《人工智能》课程系列

人工神经网络基础*

武汉纺织大学数学与计算机学院

杜小勤

2018/12/04

Contents

*本系列文档属于讲义性质，仅用于学习目的。Last updated on: November 18, 2020。

1 概述

人工神经网络（Artificial Neural Network，ANN）是对生物神经网络极其简单的抽象。与生物神经网络类似，人工神经网络的基本单元也是神经元（Neuron）。就整体而言，虽然由许多神经元组成的人工神经网络还远远未达到生物神经网络（尤其是人脑神经网络）的水平，但是在一些特定的领域，例如计算机视觉、语言翻译及棋类对弈（计算机国际象棋、计算机围棋等）领域，近年来兴起的深度神经网络已经达到或超越了人类的顶尖水平。深度神经网络取得的成就，为人类向强人工智能领域进军，吹响了出发的号角[1]。

神经网络的相关研究始于 19 世纪末和 20 世纪初，它源于物理学、心理学和神经生理学的跨学科研究。这些早期研究主要集中于有关学习、视觉和条件反射等一般理论，并没有包含有关神经元工作机制的数学模型。此阶段的主要代表人物有 Herman Von Helmholts、Ernst Mach 和 Ivan Pavlov 等。

人工神经网络的研究工作始于 20 世纪 40 年代，Warren McCulloch 和 Walter Pitts 从原理上证明了人工神经网络可以计算任何算术和逻辑函数。随后，Donald Hebb 在经典的条件反射行为（由 Pavlov 发现）与神经元之间建立了联系，并提出了 Hebb 学习规则。此后，该学习机制一直被用于训练人工神经网络。

在 20 世纪 50 年代，Frank Rosenblatt 提出了感知机（Perceptron）网络及联想学习规则。随后，与同事一道构造了一个感知机网络，并公开展示了该网络的模式识别能力。同时，Bernard Widrow 和 Ted Hoff 也引入了一个新的学习算法（Widrow-Hoff 学习算法）用于训练自适应线性神经网络，它在结构与功能上类似于 Rosenblatt 的感知机。

Marvin Minsky 和 Symour Papert 在 1969 年出版了《Perceptrons》一书。书中详细论述了感知机的局限性，并且指出了克服它们的方向。不幸的是，该书对神经网络的发展显示出悲观的态度。许多人受到了该观点的影响，同时也由于当时还缺乏功能强大的计算机来开展各种实验，从而导致许多研究者纷纷离开了神

[1]计算机围棋（Computer Go），曾经一度被认为是人工智能领域中的“圣杯”——该领域的研究者认为，一旦攻克了计算机围棋，也意味着人类实现了强人工智能。但是，即便 AlphaGo 在 2016 年战胜了人类 9 段棋手李世石并相继进化出远超人类水平的 AlphaGo Zero 和 AlphaZero，强人工智能也远未实现，离实际目标还有相当大的距离。

经网络研究领域。从此，神经网络领域的研究停滞了十几年[2]。

到了 80 年代，随着个人计算机和工作站的出现以及新概念的出现，人们对神经网络的研究热情再度点燃。有 2 个重要的事件对神经网络的复兴起到了重要的作用：使用统计原理解释了某些类型的递归网络（物理学家 John Hopfield 的研究论文 “Neural networks and physical systems with emergent collective computational abilities” 论述了这些核心思想）、几个研究者各自独立地研究出了用于训练多层感知机（Multi-Layer Perceptron，MLP）的反传（Backpropagation，BP）算法。BP 算法有力地回答了 Minsky 和 Papert 对神经网络模型的质疑。此后，出现了神经网络研究的第 2 次热潮。

1989 年，Robert Hecht-Nielsen 证明了 MLP 的逼近定理，即对于任何闭区间内的一个连续函数，都可以使用含 1 个隐藏层的 BP 网络来逼近。该定理的发现，极大地鼓舞了神经网络的研究人员。

同年，Yann LeCun 发明了卷积神经网络 LeNet[3]，将其用于数字识别，并且取得了较好的成绩。不过在当时，LeNet 并没有引起足够的重视。

然而，好景不长，在 1991 年，BP 算法被指出存在梯度消失的问题，即在误差梯度反向传递的过程中，后层梯度以乘积的方式传递到前层，但是由于 Sigmoid 激活函数的饱和特性[4]，后层的梯度本身就很小，在误差梯度反传到前层时几乎为 0，因而无法有效地训练前层的权值。

另外，人工神经网络自发明以来，也一直缺少严格的数学理论，种种因素就导致了该领域第 2 次研究热潮的退去。即便在 1997 年，LSTM（Long Short-Term Memory）模型被发明，虽然该模型在序列建模上的特性非常突出，但由于正处于人工神经网络研究的下坡期，也没有能够引起足够的重视。

在此期间，其它各种统计学习方法也大量出现，下面将按照出现年代，逐一论述典型的方法。其中的一些方法，与人工神经网络交织在一起，互相促进。

1986 年，决策树方法被提出，很快 ID3，ID4，CART 等改进的决策树方法也相继出现。到目前为止，它们仍然是常用的机器学习方法，该方法也是符号学习方法的代表。

1995 年，线性 SVM（Support Vector Machine）被统计学家 Vladimir Vapnik

[2]即便如此，20 世纪 70 年代，Teuvo Kohonen、James Anderson、Stephen Grossberg 等人在新型神经网络与自组织网络领域里仍然取得了一些重要的研究成果。

[3]它是著名的 CNN(Convolutional Neural Network) 网络的鼻祖。

[4]在函数的两端，梯度接近于 0。

提出。该方法的特点有两个：由非常完美的数学理论推导而来（统计学与凸优化等）、符合人的直观感受（最大间隔），SVM 在线性分类的问题上取得了当时的最好成绩。此后，SVM 方法被大量地应用于图象与语音识别领域，并且都取到了很好的效果，该方法成为 90 年代较为流行的算法。关于 SVM 方法，有必要稍加描述一下。原因在于，自 SVM 方法提出以来，它一直被视为人工神经网络的一个强劲对手。特别是在 20 世纪 90 年代，其风头甚至盖过当时正处于第二次低潮的人工神经网络。SVM 与神经网络的第三次变革与发展有着千丝万缕的联系。SVM 方法，首先也是从线性可分问题入手，然后扩展到线性不可分问题。对于线性不可分的情况，SVM 通过使用非线性映射，将低维输入空间的线性不可分样本转化到高维的特征空间，使其线性可分。90 年代，在贝尔实验室里，Yann LeCun 和 Vladimir Vapnik 常常就神经网络和 SVM 两种技术展开深入的讨论，为相关领域的发展奠定了基础。

1997 年，另一种统计学习方法 AdaBoost 被提出，该方法是 PAC（Probably Approximately Correct）理论在机器学习实践上的代表，也催生了集成方法。该方法通过一系列弱分类器的集成，达到了强分类器的效果。

2000 年，核函数支持向量机（Kernel SVM）被提出，核化的 SVM 通过 Kernel 函数将原空间线性不可分的问题，映射成高维空间的线性可分问题，从而成功地解决了非线性分类问题。

2001 年，随机森林被提出，这是集成方法的另一代表，该方法的理论扎实，比 AdaBoost 方法具有更好的抑制过拟合作用。同年，一种新的统一框架——图模型被提出，该方法试图统一机器学习中的几种方法，例如朴素贝叶斯、SVM、隐马尔可夫模型等，为它们提供一个统一的理论框架。

神经网络在第 2 次研究热潮退去了若干年之后，时间到了 2006 年，3 篇重要论文的发表，宣告了神经网络的第 3 次研究热潮以及深度学习时代的到来。这 3 篇重要论文是 Geoffrey Hinton 等人的 “A fast learning algorithm for deep belief nets”、Yoshua Bengio 等人的 “Greedy LayerWise Training of Deep Networks”、Yann LeCun 等人的 “Efficient Learning of Sparse Representations with an Energy-Based Model”。

在深度学习的开创性论文中，Geoffrey Hinton 提出了深度信念网络（Deep Belief Networks，DBNs）训练中梯度消失问题的解决方案：首先使用无监督的预训练对权值进行初始化（无监督学习的 RBMs），然后再使用有监督的训练对权值

进行微调。其主要思想是，先通过无监督的方法学习训练数据中的结构（自动编码器），然后在此基础上进行有监督的训练微调。

Bengio 等人的论文探讨和对比了 RBMs 和 Auto-Encoders 方法，Yann LeCun 等人的论文在一个卷积网络架构上使用了稀疏的 Auto-Encoders(类似于稀疏编码)。

总之，初期的深度学习方法，使用如下的方法解决了深度网络的训练问题：使用无监督的预训练方法，对深度网络中的每一层单独进行训练；将每一层堆叠在一起，形成一个层次关系；使用有监督的统一训练来微调所有层的权值。此后，大量的深度学习论文被发表，并引发了一股应用热潮。

2011 年，ReLU 激活函数被提出，该激活函数能够有效地抑制梯度消失的问题。同年，微软首次将深度学习应用于语音识别，取得了重大突破。

2012 年，Hinton 课题组为了证明深度学习的潜力，首次参加了 ImageNet 图像识别比赛，他们构建的 CNN 网络 AlexNet 一举夺得了冠军，且以优秀的分类性能碾压了第二名（SVM 方法）。也正是由于这次比赛，CNN 吸引到了众多研究者的注意。

AlexNet 的特点如下：

- 首次采用 ReLU 激活函数，极大地加快了训练的收敛速度，并且从根本上解决了梯度消失的问题；
- 由于 ReLU 方法可以很好地抑制梯度消失问题，AlexNet 抛弃了传统的“预训练 + 微调”方法，完全采用有监督的训练方式。也正因为如此，DL 的主流学习方法也就变成了一种纯粹的监督学习方法；
- 扩展了 LeNet5 结构，添加了 Dropout 层，以抑制过拟合，LRN 层增强了泛化能力/减小过拟合；
- 首次采用 GPU 对计算进行加速；

这次事件标志着深度学习时代的正式到来与真正的崛起，同时也标志着第 3 次神经网络研究热潮的到来！

目前，深度神经网络为许多任务提供了最好的解决方案，例如图象识别、语音识别、自然语言处理、计算机游戏的玩法及内容生成、计算机围棋等。这些任

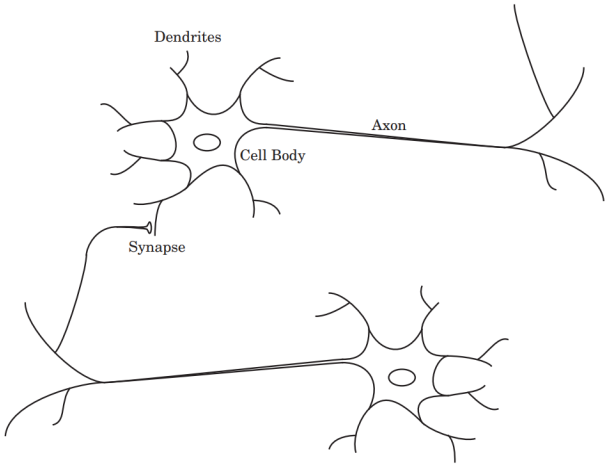

图 2-1: 2 个生物神经元模型

务有一个共同的特点——它们采用传统的方法很难得到有效的解决，也难以采用手工的方式构建合适的数学模型并通过手工编程的方式加以解决。

2 生物神经网络的启示

人脑由大量高度互联的神经元（约有 10^{11} 个，每个神经元约有 10^4 个连接）组成。这些神经元由 3 部分组成，分别是树突、细胞体和轴突。

树突是树状的神经纤维接收网络，它将电信号传送到细胞体。细胞体对接收到的输入信号进行整合并进行阈值处理。轴突是单根长纤维，它把细胞体的信号输出到其它神经元。一个神经元的轴突和另一个神经元的树突相结合的点，被称为突触。

神经元的排列和突触的强度（由复杂的化学过程决定）确定了神经网络的功能。图2-1展示了 2 个生物神经元模型的示意图。

一般认为，一些神经网络的结构与功能是与生俱来的，而另一些神经网络的结构与功能则是在后天学习的过程中形成的。

在学习的过程中，神经元之间的连接可以加强，也可以减弱，甚至可以建立新的连接，或可以让已有的连接消失。这意味着，神经网络的结构与功能在整个生命周期内可以不断地进行演化。例如，新记忆的形成就是通过突触强度的改变而实现的；如果在某一段关键时期，禁止一只小猫使用它的一只眼睛，则该眼在以后就很难形成正常的视力。

图 3-2: 手写体识别任务

人工神经网络与生物神经网络存在很大的差别，前者是对后者的一种极大简化[5]。但是，两者之间具有如下 2 个关键的相似点：

- 两种网络都是可计算单元（神经元）的高度互联；
- 神经元之间的连接及强度决定了网络具有的功能；

从面向问题的角度来看，人工神经网络的目标就是要确定合适的面向问题的神经元互联结构及其连接强度，以有效的方式合理地解决问题。

3 任务示例：手写体识别

在实际应用中，一个典型的识别任务是手写体数字的识别，如图3-2所示。

在该图中，除了极少数手写体数字因书写原因而造成识别困难外，绝大多数的手写体数字对于我们人类而言，可以被毫不费力地识别出。主要原因在于，人类大脑有 V1、V2、V3、V4、V5 等视觉皮层，它们由低到高逐级形成一个层级系统，能够逐级地处理更复杂的视觉信息。例如，V1 包含 1.4 亿个神经元，各层之间的连接达几十亿个。值得注意的是，人类的视觉系统已经演化了几百万年！

深度卷积神经网络（Deep Convolutional Neural Network，DCNN）是前向神经网络的一种类型，它是受 Hubel 和 Wiesel 对猫视觉皮层的电生理研究的启发

[5]值得注意的是，虽然生物神经元要比电子电路慢很多（$10^{-3} : 10^{-9}$），但是由于生物神经网络具有巨大的、天然的并行性，生物神经元完成任务的速度非常快，在许多任务中，丝毫不逊色于人工神经网络，甚至更加出色。

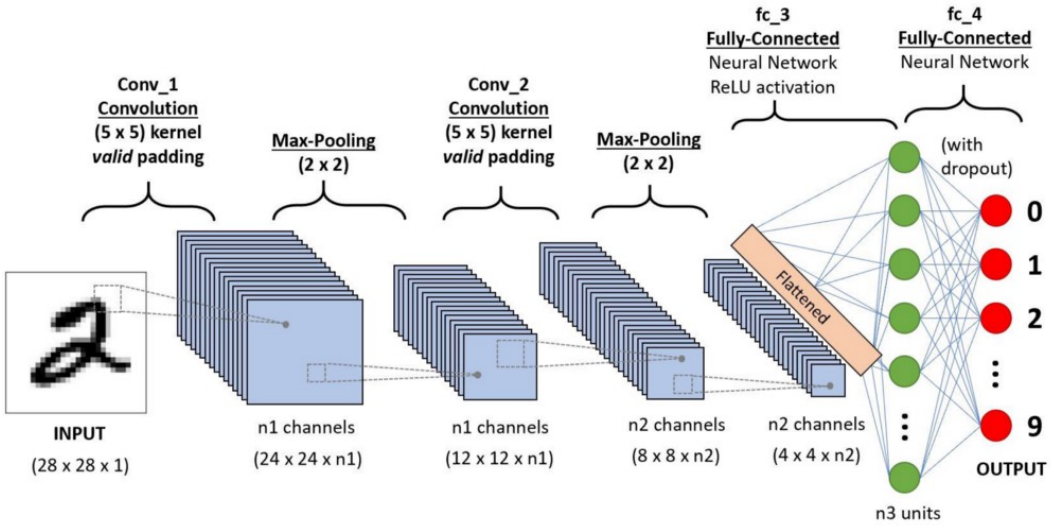

图 3-3: 一个典型的 DCNN 结构

而提出的一种网络模型，其网络模型成功地模拟了大脑视觉层级系统的主要功能。Yann LeCun 最早将 DCNN 应用于手写数字识别中，并成功地进行了商业化应用。近年来，深度卷积神经网络在多个应用领域均取得了前所未有的成功，例如在语音识别、人脸识别、通用物体识别、运动分析、自然语言处理甚至脑电波的分析等领域均有突破。图3-3展示了一个典型的 DCNN 网络结构。

针对手写体数字识别任务，如果采用传统的手工硬编码（Hard-Coded）的方式来处理，如何处理？更具体地说，如何识别 0、1、2、...、9？如何确定性地、精确地表达“识别算法”？

虽然我们人类一眼就能够识别出绝大多数的手写体数字，但是却不知道如何具体而详尽地描述出识别任务的具体过程与步骤，以至于无法提出一种合适的硬编码解决方案来实现手写体数字的识别。这意味着，采用传统的方式处理该识别任务是极其困难的。

但是，这并不意味着，没有合适的办法来解决手写体的数字识别任务。一种行之有效的方法是，遵循“简单的算法处理复杂的数据能够获得所需功能”这一重要思想[6]。显然，在大数据时代，飞速发展的计算机软硬件，已经为这一算法新思维提供了有力的保障。

[6]与之形成鲜明对比的是，传统的方法遵循的是“复杂的算法处理简单的数据能够获得所需功能”这一思想。

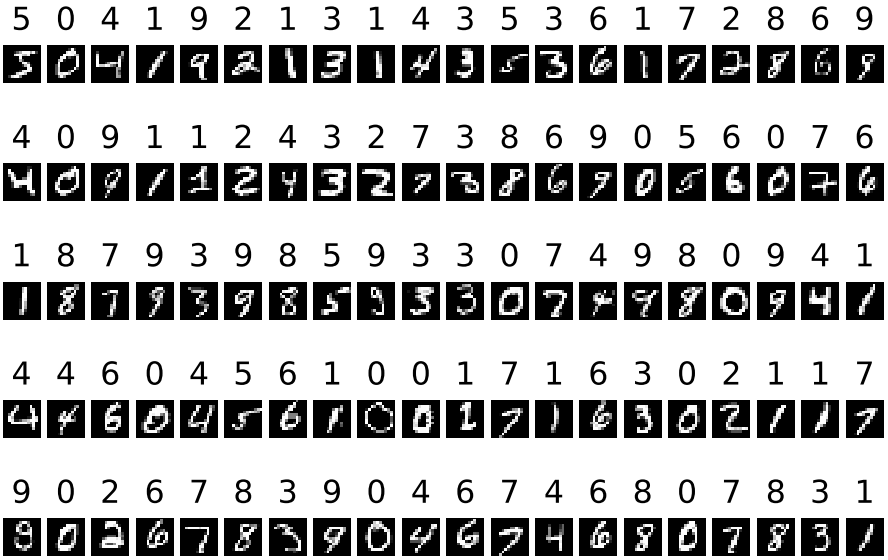

图 3-4: 手写体识别程序的训练数据样例

经过研究者的不断探索，现在已经有多种算法能够很好地解决手写体的数字识别问题，例如支持向量机、（浅）神经网络与深度神经网络等。

神经网络可以从大量的手写体数字数据（被称为是“训练数据”）中学习，并以权值的形式保存学习结果，从而可以“较容易地”识别出手写体数字——不仅仅是训练数据中的手写体数字，还包括其它未见的手写体数字[7]。如图3-4所示，展示了一些手写体数字的训练数据样例，其中偶数行表示 28×28 输入图像数据，奇数行表示对应的标签数据。

本质上，神经网络也是一种基于统计的学习方法，它能够从训练数据中自动地“推理”出手写体数字的特征。一般情况下，如果增加训练数据，还可以进一步提高识别的精度。一个非常简单的 3 层神经网络，其手写体数字的识别精度可以达到 96%；如果再结合其它抑制过拟合的技术（例如正则化等），就可以达到 99% 的识别精度，完全能够满足商业化的需求。目前，商业化的手写体数字识别程序已经被用于银行的支票处理、邮局的邮编处理等领域中。

[7]这表明神经网络还具有一定的泛化能力。对于训练者或学习者而言，这一能力是非常重要的。

4 前馈神经网络

神经网络是由神经元按照一定的方式组合在一起的，神经元之间具有某种连接。神经网络种类较多，下面将以“前馈神经网络”（Feedforward Neural Networks，FNN）为例进行介绍。

前馈神经网络是一种最简单的神经网络，各神经元分层排列。每个神经元只与前一层的神经元相连。将前一层的输出作为输入，经计算后输出给下一层，各层之间没有反馈。该网络的研究始于 20 世纪 60 年代，目前，其理论研究和实际应用都达到了很高的水平。

常见的前馈神经网络包括：

- 感知机

  感知机（Perceptron）是最简单的前馈网络，主要用于模式分类，也可用于基于模式分类的学习控制和多模态控制中。感知机网络可分为单层和多层感知机（Multi-Layer Perceptron，MLP）网络；

- BP 网络

  BP（Backpropagation）网络是指权值的调整采用了反向传播（Back Propagation）学习算法的前馈网络。与感知机不同的是，BP 网络的神经元激活函数采用了 S 形函数（Sigmoid 函数）。因此，它的输出值是 $0 \sim 1$ 之间的连续量，可实现从输入到输出的任意非线性映射；

- RBF 网络

  RBF（Radial Basis Function）网络是指隐藏层的神经元是由 RBF 神经元组成的前馈网络。RBF 神经元是指神经元的激活函数为 RBF（Radial Basis Function，径向基函数）的神经元。典型的 RBF 网络由三层组成：一个输入层，一个或多个由 RBF 神经元组成的 RBF 层（隐藏层），一个由线性神经元组成的输出层；

一个简单的前馈神经网络结构如图4-5所示[8]，该网络结构包含 1 个输入层、1 个隐藏层和 1 个输出层。而一个稍复杂的神经网络结构如图4-6所示，该网络结构包含 1 个输入层、2 个隐藏层和 1 个输出层。

[8]注意，此处将输入层也看作单独的一层。而在有些文献中，没有这样处理。

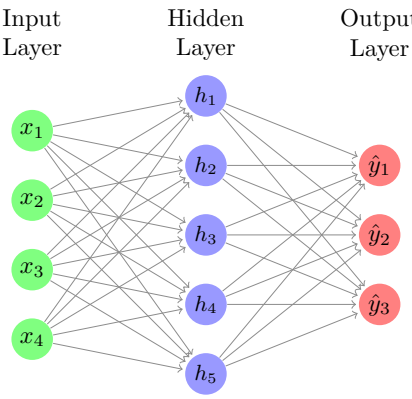

图 4-5: 一个简单的神经网络结构

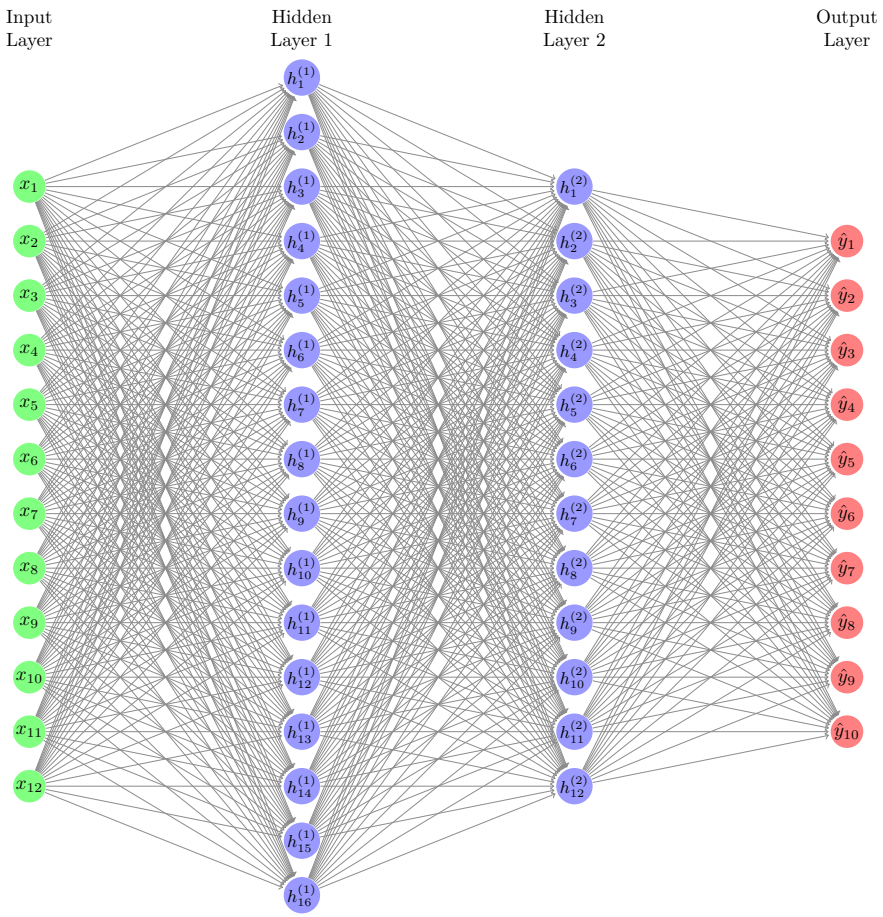

图 4-6: 一个稍复杂的神经网络结构

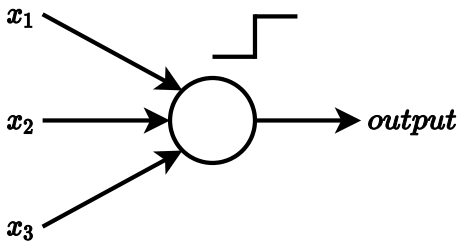

图 4-7: Perceptron 的结构示例

4.1 神经元

在神经网络发展的早期，存在 2 种常见的神经元，它们分别是 Perceptron 和 Sigmoid（S）型神经元，两者的主要区别在于激活函数的不同。实际上，当前还存在着一些被广泛应用于深度神经网络中的神经元，例如 ReLU 神经元等。

4.1.1 Perceptron

Perceptron 神经元由 Frank Rosenblatt 于 20 世纪 50 年代提出，属于神经网络发展初期提出的一种神经元。我们可以通过理解 Perceptron 神经元，来更好地理解其它类型的神经元。特别需要注意 Perceptron 神经元的学习规则与局限性。

一个 Perceptron 神经元的基本结构如图4-7所示。注意，Perceptron 神经元采用阈值函数或阶梯函数——当神经元的权值输出值[9]超过阈值时，神经元的输出值为 1；否则，输出值为 0。它的激活函数曲线如图4-8所示。

在图4-7中，神经元总共有三个输入：x_1、x_2、x_3，取值为 0 或 1。与这三个输入对应的是三个权值（图中未标出）：w_1、w_2、w_3，用来表示输入的重要程度。该 Perceptron 神经元将按如下公式产生输出：

$$output = \begin{cases} 0 & if \ \sum_j w_j x_j \leq threshold \\ 1 & if \ \sum_j w_j x_j > threshold \end{cases} \tag{1}$$

为便于理解神经元的输入、权值与阈值之间的关系，我们将赋予它们一定的含义。在此意义下，Perceptron 神经元就能够为人们提供某种决策。此时，该神经元就是一个非常简单的决策模型！

[9]指没有经过激活函数处理的输出值，下文将使用符号 z 表示该值。在有些文献中，它被称为净输出值。

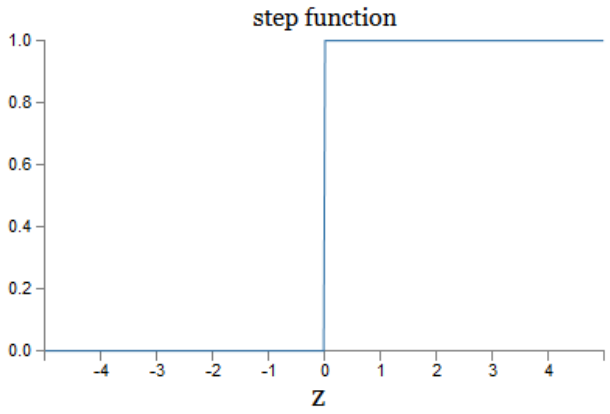

图 4-8: 阶梯函数的曲线图

假设某人要依据如下 3 个因素来决定周末自己是否参加一个节日聚会，这 3 个因素及相应的权值（表示因素的相对重要性，权值越大，因素越重要）如下：

- 天气是否好：x_1，w_1
- 是否有人陪伴：x_2，w_2
- 交通是否便利：x_3，w_3

进一步地，假设模型的权值与阈值参数分别是：$w_1 = 6$，$w_2 = 2$，$w_3 = 2$，$threshold = 5$。可以看出，因素 x_1，即天气因素，它的权值最大，表明决策者最看重该因素；另外 2 个因素的权值一样，表明决策者对它们的重视程度一样，但重视程度要低于天气因素。

一旦确定了模型参数，也就意味着确定了某种类型的决策者——该决策者最在意天气是否好，天气越好，去参加节日聚会的可能性越大。当然，参加与否还要取决于阈值参数。

现在，决策者就可以根据这 3 个因素来做出决策。当输入数据是：$x_1 = 1$，$x_2 = 0$，$x_3 = 0$，有 $output = 1(\sum_j w_j x_j = 6 > threshold)$。结果表明，正如我们所预料的那样，只要天气好，该决策者就会参加节日聚会。

在该神经元的模型参数中，权值参数的意义十分明确，而阈值参数的意义就稍微隐晦些。实际上，阈值参数用来表示决策者参加节日聚会的渴望程度：值越小，越渴望。从神经元的角度来看，值越小，越容易被激活。例如，设 $threshold = 3$，这意味着，在新的模型下，不管天气如何，只要后 2 个条件满足，决策者也会参加节日聚会。

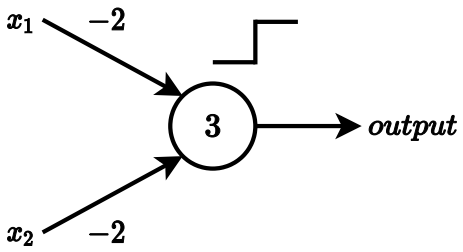

图 4-9: Perceptrons 的 NAND 实现

上述分析表明，虽然单个 Perceptron 神经元的结构与计算都十分简单，但是却能够做出一个有意义的决策。

实际上，从数学的角度看，它的计算结果也是有意义的——它可以实现基本的逻辑函数。

例如，如图4-9所示，该 Perceptron 神经元可以实现 NAND 逻辑：

$00 \to 1$：$(-2)\cdot 0 + (-2)\cdot 0 + 3 = 3 > 0$

$01 \to 1$：$(-2)\cdot 0 + (-2)\cdot 1 + 3 = 1 > 0$

$10 \to 1$：$(-2)\cdot 1 + (-2)\cdot 0 + 3 = 1 > 0$

$11 \to 0$：$(-2)\cdot 1 + (-2)\cdot 1 + 3 = -1 < 0$

如果我们将多个 Perceptron 神经元按照某种方式连接起来，形成一个更加复杂的神经网络，那么我们就可以使用该网络来计算任意的逻辑函数。

如图4-5所示，图中的每一个神经元都可以执行简单的决策，后层神经元的输入是前层神经元的输出，它们在更抽象的层次上进行决策。因此，该神经网络可以执行更加复杂的决策。

为便于讨论，对公式（1）进行简化：

$$output = \begin{cases} 0 & if\ W \cdot X + b \leq 0 \\ 1 & if\ W \cdot X + b > 0 \end{cases} \tag{2}$$

其中，W 是权值向量，X 是输入向量，$W \cdot X$ 是向量点积；b 是偏置（bias），$b = -threshold$。W 的每个分量依然表示对应的输入数据分量的重要程度，值越大，越重要。b 表示神经元被激活的难易程度，值越大，越易激活，值越小，越不易被激活（与公式（1）中的 $threshold$ 刚好相反）。

在上述例子中，权值与偏置都是事先给定的。在实际应用中，需要从一组实例（样本）数据中学习到合适的权值与偏置，即它们需要依据某种学习规则进行

调整，使得网络的实际输出与期望（目标）输出越来越接近。这一过程被称为神经网络的训练。

经过训练的神经网络，能够对已知的输入产生正确的输出，这种能力被称为神经网络的拟合（逼近）能力。然而，从某种意义上说，真正更为重要的是，对于那些未知的输入，在理想情况下，神经网络也应该能够产生正确的输出，这种能力被称为神经网络的泛化能力。这意味着，从某种角度来说，我们对神经网络泛化能力的渴望，比对拟合能力的渴望更加急迫——如果对训练样本的简单记忆就能够解决我们所面临的问题，那么神经网络等机器学习算法也就没有存在的价值了！

关于 Perceptron 神经元及神经网络的权值与偏置的学习规则，请参考文献 [1] 的第 4 章“感知机学习规则”，在此不再赘述[10]。

由 Perceptron 神经元构成的神经网络（被称为感知机），只能解决简单的线性可分问题。然而，在实际应用中，许多问题并非是线性可分的，一个典型的实例是 XOR 逻辑。

在某种程度上说，感知机的这种缺陷导致了 20 世纪 70 年代人们对神经网络研究兴趣的衰退。

4.1.2 Sigmoid

Perceptron 神经元的主要问题在于，它的输出是非连续非平滑的（在 0 和 1 之间跳变），这对于神经网络的学习来讲是一个重要的缺陷，因为神经网络的学习是一个对权值与偏置进行逐步调整并逐步反映到输出的过程。

神经网络所期待的是，输入上微小的变化将引起输出上微小的变化，而不是跳变，因为跳变将让神经网络的学习与输出变得非常不可控。

在神经网络的学习或训练过程中，需要获取实际输出与理想输出之间的误差，并将误差反传用于调整所有的权值与偏置。因此，在理想的情况下，我们希望，它们的变化是连续平滑的，而不是突变的。

如图4-10所示，展示了神经网络中权值与输出之间的理想关系。

假设采用 Perceptron 作为手写体数字识别系统的神经元，那么在学习或训练期间，随着权值与偏置的调整，神经元的输出有可能会在 0 和 1 之间跳变。导致的结果就是，系统的识别调整不是微调的，也不是稳定的，而是跳变的。例如，假

[10]无论从公式的形式上看，还是从公式的意义来看，都是反传算法的特殊情形，都遵循着同样的误差调整（学习）规则。

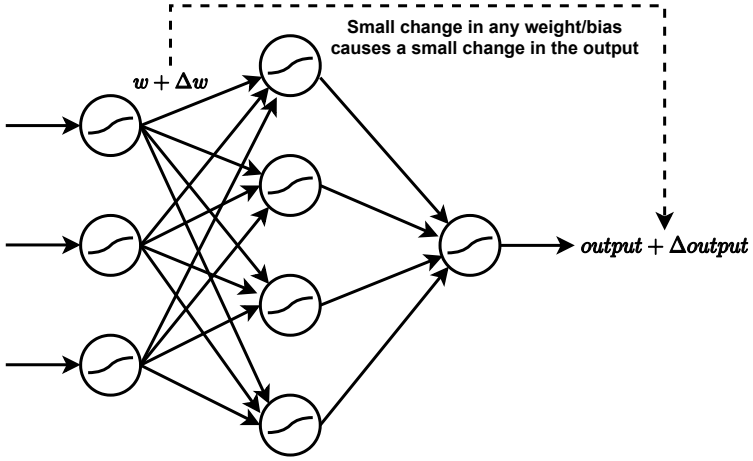

图 4-10: 神经网络权值调整的作用

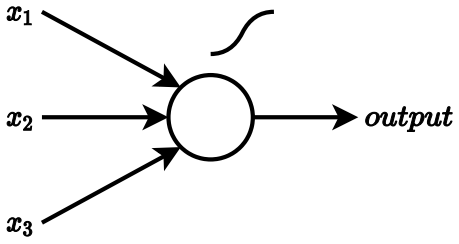

图 4-11: Sigmoid 的结构示例

定在某一时刻，系统误把“8”当作“9”，并假定经过权值与偏置的调整，这种错误已经得到解决。但是，由于 Perceptrons 神经元输出的跳变性，有可能导致其它已得到正确识别的数字也将出现识别问题。

下面引入被称为 Sigmoid 的神经元，如图4-11所示。其名称来源于该神经元所使用的激活函数 Sigmoid。

在结构上，Sigmoid 神经元与 Perceptron 神经元没有什么不同。但是，两者的输入数据与输出数据的范围不一样。主要原因在于，Sigmoid 神经元的激活函数为：

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{3}$$

其中，$z = \sum_j w_j \cdot x_j + b$。激活函数 Sigmoid 的曲线如图4-12所示。

从 Sigmoid 函数的曲线图可以看出，当 z“很大” 或 “很小” 时，两类神经元的行为是类似的。但是，当 z 适中时，两者的行为有很大的不同。其主要的区别来

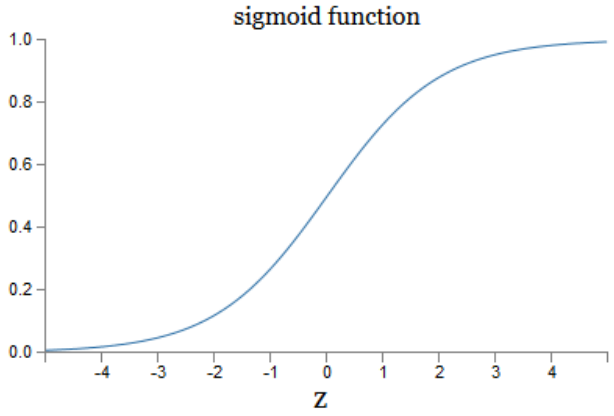

图 4-12: Sigmoid 的曲线图

自于 Sigmoid 函数与阶梯函数之间的不同——Sigmoid 函数是连续平滑的，而阶梯函数是跳跃的，非连续的。因此，对于 Sigmoid 神经元而言，其权值与偏置上微小的变化将引起输出上微小的变化，即：

$$\Delta output \approx \sum_j \frac{\partial output}{\partial w_j}\Delta w_j + \frac{\partial output}{\partial b}\Delta b \tag{4}$$

上式表明，$\Delta output$ 是 Δw_j 和 Δb 的线性函数。这种特性正是我们所需要的。

回到之前的任务示例，即手写体数字识别问题。我们可以按如下方式构建一个前馈神经网络，所使用的神经元为 Sigmoid 神经元：

- 输入层：神经元个数为 $28 \times 28 = 784$（手写体数字图像的点阵：1 表示黑色像素点，0 表示白色像素点）；
- 隐藏层：神经元个数为 15 或 30；
- 输出层：神经元个数为 10，输出值 1000000000 表示数字 0，输出值 0100000000 表示数字 1，依次类推[11]；

该前馈神经网络的示意图如图4-13所示。

训练神经网络的数据使用 MNIST 数据集，其中，50,000 幅图像，作为训练数据（Training data），用于训练；10,000 幅图像，作为测试数据（Testing data），用于测试网络的泛化性能；10,000 幅图像，作为验证数据（Validation data），用

[11]这种表示方法被称为 One-Hot 表示法，在神经网络领域很常见。

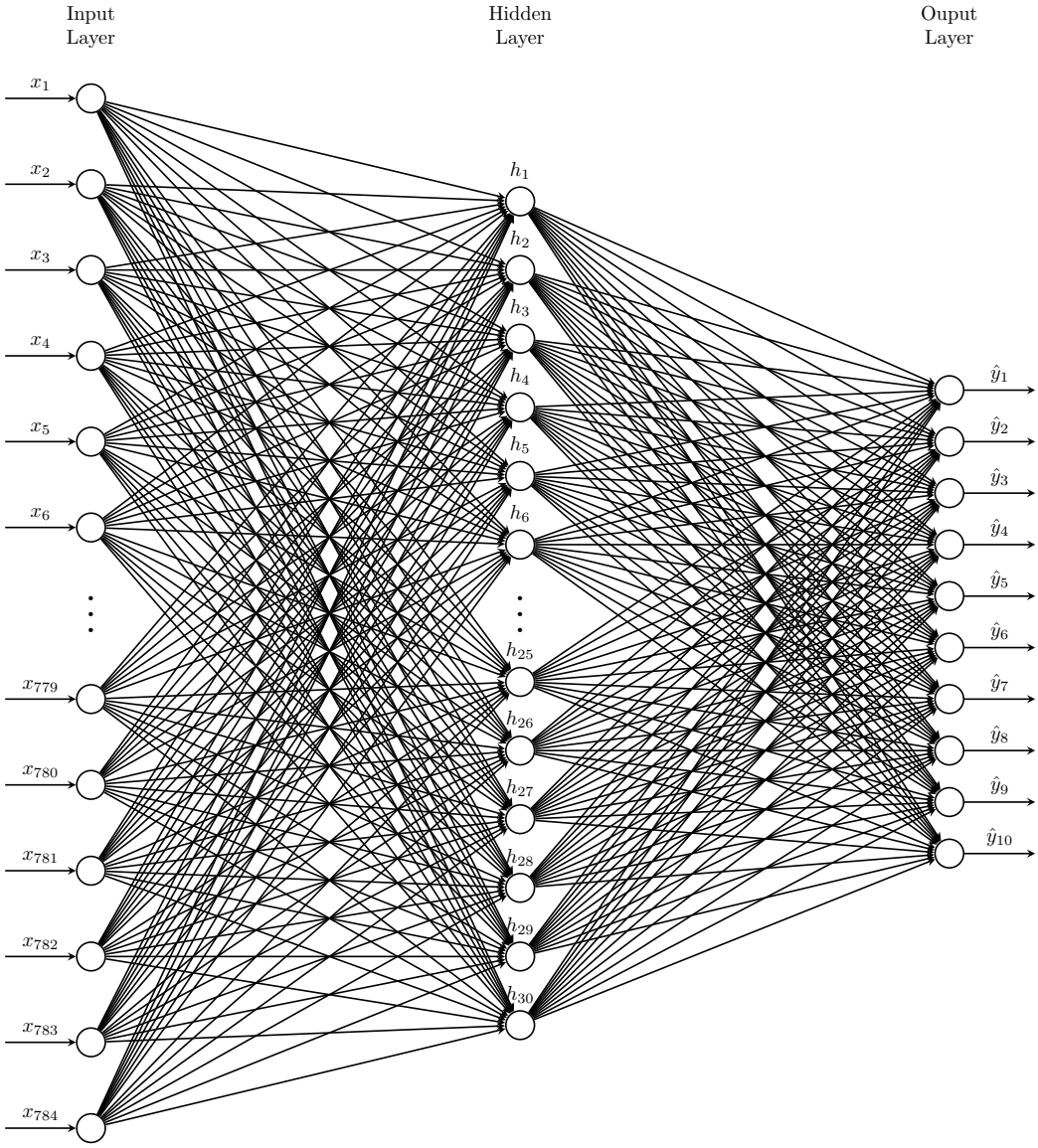

图 4-13: 用于手写体数字识别的前馈神经网络

于调节网络的超参数[12]及过拟合的早期判断。关于这 3 类数据的用途，下文将补充说明。

对于如此“庞大”的神经网络，采用手工的方式来调整网络的权值与偏置，是不可想象的，也是非常困难的。如何通过训练数据自动地调整它们呢？下面将引入 20 世纪 80 年代提出的、具有里程碑意义的误差反传（Back Propagation，BP）算法。

4.2 基于梯度下降的反传算法

4.2.1 梯度下降算法

为应用基于梯度下降的反传算法（Gradient-descent based backpropagation），首先需要定义神经网络的目标函数或代价函数。

对于手写体数字识别问题，可以定义如下的代价函数：

$$C(w,b) = \frac{1}{2n}\sum_{x}\|y_x - a_x\|^2 \tag{5}$$

其中，n 表示训练样例对的个数，这些样例具有如下形式：(x_i, y_{x_i})（y_{x_i} 是输入 x_i 的期望或理想输出）；a_x 是输入 x 的实际输出；$C(w,b)$ 被称为是二次代价函数（Quadratic Cost Function）或均方误差（Mean Squared Error，MSE）函数。

显然，代价函数 $C(w,b)$ 能够反映出网络对于训练样例的整体误差。该误差越小，表明网络的输出越符合输入样例的期望输出。从函数逼近的角度看，此时的网络对训练数据“拟合”得较好。需要注意的是，一定要防止网络训练时存在的“过拟合”现象——一旦发生过拟合，虽然网络在训练数据上的效果很好，但是网络对于那些未知数据的（泛化）逼近效果却很差，这种缺乏泛化能力的网络只不过是“牢牢记住”了训练样例而已，缺乏实用价值。过拟合是神经网络训练过程中需要极力避免和抑制的问题之一。

在定义了神经网络的训练目标或代价函数之后，我们需要应用 BP 算法调整权值 w 与偏置 b，以最小化该代价函数或目标函数。

[12]超参数指的是有关神经网络自身结构的一类参数，例如隐单元的个数、隐藏层的层数等。与权值和偏置参数不同的是，这些参数不能通过神经网络的训练而进行调整。对于输入层与输出层，其参数是问题相关的，即一般可以依据问题的性质加以确定；而对于超参数，则更多地需要依靠网络设计者的经验来设置与调整。

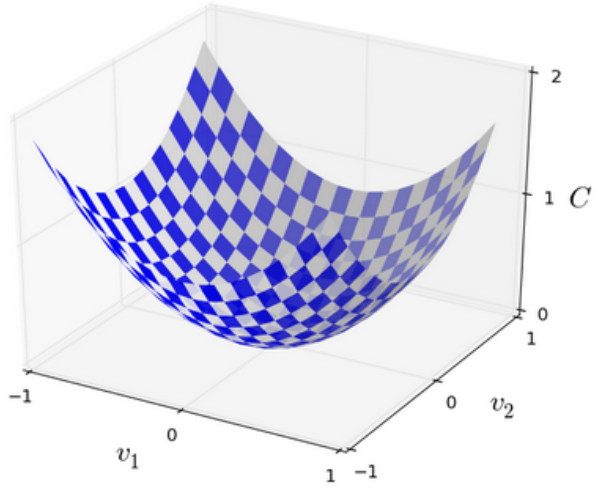

图 4-14: 2 变量的二次代价函数

为便于理解梯度下降原理，下面以 2 输入变量为例进行讲解，即假设 $C(v_1, v_2)$ 是一个只有 2 个变量 v_1 和 v_2 的二次凸代价函数，曲面如图4-14所示。

现在，我们的目标是最小化这个二次代价函数，即要找到代价函数的全局最小值[13]。

对于只有几个变量的代价函数，我们可以简单地使用微积分的方法来计算全局极小值，即通过解析法（令“导数值 =0”）计算出对应的权值与偏置。

但是，对于像神经网络这样的系统，这种解析法并不能奏效，因为神经网络涉及的变量——权值与偏置数量非常巨大。一种行之有效的方法是，采用迭代计算的方式逐步优化网络的权值与偏置。

下面，我们仍然要使用这个简单的、只有 2 个变量的代价函数作为实例来进行分析。

想象一下，假如我们从代价函数的曲面上随机地选取一个点（随机初始化权值与偏置将产生这种效果），然后把一个球放在该点处，那么该球将沿着曲面“径直”地滚落下去，最终将会停留在曲面的最低点处。

显然，这是一种很聪明的解决方案。当然，在最小化代价函数时，我们并不

[13]对于此例，局部极小值也是全局最小值。但是，对于一般的神经网络而言，由于神经元的激活函数具有非线性（例如，Sigmoid、ReLU 等），使得代价函数不再具备凸性质（即函数形状不再呈现出凸的碗形（bowl-shaped））；而是在函数的表面，呈现出多峰多谷、凹凸不平的形状，如图4-15所示，这极大地增加了函数优化的难度。虽然该图展示的是 2 变量代价函数，但是其它具有多变量的代价函数也存在相似的问题。

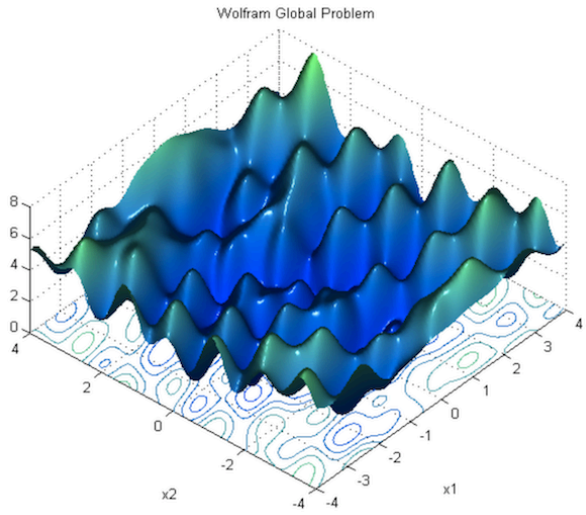

图 4-15: 2 变量的非凸代价函数

需要实际地执行上述物理过程，而是只需要沿着代价函数的梯度不断地调整它的 2 个自变量，最终我们将会停留在代价函数的最低点处。

具体来说，对于代价函数 C，选取较小的 Δv_1 和 Δv_2，下式成立：

$$\Delta C \approx \frac{\partial C}{\partial v_1}\Delta v_1 + \frac{\partial C}{\partial v_2}\Delta v_2 \tag{6}$$

此时，我们可以通过迭代的方式连续地调整 Δv_1 和 Δv_2，并使得 ΔC 总是负的，这样就可以确保代价函数 C 不断地变小了。

下面简化一下上述记法。设：

$$\Delta v = (\Delta v_1, \Delta v_2)^T \tag{7}$$

$$\nabla C = (\frac{\partial C}{\partial v_1}, \frac{\partial C}{\partial v_2})^T \tag{8}$$

这样，公式（6）可以重新写成：

$$\Delta C \approx \nabla C \cdot \Delta v \tag{9}$$

观察一下公式（9），可以发现，如果我们按如下方式设置 Δv（它是一个向量，对于本例而言，分量分别是 Δv_1 和 Δv_2）：

$$\Delta v = -\eta \nabla C \tag{10}$$

其中，η 是一个被称为学习率的小正数，那么再综合公式（9）和公式（10），可得：

$$\Delta C \approx -\eta \nabla C \cdot \nabla C = -\eta \|\nabla C\|^2 \tag{11}$$

显然，上式确保了 $\Delta C \leq 0$。这意味着，如果连续地调整 Δv_1 和 Δv_2，代价函数 C 必将持续减少，直至到达全局最小值（就本例而言）。

实际上，我们得到了一个全新的方法——梯度下降方法：

$$\Delta v = -\eta \nabla C \tag{12}$$

$$v \leftarrow v + \Delta v \tag{13}$$

上述调整过程一直迭代进行下去：计算梯度 $\nabla C \rightarrow$ 计算 $\Delta v \rightarrow$ 计算 $v \rightarrow$ 计算代价 C，直到代价函数满足某个条件为止。上述公式被称为变量更新规则，自然地，它可以被扩展到多变量情形。

其分量形式如下，假设分量为 v_1 和 v_2：

$$v_1 \leftarrow v_1 + \Delta v_1 = v_1 - \eta \frac{\partial C}{\partial v_1} \tag{14}$$

$$v_2 \leftarrow v_2 + \Delta v_2 = v_2 - \eta \frac{\partial C}{\partial v_2} \tag{15}$$

下面将梯度下降方法推广到一般的神经网络中。对于神经网络中的任一权值变量 w_{jk}^l 和偏置变量 b_j^l，它的基本形式是一样的：

$$w_{jk}^l \leftarrow w_{jk}^l + \Delta w_{jk}^l = w_{jk}^l - \eta \frac{\partial C}{\partial w_{jk}^l} \tag{16}$$

$$b_j^l \leftarrow b_j^l + \Delta b_j^l = b_j^l - \eta \frac{\partial C}{\partial b_j^l} \tag{17}$$

其中，w_{jk}^l 表示第 $(l-1)$ 层的第 k 个神经元到第 l 层的第 j 个神经元之间的权值，b_j^l 表示第 l 层的第 j 个神经元的偏置。

接下来，我们需要针对神经网络中每一层的每一个权值变量和每一个偏置变量，推导出它们的更新规则。注意到，公式（5）表示的代价函数有如下形式：

$$C = \frac{1}{n} \sum_x C_x \tag{18}$$

它表示 n 个训练样例的代价函数。其中，每一个训练样例的代价函数是：

$$C_x = \frac{\|y_x - a_x\|^2}{2} \tag{19}$$

因此，对于 n 个训练样例，下式成立：

$$\nabla C = \frac{1}{n} \sum_x \nabla C_x \tag{20}$$

上述公式表明，针对整个训练数据，我们可以计算出每个训练样例的梯度及梯度平均值，并依此调整神经网络中的每个权值与偏置。我们把这种梯度下降方法称为批梯度下降（Batch Gradient Descent，BGD）。这种方法的特点是，每次都需要使用所有的训练样例。然而，当训练样例的个数非常多时，神经网络的权值与偏置的调整过程将会变得非常缓慢。

因此，在实际应用中，可以采取一种称为随机梯度下降（Stochastic Gradient Descent，SGD）的方法——每次只使用一个训练样例或一小部分训练样例（例如 10 个或 100 个）。其代价函数的梯度可以近似为：

$$\nabla C \approx \frac{1}{m}\sum_{o=1}^{m}\nabla C_{x_o} \tag{21}$$

其中，m 是 mini-batch 的样例数目。

这样，使用 mini-batch 时的权值与偏置的更新公式，可以表示如下：

$$\begin{aligned} w_{jk}^{l} &\leftarrow w_{jk}^{l} + \Delta w_{jk}^{l} = w_{jk}^{l} - \frac{\eta}{m}\sum_{o}\frac{\partial C_{x_o}}{\partial w_{jk}^{l}} \\ b_{j}^{l} &\leftarrow b_{j}^{l} + \Delta b_{j}^{l} = b_{j}^{l} - \frac{\eta}{m}\sum_{o}\frac{\partial C_{x_o}}{\partial b_{j}^{l}} \end{aligned} \tag{22}$$

4.2.2 反传算法

到目前为止，我们已经知道了梯度下降算法的整体概况，但是还有一个重要的问题没有解决：如何有效地计算出每个梯度 $\frac{\partial C}{\partial w_{jk}^{l}}$ 和 $\frac{\partial C}{\partial b_{j}^{l}}$？

反传（Backpropagation）算法是一种计算每个权值与偏置梯度的方法。基于反传算法，我们就可以不断地改变权值与偏置，来逐渐地降低整体代价，直到代价满足我们的要求。

为方便起见，我们需要引入一个中间量 δ_j^l，它表示第 l 层的第 j 个神经元的误差，最后，我们将通过中间量 δ_j^l 来导出梯度 $\frac{\partial C}{\partial w_{jk}^{l}}$ 和 $\frac{\partial C}{\partial b_{j}^{l}}$。

注意：以下公式的推导是针对一个样例的情况，多个样例的梯度就是梯度的平均值。

首先，我们定义中间量 δ_j^l：

$$\delta_j^l = \frac{\partial C}{\partial z_j^l} \tag{23}$$

其中，z_j^l 是第 l 层的第 j 个神经元的权值输出：

$$z_j^l = \sum_o w_{jo}^l a_o^{l-1} + b_j^l \tag{24}$$

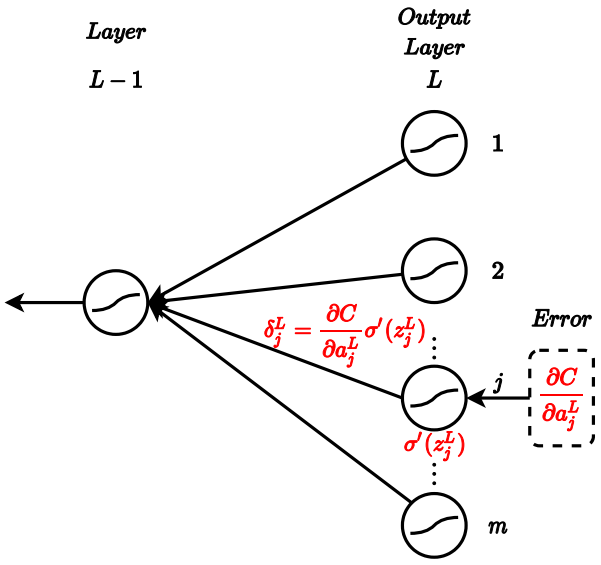

图 4-16: 最后一层第 j 个神经元的反传中间量 δ_j^L

其中 a_o^{l-1} 表示第 $(l-1)$ 层第 o 个神经元的输出 $a_o^{l-1} = \sigma(z_o^{l-1})$。

接着，推导神经网络最后一层 L 的中间量：

$$\delta_j^L = \frac{\partial C}{\partial z_j^L} = \sum_k \frac{\partial C}{\partial a_k^L} \cdot \frac{\partial a_k^L}{\partial z_j^L} = \frac{\partial C}{\partial a_j^L} \cdot \frac{\partial a_j^L}{\partial z_j^L} \tag{25}$$

$$= \frac{\partial C}{\partial a_j^L} \cdot \sigma'(z_j^L) \tag{26}$$

δ_j^L 的示意图如图4-16所示。如果代价函数使用公式 (5)，那么 $\frac{\partial C}{\partial a_j^L} = (a_j^L - y_j)$，其中 y_j 是神经网络输出层的第 j 个神经元的理想输出。如果 $\sigma(z_j^L)$ 函数是 Sigmoid 函数，那么 $\sigma'(z_j^L) = \sigma(z_j^L)(1 - \sigma(z_j^L))$。

最后一层中间量的向量形式为：

$$\boldsymbol{\delta}^L = \nabla_a \boldsymbol{C} \odot \boldsymbol{\sigma}'(\boldsymbol{z}^L) \tag{27}$$

注意，$\odot$ 表示 Hadamard 积；$\nabla_a \boldsymbol{C}$ 是一个向量，它的分量是 $\frac{\partial C}{\partial a_j^L}$；$\boldsymbol{\sigma}'(\boldsymbol{z}^L)$ 也是一个向量，其分量为 $\sigma'(z_j^L)$。如果代价函数使用公式 (5)，那么 $\nabla_a \boldsymbol{C} = (\boldsymbol{a}^L - \boldsymbol{y})$。其中，$\boldsymbol{a}^L$ 是输出层的实际输出向量，$\boldsymbol{y}$ 是输出层的理想输出向量。$\boldsymbol{\sigma}'(\cdot)$ 为逐元素运算函数。

最后，可以导出层 l 与层 $(l+1)$ 的 δ 之间的关系：

$$\delta_j^l = \frac{\partial C}{\partial z_j^l} = \sum_k \frac{\partial C}{\partial z_k^{l+1}} \frac{\partial z_k^{l+1}}{\partial z_j^l} = \sum_k \frac{\partial z_k^{l+1}}{\partial z_j^l} \delta_k^{l+1} \tag{28}$$

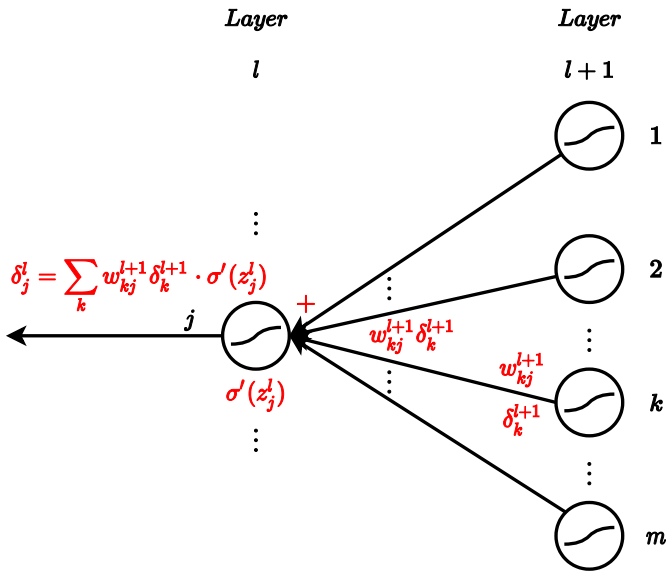

图 4-17: 相邻层的反传中间量 δ_j^l 与 δ_k^{l+1} 之间的关系

$$z_k^{l+1} = \sum_j w_{kj}^{l+1} a_j^l + b_k^{l+1} = \sum_j w_{kj}^{l+1} \sigma(z_j^l) + b_k^{l+1} \tag{29}$$

$$\frac{\partial z_k^{l+1}}{\partial z_j^l} = w_{kj}^{l+1} \sigma'(z_j^l) \tag{30}$$

$$\delta_j^l = \sum_k w_{kj}^{l+1} \delta_k^{l+1} \cdot \sigma'(z_j^l) \tag{31}$$

相邻层的反传中间量 δ_j^l 与 δ_k^{l+1} 之间的关系示意图如图4-17所示。它的向量形式如下[14]：

$$\boldsymbol{\delta}^l = ((\boldsymbol{W}^{l+1})^T \boldsymbol{\delta}^{l+1}) \odot \boldsymbol{\sigma}'(\boldsymbol{z}^l) \tag{32}$$

有了中间量 δ_j^l，就可以更方便地计算偏置的梯度：

$$\frac{\partial C}{\partial b_j^l} = \sum_k \frac{\partial C}{\partial z_k^l} \frac{\partial z_k^l}{\partial b_j^l} = \frac{\partial C}{\partial z_j^l} \frac{\partial z_j^l}{\partial b_j^l} = \delta_j^l \frac{\partial z_j^l}{\partial b_j^l} \tag{33}$$

[14]为区别，我们使用大写粗体字 $\boldsymbol{W}$ 表示权值矩阵，其分量依然使用 w_{kj} 这样的小写标量形式来表示。

$$z_j^l = \sum_k w_{jk}^l a_k^{l-1} + b_j^l \tag{34}$$

$$\frac{\partial z_j^l}{\partial b_j^l} = 1 \tag{35}$$

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l \tag{36}$$

类似地，可以计算出权值的梯度：

$$\frac{\partial C}{\partial w_{jk}^l} = \sum_i \frac{\partial C}{\partial z_i^l}\frac{\partial z_i^l}{\partial w_{jk}^l} = \frac{\partial C}{\partial z_j^l}\frac{\partial z_j^l}{\partial w_{jk}^l} = \delta_j^l \frac{\partial z_j^l}{\partial w_{jk}^l} \tag{37}$$

$$z_j^l = \sum_o w_{jo}^l a_o^{l-1} + b_j^l \tag{38}$$

$$\frac{\partial z_j^l}{\partial w_{jk}^l} = a_k^{l-1} \tag{39}$$

$$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1}\delta_j^l \tag{40}$$

上述梯度的向量与矩阵形式分别为：

$$\begin{aligned} \frac{\partial C}{\partial \boldsymbol{b}^l} &= \boldsymbol{\delta^l} \\ \frac{\partial C}{\partial \boldsymbol{W}^l} &= \boldsymbol{\delta}^l \left(\boldsymbol{a}^{l-1}\right)^T \end{aligned} \tag{41}$$

下面，给出 Backpropagation 算法。

算法 **4.1**（*Backpropagation* 算法）

def Backpropagation():
1. Input a set of training examples
2. for epoch in range(epochs):
3. Shuffling training examples randomly
4. Spliting training examples into mini-batches
5. for mini-batch in mini-batches:
6. UpdateMiniBatch(mini-batch,η)

def UpdateMiniBatch(mini-batch,η):
1. for $\boldsymbol{x}$,$\boldsymbol{y}$ in mini-batch(size=m):
2. Set the corresponding input activation $\boldsymbol{a}^{\boldsymbol{x},1}$, perform the following steps:
3. Feedforward: for each $l = 2, 3, \cdots, L$ compute $\boldsymbol{z}^{\boldsymbol{x},l} = \boldsymbol{W}^l \boldsymbol{a}^{\boldsymbol{x},l-1} + \boldsymbol{b}^l$ and
4. $\boldsymbol{a}^{\boldsymbol{x},l} = \boldsymbol{\sigma}\left(\boldsymbol{z}^{\boldsymbol{x},l}\right)$.
5. Output error:$\boldsymbol{\delta}^{\boldsymbol{x},\boldsymbol{L}} = \nabla_a \boldsymbol{C}_{\boldsymbol{x}} \odot \boldsymbol{\sigma}'(\boldsymbol{z}^{\boldsymbol{x},L})$.
6. Backpropagate the error: for each $l = L-1, L-2, \cdots, 2$ compute
7. $\boldsymbol{\delta}^{\boldsymbol{x},l} = ((\boldsymbol{W}^{l+1})^T \boldsymbol{\delta}^{\boldsymbol{x},l+1}) \odot \boldsymbol{\sigma}'(\boldsymbol{z}^{\boldsymbol{x},l})$.
8. Gradient descent: for each $l = L, L-1, \cdots, 2$ update the weights and biases:
9. $\boldsymbol{W}^l \leftarrow \boldsymbol{W}^l - \frac{\eta}{m} \sum_{\boldsymbol{x}} \boldsymbol{\delta}^{\boldsymbol{x},l} \left(\boldsymbol{a}^{\boldsymbol{x},l-1}\right)^T$
10. $\boldsymbol{b}^l \leftarrow \boldsymbol{b}^l - \frac{\eta}{m} \sum_{\boldsymbol{x}} \boldsymbol{\delta}^{\boldsymbol{x},l}$

注意，在上述反传算法的推导过程中，并不需要我们给出代价函数 C 以及各层神经元的激活函数 $\sigma(\cdot)$ 的具体形式。因此，这意味着，上面列出的 Backpropagation 算法适用于任何合适的代价函数以及激活函数，甚至允许各层神经元的激活函数具有不同的形式——在实际实现算法时，可依据 C 和 $\sigma(\cdot)$ 的具体函数形式来确定 $\nabla_a \boldsymbol{C}$ 以及相应的 $\boldsymbol{\sigma}'(\cdot)$。

4.3 神经元饱和问题

如果使用公式（5）作为代价函数，并且使用 Sigmoid 作为神经元的激活函数，那么在输出层就存在学习缓慢的问题[15]。

从 Sigmoid 的曲线可以看出，主要原因在于，Sigmoid 在两端的梯度很小，这会导致权值与偏置的调整量也很小。例如，观察输出层的误差公式（25）：$\delta_j^L = \frac{\partial C}{\partial a_j^L} \cdot \sigma'(z_j^L)$，右端表达式含有梯度项 $\sigma'(z_j^L)$。这个问题被称为神经元的饱和问题，它是制约反传算法效率与有效性的一大障碍。

一个解决办法是，定义新的代价函数 Cross-entropy（交叉熵），但是仍然维持

[15]实际上，由于 Sigmoid 神经元的使用，其它层也存在这个问题。

神经元的激活函数为 Sigmoid。Cross-entropy 代价函数定义如下：

$$C(w,b) = -\frac{1}{n}\sum_x\sum_j [y_j \ln a_j^L + (1-y_j)\ln(1-a_j^L)] \tag{42}$$

其中，n 是训练数据的个数，y_j 是输出层第 j 个神经元的理想输出，a_j^L 是输出层第 j 个神经元的实际输出。

Cross-entropy 代价函数满足如下两个性质：

- $C(w,b)$ 是非负的：$y_j \in [0,1]$，$a_j^L \in (0,1) \Rightarrow \ln a_j^L < 0$；
- 如果 y_j 和 a_j^L 很接近，那么 $C(w,b)$ 就接近于 0；

正是由于上述 2 个性质，使得 Cross-entropy 非常适合作为代价函数。更为重要的是，该代价函数的引入，可以消除网络输出层中存在的神经元饱和问题，分析如下。

还是以单个样例为例。针对新的 Cross-entropy 代价函数，可以按如下方式计算中间量 δ_j^L 中的因子 $\frac{\partial C}{\partial a_j^L}$：

$$\frac{\partial C}{\partial a_j^L} = -(\frac{y_j}{a_j^L} - \frac{1-y_j}{1-a_j^L}) = \frac{1-y_j}{1-a_j^L} - \frac{y_j}{a_j^L} \tag{43}$$

$$= \frac{a_j^L - y_j}{a_j^L(1-a_j^L)} = \frac{a_j^L - y_j}{\sigma(z_j^L)(1-\sigma(z_j^L))} = \frac{a_j^L - y_j}{\sigma'(z_j^L)} \tag{44}$$

然后，将 $\frac{\partial C}{\partial a_j^L} = \frac{a_j^L - y_j}{\sigma'(z_j^L)}$ 代入公式（25），得到：

$$\delta_j^L = \frac{\partial C}{\partial a_j^L} \cdot \sigma'(z_j^L) = a_j^L - y_j \tag{45}$$

上式表明，Cross-entropy 代价函数的引入，消去了输出层 δ_j^L 中 Sigmoid 的梯度项 $\sigma'(z_j^L)$。

一般而言，如果输出层的神经元使用 Sigmoid 激活函数（特别是分类（Classification）问题），那么使用 Cross-entropy 代价函数总是合适的。但是，如果输出层的神经元使用线性激活函数（对于回归（Regression）问题），即 $a_j^L = z_j^L$，那么使用二次代价函数就是合适的，因为 $\delta_j^L = (a_j^L - y_j)$ 本身就成立。

对于二分类情形，输出层可以使用 Sigmoid 激活函数。如果是多分类问题，那么输出层就可以使用 Softmax 激活函数。实际上，Softmax 函数是 Sigmoid 函数在多分类问题上的推广。

Softmax 函数定义如下：

$$\sigma(z_j) = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \ for\ j = 1, ..., K. \tag{46}$$

如果 $K = 2$，则有：

$$\begin{aligned} \sigma(z_0) &= \frac{e^{z_0}}{e^{z_0} + e^{z_1}} \\ &= \frac{1}{1 + e^{(z_1 - z_0)}} \end{aligned} \tag{47}$$

公式 (47) 表明，Sigmoid 函数是 Softmax 函数的特殊情形。

如果神经网络的输出层使用 Softmax 激活函数，则输出层就形成了所谓的 Softmax 层：

$$z_j^L = \sum_k w_{jk}^L a_k^{L-1} + b_j^L \tag{48}$$

$$a_j^L = \frac{e^{z_j^L}}{\sum_k e^{z_k^L}} \tag{49}$$

Softmax 层的输出可以被看作是一个概率分布。许多分类问题可以被解释成概率分布，例如，MNIST 的分类问题，它的 a_j^L 就可以被看作是正确数字分类 j 的一个概率估算。

在输出层使用 Softmax 函数的情况下，可以使用 Negative Log-likelihood 函数[16]作为代价函数，以抑制神经元的饱和问题：

$$C(w, b) = -\frac{1}{n} \sum_x \sum_k t_k^x \ln a_k^L \tag{50}$$

其中，n 是训练数据的个数；x 表示输入的训练样例；k 是 Softmax 输出层的输出神经元的编号，$k \in [0, K-1]$，K 表示分类个数；a_k^L 是第 k 个输出神经元的实际输出，一般表示样例 x 属于第 k 个分类的预测概率；t^x 表示输出神经元的理想输出向量，一般采用 One-Hot 表示法：t_k^x 是与样例 x 对应的第 k 个输出神经元的理想输出值——如果 x 属于分类 $j \in [0, K-1]$，则第 j 个输出神经元的理想输出是 1，其余神经元的理想输出是 0，也就是 $t_j = 1$，而 $t_i = 0$，对于 $i \neq j$。例如，在 MNIST 分类问题中，如果 $t_7^x = 1$，表明样例 x 的正确类别是数字 7，那么输出层的第 7 个神经元的理想输出是 1，其余全部是 0。

[16]实际上，该函数是多分类情形下的交叉熵代价函数。

下面将推导这种神经网络的权值与偏置梯度。还是以单个样例为例：

$$C_x(w,b) = -\sum_k t_k \ln a_k^L = -\sum_k t_k \ln \frac{e^{z_k^L}}{\sum_o e^{z_o^L}} \tag{51}$$

$$= \sum_k t_k (\ln \sum_o e^{z_o^L} - z_k^L) \tag{52}$$

如果 j 是当前的预期分类类别（$t_j = 1$，$t_i = 0$，$i \neq j$）：

$$\frac{\partial C_x}{\partial z_j^L} = \frac{\partial}{\partial z_j^L}(\ln \sum_o e^{z_o^L} - z_j^L) = \frac{e^{z_j^L}}{\sum_o e^{z_o^L}} - 1 \tag{53}$$

$$= a_j^L - 1 \tag{54}$$

如果 j 不是当前的预期分类类别，而 d 是当前的预期分类类别（$t_d = 1$，$t_i = 0$，$i \neq d$）：

$$\frac{\partial C_x}{\partial z_j^L} = \frac{\partial}{\partial z_j^L}(\ln \sum_o e^{z_o^L} - z_d^L) = \frac{e^{z_j^L}}{\sum_o e^{z_o^L}} - 0 \tag{55}$$

$$= a_j^L - 0 \tag{56}$$

综合上述 2 种情况，得到（假设 d 是当前的预期分类类别）：

$$\delta_j^L = \frac{\partial C_x}{\partial z_j^L} = a_j^L - \delta_{dj} \tag{57}$$

$$\delta_{dj} = \begin{cases} 1 & j = d \\ 0 & j \neq d \end{cases} \tag{58}$$

利用该中间量，可以计算出偏置的梯度：

$$\frac{\partial C}{\partial b_j^L} = \sum_k \frac{\partial C}{\partial z_k^L}\frac{\partial z_k^L}{\partial b_j^L} = \frac{\partial C}{\partial z_j^L}\frac{\partial z_j^L}{\partial b_j^L} = \delta_j^L \frac{\partial z_j^L}{\partial b_j^L} \tag{59}$$

$$z_j^L = \sum_k w_{jk}^L a_k^{L-1} + b_j^L \tag{60}$$

$$\frac{\partial z_j^L}{\partial b_j^L} = 1 \tag{61}$$

$$\frac{\partial C}{\partial b_j^L} = \delta_j^L = a_j^L - \delta_{dj} \tag{62}$$

相应地，权值的梯度为：

$$\frac{\partial C}{\partial w_{jk}^L} = \sum_i \frac{\partial C}{\partial z_i^L}\frac{\partial z_i^L}{\partial w_{jk}^L} = \frac{\partial C}{\partial z_j^L}\frac{\partial z_j^L}{\partial w_{jk}^L} = \delta_j^L \frac{\partial z_j^L}{\partial w_{jk}^L} \tag{63}$$

$$z_j^L = \sum_o w_{jo}^L a_o^{L-1} + b_j^L \tag{64}$$

$$\frac{\partial z_j^L}{\partial w_{jk}^L} = a_k^{L-1} \tag{65}$$

$$\frac{\partial C}{\partial w_{jk}^L} = a_k^{L-1}\delta_j^L = a_k^{L-1}(a_j^L - \delta_{dj}) \tag{66}$$

比较公式（57）与公式（45），可以发现，它们实际上是一致的，都可以用来解决神经元饱和问题。

Log-likelihood/Softmax 的组合与 Cross-entropy/Sigmoid 的组合，它们之间形成了一种扩展的关系——前者用于多分类神经网络，后者用于 2 分类神经网络，是前者的特殊情形。

下面将使用另一种方法来推导上述公式。由于推导过程中要用到 Softmax 激活函数的偏导数，所以先给出它的偏导数推导过程。

仍然以 a_i^L 来表示 Softmax 输出层 L 的第 i 个神经元，它的输出值为：

$$a_i^L = \frac{e^{z_i^L}}{\sum_k e^{z_k^L}} \tag{67}$$

如果 $j = i$：

$$\frac{\partial a_i^L}{\partial z_j^L} = \frac{\partial}{\partial z_j^L}\frac{e^{z_i^L}}{\sum_k e^{z_k^L}} = \frac{e^{z_i^L}}{\sum_k e^{z_k^L}} - e^{z_i^L}\cdot\frac{e^{z_i^L}}{(\sum_k e^{z_k^L})^2} \tag{68}$$

$$= a_i^L - (a_i^L)^2 = a_i^L(1-a_i^L) = a_j^L(1-a_j^L) \tag{69}$$

如果 $j \neq i$：

$$\frac{\partial a_i^L}{\partial z_j^L} = \frac{\partial}{\partial z_j^L}\frac{e^{z_i^L}}{\sum_k e^{z_k^L}} = -e^{z_i^L}\cdot\frac{e^{z_j^L}}{(\sum_k e^{z_k^L})^2} \tag{70}$$

$$= -a_i^L a_j^L \tag{71}$$

Log-likelihood 的代价函数参见公式50。还是以单个样例为例：

$$C_x(w,b) = -\sum_k t_k \ln a_k^L \tag{72}$$

可得：

$$\delta_j^L = \frac{\partial C_x}{\partial z_j^L} = \frac{\partial}{\partial z_j^L}(-\sum_k t_k \ln a_k^L) = -\sum_k t_k \frac{1}{a_k^L}\frac{\partial a_k^L}{\partial z_j^L} \tag{73}$$

$$= -t_j \frac{1}{a_j^L}\frac{\partial a_j^L}{\partial z_j^L} - \sum_{k\neq j} t_k \frac{1}{a_k^L}\frac{\partial a_k^L}{\partial z_j^L} \tag{74}$$

$$= -t_j \frac{1}{a_j^L} a_j^L(1-a_j^L) - \sum_{k\neq j} t_k \frac{1}{a_k^L}(-a_j^L a_k^L) \tag{75}$$

$$= -t_j + t_j a_j^L + \sum_{k\neq j} t_k a_j^L = -t_j + a_j^L \sum_k t_k \tag{76}$$

$$= a_j^L - t_j \tag{77}$$

可以看出，它与公式（57）的意义是一样的。权值与偏置的偏导推导过程与前面一样，此处不再赘述。

4.4 神经网络的多层函数嵌套视角

神经网络对输入样本 $\boldsymbol{x}$ 连续地应用非线性函数——第 l 层的输出是第 $l+1$ 层的输入，最后在输出层获得相应的输出，这是一种典型的数据流水线。从函数的角度看，神经网络相当于多层函数的嵌套。

下面，从多层函数嵌套的角度，应用标量对矩阵的求导法则[17]，直接求解中间量 (向量形式)、偏置梯度 (向量形式) 与权值梯度 (矩阵形式)。

假设神经网络的总层数为 L(其中第 L 层为输出层)，使用 mini-batch(训练样本点的个数为 m) 训练方式，代价函数为 C：

$$C = \frac{1}{m}\sum_{\boldsymbol{x}} C_{\boldsymbol{x}} \tag{78}$$

为方便，仍然以单样本为例推导出反传算法所需的梯度。所得结果可以直接使用求和平均的方式应用于 m 个样本情形。

首先，依据标量对矩阵的求导法则，将 $C_{\boldsymbol{x}}$ 的微分表达为对最后一层 $\boldsymbol{z}^L$ 的微分形式：

$$dC_{\boldsymbol{x}} = \mathrm{tr}\left(dC_{\boldsymbol{x}}\right) = \mathrm{tr}\left(\left(\frac{\partial C_{\boldsymbol{x}}}{\partial \boldsymbol{z}^L}\right)^T d\boldsymbol{z}^L\right) \tag{79}$$

其中，$\frac{\partial C_{\boldsymbol{x}}}{\partial \boldsymbol{z}^L}$ 就是第 L 层 (输出层) 的中间量 $\boldsymbol{\delta}^L$。其具体形式可表示为公式 (27)，即：

$$\boldsymbol{\delta}^L = \nabla_a \boldsymbol{C}_{\boldsymbol{x}} \odot \boldsymbol{\sigma}'(\boldsymbol{z}^L) \tag{80}$$

[17]具体方法，请阅读《机器学习》课程系列之“判别函数的线性分类基础”一章。

当然，也可以直接依据代价函数 $C_{\boldsymbol{x}}$ 与输出层激活函数的具体形式，应用标量对矩阵的求导方法直接进行求解。下文，将展示当输出层激活函数为 Softmax 以及代价函数为多分类交叉熵损失函数时，直接求取中间量 $\boldsymbol{\delta}^L$ 的方法。

然后，求解第 l 层与第 $l+1$ 层中间量 (即 $\boldsymbol{\delta}^l$ 与 $\boldsymbol{\delta}^{l+1}$) 之间的关系以及权值与偏置梯度公式。对于第 $l+1$ 层，其微分形式为：

$$\begin{aligned} dC_{\boldsymbol{x}} &= \operatorname{tr}\left(\left(\frac{\partial C_{\boldsymbol{x}}}{\partial \boldsymbol{z}^{l+1}}\right)^T d\boldsymbol{z}^{l+1}\right) \\ &= \operatorname{tr}\left(\left(\boldsymbol{\delta}^{l+1}\right)^T d\boldsymbol{z}^{l+1}\right) \end{aligned} \tag{81}$$

将 $\boldsymbol{z}^{l+1} = \boldsymbol{W}_{l+1}\boldsymbol{\sigma}_l\left(\boldsymbol{z}^l\right) + \boldsymbol{b}_{l+1}$ 代入上式，可得：

$$\begin{aligned} dC_{\boldsymbol{x}} &= \operatorname{tr}\left(\left(\boldsymbol{\delta}^{l+1}\right)^T d\left(\boldsymbol{W}_{l+1}\boldsymbol{\sigma}_l\left(\boldsymbol{z}^l\right) + \boldsymbol{b}_{l+1}\right)\right) \quad \Rightarrow \\ dC_{\boldsymbol{x}} &= \operatorname{tr}\left(\left(\boldsymbol{\delta}^{l+1}\right)^T d\boldsymbol{W}_{l+1}\boldsymbol{\sigma}_l\left(\boldsymbol{z}^l\right)\right) + \\ &\quad \operatorname{tr}\left(\left(\boldsymbol{\delta}^{l+1}\right)^T \boldsymbol{W}_{l+1} d\boldsymbol{\sigma}_l\left(\boldsymbol{z}^l\right)\right) + \\ &\quad \operatorname{tr}\left(\left(\boldsymbol{\delta}^{l+1}\right)^T d\boldsymbol{b}_{l+1}\right) \end{aligned} \tag{82}$$

从上式的第 1 项可以得到权值 $\boldsymbol{W}_{l+1}$ 的梯度为：

$$\frac{\partial C_{\boldsymbol{x}}}{\partial \boldsymbol{W}_{l+1}} = \boldsymbol{\delta}^{l+1}\boldsymbol{\sigma}_l^T\left(\boldsymbol{z}^l\right) \tag{83}$$

从第 3 项可以得到偏置 $\boldsymbol{b}_{l+1}$ 的梯度为：

$$\frac{\partial C_{\boldsymbol{x}}}{\partial \boldsymbol{b}_{l+1}} = \boldsymbol{\delta}^{l+1} \tag{84}$$

对第 2 项继续变换，得到：

$$\begin{aligned} &\operatorname{tr}\left(\left(\boldsymbol{\delta}^{l+1}\right)^T \boldsymbol{W}_{l+1} d\boldsymbol{\sigma}_l\left(\boldsymbol{z}^l\right)\right) \quad \Rightarrow \\ &\operatorname{tr}\left(\left(\boldsymbol{\delta}^{l+1}\right)^T \boldsymbol{W}_{l+1}\left(\boldsymbol{\sigma}_l'\left(\boldsymbol{z}^l\right) \odot d\boldsymbol{z}^l\right)\right) \quad \Rightarrow \\ &\operatorname{tr}\left(\left(\boldsymbol{W}_{l+1}^T\boldsymbol{\delta}^{l+1} \odot \boldsymbol{\sigma}_l'\left(\boldsymbol{z}^l\right)\right)^T d\boldsymbol{z}^l\right) \quad \Rightarrow \end{aligned} \tag{85}$$

进而得到中间量 $\boldsymbol{\delta}^l$：

$$\boldsymbol{\delta}^l = \frac{\partial C_{\boldsymbol{x}}}{\partial \boldsymbol{z}^l} = \boldsymbol{W}_{l+1}^T\boldsymbol{\delta}^{l+1} \odot \boldsymbol{\sigma}_l'\left(\boldsymbol{z}^l\right) \tag{86}$$

于是，在进行反传训练时，从第 L 层 (输出层) 出发，计算 $\boldsymbol{\delta}^L$ 并逐层往回计算前一层的中间量及相应的权值梯度与偏置梯度，最后到达第 2 层——在此层，简单

地令 $\boldsymbol{\sigma}_1(\boldsymbol{z}^1)=\boldsymbol{x}$ 即可——因为输入层并未使用激活函数，而是直接将样本数据点输入给第 2 层。

下面，将展示输出层 L 使用 Softmax 激活函数以及代价函数为多分类交叉熵损失函数时中间量 $\boldsymbol{\delta}^L$ 的计算方法。

设 $\boldsymbol{\sigma}_L(\boldsymbol{z}^L)=\frac{\mathbf{exp}(\boldsymbol{z}^L)}{\mathbf{1}^T\mathbf{exp}(\boldsymbol{z}^L)}$ 表示 Softmax 函数逐元素运算后得到的列向量，其中 $\mathbf{1}$ 表示全 1 向量，$\mathbf{exp}(\cdot)$ 为逐元素运算指数函数；设 $\boldsymbol{y}$ 表示对应的 K 维 One-Hot 列向量，为输入 $\boldsymbol{x}$ 的理想输出。于是，损失函数 $C_{\boldsymbol{x}}$ 可表示为：

$$\begin{aligned} C_{\boldsymbol{x}} &= -\boldsymbol{y}^T\mathbf{log}\boldsymbol{\sigma}_L\left(\boldsymbol{z}^L\right) \\ &= -\boldsymbol{y}^T\left(\boldsymbol{z}^L-\mathbf{1}\cdot\log\left(\mathbf{1}^T\mathbf{exp}(\boldsymbol{z}^L)\right)\right) \\ &= \log\left(\mathbf{1}^T\mathbf{exp}(\boldsymbol{z}^L)\right)-\boldsymbol{y}^T\boldsymbol{z}^L \end{aligned} \tag{87}$$

其中 $\mathbf{log}$ 为逐元素运算对数函数，log 为相应的 (标量运算) 对数函数。

于是，为计算中间量 $\boldsymbol{\delta}^L=\frac{\partial C_{\boldsymbol{x}}}{\partial \boldsymbol{z}^L}$，对公式 (87) 两端关于 $\boldsymbol{z}^L$ 应用全微分算子，得到：

$$\begin{aligned} dC_{\boldsymbol{x}} &= d\log\left(\mathbf{1}^T\mathbf{exp}(\boldsymbol{z}^L)\right)-\boldsymbol{y}^Td\boldsymbol{z}^L \\ &= \frac{1}{\mathbf{1}^T\mathbf{exp}(\boldsymbol{z}^L)}\mathbf{1}^Td\mathbf{exp}(\boldsymbol{z}^L)-\boldsymbol{y}^Td\boldsymbol{z}^L \\ &= \frac{1}{\mathbf{1}^T\mathbf{exp}(\boldsymbol{z}^L)}\mathbf{1}^T\left(\mathbf{exp}(\boldsymbol{z}^L)\odot d\boldsymbol{z}^L\right)-\boldsymbol{y}^Td\boldsymbol{z}^L \\ &= \frac{1}{\mathbf{1}^T\mathbf{exp}(\boldsymbol{z}^L)}\mathbf{exp}^T(\boldsymbol{z}^L)d\boldsymbol{z}^L-\boldsymbol{y}^Td\boldsymbol{z}^L \end{aligned} \tag{88}$$

最后，可以得到中间量 $\boldsymbol{\delta}^L$：

$$\boldsymbol{\delta}^L=\frac{\partial C_{\boldsymbol{x}}}{\partial \boldsymbol{z}^L}=\frac{\mathbf{exp}(\boldsymbol{z}^L)}{\mathbf{1}^T\mathbf{exp}(\boldsymbol{z}^L)}-\boldsymbol{y}=\boldsymbol{\sigma}_L(\boldsymbol{z}^L)-\boldsymbol{y} \tag{89}$$

4.5 过拟合问题

在训练和设计神经网络的过程中，需要使用三类数据：Training Data（训练数据）、Validation Data（验证数据）、Testing Data（测试数据）。

从统计学的角度来看，如果模型反映的是数据中的随机噪声或误差，而不是以模型误差的形式反映数据中的内在关系，我们称模型的训练出现了过拟合（Overfitting）现象。

对于神经网络而言，过拟合发生时，网络的泛化能力就降低了——此时，神经网络“精确”地拟合了训练数据，但是不能很好地泛化到其它未“看见”过的数据上。

上述三类数据的作用各不相同：

- Training Data：用于训练神经网络，调整权值与偏置；
- Validation Data：用于过拟合的判断及早期停止（Early stop），如果神经网络在 Training data 上的精度在提高，但是在 Validation Data 上的精度维持不变或降低，表明过拟合发生了，应该停止训练。本数据的另一个用途是，用于调整网络的超参数，例如学习率、网络结构及隐单元的个数等；
- Testing Data：用于最终测试神经网络的泛化能力；

为了减轻或解决过拟合问题，可以采取如下几种方法。第 1 种方法是，增加 Training data 的规模。例如，采集更多的 Training data，或者通过平移、旋转、镜像等方式从原始的 Training data 中生成更多新的 Training data。

第 2 种方法是，使用正则项或调节项（Regularization）。例如：

$$C = C_0 + \frac{\lambda}{2n}\sum_w w^2 \tag{90}$$

其中，C_0 是原始代价函数，例如二次代价函数、Cross-entropy 代价函数、Log-likelihood 代价函数等。$\lambda > 0$ 是调节参数，n 是训练数据的个数，w 是神经网络的权值（项）。该调节技术被称为是 L2 Regularization。

总体上来说，通过在代价函数中添加调节项，可以使神经网络的权值偏小，有利于抑制过拟合。公式（90），在形式上非常类似于 UCT（Monte Carlo Tree Search+UCB）中的 UCB 公式，该公式反映了 Exploration-Exploitation 之间的某种平衡，公式的第 1 项反映了 Exploitation，第 2 项反映了 Exploration[18]。公式（90），反映了代价函数最小化与权值最小化之间的某种平衡。λ 决定了它们之间的相对重要性。

在添加了正则项之后，权值偏导公式为 (一个样本数据点)：

$$\frac{\partial C}{\partial w_{ij}^l} = \frac{\partial C_0}{\partial w_{ij}^l} + \frac{\lambda}{n} w_{ij}^l \tag{91}$$

而偏置偏导公式保持不变，还是：

$$\frac{\partial C}{\partial b_j^l} = \frac{\partial C_0}{\partial b_j^l} \tag{92}$$

[18]请注意，这种 2 分量公式普遍存在于各种形式的学习或训练算法中。例如，一种深度强化学习算法 Soft Actor-Critic(SAC)，在标准的目标函数（最大化期望回报）基础上，添加了一个公式分量——最大化熵。理论与实验表明，这种目标函数能够增强学习算法的稳定性，请阅读相关的论文。

其中，$\frac{\partial C_0}{\partial w_{ij}^l}$ 与 $\frac{\partial C_0}{\partial b_j^l}$ 的计算公式在前面已经推导出来了。

下面给出相应的带调节项的权值更新公式 (一个样本数据点)：

$$w_{ij}^l = w_{ij}^l - \eta\frac{\partial C_0}{\partial w_{ij}^l} - \frac{\eta\lambda}{n}w_{ij}^l = (1 - \frac{\eta\lambda}{n})w_{ij}^l - \eta\frac{\partial C_0}{\partial w_{ij}^l} \tag{93}$$

偏置更新公式不变，还是：

$$b_j^l = b_j^l - \eta\frac{\partial C_0}{\partial b_j^l} \tag{94}$$

其中，$1 - \frac{\eta\lambda}{n}$ 是权值衰减（weight decay）因子。可以看出，在新的权值更新公式中，权值衰减因子取代了常数 1。

下面给出 mini-batch 下带调节项的一般性权值更新公式：

$$w_{ij}^l = w_{ij}^l - \frac{\eta}{m}\sum_x \frac{\partial C_x}{\partial w_{ij}^l} - \frac{\eta\lambda}{n}w_{ij}^l \tag{95}$$

$$= (1 - \frac{\eta\lambda}{n})w_{ij}^l - \frac{\eta}{m}\sum_x \frac{\partial C_x}{\partial w_{ij}^l} \tag{96}$$

偏置更新公式不变，还是：

$$b_j^l = b_j^l - \frac{\eta}{m}\sum_x \frac{\partial C_x}{\partial b_j^l} \tag{97}$$

其中，m 表示 mini-batch 中训练样例的个数。

在 MNIST 上的实验结果表明，给代价函数增加调节项具有如下优点：

- 减少过拟合现象；
- 增加分类精度；
- 可以提高神经网络的稳定性。如果不使用调节项，神经网络有时候会陷入局部极小，且不稳定；

5 练习

1. 使用反传算法训练前馈神经网络，实现手写体数字的识别；

6 参考文献

1. Martin T. Hagan, Howard B. Demuth, Mark H. Beale. 戴葵等译。神经网络设计，机械工业出版社，2002 年 9 月第 1 版。
2. Michael A. Nielsen. Neural Networks and Deep Learning. Determination Press, 2015.
3. 深度学习的发展历史.
4. 支持向量机.
5. Wikipedia: Convolutional neural network.
6. MNIST data set.
7. Softmax Layer and Log-likelihood loss function.
8. Softmax vs. Softmax-Loss: Numerical Stability.
9. What's the difference between 3 data sets.
10. Wikipedia: Overfitting.
11. 百度百科：前馈神经网络。
12. How neural networks are trained.