第 1 章

自然语言处理基础

自然语言处理（Natural Language Processing，NLP）是一个跨学科领域，它结合了计算科学、语言学、认知科学和人工智能，主要研究能够让计算机实现与人类语言有关的各类任务的各种理论和方法，特别是如何对计算机进行编程以处理和分析大量自然语言数据（非结构化的数据）。从科学的角度来看，NLP 旨在对人类语言理解和产生的认知机制进行建模。从工程角度来看，NLP 关注如何开发新颖的实际应用程序以促进计算机与人类语言之间的交互。在自然语言处理中，经常遇到的挑战包括语音识别、口语理解、对话系统、词汇分析、句法解析、机器翻译、知识图谱、信息检索、问题问答、情感分析、社交计算、自然语言生成和自然语言摘要等。当然，自然语言处理工作也是计算机科学中极其困难的工作任务。语言本身存在着各种各样的问题，亦因语言而异。

幸运的是，最近几年深度学习领域获得快速发展，使得深度学习算法在诸如图像分类、语音识别、文本生成、机器翻译等诸多带有很强挑战性的工作任务中表现优异，加速了深度学习与 NLP 各工作任务的深度融合，从而使得自然语言处理领域焕发出新的活力。而在深度学习被广泛应用的过程中，出现了多种技术框架，其中 TensorFlow 是目前最直观、最有效的深度学习框架之一。本书重点探讨如何利用 TensorFlow 深度学习框架去实现 NLP 的各种任务，例如句子分类、文档分类、文本生成、图像字幕自动生成、机器翻译、智能问答等。

在本章中，我们将要对于自然语言处理基础有一个初步了解，并对 NLP 的主要工作任务做一个划分；然后我们将对 NLP 领域的三个发展浪潮做详细解读，并对当前 NLP 领域中深度学习的局限性进行剖析；最后，我们还会对于 NLP 的应用场景和应用前景做个简要阐述。

1.1 认识自然语言处理

根据《2017 微信数据报告》显示，每天会有 380 亿条信息从微信上发出，如果按照每条信息都是文字“你吃饭了么”计算，通过微信发送的信息每天的数据量在 350GB 以上（一个汉字占 2 字节，1024 字节=1KB）。而实际的数据量会更多，因为这些信息会有不少语音、动画表情、图片、小视频等。其实，在实际工作中，我们每天都在处理的电子邮件、各类报告文档等同样也在以惊人的速度充斥着整个网络环境。2018 年 6 月，据科技公司 Domo 预测，到 2020 年，世界上每人每天将产生超过 140GB 的数据，并且随着物联网的迅猛发展，这个数字将会继续扩大。

正是由于这些统计数据的存在，才使得我们为界定 NLP 提供了良好的基础。简而言之，NLP 的目标就是让机器拥有真正理解人类语言并以与人类相同的方式处理它的能力。如今，NLP 的应用已经广泛存在，就像我们日常生活中常用到的虚拟助手（VA），常见的有百度语音助手、讯飞语音助手、Google 智能助理、微软的个人智能助理小娜（Cortana）、苹果系统的 Siri 等，这些虚拟助手主要是 NLP 系统在运行。比如，你告诉语音助手“请告诉我附近好吃的麻辣烫在哪儿？”首先，VA 需要将你的声音转换为文本（语音到文本）。接下来，VA 必须理解你请求的语义（例如，你正在寻找带有麻辣烫美食且好吃的餐厅）并制定结构化请求（例如，美食=麻辣烫，评级= 3-5，距离< 3 公里）。然后，VA 必须按位置和菜肴两个条件搜索并筛选出餐厅，再按收到的评级对餐厅进行排序。为了计算餐厅的整体评级，良好的 NLP 系统可以查看每个用户提供的评级和文本描述。最后，用户到达餐厅，VA 还可以将各种菜品组合的受欢迎程度进行综合推荐，以此来帮助你做出更好的选择。这个例子表明 NLP 已成为人类生活中不可或缺的一部分。

1.2 自然语言处理方面的任务

NLP 其实有许多实际的应用，而一个好的 NLP 系统会执行多个任务系统。比如，上面提到的你要在当前位置选择麻辣烫小吃店的例子，其实就是在执行多个 NLP 任务系统。关于 NLP 的任务，主要有以下几类：

- **标记化：**标记化是将文本语料库分离为原子单元（例如，单词）的任务。虽然看似微不足道，但是标记化却是一项非常重要的工作任务。例如，在日语中，单词不以空格或标点符号分隔。
- **词义消歧（Word-sense Disambiguation，WSD）**：词义消歧是识别单词正确含义的任务。例如，有两个句子，“你提供的图真好看”和“你图啥？”，其中“图”就有两种不同的含义。词义消歧对于诸如问答之类的任务至关重要。
- **命名实体识别（Named Entity Recognition，NER）**：NER 尝试从给定的文本主体或文本语料库中提取实体（例如，人、位置和组织）。例如，有一个句子，“林阿姨昨天在小区门口给了小明两瓶牛奶”，将被转换为 林阿姨$_{人}$ 昨天$_{时间}$ 在 小区门口$_{位置}$ 给了

小明$_{人}$两瓶$_{数量}$牛奶。NER 是信息检索和知识表示等领域的一个重要课题。

- **词性（Part-of-Speech，PoS）标注**：是词汇基本的语法属性，通常也称为词类，既可以是名词、动词、形容词、副词、介词等基本标签，也可以是诸如专有名词、普通名词、短语动词等。词性标注就是在给定句子中判定每个词的语法范畴，确定其词性并加以标注的过程，是中文信息处理面临的重要基础性问题，主要可以分为基于规则和基于统计的方法。
- **句子/概要分类**：句子或概要（例如，电影评论）分类有许多用例，例如垃圾邮件检测、新闻文章分类（诸如政治、科技和体育等）和产品评论评级（正面或负面）。这是通过训练带标签的数据（由人类注释的评论，带有积极或消极的标签）来训练分类模型实现的。
- **文本生成：**在文本生成中，学习模型（例如，神经网络）使用文本语料库（大量文本文档集合）进行训练，预测随后的新文本。例如，文本生成可以通过使用现有的小说故事文本进行训练来输出一个全新的小说故事文本。当然，具体的实现过程会涉及具体模型的实施，具有一定的复杂性。本书第 8 章将专门针对文本生成做详细解读。
- **问答（QA）系统**：QA 技术具有很高的商业价值，因为这些技术是聊天机器人和 VA（例如谷歌 Assistant 和苹果 Siri）实现的基础所在。聊天机器人已经被许多公司用于客户支持工作。聊天机器人可以用来回答和解决客户直接关心的问题，而不需要人工干预。QA 涉及 NLP 的许多方面，比如信息检索和知识图谱中的知识表示。因此开发 QA 系统变得更加具有挑战性。
- **机器翻译：**是将一个句子/短语从源语言（如汉语）转换为目标语言（如英语）的任务。这是一个非常具有挑战性的任务，因为不同的语言具有高度不同的形态结构，这意味着它不是一对一的转换。此外，语言之间的字对字关系可以是一对多、一对一、多对一或多对多。这在 MT 文献中被称为单词对齐问题。

为了开发一个可以帮助人们完成日常任务的系统（例如，VA 或聊天机器人），这些任务中的许多工作需要放在一起执行。正如我们在前面的例子中看到的那样，“请告诉我附近好吃的麻辣烫在哪儿？”需要完成几个不同的 NLP 任务，例如语音到文本转换、语义和情感分析、问题回答和机器翻译。在图 1.1 中，我们提供了不同 NLP 任务的层次分类。我们首先有两大类任务：分析（分析现有文本）和生成（生成新文本）任务。然后将分析分为三类：句法（基于语言结构的任务）、语义（基于意义的任务）和语用（难以解决的开放问题），如图 1-1 所示。

目前，我们对于自然语言处理及其各种任务分类有了一定的了解，下面我们将探讨一下 NLP 的起源、发展及现状。

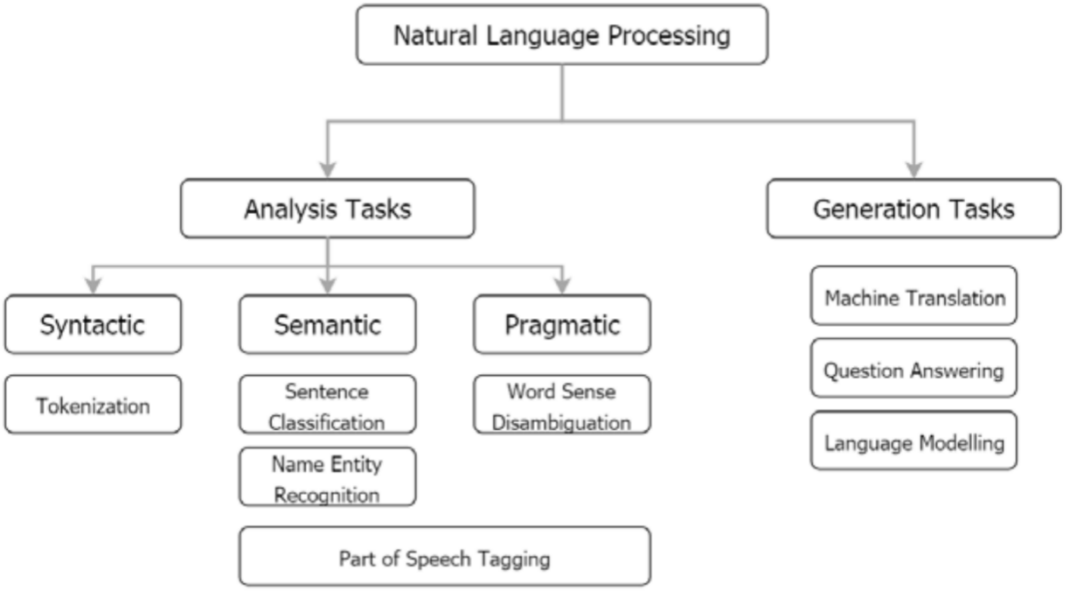

图 1-1　广义范畴下的 NLP 主要任务分类

1.3　第一阶段：偏理论的理性主义

NLP 研究的第一次浪潮持续了很长一段时间，可以追溯到 20 世纪 50 年代。1950 年，阿兰·图灵提出了图灵测试来评估计算机展示与人类无法区分的智能行为的能力（图灵，1950）（注意：本书这种“人名，年份”的说明方式用于读者必要时查阅下载资源文件，以确认参考文献的出处）。该测试基于人和计算机之间的自然语言对话，旨在产生类似人的响应。 1954 年，Georgetown-IBM 实验展示了第一个能够将 60 多个俄语句子翻译成英语的机器翻译系统。

这些方法基于这样一种信念，即人类大脑中的语言知识是通过一般遗传而提前固定下来的，这在 20 世纪 60 年代到 20 世纪 80 年代后期的 NLP 研究中占主导地位。这些方法被称为理性主义方法（Church，2007）。理性主义方法在 NLP 中占有主导地位，主要是由于诺姆·乔姆斯基关于先天语言结构的论点得到广泛接受，以及他对 N-gram 的批评（Chomsky，1957）。假设语言的关键部分在出生时就已经扎根于大脑，作为人类遗传的一部分，理性主义方法会努力设计人工制作的规则，将相关知识和推理机制融入智能 NLP 系统。直到 20 世纪 80 年代，最著名的 NLP 系统是基于复杂的手写规则集的，例如模拟罗格氏（Rogerian）心理治疗师的 ELIZA 和将现实世界的信息构造成概念本体的 MARGIE，是基于复杂的手写规则集。

这一时期大致与人工智能的早期发展相吻合，人工智能以专家知识工程为特征，行业专家根据他们所拥有的（非常狭窄的）应用领域的知识设计了计算机程序（Nilsson，1982；Winston，1993）。专家们使用基于细致的表示和工程学知识的符号逻辑规则来设计这些程序。这些基于知识的人工智能系统往往通过检查“大脑”或最重要的参数，并针对每个具体情况采取适当行动，从而有效地解决特定领域的问题。这些“大脑”参数由人类专家提前确定，使“尾部”参数和案例保持不变。由

于缺乏学习能力，很难将其解决方案推广到新的场景和领域。在此期间的典型方法是专家系统，例如模拟人类专家决策能力的计算机系统。这种系统旨在通过推理知识来解决复杂问题（Nilsson，1982）。第一个专家系统创建于 20 世纪 70 年代，而后在 20 世纪 80 年代兴起。使用的主要“算法”是“if-then-else”形式的推理规则（Jackson，1998）。这些第一代人工智能系统的主要优势在于其执行逻辑推理（有限的）能力的透明性和可解释性。就像 ELIZA 和 MARGIE 这样的 NLP 系统一样，早期的专家系统使用人工制作的专家知识库，这些知识在某些特定的问题中往往是有效的，尽管推理机制不能处理实际应用中普遍存在的不确定性。

对于语音识别的研究和系统设计，NLP 和人工智能面临的一个长期挑战是在很大程度上需要依赖于专家知识工程的范式，正如 Church 和 Mercer （Church 和 Mercer，1993）所分析的那样。在 20 世纪 70 年代和 80 年代初期，语音识别的专家系统方法非常受欢迎（Reddy，1976；Zue，1985）。然而，研究人员敏锐地认识到该阶段缺乏从数据中学习和处理推理中不确定性的能力，继而出现接下来描述的第二阶段语音识别、NLP 和人工智能。

1.4　第二阶段：偏实践应用的经验主义

该阶段 NLP 的特点是通过数据语料库和（浅）机器学习、统计或其他方法来使用数据样本（Manning 和 Schtze，1999）。由于自然语言的大部分结构和理论被数据驱动的方法所忽视或抛弃，所以这期间发展起来的主要方法被称为经验的（或实用的）方法（Church and Mercer，1993；Church，2014）。随着机器可读数据可用性的增加和计算能力的不断提高，从 1990 年开始，经验方法一直主导着 NLP。其中一个主要的 NLP 会议甚至被命名为“自然语言处理中的经验方法（EMNLP）”，以最直接地反映出 NLP 研究人员在该阶段对经验方法的强烈（积极）倾向性。

与理性主义方法相反，经验方法假设人类思维只从联想、模式识别和概括的一般操作着手。为了使得大脑更好地学习自然语言的详细结构，需要存在丰富的感官输入才可以。自 1920 年以来，经验主义在人口学中普遍存在，自 1990 年以来经验主义也一直在复苏。早期的 NLP 经验方法侧重于开发生成模型，如隐马尔可夫模型（HMM）（Baum 和 Petrie，1966）、IBM 翻译模型（Brown 等，1993）和脑部驱动的解析模型（Collins，1997）从大型语料库中发现语言的规律性。自 20 世纪 90 年代末以来，判别模型已成为各种 NLP 任务中实用的方法。NLP 中的代表性判别模型和方法包括最大熵模型（Ratnaparkhi，1997）、支持向量机（Vapnik，1998）、条件随机场（Lafferty 等，2001）、最大互信息和最小分类误差（He 等，2008）和感知器（Collins，2002）。

同样，NLP 中的经验主义时代与人工智能以及语音识别和计算机视觉中的方法相对应。这是因为有明确的证据表明，学习和感知能力对于复杂的人工智能系统至关重要，但在前一波流行的专家系统中却缺失了。例如，当 DARPA 开启其首次自动驾驶大挑战时，大多数车辆依赖于基于知识的人工智能范式。与语音识别和 NLP 非常相似，自动驾驶和计算机视觉研究人员立即意识到基于知识范式的局限性，因为机器学习必须具有不确定性处理和泛化能力。

NLP 中的经验主义和第二阶段中的语音识别是基于数据密集型的机器学习，我们现在称之为“浅层”机器学习，因为这里通常会缺少由多层或“深层”数据表示构成的抽象，第三阶段深度学

习方面将在后面继续开展。在机器学习中，研究人员无须关注构建第一阶段期间基于知识的 NLP 和语音系统所需的精确度和正确规则。他们关注统计模型（Bishop，2006；Murphy，2012）或简单的神经网络（Bishop，1995）作为潜在引擎。然后，他们使用充足的训练数据自动学习或“调整”引擎的参数，以使它们处理不确定性，并尝试从一个场景推广到另一个场景，从一个域到另一个域。用于机器学习的关键算法和方法包括 EM、贝叶斯网络、支持向量机、决策树及用于神经网络的反向传播算法。

现在回过头来看，基于机器学习的 NLP、语音识别和其他人工智能系统，比早期的基于知识的对应部分表现更佳。诸如一些成功的例子，包括机器知觉中的几乎所有人工智能任务——语音识别（Jelinek，1998）、人脸识别（Viola 和 Jones，2004）、视觉对象识别（Fei-Fei 和 Perona，2005）、手写识别（Plamondon 和 Srihari，2000）和机器翻译（Och，2003）。

具体来看，针对机器翻译应用方面，传统方法还是以统计方法主，我们也会在本书的第 11 章对机器翻译部分做详细解读

双语训练数据中句子级对齐的可用性使得不通过规则而是直接从数据中获得表层翻译成为可能，代价是丢弃或忽略自然语言中的结构化信息。当然，在本阶段的后续发展中，机器翻译的质量也得到了显著提升（Och 和 Ney，2002；Och，2003；Chiang，2007；He 和 Deng，2012），但还是没有达到现实世界中大规模部署的水平（后续深度学习阶段将会继续探讨）。

在 NLP 的对话和口语理解领域，这个经验主义时代也以数据驱动的机器学习方法为显著标志，这些方法非常适合于定量评价和具体可交付成果的要求。他们关注的是文本和域的更广泛但肤浅的表层覆盖，而不是对高度受限的文本和域的详细分析。我们训练数据的目的，不是从对话系统中设计出有关语言理解和动作反映方面的规则，而是从数据样本中自动学习（浅层）统计或神经模型方面的参数。这种学习有助于降低人工制作复杂对话管理器的设计成本，并有助于提高整体口语理解和对话系统中语音识别错误的鲁棒性水平（He 和 Deng，2013）。具体来看，对话系统中对话策略部分，在本阶段引入了基于马尔科夫决策过程的强化学习，有关评论，可以参阅 Young 等人的文章（Young 等，2013）。在口语理解方面，主要方法从第一阶段基于规则或模板的方法转移到生成模型，如隐马尔科夫模型（HMMs）（Wang 等，2011），再到判别模型，如条件随机场（Tur 和 Deng，2011）。

同样，在语音识别领域，从 20 世纪 80 年代早期到 2010 年左右，该领域主要由机器学习（浅）范式主导，使用基于与高斯混合模型集成的 HMM 的统计生成模型，以及不同版本的泛化方面（Baker 等，2009；Deng 和 O’Shaughnessy，2003；Rabiner 和 Juang，1993）。广义 HMMs 的许多版本是基于统计和神经网络的隐藏动态模型（Deng，1998；Bridle 等，1998；Deng 和 Yu，2007)。前者采用 EM 和扩展卡尔曼滤波算法来学习模型参数（Ma 和 Deng，2004；Lee 等，2004）；后者使用了反向传播（Picone 等，1999）。它们都广泛地利用了多个潜在的表示层来生成语音波形，遵循人类语音感知中长期存在的通过合成进行分析的框架。更重要的是，将这种“深层”生成过程转化为端到端判别过程的对应，引起了深度学习第一次在工业上的成功（Deng 等，2010，2013；Hinton 等，2012），形成了第三阶段的语音识别和 NLP 的驱动力，接下来我们将对此进行阐述。

1.5　第三阶段：深度学习阶段

虽然在第二阶段开发的 NLP 系统，包括语音识别、语言理解和机器翻译，比第一阶段开发系统的表现更好，具有更高的鲁棒性，但它们远未达到人类级别的水平，还有很多地方需要改进。除了少数例外，NLP 的（浅）机器学习模型通常没有足够大的容量来吸收大量的训练数据。此外，涉及的这些学习算法、方法和基础结构不够强大。所有这一切都在几年前发生了很大变化，由于深层结构化机器学习或深层学习的新范式推动（Bengio，2009；Deng 和 Yu，2014；LeCun 等，2015；Goodfellow 等，2016），引发了 NLP 的第三波浪潮。

在传统的机器学习中，由于特征是由人设计的，需要大量的人类专业知识，显然特征工程也存在一些瓶颈。同时，相关的浅层模型缺乏表示能力，因此缺乏形成可分解抽象级别的能力，这些抽象级别在形成观察到的语言数据时将自动分离复杂的因素。深度学习的进步是当前 NLP 和人工智能拐点背后的主要推动力，并且直接推动了神经网络的复兴，包括商业领域的广泛应用（Parloff，2016）。

进一步讲，尽管在第二次浪潮期间开发的许多重要的 NLP 任务中，判别模型（浅层）取得了成功，但它们仍然难以通过行业专家人工设计特征来涵盖语言中的所有规则。除了不完整性问题之外，这种浅层模型还面临稀疏性问题，因为特征通常仅在训练数据中出现一次，特别是对于高度稀疏的高阶特征。因此，在深度学习出现之前，特征设计已经成为统计 NLP 的主要障碍之一。深度学习为解决我们的特征工程问题带来了希望，其观点被称为“从头开始 NLP”（Collobert 等，2011），这在深度学习早期被认为是非同寻常的。这种深度学习方法利用了包含多个隐藏层的强大神经网络来解决一般的机器学习任务，而无须特征工程。与浅层神经网络和相关的机器学习模型不同，深层神经网络能够利用多层非线性处理单元的级联来从数据中学习表示以进行特征提取。由于较高级别的特征源自于较低级别的特征，因此这些级别构成了概念上的层次结构。

深度学习起源于人工神经网络，可以将其视为受生物神经系统启发的细胞类型的级联模型。随着反向传播算法的出现（Rumelhart 等，1986），从零开始训练深度神经网络在 20 世纪 90 年代受到了广泛的关注。其实在早期，由于没有大量的训练数据，也没有适当的设计模式和学习方法，在神经网络训练期间，学习信号在层与层之间传播时会随着层数呈指数级消失，难以调整深度神经网络的连接权重值，尤其是循环模式。Hinton 等人最初克服了这个问题（Hinton 等，2006），使用无监督的预训练，首先学习通常有用的特征检测器，然后通过监督学习进一步训练网络，进而对标记数据进行分类。因此，可以使用低级表示来学习高级表示的分布。这项开创性的工作标志着神经网络的复兴。此后各种网络架构被提出并开发出来，包括深度信念网络（deep belief networks）（Hinton 等，2006）、栈式自动编码器（stacked auto-encoders）（Vincent 等，2010）、深度玻尔兹曼机（deep Boltzmann machines）（Hinton 和 Salakhutdinov，2012）、深度卷积神经网络（Krizhevsky 等，2012）、深度堆叠网络（deep stacking networks）（Deng 等，2012）以及深度 Q 网络（Mnih 等，2015）。2010 年以来，深度学习能够发现高维数据中复杂的结构，已成功应用于人工智能的各种实际任务中，尤其是语音识别（Yu 等，2010；Hinton 等，2012）、图像分类（Krizhevsky 等，

2012；He 等，2016）和 NLP。

语音识别是 NLP 的核心任务之一，且它在工业 NLP 实际应用中受到深度学习很大的影响，所以我们这里对此进行一些解读。深度学习在大规模语音识别中的工业应用在 2010 年左右开始起飞。相关工作是由学术界和产业界合作发起的，最初的成果是在 2009 年 NIPS 语音识别和相关应用的深度学习研讨会上发布的。这次研讨会的目的是语音深层生成模型的局限性，以及大计算、大数据时代需要对深层神经网络进行认真研究的可能性。当时认为，使用基于对比散度学习算法（Contrastive Divergence Learning Algorithm）的深度信念网络生成模型进行 DNNs 预处理，可以克服 20 世纪 90 年代神经网络遇到的主要困难（Dahl 等，2011；Mohamed 等，2009）。然而，在微软早期关于这项的研究中，人们发现，没有对比散度预训练，而是使用大量的训练数据，连同深层神经网络，这些深层神经网络设计成具有相应的大型、上下文相关的输出层，并且经过精心的工程设计，可以获得比当时最先进的（浅）机器学习系统显著更低的识别误差（Yu 等，2010，2011；Dahl 等，2012）。北美的其他几个主要语音识别研究小组（Hinton 等，2012 年；Deng 等，2013 年）以及随后一些海外研究小组很快就证实了这一发现。此外，还发现这两种类型系统产生的识别错误的本质是不同的，这为如何将深度学习集成到现有的由主要参与者在语音识别中部署的高效运行的语音解码系统提供了技术支持（Yu 和 Deng，2015；Abdel-Hamid 等，2014；Xiong 等，2016；Saon 等，2017）。如今，应用于各种形式的深层神经网络的反向传播算法被统一应用于所有当前最先进的语音识别系统（Yu 和 Deng，2015；Amodei 等，2016；Saon 等，2017），以及所有主要的商业语音识别系统——微软 Cortana、Xbox、Skype 翻译、亚马逊 Alexa、谷歌助理、苹果 Siri、百度小度和 iFlyTek 语音搜索等——都基于深度学习方法。

2010 年、2011 年语音识别的惊人成功预示着第三波 NLP 和人工智能的到来。随着深度学习在语音识别领域的成功，计算机视觉（Krizhevsky 等，2012）和机器翻译（Bahdanau 等，2015）也很快被类似的深度学习范式所取代。特别是，虽然早在 2001 年就开发了强大的词汇神经词嵌入技术（Bengio 等，2001，Bengio 等人在 2001 年发表在 NIPS 上的文章《*A Neural Probabilistic Language Model*》，现在多数看到的是他们在 2003 年投到 JMLR 上的同名论文），但直到十多年后，由于大数据的可用性和计算机更快的计算能力，它才被证明在实际大规模场景下具有真正的价值（Mikolov 等，2013）。此外，还有大量的其他 NLP 应用，如图像字幕（Karpathy 和 Fei-Fei，2015；Fang 等，2015；Gan 等人，2017）、视觉问答（Fei-Fei 和 Perona，2016）、语音理解（Mesnil 等，2013）、网络搜索（Huang 等，2013）和推荐系统；由于深度学习的广泛应用，也有许多非 NLP 任务，像药物发现和毒理学、客户关系管理、手势识别、医学信息学、广告、医学图像分析、机器人、无人驾驶车辆和电子竞技游戏（例如，雅达利 Atari、Go、扑克和最新的 DOTA2）等。

在基于文本的 NLP 应用领域，机器翻译可能受到深度学习的影响最大。当前，在实际应用中表现最佳的机器翻译系统是基于深度神经网络的模式，例如，谷歌于 2016 年 9 月宣布开发第一阶段的神经网络机器翻译，而微软在 2 个月后发表了类似的声明。Facebook 已经致力于神经网络机器翻译一年左右，到 2017 年 8 月，它正在全面部署。最近，谷歌发布了机器翻译领域最强的 BERT 的多语言模型。BERT 的全称是 Bidirectional Encoder Representations from Transformers，是一种预训练语言表示的最新方法。

BERT 在机器阅读理解顶级水平测试 SQuAD1.1 中表现出惊人的成绩：两个衡量指标上全面超

越人类，而且在 11 种不同 NLP 测试中同样给出了最好的成绩，其中包括将 GLUE 基准推至 80.4%（绝对改进率 7.6%），MultiNLI 准确度达到 86.7%（绝对改进率 5.6%）等。对于机器翻译的解读，本书也会在第 11 章进行深入探讨。

在将深度学习应用于 NLP 问题的过程中，近年来出现的两个重要技术突破是序列到序列学习（Sutskevar 等，2014）和注意力建模（Bahdanau 等，2015）。序列到序列学习引入了一个强大的思想，即利用循环网络以端到端的方式进行编码和解码。虽然注意力建模最初是为了解决对长序列进行编码的困难，但随后的发展显然扩展了它的功能，能够对任意两个序列进行高度灵活的排列，且可以与神经网络参数一起进行学习。与基于统计学习和单词\短语的局部表示的最佳系统相比，序列到序列学习和注意力机制的关键思想提高了基于分布式嵌入的神经网络机器翻译的性能。在这一成功之后不久，这些概念也被成功地应用到其他一些与 NLP 相关的任务中，例如图像字幕生成（Karpathy 和 Fei-Fei，2015；Devlin 等，2015）、语音识别（Chorowski 等，2015）、句法解析、文本理解、问答系统等。

其实，基于神经网络的深度学习模型通常比早期开发的传统机器学习模型更易于设计。在许多应用中，以端到端的方式同时对模型的所有部分执行深度学习，从特征提取一直到预测。促成神经网络模型简化的另一个因素是相同模型构建的模块（例如不同类型的层）通常也可以适用于许多不同的任务。另外，还开发了软件工具包，以便更快更有效地实现这些模型。基于这些原因，深度神经网络现在是大型数据集（包括 NLP 任务）上的各种机器学习和人工智能任务的主要选择方法。

尽管深度学习已经被证明能够以革命性的方式对语音、图像和视频进行重塑处理，并且在许多实际的 NLP 任务中取得了经验上的成功，在将深度学习与基于文本的 NLP 进行交叉时，但其效果却不那么明显。在语音、图像和视频处理中，深度学习通过直接从原始感知数据中学习高级别概念，有效地解决了语义鸿沟问题。然而，在 NLP 中，研究人员在形态学、句法和语义学上提出了更强大的理论和结构化模型，提炼出了理解和生成自然语言的基本机制，但这些机制与神经网络并不容易兼容。与语音、图像和视频信号相比，从文本数据中学习到的神经表征似乎不能同样直接洞察自然语言。因此，将神经网络特别是具有复杂层次结构的神经网络应用于 NLP，近年来得到了越来越多的关注，也已经成为 NLP 和深度学习社区中最活跃的领域，并取得了显著的进步（Deng，2016；Manning 和 Socher，2017）。

1.6　NLP 中深度学习的局限性

目前，尽管深度学习在 NLP 任务中取得了巨大的成功，尤其是在语音识别/理解、语言建模和机器翻译方面，但目前仍然存在着一些巨大的挑战。目前基于神经网络作为黑盒的深度学习方法普遍缺乏可解释性，甚至是远离可解释性，而在 NLP 的理论阶段建立的“理性主义”范式中，专家设计的规则自然是可解释的。在现实工作任务中，其实是迫切需要从“黑盒”模型中得到关于预测的解释，这不仅仅是为了改进模型，也是为了给系统使用者提供有针对性的合理建议（Koh 和 Liang，2017）。

在许多应用中，深度学习方法已经证明其识别准确率接近或超过人类，但与人类相比，它需要更多的训练数据、功耗和计算资源。从整体统计的角度来看，其精确度的结果令人印象深刻，但从

个体角度来看往往不可靠。而且，当前大多数深度学习模型没有推理和解释能力，使得它们容易遭受灾难性失败或攻击，而没有能力预见并因此防止这类失败或攻击。另外，目前的 NLP 模型没有考虑到通过最终的 NLP 系统制定和执行决策目标及计划的必要性。当前 NLP 中基于深度学习方法的一个局限性是理解和推理句子间关系的能力较差，尽管在句子中的词间和短语方面已经取得了巨大进步。

目前，在 NLP 任务中使用深度学习时，虽然我们可以使用基于（双向）LSTM 的标准序列模型，且遇到任务中涉及的信息来自于另外一个数据源时可以使用端到端的方式训练整个模型，但是实际上人类对于自然语言的理解（以文本形式）需要比序列模型更复杂的结构。换句话说，当前 NLP 中基于序列的深度学习系统在利用模块化、结构化记忆和用于句子及更大文本进行递归、树状表示方面还存在优化的空间（Manning，2016）。

为了克服上述挑战并实现 NLP 作为人工智能核心领域的更大突破，有关 NLP 和深度学习研究人员需要在基础研究和应用研究方面做出一些里程碑式的工作。

1.7 NLP 的应用场景

目前，随着自然语言处理领域研究越来越深入，其应用的行业越来越广。比如在文本和语音方面的应用。其中，我们可以看到 NLP 在文本方面的应用有基于自然语言理解的智能搜索引擎和智能检索、智能机器翻译、自动摘要与文本综合、文本分类与文件整理、智能自动作文系统、智能判卷系统、信息过滤与垃圾邮件处理、文学研究与古文研究、语法校对、文本数据挖掘与智能决策以及基于自然语言的计算机程序设计等。在语音方面的应用有机器同声传译、智能远程教学与答疑、语音控制、智能客户服务、机器聊天与智能参谋、智能交通信息服务（ATIS）、智能解说与体育新闻实时解说、语音挖掘与多媒体挖掘、多媒体信息提取与文本转化以及对残疾人智能帮助系统等。下面我们给出一些常见的应用场景。

1．搜索引擎

在搜索引擎中，我们常常使用词义消歧、指代消解、句法分析等自然语言处理技术，以便更好地为用户提供更加优质的服务。因为我们的搜索引擎不仅仅是为用户提供所寻找的答案，还要做好用户与实体世界连接的贴心服务。搜索引擎最基本的模式就是自动化地聚合足够多的信息，对之进行解析、处理和组织，响应用户的搜索请求并找到对应结果再返回给用户。这里涉及的每一个环节，都需要用到自然语言处理技术。例如，我们日常生活中使用百度搜索“天气”“XX 公交线路”“火车票”等这样略显模糊的需求信息，一般情况下都会得到满意的搜索结果。自然语言处理技术在搜索引擎领域中有了更多的应用，才使得搜索引擎能够快速精准地返回给用户所要的搜索结果。当然，另一方面，正是谷歌和百度这样 IT 巨头商业上的成功，推进了自然语言处理技术的不断进步。

2．推荐系统

早在 1992 年 Goldberg 就首次给出了一个推荐系统：Tapestry。它其实只是一个个性化的邮件推荐系统，首次提出了协同过滤的思想，利用用户的标注和行为信息对邮件进行重排序。推荐系统

依赖的是数据、算法、人机交互等环节的相互配合，其中使用了数据挖掘、信息检索和计算统计学等技术。我们使用推荐系统的目的是关联用户和一些信息，协助用户找到对其有价值的信息，且让这些信息能够尽快呈现在对其感兴趣的用户面前，从而实现精准推荐。

推荐系统在音乐电影的推荐、电子商务产品推荐、个性化阅读、社交网络好友推荐等场景发挥着重要的作用，美国 Netflix 中 2/3 的电影是因为被推荐而观看的，Google News 利用推荐系统提升了 38%的点击率，Amazon 的销售中推荐占比高达 35%。

3．机器翻译

机器翻译是自然语言处理中最为人知的应用场景，一般是将机器翻译作为某个应用的组成部分，例如跨语言的搜索引流等。目前以 IBM、谷歌、微软为代表的国外科研机构和企业均相继成立机器翻译团队，专门从事智能翻译研究。例如，IBM 于 2009 年 9 月推出 ViaVoiceTranslator 机器翻译软件，为自动化翻译奠定了基础；2011 年开始，伴随着语音识别、机器翻译技术、DNN（深度神经网络）技术的快速发展和经济全球化的需求，口语自动翻译研究已成为当今信息处理领域新的研究热点，Google 于 2011 年 1 月正式在其 Android 系统上推出了升级版的机器翻译服务；微软的 Skype 于 2014 年 12 月宣布推出实时机器翻译的预览版、支持英语和西班牙语的实时翻译，并宣布支持 40 多种语言的文本实时翻译功能。

尤其值得注意的是，在“一带一路”这一发展背景下，合作沟通会涉及 60 多个国家、53 种语言，此时机器翻译的技术应用显得尤为重要，语言的畅通是“一带一路”倡议得以实施的重要基础。机器翻译涉及语义分析、上下文环境等诸多挑战，其发展道路还有很长一段路要走。

4．聊天机器人

聊天机器人是指能通过聊天 App、聊天窗口或语音唤醒 App 进行交流的计算机程序，是被用来解决客户问题的智能数字化助手，其特点是成本低、高效且持续工作。例如，Siri、小娜等对话机器人就是一个应用场景。除此之外，聊天机器人在一些电商网站有着很实用的价值，可以充当客服角色，例如京东客服 JIMI。有很多基本的问题，其实并不需要联系人工客服来解决。通过应用智能问答系统，可以排除掉大量的用户问题，比如商品的质量投诉、商品的基本信息查询等程式化问题，在这些特定的场景中，特别是会被问到高度可预测的问题中，利用聊天机器人可以节省大量的人工成本。图 1-2 给出了一些聊天机器人产品。

5．知识图谱

知识图谱能够描述复杂的关联关系，它的应用极为广泛，最为人所知的就是被用在搜索引擎中丰富搜索结果，并为搜索结果提供结构化结果来体现关联性，这也是谷歌提出知识图谱的初衷。同时微软小冰、苹果 Siri 等聊天机器人中也加入了知识图谱的应用。IBM Watson 是问答系统中应用知识图谱较为典型的例子。按照应用方式，可以将知识图谱的应用分为语义搜索、知识问答以及基于知识的大数据分析和决策等。

图 1-2　部分聊天机器人示意图

语义搜索利用建立大规模知识库对搜索关键词和文档内容进行语义标注，改善搜索结果，如谷歌、百度等在搜索结果中嵌入知识图谱。知识问答是基于知识库的问答，通过对提问句子的语义分析，将其解析为结构化的询问，在已有的知识库中获取答案。在大数据的分析和决策方面，知识图谱起到了辅助作用，典型应用是美国 Netflix 公司利用其订阅用户的注册信息以及观看行为构建的知识图谱反映出英剧版《纸牌屋》很受欢迎，于是拍摄了美剧《纸牌屋》，大受追捧。知识图谱展示如图 1-3 所示。

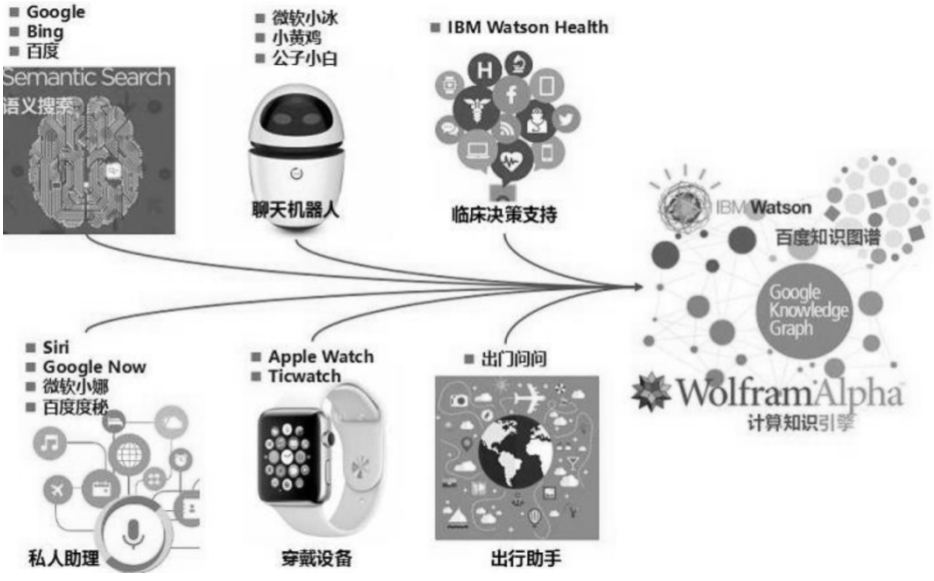

图 1-3　知识图谱展示图

1.8　NLP 的发展前景

随着深度学习时代的来临，神经网络成为一种强大的机器学习工具，并使得自然语言处理取得了许多突破性发展，如情感分析、智能问答、机器翻译等领域都在飞速发展。下面我们梳理一些自然语言处理近期热点和全球热点的情况。

1．文本理解与推理：浅层分析向深度理解迈进

谷歌等公司已经推出了以阅读理解作为深入探索自然语言理解的平台。

文本理解和推理是自然语言处理的重要部分，现在的机器软件已经可以根据文本的上下文来分辨代词等指示词，这是文本理解与推理从浅层分析向深度理解迈进的重要一步。

2．对话机器人：实用化、场景化

从最初 2012 年到 2014 年的语音助手，到 2014 年起逐渐出现的聊天机器人微软小冰、百度小度，再到 2016 年哈尔滨工业大学 SCIR 的笨笨，对话机器人越来越智能。最初的语音助手可以听得到但是听不懂，之后的对话机器人可以听得懂但是实用性却不强，现在对话机器人更多的是和场景结合，即在特定场景做有用的人机对话。

3．NLP+行业：与专业领域深度结合

银行、电器、医药、司法、教育等领域对自然语言处理的需求都非常多。自然语言处理与各行各业的结合越来越紧密，专业化的服务趋势逐渐增强。可以预测，自然语言处理首先会在信息准备充分并且服务方式本身就是知识和信息的领域产生突破，例如医疗、金融、教育和司法领域。

4．学习模式：先验语言知识与深度学习结合

自然语言处理中学习模式有一个较为明显的变化。在浅层到深层的学习模式中，浅层学习是分步骤的，深度学习的方法贯穿在浅层分析的每个步骤中，由各个步骤连接而成。而直接的深度学习则是直接从端到端，人为贡献的知识在深度学习中所占的比重大幅度减小。但如何将深度学习应用于自然语言处理需要进行更多的研究和探索，针对不同任务的不同字词表示，将先验知识和深度学习相结合是未来的一个发展趋势。

5．文本情感分析：事实性文本到情感文本

之前的研究主要是新闻领域的事实性文本，现在情感文本分析更受重视，并且在商业和政府舆情上可以得到很好的应用。例如，2017 年新浪微舆情和哈尔滨工业大学推出“情绪地图”，网民可以登录新浪舆情官方网站查询任何关键词的“情绪地图”，这是语义情绪分析在舆情分析产业上的首次正式应用。

1.9 总结

在本章中，为了建立本书的基本框架，我们首先解释了为什么我们需要 NLP，然后讨论了 NLP 的各类任务。对于 NLP 的来龙去脉，我们又从理性主义和经验主义到当前深层学习浪潮的三次自然语言处理浪潮出发，回顾了自然语言处理领域几十年来的历史发展情况，以便从历史发展中提炼出有助于指导未来方向的见解。接着，我们对于当前 NLP 中深度学习的局限性进行了解读。最后，我们对于 NLP 中的应用场景和前景做了简述。

首先，我们通过一个日常生活中常见的寻找小吃店位置的例子开启了解读 NLP 的篇章，并对 NLP 主要的工作任务进行了初步划分，包括标记化、词义消歧、词性标注、实体命名识别、句子或概要分类、文本生成、问答系统、机器翻译等。

其次，通过对于 NLP 发展的三个阶段的分析，我们知道当前的 NLP 深度学习技术是从前两波发展起来的一种 NLP 技术概念和范式上的革新。这场革新的关键支撑包括通过嵌入对语言实体（子单词、单词、短语、句子、段落、文档等）的分布式表示、嵌入引起的语义概括、语言的大跨度深层序列建模、能够从低到高有效表达语言水平的层次网络以及端到端的深度学习方法，以共同解决许多 NLP 任务中的问题。在深度学习浪潮出现之前，这些都是不可能实现的，这不仅是因为之前的两次发展浪潮缺乏大数据和强大的计算能力，更重要的是在于近年来深度学习范式出现了之前缺少的正确框架。

接着，我们对于当前 NLP 领域深度学习方面的局限性进行了解读，并给出了局限性存在的成因和简要的解决之道。

最后，我们对于 NLP 的常见应用领域进行了说明并给出了 NLP 的发展前景。

总之，深度学习开创了一个新的世界，使得 NLP 比过去任何时候都更具有活力。深度学习不仅提供了一个强大的建模框架，用于表示计算机系统中人类自然语言的认知能力，更重要的是，它已经在 NLP 的许多关键应用领域创造了卓越的实际效果。在本书的其余章节中，将提供使用深度学习框架开发（具体利用 TensorFlow 工具）的 NLP 技术的详细描述，同时也希望本书能够对 NLP 领域的人员有一些帮助，以便使得 NLP 领域有更多突破性成果的出现。接下来的一章，我们将介绍深度学习基础。

第 2 章

深度学习基础

2.1　深度学习介绍

具有机器学习基础的朋友可能都知道，机器学习实际上是一个寻找最优模型的过程。深度学习是机器学习的一个扩展领域，其概念源自于科学家对人工网络长期研究的积累，深度学习的基本结构其实也是深度神经网络。在深度学习下实现的算法集合与人类大脑中的刺激和神经元之间的关系具有相似之处。深度学习在计算机视觉、语言翻译、语音识别、图像识别等方面具有广泛的应用。这些算法集很简单，既可以在有监督的情况下学习，也可以在无监督的情况下学习。

大多数的深度学习算法都是基于人工神经网络的理念，数据丰富，计算资源充足，使得目前世界上对这种算法的训练变得更加容易、深度学习模型的性能得到不断提升。对于这一点，我们可以在图 2-1 中看到更好的表示。

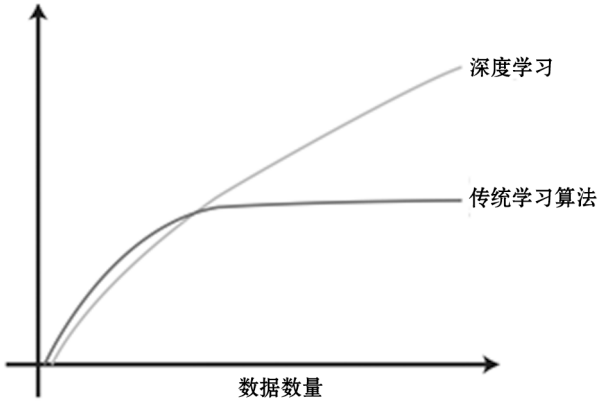

图 2-1　基于神经网络的深度学习算法和传统学习算法效果图

这里深度学习中的“深度”是指人工神经网络结构的深度，而“学习”是指通过人工神经网络

本身进行学习。图 2-2 给出了深度和浅度网络之间差异的展示，以及为什么“深度学习”会受到大家的热捧。

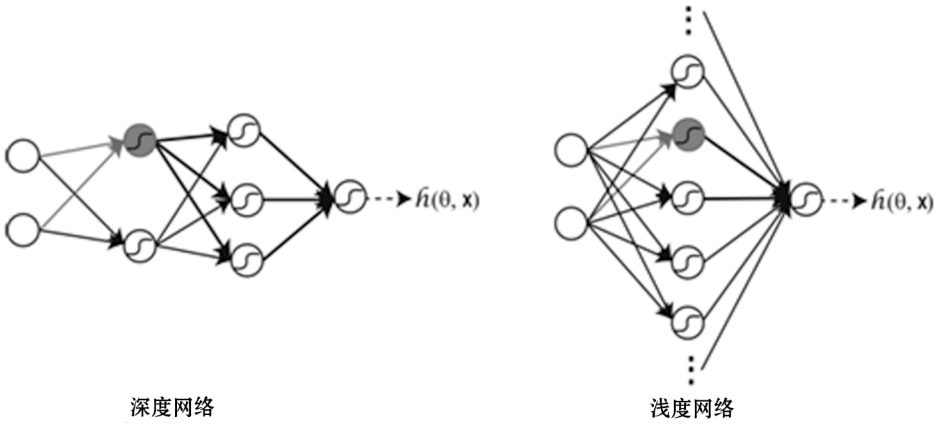

图 2-2　深度和浅度网络示意图

通过利用深度神经网络，我们能够从未标记和非结构化数据（如图像（像素数据）、文档（文本数据）或文件（音频、视频数据））中寻找出潜在的结构情况（或特征学习）。实际上，虽然深度学习中的人工神经网络和模型在本质上具有相似的结构，但这并不意味着两个人工神经网络的组合在利用数据进行训练时，我们获得的表现（或效果）与深度神经网络就相似。其中一个重要的原因是，深度神经网络与普通人工神经网络之间所使用的反向传播方式不同。

接下来，我们先介绍一下深度学习的演变过程，然后从神经网络开始详细解读。

2.2　深度学习演变简述

2.2.1　深度学习早期

1943 年，心理学家麦卡洛克和数学逻辑学家皮兹发表了论文《神经活动中内在思想的逻辑演算》，提出了 MP 模型。MP 模型是模仿神经元的结构和工作原理构建的一个基于神经网络的数学模型，本质上是一种“模拟人类大脑”的神经元模型。MP 模型作为人工神经网络的起源，开创了人工神经网络的新时代，也奠定了神经网络模型的基础。

1949 年，加拿大著名心理学家唐纳德·海布在《行为的组织》中提出了一种基于无监督学习的规则——海布学习规则（Hebb Rule）。海布学习规则模仿人类认知世界的过程建立一种“网络模型”，该网络模型针对训练数据集进行大量的训练并提取训练集的统计特征，然后按照样本的相似程度进行分类，把相互之间联系密切的样本分为一类，这样就把样本分成了若干类。海布学习规则与“条件反射”机理一致，为后面的神经网络学习算法奠定了基础，具有重大的历史意义。

20 世纪 50 年代末，在 MP 模型和海布学习规则的研究基础上，美国科学家罗森布拉特发现了一种类似于人类学习过程的学习算法——感知机学习，并于 1958 年正式提出了由两层神经元组成

的神经网络，称之为“感知器”。感知器本质上是一种线性模型，可以对输入的训练集数据进行二分类，且能够在训练集中自动更新权重值。感知器的提出吸引了大量科学家对人工神经网络研究的兴趣，对神经网络的发展具有里程碑式的意义。

随着研究的深入，在 1969 年，“AI 之父”马文·明斯基和 LOGO 语言的创始人西蒙·派珀特共同编写了一本图书《感知器》，在书中他们证明了单层感知器无法解决线性不可分问题（例如异或问题）。由于这个致命的缺陷以及没有及时推广感知器到多层神经网络中，在 20 世纪 70 年代人工神经网络进入了第一个寒冬期，人们对神经网络的研究也停滞了将近 20 年。

2.2.2　深度学习的发展

1982 年，著名物理学家约翰·霍普菲尔德发明了 Hopfield 神经网络。Hopfield 神经网络是一种结合存储系统和二元系统的循环神经网络。Hopfield 网络也可以模拟人类的记忆，根据激活函数的选取不同，有连续型和离散型两种类型，分别用于优化计算和联想记忆。由于容易陷入局部最小值的缺陷，该算法并未在当时引起很大的轰动。

直到 1986 年，深度学习之父杰弗里·辛顿提出了一种适用于多层感知器的反向传播算法——BP 算法。BP 算法在传统神经网络正向传播的基础上，增加了误差的反向传播过程。反向传播过程不断地调整神经元之间的权重值和阈值，直到输出的误差减小到允许的范围之内，或达到预先设定的训练次数为止。BP 算法完美地解决了非线性分类问题，让人工神经网络再次引起了人们广泛的关注。

20 世纪 80 年代计算机的硬件水平有限，比如运算能力跟不上，导致当神经网络的规模增大时使用 BP 算法出现“梯度消失”的问题。这使得 BP 算法的发展受到了很大的限制。再加上 20 世纪 90 年代中期，以 SVM 为代表的其他浅层机器学习算法的提出，及其在分类、回归问题上均取得了很好的效果，其原理又明显不同于神经网络模型，所以人工神经网络的发展再次进入了瓶颈期。

2.2.3　深度学习的爆发

2006 年，杰弗里·辛顿以及他的学生鲁斯兰·萨拉赫丁诺夫正式提出了深度学习的概念。他们在世界顶级学术期刊《科学》发表的一篇文章中详细地给出了“梯度消失”问题的解决方案——通过无监督的学习方法逐层训练算法，再使用有监督的反向传播算法进行调优。该深度学习方法的提出，立即在学术圈引起了巨大的反响，以斯坦福大学、多伦多大学为代表的众多世界知名高校纷纷投入巨大的人力、财力进行深度学习领域的相关研究，而后又迅速蔓延到工业界中。

2012 年，在著名的 ImageNet 图像识别大赛中，杰弗里·辛顿领导的小组采用深度学习模型 AlexNet 一举夺冠。AlexNet 采用 ReLU 激活函数，从根本上解决了梯度消失问题，并采用 GPU 极大地提高了模型的运算速度。同年，由斯坦福大学的吴恩达教授和世界顶尖计算机专家 Jeff Dean 共同主导的深度神经网络——DNN 技术在图像识别领域取得了惊人的成绩，在 ImageNet 评测中成功地把错误率从 26%降低到了 15%。深度学习算法在世界大赛中脱颖而出，再一次吸引了学术界和工业界对深度学习领域的关注。

随着深度学习技术的不断进步以及数据处理能力的不断提升，2014 年，Facebook 基于深度学习技术的 DeepFace 项目，在人脸识别方面的准确率已经能达到 97%以上，跟人类识别的准确率几乎没有差别。这样的结果也再一次证明了深度学习算法在图像识别方面的一骑绝尘。

2016 年，随着谷歌公司基于深度学习开发的 AlphaGo 以 4:1 的比分战胜了国际顶尖围棋高手李世石，深度学习的热度一时无两。后来，AlphaGo 又接连和众多世界级围棋高手过招，均取得了完胜。这也证明了在围棋界，基于深度学习技术的机器人已经超越了人类。

2017 年，基于强化学习算法的 AlphaGo 升级版 AlphaGo Zero 横空出世。其采用“从零开始”“无师自通”的学习模式，以 100:0 的比分轻而易举打败了之前的 AlphaGo。除了围棋，它还精通国际象棋等其他棋类游戏，可以说是真正的棋类“天才”。此外在这一年，深度学习的相关算法在医疗、金融、艺术、无人驾驶等多个领域均取得了显著的成果。所以，也有专家把 2017 年看作是深度学习甚至是人工智能发展最为突飞猛进的一年。

2.3 神经网络介绍

神经网络这个词，其实是一个非常泛的概念，有两个类别：生物神经网络和人工神经网络。生物神经网络是研究生物学的，一般是指生物的大脑神经元、细胞、触点等组成的网络，用于产生生物的意识、帮助生物进行思考和行动。人工神经网络（Artificial Neural Networks，ANN）有时也称为神经网络（NN）或连接模型（Connection Model），它是一种模仿动物神经网络的行为特征，进行分布式并行信息处理的算法数学模型。这种网络依靠系统的复杂程度，通过调整内部大量节点之间相互连接的关系，从而达到处理信息的目的。人工神经网络是一种应用类似于大脑神经突触联接的结构进行信息处理的数学模型。在工程与学术界也常直接简称为“神经网络”或类神经网络。所以，我们有必要先简要介绍一下生物神经网络中的神经元模型。

神经元

对于神经元的研究由来已久，1904 年生物学家就已经知晓了神经元的组成结构。一个神经元通常具有多个树突，主要用来接受传入信息；而轴突只有一条，轴突尾端有许多轴突末梢可以给其他多个神经元传递信息。轴突末梢跟其他神经元的树突产生连接，从而传递信号。这个连接的位置在生物学上叫作“突触”。神经元的基本功能是通过接受、整合、传导和输出信息实现信息交换。这样一来，神经元按照用途可以分为三种：输入神经、传出神经和连体神经。人脑中的神经元形状如图 2-3 所示。

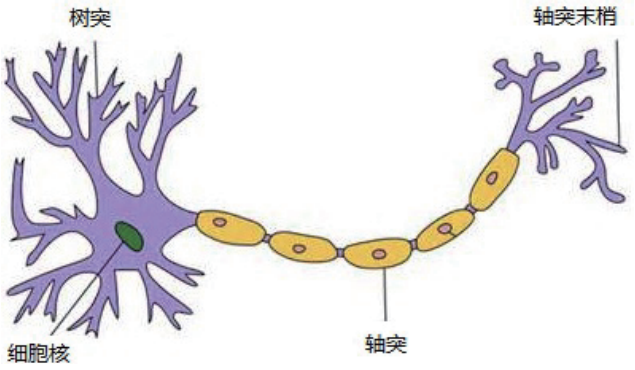

图 2-3　人脑中的神经元形状图

2.4　神经网络的基本结构

神经网络背后的原理源自于一些基本元素组成的集合，如人工神经元或感知器（Perceptron），这些元素最早是由弗兰克·罗森布拉特（Frank Rosenblatt）在 20 世纪 50 年代开发的。它们有几个二进制输入（0 或 1），x_1,x_2,…,x_n，如果这些输入的总和大于激活电位（等同于偏差），则产生单个二进制的输出。每当超过激活电位时，神经元被称为“激活”，并表现为一个阶跃函数（step function）。发射信号的神经元将信号传递给与其树突相连的其他神经元，如果超过激活电位，树突就会激活信号，从而产生级联效应，如图 2-4 所示，其实这也是早期的单层神经网络（感知器）。

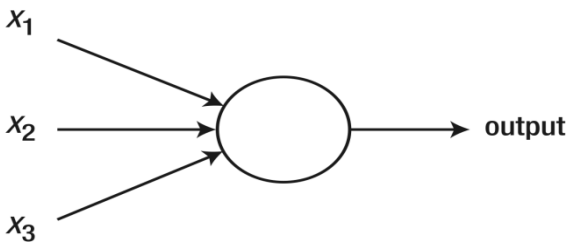

图 2-4　神经元样例示意图

由于并非所有输入都具有相同的重要性，因此每个输入 x_i 都被分配了自己的权重值，允许模型为某些输入指定更高的权重值。因此，若加权和大于激活电位或偏差，则输出为 1，即可以给出输出的结果，具体如下：

$$output = \sum_j w_j x_j + Bias \tag{2.1}$$

实际上，阶跃函数的跳跃性质使得它具有不连续、不光滑、不可导等特点，如果利用这种简单形式的函数，我们很难达到理想的效果，如图 2-5 所示。因此，在实际应用中，我们通常不会直接采用阶跃函数来作为激活函数，我们需要对其进行两处修改，以使其表现得更可预测，即权重值和偏差的微小变化仅导致输出的微小变化，这两处具体如下：

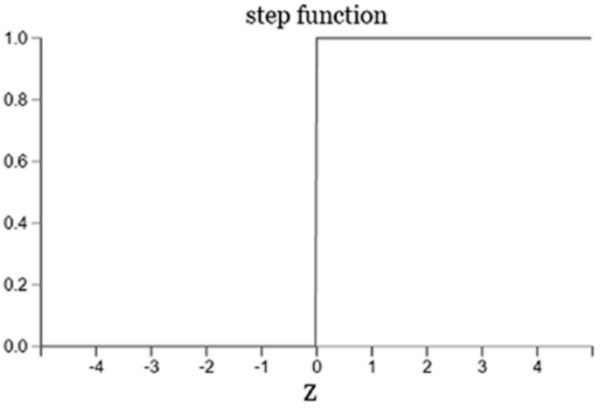

图 2-5　阶跃函数示意图

（1）输入可以取 0~1 之间的任何值，而不是二进制（0 或 1）；

（2）为了使输出在给定的输入$x_1, x_2, \ldots, x_N$、权重值$w_1, w_2, \ldots, w_N$和偏差b下更平稳地工作，我们使用以下 Sigmoid 函数（见图 2-6）：

$$\sigma(x_i, w_i) = \frac{1}{1+\exp(-\sum_j x_j w_j - b)} \tag{2.2}$$

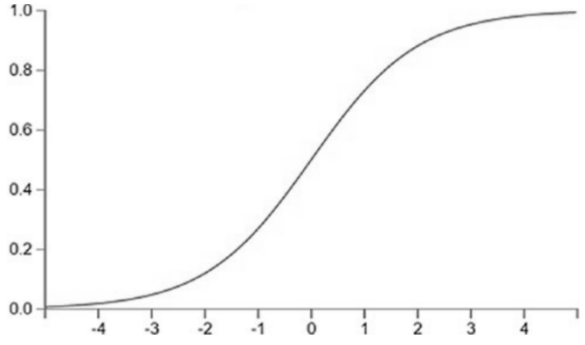

图 2-6　Sigmoid 函数

指数函数（或σ）的平滑度意味着权重值和偏差的微小变化将引起神经元输出的微小变化（变化可能是权重值和偏差变化的线性函数）。

除了通常的 Sigmoid 函数外，更常用的其他非线性函数还包括以下几个函数，它们中的每一个都可能具有类似或不同的输出范围，因此可以相应地使用。

ReLU：线性整流函数（Rectified Linear Unit，ReLU），又称修正线性单元，是一种人工神经网络中常用的激活函数（Activation Function），通常指代以斜坡函数及其变种为代表的非线性函数。这将使激活保持在 0 位，使用以下函数计算：

$$Z_j = f_j(x_j) = \max(0, x_j) \tag{2.3}$$

其中，x_j是第 j 个输入值，z_j是经过 ReLU 函数 f 处理过的相应输出值。ReLU 函数的图形如图 2-7 所示，对于所有 x≤0，函数值为 0；对于所有 x> 0，函数线性斜率为 1。

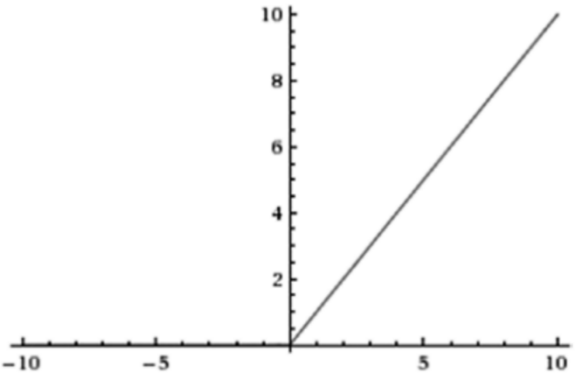

图 2-7　ReLU 函数

ReLU 经常面临着“死亡”的问题，特别是当学习率被设置为更高的数值时，因为这会触发不允许激活特定神经元的权重值更新，从而使该神经元的梯度永远为零。 ReLU 提供的另一个风险是激活函数的爆炸，因为输入值x_j本身就是输出。ReLU 也存在不少优点，例如在x_j低于 0 的情况下引入稀疏性，引起稀疏表示，并且当 ReLU 不变时返回梯度，它会导致更快的学习，伴随着梯度消失的可能性会降低。

下面给出其他几个常见的函数，这些函数能够使我们很容易对梯度下降进行训练，具体如下：

- LReLU（Leaky ReLU）：通过为 x 小于 0 的值引入略微减小的斜率（约 0.01）来减轻 ReLU 死亡的问题，LReLU 确实提供了很多成功的场景，尽管有时也会出错。
- ELU（指数线性单位）：它们提供负值，通过将附近的梯度移动到单位自然梯度，将平均单位激活推至接近零，从而加快学习过程。有关 ELU 更详细的解释，请参阅 Djork-ArnéClevert 的原始论文，网址为 https://arxiv.org/abs/1511.07289。
- softmax：也被称为归一化指数函数，它转换（0,1）范围内的一组给定实值，使得组合和为 1。softmax 函数表示如下：

$$\sigma(z)_j = e^{zk} / \sum_{k=1}^{k} e^{zk} \qquad (2.4)$$

这里，$j = 1,2,\ldots,K$。

与哺乳动物大脑一样，单个神经元也是分层组织的，在一层内与下一层相连，形成一个 ANN 或者说人工神经网络或多层感知器（MLP）。这样，整个网络的复杂性就是基于这些元素和连接相邻层的数量情况。

输入和输出之间的层称为隐藏层，层之间的连接密度和类型是结构（也叫配置）。例如，一个完全连接的结构将 L 层的所有神经元连接到 L+1 层的所有神经元。若要具有更明显的结构，我们只能将一个局部邻域连接到下一层。图 2-8 显示了两个具有密集连接的隐藏层。

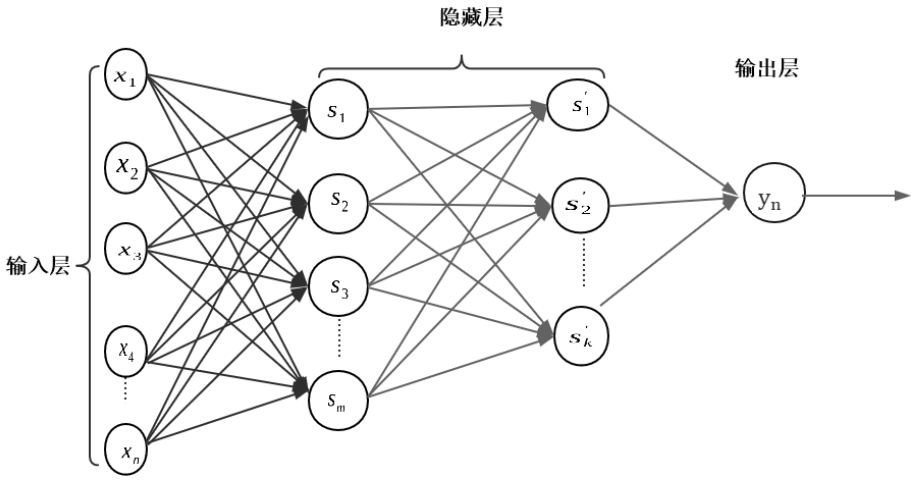

图 2-8　神经网络架构示例图

2.5　两层神经网络（多层感知器）

2.5.1　简述

两层神经网络是本文的重点，因为正是从两层结构为起点神经网络才开始了大范围的推广与使用。

大家知道，单层神经网络无法解决异或问题，但是当增加一个计算层以后，两层神经网络不仅可以解决异或问题，而且具有非常好的非线性分类效果。不过，两层神经网络的计算则是一个问题，因为没有一个较好的解法。

1986 年，Rumelhar 和 Hinton 等人提出了反向传播（Backpropagation，BP）算法，解决了两层神经网络所需要的复杂计算量的问题，从而带动了业界使用两层神经网络研究的热潮。目前，大量讲解神经网络的教材之内容基本都是以介绍两层（带一个隐藏层）神经网络为重点。

当时的 Hinton 还很年轻，30 年以后，正是他重新定义了神经网络，带来了神经网络复苏的又一波发展浪潮。

2.5.2　两层神经网络结构

两层神经网络除了包含一个输入层和一个输出层以外，还增加了一个中间层。此时，中间层和输出层都是计算层。我们扩展上节的单层神经网络，在右边新加一个层次。

现在，我们的权重值矩阵增加到了两个，我们用上标来区分不同层次之间的变量。

例如，a_x^y代表第y层中的第x个节点。z_1, z_2变成了a_1^2, a_2^2。图 2-9 给出了a_1^2, a_2^2的计算公式。

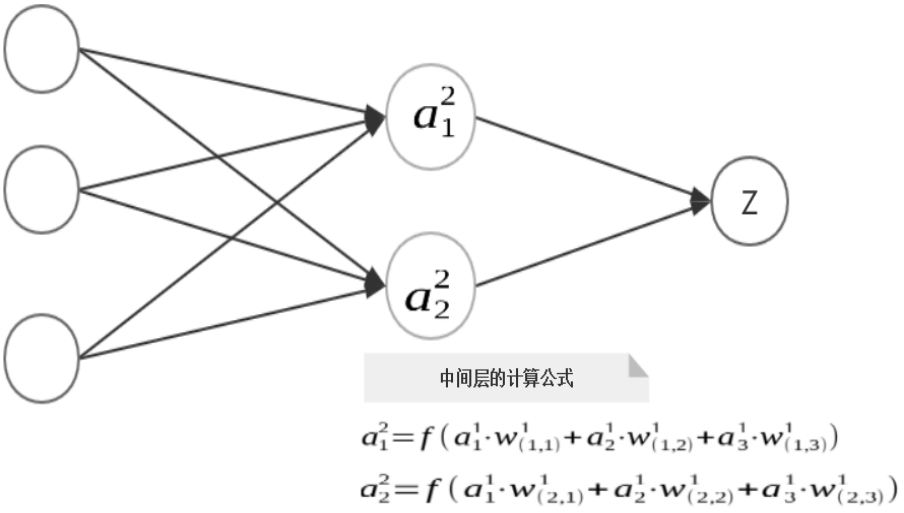

图 2-9　两层神经网络（中间层计算）

计算最终输出 z 的方式是利用了中间层的a_1^2, a_2^2和第二个权值矩阵计算得到的，如图 2-10 所示。

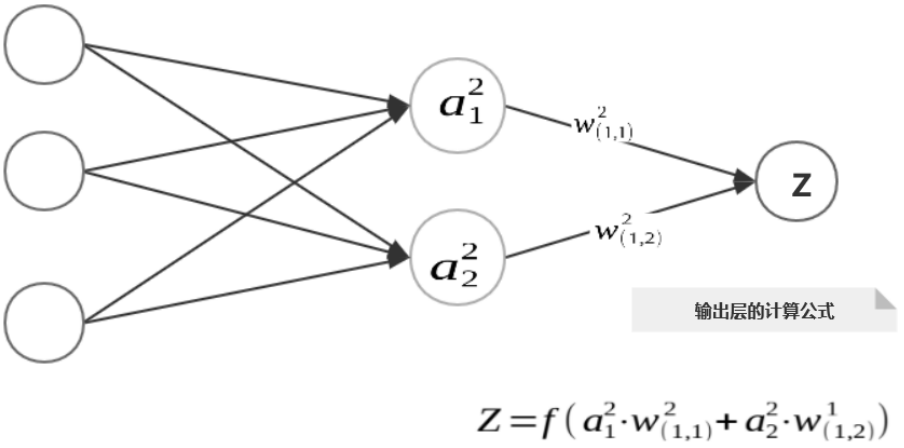

图 2-10　两层神经网络（输出层计算）

假设我们的预测目标是一个向量，那么与前面类似，只需要在“输出层”再增加节点即可。

2.6　多层神经网络（深度学习）

2.6.1　简述

人工神经网络在被人摒弃的 10 年中，有几个学者仍然在坚持研究，其中的典型代表就是加拿大多伦多大学的 Geoffery Hinton 教授。

2006 年，Hinton 在《*Science*》和相关期刊上发表了论文，首次提出了“深度信念网络”的概

念。与传统的训练方式不同，“深度信念网络”有一个“预训练”（Pre-Training）的过程，可以方便地让神经网络中的权重值找到一个接近最优解的值，之后再使用“微调”（Fine-Tuning）技术来对整个网络进行优化训练。这两个技术的运用大幅度减少了训练多层神经网络的时间。他给多层神经网络相关的学习方法赋予了一个新名词——“深度学习”。

很快，深度学习在语音识别领域崭露头角。接着，2012 年，深度学习技术又在图像识别领域大展拳脚。Hinton 与他的学生在 ImageNet 竞赛中用多层的卷积神经网络成功地对包含一千类别的一百万张图片进行了训练，取得了分类错误率 15%的好成绩，这个成绩比第二名高了近 11 个百分点，这充分证明了多层神经网络识别效果的优越性。

在这之后，关于深度神经网络的研究与应用不断涌现。

关于 CNN（Conventional Neural Network，卷积神经网络）与 RNN（Recurrent Neural Network，循环神经网络）及其变体 LSTM 的架构，我们会在后面的章节中进行单独解读，这里不做过多阐释。

2.6.2 多层神经网络结构

我们延续两层神经网络的方式来设计一个多层神经网络。

在两层神经网络的输出层后面，继续添加层次。原来的输出层变成中间层，新加的层次成为新的输出层，可以得到图 2-11。

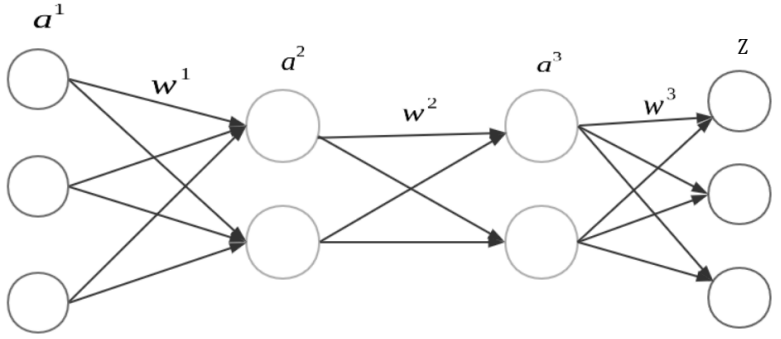

图 2-11 多层神经网络

依照这样的方式不断添加，我们可以得到更多层的多层神经网络。公式推导的话其实跟两层神经网络类似，使用矩阵运算的话就仅仅是加一个公式而已。

2.7 编码器-解码器网络

编码器-解码器（Encoder-Decoder）网络使用一个网络来创建输入的内部表示，或者对其进行“编码”，并且该表示用作另一个网络的输入以产生输出。这有助于超越输入的分类。最终输出可以是相同的模态，即语言翻译，或基于概念的不同模态，例如图像的文本标记。作为参考，可以阅

读谷歌团队发表的论文“使用神经网络进行序列学习”（https://papers.nips.cc/paper/5346-sequence-to-sequence-learning-with-neural-networks.pdf）。

编码器-解码器网络其实可以看作一个解决问题的框架，主要解决 seq2seq 类问题，Sequence（序列）在这里可以理解为一个字符串序列，当我们在给定一个字符串序列后，希望得到与之对应的另一个字符串序列，比如问答系统、翻译系统。

编码器-解码器网络的流程可以理解为“编码—存储—解码”这一流程，可以用人脑流程来类比。我们先看到源 Sequence，将其读一遍，然后在我们大脑当中就记住了这个源 Sequence，并且存在大脑的某一个位置上，形成我们自己的记忆（对应后面的 Context），然后我们再经过思考，将这个大脑里的东西转变成输出，写下来。我们大脑读入的过程叫作编码器（Encoder），即将输入的东西变成我们自己的记忆，放在大脑当中，而这个记忆可以叫作 Context，然后我们再根据这个 Context 转化成答案写下来，这个写的过程叫作解码器（Decoder）。

整个模型可以用图 2-12 表示。

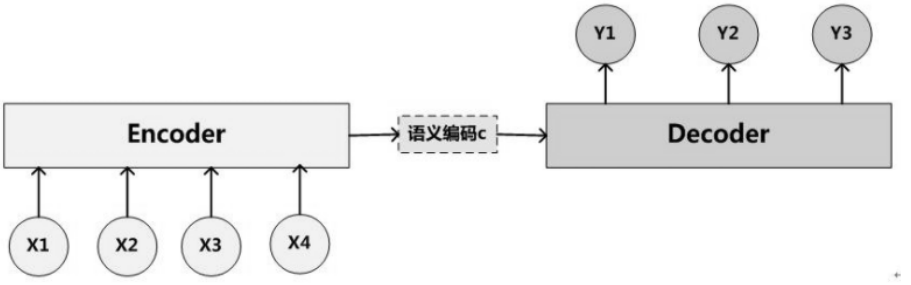

图 2-12　编码器-解码器（Encoder-Decoder）网络架构

2.8　随机梯度下降

几乎所有优化问题的解决方案都是梯度下降算法。它是一种迭代算法，通过随后更新函数的参数来最小化损失函数。

从图 2-13 中可以看到，我们首先把函数想象成一种山谷，想象一个球滚下山谷斜坡的情形。日常生活经验告诉我们，球最终会滚到谷底。也许我们可以用这个方法求出损失函数的最小值。

我们这里使用的函数依赖于两个变量：v1 和 v2。这可能是显而易见的，因为我们的损失函数看起来像前面的那个。为了达到这样一个平滑的损失函数，我们取二次损失：

$$(y - y^{\text{predicted}})^2$$

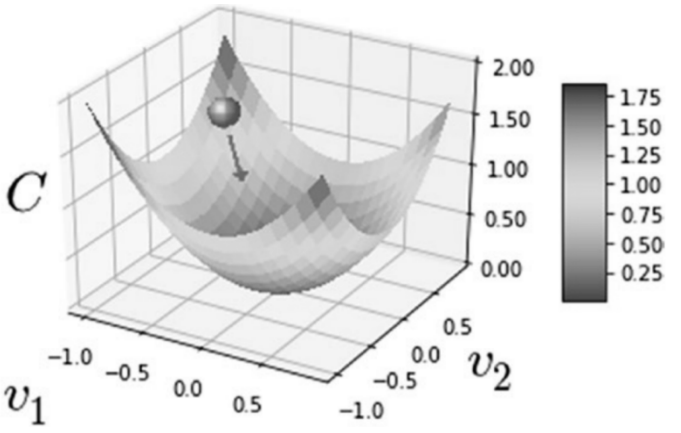

图 2-13　球滚下斜坡示意图

同样，我们应该注意到二次损失函数只是一种方法，其实还有许多其他方法来定义损失。但不管选择哪种方法，最终，我们选择不同损失函数的目的是为了得到：

（1）对权重值的平滑偏导数。

（2）一个良好的凸曲线，可以达到全局最小值。然而，在寻找全局最小值时，还有许多其他因素也在发挥作用。

我们随机选择一个（假想的）球的起点，然后模拟球在向下滚动到山谷底部时的运动。与之类比的情景中，假设我们初始化网络的权重值，或者一般来说，在曲线上的某个任意点上初始化函数的参数（就像在斜坡的任何一点上放一个球），然后检查附近的斜率（导数）。

我们知道，由于重力的作用，球会朝着最大坡度的方向下落。同样，在该点沿导数方向移动权重值，并根据以下规则更新权重值：

设 $J(w)$ = 成本作为权重值的函数，w =网络参数（$v1$ 和 $v2$），w_i=初始权重值集（随机初始化）。

$$w_{update} = w_i - \eta dJ\,(w)/dw$$

这里，$dJ(w)/dw$ 为权重值 w 对 $J(w)$的偏导数，η 是学习率（Learning Rate）。

学习率更多的是一个超参数，这里虽然没有固定的方法来找到最合适的学习率，但是我们总是可以通过批量损失来找到它。

一种方法是看到损失并分析损失的模式。一般来说，较差的学习率会导致小批量的不稳定损失。它（损失）可以递归地上升和下降，而不需要稳定。

图 2-14 给出了一个更直观的解释。

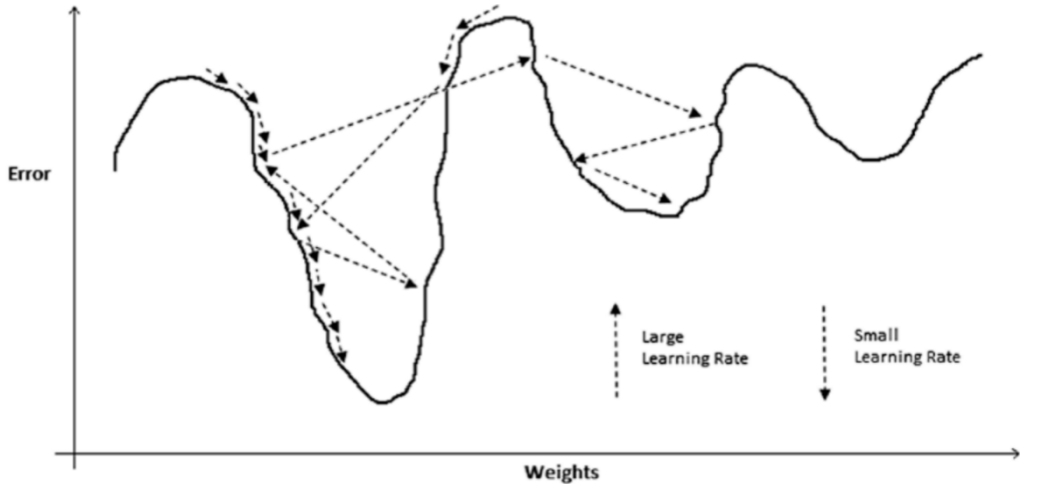

图 2-14　小型和大型学习率影响的示意图

在图 2-14 中，有两种情况：情景 1，小的学习速率；情景 2，大的学习速率。我们的目标是为了达到图 2-14 中的最小值，所以必须达到谷底（就像图 2-13 中球的情况那样）。学习率则与球滚下山时的跳跃幅度有关。

首先考虑情景 1（图 2-14 中的左侧部分），我们在其中进行小的跳跃，逐渐继续向下滚动，慢慢地，最终达到最小值，而球有可能卡在路上的一些小缝隙中，并且由于无法进行大跳跃而无法逃脱它。

在情景 2（图的右侧部分）中，与曲线的斜率相比，学习速率更大。这是一个次优的策略，实际上可能将我们从山谷中驱逐出去，在某些情况下，这可能是一个良好的开端，可以摆脱局部极小的范围，但如果我们跳过全局最小值，就会让人对其感到失望。

在图 2-14 中，我们实现了局部最小值，但这只是一个例子。这意味着权重值会停留在局部最小值上而错过全局最小值。梯度下降或随机梯度下降不保证能够收敛到神经网络的全局最小值（假设隐藏单元不是线性的），因为损失函数是非凸性的。理想情况是阶跃（步长）继续变化并且拥有更具适应性的特点，在起初时可以略高，然后在一段时间内逐渐减小，直到收敛为止。

2.9　反向传播

理解反向传播（Backpropagation）算法可能需要一些时间，读者也可以跳过本节内容，因为很多软件库都具有自动区分和执行整个训练过程的能力。但是，理解这个算法肯定会让你深入了解与深度学习相关的问题（学习问题、缓慢学习、梯度爆炸、梯度下降）。

让我们首先看一张典型的神经网络结构图，如图 2-15 所示。

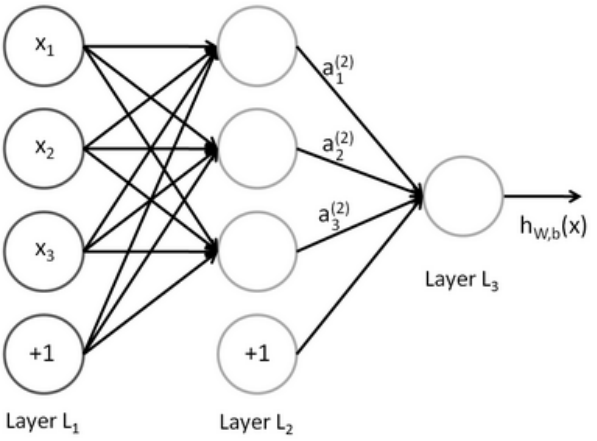

图 2-15　神经网络结构图

图 2-15 是一个包含了输入层 L1、隐藏层 L2 和输出层 L3 的简单神经网络，它的处理流程为根据输入层的 input 以及相应的权重值和配置（图中黑色带箭头的边），通过隐藏层的加工，最终将结果映射到输出层得到结果的输出。模型可以抽象表示为$y = f(x)$，x, y分别表示输入和输出向量。

根据神经网络的处理流程，我们如果要得到输出y，就必须知道图 2-15 中每条边的参数值，这也是神经网络中最重要的部分。在神经网络中是通过迭代的方法来计算这些参数的，具体来讲就是，首先初始化这些参数，通过神经网络的前向传导过程来计算得到输出y，但这些值与真实值存在着误差，我们假设累计误差函数为$err(x)$，然后利用梯度方法极小化$err(x)$来更新参数，直至误差值达到符合要求而停止计算。在更新参数这一过程中，我们就用到了著名的反向传播算法。

为了更好地去说明神经网络的传播算法，我们这里取图 2-15 中的一条路径来做说明，但这不失一般性，假设路径如图 2-16 所示。

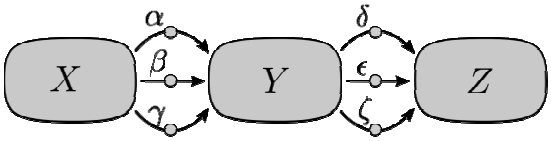

图 2-16　神经网络传播路径示例

图 2-16 中的边表示偏导数，如$\alpha = (\partial Y / \partial X)$。

想要知道输入 X 对输出 Z 的影响，我们可以用偏导数$(\partial Z / \partial X) = (\partial Z / \partial Y) \cdot (\partial Y / \partial X)$，即：

$$(\partial Z / \partial X) = \alpha\delta + \alpha\epsilon + \alpha\zeta + \beta\delta + \beta\epsilon + \beta\zeta + \gamma\delta + \gamma\epsilon + \gamma\zeta \tag{2.5}$$

我们如果直接使用链式法则进行求导就会遇到一个问题，当路径数量增加时，式（2.5）中的子项目数会呈指数增长，所以这时我们需要把上式右侧部分进行合并，合并后，我们只需要进行一次乘法就可以获得所需要的结果，这样大幅度提升了模型的运算效率，合并后的式子如下：

$$(\partial Z / \partial X) = (\alpha + \beta + \gamma)(\delta + \epsilon + \zeta) \tag{2.6}$$

接下来，我们解决式（2.6）的实现问题。根据计算方向的不同，可以分为正向微分与反向微

分。我们先看针对图 2-16 的正向微分算法，如图 2-17 所示。

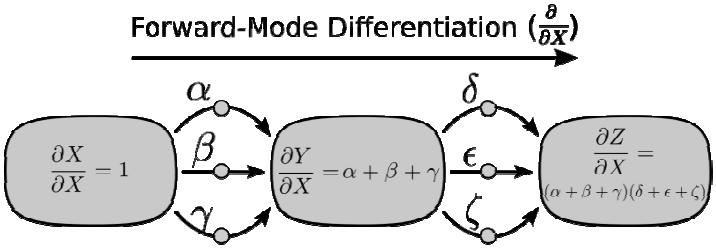

图 2-17　正向微分算法

可以看到，正向微分算法根据路径的传播方向，依次计算路径中的各结点对输入 X 的偏导数，结果中保留了输入对各结点的影响。

下面，我们看一下反向微分算法（见图 2-18）。

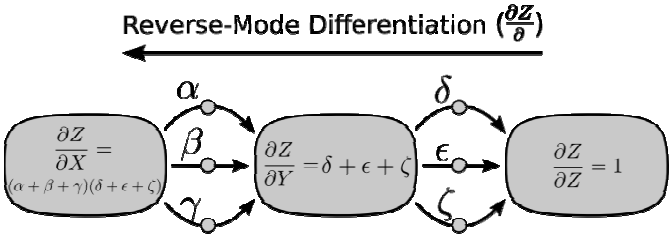

图 2-18　反向微分算法

可以看到，该算法从后向前进行计算，结果中保留了路径中各结点对输出的影响。

这里就有一个问题了，既然正向反向都可以实现式（2.6）的计算，那么我们应该选择哪个算法来实现呢？答案是反向微分算法，理由如下：

首先我们看一个计算式子$e = (a + b) * (b + 1)$的图模型（见图 2-19）。

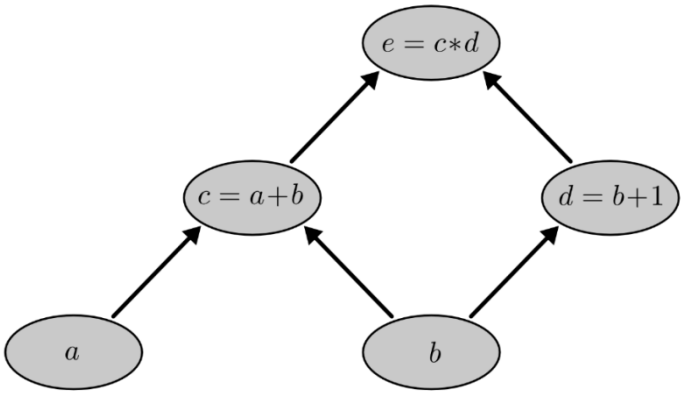

图 2-19　图模型计算示例

其中，c,d 表示中间结果，边的方向表示一个结点是另一个结点的输入。

假设输入变量 a=2、b=1，图 2-19 中各结点的偏导计算结果如图 2-20 所示。

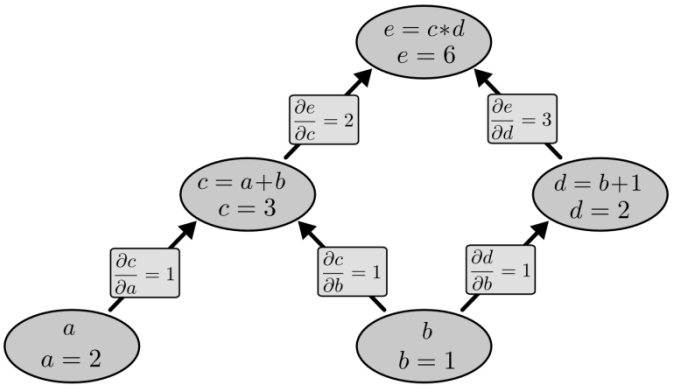

图 2-20　利用正向微分算法得到各结点的偏导计算结果

利用正向微分算法，我们得到关于变量 b 的偏导计算结果如图 2-21 所示。

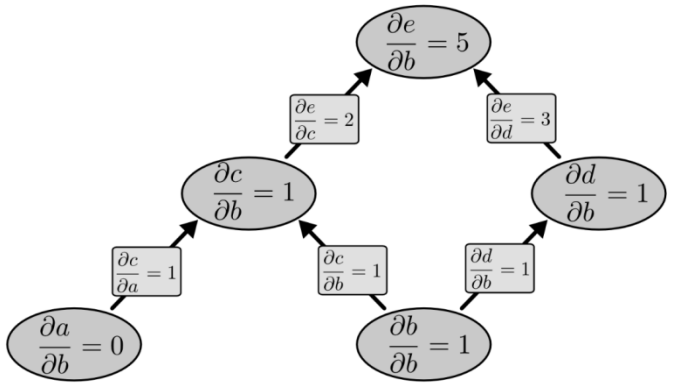

图 2-21　利用正向微分算法得到变量 b 的偏导计算结果

利用反向微分算法，我们得到的偏导计算结果如图 2-22 所示。

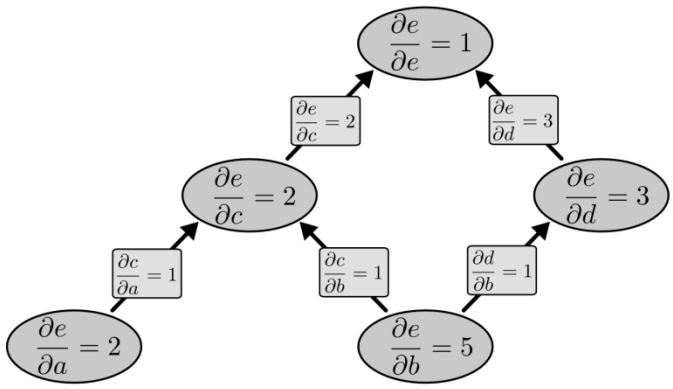

图 2-22　利用反向微分算法得到的偏导计算结果

由此可见，反向微分算法保留了所有变量（包括中间变量）对结果 *e* 的影响。若 *e* 为误差函数，则对图进行一次计算，可以得出所有结点对 *e* 的影响，也就是梯度值，下一步就可以利用这些梯度

值来更新边的权重值了；而正向微分算法得到的结果是只保留了一个输入变量对误差 e 的影响，显然，想要获得多个变量对 e 的影响，我们就需要进行多次计算。所以正向微分算法在效率上明显不如反向微分，这也是我们选择反向微分算法的原因。

2.10　总结

在本章，我们首先对深度学习的理念和产生的历史背景做了说明，认识到深度学习在人工智能当中起到非常重要的作用。

其次，我们为了对深度学习的内部原理进行解读，介绍了其中的关键部分——神经元模型。通过神经元模型的解读，我们对深度学习的基本结构有了认知。

接着，我们从单层神经网络、多层神经网络和 Encoder-Decoder 网络等方面对于深度学习的内部网络结构进行了更深层次的解析，对其内部逻辑和机制运行情况有了大致清晰的认知。因为现实中的问题一般都非常复杂，怎样选择一个高效的优化算法便成为我们着重关注的方法，所以我们就介绍了随机梯度下降算法。

最后，我们介绍了著名的反向传播算法（BP），从数学的角度做简要解读，使我们认识到反向传播算法的魅力所在，为我们进一步学习深度学习或者说神经网络模型奠定了良好的基础。

下一章，我们将介绍本书所使用的技术框架：TensorFlow。

第 3 章

TensorFlow

在本章中，首先我们会对 TensorFlow 的概念、主要特征、安装及其三个组成部分进行详细解读，梳理出这些核心概念之间的关系。其次，在了解了一些核心概念之后，我们会对 TensorFlow 的工作原理进行深度解析，将 TensorFlow 平台的“client—master—worker”架构底层中涉及到的内在逻辑和技术路径进行详细解读，以便我们对 TensorFlow 框架的运行机制有更多的认知。接着，我们会利用示例对 TensorFlow 客户端进行专门解读，读者会在示例中感受 TensorFlow 客户端内在的运行情况。然后，我们会对 TensorFlow 中的常见元素逐一详解，并做一些对比分析。最后，我们对 TensorFlow 的变量作用域机制做一些介绍，并实现一个三层神经网络对 MNIST 数字数据集进行分类的例子。

本章中涉及的完整代码，请查看文件夹 ch3 下的代码文件：3_tensorflow_introduction.ipynb。

3.1 TensorFlow 概念解读

由于深度学习研究的最终目的是为了解决我们现实世界中遇到的困难，因此在学术界不断探索新理念、新模型的同时，业界优秀的从业人员也在加速发布各类可实现的技术框架，这样深度学习框架就在学术界和工业界的相互推动下得以迅速发展起来。这里给出谷歌公司经过长期的研究和尝试，在实践的基础上推出的目前最优秀的深度学习框架之一——TensorFlow。TensorFlow 最初是由谷歌公司的大脑小组（隶属于谷歌机器智能研究机构）的研究员和工程师们开发出来的，是一个采用数据流图（Data Flow Graph）的开源分布式数值计算框架，主要用于减轻实现神经网络细节层面的工作量（例如，计算神经网络权重值的导数）。TensorFlow 通过使用统一计算设备架构（Compute Unified Device Architecture，CUDA）对数值进行有效计算，该架构是由 NVIDIA 公司引入的并行计算平台。

TensorFlow 主要是由计算图、张量以及模型会话三个部分组成的。TensorFlow 中的计算可以表示为一个计算图（Computation Graph），或者称作有向图（Directed Graph），图中的每一个数学运算或操作（Operation）都将被视为一个节点（Node），而节点与节点之间的连接被称为边（Edge），

图中的这些边（Edge）则表示在节点间相互关联的流动（Flow）的多维数据数组，即张量（Tensor）。这个计算图表述了数据的计算流程，它负责维护和更新状态，我们可以对计算图中的分支进行条件控制或循环操作。简单说，我们可以用张量表示数据，用计算图搭建神经网络，用会话执行计算图，在优化线上的权重值（参数）后得到我们的模型。

这里，我们还可以使用多种语言对计算图进行设计。TensorFlow 这样灵活的架构让我们可以在多种平台上展开计算，例如台式计算机中的一个或多个 CPU（或 GPU）、服务器、移动设备等。作为优秀的技术架构，TensorFlow 在机器学习方面解决了以下几个核心问题：

（1）使用 TensorFlow 工具可以利用很简洁的语言来实现各类复杂算法模型，这样一来，我们就可以从日常中极为消耗精力的编码、调试工作中解放出来。

（2）由于 TensorFlow 内核执行系统使用的是 C++，因此在执行效率方面非常高效。

（3）TensorFlow 极佳的分层架构使得模型能够很方便地运行在异构设备之上。

（4）TensorFlow 包含了 TensorBoard 等极好的配套工具，且第三方社区也在不断贡献很多实用的辅助工具，例如 TFLearn 等，这些工具使得代码变得更加简洁、数据处理工作更便捷。

如果读者感兴趣，可以参考下面的网址以查阅关于 TensorFlow 方面的内容。

- TensorFlow 官网：tensorflow.google.cn。
- TensorFlow 的应用程序编程接口（API）：tensorflow.google.cn/api_docs/python。
- GitHub 网址：github.com/tensorflow。
- 模型仓库：github.com/tensorflow/models。

3.2　TensorFlow 主要特征

3.2.1　自动求微分

基于梯度的机器学习算法会受益于 TensorFlow 自动求微分的能力。在使用 TensorFlow 时，我们只需要定义预测模型的结构，将这个结构和目标函数（Objective Function）结合在一起，并添加数据，TensorFlow 将自动为我们计算相关的微分导数。计算某个变量相对于其他变量的导数仅仅是通过扩展你的计算图来完成的，所以我们可以一直很清晰地看到系统中究竟在发生什么。

3.2.2　多语言支持

TensorFlow 有一个合理的 C++使用界面，也有一个易用的 Python 使用界面来构建和执行我们的计算图（Computation Graph）。我们可以直接编写 Python/C++程序，也可以用交互式的 iPython 界面来调用 TensorFlow 以尝试一些想法，它可以帮我们将笔记、代码、可视化等有条理地归置好。当然这仅仅是一个起点，它同时也在促使我们创造自己最喜欢的语言界面，比如 Go、Java、Lua、JavaScript、R 等。

3.2.3 高度的灵活性

TensorFlow 不是一个严格意义上的“神经网络”库。如果我们能够将计算表示为一个数据流图，那么就可以使用 TensorFlow 来构建图和描写驱动计算的内部循环。TensorFlow 提供了有用的工具来帮助你组装“子图”（常用于神经网络），当然用户也可以自己在 TensorFlow 基础上编写自己的“上层库”。定义顺手好用的新复合操作和编写一个 Python 函数一样容易，而且也不用担心性能损耗。当然万一我们发现找不到想要的底层数据操作，我们也可以自己编写一点 C++代码来丰富底层的操作。

3.2.4 真正的可移植性

TensorFlow 在 CPU 和 GPU 上运行，比如说可以运行在台式机、服务器、移动设备等上面。在没有特殊硬件的前提下，TensorFlow 能够帮我们在自己的笔记本上运行一下机器学习、深度学习模型。TensorFlow 也可以帮助我们将自己的训练模型在多个 CPU 上进行规模化的运算，同时不必修改代码。TensorFlow 还可以帮助我们将自己训练好的模型作为产品的一部分部署到手机 App 里。TensorFlow 更可以帮助我们将自己的模型作为云端服务运行在自己的服务器上，或者运行在 Docker 容器里。

3.2.5 将科研和产品联系在一起

过去，要将科研中的机器学习想法用到产品中，需要大量的代码重写工作。现在，谷歌公司的科学家用 TensorFlow 尝试新的算法，产品团队则用 TensorFlow 来训练和使用计算模型，并直接提供给在线用户。使用 TensorFlow 可以让应用型研究者将想法迅速地运用到产品中，也可以让学术性研究者更直接地彼此分享代码，从而提高科研产出率。

3.2.6 性能最优化

例如，现在你有一个带有 32 个 CPU 内核、4 个 GPU 显卡的工作站，就可以利用 TensorFlow 将工作站的计算潜能全部发挥出来。TensorFlow 给予了线程、队列、异步操作等最佳的支持，能够让我们将自己手边硬件的计算潜能全部发挥出来。我们可以自由地将 TensorFlow 图中的计算元素分配到不同设备上，让 TensorFlow 帮我们管理好这些不同的副本。

3.3 TensorFlow 安装

我们可以在多个平台上安装和使用 TensorFlow，例如 Linux、Mac OS 和 Windows。当然，我们也可以使用 TensorFlow 最新的 GitHub 源来构建和安装 TensorFlow。此外，如果我们的电脑运行

的是 Windows 系统，那么比较常用的是通过 Anaconda 来安装 TensorFlow。

由于 Python 3 附带了 pip3 包管理器，它是用于安装 TensorFlow 的程序，因此如果我们使用的是 Python 的这个版本，就无须安装 pip。为简单起见，在本节中将展示如何使用本机 pip 安装 TensorFlow。下面给出在 Windows 系统下安装 TensorFlow 的两个命令。启动一个“终端”程序，然后在该“终端”中执行相应的 pip3 install 命令。

（1）要安装 TensorFlow 的 CPU 版本，输入以下命令：

```
C:\> pip3 install --upgrade tensorflow
```

（2）要安装 TensorFlow 的 GPU 版本，输入以下命令：

```
C:\> pip3 install --upgrade tensorflow-gpu
```

在 Linux 方面，TensorFlow Python API 支持 Python 2.7 和 Python 3.3+，因此我们需要安装 Python 才能启动 TensorFlow 安装。我们必须安装 Cuda Toolkit 7.5 和 cuDNN v5.1 +才能获得 GPU 支持。笔者使用的是 Ubuntu 系统，是采用 pip3 的命令来安装 TensorFlow 的，更多信息也可以参考 https://tensorflow.google.cn/install/source 上的说明。

在 Linux 上安装 TensorFlow

这里将给出如何在 Ubuntu 14.04 或更高版本上安装 TensorFlow。此处提供的说明可能也适用于 Linux 其他版本，只需进行少量的调整即可。

但是，在继续执行正式步骤之前，我们需要确定在平台上安装哪个 TensorFlow，我们可以在 GPU 和 CPU 上运行数据密集型张量应用程序。因此，应该选择以下 TensorFlow 两种版本中的一个，在我们的平台上进行安装：

- 仅支持 CPU 的 TensorFlow: 如果计算机上没有安装 NVIDIA 等 GPU，就必须使用此版本安装并开始计算。这个很简单，可以在 5 到 10 分钟内完成。
- 支持 GPU 的 TensorFlow: 深度学习应用程序通常需要非常高的计算资源，TensorFlow 也不例外，我们通常可以在 GPU 上而不是在 CPU 上加快数据计算和分析速度。如果你的计算机上有 NVIDIAGPU 硬件，你应该安装并使用此版本。

根据我们的经验，即使计算机上集成了 NVIDIA GPU 硬件，也有必要安装并首先尝试仅使用 CPU 的版本，如果你没有体验到良好的性能，那么再切换到 GPU 来进行支持。

支持 GPU 的 TensorFlow 版本有几个要求，例如 64 位 Linux、Python 2.7（或 Python 3 的 3.3+），NVIDIACUDA7.5 或更高版本（Pascal GPU 需要 CUDA 8.0）和 NVIDIA cuDNN v4.0（最小）或 v5.1（推荐）。更具体地说，TensorFlow 的当前开发仅支持使用 NVIDIA 工具包和软件的 GPU 计算。因此，必须在 Linux 计算机上安装以下软件才能使得预测分析应用程序获得 GPU 的支持：

- Python
- NVIDIA Driver

- CUDA with compute capability >= 3.0
- CuDNN
- TensorFlow

1. 安装 Python 和 NVIDIA 驱动程序

在不同的平台上安装 Python 的操作相对比较简单，这里不再赘述。另外，假设你的计算机中已经安装了 NVIDIA GPU。

要检查一下你的计算机中的 GPU 是否已正确安装和正常工作，请在终端上执行以下命令：

```
$ lspci -nnk | grep -i nvidia
```

由于预测分析很大程度上依赖于机器学习和深度学习算法，因此请确保你的计算机上安装了一些基本软件包，例如 GCC 和一些用于科学计算的 Python 软件包。

只需在终端上执行如下命令：

```
$ sudo apt-get update
$ sudo apt-get install libglu1-mesa libxi-dev libxmu-dev -y
$ sudo apt-get – yes install build-essential
$ sudo apt-get install python-pip python-dev -y
$ sudo apt-get install python-numpy python-scipy -y
```

现在通过 wget 下载 NVIDIA 驱动程序（不要忘记为你的计算机选择正确的版本），并以 silent 模式运行脚本：

```
$ wget
http://us.download.nvidia.com/XFree86/Linux-x86_64/367.44/NVIDIA-Linux-x86_64-
367.44.run
$ sudo chmod +x NVIDIA-Linux-x86_64-367.35.run
$ ./NVIDIA-Linux-x86_64-367.35.run --silent
```

一些 GPU 显卡，如 NVidia GTX 1080，附带内置驱动程序。因此，如果你的计算机具有不同于 GTX 1080 的 GPU，就必须下载该 GPU 的驱动程序。

要确保驱动程序是否已正确安装，请在终端上执行以下命令：

```
$ nvidia-smi
```

命令的结果应如图 3-1 所示。

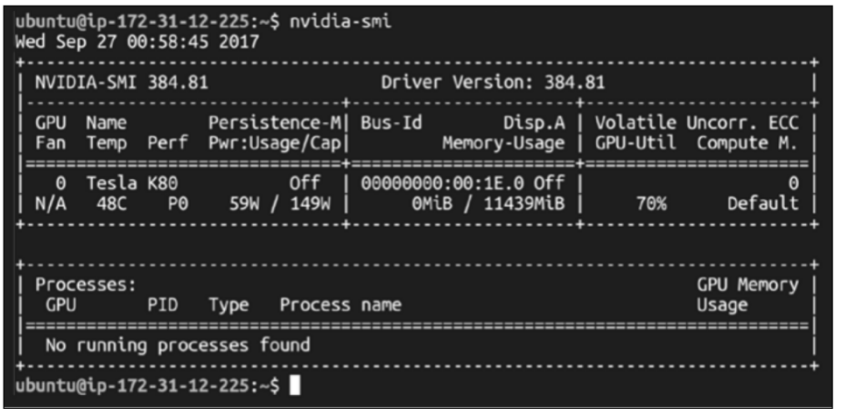

图 3-1　nvidia-smi 命令的输出结果

2. 安装 NVIDIA CUDA

为了将 TensorFlow 与 NVIDIA GPU 配合使用，需要安装 CUDA Toolkit 8.0 及其相关的 NVIDIA 驱动程序。CUDA 工具包包括：

- GPU 加速库，如用于快速傅里叶变换的 cuFFT（FFT）。
- 基本线性代数子程序（BLAS）的 cuBLAS。
- 稀疏矩阵例程的 cuSPARSE。
- 用于密集和稀疏的直接求解器的 cuSOLVER。
- 用于随机数生成的 cuRAND，用于图像的 NPP 和视频处理原语。
- NVIDIA 图形分析库的 nvGRICH。
- 推力模板并行算法和数据结构专用的 CUDA 数学库。

对于 Linux，下载和安装所需的包：

https://developer.nvidia.com/cuda-downloads

使用的 wget 命令如下：

```
$ wget https://developer.nvidia.com/compute/cuda/8.0/Prod2/local_
installers/cuda_8.0.61_375.26_linux-run
$ sudo chmod +x cuda_8.0.61_375.26_linux.run
$ ./ cuda_8.0.61_375.26_linux.run --driver --silent
$ ./ cuda_8.0.61_375.26_linux.run --toolkit --silent
$ ./ cuda_8.0.61_375.26_linux.run --samples -silent
```

另外，确保你已经将 CUDA 安装路径添加到 LD_LIBRARY_PATH 环境变量中：

```
$ echo 'export LD_LIBRARY_PATH="$LD_LIBRARY_PATH:/usr/local/cuda/lib64:/
usr/local/cuda/extras/CUPTI/lib64"' >> ~/.bashrc
$ echo 'export CUDA_HOME=/usr/local/cuda' >> ~/.bashrc
$ source ~/.bashrc
```

3. 安装 NVIDIA cuDNN v5.1+

安装CUDA Toolkit后，从Linux上下载cuDNN v5.1库，解压缩文件并将其复制到CUDA Toolkit目录（假设在/ usr / local / cuda /中）：

```
$ cd /usr/local
$sudo mkdir cuda
$ cd ~/Downloads/
$ wget http://developer2.download.nvidia.com/compute/machine-learning/
cudnn/secure/v6/prod/8.0_20170427/cudnn-8.0-linux-x64-v6.0.tgz
$ sudo tar -xvzf cudnn-8.0-linux-x64-v6.0.tgz
$ cp cuda/lib64/* /usr/local/cuda/lib64/
$ cp cuda/include/cudnn.h /usr/local/cuda/include/
```

注意，要安装 cuDNN v5.1 库，我们必须在 https://developer.nvidia.com/accelerated-computing-developer 上注册加速计算开发程序。现在，当安装了 cuDNN v5.1 库之后，请确保创建了 CUDA_HOME 环境变量。

4. 安装 libcupti-dev 库

最后，需要在你的计算机上安装 libcupti-dev 库，这是提供高级分析支持的 NVIDIA CUDA。要安装此库，请执行以下命令：

```
$ sudo apt-get install libcupti-dev
```

5. 安装 TensorFlow

有关如何为 CPU 安装最新版本的 TensorFlow，以及如何为具有 NVIDIA 、cuDNN 和 CUDA 计算能力的 GPU 提供支持，请参阅下面的部分。可以在计算机上以多种方式安装 TensorFlow，比如使用 virtualenv、pip、Docker 和 Anaconda。这里使用 pip 和 virtualenv 方式。（有兴趣的读者可以尝试使用 Docker 和 Anaconda 安装，详见 https://tensorflow.google.cn/install/。）

（1）使用本地 pip 安装 TensorFlow

对于 Python 2.7，仅支持 CPU 版本的，具体如下：

```
$ pip install tensorflow
# For Python 3.x and of course with only CPU support:
$ pip3 install tensorflow
# For Python 2.7 and of course with GPU support:
$ pip install tensorflow-gpu
# For Python 3.x and of course with GPU support:
$ pip3 install tensorflow-gpu
```

（2）使用 virtualenv 安装 TensorFlow

如果你已经在系统上安装了 Python 2+（或 3+）和 pip（或 pip3），请按照以下步骤安装 TensorFlow。

①创建 virtualenv 环境：

```
$ virtualenv --system-site-packages targetDirectory
```

targetDirectory 表示 virtualenv 树的根目录。默认情况下，它是~/ tensorflow（也可以选择任何其他目录）。

②按如下方式激活 virtualenv 环境：

```
$ source ~/tensorflow/bin/activate       # bash, sh, ksh, or zsh
   $ source ~/tensorflow/bin/activate.csh       # csh or tcsh
```

如果命令在步骤 2 中执行成功，那么在“终端”程序中上应该可以看到以下内容：

```
(tensorflow)$
```

③安装 TensorFlow。

按照以下命令之一在激活的 virtualenv 环境中安装 TensorFlow。

对于仅支持 CPU 的 Python 2.7，请使用以下命令：

```
(tensorflow)$ pip install --upgrade tensorflow
```

对于仅支持 CPU 的 Python 3.x，请使用以下命令：

```
(tensorflow)$ pip3 install --upgrade tensorflow
```

对于支持 GPU 的 Python 2.7，请使用以下命令：

```
(tensorflow)$ pip install --upgrade tensorflow-gpu
```

对于仅支持 GPU 的 Python 3.x，请使用以下命令：

```
(tensorflow)$ pip3 install --upgrade tensorflow-gpu
```

如果上述命令成功，就跳过步骤 5；如果上述命令失败，请执行步骤 5。此外，如果步骤 3 以某种方式失败，请尝试通过执行以下命令以在激活的 virtualenv 环境中安装 TensorFlow：

```
#对于 python 2.7（选择具有 CPU 或 GPU 支持的适当 URL）：
(tensorflow)$ pip install --upgrade TF_PYTHON_URL
#对于 python 3.x（选择具有 CPU 或 GPU 支持的适当 URL）：
(tensorflow)$ pip3 install --upgrade TF_PYTHON_URL
```

④验证安装。

要在步骤 3 中验证安装，必须激活虚拟环境。如果 virtualenv 环境当前未处于激活状态，请执行以下命令之一：

```
$ source ~/tensorflow/bin/activate    # bash, sh, ksh, or zsh
$ source ~/tensorflow/bin/activate.csh   # csh or tcsh
```

⑤卸载 TensorFlow。

要卸载 TensorFlow，只需删除创建的目录树即可。例如：

```
$ rm -r targetDirectory
```

6. 测试安装的 TensorFlow

启动一个 Python 终端（只需在终端上输入 python 或 python3），执行一段常见的“Hello, TensorFlow!”程序，代码行如下：

```
>>> import tensorflow as tf
>>> hello = tf.constant("Hello, TensorFlow!")
>>> sess=tf.Session()
>>> print sess.run(hello)
```

如果 TensorFlow 安装成功，我们将看到输出结果为“Hello, TensorFlow!”。

3.4 TensorFlow 计算图

在执行 TensorFlow 程序时，我们需要构建一个计算图，然后按照计算图启动一个会话，在会话中完成变量赋值、计算等操作并得到最终结果。换句话说，TensorFlow 的计算图可以划分为两部分：

- 构造部分，包含计算流图，用于构建模型。
- 执行部分，通过会话（Session）执行图中的计算，用于提供数据并获得结果。

对于计算图的构造部分而言，我们又可以分为两部分：

- 创建源节点。
- 源节点的输出传递给其他节点做运算操作。

这里，更吸引我们的是，TensorFlow 会在 C ++引擎上执行每一项运算操作，这意味着在 Python 中也不会执行任何乘法或加法，Python 只是一个包装器。从根本上讲，TensorFlow C ++引擎包含以下两方面：

- 卷积、最大池化、Sigmoid 等操作的高效实现。
- 转发模式操作的导数。

计算图基本上类似于数据流图。图 3-2 显示了一个简单计算的计算图，用于计算简单的等式，如 $z = d \times c =（a + b）\times c$。

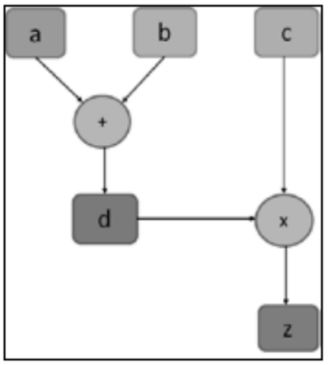

图 3-2　一个简单的执行图

在图 3-2 中，圆圈表示节点处的操作，矩形表示整个数据计算图。在 TensorFlow 的实施中，TensorFlow 计算图包含以下内容：

- 一组 tf.Operation 对象：用于表示要执行的计算单元。
- tf.Tensor 对象：用于表示控制操作之间数据流的数据单元。

使用 TensorFlow 还可以执行延迟操作。一旦在计算图的构建阶段编写了高度组合的表达式，我们仍然可以在会话的运行阶段去对它们求值。从技术上讲，TensorFlow 会给出措施并以有效的方式按时执行。例如，使用 GPU 并行执行代码的独立部分，如图 3-3 所示。

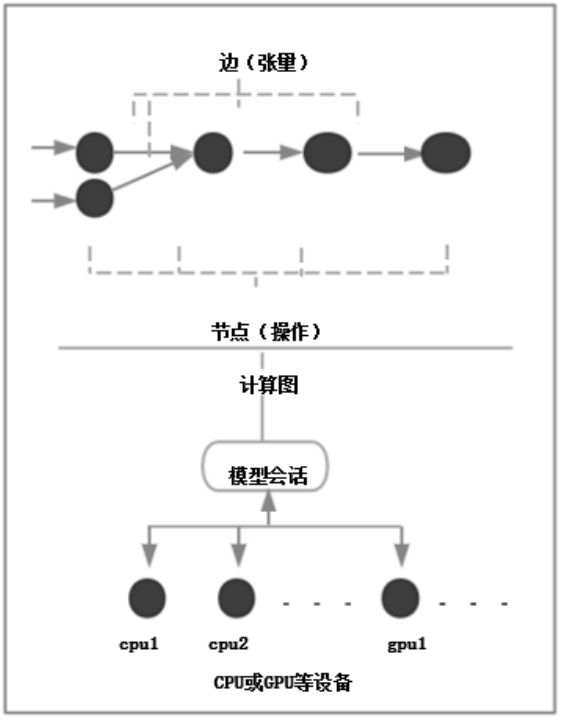

图 3-3　TensorFlow 图中的边和节点将在 CPU 或 GPU 等设备上的会话下执行

3.5 TensorFlow 张量和模型会话

3.5.1 张量

在 TensorFlow 中，张量（Tensor）是对运算结果的引用，运算结果多数情况下是以数组的形式存储的，与 numpy 中的数组不同的是，张量还具有三个重要属性：名字、维度、类型。张量的名字是张量的唯一标识符，我们通过名字可以发现张量是如何被计算出来的。例如“multiply:0”代表的是计算节点“multiply”的第 1 个输出结果，“add:2”代表的是计算节点“add”的第 3 个输出结果。

TensorFlow 有两种类型的边，或者说张量（Tensor）的类型有两种：

- 正常：它们承载节点之间的数据结构。来自一个节点的一个运算或操作的输出变为另一个运算或操作的输入。连接两个节点的线携带这些数值。
- 特殊：此边不携带数值，但是只表示两个节点（比如 X 和 Y）之间的控制依赖关系。这意味着节点 Y 只有在 X 中的运算或操作已经执行时才会被执行。

我们也可以这样理解，TensorFlow 计算图中的每个节点的输入输出都是张量（Tensor），而连接节点的边（有向线段）就是 Flow，表示从一个张量（Tensor）状态到另一个张量（Tensor）状态。我们在编写程序时，都是逐步计算的，每计算完一步便能够获得一个执行结果。对于 Tensor 数据类型而言，虽然我们可以事先定义或者根据计算图的结构推断得到，但是有一类特殊的边中并没有数据流动，这种边便是依赖控制（Control Dependencies），其作用便是让它的起始节点执行完之后再去执行目标节点，这样一来我们就可以使用边进行灵活的条件控制了。比如，我们在 TensorFlow 实施中定义了依赖控制关系，便可以在其他独立的操作之间强制执行排序，以作为限制使用内存最高峰值的一种方式。

在 TensorFlow 中，一个张量基本上是一个 *n* 维数组，下面让我们通过表 3-1 中的内容来解读一下张量和维数的关系。

表 3-1 张量和维数的关系

维数	阶	名称	示例
0-D	0	标量(Scalar)	s=1 2 3
1-D	1	向量(Vector)	v=[1,2,3]
2-D	2	矩阵	m=[[1,2,3],[4,5,6]]
3-D	3	张量	t=[[...
...	...	...	...
n-D	*n*	张量	T=[[...

张量可以表示 0 阶到 *n* 阶的数组（列表）。这里，阶是张量的维数。第 0 阶张量是一个数，也

就是标量；第 1 阶张量是一个一维数组，也就是向量；第 2 阶张量是一个二维数组，也就是矩阵；第 *n* 阶张量是一个 *n* 维数组。

3.5.2　会话

如前所述，TensorFlow 中的会话（Session）用来执行事先定义好的运算，负责完成多个计算设备或集群分布式节点的布置和数据传输节点的添加，并且还负责将子图分配给相应的执行器单元进行运行。会话拥有并管理 TensorFlow 程序运行时的所有资源。当计算完成之后需要关闭会话来帮助系统回收资源，否则就可能出现计算资源被占用的问题。

3.6　TensorFlow 工作原理

TensorFlow 是一个具有“client—master—worker”架构的分布式系统。在 TensorFlow 中，参与分布式系统的所有节点或者设备被统称为一个 cluster（集群），一个 cluster 中包含多个 server（服务器），每个 server 去执行一项 task（任务），而 server 和 task 又是一一对应的。这样看来，我们可以把 cluster 看成是 server 的集合，也可以看成是 task 的集合。TensorFlow 为每个 task 又增加了一个抽象层，将一系列相似的 task 集合称为一个 job（工作）。例如，我们在 PS 架构中，习惯称 parameter server（参数服务器）的 task 集合为 ps，而把执行梯度计算的 task 集合称为 worker。因此，cluster 又可以被看成是 job 的集合，实际上这只是逻辑上的意义，我们还需要具体看一下这个 server 真正做的是什么。在 TensorFlow 中，job 用 name（字符串）标识，task 用 index（整数索引或整数下标）标识，cluster 中的每个 task 都可以用 job 的 name 加上 task 的 index 来作唯一标识。在分布式系统中，一般情况下各个 task 在不同的节点或者设备上执行。

通常情况下，客户端（client）通过会话 tf.Session 接口和 master 展开通信，且将触发执行的请求提交给 master，而 master 会把所要执行的任务分派给单个或多个 worker 进程，相关结果通过 master 返回给客户端。这里，worker 负责计算的执行，任何一个 worker 进程都会管理和使用计算机上的计算硬件设备资源，比如一块或者多块 CPU 和 GPU，进而处理计算子图（Subgraph）的运算或操作过程。

其实，TensorFlow 的架构横向扩展是很灵活的。在单机模式的多数情况下，client、master 和 worker 均在同一个进程中启动，而 worker 进程会管理本机上所有的计算设备资源。在分布式的架构中，我们可以通过远程调用的方式将 client、master 和 worker 连接在一起，且任何一个 worker 进程各自监控关联执行机上的计算设备。关于 TensorFlow 的单机和分布式架构如图 3-4 所示。

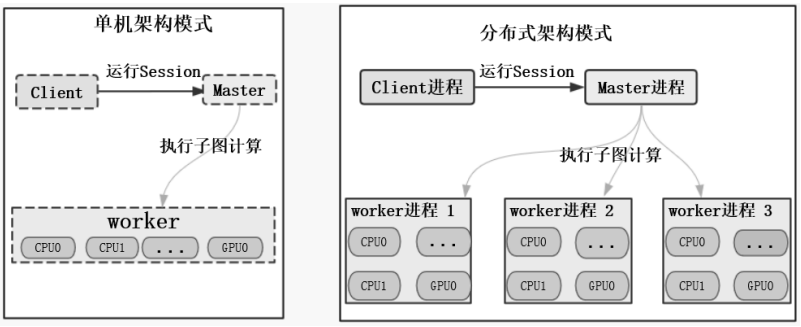

图 3-4　TensorFlow 的单机和分布式架构示意图

在创建计算图之后，TensorFlow 需要以分布式方式由多个 CPU（以及 GPU，如果可用）执行激活的会话。通常，实际上不需要明确指定是使用 CPU 还是 GPU，因为 TensorFlow 可以选择并使用其中一个。默认情况下，将选择 GPU 进行尽可能多的运算；否则，将使用 CPU。因此，从广义上看，以下是 TensorFlow 的主要组件：

- 变量（Variable）：用于包含 TensorFlow 会话之间权重值和偏差的值。
- 张量（Tensor）：在节点之间传递的一组值。
- 占位符（Placeholder）：用于在程序和 TensorFlow 图之间发送数据。
- 会话（Session）：启动会话时，TensorFlow 会自动计算图中所有计算的梯度，并在链式规则中使用它们。实际上，在执行图形时会调用会话。

从技术上说，我们要编写的程序可以被视为客户端，而客户端用于以符号方式在 C/C ++或 Python 中创建计算图，接着，我们的代码可以要求 TensorFlow 执行此该图，请参见图 3-5 中的详细信息。

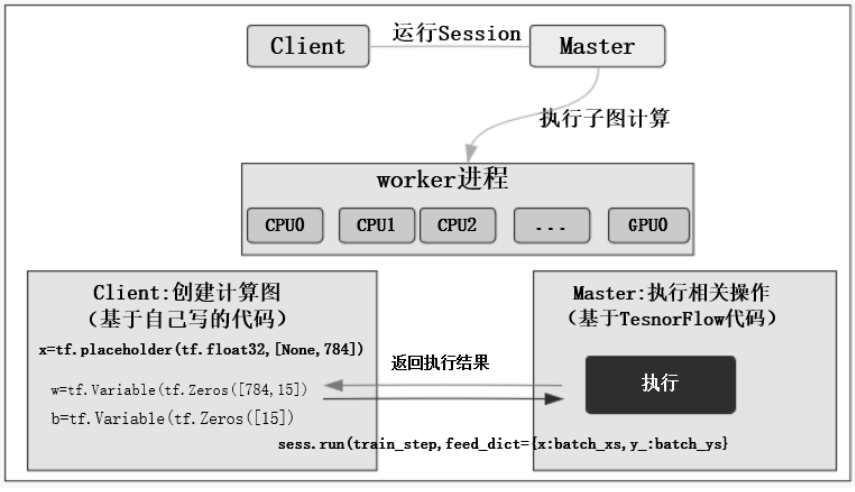

图 3-5　在 TensorFlow 分布式架构下执行代码示意图

计算图有助于在具有 CPU 或 GPU 的多个计算节点上分配工作负载。这样，神经网络等同于复

合函数，其中每个层（输入层、隐藏层或输出层）可以表示为函数。现在想要了解在张量上执行的运算或操作，有必要探讨 TensorFlow 编程模块方面的解决方案。

3.7　通过一个示例来认识 TensorFlow

现在让我们结合示例对 TensorFlow 框架中的一些基本组件进行更多的解读。我们编写一个示例来执行以下计算，这在神经网络中非常常见：

$$h = \text{sigmoid}(W * x + b)$$

这里 W 和 x 是矩阵，b 是向量，*表示点积。 sigmoid 是一个非线性变换，由下式给出：

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

我们将逐步讨论如何通过 TensorFlow 进行上述计算。

首先，我们需要导入 TensorFlow 和 NumPy。在 Python 中运行任何类型的 TensorFlow 或 NumPy 相关操作之前，均必须导入它们：

```
import tensorflow as tf
import numpy as np
```

接下来，我们定义一个图对象，稍后将使用运算或操作和变量填充它们：

```
graph = tf.Graph() # 创建一个图（graph）对象
session = tf.InteractiveSession(graph=graph) # 创建一个会话（session）对象
```

graph 对象包含一个计算图，该计算图连接我们在程序中定义的各种输入和输出，以获得最终所需的输出（即它定义 W、x 和 b 如何被连接以便我们根据图生成 h）。此外，我们将定义一个会话对象，该对象以定义的图作为输入，执行该图。我们后面会详细讨论这些元素。

要创建新的图对象，可以使用以下操作：

```
graph = tf.Graph()
```

也可以使用以下操作来获取 TensorFlow 默认计算图：

```
graph = tf.get_default_graph()
```

现在我们将定义一些张量，即 x、W、b 和 h。在 TensorFlow 中，定义张量有几种不同的方法，在这里列出三种：

（1）首先，x 是一个占位符。顾名思义，占位符没有被初始化。相反，我们将在图执行时提供一些数值。

（2）其次，这里有变量 W 和 b。由于变量是可变的，这意味着它们的值可以随时间而变化。

（3）最后，我们有 h，这是一个通过对 x、W 和 b 执行一些操作而产生的不可变的张量：

```
    x = tf.placeholder(shape=[1,10],dtype=tf.float32,name='x')
    W = tf.Variable(tf.random_uniform(shape=[10,5], minval=-0.1,maxval=0.1,
dtype=tf.float32),name='W')
    b = tf.Variable(tf.zeros(shape=[5],dtype=tf.float32),name='b')
    h = tf.nn.sigmoid(tf.matmul(x,W) + b)
```

注意，对于 W 和 b，我们提供了一些重要的参数：

```
    tf.random_uniform(shape=[10,5], minval=-0.1, maxval=0.1,dtype=tf.float32)
    tf.zeros(shape=[5],dtype=tf.float32)
```

这些被称为变量初始化器，是最初分配给 W 和 b 变量的张量。变量不能在没有初始值的情况下作为占位符流动，并且需要始终为它们分配一些数值。这里，tf.random_uniform 意味着我们在 minval(-0.1)和 maxval(0.1)之间均匀地采样以将值赋给张量，并且 tf.zeros 用零初始化张量。在定义张量时，定义张量的 shape 也非常重要。shape 属性定义张量每个维度的大小。例如，如果形状（shape）是[10,5]，这意味着它将是一个二维结构，在 0 轴上有 10 个元素，在 1 轴上有 5 个元素。

接下来，我们将进行初始化操作，初始化图中的变量 W 和 b：

```
    tf.global_variables_initializer().run()
```

现在，将执行计算图以获得我们需要的最终输出 h。这是通过运行 session.run(...)来完成的，我们将以占位符的值作为 session.run()命令的参数：

```
    h_eval = session.run(h,feed_dict={x: np.random.rand(1,10)})
```

最后，关闭会话，释放 session 对象持有的所有资源。

```
    session.close()
```

以下是此 TensorFlow 示例的完整代码。本章中的所有代码示例都将在 ch3 文件夹中的 3_tensorflow_introduction.ipynb 文件中提供：

```
    import tensorflow as tf
    import numpy as np
    # 定义 graph 和 session
    graph = tf.Graph() # Creates a graph
    session = tf.InteractiveSession(graph=graph) # Creates a session

    # 构建 graph
    # 占位符是一种符号输入
    x = tf.placeholder(shape=[1,10],dtype=tf.float32,name='x')
    W = tf.Variable(tf.random_uniform(shape=[10,5], minval=-0.1,maxval=0.1,
dtype=tf.float32),name='W') # 相关变量

    # 相关变量
    b = tf.Variable(tf.zeros(shape=[5],dtype=tf.float32),name='b')
```

```
h = tf.nn.sigmoid(tf.matmul(x,W) + b) # Operation to be performed
# 在图中执行操作和评估节点
tf.global_variables_initializer().run() # 初始化变量
#通过为输入 x 提供数值来运行该操作
h_eval = session.run(h,feed_dict={x: np.random.rand(1,10)})
# 关闭会话以释放会话中的所有已保存资源
session.close()
```

3.8　TensorFlow 客户端

前面的示例程序可以称为 TensorFlow 客户端，本节结合示例和上面的内容，对 TensorFlow 客户端做详细的解读。在使用 TensorFlow 编写的任何客户端程序中，主要将有两种主要类型的对象：运算（Operation）和张量（Tensor）。在前面的例子中，tf.nn.sigmoid 是一个运算，h 是张量。

然后，这里还有一个 graph 对象，它是存储程序数据流的计算图。当我们在代码中添加定义的 x、W、b 和 h 的后续代码行时，TensorFlow 会自动将这些张量（Tensor）和每个运算（Operation）（例如，tf.matmul()）作为节点添加到图中。该图将存储重要信息，例如张量的依赖性以及要执行的运算。在我们的示例中，如果要计算 h，那么张量 x、W 和 b 是必需的。因此，如果在运行时没有正确初始化其中一个参数，TensorFlow 将在这里指出需要修正的具体的初始化错误。

接着，由前面内容，我们知道会话（Session）将扮演执行计算图的角色，方法是将图划分为子图、更精细的图，然后将这些图分配给将执行指定任务的 worker，这是通过 session.run(...)函数来完成的。下面，我们来看看 TensorFlow 架构中执行客户端时的情况。

大家知道，TensorFlow 擅长创建一个包含所有依赖关系和运算的精确计算图，且它能够准确地知道数据流以何种方式、何时、在哪里流动。当然，还有一个元素使得 TensorFlow 变得更优秀，那就是能够有效执行计算图的会话（Session），这也是会话进入 TensorFlow 架构的入口之处。现在让我们来看看会话的内部情况，以了解图形是如何被执行的。

首先，由前面的介绍，我们知道 TensorFlow 客户端中的相关元素有计算图、张量和会话。创建会话时，它会将计算图作为 tf.GraphDef 协议缓冲区发送到分布式架构的主机。 tf.GraphDef 是图的标准化表示。分布式主服务器查看图中的所有计算，并将计算划分到不同的设备（例如，不同的 GPU 和 CPU）。我们的 sigmoid 示例中的计算图如图 3-6 所示。

接下来，计算图将被分解为子图，并进一步细分为更细的部分，这在具有许多隐藏层的现实世界解决方案中意义更大。此外，为了并行地执行任务（例如，多个设备），将计算图分解为多个部分就变得很重要。执行此图（若该图被划分为子图，则执行子图）称为单个任务，其中任务被分配给单个 TensorFlow 服务器。

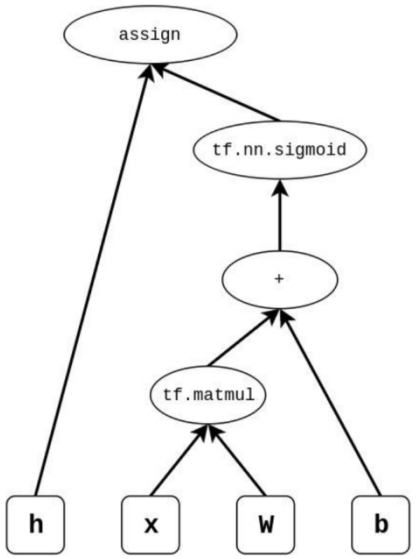

图 3-6　客户端的计算图

然而，在现实中，每个任务都是通过将其分解为两个部分来执行的，每个部分都是由单个 worker 执行的：

- 一个 worker 使用参数的当前值（运算执行器）执行 TensorFlow 操作。
- 一个 worker 存储参数，并使用执行运算器后获得新值来更新它们（参数服务器）。

TensorFlow 客户端的通用工作流程如图 3-7 所示。

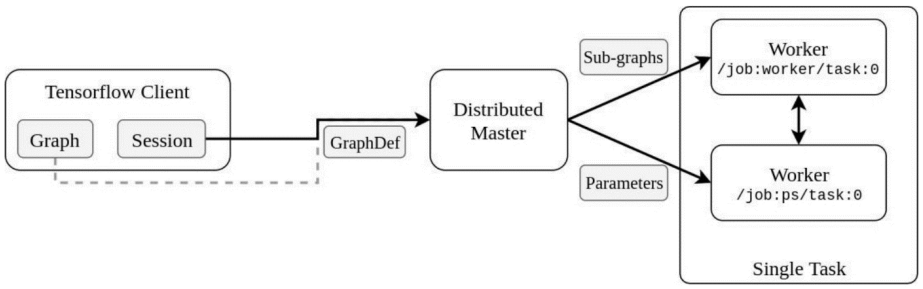

图 3-7　TensorFlow 客户端的通用执行路线图

图 3-8 说明了计算图的分解情况。除了分解计算图之外，TensorFlow 还插入发送节点和接收节点，以利于参数服务器和运算执行器之间的通信。我们可以理解为每当数据可用时，发送节点就发送数据，其中接收节点在对应的发送节点发送数据时继续侦听和捕获数据。

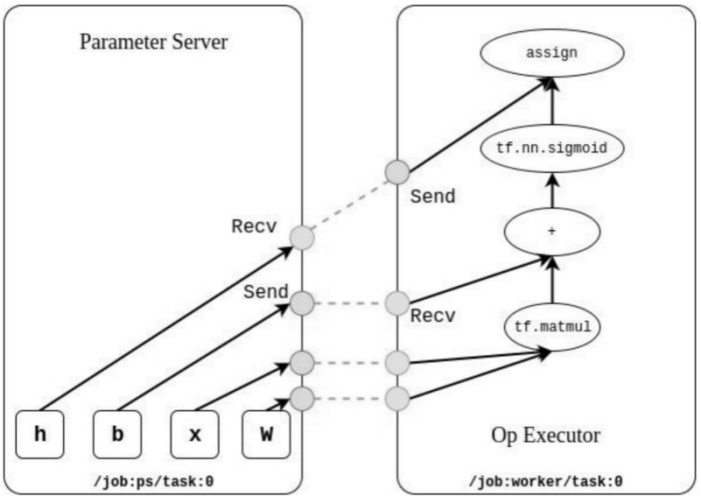

图 3-8　计算图的分解情况

最后，一旦计算完成，会话就将更新的数据从参数服务器端带回到客户端。从现在技术角度来看，我们也可以给出 TensorFlow 的技术架构，如图 3-9 所示。此解释基于 https://tensorflow.google.cn/extend/architecture 上的官方 TensorFlow 文档。

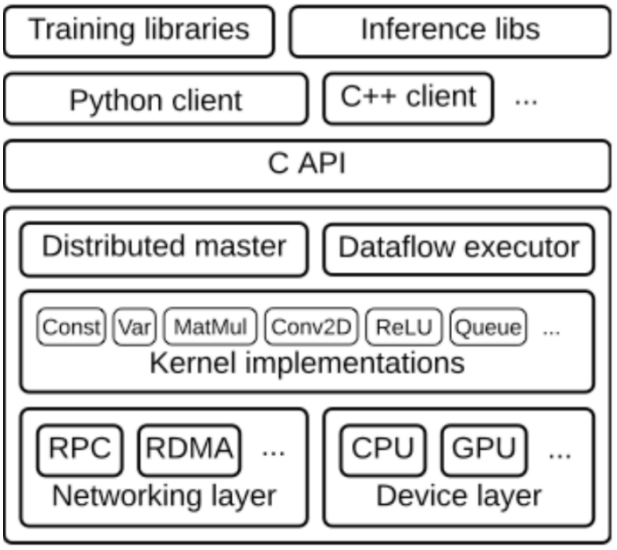

图 3-9　TensorFlow 技术架构

3.9　TensorFlow 中常见元素解读

在了解底层架构的同时，我们下面将介绍构成 TensorFlow 客户端中最常见的元素。

- **输入（Input）**：用于训练和测试算法的数据。

- **变量（Variable）**：可变张量，主要定义算法的参数。
- **输出（Output）**：存储终端和中间输出的不可变张量。
- **运算或操作（Operation）**：输入的各种变换以产生期望的输出。

在之前的 sigmoid 示例中，我们可以找到相关类别的实例，如表 3-2 中所示。

表 3-2　sigmoid 例子中相关类别的实例

TensorFlow 元素	示例客户端的值
Input—输入	x
Variable—变量	W 和 b
Output—输出	h
Operation—运算操作	tf.matmul(...), tf.nn.sigmoid(...)

接下来，我们将详细解读 TensorFlow 中的这些元素。

3.9.1　在 TensorFlow 中定义输入

客户端主要以下面三种不同的方式接收数据：

- 使用 Python 代码在算法的每个步骤中提供数据。
- 将数据预加载并存储为 TensorFlow 张量。
- 构建输入管道。

接下来，具体看看这三种不同的方式。

1. 使用 Python 代码提供数据

在第一种方法中，可以使用传统的 Python 代码方法将数据提供给 TensorFlow 客户端。在我们之前的示例中，x 是此方法的示例。为了从外部数据结构（例如，numpy.ndarray）向客户端提供数据，TensorFlow 库提供了一种极佳的数据结构符号，称为占位符（Placeholder），定义为 tf.placeholder(...)。前面我们说过，占位符在计算图构建阶段不需要实际数据，相反，我们仅通过执行 session.run(...，feed_dict = {placeholder：value})调用的图时提供数据，方法是将外部数据以 Python 字典的形式传递给 feed_dict 参数。占位符的定义如下：

```
tf.placeholder(dtype, shape=None, name=None)
```

参数如下：

- dtype：这是提供占位符的数据的类型。
- shape：这是占位符的 shape，以一维向量给出。
- name：这是占位符的名称，对于调试很重要。

2. 将数据预加载并存储为张量

第二种方法与第一种方法类似，但有一点需要注意。我们不必在计算图执行期间提供数据，因为数据是预先加载的。为了了解这一点，让我们修改一下 sigmoid 示例，将 x 定义为占位符：

```
x = tf.placeholder(shape=[1,10],dtype=tf.float32,name='x')
```

另外，也将 x 定义为包含具体数值的张量：

```
x = tf.constant(value=[[0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1.0]],
dtype=tf.float32,name='x')
```

其完整代码将变为如下：

```
import tensorflow as tf
# 定义 graph 和 session
graph = tf.Graph()

session = tf.InteractiveSession(graph=graph)
# 创建计算图 graph
#x - 预加载的输入
x = tf.constant(value=[[0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1.0]],
dtype=tf.float32,name='x')
W = tf.Variable(tf.random_uniform(shape=[10,5], minval=-0.1, maxval=0.1,
dtype=tf.float32),name='W') # 变量
# 相关变量
b = tf.Variable(tf.zeros(shape=[5],dtype=tf.float32),name='b')
          h = tf.nn.sigmoid(tf.matmul(x,W) + b) # 将要执行的运算
          # 在图中执行运算和评估节点
          tf.global_variables_initializer().run() # 初始化变量
          #执行不带有 feed_dict 的运算
          h_eval = session.run(h)
          print(h_eval)
          #关闭会话，释放资源
          session.close()
```

其实，可以发现这里与原来 sigmoid 示例存在两个主要区别。我们以不同的方式定义了 x。现在直接指定一个具体值并将 x 定义为张量，而不是使用占位符对象并在计算图执行时输入实际值。另外，正如你看到的，我们在 session.run(...)中不会提供任何额外的参数。但是，在缺点方面，现在我们无法在 session.run(..)处向 x 提供不同的值并查看输出是如何变化的。

3. 构建数据输入管道

输入管道是专门为需要快速处理大量数据的“重量级”客户端而设计的。这实际上创建了一个保存数据的队列，直到等到需要它的时候为止。TensorFlow 还提供了各种预处理步骤（例如，用于调整图像对比度/亮度或者标准化），在将数据提供给算法之前执行。为了提高效率，可以让多

个线程并行读取和处理数据。

（1）数据输入管道的结构

TensorFlow 数据输入管道也可以被抽象为一个 ETL 过程（Extract、Transform、Load）。

- Extract: 从硬盘上读取数据，可以是本地（HDD 或 SSD），也可以是网盘（GCS 或 HDFS）。
- Transform: 使用 CPU 去解析、预处理数据，比如图像解码、数据增强、变换（比如随机裁剪、翻转、颜色变换）、打乱（shuffle）、分批量（batching）。
- Load: 将 Transform 后的数据加载到计算设备，例如 GPU、TPU 等设备。

这种模式有效地利用了 CPU，从而让 GPU、TPU 等设备专注于进行模型的训练过程（提高了设备的利用率）。

具体来看，典型的管道将包含以下组件：

- 文件名列表。
- 文件名队列，为输入（记录）阅读器生成文件名。
- 用于读取输入的记录阅读器（记录）。
- 解码读取记录的解码器（例如，JPEG 图像解码）。
- 预处理步骤（可选）。
- 一个示例（解码输入）队列。

（2）数据输入管道（Pipeline）的优化

随着新的计算设备（例如 GPU 和 TPU）的应用，使得以越来越快的速度训练神经网络成为可能，CPU 处理方式很容易遇到海量数据计算的瓶颈。tf.data API 提供数据输入管道所需的各种部件，可以对数据输入管道进行优化。

在执行一个训练 step 之前，必须先做 Extract、Transform 训练数据，然后将其提供给计算设备上运行的模型。在以前，当 CPU 在准备数据时，计算设备处于闲置状态；当计算设备执行训练模型时，CPU 又处于闲置状态。因此，单个训练 step 的时间等于 CPU 准备数据的时间和计算设备执行训练 step 时间的总和。

Pipelining 操作将训练 step 中的数据准备和模型执行实现了并行操作。当计算设备在执行第 N 个训练 step 时，CPU 将为第 N+1 个训练 step 准备数据。这样，通过两个过程的重叠，单个训练 step 的时间将取 CPU 准备数据的时间和计算设备执行训练 step 的时间中的较大值。

如果我们没有进行管道化（Pipelining）操作，CPU 和 GPU/TPU 中大部分时间处于空闲状态，如图 3-10 所示。

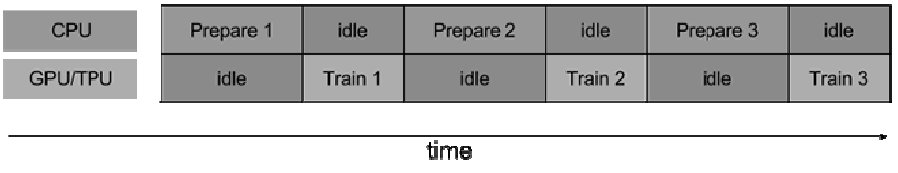

图 3-10 没有使用 Pipelining 操作的 CPU 和 GPU / TPU 处于空闲状态的时间示意图

我们如果使用 Pipelining 操作，CPU 和 GPU/TPU 中处于空闲状态的时间显著减少，如图 3-11 所示。

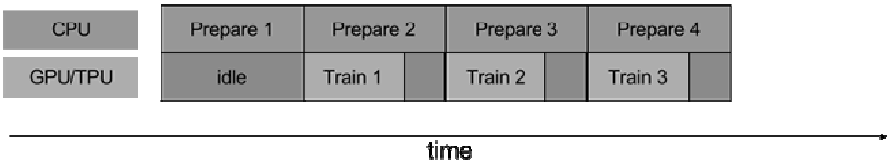

图 3-11　使用 Pipelining 操作的 CPU 和 GPU/TPU 处于空闲状态的时间示意图

提示

有关更多信息，请参阅 https://tensorflow.google.cn/guide/performance/datasets#input_pipeline_structure 上有关数据输入管道的官方说明。

（3）编写输入管道示例

让我们使用 TensorFlow 编写一个输入管道示例。在这个例子中，我们有三个 CSV 格式的文本文件（text1.txt，text2.txt 和 text3.txt），每个文件有 5 行，每行有 10 个用逗号分隔的数字（示例行：0.1、0.2、0.3、0.4、0.5、0.6、0.7、0.8、0.9、1.0）。接着，我们具体来看一下。

提示

更多相关信息，请参阅 https://tensorflow.google.cn/programmers_guide/reading_data 上有关导入数据的 TensorFlow 官方说明。

首先，和以前一样导入一些重要的库：

```
import tensorflow as tf
import numpy as np
```

接下来，我们将定义计算图 graph 和会话 session 对象：

```
graph = tf.Graph()
session = tf.InteractiveSession(graph=graph)
```

然后，我们将定义一个文件名队列，一个包含文件名的队列数据结构。这将作为参数传递给 reader（后面将被定义）。队列将根据 reader 的请求生成文件名，以便读者可以使用这些文件名获取文件进而读取数据：

```
filenames = ['test%d.txt'%i for i in range(1,4)]
filename_queue = tf.train.string_input_producer(filenames, capacity=3,

shuffle=True,name='string_input_producer')
```

这里，capacity 是在给定时间时队列中保存的数据量，shuffle 告诉队列是否在发出数据之前对数据进行重新打乱（shuffle）。

TensorFlow 有几种不同类型的 reader（https://tensorflow.google.cn/api_guides/python/io_ops#Readers 上提供了可用 reader 列表）。由于我们有一些单独的文本文件，其中一行代表一个数据点，因此，这里 TextLineReader 是最适合的：

```
reader = tf.TextLineReader()
```

在定义 reader 之后，我们可以使用 read()函数从文件中读取数据。它输出的是“键-值”对。该键识别出文件和该文件中正在读取的记录（文本行），我们可以省略这个。该值返回 reader 所读取行的实际值：

```
key, value = reader.read(filename_queue, name='text_read_op')
```

下面我们将定义 record_defaults，如果发现存在任何错误记录提示，将给出如下输出：

```
record_defaults = [[-1.0], [-1.0], [-1.0], [-1.0], [-1.0], [-1.0],[-1.0], [-1.0],
[-1.0], [-1.0]]
```

现在，我们将读取的文本行解码为数字列（就像我们的 CSV 文件一样）。为此，我们使用 decode_csv()方法。打开文件（例如，test1.txt），将看到在一行中有 10 列：

```
col1, col2, col3, col4, col5, col6, col7, col8, col9, col10 =tf.decode_csv(value,
record_defaults=record_defaults)
```

然后，我们将连接这些列来组成单个张量（称之为特征），这些张量将被传递给另一个方法 tf.train.shuffle_batch()。 tf.train.shuffle_batch()方法采用先前定义的张量，随机填充张量并输出一批给定的批量大小：

```
features = tf.stack([col1, col2, col3, col4, col5, col6, col7, col8,col9, col10])
x = tf.train.shuffle_batch([features], batch_size=3, capacity=5,
                                    name='data_batch', min_after_dequeue=1,
                                          num_threads=1)
```

batch_size 参数是我们在给定 step 中采样的数据批量大小，capacity 是数据队列的容量（大型队列需要更多内存），min_after_dequeue 表示部分元素出队后还留在队列中的最小元素数量。最后，num_threads 定义了用于生成一批数据的线程数。如果管道中进行了大量预处理，可以增加这个线程数量。此外，如果我们需要在不进行 shuffle 的情况下读取数据（与 tf.train.shuffle_batch 一样），就可以调用 tf.train.batch 方法。接着，我们将通过以下命令来启动此管道：

```
coord = tf.train.Coordinator()
threads = tf.train.start_queue_runners(coord=coord, sess=session)
```

可以将 tf.train.Coordinator()类看成为线程管理器。它控制着各种管理线程的机制。我们需要 f.train.Coordinator()类，因为输入管道产生许多线程来填充入队队列、出列队列以及许多其他任务。接下来，我们将使用之前创建的线程管理器去执行 tf.train.start_queue_runners(...)。QueueRunner()保存队列的入队操作，这些操作是在定义输入管道时自动创建的。因此，要填充已定义的队列，我

们需要调用 tf.train.start_queue_runners 函数来启动这些队列运行的程序。

接下来，在指定任务完成之后，我们需要停止相关线程并将它们连接到主线程，否则眼前的程序将无限期挂起。这是通过调用 coord.request_stop()和 coord.join(threads)来实现的。这个数据输入管道与我们的 sigmoid 示例相结合，便能够直接从文件中读取数据，如下所示：

```
    import tensorflow as tf
    import numpy as np
    import os
    # 定义 graph 和 session
    graph = tf.Graph()
    session = tf.InteractiveSession(graph=graph)
    ### 创建数据输入管道 ###
    # 文件名队列
    filenames = ['test%d.txt'%i for i in range(1,4)]
    filename_queue = tf.train.string_input_producer(filenames, capacity=3,
    shuffle=True,name='string_input_producer')
    # 检查所有文件是否存在
    for f in filenames:
    if not tf.gfile.Exists(f):
        raise ValueError('Failed to find file: ' + f)

    else:
        print('File %s found.'%f)
    #reader 接受一个文件名队列和 read()函数，read()函数依次输出数据
    reader = tf.TextLineReader()
    # 输出“键-值”对
    key, value = reader.read(filename_queue, name='text_read_op')
    # 如果在读取文件时遇到任何问题，这是返回的值
    record_defaults = [[-1.0], [-1.0], [-1.0], [-1.0], [-1.0], [-1.0],[-1.0], [-1.0],
[-1.0], [-1.0]]
    # 将读取到的数值解码为列
    col1, col2, col3, col4, col5, col6, col7, col8, col9, col10 =
    tf.decode_csv(value, record_defaults=record_defaults)

    # 现在我们将这些列叠加在一起，形成一个包含所有列的单个张量
    features = tf.stack([col1, col2, col3, col4, col5, col6, col7, col8,col9, col10])
    # 输出 x 被随机分配一批 batch_size 数据，其中从.txt 文件中读取数据
    x = tf.train.shuffle_batch([features], batch_size=3, capacity=5,
name='data_batch',min_after_dequeue=1,num_threads=1)
    #QueueRunner 从队列中检索数据，我们需要显式地启动它们
    # Coordinator 协调多个 QueueRunners
    coord = tf.train.Coordinator()
    threads = tf.train.start_queue_runners(coord=coord, sess=session)
```

```
    # 通过定义变量和计算来构建计算图
    W = tf.Variable(tf.random_uniform(shape=[10,5], minval=-0.1,maxval=0.1,
dtype=tf.float32),name='W') # 变量
    # 相关变量
    b = tf.Variable(tf.zeros(shape=[5],dtype=tf.float32),name='b')
    h = tf.nn.sigmoid(tf.matmul(x,W) + b) # 要执行的运算
    #在图中执行运算和评估各节点
    tf.global_variables_initializer().run() # 初始化变量
    # 用 x 计算 h 并打印 5 个步的结果
    for step in range(5):
    x_eval, h_eval = session.run([x,h])
    print('========== Step %d ==========='%step)
    print('Evaluated data (x)')
    print(x_eval)
    print('Evaluated data (h)')
    print(h_eval)
    print('')
    # 关闭 coordinator，否则程序将无限期挂起
    coord.request_stop()
    coord.join(threads)
    session.close()
```

3.9.2 在 TensorFlow 中定义变量

变量在 TensorFlow 中扮演着重要角色。变量本质上是一个张量，具有特定的形状（shape），该形状（shape）用于定义变量将具有多少个维度以及每个维度的大小。然而，与常规张量不同，变量是可变的，也就意味着变量的值在定义后是可以改变的。这是实现学习模型参数（例如，神经网络权重值）的理想属性，其中权重值在每个学习步骤之后会稍有变化。例如，如果使用 x = tf.Variable(0,dtype=tf.int32)定义变量，则可以使用诸如 tf.assign(x,x+1)之类的 TensorFlow 运算来更改该变量的值。但是，如果这样定义张量，例如 x = tf.constant(0，dtype = tf.int32)，就无法更改该张量的值，它应该保持 0 的状态直到程序执行结束。

变量的创建很简单。在我们 sigmoid 的示例中，已经创建了两个变量 W 和 b。在创建变量时，有一些事情非常重要，在这里列出它们并在后续中详细讨论：

- Variable shape：变量形状。
- Data type：数据类型。
- Initial value：初始值。
- Name (optional)：名称（可选）。

变量形状是[x，y，z，...]格式的一维向量。列表中的每个值表示相应维度或轴的大小。例如，如果需要具有 50 行和 10 列的二维张量作为变量，那其形状将是 [50,10]。

变量的维数（形状向量的长度）在 TensorFlow 中被识别为张量的秩。不要把它与矩阵的秩混淆起来。

提示

TensorFlow 中张量的秩表示张量的维数；但对于二维矩阵来说，其秩等于 2。

其实，数据类型在确定变量大小方面起着重要作用。有许多不同的数据类型，包括常用的 tf.bool、tf.uint8、tf.float32 和 tf.int32（代码中的字段类型，其意义基本上都是通用的，这和我们平时编写 Java 或.net 程序时用到的布尔类型、浮点类型、整数类型类似，而 unit8 是 8 位无符号整型）。每种数据类型都具有属于该类型的单个值所需的位数（bit）。例如，tf.uint8 类型需要有 8 位，而 tf.float32 需要 32 位。通常的做法是使用相同的数据类型进行计算，否则会导致数据类型的不匹配。因此，如果需要对两个具有不同数据类型的张量进行转换，需要使用 tf.cast(...)操作将一个张量显式地转换为另一个张量的类型。例如，对于具有 tf.int32 数据类型的 x 变量，需要将其转换为 tf.float32 的类型，tf.cast(x，dtype=tf.float32)的调用就是将 x 变量转换为 tf.float32 类型。

接下来，我们需要对变量进行初始化，TensorFlow 也提供了几种不同的初始化器，包括常数初始化器和正态分布初始化器。 TensorFlow 一些主要的初始化器如下所示：

- tf.zeros
- tf.constant_initializer
- tf.random_uniform
- tf.truncated_normal

最后，变量的名称（name）将作为一个标识符（ID），使我们能够在图（Graph）中对该变量进行识别。因此，如果我们对计算图进行可视化操作，那变量将通过传递 name 关键字参数的形式进行显示。如果未指定名称，TensorFlow 将使用默认命名方案。

注意

Python 变量 tf.variable 是赋给计算图的，它不是 TensorFlow 变量命名的一部分。考虑下面的示例，可以在其中指定 TensorFlow 变量：

```
a = tf.Variable(tf.zeros([5]), name ='b')
```

这里，TensorFlow 的图将通过名称 b 而不是 a 来识别该变量。

3.9.3　定义 TensorFlow 输出

TensorFlow 的输出通常是张量，可以是输入或变量或两者转换后的结果。在我们的例子中，h 是一个输出，其中 h = tf.nn.sigmoid（tf.matmul（x，W）+ b）。当然，有时候 TensorFlow 的一个输出也可能变成下一个输入的内容，依次输出下去可以形成一组链式运算或操作，而这里的运算或操作也不一定必须是 TensorFlow 运算或操作，还可以使用标准 Python 算法，例如：

```
x = tf.matmul(w,A)
y = x + B
z = tf.add(y,C)
```

3.9.4 定义 TensorFlow 运算或操作

在 https://tensorflow.google.cn/api_docs/python/上查看 TensorFlow API，你会发现 TensorFlow 有大量可用的运算或操作。下面，我们将对 TensorFlow 中几个常见的运算或操作进行解读。

1．比较运算

比较运算对于比较两个张量非常有用。对于比较运算部分的详细资料，读者可以查阅 https://github.com/tensorflow/docs/tree/master/site/en/api_guides/python 中比较运算符的详细内容。当然，对于比较运算工作原理的理解，通过代码层面可能会更直观些。这里，我们给出示例张量 x 和 y：

```
#假设 x 和 y 的取值如下
#x (2-D tensor) => [[1,2],[3,4]]
#y (2-D tensor) => [[4,3],[3,2]]
x= tf.constant([[1,2],[3,4]], dtype=tf.int32)
y= tf.constant([[4,3],[3,2]], dtype=tf.int32)

# 检查两个张量在元素方面是否相等，并返回布尔类型的张量
# x_equal_y => [[False,False],[True,False]]
x_equal_y = tf.equal(x, y, name=None)
#检查 x 在对应元素方面是否小于 y 并返回布尔类型的张量
# x_less_y => [[True,True],[False,False]]
x_less_y = tf.less(x, y, name=None)
# 检查 x 在对应元素方面是否大于或等于 y 并返回布尔类型的张量
# x_great_equal_y => [[False,False],[True,True]]

x_great_equal_y = tf.greater_equal(x, y, name=None)
# 从 x 和 y 中选择元素，具体取决于条件是否满足（从 x 中选择元素）或
# 条件失败（从 y 中选择元素）
condition = tf.constant([[True,False],[True,False]],dtype=tf.bool)
# x_cond_y => [[1,3],[3,2]]
x_cond_y = tf.where(condition, x, y, name=None)
```

2. 比较数学运算

TensorFlow 允许我们对从简单到复杂的张量执行数学运算。这里我们将讨论 TensorFlow 中提供的一些数学运算。完整的运算集可在 https://github.com/tensorflow/docs/tree/master/site/en/api_guides/python 上找到。

```
#假设 x 和 y 的取值如下
#x (2-D tensor) => [[1,2],[3,4]]
#y (2-D tensor) => [[4,3],[3,2]]
x= tf.constant([[1,2],[3,4]], dtype=tf.float32)
y = tf.constant([[4,3],[3,2]], dtype=tf.float32)
# 以元素方式增加两个张量 x 和 y
# x_add_y => [[5,5],[6,6]]
x_add_y = tf.add(x, y)
# 执行矩阵乘法（不是元素方面）
# x_mul_y => [[10,7],[24,17]]
x_mul_y = tf.matmul(x, y)
# 计算 x 元素的自然对数，相当于计算 ln（x）
# log_x => [[0,0.6931],[1.0986,1.3863]]
log_x = tf.log(x)
# 在指定轴上执行归约（reduction）
# x_sum_1 => [3,7]
x_sum_1 = tf.reduce_sum(x, axis=[1], keepdims=False)
# x_sum_2 => [[4],[6]]
x_sum_2 = tf.reduce_sum(x, axis=[0], keepdims=True)

# 根据 segment_ids（同一段中具有相同 id 的项）对张量进行分段，并计算数据的分段总和
data = tf.constant([1,2,3,4,5,6,7,8,9,10], dtype=tf.float32)
segment_ids = tf.constant([0,0,0,1,1,2,2,2,2,2 ], dtype=tf.int32)
# x_seg_sum => [6,9,40]
x_seg_sum = tf.segment_sum(data, segment_ids)
```

3. 比较分散（Scatter）和聚合（Gather）操作

随着人工智能的迅猛发展，尤其是 GPU 可编程性能的增强以及 GPGPU（General Purpose Computing on GPU，在图形处理器上进行通用计算）技术的不断发展，相关研究/技术人员也迫切希望基于流处理器模型的 GPU 也可以和 CPU 一样，在支持流程分支的同时，也能够实现对存储器进行灵活的读写操作。其实，Ian Buck 在进行早期的 GPU 通用可编程技术研究时，就发现 GPU 完成复杂计算任务时存在一个关键性的缺陷，那就是缺乏灵活的存储器操作。所以，他在后来的研究中就增加了对分散和聚合操作的支持，但是结果还是以牺牲一些性能为代价的情况下完成了整个过程。

在 GPU 中，CUDA 中分散和聚合操作实现的结构示意图与第一向量机中的很相似，分散允许将数据输出到非连续的存储器地址内，而聚合则允许从非连续的存储器地址内读取数据。因此，如果认为存储器（如 DRAM）是一个二维数组，分散则可以看作利用数组下标或索引将数据写入数组中的任意位置，即 a[i] = x，而聚合则可以看作是利用数组下标或索引从数组中的任意位置读出数据，即 x = a[i]。

下面，我们给出 CUDA 中分散（Scatter）和聚合（Gather）操作的结构示意图，如图 3-12 所示。其中，每个 ALU 可以看作是一个处理核心，通过分散/聚合操作，多个 ALU 之间可以共享存

储器，实现对任意地址数据的读写操作。

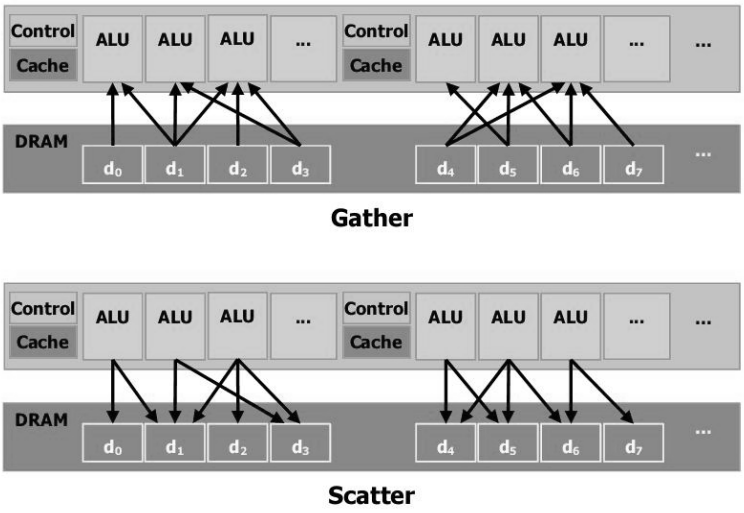

图 3-12　CUDA 中分散（Scatter）和聚合（Gather）操作的结构示意图

分散和聚合操作在矩阵运算任务中起着至关重要的作用，因为目前来看这两种变体在 TensorFlow 中是把张量编入索引的唯一方法。换句话说，不能像在 NumPy 中那样访问 TensorFlow 中的张量元素。分散操作允许我们将值分配给指定张量的特定索引，而聚合操作允许我们提取指定张量的切片（或单个元素）。以下代码显示了分散和聚合操作的一些变体：

```
    # 1-D scatter 操作
    ref = tf.Variable(tf.constant([1,9,3,10,5],dtype=tf.float32),
name='scatter_update')
    indices = [1,3]
    updates = tf.constant([2,4],dtype=tf.float32)
    tf_scatter_update = tf.scatter_update(ref, indices, updates, use_locking=None,
name=None)
            # n-D scatter 操作
            indices = [[1],[3]]
            updates = tf.constant([[1,1,1],[2,2,2]])
            shape = [4,3]
            tf_scatter_nd_1 = tf.scatter_nd(indices, updates, shape, name=None)
            # n-D scatter 操作
            indices = [[1,0],[3,1]] # 2 x 2
            updates = tf.constant([1,2]) # 2 x 1
            shape = [4,3] # 2
            tf_scatter_nd_2 = tf.scatter_nd(indices, updates, shape, name=None)
            # 1-D gather 操作
            params = tf.constant([1,2,3,4,5],dtype=tf.float32)
            indices = [1,4]
```

```
            tf_gather = tf.gather(params, indices, validate_indices=True,name=None)
#=> [2,5]
            # n-D gather 操作
            params =
tf.constant([[0,0,0],[1,1,1],[2,2,2],[3,3,3]],dtype=tf.float32)
            indices = [[0],[2]]
            tf_gather_nd = tf.gather_nd(params, indices, name=None)
#=>[[0,0,0],[2,2,2]]
            params =
tf.constant([[0,0,0],[1,1,1],[2,2,2],[3,3,3]],dtype=tf.float32)
            indices = [[0,1],[2,2]]
           tf_gather_nd_2 = tf.gather_nd(params, indices, name=None)
#=>[[0,0,0],[2,2,2]]
```

4. 比较与神经网络相关的运算或操作

下面让我们看看几个很有用的神经网络运算或操作，这些将在后面的章节中使用到。这里，我们会对简单元素的转换进行讨论，也会对一组参数对于另一个值的偏导数的运算进行讨论，并给出一个简单的神经网络实现例子。

（1）神经网络的非线性激活

非线性激活能够使神经网络很好地执行许多任务。通常，在神经网络的每一层输出后（最后一层除外）都会有一个非线性的激活转换（激活层）。非线性变换有助于神经网络学习数据中出现的各种非线性模式。这对于解决现实世界中复杂的问题非常有用，因为与线性模式相比，数据通常具有更复杂的非线性模式。

提示

让我们通过一个例子来观察一下非线性激活的重要性。首先，回想一下我们在 sigmoid 示例中看到的神经网络的计算。如果我们忽视 b，那将是这样的：

```
h = sigmoid(W*x)
```

假设有一个三层神经网络（W1、W2 和 W3 是层的权重值），其中每层都执行前面的计算；我们可以给出完整计算：

```
h = sigmoid(W3*sigmoid(W2*sigmoid(W1*x)))
```

但是，如果我们删除非线性激活（sigmoid），我们就可以得到：

```
h = (W3 * (W2 * (W1 *x))) = (W3*W2*W1)*x
```

因此，在没有非线性激活的情况下，可以将三个层降为单个线性层。

如果没有各层之间的非线性激活，深度神经网络就将是一堆相互叠加的线性层而已，而且一组线性层基本上可以压缩成一个更大的线性层。综上所述，如果没有非线性激活，我们就无法创建具

有多个层的神经网络。

现在，我们将列出神经网络中两种常用的非线性激活函数以及它们是如何在 TensorFlow 中实现的：

```
#x 的 Sigmoid 激活由 1 / (1 + exp (-x) ) 给出
tf.nn.sigmoid(x,name=None)
#x 的 ReLU 激活由 max(0,x)给出
tf.nn.relu(x, name=None)
```

①卷积运算

卷积运算是一种广泛使用的信号处理技术。对于图像，卷积运算可以给出不同的图像效果。这里，图 3-13 给出了使用卷积进行边缘检测（包括横向边缘检测和纵向边缘检测）的示例。我们可以通过在图像顶部移动卷积过滤器以便在每个位置产生不同的输出来实现边缘检测，如图 3-14 所示（在本书第 5 章介绍卷积神经网络时会详细解读卷积运算的工作原理）。具体来说，在每个位置，我们使用与卷积过滤器重叠的图像块（与卷积过滤器相同的大小）对卷积过滤器中的元素进行逐元素乘法，并获取乘法的总和。

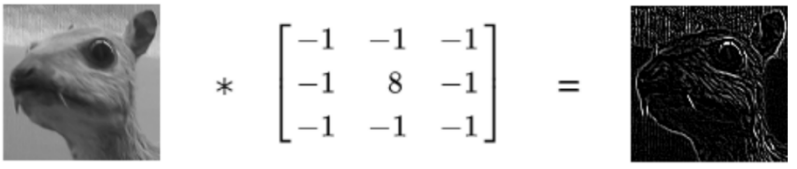

图 3-13　利用卷积运算在图像中进行边缘检测示意图

提示

源自 https://en.wikipedia.org/wiki/Kernel_(image_processing)。

以下是卷积运算的实现：

```
x = tf.constant(
            [[
                [[1],[2],[3],[4]],
                [[4],[3],[2],[1]],
                [[5],[6],[7],[8]],
                [[8],[7],[6],[5]]
          ]],
              dtype=tf.float32)
   x_filter = tf.constant(
       [
         [
           [[0.5]],[[1]]
         ],
         [
```

```
        [[0.5]],[[1]]
      ]
  ],
      dtype=tf.float32)
    x_stride = [1,1,1,1]
    x_padding = 'VALID'
    x_conv = tf.nn.conv2d(
    input=x, filter=x_filter,
    strides=x_stride, padding=x_padding
)
```

这里，对于 tf.conv2d(...)方法中涉及的 input 、filter 和 stride 等参数格式而言，TensorFlow 对它们的要求是很精确的，下面我们将对这些参数（input、filter、strides、padding）做进一步的解释。

- 输入（input）：通常是四维张量，其尺寸应按[batch_size，height，width，channels]排序。
  - ★ **batch_size**：这是一批数据中的数据量（例如，输入的图像和单词等）。我们通常按批量处理数据，因为模型可以使用大型数据集进行深入学习。在给定的训练步骤（step）中，我们随机抽样一小部分可以大致代表完整的数据集的数据，然后重复足够多次该操作，我们便可以很好地逼近这个完整的数据集。此 batch_size 参数与我们在 TensorFlow 输入管道示例中讨论的参数相同。
  - ★ **height and width**：这是输入的高度和宽度。
  - ★ **channels**：这是输入的深度（例如，对于 RGB 图像，将为 3 通道）。
- 过滤器（filter）：这是一个四维张量，表示卷积运算的卷积窗口。过滤器尺寸应为[height，width，in_channels，out_channels]。
  - ★ height and width：这是滤镜的高度和宽度（通常小于输入的高度和宽度）。
  - ★ in_channels：这是图层输入的通道数。
  - ★ out_channels：这是在图层输出中生成的通道数。
- 步幅（strides）：这是一个包含四个元素的列表，具体为 [batch_stride，height_stride，width_stride，channels_stride]。
- 填充（padding）：这里可以选择['SAME'，'VALID']中的任何一个选项。它能够决定如何处理输入边界附近的卷积运算。VALID 操作是在没有填充的情况下执行卷积。如果我们用大小为 h 的卷积窗口、卷积长度为 n 的输入，这将给出输出的尺寸（或大小）。输出尺寸的减小会严重限制神经网络的深度。SAME 将零填充到边界，使输出具有与输入相同的高度和宽度。

为了更好地了解过滤器大小、步幅和填充是什么，请参见图 3-14（我们在本书第 5 章介绍卷积神经网络时做详细解读）。

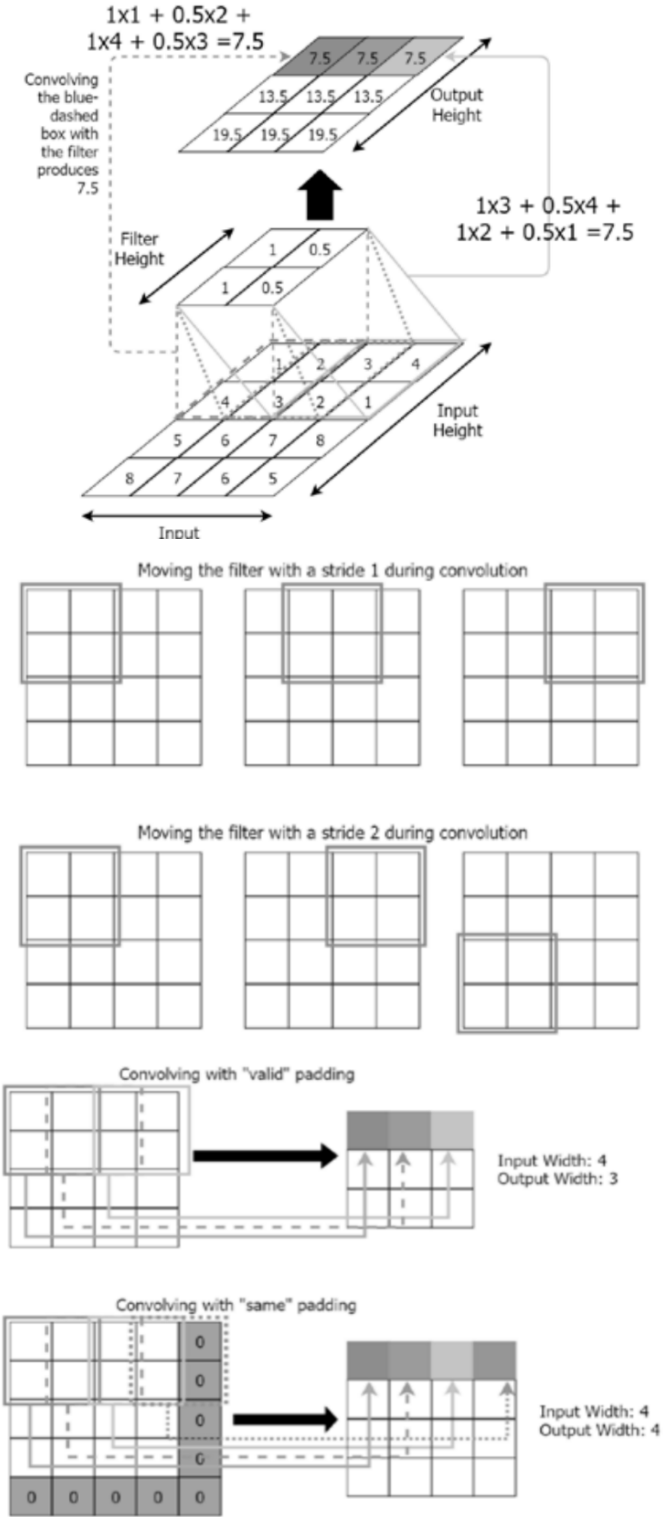

图 3-14　卷积网络运算示意图

②池化操作

池化运算与卷积运算的行为类似，但最终输出是不同的。我们这里选取的是该位置中图像 patch 的最大元素，而不是输出过滤器和图像 patch 中按元素相乘得到的总和（我们在在本书第 5 章介绍卷积神经网络中会做详细解读），如图 3-15 所示。

```
x = tf.constant(
              [[
                  [[1],[2],[3],[4]],
                  [[4],[3],[2],[1]],
                  [[5],[6],[7],[8]],
                  [[8],[7],[6],[5]]
              ]],
             dtype=tf.float32)

        x_ksize = [1,2,2,1]
        x_stride = [1,2,2,1]
        x_padding = 'VALID'
        x_pool = tf.nn.max_pool(
                value=x, ksize=x_ksize,
                 strides=x_stride, padding=x_padding
         )
```

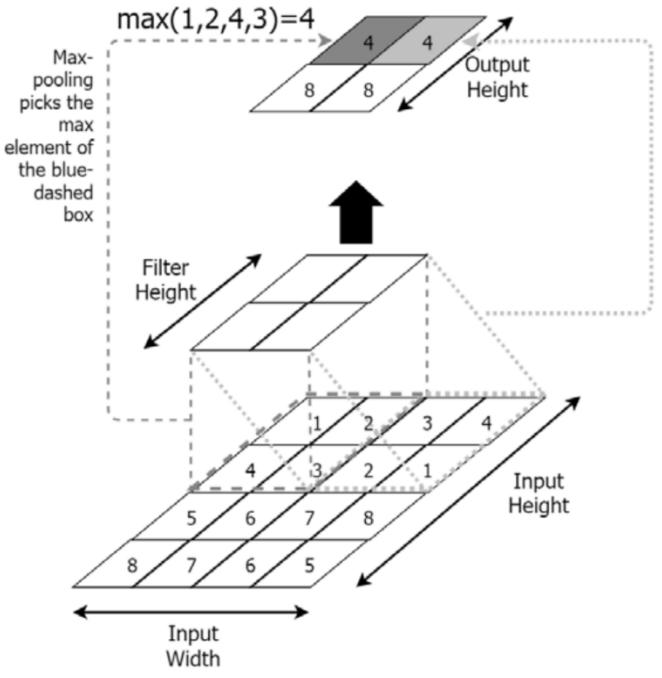

图 3-15　最大池化运算示意图

代码运行后返回的结果如下（完整代码和结果，读者可以查看代码文件中的“二维操作（2D

卷积和 2D 最大池化）”部分）：

```
[[[[ 4.]
[ 4.]],
[[ 8.]
[ 8.]]]]
```

③定义损失

为了让神经网络模型能够学习到有用的东西，我们需要定义一个损失函数。这里有几种可以自动计算 TensorFlow 中损失的函数。其中，tf.nn.l2_loss 是均方误差损失函数，tf.nn.softmax_cross_entropy_with_logits_v2 是交叉熵损失函数。交叉熵损失函数在分类任务中能够使模型表现更佳。这里涉及均方误差损失函数和交叉熵损失函数的代码如下：

```
x = tf.constant([[2,4],[6,8]],dtype=tf.float32)
x_hat = tf.constant([[1,2],[3,4]],dtype=tf.float32)
# MSE = (1**2 + 2**2 + 3**2 + 4**2)/2 = 15
MSE = tf.nn.l2_loss(x-x_hat)
# 神经网络中用于优化网络的常见损失函数
# 使用 logits（最后一层的归一输出）代替输出来计算交叉熵,会使数值获得的更加稳定
y = tf.constant([[1,0],[0,1]],dtype=tf.float32)
y_hat = tf.constant([[3,1],[2,5]],dtype=tf.float32)
# 此函数并不能平均所有数据点的交叉熵损失，我们需要使用 reduce_mean 函数手动实现这一点
CE = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits_v2(logits=y_hat,
labels=y))
```

④神经网络的优化

在定义了神经网络的损失函数之后，我们的目标是随着时间的推移尽量减少这种损失，这个过程就是常说的模型优化工作。换言之，优化器的目标是找到为所有输入提供最小损失的神经网络参数（权重值和偏差）。TensorFlow 为我们提供了几种不同的优化器，因此我们不需要从头开始实现相关模型。

图 3-16 说明了一个简单的优化问题，并显示了优化是如何随时间变化的。该曲线可以想象为损失曲线（对于高维空间的情况，我们称之为损失面），其中 x 可以被认为是神经网络的参数（在这种情况下，是一个单一权重值的神经网络），y 可以被认为是损失。我们初步估计起始点是 x=2 位置。从这一点开始，我们使用优化器来实现在 x=0 处得到最小的 y（损失）。然而，在实际问题中，损失表面不会像图 3-16 中所示的这么简单，它会更加复杂。

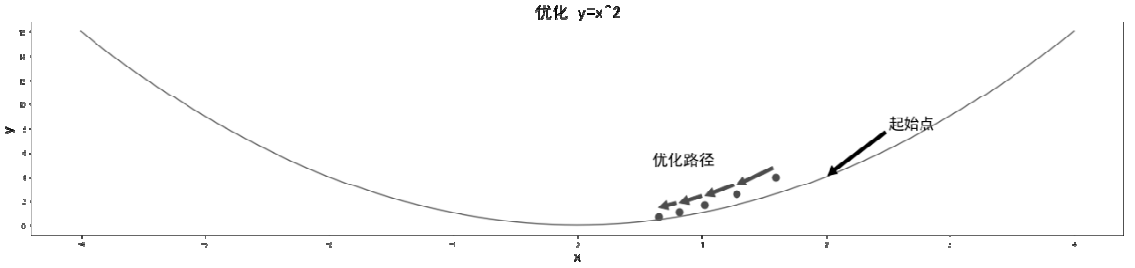

图 3-16　优化过程示意图

在此示例中，我们使用的是常见的梯度下降优化法：GradientDescentOptimizer。learning_rate 参数表示在最小化方向上的步长（图 3-16 中两个圆点之间的距离）：

```
    #优化器起到调整神经网络参数的作用，以便最小化工作任务中的错误
    tf_x = tf.Variable(tf.constant(2.0,dtype=tf.float32),name='x')
    tf_y = tf_x**2
    minimize_op = tf.train.GradientDescentOptimizer(learning_rate=0.1).
minimize(tf_y)
```

完整代码详见代码文件 3_tensorflow_introduction.ipynb 中随机优化（Stochastic Optimization）部分。执行该部分代码后，除了会得到图 3-16 之外，还会得到如下结果：

```
    第  1 个步长上, x:  1.28 , y:  2.5600002
    第  2 个步长上, x:  1.0239999 , y:  1.6384
    第  3 个步长上, x:  0.8191999 , y:  1.0485759
    第  4 个步长上, x:  0.6553599 , y:  0.6710885
```

从上面的代码运行结果来看，显然，当我们每次调用 session.run(minimize_op)执行损失最小化运算时，将会接近 tf_x 值，进而可以得到最小的 tf_y 值。

⑤控制流操作

顾名思义，控制流操作控制图中的执行顺序。例如，假设我们需要按以下顺序执行计算：

```
    x = x+5
    z = x*2
```

实际上，如果 x = 2，我们应该得到 z = 14。下面，让我们尝试以一种简单的方法来实现这一点：

```
    session = tf.InteractiveSession()
    x = tf.Variable(tf.constant(2.0), name='x')
    x_assign_op = tf.assign(x, x+5)
    z = x*2
    tf.global_variables_initializer().run()
    print('z=',session.run(z))
    print('x=',session.run(x))
```

```
session.close()
```

我们期望的输出结果是 x = 7 和 z = 14，而在 TensorFlow 中，上面的代码运行结果却是 x = 2 和 z = 14。引起这种错误的原因是，TensorFlow 不关心对象的执行顺序，除非我们在程序中给出明确的执行顺序。我们本部分讨论的控制流操作，就可以实现执行指定顺序的操作。为了得到期望的运行结果（x = 7 和 z = 14），我们需要对上述代码进行调整，具体如下：

```
session = tf.InteractiveSession()
x = tf.Variable(tf.constant(2.0), name='x')
with tf.control_dependencies([tf.assign(x, x+5)]):
                z = x*2
tf.global_variables_initializer().run()
print('z=',session.run(z))
print('x=',session.run(x))
session.close()
```

这样一来，我们就可以得到想要的结果（x = 7 和 z = 14）了。这里，tf.control_dependencies（...）方法是确保在执行嵌套操作之前将会优先执行参数传递给它的运算操作。读者也可以在代码文件“调用 tf.control_dependencies(...)方法”部分执行这些代码并查看运行结果。

3.10 变量作用域机制

3.10.1 基本原理

目前为止，我们已经对 TensorFlow 架构和 TensorFlow 客户端的实现有了基本的认识。但是，在深度学习过程中，一方面，我们需要减少训练参数的个数（比如 CNN 和 LSTM 模型）或是面对多机多卡并行化训练大数据大模型（比如数据并行化）等情况；另一方面，当我们的深度学习模型变得异常复杂的时候，往往存在大量的变量和调用方法，如何有效地维护这些变量名称和方法名称的唯一性（即不重复），同时又能维护好一个条理清晰的图（Graph）就变得非常重要了。这时，变量共享机制就变得非常重要。比如，我们构建 CNN、LSTM 等模型时，需要使用很多变量集去验证权重值（Weight）和偏差（Bias）等训练参数，非常希望在输入不同的数据时这些参数是可以共享的（本书 5.3.4 小节“参数共享机制”部分会进行详细解读）。过去，我们创建一个全局变量就可以使用了，但在深度学习中则不可以，不方便管理而且使代码的封装性受到极大影响。所以，TensorFlow 提供了一种变量管理方法：变量作用域机制，以此解决上面出现的问题。

关于变量作用域机制，有的文档也叫共享变量机制，根据笔者查阅的文献资料，大部分文章的参考资料来自于 TensorFlow 官方说明（详见 https://tensorflow.google.cn/guide/variables）。TensorFlow 中是通过调用四个函数来进行变量作用域共享的，这四个函数是 tf.Variable(<variable_name>)、tf.get_variable(<variable_name>)、tf.name_scope(<scope_name>)和 tf.variable_scope(<scope_name>)。下面，我们具体来看一下这几个函数。

如果使用 Variable，那么每次都会新建变量，但是大多数时候我们是希望一些变量可以重用的，所以就用到了 get_variable()。get_variable()会去搜索变量名称，搜索结果如果没有就新建变量，如果有就直接使用该变量。既然用到变量名称，这就会涉及名称域的概念。通过不同的域来对变量名称加以区分，因为如果让我们给所有变量都直接取不同名字其实是非常辛苦的且没必要，这就是为什么会用到作用域（Scope）的概念了。name_scope 主要用于图（Graph）中的各种运算，variable_scope 可以通过设置 reuse 标志以及初始化方式来影响作用域中的变量。当然对我们而言，还有一个更直观的感受就是：在使用 tensorboard 可视化的时候用名字作用域进行封装后会更清晰。

3.10.2　通过示例解读

假设我们需要一个执行某种计算的函数，给定 w，需要计算 x * w + y ** 2。让我们编写一个 TensorFlow 客户端：

```
import tensorflow as tf
session = tf.InteractiveSession()
def very_simple_computation(w):
     x = tf.Variable(tf.constant(5.0, shape=None, dtype=tf.float32),name='x')
     y = tf.Variable(tf.constant(2.0, shape=None, dtype=tf.float32),name='y')
     z = x*w + y**2
     return z
```

这里，为了得到想要的结果，我们可以调用 session.run（very_simple_computation（2））函数（当然是在调用 tf.global_variables_initializer().run()函数之后）。实际上，每次调用这个函数时都会创建两个 TensorFlow 变量。在多次调用该方法（在面向对象程序设计中把封装的函数也称为方法）时，图（Graph）中 x 和 y 变量不会被替换，相反，将会保留这些旧变量，并在图（Graph）中创建新的变量，直到内存不足为止。不管如何，最终的结果是正确的。为了更好地验证这个情况，我们在 for 循环中运行 session.run(very_simple_computation(2))方法，并将变量名称也打印出来，循环 10 次的输出结果如下（完整的代码请在 ch3 文件夹中的 3_tensorflow_introduction.ipynb 文件“变量作用域机制（Variable Scoping）”部分查看）：

```
14.0
['x:0', 'y:0', 'x_1:0', 'y_1:0', 'x_2:0', 'y_2:0', 'x_3:0', 'y_3:0', 'x_4:0',
'y_4:0', 'x_5:0', 'y_5:0', 'x_6:0', 'y_6:0', 'x_7:0', 'y_7:0', 'x_8:0', 'y_8:0',
'x_9:0', 'y_9:0', 'x_10:0', 'y_10:0']
```

每次运行该函数时都会创建一对变量。如果运行这个函数 100 次，那么计算图中将有 198 个过时变量（99x 变量和 99y 变量）。

作用域允许我们重复使用变量，而不是每次调用函数时都创建一个新的变量。我们现在为上面的例子添加变量可复用性的操作，将代码更改为以下内容：

```
def not_so_simple_computation(w):
```

```
        x = tf.get_variable('x', initializer=tf.constant (5.0,
shape=None,dtype=tf.float32))
        y = tf.get_variable('y', initializer=tf.constant(2.0,
shape=None,dtype=tf.float32))
        z = x*w + y**2
        return z
    def another_not_so_simple_computation(w):
        x = tf.get_variable('x', initializer=tf.constant(5.0,
shape=None,dtype=tf.float32))
        y = tf.get_variable('y', initializer=tf.constant(2.0,
shape=None,dtype=tf.float32))
        z = w*x*y
        return z
    # 这是第一次调用，因此将使用以下名称创建变量
    # x => scopeA/x, y => scopeA/y
    with tf.variable_scope('scopeA'):
        z1 = not_so_simple_computation(tf.constant(1.0, dtype=tf.float32))
    #我们重复使用已创建的 scopeA/x 和 scopeA/y
    with tf.variable_scope('scopeA',reuse=True):
         z2 = another_not_so_simple_computation(z1)
    # 由于这是第一次调用，因此将使用以下名称创建变量：x => scopeB/x，y => scopeB/y
    with tf.variable_scope('scopeB'):
         a1 = not_so_simple_computation(tf.constant(1.0, dtype=tf.float32))
    #我们重复使用已创建的 scopeB/x 和 scopeB/y

    with tf.variable_scope('scopeB',reuse=True):
         a2 = another_not_so_simple_computation(a1)
    # 假设我们想再次重复使用“scopeA”，因为已经创建了变量，
    #所以我们应该在调用时将“reuse”参数设置为 True
    with tf.variable_scope('scopeA',reuse=True):
         zz1 = not_so_simple_computation(tf.constant(1.0, dtype=tf.float32))
         zz2 = another_not_so_simple_computation(z1)
```

在这个例子中，如果执行 session.run（[z1，z2，a1，a2，zz1，zz2]）操作，就应该会看到 z1,z2,a1,a2,zz1,zz2 的值按顺序为 9.0,90.0,9.0, 90.0,9.0,90.0 值。现在，如果打印变量，应该只看到四个不同的变量：scopeA / x、scopeA / y、scopeB / x 和 scopeB / y。我们现在可以在循环中多次运行它，而不必担心创建冗余变量和内存不足。

现在你可能想知道，为什么我们不能在代码的开头创建这四个变量并在后面的方法中使用它们。如果这样做，就会破坏代码的封装性，因为这样是在明显地依赖于代码之外的内容。

最后，作用域支持了可复用性，同时也保留了代码的封装性。此外，作用域使代码流更直观，减少了错误的可能性，因为我们通过作用域和名称显式地获取变量，而不是使用 TensorFlow 变量分配的 Python 变量。

3.11　实现一个神经网络

目前，我们已经对 TensorFlow 的架构体系和作用域机制有了大体的认知，接下来，我们将实现一个稍微复杂的模型，那就是完全连接的神经网络模型。这里使用的数据集是在深度学习过程中都经常使用的 MNIST 数据集（详见 http://yann.lecun.com/exdb/mnist/），利用神经网络模型能够实现对数字进行分类。

由于这是我们的第一个神经网络示例，因此，我们将逐步介绍其中的关键部分。如果要了解程序从头到尾的运行情况，读者可以在 ch3 文件夹中的 3_tensorflow_introduction.ipynb 文件“MNIST 数字识别分类”部分自行查验。

3.11.1　数据准备

首先，我们需要调用 maybe_download(...)函数来下载数据集，并调用 read_mnist(...)函数对其进行预处理。这里，read_mnist(...)函数执行以下两个主要步骤：

- 读取数据集的字节流并将其形成适当的 NUPYP.NDARRY 对象。将图像标准化为零均值和单位方差。

```
def read_mnist(fname_img, fname_lbl):
     print('\nReading files %s and %s'%(fname_img, fname_lbl))
     with gzip.open(fname_img) as fimg:
          magic, num, rows, cols = struct.unpack(">IIII", fimg.read(16))
          print(num,rows,cols)
          img = (np.frombuffer(fimg.read(num*rows*cols), dtype=np.uint8).
                        reshape(num, rows * cols)).astype(np.float32)
          print('(Images) Returned a tensor of shape ',img.shape)
          # 对图像进行标准化处理
          img = (img - np.mean(img))/np.std(img)
    with gzip.open(fname_lbl) as flbl:
          #flbl.read（8）读取最多 8 个字节
          magic, num = struct.unpack(">II", flbl.read(8))
          lbl = np.frombuffer(flbl.read(num), dtype=np.int8)
          print('(Labels) Returned a tensor of shape: %s'%lbl.shape)
          print('Sample labels: ',lbl[:10])
     return img, lbl
```

3.11.2　定义 TensorFlow 计算图

为了定义 TensorFlow 计算图，我们首先为输入图像（tf_input）和其对应标签（tf_lab）定义占位符：

```
    # 定义输入和输出
    tf_inputs = tf.placeholder(shape=[batch_size, input_size], dtype=tf.float32,
name = 'inputs')
    tf_labels = tf.placeholder(shape=[batch_size, num_labels], dtype=tf.float32,
name = 'labels')
```

接下来，编写一个 Python 函数，它将首次创建变量。需要说明的是，我们使用作用域来确保变量的可重用性，并确保变量被正确命名：

```
    # 定义 TensorFlow 相关变量
    def define_net_parameters():
        with tf.variable_scope('layer1'):
            tf.get_variable(WEIGHTS_STRING,shape=[input_size,500],
                    initializer=tf.random_normal_initializer(0,0.02))
            tf.get_variable(BIAS_STRING, shape=[500],
                    initializer=tf.random_uniform_initializer(0,0.01))

        with tf.variable_scope('layer2'):
            tf.get_variable(WEIGHTS_STRING,shape=[500,250],
                    initializer=tf.random_normal_initializer(0,0.02))
            tf.get_variable(BIAS_STRING, shape=[250],
                    initializer=tf.random_uniform_initializer(0,0.01))

        with tf.variable_scope('output'):
            tf.get_variable(WEIGHTS_STRING,shape=[250,10],
                    initializer=tf.random_normal_initializer(0,0.02))
            tf.get_variable(BIAS_STRING, shape=[10],
                    initializer=tf.random_uniform_initializer(0,0.01))
```

下面，我们将定义神经网络的推理过程。这里需要说明一点，就是与没有使用作用域的变量相比，作用域为函数中的代码提供了非常直观的流程。这里的神经网络有三层，具体如下：

- 具有 ReLU 激活的完全连接层（第 1 层）。
- 具有 ReLU 激活的完全连接层（第 2 层）。
- 完全连接的 softmax 层（输出）。

通过作用域，我们将每个层的变量（权重值和偏差）命名为 layer1 / weight、layer1 / bias、layer2 / weight、layer2 / bias、output / weights 和 output / bias。在下面的代码中，它们都具有相同的名称，但作用域是不同的：

```
    #在神经网络中定义根据输入进行逻辑推理的计算过程
    #logit 是将 softmax 应用到最终输出之前的评估模型

    def inference(x):
```

```
        # calculations for layer 1
        with tf.variable_scope('layer1',reuse=True):
              w,b = tf.get_variable(WEIGHTS_STRING), tf.get_variable(BIAS_STRING)
              tf_h1 = tf.nn.relu(tf.matmul(x,w) + b, name = 'hidden1')

        # calculations for layer 2
        with tf.variable_scope('layer2',reuse=True):
              w,b = tf.get_variable(WEIGHTS_STRING), tf.get_variable(BIAS_STRING)
              tf_h2 = tf.nn.relu(tf.matmul(tf_h1,w) + b, name = 'hidden1')

        # calculations for output layer
        with tf.variable_scope('output',reuse=True):
               w,b = tf.get_variable(WEIGHTS_STRING),
tf.get_variable(BIAS_STRING)
               tf_logits = tf.nn.bias_add(tf.matmul(tf_h2,w), b, name = 'logits')

         return tf_logits
```

现在，我们将定义一个损失函数并将损失进行最小化操作。损失最小化操作是通过在最小化损失方向上对神经网络参数进行微移来开展的。TensorFlow 中提供了多种优化器，我们在这里将使用 MomentumOptimizer，它提供了比 GradientDescentOptimizer 更好的最终精度和收敛性：

```
    # 定义损失函数
    tf_loss =
tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits_v2(logits=inference(tf_
inputs), labels=tf_labels))

    # 定义损失优化函数
    tf_loss_minimize = tf.train.MomentumOptimizer(momentum=0.9,
    learning_rate=0.01).minimize(tf_loss)
```

最后，我们将定义一个操作来检索给定批量输入所预测的 softmax 概率。这个预测值将反过来用于计算神经网络的准确性：

```
    # 定义预测
    tf_predictions = tf.nn.softmax(inference(tf_inputs))
```

3.11.3　运行神经网络

现在，我们实现了运行神经网络所需要的所有基本操作，可以对它进行检查，看它是否有能力学习对图像中的数字进行正确分类：

```
    for epoch in range(NUM_EPOCHS):
        train_loss = []
```

```
        # 训练阶段
        for step in range(train_inputs.shape[0]//batch_size):
            # Creating one-hot encoded labels with labels
            # One-hot encoding dight 3 for 10-class MNIST data set will result in
            # [0,0,0,1,0,0,0,0,0,0]
            labels_one_hot = np.zeros((batch_size, num_labels),dtype=np.float32)
            labels_one_hot[np.arange(batch_size),train_labels[step*batch_size:
(step+1)*batch_size]] = 1.0

            # 运行优化程序
            loss, _ = session.run([tf_loss,tf_loss_minimize],feed_dict={ tf_inputs:
train_inputs[step*batch_size: (step+1)*batch_size,:], tf_labels:
labels_one_hot} )
            train_loss.append(loss)

         test_accuracy = []
        # 测试阶段
        for step in range(test_inputs.shape[0]//batch_size):
            test_predictions = session.run(tf_predictions,feed_dict={tf_inputs:
test_inputs[step*batch_size: (step+1)*batch_size,:]})
            batch_test_accuracy = accuracy(test_predictions,
    test_labels[step*batch_size:(step+1)*batch_size])
    test_accuracy.append(batch_test_accuracy)

        print('Average train loss for the %d
epoch: %.3f\n'%(epoch+1,np.mean(train_loss)))
        train_loss_over_time.append(np.mean(train_loss))
        print('\tAverage test accuracy for the %d
epoch: %.2f\n'%(epoch+1,np.mean(test_accuracy)*100.0))
        test_accuracy_over_time.append(np.mean(test_accuracy)*100)
```

在这段代码中，accuracy(test_prediction, test_tags)是一个函数，它接受一些预测和标签作为输入，并提供准确性（有多少预测与实际标签匹配）。最终，我们会得到类似于图 3-17 所示的示意图，从运行的结果来看，该模型表现良好，读者也可以自行运行代码进行查验。

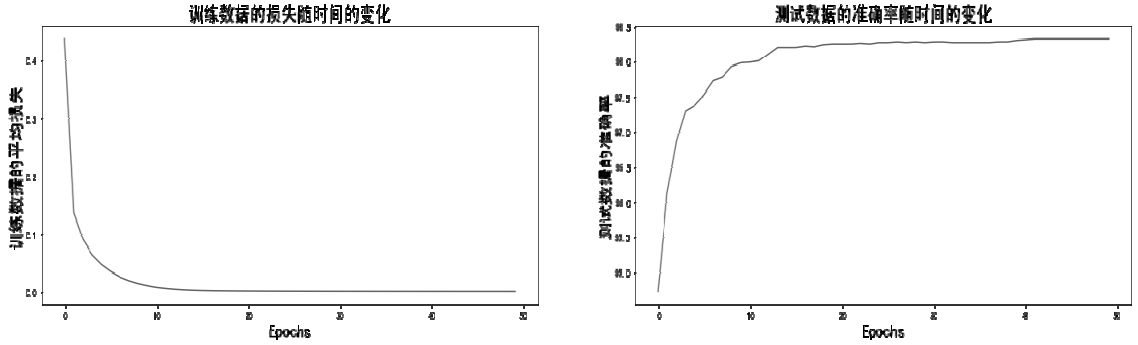

图 3-17　MNIST 数字分类任务的训练损失和测试准确性示意图

3.12　总结

在本章中，我们通过一些示例对算法的主要底层平台（TensorFlow）有了大体上的认知，从而迈出了解决 NLP 任务的第一步。

首先，我们讨论了 TensorFlow 的概念、主要特征及其安装情况。谷歌推出的 TensorFlow 是一个采用数据流图、用于数值计算的开源软件库。它有一些主要特征，例如自动求微分、多语言支持、高度的灵活性、可移植性、性能最优化等。对于 TensorFlow 的安装，我们重点介绍了 Linux 下的安装情况。

其次，我们对于 TensorFlow 的三个主要组成部分（计算图、张量和会话）进行了详细解读。概括起来讲，我们可以用张量表示数据，用计算图搭建神经网络，用会话执行计算图，再优化计算图中线上的权重值（参数）后得到模型。

然后，我们对于 TensorFlow 的工作原理进行了深度解读，了解到 TensorFlow 是一个“client—master—worker”分布式的架构系统。

客户端借助于会话界面可以和 master 进行交互，把将要触发执行的请求发送给 master，而 master 则会把全部要执行的任务分配给单个或多个 worker，对应的结果通过 master 再返回给客户端。作为专注于执行计算的 worker，任何一个 worker 进程都在管理并使用着整个计算硬件资源，进而采取最优的工作方式来计算子图。

下面我们通过一个 sigmoid 示例对 TensorFlow 进行更深一层的解读，并且在后面结合该示例对 TensorFlow 的客户端做了专门的深度解读。

接下来，我们详细讨论了构成一个典型 TensorFlow 客户端的常见元素：输入、变量、输出和运算（或操作）。输入是我们提供给算法的数据，目的是用于模型的训练和测试。我们讨论了三种不同的输入方式：使用占位符、预加载数据并将数据存储为 TensorFlow 张量以及使用输入管道。然后我们讨论了 TensorFlow 变量，它们与其他张量的区别，以及如何创建和初始化变量。之后，我们讨论了如何使用变量来创建中间和最终的输出。最后，我们讨论了几个可用的 TensorFlow 运算或操作，例如数学运算、矩阵运算、神经网络相关的运算和控制流的操作，这些运算和操作将在

本书后面使用。在对这些常见元素解读的过程中，我们逐步分析并结合代码加以实现，力求让每一位读者都能够更直观地理解其内在逻辑和实现机制。

我们还讨论了在实现 TensorFlow 客户端时如何使用变量作用域来避免某些缺陷。作用域允许我们轻松使用变量，同时也能保持代码的封装性。

最后，我们使用之前学习的所有概念实现了一个神经网络，我们使用三层神经网络对 MNIST 数字数据集进行分类。

下一章，我们将正式开始讲解 NLP 领域的重要模型：词嵌入（Word Embedding）。