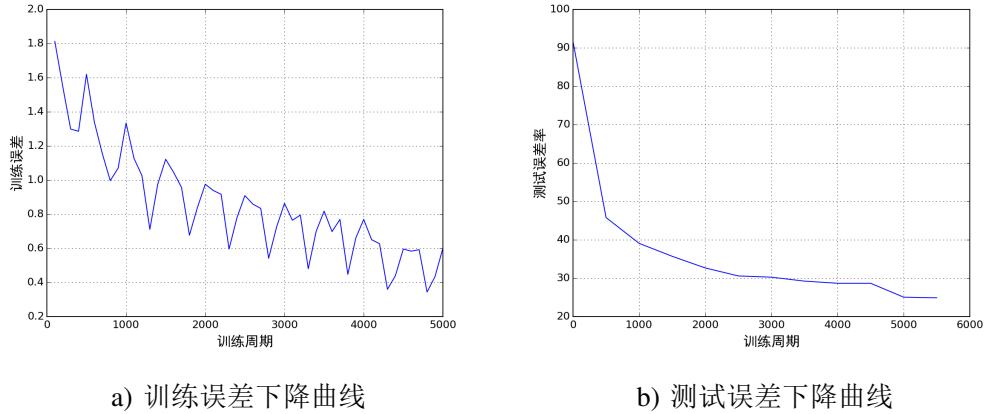


图 9-13 使用Caffe训练模型得到的误差

由于Caffe的源代码是每100个周期对训练误差做一次测试，每500个周期对测试误差做一次测试，所以图9-13中的曲线看起来并不光滑。此外我们可以看到，训练误差有很大的波动，这是因为Alex Krizhevsky设计的网络为了避免过学习使用了dropout技巧，这个技巧会对训练集产生一个很大的惩罚，从而导致训练误差出现波动，然而这个技巧并不会对测试误差产生较大的波动。

Alex Krizhevsky设计的网络规模远远超过我们设计的网络，例如他们的网络仅特征图的数量就达几百张，而我们的网络中只有9张。从图9-13中的测试误差我们可以看到，经过5000个周期的训练后，这个网络实现了75.1%的正确识别率，如果让网络继续训练，那么它将会收敛到文章[41]中号称的89%正确率。

对于比Alex Krizhevsky设计的网络，我们的网络实现的62%正确率看起来低得可怜，但情况似乎并没有这么糟糕，我们的网络只训练了大约300个周期后实现62%正确率，而Alex Krizhevsky的网络训练了5000个周期实现75.1%的正确率，如果我们横向对比，Alex Krizhevsky设计的网络在训练500个周期时得到的正确率为54.21%，这个效果低于我们的网络效果。当然，这其中并不能排除一些因素的印象，比如我们的实验中通过对数据镜像处理将训练集规模扩大为两倍，此外，我们也没有引入避免过学习的惩罚，这种惩罚在一定程度上会降低收敛速度，但会提高最终的收敛性能。尽管如此，我们乐观地估计，如果将我们的网络规模扩大，并且延长训练周期，那么应该能实现80%多的识别正确率。

在CIFAR-10这个任务中，我们可以看到卷积网络的威力所在，这个任务使用

传统的全连接神经网络大约只能实现40%左右的正确率，而使用模板匹配的方法基本是不可行的。卷积网络在图像识别中特征自学习的性能使得它远远超过机器学习中别的算法，目前主流的图像识别技术基本由卷积网络实现。

9.6 本章小结

本章讨论了我们在MNIST数据集与CIFAR-10数据集上获得的实验现象与结果讨论。对于MNIST这种数据特征已经被很好地预处理，没有过多噪声，也没有太多的旋转、伸缩、位移等变化的图像，那么采用采用全连接神经网络或采用机器学习中很多别的方法也能实现很好的识别效果。但对于CIFAR-10这种源自于真实生活中的图片，特征并没有经过预处理，并且每个类别的个体都带有很大的差异性，那么使用全连接网络并不是一个较好的效果。有些时候，卷积网络也被看做一种滤波，正如我们试验中看到的特征图，它会将原始图像中的特征自动过滤出来供给下一层网络，并且这个特征提取过程对位移等变换是不敏感的。我们曾经提过，深度学习也称特征学习，即神经网络对数据进行逐层的特征提取，这些特征必须是能描述原始图像的特征，根据这些特征能够反向还原出数据的原始大致面貌。无论是那种方法，对数据的需求量都很大，并且任务越复杂，需求越大，小样本的数据并不适合使用深度学习处理，因为小样本机器无法探索出那些特征才是对分类有帮助的。对于小样本数据，应该在模型中加入人的知识，而不是奢望机器自动学习这些知识，对于大的数据，如果能往模型中添加人的知识¹，那么也能很大程度地提高机器的学习性能，毕竟人工智能业内有一句话：有多少人工，就有多少智能。

¹例如卷积网络中我们就是这样干的

结 论

本文的主要目的在于介绍深度学习在图像识别中的应用，文中有两条主线贯穿全文。我们先从控制论与机器学习的关系说起，随后引入了第一个刻画数据分布的模型即受限玻尔兹曼机，为了介绍受限玻尔兹曼机的训练方法，我们讨论了马尔可夫链蒙特卡罗方法。在受限玻尔兹曼机的基础上，我们讨论了如何利用它们堆叠得到深度置信网络，并介绍了反向传播算法。至此我们完成了深度学习的第一条主线即深度置信网络的讨论。随后我们展开了第二条主线即卷积神经网络的讨论，并在其后介绍了神经网络的设计技巧与GPU高性能计算。

最后，我们通过三个实验测试了深度学习在图像识别中的识别效果。在MNIST数据集中，使用深度置信网络实现了98.7%的识别正确率，使用卷积神经网络实现了98.9%的识别正确率。在CIFAR-10数据集中，我们使用卷积神经网络实现了62%的识别正确率，尽管这个结果与当前世界顶尖的结果91%的正确率相差较远，但通过与Caffe训练的卷积网络做对比，我们乐观地认为我们的网络仍有很大的收敛空间。

深度学习作为一种特征学习，较传统模式识别方法有本质化的改变，尤其是在图像识别领域。这套方法应当值得研究人员关注。

参考文献

- [1] Bishop C M, et al. Pattern recognition and machine learning[M]. Vol. 1.[S.l.]: springer New York, 2006.
- [2] Vladimir N. Vapnik 原著, 张学工 译. 统计机器学习理论的本质[M]. 北京: 清华大学出版社, 2000.
- [3] Duda R O, Hart P E, Stork D G. Pattern classification[M].[S.l.]: John Wiley & Sons,, 1999.
- [4] Bengio Y. Learning deep architectures for AI[J]. Foundations and trends® in Machine Learning, 2009, 2(1):1–127.
- [5] Hinton G E. To recognize shapes, first learn to generate images[J]. Progress in brain research, 2007, 165:535–547.
- [6] Hinton G, Osindero S, Teh Y W. A fast learning algorithm for deep belief nets[J]. Neural computation, 2006, 18(7):1527–1554.
- [7] Hinton G. A practical guide to training restricted Boltzmann machines[J]. Momentum, 2010, 9(1):926.
- [8] Erhan D, Manzagol P A, Bengio Y, et al. The difficulty of training deep architectures and the effect of unsupervised pre-training[C]//International Conference on Artificial Intelligence and Statistics. .[S.l.]: [s.n.] , 2009:153–160.
- [9] Bengio Y, Lamblin P, Popovici D, et al. Greedy layer-wise training of deep networks[J]. Advances in neural information processing systems, 2007, 19:153.
- [10] Poultney C, Chopra S, Cun Y L, et al. Efficient learning of sparse representations with an energy-based model[C]//Advances in neural information processing systems. .[S.l.]: [s.n.] , 2006:1137–1144.
- [11] Hinton G E, Salakhutdinov R R. Reducing the dimensionality of data with neural networks[J]. Science, 2006, 313(5786):504–507.
- [12] LeCun Y, Bengio Y, Hinton G. Deep learning[J]. Nature, 2015, 521(7553):436–444.
- [13] 冯·诺伊曼 原著, 甘子玉 译. 计算机与人脑[M]. 北京: 北京大学出版社, 2010.
- [14] Schmidhuber J. Deep Learning in Neural Networks: An Overview[J]. arXiv preprint arXiv:1404.7828, 2014.
- [15] Williams D R G H R, Hinton G. Learning representations by back-propagating errors[J]. Nature, 1986:323–533.

- [16] Ma-ckay, D.J.C. 原著, 肖明波 等译. 信息论、推理与学习算法[M]. 北京: 高等教育出版社, 2006.
- [17] Ackley D H, Hinton G E, Sejnowski T J. A learning algorithm for boltzmann machines[J]. Cognitive science, 1985, 9(1):147–169.
- [18] LeCun Y, Boser B, Denker J S, et al. Backpropagation applied to handwritten zip code recognition[J]. Neural computation, 1989, 1(4):541–551.
- [19] LeCun Y, Bottou L, Bengio Y, et al. Gradient-based learning applied to document recognition[J]. Proceedings of the IEEE, 1998, 86(11):2278–2324.
- [20] Arel I, Rose D C, Karnowski T P. Deep machine learning-a new frontier in artificial intelligence research [research frontier][J]. Computational Intelligence Magazine, IEEE, 2010, 5(4):13–18.
- [21] Salakhutdinov R. Learning deep generative models[D].[S.l.]: University of Toronto, 2009.
- [22] 余凯, 贾磊, 陈雨强, et al. 深度学习的昨天, 今天和明天[J]. 计算机研究与发展, 2013, 50(9):1799–1804.
- [23] He K, Zhang X, Ren S, et al. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification[J]. arXiv preprint arXiv:1502.01852, 2015.
- [24] 维纳 原著, 陈步 译. 人有人的用处: 控制论与社会[M]. 北京: 北京大学出版社, 2010.
- [25] Tom M.Mitchell 原著, 曾华军 等译. 机器学习[M]. 北京: 机械工业出版社, 2003.
- [26] LeCun Y, Chopra S, Hadsell R, et al. A tutorial on energy-based learning[J]. Predicting structured data, 2006, 1:0.
- [27] Andrieu C, De Freitas N, Doucet A, et al. An introduction to MCMC for machine learning[J]. Machine learning, 2003, 50(1-2):5–43.
- [28] Carreira-Perpinan M A, Hinton G E. On contrastive divergence learning[C]//. .[S.l.]: [s.n.] , 2005.
- [29] Tieleman T. Training restricted Boltzmann machines using approximations to the likelihood gradient[C]//Proceedings of the 25th international conference on Machine learning. .[S.l.]: [s.n.] , 2008:1064–1071.
- [30] Barron A R. Universal approximation bounds for superpositions of a sigmoidal function[J]. Information Theory, IEEE Transactions on, 1993, 39(3):930–945.

- [31] Glorot X, Bordes A, Bengio Y. Deep sparse rectifier neural networks[C]/International Conference on Artificial Intelligence and Statistics. .[S.l.]: [s.n.] , 2011:315–323.
- [32] Boser B E, Guyon I M, Vapnik V N. A training algorithm for optimal margin classifiers[C]/Proceedings of the fifth annual workshop on Computational learning theory. .[S.l.]: [s.n.] , 1992:144–152.
- [33] Cortes C, Vapnik V. Support-vector networks[J]. Machine learning, 1995, 20(3):273–297.
- [34] Erhan D, Bengio Y, Courville A, et al. Why does unsupervised pre-training help deep learning?[J]. The Journal of Machine Learning Research, 2010, 11:625–660.
- [35] Hinton G E. Learning multiple layers of representation[J]. Trends in cognitive sciences, 2007, 11(10):428–434.
- [36] Abdel-Hamid O, Mohamed A r, Jiang H, et al. Applying convolutional neural networks concepts to hybrid NN-HMM model for speech recognition[C]/Acoustics, Speech and Signal Processing (ICASSP), 2012 IEEE International Conference on. .[S.l.]: [s.n.] , 2012:4277–4280.
- [37] Sukittanon S, Surendran A C, Platt J C, et al. Convolutional networks for speech detection.[C]//. .[S.l.]: [s.n.] .
- [38] LeCun Y, Bengio Y. Convolutional networks for images, speech, and time series[J]. The handbook of brain theory and neural networks, 1995, 3361(10).
- [39] Bouvrie J. Notes on convolutional neural networks[J]. 2006.
- [40] Lee H, Grosse R, Ranganath R, et al. Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations[C]/Proceedings of the 26th Annual International Conference on Machine Learning. .[S.l.]: [s.n.] , 2009:609–616.
- [41] Krizhevsky A, Sutskever I, Hinton G E. Imagenet classification with deep convolutional neural networks[C]/Advances in neural information processing systems. .[S.l.]: [s.n.] , 2012:1097–1105.
- [42] Kruskal J B. Multidimensional scaling by optimizing goodness of fit to a nonmetric hypothesis[J]. Psychometrika, 1964, 29(1):1–27.
- [43] Simard P Y, Steinkraus D, Platt J C. Best practices for convolutional neural networks applied to visual document analysis[C]/null. .[S.l.]: [s.n.] , 2003:958.

- [44] Baldi P, Sadowski P J. Understanding Dropout[C]//Advances in Neural Information Processing Systems. .[S.l.]: [s.n.] , 2013:2814–2822.
- [45] Goodfellow I J, Warde-Farley D, Mirza M, et al. Maxout networks[J]. arXiv preprint arXiv:1302.4389, 2013.
- [46] Tenenbaum J B, De Silva V, Langford J C. A global geometric framework for non-linear dimensionality reduction[J]. Science, 2000, 290(5500):2319–2323.
- [47] Roweis S T, Saul L K. Nonlinear dimensionality reduction by locally linear embedding[J]. Science, 2000, 290(5500):2323–2326.
- [48] Schölkopf B, Mika S, Smola A, et al. Kernel PCA pattern reconstruction via approximate pre-images[M]//ICANN 98.[S.l.]: Springer, 1998:147–152.
- [49] Mnih V. Cudamat: a CUDA-based matrix class for python[J]. Department of Computer Science, University of Toronto, Tech. Rep. UTML TR, 2009, 4.
- [50] Tieleman T. Gnumpy: an easy way to use GPU boards in Python[J]. Department of Computer Science, University of Toronto, 2010.
- [51] Klöckner A, Pinto N, Lee Y, et al. PyCUDA and PyOpenCL: A Scripting-Based Approach to GPU Run-Time Code Generation[J]. Parallel Computing, 2012, 38(3):157–174.
- [52] Jia Y, Shelhamer E, Donahue J, et al. Caffe: Convolutional Architecture for Fast Feature Embedding[J]. arXiv preprint arXiv:1408.5093, 2014.
- [53] Shane Cook 原著, 苏统华 等译. CUDA并行程序设计GPU编程指南[M]. 北京: 机械工业出版社, 2014.
- [54] Jason Sanders, Edward Kandrot 原著, 聂雪军 等译. GPU高性能编程CUDA实战[M]. 北京: 机械工业出版社, 2011.
- [55] LeCun Y, Cortes C. MNIST handwritten digit database[J]. AT&T Labs [Online]. Available: <http://yann.lecun.com/exdb/mnist>, 2010.
- [56] Krizhevsky A, Hinton G. Learning multiple layers of features from tiny images[J]. 2009.
- [57] Krizhevsky A, Hinton G. CIFAR-10 database[J]. Canadian Instistute for Advanced Resscherch [Online]. Available: <http://www.cs.toronto.edu/~kriz/cifar.html>, 2009.

哈尔滨工业大学本科毕业设计（论文）原创性声明

本人郑重声明：在哈尔滨工业大学攻读学士学位期间，所提交的毕业设计（论文）《深度学习在图像识别中的应用》，是本人在导师指导下独立进行研究工作所取得的成果。对本文的研究工作做出重要贡献的个人和集体，均已在文中以明确方式注明，其它未注明部分不包含他人已发表或撰写过的研究成果，不存在购买、由他人代写、剽窃和伪造数据等作假行为。

本人愿为此声明承担法律责任。

作者签名:

日期: 年 月 日

致 谢

首先我需要感谢我的导师杨旭东先生。在这个课题开始之前我和杨老师对深度学习都没有过多的接触，整个课题研究过程中我们多次进行讨论，共同学习。杨老师对我十分信任，并不过多地干预我的工作内容，让我随兴趣展开研究，只在关键地方为我指导，并且深信我的实验能成功。感谢杨老师对我的信任与支持，让我有足够多的时间与信心来深入研究这个课题，特此致敬。

其次我需要感谢计算机学院硬件实验室的陈慧鹏先生。陈老师是我大学四年思想启蒙教师，他教会了我如何思考以及人生的意义，他有趣的《计算机组成原理》课程最终使我走上计算机的道路，如果没有遇到陈老师，或许大学四年我都找不到自己兴趣所在。陈老师为我提供了舒适的实验室条件与计算资源，在这个实验室中我完成了论文的大部分研究工作，再次感谢陈老师的帮助与支持。

特别要感谢计算机学院模式识别研究中心的刘家锋先生。我曾旁听过刘老师的《模式识别》课程，刘老师课堂上严谨的授课方式，信手拈来的公式推导使我折服。承蒙刘老师不嫌弃我学识疏浅，在课后仍为我解惑释疑，在百忙之中抽出时间帮我审阅论文，我的感激之情难以言表。

我还需要感谢计算机学院自然计算实验室的刘扬先生，刘老师的《机器学习》课程深入浅出，生动地将枯燥复杂的数学语言形象化，在年初我实验遇到瓶颈时给予我鼓励与建设性意见，并带我领略了当前机器学习的研究进展与热点，十分感谢他的无私帮助。

最后我还需要感谢在实验室遇到的几位不知名的老师，其中一位老师为我解答了机器学习与模式识别两者间的差异性，这个问题长期困扰着我。另一位与我一同讨论的老师，我俩的讨论使我对深度学习中的一些模棱两可的概念清晰化。还有多位给予我鼓励的老师，遗憾的是我并不知道他们的姓名，但我仍需要对他们表示崇高的敬意。

我的文字表达能力不强，大家从我的论文也可以感受到，文字不足以表达我的感激之情，没有以上多位老师以及我身边的同学给予我的支持，这篇论文将无法完成。最后的最后，让我再一次将我的敬意献给以上的多位老师以及同学。

附录 A 深度置信网络源代码

本附录中我们提供深度置信网络在MNIST中的实现源代码。在这个实验中，我们使用Gnumpy库为程序加速，使得原本需要执行大约需要一个月的训练只需要在1天之内完成。出于页面限制，我们对代码进行了一些格式调整，这使得代码格式看起来并不规范。此外，为了页面简洁，我们删除了一部分无用代码，由于在我写下这些代码的时候，只有我和上帝知道这些代码究竟干了什么而现在只剩下上帝知道了，所以我不确定是否删掉了一些看起来无用但有用的代码。因此这些代码不一定能执行成功，但是读者可按照这些代码的大体思路自行根据自身的语言背景重新实现一遍。

这几个文件中，Main.py是整个程序的主入口。MNIST.py是一个工具文件，主要提供获取数据集以及绘制图像功能，需要注意的是，对于MNIST，我们已经将数据的前几个字节中的魔数去掉了，所以如果你的数据是直接从MNIST官网上获取的话，在读取数据的时候需要跳过几个字节，具体的参考官方文档。RBM.py是受限玻尔兹曼机的类文件。softmax.py是softmax分类器的类，它继承于RBM这个类，并且对training()方法进行重写，从而实现多态性。DBNs.py是深度置信网络的类文件，它由多个RBM与一个softmax组成，可对其执行分层的预训练与全局的权值微调。

代码 A.1 Main.py

```

1 from numpy import *
2 from DBNs import *
3 import numpy as np
4
5 nodeNum = [784, 621, 982, 600, 410, 569, 10]
6 dbn = DBNs(nodeNum)
7 #dbn = DBNs(nodeNum, loadParameter=True)
8 dbn.preTraining()
9 #dbn.reconstruct()
10 dbn.recognize()
```

代码 A.2 MNIST.py

```
1 from struct import *
2 from numpy import *
3 from scipy import misc
4 import Image
5
6 def getData():
7     trainingFile = open(r'trainingData', 'rb')
8     trainingData = fromfile(trainingFile, dtype = uint8)
9             .reshape(-1, 784)
10    trainingFile.close()
11
12    trainingLabelFile = open(r'trainingLabel', 'rb')
13    trainingLabel = fromfile(trainingLabelFile, dtype=uint8)
14    trainingLabelFile.close()
15
16    testImageFile = open(r'testData', 'rb')
17    testData = fromfile(testImageFile, dtype = uint8)
18            .reshape(-1, 784)
19    testImageFile.close()
20
21    testLabelFile = open(r'testLabel', 'rb')
22    testLabel = fromfile(testLabelFile, dtype = uint8)
23    testLabelFile.close()
24    return trainingData, trainingLabel, testData, testLabel
25
26 def createBinData(trainingData, trainingLabel,
27                   testData, testLabel):
28     trainingData = (255 - trainingData) / 255.0
29     scale = random.random(trainingData.shape)
30     trainingData[greater(trainingData, scale)] = 1
31     trainingData[less_equal(trainingData, scale)] = 0
32     trainingData = uint8(trainingData)
```

```
33     fp = open(r'binTrainingData0', 'wb')
34     fp.write(trainingData)
35     fp.close()
36
37     testData = (255 - testData) / 255.0
38     scale = random.random(testData.shape)
39     testData[greater(testData, scale)] = 1
40     testData[less_equal(testData, scale)] = 0
41     testData = uint8(testData)
42     fp = open(r'binTestData', 'wb')
43     fp.write(testData)
44     fp.close()
45
46 def getBinData():
47     trainingImageFile = open(r'binTrainingData0', 'rb')
48     trainingData = fromfile(trainingImageFile, dtype=uint8)
49                 .reshape(-1, 784)
50     trainingImageFile.close()
51
52     trainingLabelFile = open(r'trainingLabel', 'rb')
53     trainingLabel = fromfile(trainingLabelFile, dtype=uint8)
54     trainingLabelFile.close()
55
56     testImageFile = open(r'binTestData', 'rb')
57     testData = fromfile(testImageFile, dtype = uint8)
58                 .reshape(-1, 784)
59     testImageFile.close()
60
61     testLabelFile = open(r'testLabel', 'rb')
62     testLabel = fromfile(testLabelFile, dtype = uint8)
63     testLabelFile.close()
64     return double(trainingData), trainingLabel,
65                 double(testData), testLabel
```

```
66 def loadTrainingData(layer, dimension):
67     fileName = 'binTrainingData' + str(layer)
68     fp = open(fileName, 'rb')
69     trainingData = fromfile(fp, dtype = uint8)
70             .reshape(-1, dimension)
71     fp.close()
72     return trainingData
73
74 def createImage(data, imageName = 'temp', mode = 'jpg',
75                 trans = True, showNow = False):
76     if trans == True:
77         image = 255 - data
78     else:
79         image = data
80     image.shape = 28, 28
81     imageName = imageName + '.' + mode
82     misc.imsave(imageName, image)
83     if showNow == True:
84         image = Image.open(imageName)
85         image.show()
86 def createBinImage(data, imageName = 'temp', mode = 'jpg',
87                     trans = False, showNow = False):
88     if trans == True:
89         image = float64(1 - data)
90     else:
91         image = float64(data)
92     image.shape = 28, 28
93     imageName = imageName + '.' + mode
94     misc.imsave(imageName, image)
95     if showNow == True:
96         image = Image.open(imageName)
97         image.show()
```

代码 A.3 RBM.py

```
1 import numpy as np
2 import gnumpy as GPU
3 from scipy import stats
4 import scipy.weave as weave
5 from MNISTData import *
6
7 class RBM:
8     def __init__(self, layer, visHid, k=1,
9                  learningRate = 0.001, maxEpoch = 100,
10                 loadParameter = False):
11         self.layer = layer
12         self.numVis = visHid[0]
13         self.numHid = visHid[1]
14         self.k = k
15         self.learningRate = learningRate
16         self.maxEpoch = maxEpoch
17         if loadParameter == False:
18             print "init", str(layer)+"th RBM's parameters"
19             Gaussian = stats.norm(0, 0.01)
20             self.W = np.array(Gaussian.rvs(
21                             (self.numHid, self.numVis)), np.float32)
22             trainingData = loadTrainingData(self.layer,
23                                             self.numVis)
24             p = np.mean(trainingData, axis = 0)
25             p = np.true_divide(p, subtract(1.0001, p))
26             self.visBais = np.log(p).reshape(self.numVis,1)
27             self.hidBais = np.zeros((self.numHid,1), double)
28             self.featureExtract()
29         else:
30             print "load", str(layer)+"th RBM's parameters"
31             self.W, self.visBais, self.hidBais =
32                             self.loadParameter()
```

```

33     def training(self):
34         from numpy import logistic
35         def sampleHidGivenVis(weight , v_t , hidBais):
36             h = GPU.logistic(GPU.dot(weight , v_t ) + hidBais)
37             hSample = h .rand () < h
38             return hSample
39         def sampleVisGivenHid(weight , h_t , visBais ):
40             v = GPU.logistic(GPU.dot(weight .T , h_t )+visBais )
41             vSample = v .rand () < v
42             return vSample
43
44         S = loadTrainingData(self .layer , self .numVis)
45         S = np .float32(S)
46         maxBatch = 100
47         S .shape = maxBatch , -1 , self .numVis
48         momentum = 0.9
49         eta = 0.001
50
51         for epoch in range(self .maxEpoch):
52             weight = GPU .garray (self .W)
53             visBais = GPU .garray (self .visBais )
54             hidBais = GPU .garray (self .hidBais )
55             deltaWeight = GPU .zeros (weight .shape)
56             deltaVisBais = GPU .zeros (self .visBais .shape)
57             deltaHidBais = GPU .zeros (self .hidBais .shape)
58             for batch in range(maxBatch):
59                 v_0 = GPU .garray (S[batch ].T)
60                 v_t = GPU .garray (S[batch ].T)
61                 for i in range(self .k):
62                     h_t=sampleHidGivenVis(weight , v_t , hidBais )
63                     v_t=sampleVisGivenHid(weight , h_t , visBais )
64                     prob_0 = GPU .logistic (GPU .dot (weight , v_0)
65                                         + hidBais )

```

```

66         prob_t = GPU.logistic(GPU.dot(weight, v_t)
67                         + hidBais)
68         deltaWeight = momentum*deltaWeight + eta
69                         * (GPU.dot(prob_0, v_0.T)
70                         - GPU.dot(prob_t, v_t.T))
71         deltaVisBais = momentum*deltaVisBais + eta
72                         * (v_0.sum(1) - v_t.sum(1))
73                         .reshape(-1, 1)
74         deltaHidBais = momentum*deltaHidBais + eta
75                         *(prob_0.sum(1)-prob_t.sum(1))
76                         .reshape(-1, 1)
77         weight += deltaWeight / maxBatch
78         hidBais += deltaHidBais / maxBatch
79         visBais +=deltaVisBais / maxBatch
80         self.W = weight.asarray()
81         self.visBais = visBais.asarray()
82         self.hidBais = hidBais.asarray()
83         print epoch, 'epoch_complete!'
84         self.saveParameter()
85         print "training", str(self.layer)+"th","RBM_complete"
86         self.featureExtract()
87
88     def saveParameter(self):
89         fp = open(r'weight' + str(self.layer), 'wb')
90         fp.write(self.W)
91         fp.close()
92         fp = open(r'veisBais' + str(self.layer), 'wb')
93         fp.write(self.visBais)
94         fp.close()
95         fp = open(r'hidBais' + str(self.layer), 'wb')
96         fp.write(self.hidBais)
97         fp.close()
98         print "layer", self.layer,"parameters_had_been_save"

```

```

99     def loadParameter(self):
100         fp = open(r'weight' + str(self.layer), 'rb')
101         weight = fromfile(fp, dtype = double)
102             .reshape(self.numHid, self.numVis)
103         fp.close()
104         fp = open(r'visBais' + str(self.layer), 'rb')
105         visBais = fromfile(fp, dtype = double)
106             .reshape(self.numVis, 1)
107         fp.close()
108         fp = open(r'hidBais' + str(self.layer), 'rb')
109         hidBais = fromfile(fp, dtype = double)
110             .reshape(self.numHid, 1)
111         fp.close()
112     return weight, visBais, hidBais
113
114     def featureExtract(self, lowFeature = None):
115         if lowFeature == None:
116             lowFeature = loadTrainingData(self.layer,
117                                         self.numVis)
118             numCase = lowFeature.shape[0]
119             highFeature = zeros((numCase, self.numHid),
120                                 dtype = uint8)
121             for case in range(numCase):
122                 prob = 1.0 / (1 + np.exp(
123                     - np.dot(lowFeature[case], self.W.T)
124                     - self.hidBais.T))
125                 scale = np.random.random(prob.shape)
126                 temp = zeros(prob.shape, dtype = np.uint8)
127                 temp[less(scale, prob)] = 1
128                 highFeature[case] = temp
129
130             fileName = 'binTrainingData' + str(self.layer+1)
131             fp = open(fileName, 'wb')

```

```
132     fp . write ( highFeature )
133     fp . close ()
134     print "high_feature_has_been_extracted !"
135 else :
136     prob = 1.0 / ( 1 + np . exp ( - np . dot ( lowFeature , self . W . T )
137                                     - self . hidBais . T ) )
138     return prob
139
140 def reconstruct ( self , highFeature ):
141     highFeature . shape = 1 , -1
142     prob = 1.0 / ( 1 + np . exp ( - np . dot ( highFeature , self . W )
143                                     - self . visBais . T ) )
144     return prob
```

代码 A.4 softmax.py

```
1 from RBM import *
2 from scipy.odr.odrpack import Output
3
4 class softmax(RBM):
5     def __init__(self, layer, visHid,
6                  learningRate = 0.001, maxEpoch = 5,
7                  loadParameter = False):
8         self.layer = layer
9         self.numVis = visHid[0]
10        self.numHid = visHid[1]
11        self.learningRate = learningRate
12        self.maxEpoch = maxEpoch
13        if loadParameter == False:
14            print "init", str(layer)+"th_RBM's parameters"
15            Gaussian = stats.norm(0, 0.01)
16            self.W = np.array(Gaussian.rvs(
17                (self.numHid, self.numVis)), double)
18            trainingData = loadTrainingData(self.layer,
19                                            self.numVis)
20            p = np.mean(trainingData, axis = 0)
21            p = np.true_divide(p, subtract(1.0001, p))
22            self.visBais = np.log(p).reshape(self.numVis,1)
23            self.hidBais = np.zeros((self.numHid,1), double)
24            self.saveParameter()
25        else:
26            print "load", str(layer)+"th_RBM's parameters"
27            self.W, self.visBais, self.hidBais =
28                        self.loadParameter()
29    def training(self):
30        Gaussian = stats.norm(0, 0.001)
31        trainingData = loadTrainingData(self.layer,
32                                       self.numVis)
```

```

33     trainingLabel = getBinData()[1]
34     numCase = trainingData.shape[0]
35     weight = np.array(Gaussian.rvs(size=(self.numHid,
36                                         self.numVis+1)),
37                         dtype = double)
38     for epoch in range(self.maxEpoch):
39         for i in range(numCase):
40             sample = trainingData[i].reshape(-1, 1)
41             target = trainingLabel[i]
42             sample = vstack((1, sample))
43             targetOut = np.exp(dot(weight, sample))
44                         .reshape(-1)
45             Z = sum(targetOut)
46             targetOut = (targetOut / Z).reshape(-1, 1)
47             delta=-dot(targetOut, sample.reshape(1,-1))
48             delta[target] += sample.reshape(-1)
49             delta += 0.005 * weight
50             weight += 0.001 * delta
51             self.W = copy(weight[:, 1:])
52                         .reshape(self.W.shape))
53             self.hidBais = copy(weight[:, 0].reshape(
54                                         self.hidBais.shape))
55             print epoch, "complete"
56             print "training", str(self.layer)+"th", "RBM_complete"
57             self.saveParameter()
58
59     def featureExtract(self, lowFeature):
60         prob=np.exp(dot(lowFeature, self.W.T)+self.hidBais.T)
61         Z = sum(prob)
62         prob = (prob / Z)
63         return prob

```

代码 A.5 DBNs.py

```
1 from numpy import *
2 from RBM import *
3 from softmax import *
4 from MNISTData import *
5
6 class DBNs:
7     def __init__(self, nodeNum, loadParameter = False):
8         self.rbmPair = zip(nodeNum[:-1], nodeNum[1:])
9         rbmsNum = len(nodeNum) - 1
10        self.rbms = [None for i in range(rbmsNum)]
11        if loadParameter == True:
12            for i in range(rbmsNum):
13                if i != rbmsNum - 1:
14                    self.rbms[i] = RBM(i, self.rbmPair[i],
15                                         loadParameter = True)
16                else:
17                    self.rbms[i] = softmax(i, self.rbmPair[i],
18                                         loadParameter = True)
19        else:
20            for i in range(rbmsNum):
21                if i != rbmsNum - 1:
22                    self.rbms[i] = RBM(i, self.rbmPair[i])
23                else:
24                    self.rbms[i] = softmax(i, self.rbmPair[i])
25
26    def preTraining(self):
27        for rbm in self.rbms:
28            rbm.training()
29        print "DBNs_had_pre-trained_complete!"
```

```

33     def fineTuning(self, maxEpoch = 10):
34         import numpy as GPU
35         def makeBatch(maxBatch = 100):
36             trainingData = (255 - getData()[0]) / 255.0
37             trainingData = getBinData()[0]
38             trainingData=hstack(
39                 (ones((trainingData.shape[0], 1)),
40                  trainingData))
41             label = getBinData()[1]
42             trainingLabel = np.zeros((numCase, 10))
43             trainingLabel[[i for i in range(numCase)],
44                           label] = 1
45             return trainingData.reshape(maxBatch, -1, 785),
46                   trainingLabel.reshape(maxBatch, -1,10 ),
47
48         def getAbstract(weight, input):
49             prob = [input]
50             for W in weight[:-1]:
51                 temp = GPU.dot(prob[-1], W.T).logistic()
52                 temp = hstack((ones((temp.shape[0], 1)),
53                               temp.as_numpy_array()))
54                 prob.append(GPU.garray(temp))
55             output=GPU.exp(GPU.dot(prob[-1], weight[-1].T))
56             Z = GPU.sum(output, axis=1).reshape(-1, 1)
57             output = output / Z
58             prob.reverse()
59             return prob, output
60         def weight2GPU():
61             weight = []
62             for rbm in self.rbms:
63                 temp=GPU.garray(hstack((rbm.hidBais ,rbm.W)))
64                 weight.append(temp)
65             return weight

```

```
66     print "start_to_fine_tuning"
67     maxEpoch = 200
68     maxBatch = 300
69     trainingData, trainingLabel = makeBatch(maxBatch)
70     numCase = trainingData.shape[1]
71     weight = weight2GPU()
72     epsilon = 0.999
73     learningRate = 0.1
74     trainingData = GPU.garray(trainingData)
75     trainingLabel = GPU.garray(trainingLabel)
76     for epoch in range(maxEpoch):
77         self.recognize()
78         for i in range(maxBatch):
79             input = trainingData[i]
80             target = trainingLabel[i]
81             prob, output = getAbstract(weight, input)
82             sens = target - output
83             weight.reverse()
84             for W, X, index in zip(weight, prob,
85                                     range(len(weight))):
86                 delta = GPU.dot(sens.T, X)
87                 sens = GPU.dot(sens, W) * X * (1 - X)
88                 sens = sens[:, 1:]
89                 weight[index] += learningRate * delta
90                         /(numCase)
91             weight.reverse()
92             print epoch, "complete!"
93             for rbm, W in zip(self.rbms, weight):
94                 rbm.W = (epsilon * W[:, 1:]).as_numpy_array()
95                 rbm.hidBais = W[:, 0].as_numpy_array()
96             for rbm in (self.rbms):
97                 rbm.saveParameter()
98
```

```
99  def reconstruct(self , numCase=100, trainingData=None):
100     if trainingData == None:
101         trainingData = getBinData()[0]
102         pureRBM = self.rbms[:-1]
103         for i in range(numCase):
104             sample = trainingData[i]
105             abstract = sample
106             for rbm in pureRBM:
107                 abstract = rbm.featureExtract(abstract)
108             pureRBM.reverse()
109             for rbm in pureRBM:
110                 abstract = rbm.reconstruct(abstract)
111             pureRBM.reverse()
112             imageName = 'reconstruct' + str(i)
113             createImage(255 * abstract , imageName ,
114                         showNow = False , trans = False)
115     else:
116         pureRBM = self.rbms[:-1]
117         abstract = trainingData
118         for rbm in pureRBM:
119             abstract = rbm.featureExtract(abstract)
120         pureRBM.reverse()
121         for rbm in pureRBM:
122             abstract = rbm.reconstruct(abstract)
123         scale = np.random.random(abstract.shape)
124         temp = zeros(abstract.shape)
125         temp[less(scale , abstract)] = 1
126         createImage(255 * abstract ,
127                         showNow = False , trans = False)
128     return temp
129
130
131
```

```
132     def recognize(self, testData = None, testLabel = None):
133         if testData == None or testLabel == None:
134             testData = (255 - getData()[2]) / 255.0
135             #testData = getBinData()[2]
136             testLabel = getBinData()[3]
137             numCase = testData.shape[0]
138             maxBatch = 100
139             testData.shape = maxBatch, -1, 784
140             testLabel.shape = maxBatch, -1
141     else:
142         numCase = testData.shape[0] * testData.shape[1]
143         maxBatch = testData.shape[0]
144
145         count = 0
146         for i in range(maxBatch):
147             input = testData[i]
148             output = input
149             for rbm in self.rbms:
150                 output = rbm.featureExtract(output)
151                 guest = argmax(output, axis = 1)
152                 bingo = guest == testLabel[i]
153                 count += sum(bingo)
154         print "correct_rate=", str(100.0 * count / numCase) + "%"
```

附录 B 卷积神经网络源代码

在本附录中给出CIFAR中使用的卷积神经网络源代码，由于MNIST源代码与这里类似，只是构造器有略微的不同，因此我们不打算提供MNIST中卷积网络的代码，读者可经过很少的改动便可将以下的代码改写成训练MNIST数据集的代码。在这几个文件中，Main.py是整个程序的入口。CIFAR.py是工具文件，它提供获取数据集以及绘制图像的功能。FormatLayer.py是最顶层特征图展开成列向量所经过的一个格式转化层。CNNs.py是整个卷积网络的类文件，它能实现网络的前向传播和反向传播。SubsamplingLayer.py是采样层类文件。ConvLayer.py是卷积层类文件。FullConnectLayer.py是全连接层的类文件。可以看到，整个网络的代码已经被我们写成紧耦合的了，我们没有时间将这些代码写成松耦合形式，有兴趣的读者可进行尝试。另外，鉴于版面的问题，我们对代码的格式进行了调整，因此直接拷贝这些代码是不会执行的，你需要再重新调整他们的格式。我们对数据集进行了镜像处理，因此我们总共有10个批次的数据集，进行图像镜像仅仅只是将矩阵翻转，很容易实现，所以在里我们就不贴代码了。

代码 B.1 FormatLayer.py

```
1 from CNNs import *
2 cnn = CNNs(load=False)
3 cnn.train()
4 correctRate = cnn.testing()
5 print correctRate
```

代码 B.2 CIFAR.py

```
1 from numpy import *
2 from scipy import misc
3 import Image
4
5 def unpickle(file):
6     import cPickle
7     fo = open(file, 'rb')
8     dict = cPickle.load(fo)
9     fo.close()
10    return dict['data'], array(dict['labels'])
11
12 def getData(i):
13     fileName = 'data_batch_' + str(i)
14     trainingData, trainingLabel = unpickle(fileName)
15     return trainingData/255.0, trainingLabel
16
17 def getTestData():
18     fileName = 'test_batch'
19     testData, testLabel = unpickle(fileName)
20     return testData/255.0, testLabel
21
22 def createImage(data, imageName = 'temp', mode = 'jpg'):
23     red = data[:1024]
24     green = data[1024:2048]
25     blue = data[2048:]
26     image = array(zip(red, green, blue)).reshape(32,32,3)
27     imageName = imageName + '.' + mode
28     misc.imsave(imageName, image)
```

代码 B.3 CNNs.py

```
1 from ConvLayer import *
2 from SubsamplingLayer import *
3 from FullConnectLayer import *
4 import numpy as np
5 from CIFAR import *
6 from FormatLayer import *
7
8 class CNNs():
9
10    def __init__(self, load=False):
11        self.CNN = [ConvLayer(1, 3, 9, loadParameter=load),
12                    SubsamplingLayer(2),
13                    FormatLayer(6, 9, 14, 14),
14                    FullConnectLayer(7, 1764, 10,
15                                     loadParameter=load)]
16
17    def training(self):
18        for i in range(400):
19            self.trainAepoch()
20            correctRate = self.test()/100.0
21            print 'epoch', i, 'complete',
22            'correct_rate=' , correctRate
23        for layer in self.CNN:
24            layer.saveParameter()
25
26
27    def trainAepoch(self):
28        error = 0
29        for batch in range(10):
30            numCase = 10000
31            trainingData = getData(batch + 1)[0][:numCase]
32            label = getData(batch + 1)[1][:numCase]
```

```

33     trainingLabel = np.zeros((numCase, 10))
34     trainingLabel[[ i  for i  in  range(numCase)] ,
35                           label] = 1
36
37     for i  in  range(numCase):
38         red = trainingData[i][:1024]
39             .reshape(32, 32)
40         green = trainingData[i][1024:2048]
41             .reshape(32, 32)
42         blue = trainingData[i][2048:]
43             .reshape(32, 32)
44         featureMap = [red, green, blue]
45         temp = [featureMap]
46 #fprop
47         for layer  in  self.CNN:
48             featureMap = layer.fprop(featureMap)
49             temp.insert(0, featureMap)
50 # clac error
51         outIn = zip(temp[:-1], temp[1:])
52         guest = temp[0]
53         loseFuncDiffAct = guest
54             - trainingLabel[i].reshape(-1, 1)
55         error += np.dot(loseFuncDiffAct.T,
56                         loseFuncDiffAct)
57         self.CNN.reverse()
58 # bprop
59         for layer, outIn  in  zip(self.CNN[:],
60                               outIn [:]):
61             loseFuncDiffAct = layer.bprop(
62                             loseFuncDiffAct,
63                             outIn [1], outIn [0])
64         self.CNN.reverse()
65     print 'a_batch_complete , _error_= ' , error

```

```
66     def testing(self):
67         testData = getTestData()[0]
68         testLabel = getTestData()[1]
69         numCase = 10000
70         hit = 0
71         for i in range(numCase):
72             red = testData[i][:1024].reshape(32, 32)
73             green = testData[i][1024:2048].reshape(32, 32)
74             blue = testData[i][2048:].reshape(32, 32)
75             feature = [red, green, blue]
76             for layer, layerIndex in zip(self.CNN,
77                                           range(len(self.CNN))):
78                 feature = layer.fprop(feature)
79                 guest = np.argmax(feature)
80                 label = testLabel[i]
81                 if guest == label:
82                     hit += 1
83         return hit
```

代码 B.4 ConvLayer.py

```
1 import numpy as np
2 from scipy.signal import convolve2d
3 from scipy import stats
4
5 class ConvLayer():
6     def __init__(self, layer, inputMapNum, outputMapNum,
7                  kernelSize=(5, 5), loadParameter=False):
8         self.layer = layer
9         self.inputMapNum = inputMapNum
10        self.outputMapNum = outputMapNum
11        self.kernelSize = kernelSize
12        Gaussian = stats.norm(0, 0.0001)
13        self.kernel=[[None for j in range(self.outputMapNum)]
14                     for i in range(self.inputMapNum)]
15        if loadParameter == False:
16            self.kernel=[[np.array(Gaussian.rvs(kernelSize))
17                          for j in range(self.outputMapNum)]
18                          for i in range(self.inputMapNum)]
19            self.bias = np.array([np.random.random()
20                                for i in range(self.outputMapNum)])
21        else:
22            for i in range(self.inputMapNum):
23                for j in range(self.outputMapNum):
24                    fp = open(r'kernel' + str(self.layer)
25                            + ('+' + str(i) + ',' + str(j) + ')', 'rb')
26                    self.kernel[i][j] = np.fromfile(fp,
27                                              dtype=np.double)
28                    .reshape(self.kernelSize)
29            fp.close()
30            fp = open(r'bias' + str(self.layer), 'rb')
31            self.bias = np.fromfile(fp, dtype=np.double)
32            fp.close()
```

```

33         print "conv_layer", self.layer,
34                     "parameters had been loaded."
35         self.learningRate = 0.01
36
37     def saveParameter(self):
38         for i in range(self.inputMapNum):
39             for j in range(self.outputMapNum):
40                 fp = open(r'kernel' + str(self.layer)
41                           +'('+str(i)+','+str(j)+')', 'wb')
42                 fp.write(self.kernel[i][j])
43                 fp.close()
44         fp = open(r'bias' + str(self.layer), 'wb')
45         fp.write(np.array(self.bias))
46         fp.close()
47         print "conv_layer", self.layer,
48                     "parameters had been save."
49
50     def conv(self, data, kernel):
51         featureMap = convolve2d(data, kernel, mode='valid')
52         return featureMap
53
54     def fprop(self, inputMaps):
55         highFeatureMaps = [np.zeros_like(
56                             self.conv(inputMaps[0],
57                                     self.kernel[0][0]))]
58                     for i in range(self.outputMapNum)]
59
60         for inputMap, kernel_i in zip(inputMaps, self.kernel):
61             outputMaps = [self.conv(inputMap, kernel_j)
62                           for kernel_j in kernel_i]
63             for i, outputMap in zip(range(self.outputMapNum),
64                                    outputMaps):
65                 highFeatureMaps[i] += outputMap

```

```

66
67     for i in range(self.outputMapNum):
68         highFeatureMaps[i] = 1.0/(1
69                         + np.exp(-highFeatureMaps[i]
70                         - self.bias[i]))
71
72
73     def bprop(self, lossDiffActMap, inputMap, outputMap):
74         def rot180(Matrix):
75             return np.rot90(Matrix, 2)
76
77         actDiffNetMap = [output * (1 - output)
78                         for output in outputMap]
79
80         sensMap = [lossDiffAct * actDiffNet
81                         for lossDiffAct, actDiffNet in
82                         zip(lossDiffActMap, actDiffNetMap)]
83
84         deltaKernel=[[None for j in range(self.outputMapNum)]
85                         for i in range(self.inputMapNum)]
86
87         for i in range(self.inputMapNum):
88             for j in range(self.outputMapNum):
89                 deltaKernel[i][j] = rot180(self.conv(
90                                 inputMap[i],
91                                 rot180(sensMap[j])))
92                 self.kernel[i][j] -= self.learningRate
93                                 * deltaKernel[i][j]
94
95         for j in range(self.outputMapNum):
96             self.bias[j] -= self.learningRate
97                                 * np.sum(sensMap[j])
98

```

```
99     ans = [np.zeros_like(inputMap[0])
100         for i in range(self.inputMapNum)]
101     for i in range(self.inputMapNum):
102         for j in range(self.outputMapNum):
103             ans[i] += convolve2d(sensMap[j],
104                                 rot180(self.kernel[i][j]),
105                                 mode='full')
106
return ans
```

代码 B.5 SubsamplingLayer.py

```
1 import numpy as np
2 from scipy.signal import convolve2d
3 from scipy.linalg import kron
4
5 class Subsampling():
6
7     def __init__(self, layer):
8         self.layer = layer
9         self.kernel = 0.25 * np.ones((2, 2))
10
11    def subsampling(self, data):
12        featureMap = convolve2d(data, self.kernel,
13                                mode='valid')
14        return featureMap[::2, ::2]
15
16    def saveParameter(self):
17        print 'subsampleint_layer_has_no_parameter_to_save'
18
19    def fprop(self, lowFeatureMaps):
20        highFeatureMaps = [self.subsampling(data)
21                            for data in lowFeatureMaps]
22        return highFeatureMaps
23
24    def bprop(self, lossDiffActMap, inputMap, outputMap):
25        I2x2 = 0.25 * np.ones((2, 2))
26        return [kron(sens, I2x2) for sens in lossDiffActMap]
```

代码 B.6 FormatLayer.py

```
1 import numpy as np
2 class FormatLayer():
3     def __init__(self, layer, mapNum, mapRow, mapCol):
4         self.mapNum = mapNum
5         self.mapRow = mapRow
6         self.mapCol = mapCol
7         self.layer = layer
8
9     def fprop(self, data):
10        return np.array(data).reshape((-1, 1))
11
12    def saveParameter(self):
13        print 'format_layer_has_no_parameter_to_save'
14
15    def bprop(self, lossDiffAct, feedIn, output):
16        lossDiffAct.shape = self.mapNum, self.mapRow,
17                                         self.mapCol
18        return [maps for maps in lossDiffAct]
```

代码 B.7 FullConnectLayer.py

```
1 from scipy import stats
2 import numpy as np
3
4 class FullConnectLayer():
5
6     def __init__(self, layer, visNum, hidNum,
7                  loadParameter = False):
8         self.layer = layer
9         Gaussian = stats.norm(0, 0.01)
10        if loadParameter == False:
11            self.weight = np.array(Gaussian.rvs(
12                                    (hidNum, visNum)))
13            self.bias = np.zeros((hidNum, 1))
14        else:
15            fp = open(r'weight' + str(self.layer), 'rb')
16            self.weight = np.fromfile(fp, dtype = np.double)
17            .reshape(hidNum, visNum)
18            fp.close()
19            fp = open(r'bias' + str(self.layer), 'rb')
20            self.bias = np.fromfile(fp, dtype = np.double)
21            .reshape(hidNum, 1)
22            fp.close()
23            print "full_connect_layer", self.layer,
24            "parameters had been loaded."
25        self.learningRate = 0.001
26
27    def saveParameter(self):
28        fp = open(r'weight' + str(self.layer), 'wb')
29        fp.write(self.weight)
30        fp.close()
31        fp = open(r'bias' + str(self.layer), 'wb')
32        fp.write(self.bias)
```

```
33     fp.close()
34     print "full_connect_layer", self.layer,
35             "parameters had been save."
36
37     def fprop(self, data):
38         net = np.dot(self.weight, data) + self.bias
39         feature = 1.0/(1 + np.exp(-net))
40         return feature
41
42
43     def bprop(self, lossDiffAct, feedIn, output):
44         actDiffNet = output * (1 - output)
45         sens = lossDiffAct * actDiffNet
46         deltaWeight = np.dot(sens, feedIn.T)
47         self.weight -= self.learningRate * deltaWeight
48         self.bias -= self.learningRate * sens
49         return np.dot(self.weight.T, sens)
```