

李金洪◎编著

赠送超值学习资料

- 12套同步配套教学视频
- 113套源代码文件（带配套样本）

深度学习之 TensorFlow

入门、原理与进阶实战

Getting Started and Best Practices with TensorFlow for Deep Learning

一线研发工程师以14年开发经验的视角全面解析TensorFlow应用
涵盖数值、语音、语义、图像等多个领域的96个深度学习应用实战案例

李大学

磁云科技创始人/京东终身荣誉技术顾问

李建军

创客总部/创客共赢基金合伙人

共同
推荐



机械工业出版社
China Machine Press

深度学习之TensorFlow：入门、原理与进阶实战

李金洪 编著

ISBN: 978-7-111-59005-7

本书纸版由机械工业出版社于2018年出版，电子版由华章分社（北京华章图文信息有限公司，北京奥维博世图书发行有限公司）全球范围内制作与发行。

版权所有，侵权必究

客服热线：+ 86-10-68995265

客服信箱：service@bbbvip.com

官方网址：www.hzmedia.com.cn

新浪微博 @华章数媒

微信公众号 华章电子书（微信号：hzebook）

*****ebook converter DEMO Watermarks*****

目录

配套学习资源

前言

第1篇 深度学习与TensorFlow基础

 第1章 快速了解人工智能与TensorFlow

 1.1 什么是深度学习

 1.2 TensorFlow是做什么的

 1.3 TensorFlow的特点

 1.4 其他深度学习框架特点及介绍

 1.5 如何通过本书学好深度学习

 第2章 搭建开发环境

 2.1 下载及安装Anaconda开发工具

 2.2 在Windows平台下载及安装TensorFlow

 2.3 GPU版本的安装方法

 2.4 熟悉Anaconda 3开发工具

第3章 TensorFlow基本开发步骤——以逻辑回归拟合二维数据为例

 3.1 实例1：从一组看似混乱的数据中找出 $y \approx 2x$ 的规律

 3.2 模型是如何训练出来的

 3.3 了解TensorFlow开发的基本步骤

第4章 TensorFlow编程基础

 4.1 编程模型

 4.2 TensorFlow基础类型定义及操作函数介绍

 4.3 共享变量

 4.4 实例19：图的基本操作

4.5 配置分布式TensorFlow

4.6 动态图（Eager）

4.7 数据集（tf.data）

第5章 识别图中模糊的手写数字（实例21）

5.1 导入图片数据集

5.2 分析图片的特点，定义变量

5.3 构建模型

5.4 训练模型并输出中间状态参数

5.5 测试模型

5.6 保存模型

5.7 读取模型

第2篇 深度学习基础——神经网络

第6章 单个神经元

6.1 神经元的拟合原理

6.2 激活函数——加入非线性因素，解决线性模型缺陷

6.3 softmax算法——处理分类问题

6.4 损失函数——用真实值与预测值的距离来指导模型的收敛方向

6.5 softmax算法与损失函数的综合应用

6.6 梯度下降——让模型逼近最小偏差

6.7 初始化学习参数

6.8 单个神经元的扩展——Maxout网络

6.9 练习题

第7章 多层神经网络——解决非线性问题

7.1 线性问题与非线性问题

7.2 使用隐藏层解决非线性问题

7.3 实例31：利用全连接网络将图片进行分

类

7.4 全连接网络训练中的优化技巧

7.5 练习题

第8章 卷积神经网络——解决参数太多问题

8.1 全连接网络的局限性

8.2 理解卷积神经网络

8.3 网络结构

8.4 卷积神经网络的相关函数

8.5 使用卷积神经网络对图片分类

8.6 反卷积神经网络

8.7 实例50：用反卷积技术复原卷积网络各层图像

8.8 善用函数封装库

8.9 深度学习的模型训练技巧

第9章 循环神经网络——具有记忆功能的网络

9.1 了解RNN的工作原理

9.2 简单RNN

9.3 循环神经网络（RNN）的改进

9.4 TensorFlow实战RNN

9.5 实例68：利用BiRNN实现语音识别

9.6 实例69：利用RNN训练语言模型

9.7 语言模型的系统学习

9.8 处理Seq2Seq任务

9.9 实例75：制作一个简单的聊天机器人

9.10 时间序列的高级接口TFTS

第10章 自编码网络——能够自学习样本特征的网络

- 10.1 自编码网络介绍及应用
- 10.2 最简单的自编码网络
- 10.3 自编码网络的代码实现
- 10.4 去噪自编码
- 10.5 去噪自编码网络的代码实现
- 10.6 栈式自编码
- 10.7 深度学习中自编码的常用方法
- 10.8 去噪自编码与栈式自编码的综合实现
- 10.9 变分自编码
- 10.10 条件变分自编码

第3篇 深度学习进阶

- 第11章 深度神经网络
 - 11.1 深度神经网络介绍
 - 11.2 GoogLeNet模型介绍
 - 11.3 残差网络（ResNet）
 - 11.4 Inception-ResNet-v2结构
 - 11.5 TensorFlow中的图片分类模型库——slim
 - 11.6 使用slim中的深度网络模型进行图像的识别与检测
 - 11.7 实物检测模型库——Object Detection API
 - 11.8 实物检测领域的相关模型
 - 11.9 机器自己设计的模型（NASNet）
- 第12章 对抗神经网络（GAN）
 - 12.1 GAN的理论知识
 - 12.2 DCGAN——基于深度卷积的GAN
 - 12.3 InfoGAN和ACGAN：指定类别生成模

拟样本的GAN

12.4 AEGAN：基于自编码器的GAN

12.5 WGAN-GP：更容易训练的GAN

12.6 LSGAN（最小乘二GAN）：具有
WGAN同样效果的GAN

12.7 GAN-cls：具有匹配感知的判别器

12.8 SRGAN——适用于超分辨率重建的
GAN

12.9 GAN网络的高级接口TFGAN

12.10 总结

配套学习资源

本书提供了配套的超值学习资料，下面分别介绍。

1. 同步配套教学视频

作者按照图书的内容和结构，录制了同步对应的《深度学习之TensorFlow：入门、原理与进阶实战》系列教学视频，如图1所示。



图1 《深度学习之TensorFlow——入门、原理与进阶实战》系列教学视频

2. 书中的实例源文件

本书提供了书中涉及的所有实例源文件，共计123段代码，如图2所示。读者可以一边阅读本书，一边参照源文件动手练习，这样不仅提高了学习效率，而且可以对书中的内容有更加直观的认识，从而逐渐培养自己的编码能力。

*****ebook converter DEMO Watermarks*****

3-1 线性回归.py	3-2 字典.py	3-3 无占位符.py	3-4 字典2.py
4-1 sessionhello.py	4-2 withsession.py	4-3 withsessionfeed.py	4-4 线性回归模型保存及载入.py
4-5 模型内容.py	4-6 保存检查点.py	4-7 trainmonitored.py	4-8 线性回归的tensorboard可视化.py
4-9 get_variable和Variable的区别.py	4-10 get_variable配合variable_scope....	4-11 get_variable配合variable_scope2...	4-12 共享变量的作用域与初始化.py
4-13 作用域与操作符的受限范围.py	4-14 图的基本操作.py	4-15 ps.py	4-16 worker.py
4-17 worker2.py	5-1 mnist数据集.py	5-2 mnist分类.py	6-1 softmax应用.py
6-2 sparesoftmaxwithminist.py	6-3 退化学习率.py	6-4 Maxout网络实现mnist分类.py	7-1 线性逻辑回归.py
7-2 线性多分类.py	7-3 异或.py	7-4 异或one_hot.py	7-5 mnist多层分类.py
7-6 异或集的过拟合.py	7-7 异或集的L2_loss.py	7-8 异或集dropout.py	7-9 xorerr1.py
7-10 xorerr2.py	7-11 xorerr3.py	8-1 卷积函数使用.py	8-2 sobel.py
8-3 池化函数使用.py	8-4 cifar下载.py	8-5 cifar.py	8-6 cifar手动读取.py
8-7 queue.py	8-8 cifar队列协调器.py	8-9 cifar卷积.py	8-10 MNIST卷积.py
8-11 Cifar全连接卷积.py	8-12 反卷积操作.py	8-13 反池化操作.py	8-14 gradients0.py
8-15 gradients1.py	8-16 gradients2.py	8-17 cifar反卷积.py	8-18 cifar简洁代码.py
8-19 cifar卷积核优化.py	8-20 cifar多通道卷积.py	8-21 cifarBN.py	8-22 带BN的多通道mnist.py
8-23 多通道mnist.py	9-1 subtraction.py	9-2 echo模拟.py	9-3 LSTMmnist.py
9-4 LSTMCell.py	9-5 gru.py	9-6 创建动态RNN.py	9-7 McellMNIST.py
9-8 mcellLSTMGRU.py	9-9 动态多层.py	9-10 LSTM改错.py	9-11 lstm改错1.py
9-12 GRU改错2.py	9-13 LSTM改错3.py	9-14 BiRNNMnist.py	9-15 单层静态双向rnn.py
9-16 多层双向rnn.py	9-17 list多层双向rnn.py	9-18 Multi双向rnn.py	9-19 动态Multi双向rnn.py
9-20 InGRUonMnist.py	9-21 Ln多GRU1-1.py	9-22 Ln多GRU1-2.py	9-23 yuyincha1.py
9-24 yuyinutils.py	9-25 rnntwordtest.py	9-26 word2vect.py	9-27 word2vect自定义候选采样.py
9-28 word2vect -2.py	9-29 word2vect学习样本候选采样.py	9-30 基本seq2seq.py	9-31 STOCKDATA.py
9-32 seq2seqstock.py	9-33 datautil.py	9-34 seq2seq_model.py	9-35 train.py
9-36 test.py	9-37 datautil.py	9-38 seq2seq_model.py	9-39 train.py
9-40 test.py	10-1 自编码.py	10-2 自编码进阶.py	10-3 卷积网络自编码.py
10-4 自编码练习题.py	10-5 去噪自编码.py	10-6 自编码综合.py	10-7 分布自编码综合.py
10-8 变分自编码器.py	10-9 条件变分自编码器.py	11-1 tfrecordtest.py	11-2 inception_resnet_v2使用.py
11-3 vgg19图片检测使用.py	11-4 Object Detection使用.py	12-1 Mnistinfogan.py	12-2 aegan.py
12-3 wgan_gp.py	12-4 mnistLsgan.py	12-5 GAN-cls.py	12-6 mnistEspcn.py
12-7 tfrecoderSRESPCN.py	12-8 resESPCN.py	12-9 rsgan.py	

图2 本书实例源文件

3. 书中实例用到的素材和样本

本书提供了书中实例用到的全部素材和样本。读者可以采用这些素材和样本，完全再现书中的实例效果。

11.5.3 代码参考	实例38 素材	实例41 素材
实例69 素材	实例70 素材	实例73 素材
实例74 素材	实例75 素材	实例85 素材
实例86 素材	素材说明.txt	

图3 本书实例用到的素材和样本

4. 配套学习资源获取方式

*****ebook converter DEMO Watermarks*****

本书提供的配套学习资源需要读者自行下载。有以下两种途径：

(1) 登录机械工业出版社华章公司的网站 www.hzbook.com，然后搜索到本书页面，找到下载模块下载即可。

(2) 扫描图4所示的二维码，关注并访问微信公众号xiangyuejqiren，在公众号中回复“深1”得到相关资源的下载链接。



图4 微信公众号xiangyuejqiren二维码

前言

最近，人工智能话题热度不减，IT领域甚至言必称之。

从人工智能的技术突破看，在语音和图像识别等方面，在特定领域和特定类别下，计算机的处理能力已经接近甚至超过人类。此外，人工智能在人们传统认为很难由机器取得成功的认知领域也有所突破。

我国目前在人工智能技术研究方面已经走在了世界前列，人工智能应用领域已经非常宽广，涵盖了从智能机器人到智能医疗、智能安防、智能家居和智慧城市，再到语音识别、手势控制和自动驾驶等领域。

百度CEO李彦宏判断：人工智能是一个非常大的产业，会持续很长时间，在未来的20年到50年间都会是快速发展的。

人工智能“火”起来主要有3个原因：互联网大量的数据、强大的运算能力、深度学习的突破。其中，深度学习是机器学习方法之一，是让计算机从周围世界或某个特定方面的范例中进行学习从而变得更加智能的一种方式。

面对人工智能如火如荼的发展趋势，IT领域也掀起了一波深度学习热潮，但是其海量的应用数学术语和公式，将不少爱好者拒之门外。本书由浅入深地讲解了深度学习的知识体系，将专业性较强的公式和理论转化成通俗易懂的简单逻辑描述语言，帮助非数学专业的爱好者搭上人工智能的“列车”。

本书特色

1. 配教学视频

为了让读者更好地学习本书内容，作者对每一章内容都录制了教学视频。借助这些视频，读者可以更轻松地学习。

2. 大量的典型应用实例，实战性强，有较高的应用价值

本书提供了96个深度学习相关的网络模型实例，将原理的讲解最终都落实到了代码实现上。而且这些实例会随着图书内容的推进，不断趋近于工程化的项目，具有很高的应用价值和参考性。

3. 完整的源代码和训练数据集

书中所有的代码都提供了免费下载途径，使

读者学习更方便。另外，读者可以方便地获得书中案例的训练数据集。如果数据集是来源于网站，则提供了有效的下载链接；如果是作者制作的，在随书资源中可直接找到。

4. 由浅入深、循序渐进的知识体系，通俗易懂的语言

本书按照读者的接受度搭建知识体系，由浅入深、循序渐进，并尽最大可能地将学术语言转化为容易让读者理解的语言。

5. 拒绝生僻公式和符号，落地性强

在文字表达上，本书也尽量使用计算机语言编写的代码来表述对应的数学公式，这样即使不习惯用数学公式的读者，也能够容易地理解。

6. 内容全面，应用性强

本书提供了从单个神经元到对抗神经网络，从有监督学习到半监督学习，从简单的数据分类到语音、语言、图像分类乃至样本生成等一系列前沿技术，具有超强的实用性，读者可以随时查阅和参考。

7. 大量宝贵经验的分享

授之以鱼不如授之以渔。本书在讲解知识点

的时候，更注重方法与经验的传递。全书共有几十个“注意”标签，其中内容都是“含金量”很高的成功经验分享与易错事项总结，有关于理论理解的，有关于操作细节的。这些内容可以帮助读者在学习的路途上披荆斩棘，快速融会贯通。

本书内容

第1篇 深度学习与TensorFlow基础（第1~5章）

第1章快速了解人工智能与TensorFlow，主要介绍了以下内容：

- (1) 人工智能、深度学习、神经网络三者之间的关系，TensorFlow软件与深度学习之间的关系及其特点；
- (2) 其他主流深度学习框架的特点；
- (3) 一些关于如何学习深度学习和使用本书的建议。

第2章搭建开发环境，介绍了如何搭建TensorFlow开发环境。具体包括：

- (1) TensorFlow的下载及在不同平台上的安装方法；

(2) TensorFlow开发工具（本书用的是Anaconda开发工具）的下载、安装和使用。

如要安装GPU版的TensorFlow，书中也详细介绍了如何安装CUDA驱动来支持GPU运算。

第3章TensorFlow基本开发步骤——以逻辑回归拟合二维数据为例，首先是一个案例，有一组数据，通过TensorFlow搭配模型并训练模型，让模型找出其中 $y \approx 2x$ 的规律。在这个案例的基础上，引出了在神经网络中“模型”的概念，并介绍了TensorFlow开发一个模型的基本步骤。

第4章TensorFlow编程基础，主要介绍了TensorFlow框架中编程的基础知识。具体包括：

- (1) 编程模型的系统介绍；
- (2) TensorFlow基础类型及操作函数；
- (3) 共享变量的作用及用法；
- (4) 与“图”相关的一些基本操作；
- (5) 分布式配置TensorFlow的方法。

第5章识别图中模糊的手写数字（实例21），是一个完整的图像识别实例，使用TensorFlow构建并训练了一个简单的神经网络模

型，该模型能识别出图片中模糊的手写数字5、0、4、1。通过这个实例，读者一方面可以巩固第4章所学的TensorFlow编程基础知识，另一方面也对神经网络有一个大体的了解，并掌握最简单的图像识别方法。

第2篇 深度学习基础——神经网络中（第6 ~10章）

第6章单个神经元，介绍了神经网络中最基础的单元。首先讲解了神经元的拟合原理，然后分别介绍了模型优化所需的一些关键技术：

- 激活函数——加入非线性因素，解决线性模型缺陷；
- softmax算法——处理分类问题；
- 损失函数——用真实值与预测值的距离来指导模型的收敛方向；
- 梯度下降——让模型逼近最小偏差；
- 初始化学习参数。

最后还介绍了在单个神经元基础上扩展的网络——Maxout。

第7章多层神经网络——解决非线性问题，

先通过两个例子（分辨良性与恶性肿瘤、将数据按颜色分为3类）来说明线性问题，进而引出非线性问题。然后介绍了如何使用多个神经元组成的全连接网络进行非线性问题的分类。最后介绍了全连接网络在训练中常用的优化技巧：正则化、增大数据集和Dropout等。

第8章卷积神经网络——解决参数太多问题，通过分析全连接网络的局限性，引出卷积神经网络。首先分别介绍了卷积神经网络的结构和函数，并通过一个综合的图片分类实例介绍了卷积神经网络的应用。接着介绍了反卷积神经网络的原理，并通过多个实例介绍了反卷积神经网络的应用。最后通过多个实例介绍了深度学习中模型训练的一些技巧。

第9章循环神经网络——具有记忆功能的网络，本章先解释了人脑记忆，从而引出了机器学习中具有类似功能的循环神经网络，介绍了循环神经网络（RNN）的工作原理，并通过实例介绍了简单RNN的一些应用。接着介绍了RNN的一些改进技术，如LSTM、GRU和BiRNN等，并通过大量的实例，介绍了如何通过TensorFlow实现RNN的应用。从9.5节起，用了大量的篇幅介绍RNN在语音识别和语言处理方面的应用，先介绍几个案例——利用BiRNN实现语音识别、利用RNN训练语言模型及语言模型的系统学习等，然后将前面的内容整合成一个功能更完整的机器

人，它可以实现中英文翻译和聊天功能。读者还可以再扩展该机器人的功能，如实现对对联、讲故事、生成文章摘要等功能。

第10章自编码网络——能够自学习样本特征的网络，首先从一个最简单的自编码网络讲起，介绍其网络结构和具体的代码实现。然后分别介绍了去噪自编码、栈式自编码、变分自编码和条件变分自编码等网络结构，并且在讲解每一种结构时都配有对应的实例。

第3篇 深度学习进阶（第11、12章）

第11章深度神经网络，从深度神经网络的起源开始，逐步讲解了深度神经网络的历史发展过程和一些经典模型，并分别详细介绍了这些经典模型的特点及内部原理。接着详细介绍了使用slim图片分类模型库进行图像识别和图像检测的两个实例。最后介绍了实物检测领域的其他一些相关模型。

第12章对抗神经网络，从对抗神经网络(GAN)的理论开始，分别介绍了DCGAN、AEGAN、InfoGAN、ACGAN、WGAN、LSGAN和SRGAN等多种GAN的模型及应用，并通过实例演示了生成指定模拟样本和超分辨率重建的过程。

本书读者对象

- 深度学习初学者；
- 人工智能初学者；
- 深度学习爱好者；
- 人工智能工程师；
- TensorFlow初级开发人员；
- 需要提高动手能力的深度学习技术人员；
- 名大院校的相关学生。

关于作者

本书由李金洪主笔编写。其他参与本书编写的人员还有马峰、孙朝晖、郑一友、王其景、张弨、白林、彭咏文、宋文利。

另外，吴宏伟先生也参与了本书后期的编写工作，为本书做了大量的细节调整。因为有了他的逐字推敲和一丝不苟，才使得本书行文更加通畅和通俗易懂。在此表示深深的感谢！

虽然我们对书中所述内容都尽量核实，并多

次进行了文字校对，但因时间所限，加之水平所限，书中疏漏和错误在所难免，敬请广大读者批评指正。联系我们可以加入本书讨论QQ群40016981，也可发E-mail到hzbook2017@163.com。

第1篇 深度学习与TensorFlow基础

本篇将介绍人工智能与TensorFlow的基本概念、如何搭建TensorFlow的开发环境、TensorFlow的基本开发步骤、TensorFlow编程基础，并通过一个识别图中模糊手写数字的实例，使读者巩固TensorFlow的编程基础知识，并对神经网络有个大体的了解，为后面的学习打好基础。

第1章 快速了解人工智能与TensorFlow

第2章 搭建开发环境

第3章 TensorFlow基本开发步骤——以逻辑回归拟合二维数据为例

第4章 TensorFlow编程基础

第5章 识别图中模糊的手写数字（实例21）

第1章 快速了解人工智能与TensorFlow

本章是一个相对比较轻松的开篇，这里不会介绍太深的知识，而是普及一下什么是TensorFlow，什么是深度学习，深度学习与TensorFlow的关系，以及当今都有哪些与TensorFlow同级的开源框架，它们之间都是什么关系，各有什么特点和阅读本书的建议。本章的内容，就好比通往深度学习领域的大门。快来打开它，开始你的TensorFlow学习之旅吧。

本章含有教学视频共17分钟。

作者按照本章的内容结构，对主要内容进行了快速讲解，包括深度学习与人工智能的关系、TensorFlow与其他深度学习框架的优劣特性比较，以及如何利用本书学好深度学习这门学科（重点是要用对深度学习的热情火焰，烧出足以融化沙漠的温度，将这本入门书籍“化为灰烬”）。



*****ebook converter DEMO Watermarks*****

1.1 什么是深度学习

提到人工智能，人们往往回想到深度学习，然而，深度学习不像人工智能那样容易从字面上理解。这是因为深度学习是从内部机理来阐述的，而人工智能是从其应用的角度来阐述的，即深度学习是实现人工智能的一种方法。

人工智能领域，起初是进行神经网络的研究。但神经网络发展到一定阶段后，模型越来越庞大，结构也越来越复杂，于是人们将其命名为“深度学习”。可以这样理解——深度学习属于后神经网络时代。

深度学习近年来的发展突飞猛进，越来越多的人工智能应用得以实现。其本质为一个可以模拟人脑进行分析、学习的神经网络，它模仿人脑的机制来解释数据（如图像、声音和文本），通过组合低层特征，形成更加抽象的高层特征或属性类别，来拟合人们日常生活中的各种事情。

深度学习被广泛用于与人们生活息息相关的各种领域，可以实现机器翻译、人脸识别、语音识别、信号恢复、商业推荐、金融分析、医疗辅助和智能交通等。

在国内乃至世界，越来越多的资金涌向人工

智能领域，人工智能领域新成立的创业公司每年呈递增趋势，越来越多的学校也开始开设与深度学习相关的课程。这个时代，正像是移动互联网的前夜。如果你也感觉到了，那么现在正是时候，一起加入进来，通过系统的学习，将自己打造成为一名深度学习的专业人才吧。

1.2 TensorFlow是做什么的

TensorFlow是Google开源的第二代用于数字计算的软件库。起初，它是Google大脑团队为了研究机器学习和深度神经网络而开发的，但后来发现这个系统足够通用，能够支持更加广泛的应用，就将其开源贡献了出来。

概括地说，TensorFlow可以理解为一个深度学习框架，里面有完整的数据流向与处理机制，同时还封装了大量高效可用的算法及神经网络搭建方面的函数，可以在此基础之上进行深度学习的开发与研究。本书是基于TensorFlow来进行深度学习研究的。

TensorFlow是当今深度学习领域中最火的框架之一。在GitHub上，TensorFlow的受欢迎程度目前排名第一（如图1-1所示），以3倍左右的数量遥遥领先于第二名。

Top Deep Learning Projects

A list of popular github projects related to deep learning (ranked by stars automatically).

Please update list.txt (via pull requests)

Project Name	Stars	Description
tensorflow	68684	Computation using data flow graphs for scalable machine learning
caffe	19958	Caffe: a fast open framework for deep learning.
keras	19190	Deep Learning library for Python. Runs on TensorFlow, Theano, or CNTK.
neural-style	14432	Torch implementation of neural style algorithm
CNTK	12240	Microsoft Cognitive Toolkit (CNTK), an open source deep-learning toolkit
incubator-mxnet	10944	Lightweight, Portable, Flexible Distributed/Mobile Deep Learning with Dynamic, Mutation-aware Dataflow Dep Scheduler; for Python, R, Julia, Scala, Go, Javascript and more
deepdream	10496	
data-science-ipython-notebooks	10021	Data science Python notebooks: Deep learning (TensorFlow, Theano, Caffe, Keras), scikit-learn, Kaggle, big data (Spark, Hadoop MapReduce, HDFS), matplotlib, pandas, NumPy, SciPy, Python essentials, AWS, and various command lines.

图1-1 GitHub上TensorFlow受欢迎程度排名第一

图1-1来源于地址<https://github.com/hunkim/DeepLearningStars>。

选择TensorFlow进行学习的优势是，在深度学习道路上不会孤单，会有大于同等框架几倍的资料可供学习，以及更多的爱好者可以相互学习、交流。更重要的是，目前越来越多的学术论文都更加倾向于在TensorFlow上开发自己的示例原型。这一得天独厚的优势，可以让学习者在同步当今最新技术的过程中，省去不少时间。

1.3 TensorFlow的特点

TensorFlow是用C++语言开发的，支持C、Java、Python等多种语言的调用，目前主流的方式通常会使用Python语言来驱动应用。这一特点也是其能够广受欢迎的原因。利用C++语言开发可以保证其运行效率，Python作为上层应用语言，可以为研究人员节省大量的开发时间。

TensorFlow相对于其他框架有如下特点。

1. 灵活

TensorFlow与CNTK、MXNET、Theano同属于符号计算构架，允许用户在不需要使用低级语言（如在Caffe中）实现的情况下，开发出新的复杂层类型。基于图运算是其基本特点，通过图上的节点变量可以控制训练中各个环节的变量，尤其在需要对底层操作时，TensorFlow要比其他框架更容易。当然它也有缺点，灵活的操作会增加使用复杂度，从而在一定程度上增加了学习成本。

2. 便捷、通用

作为主流的框架，TensorFlow生成的模型，具有便捷、通用的特点，可以满足更多使用者的

需求。TensorFlow可以适用于Mac、Linux、Windows系统上开发。其编译好的模型几乎适用于当今所有的平台系统，并能满足“开箱即用”的模型使用理念，使模型应用起来更简单。

3. 成熟

由于TensorFlow被使用的情况最多，所以其框架的成熟度绝对是第一的。在Google的白皮书上写道，Google内部有大量的产品几乎都用到了TensorFlow，如搜索排序、语音识别、谷歌相册和自然语言处理等。有这么多在该框架上的成功案例，先不说能够提供多少经验技巧，至少可以确保学习者在研究的道路上，遇到挫折时不会怀疑是框架的问题。

4. 超强的运算性能

虽然TensorFlow在大型计算机集群的并行处理中，运算性能仅略低于CNTK，但是，其在个人机器使用场景下，会根据机器的配置自动选择CPU或GPU来运算，这方面做得更加友好与智能化。

1.4 其他深度学习框架特点及介绍

下面再来了解一下深度学习领域中的其他常见框架。

- Theano：是一个十余年的Python深度学习和机器学习框架，用来定义、优化和模拟数学表达式计算，用于高效地解决多维数组的计算问题，有较好的扩展性。
- Torch：同样具有很好的扩展性，但某些接口不够全面，如WGAN-GP这样的网络需要手动来修改梯度就没有对应的接口。其最大的缺点是，需要LuaJIT的支持，用于Lua语言，在Python为王的今天，通用性方面显得较差。
- Keras：可以理解为一个Theano框架与TensorFlow前端的一个组合。其构建模型的API调用方式逐渐成为主流，包括TensorFlow、CNTK、MXNet等知名框架，都提供对Keras调用语法的支持。可以说，使用Keras编写的代码，会有更好的可移植性。
- DeepLearning4j：是基于Java和Scala语言开发的，应用在Hadoop和Spark系统之上的深度学习软件。

· Caffe：当年深度学习的老大。最初是一个强大的图像分类框架，是最容易测试评估性能的标准深度学习框架，并且提供很多预训练模型，尤其该模型的复用价值在其他框架的学习中都会出现，大大提升了现有模型的训练时间。但是现在的Caffe似乎停滞不前，没有更新。尽管Caffe又重新掘起，从架构上看更像是TensorFlow，而且与原来的Caffe也不在一个工程里，可以独立成一个框架来看待，与原Caffe关系不大。但仍不建议使用。

· MXNet：是一个可移植的、可伸缩的深度学习库，具有Torch、Theano、Chainer和Caffe的部分特性。不同程度的支持Python、R、Scala、Julia和C++语言，也是目前比较热门的主流框架之一。

· CNTK：是一个微软开发的深度学习软件包，以速度快著称，有其独有的神经网络配置语言Brain Script，大大降低了学习门槛。有微软作为后盾，CNTK成为了最具有潜力与Tensor Flow争夺天下的框架。但目前其成熟度要比Tensor Flow差太多，即便是发行的版本也会有大大小小的bug。与其他框架一样，CNTK具有文档资料不足的特点。但其与Visual Studio的天生耦合，以及其特定的MS编程风格，使得熟悉Visual Studio工具的小伙伴们从代码角度极易上手。另外，CNTK目前还不支持Mac操作系统。

1.5 如何通过本书学好深度学习

从小老师就教导我们，做事情要讲究方法，一个好的学习方法能带给你事半功倍的效果。对于深度学习也一样，如果之前是因为没有一本系统的教材，让你对深度学习毫无头绪的话，那么现在机会来了。通过本书的指引，你将会通过实例由浅入深逐步上手，直到最终掌握深度学习的相关知识。下面就来说下如何通过本书来学习深度学习。

1.5.1 深度学习怎么学

这个问题完全是主观回答，因为不同的人有不同的领悟。所以笔者也只能聊聊自己对学习深度学习方法的理解。

举个例子，在笔者的家乡有练武术的习惯，平时有人找老师傅学拳时，一般老师傅都会先了解他学拳的目的是什么，然后再根据他的目的来选择需要教哪些内容。

- 对于只为了打架能赢的人，老师傅会先以力量和重拳的训练开始，中间穿插点对抗，一般1个月左右对付2、3个普通人没什么问题。

- 对于想集训打比赛的人，老师傅会以体

能、力量、抗击打等身体素质训练为主，配合大量的对抗练习刺激反应，起码上场要保证能够打完全程。

· 对于爱好武学想系统学习的人，则需要从步伐、拳、腿一点一点练习。然后再加上摔法，对抗之类的技巧，同时配合阵图、战机等理论。

· 笔者觉得用这个例子来类比深度学习非常恰当。

· 假如你手里有短期任务，想快速用深度学习解决某一个功能，那么就针对该领域找现成的例子，扫清例子中的盲点，快速熟悉并修改、使用。

· 假如想近期提升一下自己，应对跳槽，挑战工作等，那就需要将主干知识点记住并能说出来，然后亲自演练每个领域的例子，保证自己知道其原理。

· 如果想在这条路上一直走下去，而眼前并没有紧急要应对的事情，那么可以一步一步地学，通过“努力+时间”的积累，得到的才是功夫。

如同学拳一样，拳击训练是必不可少的，出过百万次拳的水平跟出过1万次拳的水平绝对是不一样的。同理，编写代码也是必不可少的。有过*****ebook converter DEMO Watermarks*****

百万行代码编写经验的水平也远远胜过1万行代码编写经验的水平。时间在你努力的期间起到催化剂的作用，在空余时间多去思考，多尝试用自己的思维和角度去理解你所接触的相对生僻的事情，这个习惯不仅会使你学习深度学习变得容易，还会使你对它越学越有兴趣，而且这个习惯也适用于其他领域。

1.5.2 如何学习本书

前面的道理懂了之后，我们就来看看如何学习本书。针对与前面讲述的3个场景，可以在本书中依次找到对应关系。书中的每一节都由理论+实例的结构组成。针对三种场景可以有如下策略。

1. 短期任务

快速定位你手中的任务所需要的知识点，依次在本书中找到最匹配的例子，按照步骤一步一步实现它。细节原理可以先不去管，主要把数据源即数据流向和知识结构弄清楚即可。按照例子做完之后，相信你会有个大概的感觉，然后再应用里面的知识着手去做自己的任务。

2. 应对挑战突击

这个策略需要将书中的文字理论部分快速读

完，并且理解、记住。对于实例代码，可以大致过一遍，但需要注意的提示内容必须要看，并且记住，这些提示内容会使你给人留下一种很有经验的印象。

3. 踏实学习

按照本书的章节一步一步地学习，该学理论学理论，该做配套的例子做例子。因为本书的知识结构并不是按照知识面的属性排列的，而是考虑到读者的接受程度排列的，例如对属于第3章的某个知识点，考虑到刚学习的读者接受起来会很费劲，而且短时期用不到这个知识，那么就将其移到第5章，需要用到这个知识点时再介绍。假设读者是从第1章学过来的，那么学到第5章时，对于这个知识点已经可以很轻松地理解了。

另外，本书尽可能地不用学术术语及公式来描述理论，但由于无法预知读者在学习此书时的知识基础与接受程度，难免在阅读时会遇到没有接触过的生僻术语及理论，此时可以自己多上网查阅相关资料，或给笔者发邮件，只要有时间笔者都会认真回复。

第2章 搭建开发环境

本章将进入本书的入门阶段，先从环境的搭建开始。虽然TensorFlow支持CPU运行，但是也会有一些实例只能在GPU上运行。所以很有必要在学习本书之前购买一个带有GPU显卡的计算机。

本书使用的是Python 3.5开发环境，开发工具使用Anaconda，操作系统使用Windows 10。TensorFlow的学习中与操作系统无关，读者可以使用Linux或Mac，也可以使用其他操作系统。如果读者对安装过程已经掌握了，可以跳过本章。

本章含有教学视频共6分52秒。

作者按照本章的内容结构，对主要内容进行了快速讲解，包括关于TensorFlow的开发环境和GTX显卡驱动部分的介绍（重点是TensorFlow的完整安装）。

深度学习之TensorFlow

入门、原理与进阶实战

第2章 搭建开发环境

配套视频



代码医生

qq群: 40016981

Blog: <http://blog.csdn.net/ljin6249>



*****ebook converter DEMO Watermarks*****

2.1 下载及安装Anaconda开发工具

下面介绍Anaconda的下载及安装方法。

(1) 通过百度找到Anaconda官网，单击第一个链接，如图2-1所示。或者直接访问网站<http://www.anaconda.com>。

(2) 进入Anaconda官网，单击右上角的DOWNLOAD按钮，如图2-2所示。

(3) 将屏幕拉到下面，单击最右侧的链接 Packages Included in Anaconda，如图2-3所示。



图2-1 找到Anaconda官网

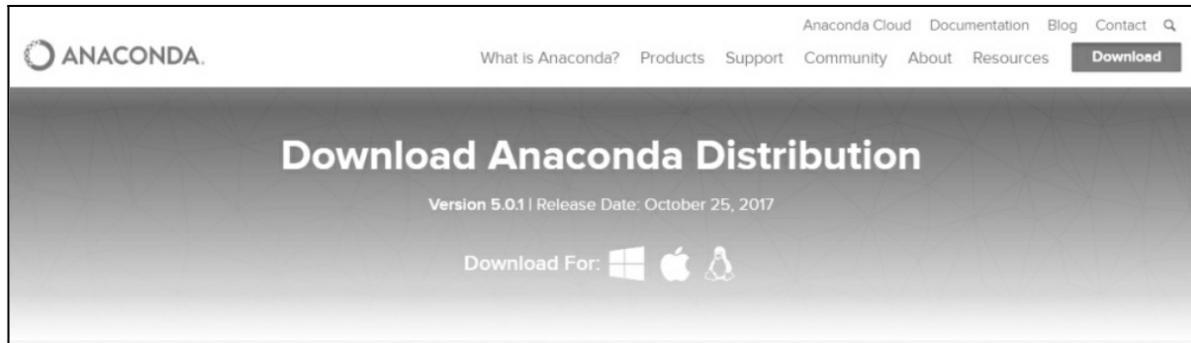


图2-2 Anaconda首页

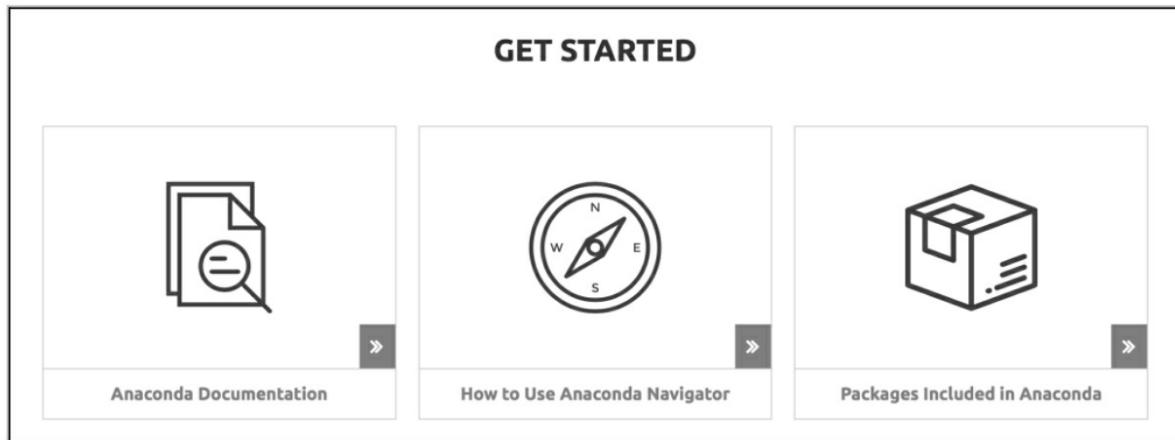


图2-3 DOWNLOAD选项

(4) 进入 Packages Included in Anaconda 页面，单击图中最后一行的 package repository 链接，如图2-4所示。

(5) 进入 Package repository 页面，如图2-5所示。最后一行是下载裁剪后的版本。如果硬盘足够大，建议选倒数第二行的链接下载。

(6) 进入完全版本的安装，如图2-6所示。这里有Linux、Windows、Mac OSX的各种版本，可以任意选择。

The screenshot shows the 'Anaconda package lists' page. The left sidebar has a 'Anaconda Platform' section with links to 'Welcome', 'Anaconda Distribution' (which is expanded), 'Installation', 'Packages', 'Anaconda package lists' (which is selected and highlighted in dark grey), and 'R language packages for Anaconda'. The main content area shows the title 'Anaconda package lists' and a paragraph explaining that all packages available in the latest release of Anaconda are listed on the pages linked below. It mentions that these packages may be installed with the command `conda install PACKAGENAME` and are located in the [package repository](#). Below this, there's a note about Python versions (3.6, 3.5 or 2.7) and operating systems (Windows, macOS, Linux). An RSS feed link is also provided.

图2-4 conda安装包

The screenshot shows the 'Anaconda repositories' page. The main title is 'Anaconda repositories'. Below it, there are four text blocks: 'Updates are available as an [RSS feed](#)', 'Anaconda is brought to you by [Anaconda, Inc.](#)', 'Anaconda installer archive [download page](#). ([winzip](#))', and 'Miniconda installers [download page](#). ([winzip](#))'.

图2-5 下载链接

Anaconda installer archive

Filename	Size
Anaconda2-4.4.0-Linux-x86.sh	415.0M
Anaconda2-4.4.0-Linux-x86_64.sh	485.2M
Anaconda2-4.4.0-MacOSX-x86_64.pkg	438.0M
Anaconda2-4.4.0-MacOSX-x86_64.sh	375.4M
Anaconda2-4.4.0-Windows-x86.exe	354.4M
Anaconda2-4.4.0-Windows-x86_64.exe	430.7M
Anaconda3-4.4.0-Linux-x86.sh	428.7M
Anaconda3-4.4.0-Linux-x86_64.sh	499.0M
Anaconda3-4.4.0-MacOSX-x86_64.pkg	442.5M
Anaconda3-4.4.0-MacOSX-x86_64.sh	380.4M
Anaconda3-4.4.0-Windows-x86.exe	362.2M
Anaconda3-4.4.0-Windows-x86_64.exe	437.6M
Anaconda2-4.4.0-Linux-ppc64le.sh	276.6M
Anaconda3-4.4.0-Linux-ppc64le.sh	290.7M
Anaconda2-4.3.1-Linux-x86.sh	387.7M
Anaconda2-4.3.1-Linux-x86_64.sh	462.0M
Anaconda2-4.3.1-MacOSX-x86_64.pkg	419.4M

图2-6 下载列表

注意：Anaconda的不同版本默认支持的Python版本是不一样的。对于支持Python 2的版本，统一以Anaconda 2为开头来命名；对于支持Python 3的版本，统一以Anaconda 3为开头来命名。当前最新的版本为5.0.0。可以支持Python 3.6版本。

TensorFlow中的1.3以前的版本不支持Python 3.6版本。为了更好地兼容，不建议下载最新的Anaconda 3版本，而是推荐使用Anaconda 3中支持Python 3.5的版本。例如：4.1.1、4.2.0等。

本书中使用的是Python 3.5版本，全文以该版本为例。

下面以Windows为例，来介绍具体的安装步骤。

以Anaconda 3-4.1.1版本（默认使用Python 3.5）为例，下载地址为https://repo.continuum.io/archive/Anaconda3-4.1.1-Windows-x86_64.exe。

假设安装位置为C:\local\Anaconda3-4.1.1-Windows-x86_64，安装好之后自动带有pip软件，可以通过pip安装其他软件。

2.2 在Windows平台下载及安装TensorFlow

首先来到<https://github.com/tensorflow/tensorflow>，在该页面中有安装文件的下载地址，如图2-7所示。

The screenshot shows the 'Installation' section of the TensorFlow GitHub repository. It includes instructions for installing from source or binaries, information about nightly pip packages, and links to individual whl files for various platforms.

Installation

See [Installing TensorFlow](#) for instructions on how to install our release binaries or how to build from source.

People who are a little more adventurous can also try our nightly binaries:

Nightly pip packages

- We are pleased to announce that TensorFlow now offers nightly pip packages under the `tf-nightly` and `tf-nightly-gpu` project on pypi. Simply run `pip install tf-nightly` or `pip install tf-nightly-gpu` in a clean environment to install the nightly TensorFlow build. We support CPU and GPU packages on Linux, Mac, and Windows.

Individual whl files

- Linux CPU-only: Python 2 ([build history](#)) / Python 3.4 ([build history](#)) / Python 3.5 ([build history](#))
- Linux GPU: Python 2 ([build history](#)) / Python 3.4 ([build history](#)) / Python 3.5 ([build history](#))
- Mac CPU-only: Python 2 ([build history](#)) / Python 3 ([build history](#))
- Windows CPU-only: Python 3.5 64-bit ([build history](#)) / Python 3.6 64-bit ([build history](#))
- Windows GPU: Python 3.5 64-bit ([build history](#)) / Python 3.6 64-bit ([build history](#))
- Android: demo APK, native libs ([build history](#))

图2-7 TensorFlow安装文件

1. 在线安装nightly包

nightly安装包是TensorFlow团队2017年下半年推出的安装模式。适用于在一个全新的环境下进行TensorFlow的安装。在安装TensorFlow的同时，默认会把需要依赖的库也一起装上，是非常方便、快捷的安装方式。

按照图2-7中的方法直接使用命令：

```
pip install tf-nightly
```

即可下载并安装TensorFlow的最新CPU版本。若要安装最新的GPU版本可以使用如下命令：

```
pip install tf-nightly-gpu
```

2. 安装纯净的TensorFlow

如果想安装纯净的TensorFlow版本，直接输入下面命令即可。

```
pip install tensorflow
```

上面是CPU版本，GPU版本的安装命令如下：

```
pip install tensorflow-gpu
```



注意： 在网速不稳定的情况下，在线安装有时会因为无法成功下载到完整的安装包而导致安装失败。可以通过重复执行安装命令或采用

*****ebook converter DEMO Watermarks*****

离线安装的方式来解决。

3. 更新安装TensorFlow

如果本地已经装有TensorFlow，需要升级为新版本的TensorFlow，只需要将原有版本卸载，再次安装即可。卸载命令如下：

```
pip uninstall <安装时的TensorFlow 名称>
```

4. 离线安装

有时由于网络环境的因素，无法实现在线安装，需要在网络环境好的地方提前将安装包下载下来进行离线安装。

(1) 下载安装包。

可以访问以下网站来查找TensorFlow的发布版本：

```
h https://storage.googleapis.com/tensorflow/
```

该网站内容是以XML方式提供的，查找起来不是很方便。可以通过地址加上指定的文件名方式进行下载。例如，一个TensorFlow 1.4.0的CPU版本安装包下载路径为：

<https://storage.googleapis.com/tensorflow/windows/cpu/tensorflow-1.4.0-cpu-0.9.1.exe>

TensorFlow1.4.0的GPU版本安装包下载路径
为：

<https://storage.googleapis.com/tensorflow/windows/gpu/tensorflow-1.4.0-gpu-0.9.1.exe>

如果要下载1.3.0的版本，直接将上面链接中的1.4.0改成1.3.0即可。

(2) 安装安装包。

下载完TensorFlow二进制文件后，假设使用CPU版本并且安装在D:\tensorflow下。选择“开始”|“运行”命令，在弹出的窗口中输入cmd，打开命令行窗口，然后输入如下命令来安装TensorFlow二进制文件。

C: \Users\Administrator>D:

D: \>cd tensorflow

D: \tensorflow>

D: \tensorflow>pip install tensorflow-1.1.0-cp35-cp35m-win_amd64.whl

2.3 GPU版本的安装方法

如果使用GPU版本，在执行pip之后，还需要安装CUDA和CuDNN。

2.3.1 安装CUDA软件包

首先来到CUDA官方网站
<https://developer.nvidia.com/cuda-downloads>，单击Windows按钮后，如图2-8所示。

Select Target Platform 

Click on the green buttons that describe your target platform. Only supported platforms will be shown.

Operating System Windows Linux Mac OSX

Architecture  x86_64

Version 10 8.1 7 Server 2012 R2 Server 2008 R2

Installer Type  exe (network) exe (local)

Download Installer for Windows 10 x86_64

The base installer is available for download below.

» Base Installer Download (1.2 GB) 

Installation Instructions:

1. Double click cuda_8.0.44_win10.exe
2. Follow on-screen prompts

The checksums for the installer and patches can be found in [Installer Checksums](#).
For further information, see the [Installation Guide for Microsoft Windows](#) and the [CUDA Quick Start Guide](#).

图2-8 CUDA页面

根据自己的环境选择对应的版本，.exe安装文件分为网络版和本地版。网络版安装包比较小，执行安装时再去下载需要的安装包；本地版安装包是直接下载完整的安装包。下载完成后正常安装就可以了。



注意： CUDA软件包也有很多个版本，必须与TensorFlow的版本对应才行。比如TensorFlow 1.0以后，直到TensorFlow 1.5的版本只支持CUDA 8.0。在本书中也是使用的CUDA 8.0版

*****ebook converter DEMO Watermarks*****

本来做演示的。可以根据链接
<https://developer.nvidia.com/cuda-toolkit-archive> 找到更多版本。

2.3.2 安装cuDNN库

输入<https://developer.nvidia.com/cudnn> 网址来到下载页面，需要注册并填一些问卷才能下载这个安装包。

cuDNN的版本选择也是有规定的。以Windows 10操作系统为例，TensorFlow 1.0到TensorFlow 1.2版本使用的是cuDNN的5.1版本（安装包文件为cudnn-8.0-windows10-x64-v5.1.zip），从TensorFlow 1.3版本之后使用的是cuDNN的6.0版本（cudnn-8.0-windows10-x64-v6.0.zip）。

得到相关包后解压，直接复制到cuda路径对应的文件夹下面就行，如图2-9所示。

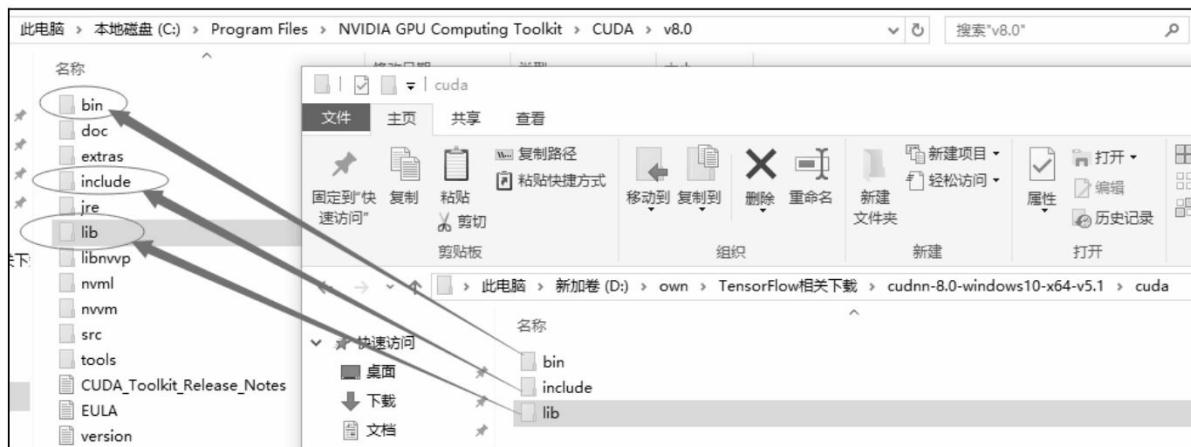


图2-9 安装cuDNN

2.3.3 测试显卡

这里再额外介绍两个小命令，它可以检测出在安装过程中产生的问题。

1. 使用nvidia-smi命令查看显卡信息

nvidia-smi指的是NVIDIA System Management Interface。在安装完成NVIDIA显卡驱动之后，对于Windows用户而言，cmd命令行界面还无法识别nvidia-smi命令，需要将相关环境变量添加进去。如果将NVIDIA显卡驱动安装在默认位置，nvidia-smi命令所在的完整路径应为：

C:\Program Files\NVIDIA Corporation\NVSMI

将上述路径添加进Path系统环境变量中。之后在cmd中运行nvidia-smi命令，可以看到显卡信息如图2-10所示。

NVIDIA-SMI 376.53						Driver Version: 376.53		
GPU	Name	TCC/WDDM	Bus-Id	Disp.A	Volatile	Uncorr.	ECC	
Fan	Temp	Perf	Pwr:Usage/Cap	Memory-Usage	GPU-Util	Compute M.		
0	GeForce GTX 1070	WDDM	0000:01:00.0	On			N/A	
59%	64C	P2	48W / 151W	6997MiB / 8192MiB	1%	Default		

Processes:					GPU Memory
GPU	PID	Type	Process name	Usage	
0	1248	C+G	Insufficient Permissions		N/A
0	3768	C	C:\local\Anaconda3\python.exe		N/A
0	5376	C+G	C:\Windows\explorer.exe		N/A
0	5668	C+G	...ost_cw5nlh2txyewy\ShellExperienceHost.exe		N/A
0	6460	C+G	...oftEdge_8wekyb3d8bbwe\MicrosoftEdgeCP.exe		N/A
0	7104	C+G	...iles (x86)\Internet Explorer\iexplore.exe		N/A
0	8260	C+G	...osoftEdge_8wekyb3d8bbwe\MicrosoftEdge.exe		N/A
0	8620	C+G	...indows.Cortana_cw5nlh2txyewy\SearchUI.exe		N/A

图2-10 显卡信息

图2-10中第1行是笔者的驱动信息，第3行是笔者的显卡信息GeForce GTX 1070。第4行和第5行是当前使用显卡的进程。

这些信息都存在了，表明笔者的安装是正确的。

2. 查看CUDA的版本

同样在cmd中使用命令nvcc -V，显示如图2-11所示。

```
C:\Users\11111111>nvcc -V
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2016 NVIDIA Corporation
Built on Mon_Jan_9_17:32:33_CST_2017
Cuda compilation tools, release 8.0, V8.0.60
```

图2-11 查看CUDA版本

3. 在Linux和Mac平台上安装

关于在Linux和Mac上安装TensorFlow的方法，可以参考网址http://www.tensorfly.cn/tfdoc/get_started/os_setup.html，这里不再展开讲述。

4. 问题处理

如果遇到问题的话，可以尝试下面的解决办法：

在命令行里输入where MSVCP140.DLL看看本机是否有MSVCP140.DLL，如果没有可以按照如下网址安装Visual C++ Redistributable 2015。

安装Visual C++ Redistributable 2015 x64（操作系统Windows10 64位），下载地址如下：

<https://www.microsoft.com/en-us/download/details.aspx?id=53587>

2.4 熟悉Anaconda 3开发工具

在本书中使用到的开发环境是Anaconda 3，在Anaconda 3里常用的有两个工具，即Spyder和Jupyter Notebook，它们的位置在开始菜单的Anaconda 3（64-bit）目录下，如图2-12所示。

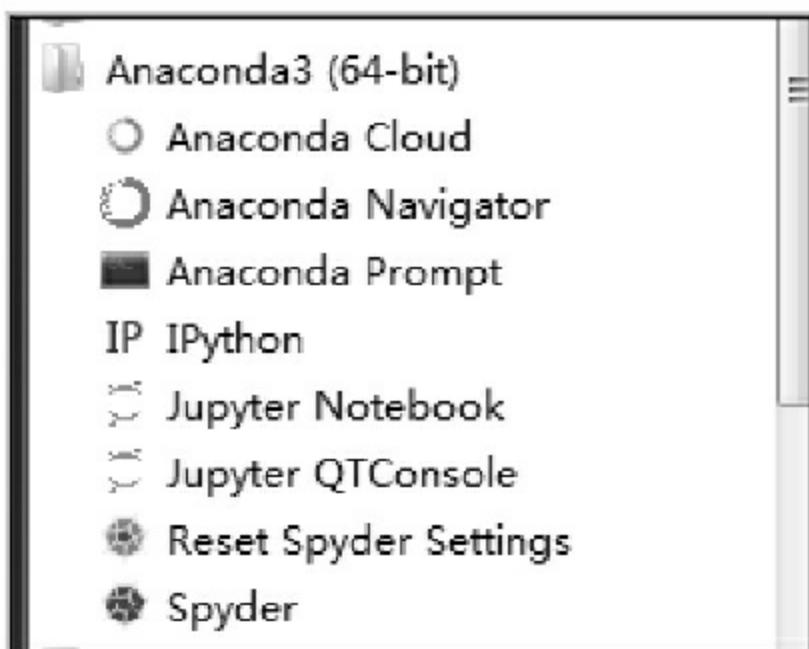


图2-12 Spyder和Jupyter Notebook的安装目录

2.4.1 快速了解Spyder

本书推荐使用Spyder作为编译器的原因是它比较方便，从安装到使用都做了相关的集成，只下载一个安装包即可，省去了大量的搭建环境时间。另外，Spyder的IDE功能也很强大，基本上可以满足日常需要。下面通过几个常用的功能来介

绍下其使用细节。

1. 面板介绍

如图2-13所示，Spyder启动后可以分为7个区域。



图2-13 Spyder面板

- 菜单栏：放置所有的功能。
- 快捷菜单栏：是菜单栏的快捷方式，其上面需要放置哪些快捷菜单，可以通过菜单栏中 View 的 Toolbars 的复选框来勾选，如图2-14所示。
- 工作区：就是代码要写的地方。
- 属性页的标题栏：可以显示当前代码的名字及位置。

- 查看栏：可以查看文件、调试时的对象及变量。

- 输出栏：可以看到程序的输出信息，也可以作为shell终端来输入Python语句。

- 状态栏：用来显示当前文件权限、编码，光标指向位置和系统内存。

2. 注释功能

注释功能为编写代码中很常用的功能，下面介绍Spyder的批量注释功能，在图2-14中，勾选Edit toolbar复选框，会看到如图2-15所示的注释按钮。

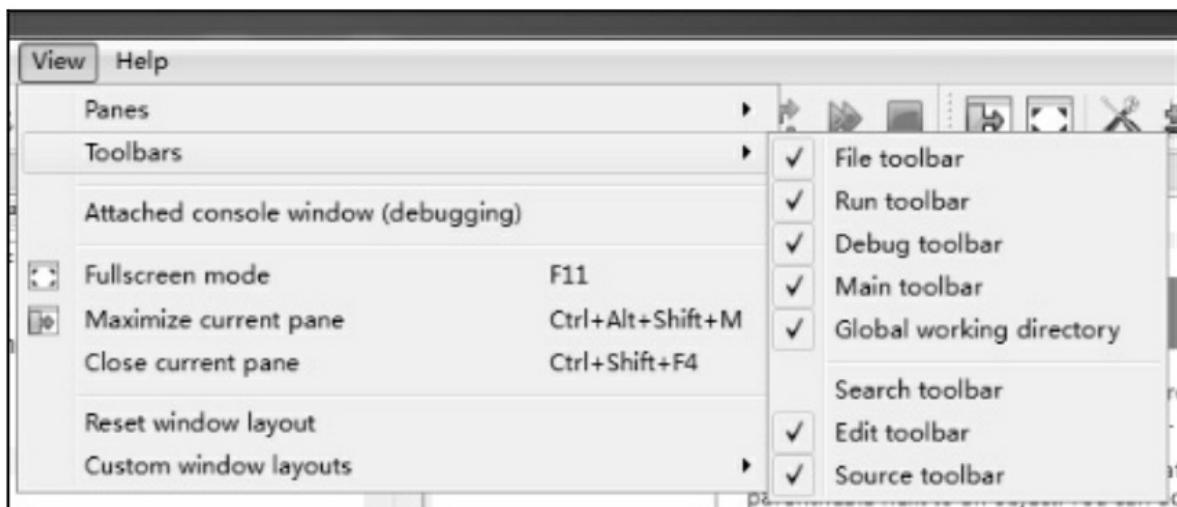


图2-14 快捷菜单设置

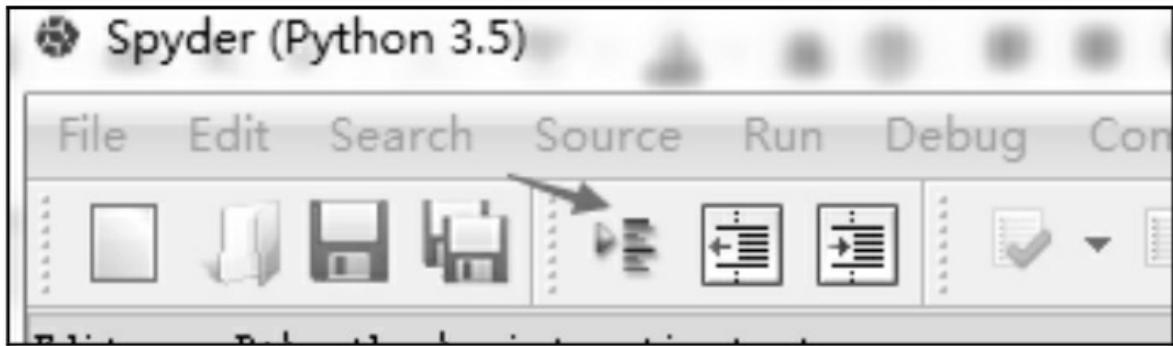


图2-15 注释按钮

当选中几行代码之后，单击该按钮即可对代码进行注释，再次单击为取消注释。该按钮右边两个按钮是代码缩进与不缩进按钮，不常用。可以通过Tab键与Shift+Tab键来实现。

3. 运行程序功能

如图2-16中，标注1按钮为运行当前工作区内的Python文件，单击2按钮会弹出一个Run settings对话框，可以输入启动程序的参数，如图2-16中标注框所示。

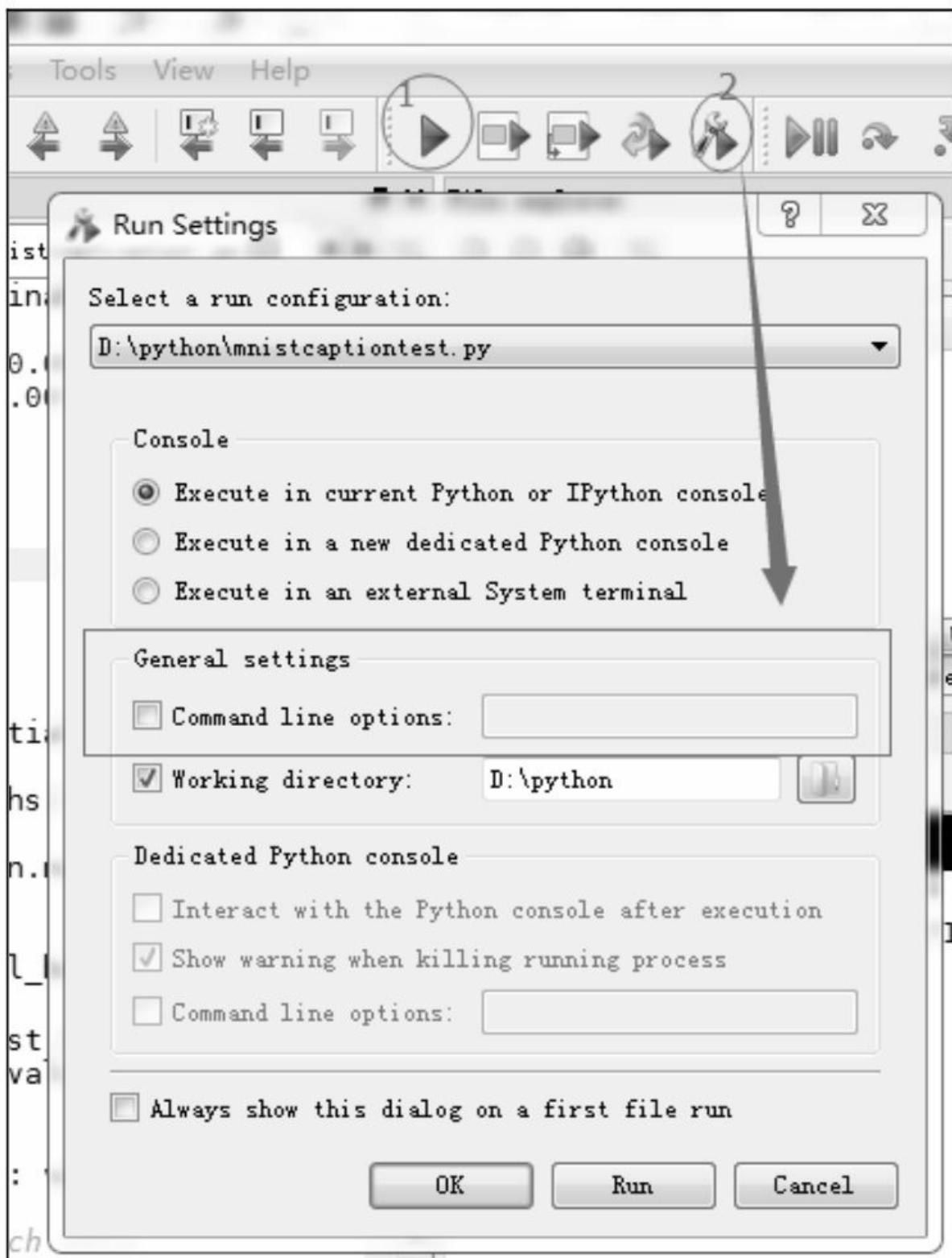


图2-16 运行程序

4. 调试功能

*****ebook converter DEMO Watermarks*****

如图2-16中右侧的按钮为调试功能的按钮，Python在运行中同样可以通过设置断点来进行调试。

5. Source操作

当同时打开多个代码时，有时想回到刚刚看的代码的位置，Spyder中有一个功能可以实现，在图2-14中，勾选Source toolbar复选框会看到如图2-17所示按钮，左边第一个按钮为建立书签，第二个按钮为回退上次的代码位置，第三个按钮为前进到下次代码位置。

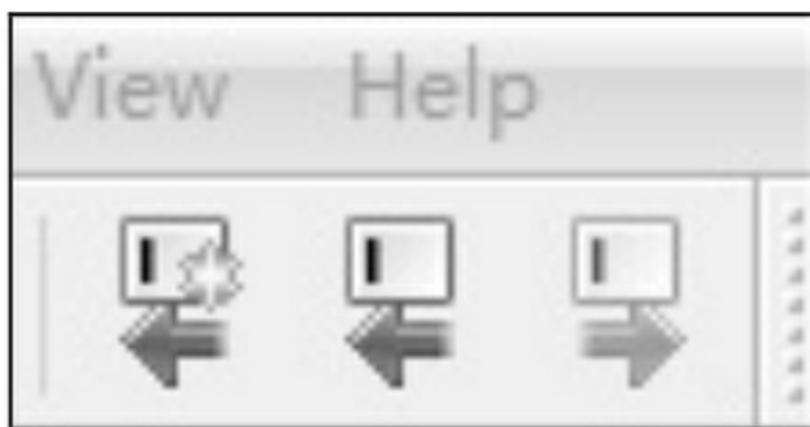


图2-17 Source

以上都是关于Spyder的常用操作。当然Spyder还有很多功能这里就不一一介绍了。

2.4.2 快速了解Jupyter Notebook

在深度学习中，有好多代码都被做成扩展名
*****ebook converter DEMO Watermarks*****

为ipynb的文件，这是一个关于Jupyter Notebook的文件，可以既当说明文档，又能运行Python代码的文件。Anaconda中也集成了这个软件。在图2-12中找到Jupyter Notebook项，单击即可看到如图2-18所示界面。

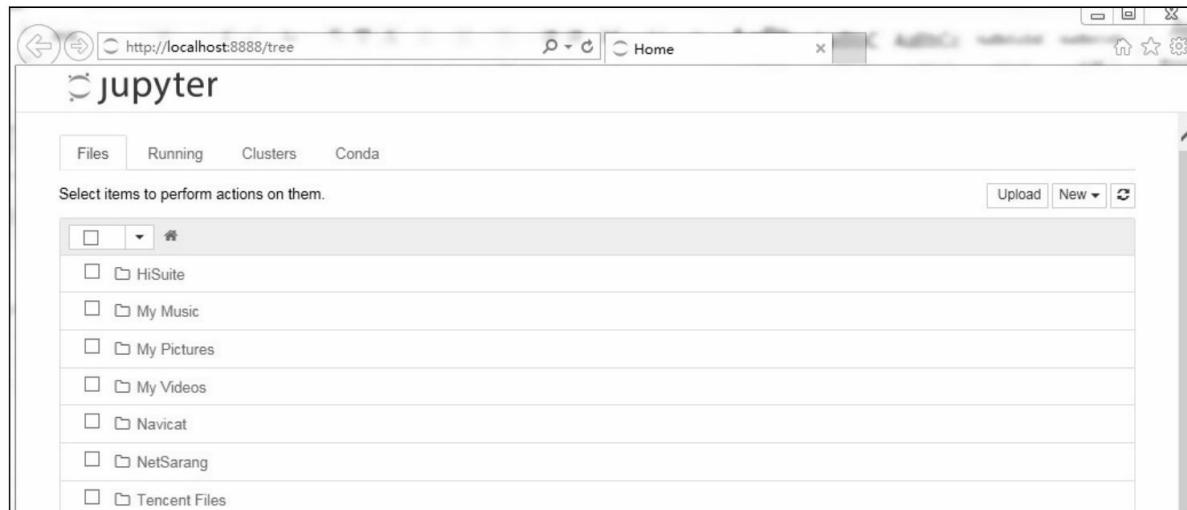


图2-18 Jupyter界面

该程序是B/S结构，会先启动一个Web服务器，然后再启动一个浏览器，通过浏览器来访问本机的服务。在这里面可以上传、下载，并编写自己的ipynb文件代码。

关于Jupyter Notebook工具的具体使用，这里不做过多介绍。有兴趣的读者可以参考网络上的众多使用教程。

第3章 TensorFlow基本开发步骤 ——以逻辑回归拟合二维数据为例

环境搭建好之后，读者一定迫不及待地想试试深度学习的程序了吧。本章就直接将一个例子拿出来，在没有任何基础的前提下，一步一步实现一个简单的神经网络。通过这个实例来理解模型，并了解TensorFlow开发的基本步骤。

本章含有教学视频共3分51秒。

作者按照本章的内容，讲解了一个使用神经网络拟合简单算式的例子，并借助这个例子介绍了TensorFlow的基本开发步骤（重点为了解基本开发步骤部分）。



*****ebook converter DEMO Watermarks*****

3.1 实例1：从一组看似混乱的数据中找出 $y \approx 2x$ 的规律

本节通过一个简单的逻辑回归实例为读者展示深度学习的神奇。通过对代码的具体步骤，让读者对深度学习有一个直观的印象。

实例描述

假设有一组数据集，其x和y的对应关系为 $y \approx 2x$ 。

本实例就是让神经网络学习这些样本，并能够找到其中的规律，即让神经网络能够总结出 $y \approx 2x$ 这样的公式。

深度学习大概有如下4个步骤：

- (1) 准备数据。
- (2) 搭建模型。
- (3) 迭代训练。
- (4) 使用模型。

准备数据阶段一般就是把任务的相关数据收集起来，然后建立网络模型，通过一定的迭代训
*****ebook converter DEMO Watermarks*****

练习让网络学习到收集来的数据特征，形成可用的模型，之后就是使用模型来为我们解决问题。

3.1.1 准备数据

这里使用 $y=2x$ 这个公式来做主体，通过加入一些干扰噪声让它的“等号”变成“约等于”。

具体代码如下：

- 导入头文件，然后生成 $-1 \sim 1$ 之间的100个数作为 x ，见代码第1~5行。
- 将 x 乘以2，再加上一个 $[-1, 1]$ 区间的随机数 $\times 0.3$ 。即， $y=2 \times x + a \times 0.3$ (a 属于 $[-1, 1]$ 之间的随机数)，见代码第6行。

代码3-1 线性回归

```
01 import tensorflow as tf
02      import numpy as np
03      import matplotlib.pyplot as plt
04
05      train_X = np.linspace(-1, 1, 100)
06      train_Y = 2 * train_X + np.random.randn(*train_X.shape) * 0.3
07      #但是加入了噪声
08      #显示模拟数据点
09      plt.plot(train_X, train_Y,
10              'ro', label='Original data')
11      plt.legend()
12      plt.show()
```



注意：

`np.random.randn (*train_X.shape)` 这个代码如果看起来比较奇怪，现在给出解释——它等同于 `np.random. randn (100)`

运行上面代码，显示结果如图3-1所示。

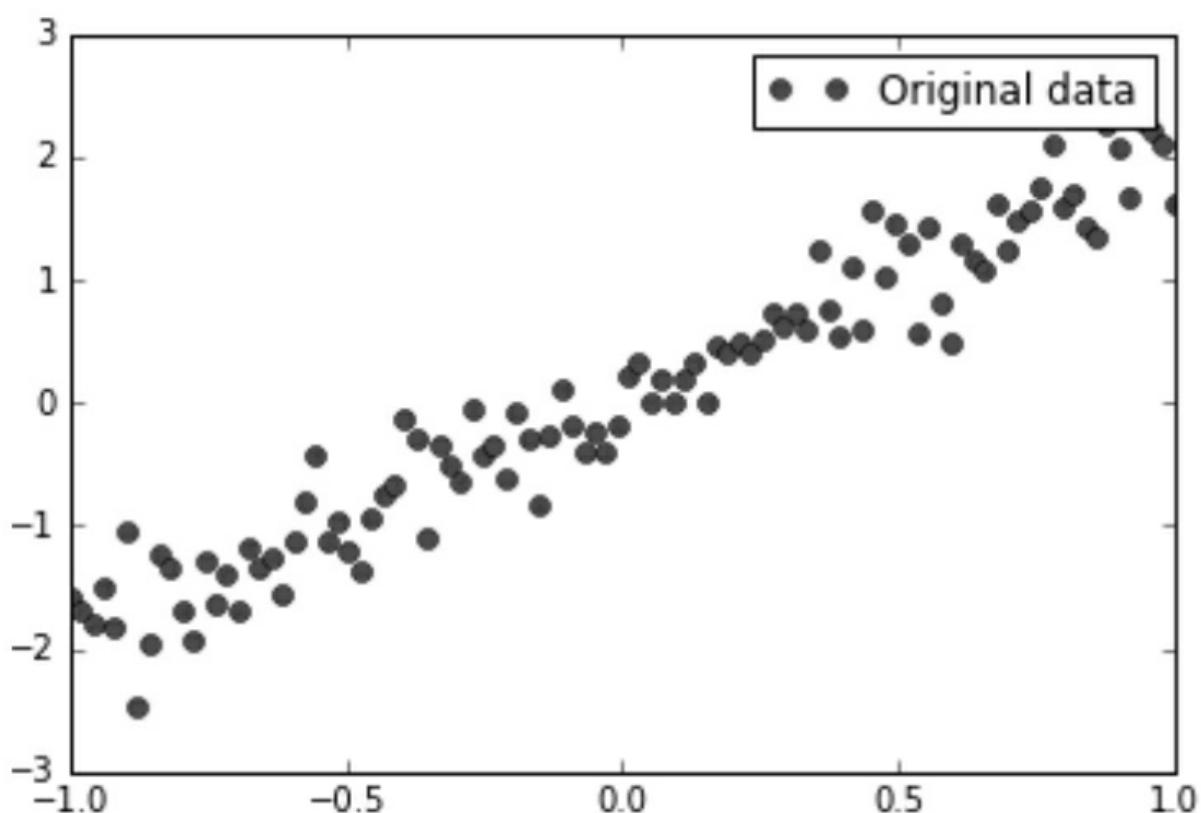


图3-1 准备好的线性回归数据集

3.1.2 搭建模型

现在开始进行模型搭建。模型分为两个方向：正向和反向。

1. 正向搭建模型

(1) 了解模型及其公式

在具体操作之前，先来了解一下模型的样子。神经网络是由多个神经元组成的，单个神经元的网络模型如图3-2所示。

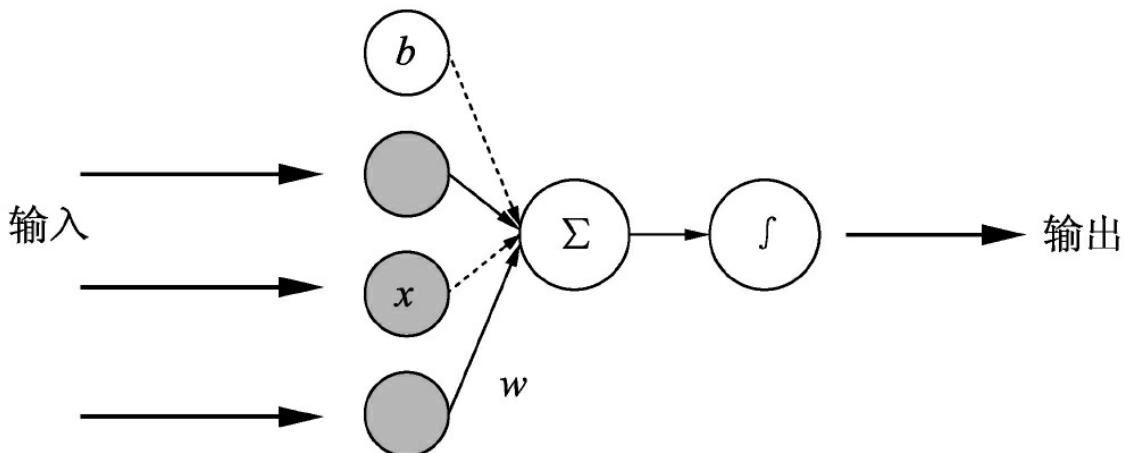


图3-2 神经元模型

其计算公式见式（3-1）：

$$z = \sum_{i=1}^n w_i \times x_i + b = w \cdot x + b \quad \text{式 (3-1)}$$

式中， z 为输出的结果， x 为输入， w 为权重， b 为偏执值。

z 的计算过程是将输入的 x 与其对应的 w 相乘，然后再把结果相加上偏执 b 。

例如，有3个输入 x_1 ， x_2 ， x_3 ，分别对应 w_1 ， w_2 ， w_3 ，则， $z=x_1 \times w_1 + x_2 \times w_2 + x_3 \times w_3 + b$ 。这一过程中，在线性代数中正好可以用两个矩阵来表示，于是就可以写成（矩阵W）×（矩阵X）+b。矩阵相乘的展开如式（3-2）：

$$\{w_1, w_2, w_3\} \times \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \end{Bmatrix} = x_1 \times w_1 + x_2 \times w_2 + x_3 \times w_3 \quad \text{式 (3-2)}$$

上面的算式（3-2）表明：形状为1行3列的矩阵与3行1列的矩阵相乘，结果的形状为1行1列的矩阵，即 $(1, 3) \times (3, 1) = (1, 1)$



注意：这里有个小窍门，如果想得到两个矩阵相乘后的形状，可以将第一个矩阵的行与第二个矩阵的列组合起来，就是相乘后的形状。

在神经元中，w和b可以理解为两个变量。模型每次的“学习”都是调整w和b以得到一个更合适的值。最终，有这个值配合上运算公式所形成的逻辑就是神经网络的模型。

（2）创建模型

下面的代码演示了如何创建图3-2中的模型。

代码3-1 线性回归（续）

```
11 # 创建模型  
12 # 占位符  
13 X = tf.placeholder("float")  
14 Y = tf.placeholder("float")  
15 # 模型参数  
16 W = tf.Variable(tf.random_normal([1]), name="weight")  
17 b = tf.Variable(tf.zeros([1]), name="bias")  
18 # 前向结构  
19 z = tf.multiply(X, W) + b
```

下面解说一下代码。

(1) X和Y：为占位符，使用了placeholder函数进行定义。一个代表x的输入，一个代表对应的真实值y。占位符的意思后面再解释。

(2) W和b：就是前面说的参数。W被初始化成[-1, 1]的随机数，形状为一维的数字，b的初始化为0，形状也是一维的数字。

(3) Variable：定义变量，在3.3节会有详细介绍。

(4) tf.multiply：是两个数相乘的意思，结果再加上b就等于z了。

2. 反向搭建模型

神经网络在训练的过程中数据的流向有两个

方向，即先通过正向生成一个值，然后观察其与真实值的差距，再通过反向过程将里面的参数进行调整，接着再次正向生成预测值并与真实值进行比对，这样循环下去，直到将参数调整为合适值为止。

正向相对比较好理解，反向传播会引入一些算法来实现对参数的正确调整。

下面先看一下反向优化的相关代码。

代码3-1 线性回归（续）

```
20 #反向优化
21 cost = tf.reduce_mean(tf.square(Y - z))
22 learning_rate = 0.01
23 optimizer = tf.train.GradientDescentOptimizer(learning_
(cost) #梯度下降
```

代码说明如下：

(1) 第21行定义一个cost，它等于生成值与真实值的平方差。

(2) 第22行定义一个学习率，代表调整参数的速度。这个值一般是小于1的。这个值越大，表明调整的速度越大，但不精确；值越小，表明调整的精度越高，但速度慢。这就好比生物课上的显微镜调试，显微镜上有两个调节焦距的旋转钮，分为粗调和细调。

(3) 第23行GradientDescentOptimizer函数是一个封装好的梯度下降算法，里面的参数learning_rate叫做学习率，用来指定参数调节的速度。如果将“学习率”比作显微镜上不同档位的“调节钮”，那么梯度下降算法也可以理解成“显微镜筒”，它会按照学习参数的速度来改变显微镜上焦距的大小。

3.1.3 迭代训练模型

迭代训练的代码分成两步来完成：

1. 训练模型

建立好模型后，可以通过迭代来训练模型了。TensorFlow中的任务是通过session来进行的。

下面的代码中，先进行全局初始化，然后设置训练迭代的次数，启动session开始运行任务。

代码3-1 线性回归（续）

```
24 #初始化所有变量
25 init = tf.global_variables_initializer()
26 #定义参数
27 training_epochs = 20
28 display_step = 2
29
30 #启动session
31 with tf.Session() as sess:
```

```
32     sess.run(init)
33     plotdata={"batchsize":[], "loss":[]} #存放批次值和损失值
34     #向模型输入数据
35     for epoch in range(training_epochs):
36         for (x, y) in zip(train_X, train_Y):
37             sess.run(optimizer, feed_dict={X: x, Y: y})
38
39     #显示训练中的详细信息
40     if epoch % display_step == 0:
41         loss = sess.run(cost, feed_dict={X:train_X, Y:train_Y})
42         print ("Epoch:", epoch+1, "cost=", loss, "W=", sess.run(W),
43               "b=", sess.run(b))
44         if not (loss == "NA" ):
45             plotdata["batchsize"].append(epoch)
46             plotdata["loss"].append(loss)
47
48     print (" Finished!")
49     print ("cost=", sess.run(cost, feed_dict={X: train_X, "W": sess.run(W), "b": sess.run(b)})
```

上面的代码中迭代次数设置为20次，通过sess.run来进行网络节点的运算，通过feed机制将真实数据灌到占位符对应的位置（feed_dict={X: x, Y: y}），同时，每执行一次都会将网络结构中的节点打印出来。

运行代码，输出信息如下：

```
Epoch: 1 cost= 0.714926 W= [ 0.71911603] b= [ 0.40933588]
Epoch: 3 cost= 0.114213 W= [ 1.63318455] b= [ 0.17000227]
Epoch: 5 cost= 0.0661118 W= [ 1.88165665] b= [ 0.0765276]
Epoch: 7 cost= 0.0633376 W= [ 1.94610846] b= [ 0.05182607]
Epoch: 9 cost= 0.0632785 W= [ 1.96277654] b= [ 0.0454303]
Epoch: 11 cost= 0.0633072 W= [ 1.96708632] b= [ 0.04377643]
Epoch: 13 cost= 0.0633176 W= [ 1.96820116] b= [ 0.04334867]
Epoch: 15 cost= 0.0633205 W= [ 1.96848941] b= [ 0.04323809]
Epoch: 17 cost= 0.0633212 W= [ 1.9685632] b= [ 0.04320973]
Epoch: 19 cost= 0.0633214 W= [ 1.96858287] b= [ 0.04320224]
    Finished!
cost= 0.0633215 W= [ 1.96858633] b= [ 0.04320095]
```

*****ebook converter DEMO Watermarks*****

可以看出，cost的值在不断地变小，w和b的值也在不断地调整。

2. 训练模型可视化

上面的数值信息理解起来还是比较抽象。为了可以得到更直观的表达，下面将模型中的两个信息可视化出来，一个是生成的模型，另一个是训练中的状态值。具体代码如下：

代码3-1 线性回归（续）

```
49 #图形显示
50     plt.plot(train_X, train_Y, 'ro', label='Original dat
51     plt.plot(train_X, sess.run(w) * train_X + sess.run(b
52     plt.legend()
53     plt.show()
54
55     plotdata["avgloss"] = moving_average(plotdata["loss"]
56     plt.figure(1)
57     plt.subplot(211)
58     plt.plot(plotdata["batchsize"], plotdata["avgloss"])
59     plt.xlabel('Minibatch number')
60     plt.ylabel('Loss')
61     plt.title('Minibatch run vs. Training loss')
62
63     plt.show()
```

这段代码中引入了一个变量和一个函数，可以在代码的最顶端定义它们，见如下代码：

```
plotdata = { "batchsize":[], "loss":[] }
```

*****ebook converter DEMO Watermarks*****

```
def moving_average(a, w=10):
    if len(a) < w:
        return a[:]
    return [val if idx < w else sum(a[(idx-w):idx])/w for idx, val in enumerate(a)]
```

现在所有的代码都准备好了，运行程序，生成如图3-3和图3-4所示两幅图。

图3-3中所示的斜线，是模型中的参数 w 和 b 为常量所组成的关于 x 与 y 的直线方程。可以看到是一条几乎 $y=2x$ 的直线（ $W=1.96858633$ 接近于2， $b=0.04320095$ 接近于0）。

在图3-4中可以看到刚开始损失值一直在下降，直到6次左右趋近平稳。

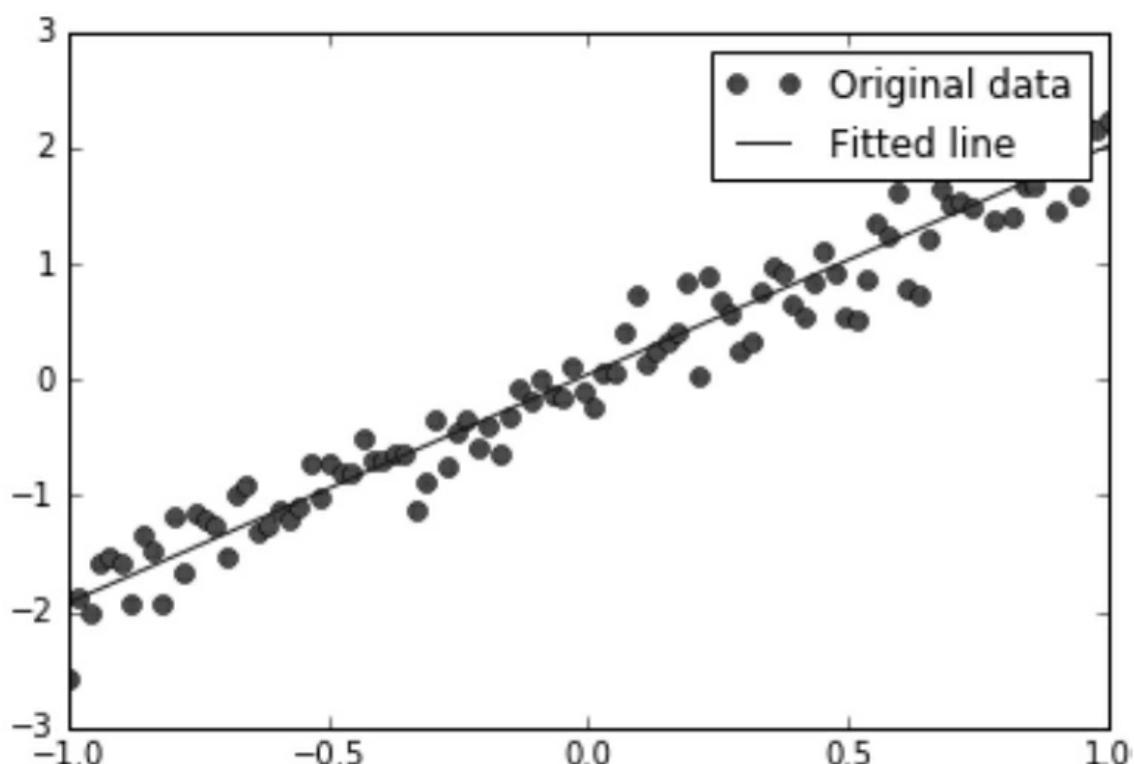


图3-3 可视化模型

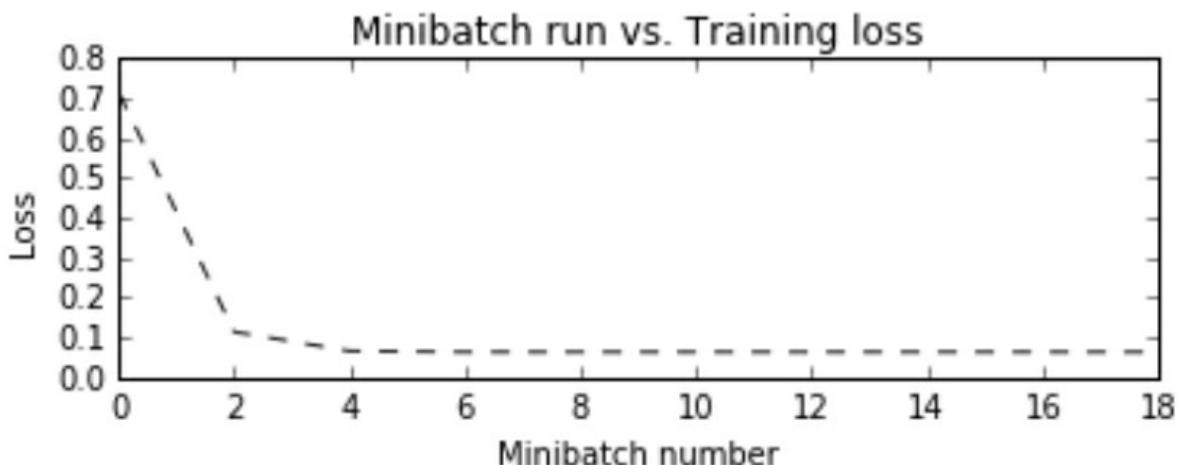


图3-4 可视化训练loss

3.1.4 使用模型

模型训练好后，用起来就比较容易了，往里面传一个0.2（通过`feed_dict={X: 0.2}`），然后使用`sess.run`来运行模型中的`z`节点，见如下代码第64行，看看它生成的值。

代码3-1 线性回归（续）

```
64 print ("x=0.2, z=", sess.run(z, feed_dict={X: 0.2}))
```

将上述代码加到代码文件“3-1线性回归.py”的最后一行，运行后可以得到如下信息：

```
x=0.2, z= [ 0.4324449]
```

训练好的模型，可以根据已有数据的规律推算出输入值0.2对应的z值。



注意： 读者在自己的计算机上运行该程序，得到的z值与书上的会不一样。这是因为b和w不一样。神经网络学习的是一种规律，能表示这一种规律的b和w会有很多值，即模型学出来的并非是唯一值。

3.2 模型是如何训练出来的

在上面的例子中仅仅迭代了20次就得到了一个可以拟合 $y \approx 2x$ 的模型。下面来具体了解一下模型是如何得来的。

3.2.1 模型里的内容及意义

一个标准的模型结构分为输入、中间节点、输出三大部分，而如何让这三个部分连通起来学习规则并可以进行计算，则是框架TensorFlow所做的事情。

TensorFlow将中间节点及节点间的运算关系（OPS）定义在自己内部的一个“图”上，全通过一个“会话（session）”进行图中OPS的具体运算。

可以这样理解：

- “图”是静态的，无论做任何加、减、乘、除，它们只是将关系搭建在一起，不会有任何运算。

- “会话”是动态的，只有启动会话后才会将数据流向图中，并按照图中的关系运算，并将最终的结果从图中流出。

TensorFlow用这种方式分离了计算的定义和执行，“图”类似于施工图（blueprint），而“会话”更像施工地点。

构建一个完整的图一般需要定义3种变量，如图3-5所示。

- 输入节点：即网络的入口。
- 用于训练的模型参数（也叫学习参数）：是连接各个节点的路径。
- 模型中的节点（OP）：最复杂的就是OP。OP可以用来代表模型中的中间节点，也可以代表最终的输出节点，是网络中的真正结构。

如图3-5所示为这3种变量放在图中所组成的网络静态模型。在实际训练中，通过动态的会话将图中的各个节点按照静态的规则运算起来，每一次的迭代都会对图中的学习参数进行更新调整，通过一定次数的迭代运算之后最终所形成的图便是所要的“模型”。而在会话中，任何一个节点都可以通过会话的run函数进行计算，得到该节点的真实数值。

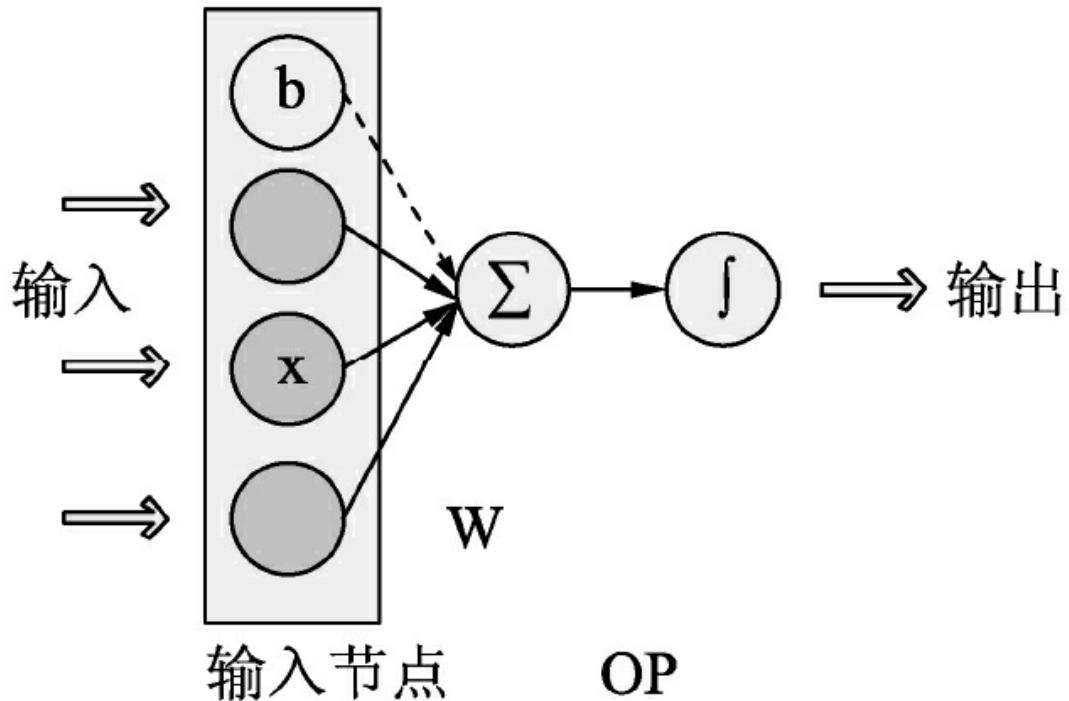


图3-5 模型中的图

3.2.2 模型内部的数据流向

模型内部的数据流向分为正向和反向。

1. 正向

正向，是数据从输入开始，依次进行各节点定义的运算，一直运算到输出，是模型最基本的数据流向。它直观地表现了网络模型的结构，在模型的训练、测试、使用的场景中都会用到。这部分是必须要掌握的。

2. 反向

反向，只有在训练场景下才会用到。这里使

*****ebook converter DEMO Watermarks*****

用了一个叫做反向链式求导的方法，即先从正向的最后一个节点开始，计算此时结果值与真实值的误差，这样会形成一个用学习参数表示误差的方程，然后对方程中的每个参数求导，得到其梯度修正值，同时反推出上一层的误差，这样就将该层节点的误差按照正向的相反方向传到上一层，并接着计算上一层的修正值，如此反复下去一步一步地进行转播，直到传到正向的第一个节点。

这部分原理TensorFlow已经实现好了，读者简单理解即可，应该把重点放在使用什么方法来计算误差，使用哪些梯度下降的优化方法，如何调节梯度下降中的参数（如学习率）问题上。

3.3 了解TensorFlow开发的基本步骤

通过上面的例子，现在将TensorFlow开发的基本步骤总结如下：

- (1) 定义TensorFlow输入节点。
- (2) 定义“学习参数”的变量。
- (3) 定义“运算”。
- (4) 优化函数，优化目标。
- (5) 初始化所有变量。
- (6) 迭代更新参数到最优解。
- (7) 测试模型。
- (8) 使用模型。

下面进行逐项介绍。

3.3.1 定义输入节点的方法

TensorFlow中有如下几种定义输入节点的方法。

- 通过占位符定义：一般使用这种方式。
- 通过字典类型定义：一般用于输入比较多的情况。
- 直接定义：一般很少使用。

本章开篇的第一个例子“3-1线性回归.py”就是通过占位符来定义输入节点的，具体使用了tf.placeholder函数，见如下代码。

```
X = tf.placeholder("float")
Y = tf.placeholder("float")
```

下面介绍“通过字典定义”与“直接定义”的方法。

3.3.2 实例2：通过字典类型定义输入节点

实例描述

在代码“3-1线性回归.py”文件的基础上，使用字典占位符来代替用占位符定义的输入。

通过字典定义的方式和第一种比较像，只不过是堆叠到了一起。具体代码如下：

代码3-2 通过字典类型定义输出节点

```
.....  
# 占位符  
inputdict = {  
    'x': tf.placeholder("float"),  
    'y': tf.placeholder("float")  
}
```

3.3.3 实例3：直接定义输入节点

实例描述

在代码“3-1线性回归.py”文件的基础上，使用直接定义法来代替用占位符定义的输入。

直接定义，就是将定义好的Python变量直接放到OP节点中参与输入的运算，将模拟数据的变量直接放到模型中进行训练。代码如下：

代码3-3 直接定义输入节点

```
.....  
  
#生成模拟数据  
train_X = np.float32( np.linspace(-1, 1, 100))  
train_Y = 2 * train_X + np.random.randn(*train_X.shape) * 0.  
#图形显示  
plt.plot(train_X, train_Y, 'ro', label='Original data')  
plt.legend()  
plt.show()  
  
# 模型参数  
W = tf.Variable(tf.random_normal([1]), name="weight")  
b = tf.Variable(tf.zeros([1]), name="bias")  
# 前向结构  
z = tf.multiply(W, train_X)+ b
```



提示：上面只列出了3种方法中的关键代码，全部的代码在本书的配套代码中可以找到。

3.3.4 定义“学习参数”的变量

学习参数的定义与输入的定义很像，分为直接定义和字典定义两部分。这两种都是常见的使用方式，只不过在深层神经网络里由于参数过多，普遍都会使用第二种情况。

在前面“3-1线性回归.py”的例子中使用的就是第一种方法，通过tf.Variable可以对参数直接定义。代码如下：

```
# 模型参数  
W = tf.Variable(tf.random_normal([1]), name="weight")  
b = tf.Variable(tf.zeros([1]), name="bias")
```

下面通过例子演示使用字典定义学习参数。

3.3.5 实例4：通过字典类型定义“学习参数”

实例描述

在代码“3-1线性回归.py”文件的基础上，使用

*****ebook converter DEMO Watermarks*****

字典的方式来定义学习参数。

通过字典的方式定义和直接定义比较相似，只不过是堆叠到了一起。修改“3-1线性回归.py”例子代码如下。

代码3-4 通过字典类型定义学习参数

```
.....  
# 模型参数  
paradict = {  
    'w': tf.Variable(tf.random_normal([1])),  
    'b': tf.Variable(tf.zeros([1]))  
}  
# 前向结构  
z = tf.multiply(x, paradict['w'])+ paradict['b']
```

上面代码同样只是列出了关键部分，全部的代码都可以在本书的配套代码中找到。

3.3.6 定义“运算”

定义“运算”的过程是建立模型的核心过程，直接决定了模型的拟合效果，具体的代码演示在前面也介绍过了。这里主要阐述一下定义运算的类型，以及其在深度学习中的作用。

1. 定义正向传播模型

在前面“3-1线性回归.py”的例子中使用的网络

结构很简单，只有一个神经元。在后面会学到多层神经网络、卷积神经网、循环神经网络及更深层的GoogLeNet、Resnet等，它们都是由神经元以不同的组合方式组成的网络结构，而且每年还会有很多更高效且拟合性更强的新结构诞生。

2. 定义损失函数

损失函数主要是计算“输出值”与“目标值”之间的误差，是配合反向传播使用的。为了在反向传播中可以找到最小值，要求该函数必须是可导的。



提示： 损失函数近几年来没有太大变化。读者只需要记住常用的几种，并能够了解内部原理就可以了，不需要掌握太多细节，因为TensorFlow框架已经为我们做好了。

3.3.7 优化函数，优化目标

在有了正向结构和损失函数后，就是通过优化函数来优化学习参数了，这个过程也是在反向传播中完成的。

反向传播过程，就是沿着正向传播的结构向相反方向将误差传递过去。这里面涉及的技术比较多，如L1、L2正则化、冲量调节、学习率自适

应、adm随机梯度下降算法等，每一个技巧都代表一个时代。



提示： 随着深度学习的飞速发展，反向传播过程的技术会达到一定程度的瓶颈，更新并不如网络结构变化得那么快，所以读者也只需将常用的几种记住即可。

3.3.8 初始化所有变量

初始化所有变量的过程，虽然只有一句代码，但也是一个关键环节，所以特意将其列出来。

在session创建好了之后，第一件事就是需要初始化。还以“3-1线性回归.py”举例，代码如下：

```
init = tf.global_variables_initializer()
# 启动Session
with tf.Session() as sess:
    sess.run(init)
```



注意： 使用tf.global_variables_initializer函数初始化所有变量的步骤，必须在所有变量和OP定义完成之后。这样才能保证定义的内容有效，否则，初始化之后定义的变量和OP都无法使用session中的run来进行算值。

3.3.9 迭代更新参数到最优解

在迭代训练环节，都是需要通过建立一个session来完成的，常用的是使用with语法，可以在session结束后自行关闭，当然还有其他方法，第4章会详细介绍。

```
with tf.Session() as sess:
```

前面说过，在session中通过run来运算模型中的节点，在训练环节也是如此，只不过run里面放的是优化操作的OP，同时会在外层加上循环次数。

```
for epoch in range(training_epochs):
    for (x, y) in zip(train_X, train_Y):
        sess.run(optimizer, feed_dict={X: x, Y: y})
```

真正使用过程中会引入一个叫做MINIBATCH概念进行迭代训练，即每次取一定量的数据同时放到网络里进行训练，这样做的好处和意义会在后面详细介绍。

3.3.10 测试模型

测试模型部分已经不是神经网络的核心环节了，同归对评估节点的输出，得到模型的准确率

(或错误率) 从而来描述模型的好坏，这部分很简单没有太多的技术，在“3-1线性回归.py”中可以找到如下代码：

```
print ("cost=", sess.run(cost, feed_dict={X: train_X, Y: train_Y}))  
print ("w=", sess.run(w), "b=", sess.run(b))
```

当然这句话还可以改写成以下这样：

```
print ("cost:", cost.eval({X: train_X, Y: train_Y}))
```

3.3.11 使用模型

使用模型也与测试模型类似，只不过是将损失值的节点换成输出的节点即可。在“3-1线性回归.py”例子中也有介绍。

这里要说的是，一般会把生成的模型保存起来，再通过载入已有的模型来进行实际的使用。关于模型的载入和读取，后面章节会有介绍。

第4章 TensorFlow编程基础

本章主要介绍TensorFlow的基础语法及功能函数。学完本章后，TensorFlow代码对读者来说将不再陌生，读者可以很轻易看懂网上和书中例子的代码，并可以尝试写一些简单的模型或算法。

学习一个开发环境，应先从其内部入手，这样会起到事半功倍的效果。本章先从编程模型开始了解其运行机制，然后再介绍TensorFlow常用操作及功能函数，最后是共享变量、图和分布式部署。

本章含有教学视频共12分39秒。

作者按照本章的内容结构，对主要内容进行了快速讲解，包括基本的模型工作机制、基础类型及操作、共享变量、图、分布式部署等内容（其中，共享变量是本章的重点和难点）。

深度学习之TensorFlow

入门、原理与进阶实战

第4章 TensorFlow编程基础

配套视频



代码医生

qq群: 40016981

http://blog.csdn.net/ljjin6249



字幕



*****ebook converter DEMO Watermarks*****

4.1 编程模型

TensorFlow的命名来源于本身的运行原理。Tensor（张量）意味着N维数组，Flow（流）意味着基于数据流图的计算。TensorFlow是张量从图像的一端流动到另一端的计算过程，这也是TensorFlow的编程模型。

4.1.1 了解模型的运行机制

TensorFlow的运行机制属于“定义”与“运行”相分离。从操作层面可以抽象成两种：模型构建和模型运行。

在模型构建过程中，需要先了解几个概念，如表4-1所示。

表4-1 模型构建中的概念

名 称	含 义
张量 (tensor)	数据，即某一类型的多维数组
变量 (Variable)	常用于定义模型中的参数，是通过不断训练得到的值
占位符 (placeholder)	输入变量的载体。也可以理解成定义函数时的参数
图中的节点操作 (operation, OP)	即一个OP获得0个或多个tensor，执行计算，输出额外的0个或多个tensor

表4-1中定义的内容都是在一个叫做“图”的容器中完成的。关于“图”，有以下几点需要理解。

- 一个“图”代表一个计算任务。

- 在模型运行的环节中，“图”会在会话(session)里被启动。
- session将图的OP分发到如CPU或GPU之类的设备上，同时提供执行OP的方法。这些方法执行后，将产生的tensor返回。在Python语言中，返回的tensor是numpy ndarray对象；在C和C++语言中，返回的tensor是TensorFlow::Tensor实例。

如图4-1所示为session与图的工作关系。

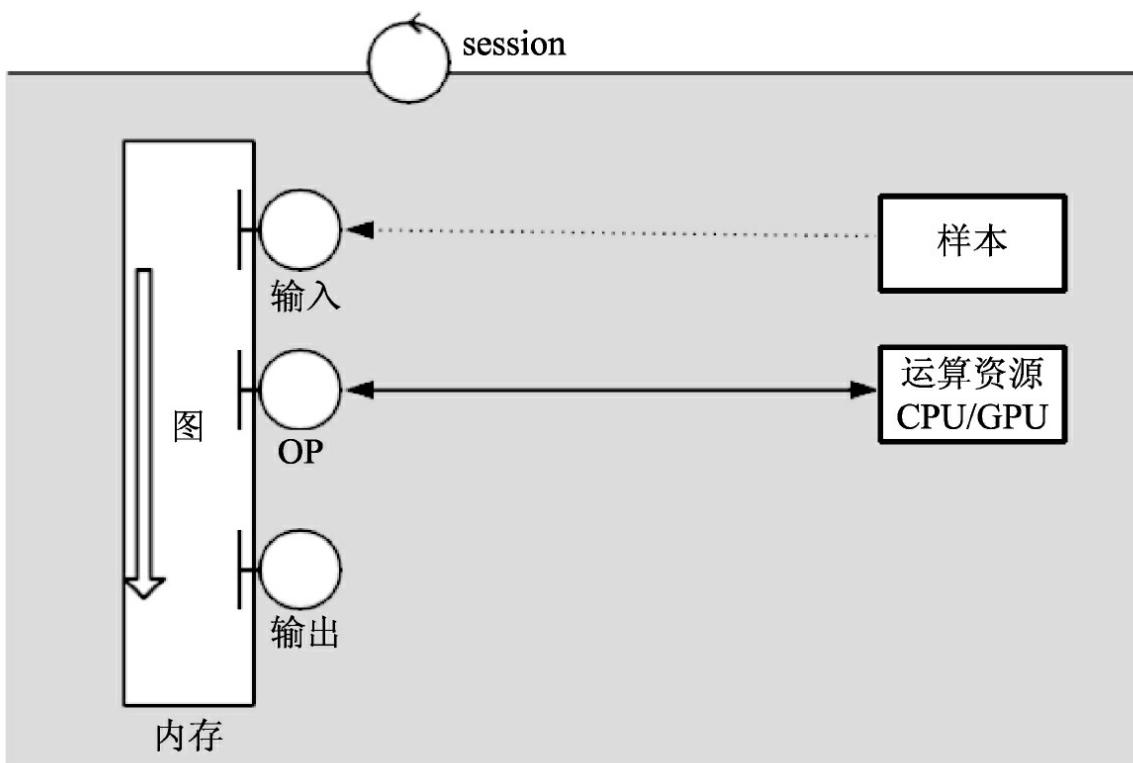


图4-1 session与图的关系

在实际环境中，这种运行情况会有3种应用场景，分别是训练场景、测试场景与使用场景。在训练场景下图的运行方式与其他两种不同，具

体介绍如下。

(1) 训练场景：是实现模型从无到有的过程，通过对样本的学习训练，调整学习参数，形成最终的模型。其过程是将给定的样本和标签作为输入节点，通过大量的循环迭代，将图中的正向运算（从输入的样本通过OP运算得到输出的方向）得到的输出值，再进行反向运算（从输出到输入的方向），以更新模型中的学习参数，最终使模型产生的正向结果最大化地接近样本标签。这样就得到了一个可以拟合样本规律的模型。

(2) 测试场景和使用场景：测试场景是利用图的正向运算得到的结果与真实值进行比较的差别；使用场景也是利用图的正向运算得到结果，并直接使用。所以二者的运算过程是一样的。对于该场景下的模型与正常编程用到的函数特别相似。在函数中，可以分为实参、形参、函数体与返回值。同样在模型中，实参就是输入的样本，形参就是占位符，运算过程就相当于函数体，得到的结果相当于返回值。

另外，session与图的交互过程中还定义了以下两种数据的流向机制。

- 注入机制 (feed) : 通过占位符向模式中传入数据。

- 取回机制 (fetch) : 从模式中得到结果。

下面通过实例逐个演示session在各种情况下的用法。先从session的建立开始，接着演示session与图的交互机制，最后演示如何在session中指定GPU运算资源。

4.1.2 实例5：编写hello world程序演示session的使用

下面先从一个hello world开始来理解session的作用。

实例描述

建立一个session，在session中输出hello，TensorFlow。

代码4-1 sessionhello

```
import tensorflow as tf
hello = tf.constant('Hello, TensorFlow!') # 定义一个常量
sess = tf.Session() # 建立一个session
print (sess.run(hello)) # 通过session里面run
sess.close() # 关闭session
```

运行代码4-1会得到如下输出：

```
b'Hello, TensorFlow!'
```

`tf.constant`定义的是一个常量，`hello`的内容只有在`session`的`run`内才可以返回。

可以试着在2和3行之间加入`print (hello)`看一下效果，这时并不能输出`hello`的内容。

接下来换种写法，使用`with`语法来开启`session`。

4.1.3 实例6：演示`with session`的使用

`with session`的用法是最常见的，它沿用了Python中`with`的语法，即当程序结束后会自动关闭`session`，而不需要再去写`close`。代码如下。

实例描述

使用`with session`方法建立`session`，并在`session`中计算两个变量（3和4）的相加与相乘值。

代码4-2 `with session`

```
import tensorflow as tf
a = tf.constant(3)                      # 定义常量3
b = tf.constant(4)                      # 定义常量4
with tf.Session() as sess:               # 建立session
    print ("相加: %i" % sess.run(a+b))
    print("相乘: %i" % sess.run(a*b))
```

运行后得到如下输出：

```
相加: 7  
相乘: 12
```

4.1.4 实例7：演示注入机制

扩展上面代码：使用注入机制，将具体的实参注入到相应的placeholder中。feed只在调用它的方法内有效，方法结束后feed就会消失。

实例描述

定义占位符，使用feed机制将具体数值（3和4）通过占位符传入，并进行相加和相乘运算。

代码4-3 withsessionfeed

```
01 import tensorflow as tf  
02 a = tf.placeholder(tf.int16)  
03 b = tf.placeholder(tf.int16)  
04 add = tf.add(a, b)  
05 mul = tf.multiply(a, b) #a与b相乘  
06 with tf.Session() as sess:  
07     #计算具体数值  
08     print ("相加: %i" % sess.run(add, feed_dict={a: 3, b  
09     print ("相乘: %i" % sess.run(mul, feed_dict={a: 3, b
```

运行代码，输出如下：

相加:7
相乘:12

标记的方法是：使用tf.placeholder为这些操作创建占位符，然后使用feed_dict把具体的值放到占位符里。



注意： 关于feed中的feed_dict还有其他的方法，如update等，在后面的例子中用到时还会介绍，这里只是介绍最常用的方法。

4.1.5 建立session 的其他方法

建立session还有以下两种方式。

· 交互式session方式：一般在Jupyter环境下使用较多，具体用法与前面的with session类似。代码如下：

```
sess = tf.InteractiveSession()
```

· 使用Supervisor方式：该方式会更高级一些，使用起来也更加复杂，可以自动来管理session中的具体任务，例如，载入/载出检查点文件、写入TensorBoard等，另外该方法还支持分布式训练的部署（在本书的后面会有介绍）。

4.1.6 实例8：使用注入机制获取节点

在实例7中，其实还可以一次将多个节点取出来。例如，在最后一句可以加上以下代码（见代码4-3）：

实例描述

使用fetch机制将定义在图中的节点数值算出来。

代码4-3 withsessionfeed（续）

```
10 .....  
11 mul = tf.multiply(a, b)  
12 with tf.Session() as sess:  
13     #将op运算通过run打印出来  
14     print ("相加: %i" % sess.run(add, feed_dict={a: 3, b  
15         #将add节点打印出来  
16         print ("相乘: %i" % sess.run(mul, feed_dict={a: 3, b  
16         print (sess.run([mul, add], feed_dict={a: 3, b: 4}))
```

运行代码，输出如下：

```
相加: 7  
相乘: 12  
[12, 7]
```

4.1.7 指定GPU运算

如果下载的是GPU版本，在运行过程中TensorFlow能自动检测。如果检测到GPU，TensorFlow会尽可能地利用找到的第一个GPU来执行操作。

如果机器上有超过一个可用的GPU，除第一个之外的其他GPU默认是不参与计算的。为了让TensorFlow使用这些GPU，必须将OP明确指派给它们执行。with.....device语句能用来指派特定的CPU或GPU执行操作：

```
with tf.Session() as sess:  
    with tf.device("/gpu:1"):  
        a = tf.placeholder(tf.int16)  
        b = tf.placeholder(tf.int16)  
        add = tf.add(a, b)  
        ....
```

设备用字符串进行标识。目前支持的设备包括以下几种。

- cpu: 0: 机器的CPU。
- gpu: 0: 机器的第一个GPU，如果有的话。
- gpu: 1: 机器的第二个GPU，依此类推。

类似的还有通过tf.ConfigProto来构建一个config，在config中指定相关的GPU，并且在

session中传入参数config="自己创建的config"来指定GPU操作。

#tf.ConfigProto函数的参数如下。

- log_device_placement=True: 是否打印设备分配日志。
- allow_soft_placement=True: 如果指定的设备不存在，允许TF自动分配设备。

使用举例：

```
config = tf.ConfigProto(log_device_placement=True, allow_soft_placement=True)
session = tf.Session(config=config, ...)
```

4.1.8 设置GPU使用资源

上文的tf.ConfigProto函数生成config之后，还可以设置其属性来分配GPU的运算资源。如下代码就是按需分配的意思：

```
config.gpu_options.allow_growth = True
```

使用allow_growth option，刚开始会分配少量的GPU容量，然后按需慢慢地增加，由于不会释放内存，所以会导致碎片。

*****ebook converter DEMO Watermarks*****

同样，上述代码也可以放在config创建的时候指定，例如：

```
gpu_options = tf.GPUOptions(allow_growth=True)
config=tf.ConfigProto(gpu_options=gpu_options)
```

以下代码还可以给GPU分配固定大小的计算资源。

```
gpu_options = tf.GPUOptions(per_process_gpu_memory_fraction=
```

代表分配给tensorflow的GPU显存大小为：
GPU实际显存×0.7。

(该方法暂时用不到，读者在以后遇到这样的代码时明白是什么意思即可)

4.1.9 保存和载入模型的方法介绍

一般而言，训练好的模型都需要保存。下面将举例演示如何保存和载入模型。

1. 保存模型

首先需要建立一个saver，然后在session中通过saver的save即可将模型保存起来。代码如下：

```
#之前是各种构建模型graph的操作(矩阵相乘, sigmoid等)
saver = tf.train.Saver() #生成save
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer()) #先对模型初始化
    #然后将数据丢入模型进行训练blablabla
    #训练完以后, 使用saver.save 来保存
    saver.save(sess, "save_path/file_name")
#file_name如果不存在, 会自动创建
```

2. 载入模型

将模型保存好以后，载入也比较方便。在 session 中通过调用 saver 的 restore () 函数，会从指定的路径找到模型文件，并覆盖到相关参数中。代码如下：

```
saver = tf.train.Saver()

with tf.Session() as sess:
    #参数可以进行初始化, 也可不进行初始化。即使初始化了, 初始化的值也会
    #值给覆盖
    sess.run(tf.global_variables_initializer())
    saver.restore(sess, "save_path/file_name") #会将已经保存的
```

4.1.10 实例9：保存/载入线性回归模型

实例描述

在代码“3-1线性回归.py”文件的基础上，添加模型的保存及载入功能。

通过扩展上一章的例子，来演示一下模型的保存及载入。在代码“3-1线性回归.py”文件中生成*****ebook converter DEMO Watermarks*****

模拟数据之后，加入对图变量的重置，在session创建之前定义saver及保存路径，在session中训练结束后，保存模型。

代码4-4 线性回归模型保存及载入

```
01 import tensorflow as tf
02 import numpy as np
03 import matplotlib.pyplot as plt
04
05 #模拟数据
06 .....
07 plt.plot(train_X, train_Y, 'ro', label='Original data')
08 plt.legend()
09 plt.show()
10
11 #重置图
12 tf.reset_default_graph()
13
14 #初始化等操作
15 .....
16 display_step = 2
17
18 saver = tf.train.Saver() #生成模型
19 savedir = "log/" #生成文件夹
20
21 #启动session
22 with tf.Session() as sess:
23     sess.run(init)
24     #在这里添加Sess中的训练代码
25     .....
26     print (" Finished!")
27     saver.save(sess, savedir+"linermodel.cpkt") #保存模型
28     print ("cost=", sess.run(cost, feed_dict=
29         {X: train_X, Y: train_Y}),
30         "W=", sess.run(W), "b=", sess.run(b))
31 #其他代码
32 .....
```

运行上面代码可以看到，在代码的同级目录下log文件夹里生成了几个文件，如图4-2所示。

*****ebook converter DEMO Watermarks*****

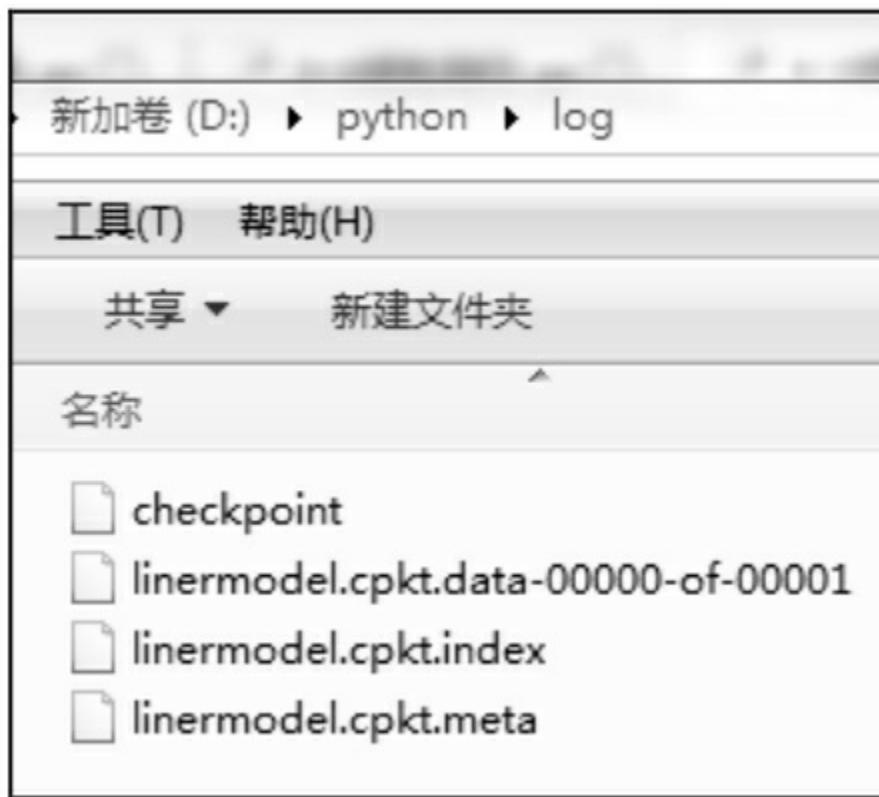


图4-2 模型文件

再重启一个session，并命名为sess2，在代码里通过使用saver的restore函数将模型载入。

代码4-4 线性回归模型保存及载入（续）

```
31 with tf.Session()as sess2:  
32     sess2.run(tf.global_variables_initializer())  
33     saver.restore(sess2,savedir+"linermodel.  
            cpkt")  
34     print ("x=0.2, z=", sess2.run(z, feed_dict={X: 0.2}))
```

为了测试效果，可以将前面一个session注释掉，运行之后可以看到如下输出：

```
INFO:tensorflow:Restoring parameters from log/linermodel.cp  
*****ebook converter DEMO Watermarks*****
```

```
x=0.2, z= [ 0.42615247]
```

表明模型已经成功载入，并计算出正确的值了。

4.1.11 实例10：分析模型内容，演示模型的其他保存方法

下面再来详细介绍下关于模型保存的其他细节。

实例描述

将4.1.10节生成的模型里面的内容打印出来，观察其存放的具体数据方式。同时演示如何将指定内容保存到模型文件中。

1. 模型内容

虽然模型已经保存了，但是仍然对我们不透明。下面通过编写代码将模型里的内容打印出来，看看到底保存了哪些东西，都是什么样的。

代码4-5 模型内容

```
01 from tensorflow.python.tools.inspect_checkpoint import *
02     in_checkpoint_file
03 savedir = "log/"
04 print_tensors_in_checkpoint_file(savedir+"linermode1.cpt"
05                                     , tensor_name="",
06                                     , all_tensors=True)
```

运行代码， 打印如下信息：

```
tensor_name: bias  
[ 0.01919404]  
tensor_name: weight  
[ 2.03479218]
```

可以看到， tensor_name： 后面跟的就是创建的变量名， 接着是它的数值。

2. 保存模型的其他方法

前面的例子中Saver的创建比较简单， 其实tf.train.Saver函数里面还可以放参数来实现更高级的功能， 可以指定存储变量名字与变量的对应关系。可以写成这样：

```
saver = tf.train.Saver({'weight': w, 'bias': b})
```

代表将w变量的值放到weight名字中。类似的写法还有以下两种：

```
saver = tf.train.Saver([w, b]) #放到一个list  
saver = tf.train.Saver({v.op.name: v for v in [w, b]}) #将op
```

下面扩展上述的例子， 给b和w分别指定一个固定值，并将它们颠倒放置。

代码4-5 模型内容（续）

```
03 W = tf.Variable(1.0, name="weight")
04 b = tf.Variable(2.0, name="bias")
05
06 #放到一个字典里
07 saver = tf.train.Saver({'weight': b, 'bias': W})
08
09 with tf.Session() as sess:
10     tf.global_variables_initializer().run()
11     saver.save(sess, savedir+"linermodel.cpkt")
12
13 print_tensors_in_checkpoint_file(savedir+"linermodel.cpkt", True)
```

运行上面代码，输出如下信息：

```
tensor_name: bias
1.0
tensor_name: weight
2.0
```

例子中，W值设为1.0，b的值设为2.0。在创建saver时将它们颠倒，保存的模型打印出来之后可以看到，bias变成了1.0，而weight变成了2.0。

4.1.12 检查点（Checkpoint）

保存模型并不限于在训练之后，在训练之中也需要保存，因为TensorFlow训练模型时难免会出现中断的情况。我们自然希望能够将辛苦得到的中间参数保留下来，否则下次又要重新开始。

这种在训练中保存模型，习惯上称之为保存检查点。

4.1.13 实例11：为模型添加保存检查点

实例描述

为一个线性回归任务的模型添加“保存检查点”功能。通过该功能，可以生成载入检查点文件，并能够指定生成检测点文件的个数。

该例与保存模型的功能类似，只是保存的位置发生了些变化，我们希望在显示信息时将检查点保存起来，于是就将保存位置放在了迭代训练中的打印信息后面。

另外，本例用到了saver的另一个参数——max_to_keep=1，表明最多只保存一个检查点文件。在保存时使用了如下代码传入了迭代次数。

```
saver.save(sess, savedir+"linermodel.cpkt", global_step=epoch)
```

TensorFlow会将迭代次数一起放在检查点的名字上，所以在载入时，同样也要指定迭代次数。

```
saver.restore(sess2, savedir+"linermodel.cpkt-" + str(load_epoch))
```

完整的代码如下：

代码4-6 保存检查点

```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt

# 定义生成loss可视化的函数
plotdata = { "batchsize":[], "loss":[] }
def moving_average(a, w=10):
    if len(a) < w:
        return a[:]
    return [val if idx < w else sum(a[(idx-w):idx])/w for idx, val in enumerate(a)]

# 生成模拟数据
train_X = np.linspace(-1, 1, 100)
train_Y = 2 * train_X + np.random.randn(*train_X.shape) * 0.1
# 图形显示
plt.plot(train_X, train_Y, 'ro', label='Original data')
plt.legend()
plt.show()

tf.reset_default_graph()

# 创建模型
# 占位符
X = tf.placeholder("float")
Y = tf.placeholder("float")
# 模型参数
W = tf.Variable(tf.random_normal([1]), name="weight")
b = tf.Variable(tf.zeros([1]), name="bias")
# 前向结构
z = tf.multiply(X, W)+ b

# 反向优化
cost =tf.reduce_mean( tf.square(Y - z))
learning_rate = 0.01
optimizer = tf.train.GradientDescentOptimizer(learning_rate)
(cost) # 梯度下降

# 初始化所有变量
init = tf.global_variables_initializer()
# 定义学习参数
```

*****ebook converter DEMO Watermarks*****

```

training_epochs = 20
display_step = 2
saver = tf.train.Saver(max_to_keep=1) # 生成saver
savedir = "log/"
# 启动图
with tf.Session() as sess:
    sess.run(init)

    # 向模型中输入数据
    for epoch in range(training_epochs):
        for (x, y) in zip(train_X, train_Y):
            sess.run(optimizer, feed_dict={X: x, Y: y})

    #显示训练中的详细信息
    if epoch % display_step == 0:
        loss = sess.run(cost, feed_dict={X: train_X, Y:train_Y})
        print ("Epoch:",epoch+1,"cost=",loss,"W=",sess.run(W))
        sess.run(b)
        if not (loss == "NA" ):
            plotdata["batchsize"].append(epoch)
            plotdata["loss"].append(loss)
        saver.save(sess, savedir+"linermode1.cpkt", global_step=epoch)

    print (" Finished!")

    print ("cost=",sess.run(cost, feed_dict={X: train_X, Y:train_Y}),
           "W=", sess.run(W), "b=", sess.run(b))

    #显示模型
    plt.plot(train_X, train_Y, 'ro', label='Original data')
    plt.plot(train_X, sess.run(W) * train_X + sess.run(b), 'b-',
             label='Wline')
    plt.legend()
    plt.show()

    plotdata["avgloss"] = moving_average(plotdata["loss"])
    plt.figure(1)
    plt.subplot(211)
    plt.plot(plotdata["batchsize"], plotdata["avgloss"], 'b-')
    plt.xlabel('Minibatch number')
    plt.ylabel('Loss')
    plt.title('Minibatch run vs. Training loss')

    plt.show()

#重启一个session , 载入检查点
load_epoch=18
with tf.Session() as sess2:
    sess2.run(tf.global_variables_initializer())
*****ebook converter DEMO Watermarks*****

```

```
saver.restore(sess2, savedir+"linermodel.cpkt-" + str(10))
print ("x=0.2, z=", sess2.run(z, feed_dict={X: 0.2}))
```

上面代码运行完后，会看到在log文件夹下多了几个linermodel.cpkt-18*文件，就是检查点文件。

这里使用tf.train.Saver（max_to_keep=1）代码创建saver时传入的参数max_to_keep=1代表：在迭代过程中只保存一个文件。这样，在循环训练过程中，新生成的模型就会覆盖以前的模型。



注意：如果觉得通过指定迭代次数比较麻烦，还有一个好方法可以快速获取到检查点文件。示例代码如下：

```
ckpt = tf.train.get_checkpoint_state(ckpt_dir)
if ckpt and ckpt.model_checkpoint_path:
    saver.restore(sess, ckpt.model_checkpoint_path)
```

还可以再简洁一些，写成以下这样：

```
kpt = tf.train.latest_checkpoint(savedir)
if kpt!=None:
    saver.restore(sess, kpt)
```

4.1.14 实例12：更简便地保存检查点

本例中介绍另一种更简便地保存检查点功能代码的方法——tf.train.MonitoredTrainingSession函数。该函数可以直接实现保存及载入检查点模型的文件。与前面的方式不同，本例中并不是按照循环步数来保存，而是按照训练时间来保存的。通过指定save_checkpoint_secs参数的具体秒数，来设置每训练多久保存一次检查点。

实例描述

演示使用MonitoredTrainingSession函数来自动管理检查点文件。

具体代码如下：

代码4-7 trainMonitored

```
import tensorflow as tf
tf.reset_default_graph()
global_step = tf.train.get_or_create_global_step()
step = tf.assign_add(global_step, 1)
#设置检查点路径为log/checkpoints
with tf.train.MonitoredTrainingSession(checkpoint_dir='log/c
    print(sess.run([global_step]))
    while not sess.should_stop(): #启用死循环，当sess不结束时就
        i = sess.run( step)
        print( i)
```

运行代码，得到如下输出：

```
252 12851
252 12852
```

```
252 12853  
252 12854  
252 12855  
252 12856
```

将程序停止，可以看到log/checkpoints下面生成了检测点文件model.ckpt-8968.meta。再次运行代码，输出如下：

```
252 8969  
252 8970  
252 8971
```

可见，程序自动载入检查点文件是从第8969次开始运行的。



注意： (1) 如果不设置 save_checkpoint_secs 参数，默认的保存时间间隔为 10 分钟。这种按照时间保存的模式更适用于使用大型数据集来训练复杂模型的情况。

(2) 使用该方法时，必须要定义 global_step 变量，否则会报错误。

4.1.15 模型操作常用函数总结

下面将模型操作的相关函数进行系统的介绍，如表4-2所示。

表4-2 模型操作相关函数

函 数	说 明
tf.train.Saver (var_list=None, reshape=False, sharded=False, max_to_keep=5, keep_checkpoint_every_n_hours=10000.0, name=None, restore_sequentially=False, saver_def=None, builder=None)	创建存储器Saver
tf.train.Saver.save(sess, save_path, global_step=None, latest_filename=None, meta_graph_suffix='meta', write_meta_graph=True)	保存
tf.train.Saver.restore(sess, save_path)	恢复
tf.train.Saver.last_checkpoints	列出最近未删除的checkpoint文件名
tf.train.Saver.set_last_checkpoints(last_checkpoints)	设置checkpoint文件名列表
tf.train.Saver.set_last_checkpoints_with_time(last_checkpoints_with_time)	设置checkpoint文件名列表和时间戳

函 数	描 述
tf.ones_like (input)	生成和输入张量一样形状和类型的0。例如: tensor=[[1, 2, 3], [4, 5, 6]] tf.ones_like (tensor) ==> [[0 0 0] [0 0 0]]
tf.zeros_like (input)	生成和输入张量一样形状和类型的1。例如: tensor=[[1, 2, 3], [4, 5, 6]] tf.zeros_like (tensor) ==> [[0 0 0] [0 0 0]]
tf.fill(shape,value)	为指定形状填值。例如: tf.fill([2,3],1) ==> [[1 1 1] [1 1 1]]
tf.constant(value, shape)	生成常量。例如: tf.constant(1,[2,3]) ==> [[1 1 1] [1 1 1]]
tf.random_normal(shape, mean=0.0, stddev=1.0, dtype=tf.float32, seed=None, name=None)	正太分布随机数，均值mean,标准差stddev
tf.truncated_normal(shape, mean=0.0, stddev=1.0, dtype=tf.float32, seed=None, name=None)	截断正态分布随机数，均值mean,标准差stddev,只保留[mean-2*stddev,mean+2*stddev]范围内的随机数
tf.random_uniform(shape,minval=0, maxval= None , dtype=tf.float32, seed=None, name=None)	均匀分布随机数，范围为[minval,maxval]
tf.random_crop(value, size, seed=None, name=None)	将输入值value按照size尺寸随机剪辑
tf.set_random_seed(seed)	设置随机数种子
tf.linspace(start,stop,num,name=None)	在[start,stop]范围内产生num个数的等差数列。注意，start和stop要用浮点数表示，否则会报错。例如： tf.linspace(start=1.0,stop=5.0,num=5,name=None) [1. 2. 3. 4. 5.]
tf.range(start,limit=None,delta=1,name='range')	在[start,limit)范围内以步进值delta产生等差数列。注意，不包括limit在内的。例如： tf.range(start=1,limit=5,delta=1) [1 2 3 4]

4.1.16 TensorBoard可视化介绍

TensorFlow还提供了一个可视化工具TensorBoard。它可以将训练过程中的各种绘制数据展示出来，包括标量（Scalars）、图片（Images）、音频（Audio）、计算图（Graph）、数据分布、直方图（Histograms）和嵌入式向量。可以通过网页来观察模型的结构和训练过程中各个参数的变化。

当然，TensorBoard不会自动把代码展示出来，其实它是一个日志展示系统，需要在session中运算图时，将各种类型的数据汇总并输出到日志文件中。然后启动TensorBoard服务，TensorBoard读取这些日志文件，并开启6006端口提供Web服务，让用户可以在浏览器中查看数据。

TensorFlow提供了一系列API来生成这些数据，具体如表4-3所示。

表4-3 模型操作相关函数

函 数	说 明
tf.summary.scalar(tags, values, collections=None, name=None)	标量数据汇总，输出protobuf
tf.summary.histogram(tag, values, collections=None, name=None)	记录变量var的直方图，输出带直方图的汇总的protobuf
tf.summary.image(tag, tensor, max_images=3, collections=None, name=None)	图像数据汇总，输出protobuf
tf.summary.merge(inputs, collections=None, name=None)	合并所有的汇总日志
tf.summary.FileWriter	创建一个SummaryWriter
Class SummaryWriter: add_summary(), add_sessionlog(), add_event(), or add_graph()	将protobuf写入文件的类

4.1.17 实例13：线性回归的TensorBoard可视化

下面举例演示TensorBoard的可视化效果。

实例描述

为“3-1线性回归.py”代码文件添加支持输出TensorBoard信息的功能，演示通过TensorBoard来

观察训练过程。

本例还是以“3-1线性回归.py”文件的代码为原型，在上面添加支持TensorBoard的功能。该例子中，通过添加一个标量数据和一个直方图数据到log里，然后通过TensorBoard显示出来。代码改动量非常小，第一步加入到summary，第二步写入文件。

将模型的生成值加入到直方图数据中，将损失值加入到标量数据中，代码如下：

代码4-8 线性回归的TensorBoard可视化

```
01 import tensorflow as tf
02 import numpy as np
03 import matplotlib.pyplot as plt
04
05 .....
06 # 前向结构
07 z = tf.multiply(X, w)+ b
08 tf.summary.histogram('z', z) #将预测值以直
09 #反向优化
10 cost =tf.reduce_mean( tf.square(Y - z))
11 tf.summary.scalar('loss_function', cost) #将损失以标量形
12 .....
```

给直方图起名仍然叫z，标量的名字叫
loss_function。

下面的代码是在启动session之后加入代码，
创建一个summary_writer，在迭代中将summary的

值运行生成出来，同时添加到文件里。

代码4-8 线性回归的TensorBoard可视化 (续)

```
23 # 启动session
24 with tf.Session() as sess:
25     sess.run(init)
26
27     merged_summary_op = tf.summary.merge_all()#合并所有sur
28     #创建summary_writer, 用于写文件
29     summary_writer =
30     tf.summary.FileWriter('log/mnist_with_summaries', sess.gi
31
32     # 向模型中输入数据
33     for epoch in range(training_epochs):
34         for(x, y)in zip(train_X, train_Y):
35             sess.run(optimizer, feed_dict={X: x, Y: y})
36
37     #生成summary
38     summary_str = sess.run(merged_summary_op, feed_dict=)
39     summary_writer.add_summary(summary_str, epoch);#将su
40
41     .....
```

运行代码，显示的内容和以前一样没什么变化，来到生成的路径下可以看到多了一个文件，如图4-3所示。

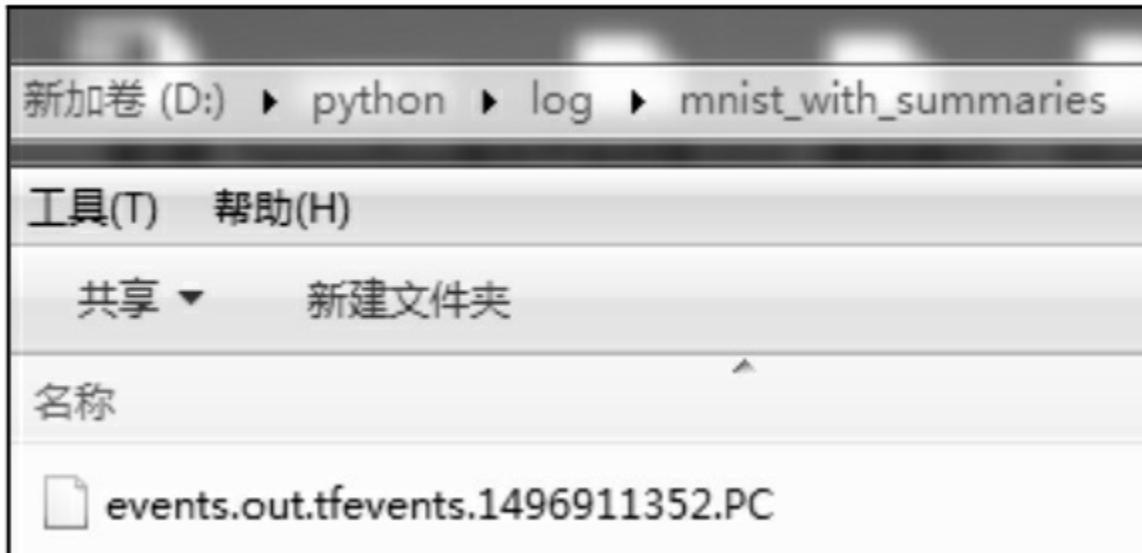


图4-3 summary文件

然后单击“开始”|“运行”，输入cmd，启动“命令行”窗口。首先来到summary日志的上级路径下，输入如下命令：

```
tensorboard --logdir D:\python\log\mnist_with_summaries
```

结果如图4-4所示。

```
C:\Users\Administrator>d:  
D:>cd python  
D:\python>cd log  
D:\python\log>tensorboard --logdir D:\python\log\mnist_with_summaries  
Starting TensorBoard b'54' at http://PC:6006  
(Press CTRL+C to quit)
```

图4-4 启动TensorBoard

接着打开Chrome浏览器，输入<http://127.0.0.1:6006>，会看到如图4-5所示界

*****ebook converter DEMO Watermarks*****

面。单击SCALARS，会看到之前创建的loss_fuction。这个loss_fuction也是可以点开的，点开后可以看到损失值随迭代次数的变化情况，如图4-6所示。

在图4-6中可以调节平滑数来改变右边标量的曲线。类似的还可以点开图4-5中的GRAPHS看看神经网络的内部结构，还可以点开图4-5中的HISTOGRAMS来看例子中的另一个显示值z。

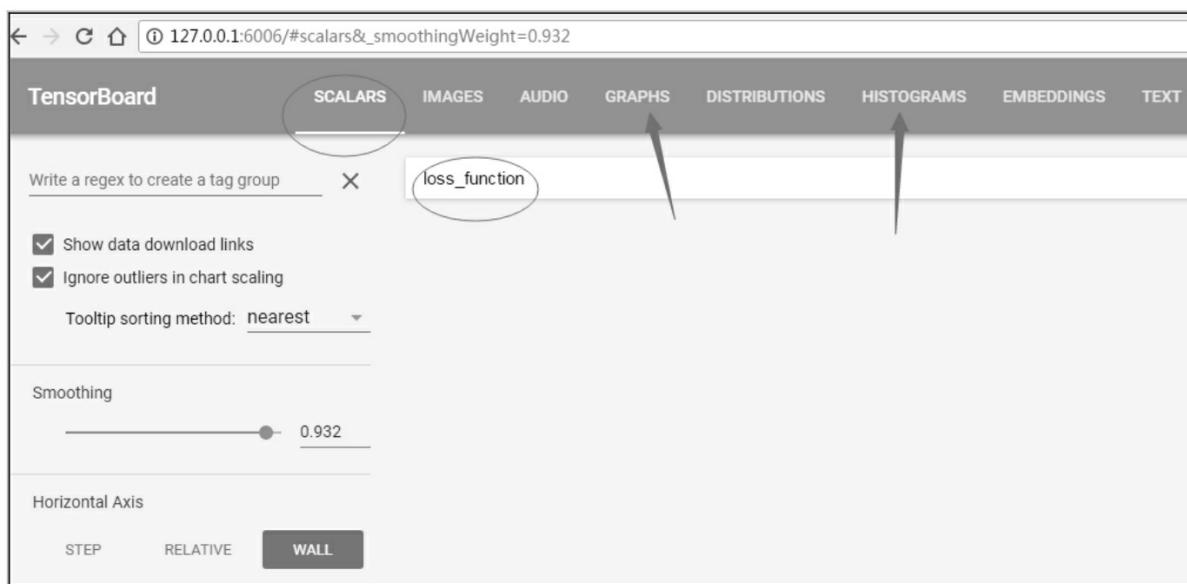


图4-5 TensorBoard界面

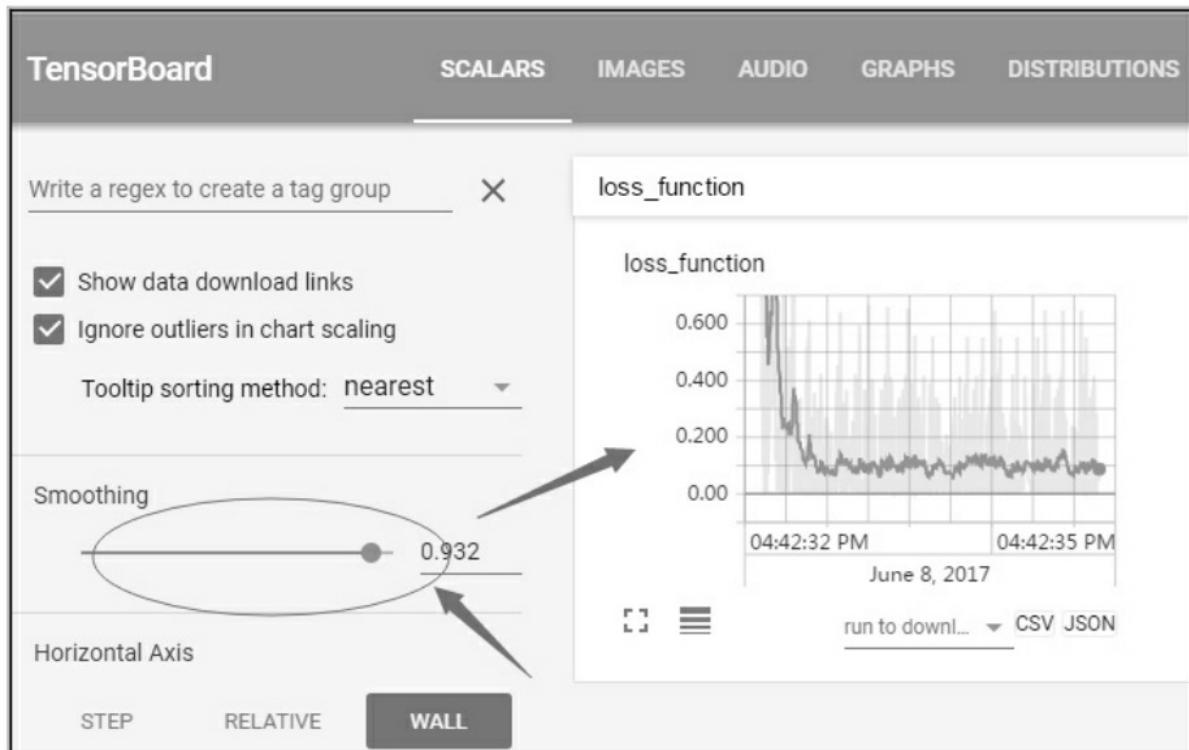


图 4-6 TensorBoard 标量

 注意：在显示TensorBoard界面的过程中，下面两点需要强调一下。

- 浏览器最好要使用Chrome。
- 在命令行里启动TensorBoard时，一定要先进入到日志所在的上级路径下，否则打开的页面里找不到创建好的信息。

4.2 TensorFlow基础类型定义及操作函数介绍

下面介绍TensorFlow的基础类型、基础函数。这部分学完，读者将会对TensorFlow的基础语法有了系统的了解，为后面学习写代码或读代码扫清障碍。

本节表格中的示例代码前面默认都有以下代码。

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

代码中的tf代表tensorflow库，np代表numpy库。

4.2.1 张量及操作

张量可以说是TensorFlow的标志，因为整个框架的名称TensorFlow就是张量流的意思。下面来一起全面地认识一下张量。

1. 张量介绍

TensorFlow程序使用tensor数据结构来代表所

有的数据。计算图中，操作间传递的数据都是Tensor。

可以把tensor看为一个n维的数组或列表，每个tensor中包含了类型（type）、阶（rank）和形状（shape）。

(1) tensor类型

为了方便理解，这里将tensor的类型与Python的类型放在一起做个比较，如表4-4所示。

表4-4 张量类型

tensor类型	Python类型	描述
DT_FLOAT	tf.float32	32位浮点数
DT_DOUBLE	tf.float64	64位浮点数
DT_INT64	tf.int64	64位有符号整型
DT_INT32	tf.int32	32位有符号整型
DT_INT16	tf.int16	16位有符号整型
DT_INT8	tf.int8	8位有符号整型
DT_UINT8	tf.uint8	8位无符号整型
DT_STRING	tf.string	可变长度的字节数组.每一个张量元素都是一个字节数组
DT_BOOL	tf.bool	布尔型
DT_COMPLEX64	tf.complex64	由两个32位浮点数组成的复数:实数和虚数

(2) rank (阶)

rank (阶) 指的就是维度。但张量的阶和矩阵的阶并不是同一个概念，主要是看有几层中括号。例如，对于一个传统意义上的3阶矩阵a=[[1, 2, 3], [4, 5, 6], [7, 8, 9]]来讲，在张量

中的阶数表示为2阶（因为它有两层中括号）。

表4-5列出了标量、向量、矩阵的阶数。

表4-5 标量向量和矩阵的阶数

rank	实 例	例 子
0	标量(只有大小)	$a = 1$
1	向量(大小和方向)	$b = [1,1,1,1]$
2	矩阵(数据表)	$C = [[1,1],[1,1]]$
3	3阶张量(数据立体)	$D = [[[1],[1]],[[1],[1]]]$
n	n阶	$E = [[[[[...[[1],[1]],...]]]] \quad (n\text{层中括号})$

(3) shape (形状)

shape (形状) 用于描述张量内部的组织关系。“形状”可以通过Python中的整数列表或元组(int list或tuples) 来表示，也可以用TensorFlow中的相关形状函数来表示。

举例：一个二阶张量 $a=[[1, 2, 3], [4, 5, 6]]$ 形状是两行三列，描述为(2, 3)。

2. 张量相关操作

张量的相关操作包括类型转换、数值操作、形状变换和数据操作。

(1) 类型转换

类型转换的相关函数如表4-6所示。

表4-6 类型变换相关函数

函 数	描 述
tf.string_to_number(string_tensor,out_type=None, name=None)	字符串转为数字
tf.to_double(x, name='.ToDouble')	转为64位浮点类型
tf.to_float(x, name='ToFloat')	转为32位浮点类型
tf.to_int32(x, name='ToInt32')	转为32位整型
tf.to_int64(x, name='ToInt64')	转为64位整型
tf.cast(x,dtype,name=None)	将x或者x.values转换为dtype所指定的类型。例如： W = tf.Variable(1.0) tf.cast(W, tf.int32) ==> W=1 # dtype=tf.int32

(2) 数值操作

数值操作的相关函数如表4-7所示。

表4-7 类型变换相关函数

函 数	描 述
tf.ones(shape,dtype)	按指定类型与形状生成值为1的张量。例如： tf.ones([2, 3], tf.int32)==> [[1 1 1] [1 1 1]]
tf.zeros(shape,dtype)	按指定类型与形状生成值为0的张量。例如： tf.zeros([2, 3], tf.int32)==> [[0 0 0] [0 0 0]]

(续)

函 数	描 述
tf.ones_like (input)	生成和输入张量一样形状和类型的0。例如: tensor=[[1, 2, 3], [4, 5, 6]] tf.ones_like (tensor) ==> [[0 0 0] [0 0 0]]
tf.zeros_like (input)	生成和输入张量一样形状和类型的1。例如: tensor=[[1, 2, 3], [4, 5, 6]] tf.zeros_like (tensor) ==> [[0 0 0] [0 0 0]]
tf.fill(shape,value)	为指定形状填值。例如: tf.fill([2,3],1) ==> [[1 1 1] [1 1 1]]
tf.constant(value, shape)	生成常量。例如: tf.constant(1,[2,3]) ==> [[1 1 1] [1 1 1]]
tf.random_normal(shape, mean=0.0, stddev=1.0, dtype=tf.float32, seed=None, name=None)	正太分布随机数，均值mean,标准差stddev
tf.truncated_normal(shape, mean=0.0, stddev=1.0, dtype=tf.float32, seed=None, name=None)	截断正态分布随机数，均值mean,标准差stddev,只保留[mean-2*stddev,mean+2*stddev]范围内的随机数
tf.random_uniform(shape,minval=0, maxval= None , dtype=tf.float32, seed=None, name=None)	均匀分布随机数，范围为[minval,maxval]
tf.random_crop(value, size, seed=None, name=None)	将输入值value按照size尺寸随机剪辑
tf.set_random_seed(seed)	设置随机数种子
tf.linspace(start,stop,num,name=None)	在[start,stop]范围内产生num个数的等差数列。注意，start和stop要用浮点数表示，否则会报错。例如： tf.linspace(start=1.0,stop=5.0,num=5,name=None) [1. 2. 3. 4. 5.]
tf.range(start,limit=None,delta=1,name='range')	在[start,limit)范围内以步进值delta产生等差数列。 注意，不包括limit在内的。例如： tf.range(start=1,limit=5,delta=1) [1 2 3 4]

(3) 形状变换

形状变换的相关函数如表4-8所示。

表4-8 形状变换的相关函数

函 数	描 述
tf.shape(input, name=None)	返回一个张量，其值为输入参数input的shape。这个input可以是个张量，也可以是一个数组或list。例如： t=[1,2,3,4,5,6,7,8,9] print(np.shape(t)) #输出 (9,) tshape = tf.shape(t) #返回一个张量，值为python自有类型t的shape tshape2 = tf.shape(tshape) #返回一个张量，值为张量tshape的shape sess = tf.Session()

*****ebook converter DEMO Watermarks*****

(续)

函 数	描 述
tf.shape(input, name=None)	<pre>print(sess.run(tshape)) #输出[9], 表示t的shape的值 print(sess.run(tshape2)) #输出[1], 表示tshape的shape的值 t=[[1, 1, 1], [2, 2, 2], [[3, 3, 3], [4, 4, 4]]]</pre>
tf.size(input, name=None)	<p>返回一个张量，其内容为输入数据的元素数量。例如：</p> <pre>t=[[[1, 1, 1], [2, 2, 2]], [[3, 3, 3], [4, 4, 4]]] sizet = tf.size(t) sess = tf.Session() print(sess.run(sizet)) #输出12, 表示列表t中的元素个数</pre>
tf.rank(input, name=None)	<p>返回一个张量，其内容为输入数据input的rank。注意，此rank不同于矩阵的rank，详见4.2.1节中的rank介绍。例如：</p> <pre>t=[[[1, 1, 1], [2, 2, 2]], [[3, 3, 3], [4, 4, 4]]] rankt = tf.rank(t) sess = tf.Session() print(sess.run(rankt)) #输出4, 表示列表t的阶(一共有4层中括号, [[[[]]])</pre>
tf.reshape(input, shape, name=None)	<p>将原有输入数据的shape按照指定形状进行变化，生成一个新的张量。例如： t=[1,2,3,4,5,6,7,8,9]</p> <pre>tt=tf.reshape(t,[3,3]) sess = tf.Session() print(sess.run(tt)) #此时输出的张量如下: #[[1, 2, 3], #[4, 5, 6], #[7, 8, 9]]</pre> <p>#如果shape有元素[-1]，则表示在该维度下按照原有数据自动计算。见下面的代码：</p> <pre>ttt=tf.reshape(tt, [1, -1]) print(ttt.shape) #输出 (1,9) 表示ttt的shape, 9是自动计算得来的 print(tt.shape) #输出 (3,3) 表示tt并没有被修改</pre>
tf.expand_dims(input, dim, name=None)	<p>插入维度1进入一个tensor中。例如：</p> <pre>t=[[2,3,3],[1,5,5]] t1 = tf.expand_dims(t, 0) t2 = tf.expand_dims(t, 1) t3 = tf.expand_dims(t, 2) t4 = tf.expand_dims(t, -1) #如果写成t4 = tf.expand_dims(t, 3)，则会出错，因为只有两个维度 print(np.shape(t)) # 输出(2, 3) print(np.shape(t1)) # 输出(1, 2, 3) print(np.shape(t2)) # 输出(2, 1, 3) print(np.shape(t3)) # 输出(2, 3, 1) print(np.shape(t4)) # 输出(2, 3, 1)</pre>

*****ebook converter DEMO Watermarks*****

(续)

函 数	描 述
tf.squeeze(input, dim, name=None)	将dim指定的维度去掉 (dim所指定的维度必须为1, 如果不为1则会报错)。例如: t = [[[2],[1]]] t1 = tf.squeeze(t, 0) t2 = tf.squeeze(t, 1) t3 = tf.squeeze(t, 3) t4 = tf.squeeze(t, -1) #如果写成t4 = tf.squeeze(t, 2)会出错, 因为2对应的维度为2, 不为1 print(np.shape(t)) #(1, 1, 2, 1) print(np.shape(t1)) #(1, 2, 1) print(np.shape(t2)) #(1, 2, 1) print(np.shape(t3)) #(1, 1, 2) print(np.shape(t4)) #(1, 1, 2)

(4) 数据操作

数据操作的相关函数如表4-9所示。

表4-9 数据操作相关函数

函 数	描 述
tf.slice(input, begin, size, name=None)	<p>对输入数据input进行切片操作，begin与size可以为list类型。要求begin与size的值必须一一对应，并且begin中每个值都要大于等于0且小于等于size中对应的值。例如：</p> <pre>t=[[1, 1, 1], [2, 2, 2], [3, 3, 3], [4, 4, 4], [5, 5, 5], [6, 6, 6]] slicet1 = tf.slice(t, [1, 0, 0], [1, 1, 3]) slicet2 = tf.slice(t, [1, 0, 0], [1, 2, 3]) slicet3 = tf.slice(t, [1, 0, 0], [2, 1, 3]) sess = tf.Session(), print(sess.run(slicet1)) #输出 [[3 3 3]] print(sess.run(slicet2)) #输出 [[[3 3 3] [4 4 4]]] print(sess.run(slicet3)) #输出 [[[3 3 3]] [[5 5 5]]]</pre>
tf.split(value, num_or_size_splits, axis=0, num=None, name="split")	<p>沿着某一维度将tensor分离为num_or_size_splits Value是一个shape 为[5, 30]的张量</p> <pre># 沿着第一列将value按[4, 15, 11]分成3个张量 split0, split1, split2 = tf.split(value, [4, 15, 11], 1) tf.shape(split0) ==> [5, 4] tf.shape(split1) ==> [5, 15] tf.shape(split2) ==> [5, 11]</pre>
tf.concat(concat_dim,values, name='concat')	<p>沿着某一维度连接tensor</p> <pre>t1 = [[1, 2, 3], [4, 5, 6]] t2 = [[7, 8, 9], [10, 11, 12]] tf.concat([t1, t2], 0) ==> [[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]]</pre>

*****ebook converter DEMO Watermarks*****

(续)

函数	描述
tf.concat(concat_dim,values, name='concat')	<pre>tf.concat([t1, t2],1) => [[1, 2, 3, 7, 8, 9], [4, 5, 6, 10, 11, 12]] 如果想沿着Tensor一新轴连接打包,则可以: tf.concat(axis, [tf.expand_dims(t, axis) for t in tensors]) 等同于tf.stack(tensors, axis=axis)</pre>
tf.stack(input, axis=0)	<p>将两个N维张量列表沿着axis轴组合成一个沿着axis轴组合成一个N+1维的张量</p> <pre>tensor=[[1, 2, 3], [4, 5, 6]] tensor2=[[10, 20, 30],[40, 50, 60]] tf.stack([tensor, tensor2]) =>[[[1 2 3] [4 5 6]] [[10 20 30][40 50 60]]] tf.stack([tensor, tensor2], axis=1) =>[[[1 2 3][10 20 30]] [[4 5 6][40 50 60]]]</pre>
defunstack(value,num=None, axis=0, name="unstack")	<p>将输入value按照指定的行或列进行拆分，并输出含有num个元素的列表 (list) axis=0表示按行拆分，axis=1表示按列拆分。</p> <p>num为输出list的个数，必须与预计输出的个数相等，否则会报错。可忽略这个参数</p> <pre>tensor=[[1, 2, 3], [4, 5, 6]] tf.unstack(tensor) => [array([1, 2, 3]), array([4, 5, 6])] tf.unstack(tensor, axis=1) =>[array([1, 4]), array([2, 5]), array([3, 6])] #tensor.shape=[2,3],axis=0,就是分成2个。axis=1就是分成3个 #ten2.shape=[2,3,4],axis=2 就是分成4个</pre>
tf.gather(params,indices, validate_indices=None, name=None)	<p>合并索引indices所指示params中的切片</p> <pre>y= tf.constant([0.,2.,-1.]) t = tf.gather(y, [2,0]) sess=tf.Session() t2 = sess.run([t]) print(t2) #输出[array([-1., 0.], dtype=float32)]</pre>
tf.one_hot(indices,depth, on_value= None, off_value =None, axis=None, dtype= None, name=None)	<p>生成符合onehot编码的张量。</p> <ul style="list-style-type: none"> • indices: 要生成的张量。 • depth: 在 depth 长度的数组中, 哪个索引的值为 onehot 值。 • on_value: 为 onehot 值时, 该值为多少。 • off_value: 非 onehot 值时, 该值为多少。 <p>Axis为-1时生成的shape为[indices长度, depth]，为0时shape为[depth, indices长度]。还可以是1，是指在类似时间序列（三维度以上）情况下，以时间序列优先而非batch优先，即[depth,batch,indices长度]（这里的indices长度可以当成样本中的feature特征维度），例如：</p> <pre>indices = [0, 2, -1, 1]</pre>

*****ebook converter DEMO Watermarks*****

(续)

函 数	描 述
tf.one_hot(indices, depth, on_value=None, off_value=None, axis=None, dtype=None, name=None)	depth=3 on_value=5.0 off_value = 0.0 axis = -1 t=tf.one_hot(indices,depth,on_value,off_value,axis) sess = tf.Session() print(sess.run(t)) 则输出如下: [[5. 0. 0.] #0 [0. 0. 5.] #2 [0. 0. 0.] #-1 [0. 5. 0.]] #1
tf.count_nonzero (input_tensor, axis=None, keepdims=False, dtype = dtypes.int64, name = None, reduction_indices=None)	统计非0个数



注意：TensorFlow开头的代码都不能直接运行，必须放到session里面才可以。例如：

```
01 import numpy as np
02 import tensorflow as tf
03
04 x = tf.constant(2)
05 y = tf.constant(5)
06 def f1(): return tf.multiply(x, 17)
07 def f2(): return tf.add(y, 23)
08 r = tf.cond(tf.less(x, y), f1, f2)
09 print(r) #这样是错的
10
11 #生成两行三列的张量，值为1
12 with tf.Session() as sess:
13     print(sess.run( r )) #这样才可以
```

4.2.2 算术运算函数

如表4-10中列出了TensorFlow关于算术运算方面的函数。

表4-10 算术操作

函 数	描 述
<code>tf.assign(x, y, name=None)</code>	令 $x=y$
<code>tf.add(x, y, name=None)</code>	求和
<code>tf.subtract (x, y, name=None)</code>	减法

(续)

函 数	描 述
tf.multiply (x, y, name=None)	乘法
tf.divide (x, y, name=None)	除法,也可以使用tf.div函数
tf.mod(x, y, name=None)	取模
tf.abs(x, name=None)	求绝对值
tf.negative(x, name=None)	取负 ($y = -x$)
tf.sign(x, name=None)	返回输入x的符号。如果x小于0, 则返回-1; 如果x=0, 则返回0; 如果x大于0, 则返回1
tf.inv(x, name=None)	对取反操作
tf.square(x, name=None)	计算平方 ($y = x * x = x^2$)
tf.round(x, name=None)	舍入最接近的整数。例如: a=[0.9, 2.5, 2.3, 1.5, -4.5] tf.round(a) ==> [1.0, 2.0, 2.0, 2.0, -4.0] 如果需要真正的四舍五入, 可以用tf.int32类型强制转换
tf.sqrt(x, name=None)	开根号 ($y = \sqrt{x} = x^{1/2}$).
tf.pow(x, y, name=None)	幂次方计算。例如: x=[[2,2],[3,3]] y=[[8,16],[2,3]] tf.pow(x,y)==> [[256, 65536], [9, 27]] # [2的2次方, 2的16次方], [3的2次方, 3的3次方]
tf.exp(x, name=None)	计算e的次方
tf.log(x, name=None)	计算log, 一个输入计算e的ln, 两输入以第二输入为底
tf.maximum(x, y, name=None)	返回最大值 ($x > y ? x : y$)
tf.minimum(x, y, name=None)	返回最小值 ($x < y ? x : y$)
tf.cos(x, name=None)	三角函数cosine
tf.sin(x, name=None)	三角函数sine
tf.tan(x, name=None)	三角函数tan
tf.atan(x, name=None)	三角函数ctan
tf. cond(pred, true_fn=None, false_fn=None, strict=False, name=None, fn1=None, fn2=None)	满足条件就执行fn1,否则执行fn2。例如: x = tf.constant(2) y = tf.constant(5) def f1(): return tf.multiply(x, 17) def f2(): return tf.add(y, 23) r = tf.cond(tf.less(x, y), f1, f2) 则r的值为34

4.2.3 矩阵相关的运算

矩阵相关的操作函数如表4-11所示。

表4-11 矩阵操作函数

*****ebook converter DEMO Watermarks*****

操作	描述
tf.diag(diagonal, name=None)	返回一个给定对角值的对角tensor。 diagonal = [1, 2, 3, 4] tf.diag(diagonal) 会得到如下矩阵： [[1, 0, 0, 0] [0, 2, 0, 0] [0, 0, 3, 0] [0, 0, 0, 4]]
tf.diag_part(input, name=None)	功能与上面相反
tf.trace(x, name=None)	求一个二维Tensor足迹，即对角值diagonal之和
tf.transpose(a,perm=None, name='transpose')	让输入a按照参数perm指定的维度顺序进行转置操作。如果不设定perm，默认是一个全转置。例如： t = [[1, 2, 3],[4, 5, 6]] tt = tf.transpose(t) #等价于tt = tf.transpose(t,[1,0]) sess = tf.Session() print(sess.run(tt)) #将原有shape[2,3]中的第1和第2维度顺序颠倒，变为新的shape[3,2] 则输出如下： [[1 4] [2 5] [3 6]]
tf.reverse(tensor, dims,name=None)	沿着指定的维度对输入进行反转。其中，dims为列表，元素含义为指向输入shape的索引。例如： t= [[[0, 1, 2, 3], #定义一个4阶的数组 [4, 5, 6, 7], [8, 9, 10, 11]], [[12, 13, 14, 15], [16, 17, 18, 19], [20, 21, 22, 23]]] print(np.shape(t)) #输出[1, 2, 3, 4] dim=[3] #dim为tshape中的索引，3就代表shape中的最后一个值4。同理，使用-1也可以 rt = tf.reverse(t, dim) #进行反转操作 sess = tf.Session() print(sess.run(rt)) #输出反转后的结果为： # [[[3, 2, 1, 0], # [7, 6, 5, 4], # [11, 10, 9, 8]], # [[15, 14, 13, 12], # [19, 18, 17, 16], # [23, 22, 21, 20]]] rt = tf.reverse(t, [1,2]) #也可以同时按照多个轴反转

*****ebook converter DEMO Watermarks*****

(续)

操作	描述
tf.reverse(tensor, dims, name=None)	print(sess.run(rt))#按照shape中1、2的索引指向的值为2、3， 基于这两个维度反转输出的结果为： <pre># [[[20 21 22 23] # [16 17 18 19] # [12 13 14 15]] # # [[8 9 10 11] # [4 5 6 7] # [0 1 2 3]]]]</pre>
tf.matmul(a,b,transpose_a=False, transpose_b=False,a_is_sparse=False, b_is_sparse=False, name=None)	矩阵相乘
tf.matrix_determinant(input, name=None)	返回方阵的行列式
tf.matrix_inverse(input,adjoint=None, name=None)	求方阵的逆矩阵，adjoint为True时，计算输入共轭矩阵的逆矩阵
tf.cholesky(input, name=None)	对输入方阵cholesky分解，即把一个对称正定的矩阵表示成一个下三角矩阵L和其转置的乘积的分解A=LL^T
tf.matrix_solve(matrix, rhs, adjoint=None, name=None)	求解矩阵方程，返回矩阵变量。其中，matrix为矩阵变量的系数，rhs为矩阵方程的结果。例如： $2x+3y=12$ $x+y=5$ 代码可以写为： <pre>sess = tf.InteractiveSession() a = tf.constant([[2.,3.], [1.,1.]]) print(tf.matrix_solve(a, [[12.],[5.]]).eval()) #输出方程中x和y的解 #[[3.00000024] #[1.99999988]] #即 x=3, y=2</pre>

4.2.4 复数操作函数

关于复数的操作函数如表4-12所示。

表4-12 复数操作函数

函数	描述
tf.complex(real, imag, name=None)	将两实数转换为复数形式。例如： <code>real = [2.25, 3.25]</code> <code>Imag = [4.75, 5.75]</code> <code>tf.complex(real, imag) ==> [[2.25 + 4.75j], [3.25 + 5.75j]]</code>

函 数	描 述
<code>tf.complex_abs(x, name=None)</code>	计算复数的绝对值，即长度。例如： <code>x = [[-2.25 + 4.75j], [-3.25 + 5.75j]]</code> <code>tf.complex_abs(x) ==> [5.25594902, 6.60492229]</code>
<code>tf.conj(input, name=None)</code>	计算共轭复数
<code>tf.imag(input, name=None)</code>	提取复数的虚部和实部
<code>tf.real(input, name=None)</code>	
<code>tf.fft(input, name=None)</code>	计算一维的离散傅里叶变换，输入数据类型为complex64

4.2.5 规约计算

规约计算的操作都会有降维的功能，在所有`reduce_xxx`系列操作函数中，都是以`xxx`的手段降维，每个函数都有`axis`这个参数，即沿某个方向，使用`xxx`方法对输入的Tensor进行降维。



提示： `axis`的默认值是`None`，即把`input_tensor`降到0维，即一个数。

对于二维`input_tensor`而言：`axis=0`，则按列计算；`axis=1`，则按行计算。

参数`reduction_indices`是为了兼容以前的版本与`axis`保证相同的含义。如表4-13所示为规约计算函数及其说明。

表4-13 规约计算函数

操作	描述
<code>tf.reduce_sum(input_tensor, axis=None, keep_dims=False, name=None, reduction_indices=None)</code>	计算输入tensor元素的和，或者按照axis指定的轴进行求和。例如： <code>x = [[1, 1, 1],[1, 1, 1]] tf.reduce_sum(x) ==> 6 tf.reduce_sum(x, 0) ==> [2, 2, 2] tf.reduce_sum(x, 1) ==> [3, 3] tf.reduce_sum(x, 1, keep_dims=True) ==> [[3], [3]] tf.reduce_sum(x, [0, 1]) ==> 6</code>
<code>tf.reduce_prod(input_tensor, axis=None, keep_dims=False, name=None, reduction_indices=None)</code>	计算输入tensor元素的乘积，或者按照axis指定的轴进行求乘积。例如： <code>fi=tf.Variable(tf.constant([2,3,4,5]), shape=[2,2]) ff=tf.reduce_prod(fi, 0) with tf.Session() as sess: sess.run(tf.global_variables_initializer()) print(sess.fun(fi)) print(sess.fun(ff))</code> 运行代码如下： <code>[[2 3] [4 5]] [8 15]</code>

(续)

操作	描述
<code>tf.reduce_min(input_tensor, axis=None, keep_dims=False, name=None, reduction_indices=None)</code>	求tensor中的最小值
<code>tf.reduce_max(input_tensor, axis=None, keep_dims=False, name=None, reduction_indices=None)</code>	求tensor中的最大值
<code>tf.reduce_mean(input_tensor, axis = None , keep_dims=False, name=None, reduction_indices=None)</code>	求tensor中的平均值
<code>tf.reduce_all(input_tensor, axis=None, keep_dims=False, name=None, reduction_indices=None)</code>	对tensor中的各个元素求逻辑'与'。例如： <code>x = [[True, True],[False, False]] tf.reduce_all(x) ==> False tf.reduce_all(x, 0) ==> [False, False] tf.reduce_all(x, 1) ==> [True, False]</code>
<code>tf.reduce_any(input_tensor, axis=None, keep_dims=False, name=None, reduction_indices=None)</code>	对tensor中各个元素求逻辑'或'

4.2.6 分割

分割操作是TensorFlow不常用的操作，在复杂的网络模型里偶尔才会用到。如表4-14所示为分割操作的相关函数。

表4-14 分割相关函数

操作	描述
<code>tf.segment_sum(data, segment_ids, name=None)</code>	按照segment_ids指定的维度，分割张量data中的值，还可以返回data中指定片段的累加和。例如： <code>c = tf.constant([[1,2,3,4], [-1,-2,-3,-4], [5,6,7,8]])</code> <code>tf.segment_sum(c, tf.constant([0, 0, 1]))</code> 其输出如下： <code>[[0 0 0]</code> <code>[5 6 7 8]]</code> 这个例子表明：将c按照 [0, 0, 1]的维度来分割，并将c中的头两行加起来作为结果的第1行，将c中的第3行作为结果的第2行
<code>tf.segment_prod(data, segment_ids, name=None)</code>	根据segment_ids的分段计算各个片段的积
<code>tf.segment_min(data, segment_ids, name=None)</code>	根据segment_ids的分段计算各个片段的最小值
<code>tf.segment_max(data, segment_ids, name=None)</code>	根据segment_ids的分段计算各个片段的最大值
<code>tf.segment_mean(data, segment_ids, name=None)</code>	根据segment_ids的分段计算各个片段的平均值
<code>tf.unsorted_segment_sum(data, segment_ids, num_segments, name=None)</code>	与tf.segment_sum函数类似，不同在于segment_ids中id顺序可以是无序的

(续)

操作	描述
<code>tf.sparse_segment_sum(data, indices, segment_ids, name=None)</code>	输入进行稀疏分割求和。例如： <code>c = tf.constant([[1,2,3,4], [-1,-2,-3,-4], [5,6,7,8]])</code> #取c的头两行，并将返回的结果[1,2,3,4],[-1,-2,-3,-4]中的第1行和第2行相加作为最终结果的第一行返回 <code>tf.sparse_segment_sum(c, tf.constant([0,1]), tf.constant([0, 0]))</code>
<code>tf.sparse_segment_sum(data, indices, segment_ids, name=None)</code>	则输出如下： [[0 0 0]] 对原data的indices为[0,1]位置的进行分割，并按照segment_ids的分组进行求和

4.2.7 序列比较与索引提取

对于序列和数组的操作，是本书中常用的方法，具体的函数如表4-15所示。

表4-15 序列比较与索引提取相关函数

操作	描述
tf.argmin(input, axis, name=None)	返回input最小值的索引index
tf.argmax(input, axis, name=None)	返回input最大值的索引index。axis:0表示按列，axis:1表示按行
tf.setdiff1d(x, y, name=None)	返回x, y中不同值的索引
tf.where(condition, x=None, y=None, name=None)	根据指定条件，返回对应的值或坐标。若x,y都为None，返回condition值为True的坐标，若x、y都不为None，返回condition值为True的坐标在x内的值，condition值为False的坐标在y内的值。例如： cond=[True, False, False, True] x=[1, 2, 3, 4] y=[5, 6, 7, 8] tf.where(cond)==>[[0] [3]] tf.where(cond, x, y)==>[1 6 7 4]
tf.unique(x, name=None)	返回一个元组tuple(y, idx)。其中，y为x列表的唯一化数据列表，idx为x数据对应y元素的index。例如： x = [1, 1, 2, 4, 4, 4, 7, 8, 8] y, idx = unique(x) y ==> [1, 2, 4, 7, 8] idx ==> [0, 0, 1, 2, 2, 2, 3, 4, 4]
tf.invert_permutation(x, name=None)	将x中元素的值当做索引，返回新的张量。例如： x = [3, 4, 0, 2, 1] invert_permutation(x) ==> [2, 4, 3, 0, 1]
tf.random_shuffle(input)	沿着input的第一维进行随机重新排列

4.2.8 错误类

作为一个完整的框架，有它自己的错误处理。TensorFlow中的错误类如表4-16所示，该部分不常用，可以作为工具，使用时查询一下即可。

表4-16 错误类

操作	描述
class tf.OpError	一个基本的错误类型，在当TF执行失败时报错
tf.OpError.op	返回执行失败的操作节点 有的操作如Send或Recv可能不会返回，则要用到node_def方法
tf.OpError.node_def	以NodeDef proto形式表示失败的OP
tf.OpError.error_code	描述该错误的整数错误代码
tf.OpError.message	返回错误信息
class tf.errors.CancelledError	当操作或者阶段呗取消时报错
class tf.errors.UnknownError	未知错误类型
class tf.errors.InvalidArgumentError	在接收到非法参数时报错
class tf.errors.NotFoundError	当发现不存在所请求的一个实体时，比如文件或目录
class tf.errors.AlreadyExistsError	当创建的实体已经存在时报错
class tf.errors.PermissionDeniedError	没有执行权限做某操作时报错
class tf.errors.ResourceExhaustedError	资源耗尽时报错
class tf.errors.FailedPreconditionError	系统没有条件执行某个行为时报错
class tf.errors.AbortedError	操作中止时报错，常常发生在并发情形
class tf.errors.OutOfRangeError	超出范围报错
class tf.errors.UnimplementedError	某个操作没有执行时报错
class tf.errors.InternalError	当系统经历了一个内部错误时报出
class tf.errors.DataLossError	当出现不可恢复的错误，例如在运行 tf.WholeFileReader.read 读取整个文件的同时文件被删减
tf.errors.XXXXXX.__init__(node_def,op,message)	使用该方式创建以上各种错误类

*****ebook converter DEMO Watermarks*****

4.3 共享变量

下面来到本章的重点——共享变量。共享变量在复杂的网络中用处非常之广泛，所以读者一定要学好。

4.3.1 共享变量用途

在构建模型时，需要使用tf.Variable来创建一个变量（也可以理解成节点）。例如代码：

```
biases = tf.Variable(tf.zeros([2]), name="biases")      #创建
```

但在某种情况下，一个模型需要使用其他模型创建的变量，两个模型一起训练。比如，对抗网络中的生成器模型与判别器模型（后文12章会有详细讲解）。如果使用tf.Variable，将会生成一个新的变量，而我们需要的是原来的那个biases变量。这时怎么办呢？

这时就是通过引入get_variable方法，实现共享变量来解决这个问题。这个种方法可以使用多套网络模型来训练一套权重。

4.3.2 使用get-variable获取变量

*****ebook converter DEMO Watermarks*****

`get_variable`一般会配合`variable_scope`一起使用，以实现共享变量。`variable_scope`的意思是变量作用域。在某一作用域中的变量可以被设置成共享的方式，被其他网络模型使用。后文的4.3.4节中会有共享变量的实例。下面先介绍下`get_variable`的详细使用。

`get_variable`函数的定义如下：

```
tf.get_variable(<name>, <shape>, <initializer>)
```

在TensorFlow里，使用`get_variable`生成的变量是以指定的`name`属性为唯一标识，并不是定义的变量名称。使用时一般通过`name`属性定位到具体变量，并将其共享到其他模型中。

下面通过两个例子来深入介绍。

4.3.3 实例14：演示`get_variable`和`Variable`的区别

实例描述

分别使用`Variable`定义变量和使用`get_variable`来定义变量。请读者仔细观察它们的用法区别。

1. `Variable`的用法

首先先来看一下`Variable`的用法。

代码4-9 get_variable和Variable的区别

```
01 import tensorflow as tf
02
03 var1 = tf.Variable(1.0 , name='firstvar')
04 print ("var1:",var1.name)
05 var1 = tf.Variable(2.0 , name='firstvar')
06 print ("var1:",var1.name)
07 var2 = tf.Variable(3.0 )
08 print ("var2:",var2.name)
09 var2 = tf.Variable(4.0 )
10 print ("var1:",var2.name)
11
12 with tf.Session() as sess:
13     sess.run(tf.global_variables_initializer())
14     print("var1=",var1.eval())
15     print("var2=",var2.eval())
```

上面的代码运行后输出如下：

```
var1: firstvar:0
var1: firstvar_1:0
var2: Variable:0
var1: Variable_1:0
var1= 2.0
var2= 4.0
```

上面代码中定义了两次var1，可以看到在内存中生成了两个var1（因为它们的name不一样），对于图来讲后面的var1是生效的（var1=2.0）。

var2表明了：Variable定义时没有指定名字，系统会自动给加上一个名字Variable: 0。

2. get_variable用法演示

接着上面的代码，使用get_variable添加get_var1变量。

代码4-9 get_variable和Variable的区别
(续)

```
16 get_var1 = tf.get_variable("firstvar",[1], initializer=t1
initializer(0.3))
17 print ("get_var1:",get_var1.name)
18
19 get_var1 = tf.get_variable("firstvar",[1], initializer=t1
initializer(0.4))
20 print ("get_var1:",get_var1.name)
```

代码运行之后结果如下：

```
var1: firstvar:0
var1: firstvar_1:0
var2: Variable:0
var1: Variable_1:0
var1= 2.0
var2= 4.0
get_var1: firstvar_2:0
Traceback (most recent call last):
....
```

可以看到，程序在定义第2个get_var1时发生崩溃了。这表明，使用get_variable只能定义一次指定名称的变量。同时由于变量firstvar在前面使用Variable函数生成过一次，所以系统自动变成了firstvar_2: 0。

如果将崩溃的句子改成下面的样子：

代码4-9 get_variable和Variable的区别 (续)

```
21 get_var1 = tf.get_variable("firstvar",[1], initializer=t1
initializer(0.3))
22 print ("get_var1:",get_var1.name)
23
24 get_var1 = tf.get_variable("firstvar1",[1], initializer=t1
initializer(0.4))
25 print ("get_var1:",get_var1.name)
26
27 with tf.Session() as sess:
28     sess.run(tf.global_variables_initializer())
29     print("get_var1=",get_var1.eval())
```

运行代码，输出如下：（部分内容）

```
.....
get_var1: firstvar_2:0
get_var1: firstvar1:0
get_var1= [ 0.40000001]
```

可以看到，这次仍然是又定义了一个get_var1，不同的是改变了它的名字firstvar1，这样就没有问题了。同样，新的get_var1会在图中生效，所以它的输出值是4.0而不是3.0。

4.3.4 实例15：在特定的作用域下获取变量

实例描述

在作用域下，使用get_variable，以及嵌套variable_scope。

在前面的例子中，大家已经知道使用get_variable创建两个同样名字的变量是行不通的，如下代码会报错。

```
var1 = tf.get_variable("firstvar", shape=[2], dtype=tf.float32)
var2 = tf.get_variable("firstvar", shape=[2], dtype=tf.float32)
```

如果真的想要那么做，可以使用variable_scope将它们隔开，代码如下。

代码4-10 get_variable配合variable_scope

```
import tensorflow as tf
with tf.variable_scope("test1", ):#定义一个作用域test1
    var1 = tf.get_variable("firstvar", shape=[2], dtype=tf.float32)

with tf.variable_scope("test2"):
    var2 = tf.get_variable("firstvar", shape=[2], dtype=tf.float32)

print ("var1:", var1.name)
print ("var2:", var2.name)
```

运行代码，输出结果如下：

```
var1: test1/firstvar:0
var2: test2/firstvar:0
```

var1和var2都使用firstvar的名字来定义。通过*****ebook converter DEMO Watermarks*****

输出可以看出，其实生成的两个变量var1和var2是不同的，它们作用在不同的scope下，这就是scope的作用。

scope还支持嵌套，将上面代码中的第二个scope缩进一下，得到如下代码：

代码4-11 get_variable配合variable_scope2

```
01 .....
02
03 with tf.variable_scope("test1", ):
04     var1 = tf.get_variable("firstvar", shape=[2], dtype=tf.
05
06     with tf.variable_scope("test2"):
07         var2 = tf.get_variable("firstvar", shape=[2], dtype=
08
09 print ("var1:", var1.name)
10 print ("var2:", var2.name)
```

运行代码，输出结果如下：

```
var1: test1/firstvar:0
var2: test1/test2/firstvar:0
```

4.3.5 实例16：共享变量功能的实现

实例描述

使用作用域中的reuse参数来实现共享变量功能。

费了这么大的劲来使用get_variable，目的其实是为了要通过它实现共享变量的功能。

variable_scope里面有个reuse=True属性，表示使用已经定义过的变量。这时get_variable将不会再创建新的变量，而是去图（一个计算任务）中get_variable所创建过的变量中找与name相同的变量。

在上文代码中再建立一个同样的scope，并且设置reuse=True，实现共享firstvar变量。

代码4-11 get_variable配合variable_scope2（续）

```
11 with tf.variable_scope("test1", reuse=True ):  
12     var3= tf.get_variable("firstvar", shape=[2], dtype=tf.1  
13     with tf.variable_scope("test2"):  
14         var4 = tf.get_variable("firstvar", shape=[2], dtype=  
15  
16 print ("var3:", var3.name)  
17 print ("var4:", var4.name)
```

运行上面代码，输出如下：

```
var1: test1/firstvar:0  
var2: test1/test2/firstvar:0  
var3: test1/firstvar:0  
var4: test1/test2/firstvar:0
```

var1和var3的输出名字是一样的，var2和var4

的名字也是一样的。这表明var1和var3共用了一个变量，var2和var4共用了一个变量，这就实现了共享变量。在实际应用中，可以把var1和var2放到一个网络模型里去训练，把var3和var4放到另一个网络模型里去训练，而两个模型的训练结果都会作用于一个模型的学习参数上。



注意：如果读者使用的是Anaconda工具包里面的Spyder工具（第2章介绍过）运行，该代码只能运行一次，第二次会报错。

解决办法：需要在Anaconda的Consoles菜单里退出当前的kernel，再重新进入一下。再运行才不会报错。否则会提示已经有这个变量了。

为什么会这样呢？

tf.get_variable在创建变量时，会去检查图（一个计算任务）中是否已经创建过该变量。如果创建过并且本次调用时没有被设为共享方式，则会报错。

明白原理后可以加一条语句
tf.reset_default_graph()，将图（一个计算任务）里面的变量清空，就可以解决这个问题了。图（一个计算任务）的更多内容将在后面章节介绍。

4.3.6 实例17：初始化共享变量的作用域

实例描述

演示variable_scope中get_variable初始化的继承功能，以及嵌套variable_scope的继承功能。

variable_scope和get_variable都有初始化的功能。在初始化时，如果没有对当前变量初始化，则TensorFlow会默认使用作用域的初始化方法对其进行初始化，并且作用域的初始化方法也有继承功能。下面演示代码。

代码4-12 共享变量的作用域与初始化

```
01 import tensorflow as tf
02
03 with tf.variable_scope("test1", initializer=tf.constant_
04     (0.4)):
05     var1 = tf.get_variable("firstvar", shape=[2], dtype=tf.1
06
07     with tf.variable_scope("test2"):
08         var2 = tf.get_variable("firstvar", shape=[2], dtype=tf.
09         var3 = tf.get_variable("var3", shape=[2], initializer=1
10         initializer (0.3))
11
12         with tf.Session() as sess:
13             sess.run(tf.global_variables_initializer())
14             print("var1=", var1.eval())          #作用域test1下的变量
15             print("var2=", var2.eval())          #作用域test2下的变量，继承
16             print("var3=", var3.eval())          #作用域test2下的变量
```

上述代码大致操作如下：

- 将test1作用域进行初始化为4.0，见代码第3行。
- var1没有初始化，见代码第4行。
- 嵌套的test2作用域也没有初始化，见代码第6行。
- test2下的var3进行了初始化，见代码第8行。

运行代码，输出如下：

```
var1= [ 0.40000001  0.40000001]
var2= [ 0.40000001  0.40000001]
var3= [ 0.30000001  0.30000001]
```

var1数组值为0.4，表明继承了test1的值；var2数组值为0.4，表明其所在的作用域test2也继承了test1的初始化；变量var3在创建时同步指定了初始化操作，所以数组值为0.3。



注意： 在多模型训练中，常常会使用variable_scope对模型间的张量进行区分。同时，统一为学习参数进行默认的初始化。在变量共享方面，还可以使用tf.AUTO_REUSE来为reuse属性赋值。tf.AUTO_REUSE可以实现第一次调用variable_scope时，传入的reuse值是False；再次调

用variable_scope时，传入reuse的值就会自动变为True。

4.3.7 实例18：演示作用域与操作符的受限范围

实例描述

演示variable_scope的as用法，以及对应的作用域。

variable_scope还可以使用with variable_scope ("name") as xxxscope的方式定义作用域，当使用这种方式时，所定义的作用域变量xxxscope将不再受到外围的scope所限制。看下面的例子。

代码4-13 作用域与操作符的受限范围

```
01 import tensorflow as tf
02
03 with tf.variable_scope("scope1") as sp:
04     var1 = tf.get_variable("v", [1])
05
06 print("sp:", sp.name)                      #作用域名称
07 print("var1:", var1.name)
08
09 with tf.variable_scope("scope2"):
10     var2 = tf.get_variable("v", [1])
11
12     with tf.variable_scope(sp) as sp1:
13         var3 = tf.get_variable("v3", [1])
14
15 print("sp1:", sp1.name)
16 print("var2:", var2.name)
17 print("var3:", var3.name)
```

例子中定义了作用域scope1 as sp（见代码第3行），然后将sp放在作用域scope2中，并as成sp1（见代码第12行）。运行代码输出如下：

```
sp: scope1
var1: scope1/v:0
sp1: scope1
var2: scope2/v:0
var3: scope1/v3:0
```

sp和var1的输出前面已经交代过。sp1在scope2下，但是输出仍是scope1，没有改变。在它下面定义的var3的名字是scope1/v3: 0，表明也在scope1下，再次说明sp没有受到外层的限制。

另外再介绍一个操作符的作用域tf.name_scope，如下所示。操作符不仅受到tf.name_scope作用域的限制，同时也受到tf.variable_scope作用域的限制。

代码4-13 作用域与操作符的受限范围 (续)

```
18 with tf.variable_scope("scope"):
19     with tf.name_scope("bar"):
20         v = tf.get_variable("v", [1])      #v为一个变量
21         x = 1.0 + v                      # x为一个op,
22 print("v:", v.name)
23 print("x.op:", x.op.name)
)
```

上面的代码运行后输出如下：

```
v: scope/v:0  
x.op: scope/bar/add
```

可以看到，虽然v和x都在scope的bar下面，但是v的命名只受到scope的限制，tf.name_scope只能限制op，不能限制变量的命名。

在tf.name_scope函数中，还可以使用空字符将作用域返回到顶层。

下面举例来比较tf.name_scope与variable_scope在空字符情况下的处理：

- 在代码第28行var3的定义之后添加空字符的variable_scope。
- 定义var4，见代码第31行。

代码4-13 作用域与操作符的受限范围
(续)

```
24 with tf.variable_scope("scope2"):  
25     var2 = tf.get_variable("v", [1])  
26  
27     with tf.variable_scope(sp) as sp1:  
28         var3 = tf.get_variable("v3", [1])  
29  
30         with tf.variable_scope("") :  
31             var4 = tf.get_variable("v4", [1])
```

在 $x = 1.0 + v$ 之后添加空字符的
tf.name_scope，并定义y。代码如下：

代码4-13 作用域与操作符的受限范围 (续)

```
32 with tf.variable_scope("scope"):
33     with tf.name_scope("bar"):
34         v = tf.get_variable("v", [1])
35         x = 1.0 + v
36         with tf.name_scope(""):  
37             y = 1.0 + v
```

将var4和y的值打印出来，得出如下信息：

```
var4: scope1//v4:0
y.op: add
```

可以看到，y变成顶层了，而var4多了一个空层。

4.4 实例19：图的基本操作

前面接触了一些图（一个计算任务）的概念，这里来系统地了解一下TensorFlow中的图可以做哪些事情。

实例描述

- (1) 本例演示使用3种方式来建立图，并依次设置为默认图，使用`get_default_graph()`方法来获取当前默认图，验证默认图的设置生效。
- (2) 演示获取图中相关内容的操作。

一个TensorFlow程序默认是建立一个图的，除了系统自动建图以外，还可以手动建立，并做一些其他的操作。

4.4.1 建立图

可以在一个TensorFlow中手动建立其他的图，也可以根据图里的变量获得当前的图。

下面代码演示了使用`tf.Graph`函数建立图，使用`tf.get_default_graph`函数获得图，以及使用`reset_default_graph`的过程来重置图的过程。

代码4-14 图的基本操作

```
01 import numpy as np
02 import tensorflow as tf
03 c = tf.constant(0.0)
04
05 g = tf.Graph()
06 with g.as_default():
07     c1 = tf.constant(0.0)
08     print(c1.graph)
09     print(g)
10     print(c.graph)
11
12 g2 = tf.get_default_graph()
13 print(g2)
14
15 tf.reset_default_graph()
16 g3 = tf.get_default_graph()
17 print(g3)
```

代码运行结果如下：

```
<tensorflow.python.framework.ops.Graph object at 0x000000000000>
```

可以看出。

(1) c是在刚开始的默认图中建立的，所以图的打印值就是原始的默认图的打印值923CCF8。

(2) 然后使用tf.Graph函数建立了一个图

B854940（见代码第5行），并且在新建的图里添加变量，可以通过变量的“.graph”获得所在的图。

(3) 在新图B854940的作用域外，使用tf.get_default_graph函数又获得了原始的默认图923CCF8（见代码第12行）。接着又使用tf.reset_default_graph函数（见代码第15行），相当于重新建了一张图来代替原来的默认图，这时默认的图变成了B8546D8。



注意： 在使用tf.reset_default_graph函数时必须保证当前图的资源已经全部释放，否则会报错。例如，在当前图中使用tf.InteractiveSession函数建立了一个会话，在会话结束时却没有调用close进行关闭，那么再执行tf.reset_default_graph函数时，就会报错。

4.4.2 获取张量

在图里面可以通过名字得到其对应的元素，例如，get_tensor_by_name可以获得图里面的张量。在上个实例中添加如下代码。

代码4-14 图的基本操作（续）

```
18 print(c1.name)
19 t = g.get_tensor_by_name(name = "Const:0")
20 print(t)
```

该部分代码运行结果如下：

```
Const:0  
Tensor("Const:0", shape=(), dtype=float32)
```

常量c1是在一个子图g中建立的。with tf.Graph() as default代码表示使用tf.Graph函数来创建一个图，并在其上面定义OP，见代码第5、6行。

接着演示了如何访问该图中的变量：将c1的名字放到get_tensor_by_name里来反向得到其张量（见代码第19行），通过对t的打印可以看到所得的t就是前面定义的张量c1。



注意：不必花太多精力去关注TensorFlow中默认的命名规则。一般在需要使用名字时，都会在定义的同时为它指定好固定的名字。如果真的不清楚某个元素的名字，可将其打印出来，回填到代码中，再次运行即可。

4.4.3 获取节点操作

获取节点操作OP的方法和获取张量的方法非常类似，使用的方法是get_operation_by_name。下面将获取张量和获取OP的例子放在一起比较一

*****ebook converter DEMO Watermarks*****

下，具体代码如下。

代码4-14 图的基本操作（续）

```
21 a = tf.constant([[1.0, 2.0]])
22 b = tf.constant([[1.0], [3.0]])
23
24 tensor1 = tf.matmul(a, b, name='exampleop')
25 print(tensor1.name, tensor1)
26 test = g3.get_tensor_by_name("exampleop:0")
27 print(test)
28
29 print(tensor1.op.name)
30 testop = g3.get_operation_by_name("exampleop")
31 print(testop)
32
33 with tf.Session() as sess:
34     test = sess.run(test)
35     print(test)
36     test = tf.get_default_graph().get_tensor_by_name("exa
37     print (test)
```

上面示例中，先将张量及其名字打印出来，然后使用g3图的get_tensor_by_name函数又获得了该张量，此时test和tensor1是一样的。为了证明这一点，直接把test放到session的run里，发现它运行后也能得到正确的结果。



注意： 使用默认的图时，也可以用上述代码中的tf.get_default_graph函数获取当前图，然后可以调用get_tensor_by_name函数获取元素。

上面代码运行后会显示如下信息：

```
exampleop:0 Tensor("exampleop:0", shape=(1, 1), dtype=float32)
Tensor("exampleop:0", shape=(1, 1), dtype=float32)
exampleop
name: "exampleop"
op: "MatMul"
input: "Const"
input: "Const_1"
attr {
    key: "T"
    value {
        type: DT_FLOAT
    }
}
attr {
    key: "transpose_a"
    value {
        b: false
    }
}
attr {
    key: "transpose_b"
    value {
        b: false
    }
}
[[ 7.]]
Tensor("exampleop:0", shape=(1, 1), dtype=float32)
```

再仔细看上例中的OP，通过打印 tensor1.op.name的信息，获得了OP的名字，然后通过get_operation_by_name函数获得了相同的OP，可以看出OP与tensor1之间的对应关系。



注意： 这里之所以要放在一起举例，原因就是OP和张量在定义节点时很容易被混淆。上例中的`tensor1 = tf.matmul (a, b, name='exampleop')` 并不是OP，而是张量。OP其实是描述张量中的运算关系，是通过访问张量的

属性找到的。

4.4.4 获取元素列表

如果想看一下图中的全部元素，可以使用get_operations函数来实现。具体代码如下。

代码4-14 图的基本操作（续）

```
38 tt2 = g.get_operations()  
39 print(tt2)
```

运行后显示如下信息：

```
[<tf.Operation 'Const' type=Const>]
```

由于g里面只有一个常量，所以打印了一条信息。

4.4.5 获取对象

前面是根据名字来获取元素，还可以根据对象来获取元素。使用tf.Graph.as_graph_element(obj, allow_tensor=True, allow_operation=True) 函数，即传入的是一个对象，返回一个张量或是一个OP。该函数具有验证和转换功能，在多线程方面会偶尔用到。举例如*****ebook converter DEMO Watermarks*****

下。

代码4-14 图的基本操作（续）

```
40 tt3 = g.as_graph_element(c1)
41 print(tt3)
```

运行代码，输出结果如下：

```
Tensor("Const:0", shape=(), dtype=float32)
```

上述代码通过对tt3的打印可以看到，变量tt3所指的张量名字为Const0，而在4.4.2节中可以看到量名c1所指向的真实张量名字也为Const0。这表明：函数as_graph_element 获得了c1的真实张量对象，并赋给了变量tt3。



备注：这里只是介绍了图中比较简单的操作，图的操作还有很多，有的还很常用。但考虑到初学者的接受程度，更复杂的图操作（如冻结图，将一个图导入另一个图中等）将会在后面的章节中进行介绍。

4.4.6 练习题

试试将tf.get_default_graph函数放在with

*****ebook converter DEMO Watermarks*****

`tf.Graph().as_default()`：作用域里，看看会得到什么，是全局的默认图，还是`tf.Graph`函数新建的图？（示例代码在“代码4-14图的基本操作”中）

4.5 配置分布式TensorFlow

在大型的数据集上进行神经网络的训练，往往需要更大的运算资源，而且还要耗费若干天才能完成运算量。

TensorFlow提供了一个可以分布式部署的模式，将一个训练任务拆成多个小任务，分配到不同的计算机上来完成协同运算，这样使用计算机群运算来代替单机计算，可以使训练时间大大缩短。

4.5.1 分布式TensorFlow的角色及原理

要想配置TensorFlow为分布训练，需要先了解TensorFlow中关于分布式的角色分配。

- ps：作为分布式训练的服务端，等待各个终端（supervisors）来连接。
- worker：在TensorFlow的代码注释中被称为supervisors，作为分布式训练的运算终端。
- chief supervisors：在众多运算终端中必须选择一个作为主要的运算终端。该终端是在运算终端中最先启动的，它的功能是合并各个终端运算后的学习参数，将其保存或载入。

每个具体角色网络标识都是唯一的，即分布在不同IP的机器上（或者同一个机但不同的端口）。

在实际运行中，各个角色的网络构建部分代码必须100%的相同。三者的分工如下：

- 服务端作为一个多方协调者，等待各个运算终端来连接。
- chief supervisors会在启动时统一管理全局的学习参数，进行初始化或从模型载入。
- 其他的运算终端只是负责得到其对应的任务并进行计算，并不会保存检查点，用于TensorBoard可视化中的summary日志等任何参数信息。

整个过程都是通过RPC协议来通信的。

4.5.2 分布部署TensorFlow的具体方法

配置过程中，首先需要建一个server，在server中会将ps及所有worker的IP端口准备好。接着，使用tf.train.Supervisor中的managed_session来管理一个打开的session。session中只是负责运算，而通信协调的事情就都交给supervisor来管理了。

4.5.3 实例20：使用TensorFlow实现分布式部署训练

下面开始实现一个分布式训练的网络模型。本例以“代码4-8线性回归的TensorBoard可视化.py”为原型，在其中添加代码将其改成分布式。

实例描述

在本机通过3个端口来建立3个终端，分别是一个ps，两个worker，实现TensorFlow的分布式运算。

具体步骤如下。

1. 为每个角色添加IP地址和端口，创建server

在一台机器上开3个不同的端口，分别代表ps、chief supervisors 和worker。角色的名称用strjob_name表示。以ps为例，代码如下：

代码4-15 ps

```
01 .....
02 #定义IP和端口
03 strps_hosts="localhost:1681"
04 strworker_hosts="localhost:1682,localhost:1683"
05
06 #定义角色名称
```

*****ebook converter DEMO Watermarks*****

```
07 strjob_name = "ps"
08 task_index = 0
09 #将字符串转成数组
10 ps_hosts = strps_hosts.split(',')
11 worker_hosts = strworker_hosts.split(',')
12 cluster_spec = tf.train.ClusterSpec({'ps': ps_hosts, 'worker': worker_hosts})
13 #创建server
14 server = tf.train.Server(
15         {'ps': ps_hosts, 'worker': worker_hosts},
16         job_name=strjob_name,
17         task_index=task_index)
```

 **注意：** 没有网络基础的读者可能看不明白localhost，说好的IP地址呢？localhost即是本机域名的写法，等同于127.0.0.1（本机IP）。如果是跨机器来做分布式训练，直接写成对应机器的IP地址即可。

2. 为ps角色添加等待函数

ps角色使用server.join函数进行线程挂起，开始接收连接消息。

代码4-15 ps（续）

```
18 #ps角色使用join进行等待
19 if strjob_name == 'ps':
20     print("wait")
21     server.join()
```

3. 创建网络结构

与正常的程序不同，在创建网络结构时，使用tf.device函数将全部的节点都放在当前任务下。

在tf.device函数中的任务是通过tf.train.replica_device_setter来指定的。

在tf.train.replica_device_setter中使用worker_device来定义具体任务名称；使用cluster的配置来指定角色及对应的IP地址，从而实现管理整个任务下的图节点。代码如下：

代码4-15 ps（续）

```
22 with tf.device(tf.train.replica_device_setter(  
23             worker_device="/job:worker/task:%d" % task  
24             cluster=cluster_spec)):  
25     X = tf.placeholder("float")  
26     Y = tf.placeholder("float")  
27     # 模型参数  
28     W = tf.Variable(tf.random_normal([1]), name="weight")  
29     b = tf.Variable(tf.zeros([1]), name="bias")  
30  
31     global_step = tf.train.get_or_create_global_step() #  
32  
33     # 前向结构  
34     z = tf.multiply(X, W)+ b  
35     tf.summary.histogram('z', z) #将预测  
36     #反向优化  
37     cost =tf.reduce_mean( tf.square(Y - z))  
38     tf.summary.scalar('loss_function', cost) #将损失以  
39     learning_rate = 0.01  
40     optimizer = tf.train.GradientDescentOptimizer(learnin  
minimize(cost,global_step=global_step) #梯度下降  
41  
42     saver = tf.train.Saver(max_to_keep=1)  
43     merged_summary_op = tf.summary.merge_all() #合并所有s  
44  
45     init = tf.global_variables_initializer()
```

为了使载入检查点文件时能够同步循环次数，这里加了一个global_step变量，并将其放到优化器中。这样，每次运行一次优化器，global_step就会自动获得当期迭代的次数。



注意： init =

tf.global_variables_initializer () 这个代码是将其前面的变量全部初始化，如果后面再有变量，则不会被初始化。所以，一般要将init = tf.global_variables_initializer () 这个代码放在最后。这是个很容易出错的地方，常常令开发者找不到头绪。读者也可以试着在最前面运行，看看会发生什么。

4. 创建Supervisor，管理session

代码4-15 ps (续)

```
46 # 定义参数
47 training_epochs = 2200
48 display_step = 2
49
50 sv = tf.train.Supervisor(is_chief=(task_index == 0),#0号w
51                         logdir="log/super/",
52                         init_op=init,
53                         summary_op=None,
54                         saver=saver,
55                         global_step=global_step,
56                         save_model_secs=5)
57
58 #连接目标角色创建session
59 with sv.managed_session(server.target) as sess:
```

在tf.train.Supervisor函数中，is_chief表明了是否为chief supervisors角色。这里将task_index=0的worker设置成chief supervisors。

logdir为检查点文件和summary文件保存的路径。

init_op表示使用初始化变量的函数。

saver需要将保存检查点的saver对象传入，supervisor就会自动保存检查点文件。如果不想自动保存，可以设为None。

同理，summary_op也是自动保存summary文件。这里设为None，表示不自动保存。

save_model_secs为保存检查点文件的时间间隔。这里设为5，表示每5秒自动保存一次检查点文件。以上代码，为了让分布运算的效果明显一些，将迭代次数改成了2200，使其运算时间变长。

5. 迭代训练

session中的内容与以前一样，直接迭代训练即可。由于使用了supervisor管理session，将使用sv.summary_computed函数来保存summary文件。同样，如想要手动保存检测点文件，也可以使用sv.saver.save。代码如下：

*****ebook converter DEMO Watermarks*****

代码4-15 ps (续)

```
60 print("sess ok")
61     print(global_step.eval(session=sess))
62
63     for epoch in range(global_step.eval(session=sess), train_epochs*len(train_X)):
64
65         for (x, y) in zip(train_X, train_Y):
66             _, epoch = sess.run([optimizer, global_step] , {X: x, Y: y})
67             #生成summary
68             summary_str = sess.run(merged_summary_op, feed_dict={X: x, Y: y});
69             #将summary 写入文件
70             sv.summary_computed(sess, summary_str, global_step)
71             if epoch % display_step == 0:
72                 loss = sess.run(cost, feed_dict={X: train_X, Y: train_Y})
73                 print ("Epoch:", epoch+1, "cost=", loss, "batchsize=", batch_size, "b=", sess.run(b))
74                 if not (loss == "NA" ):
75                     plotdata["batchsize"].append(epoch)
76                     plotdata["loss"].append(loss)
77
78     print (" Finished!")
79     sv.saver.save(sess,"log/mnist_with_summaries/"+sv.checkpoint_name, step=epoch)
80
81 sv.stop()
```

 **注意：** (1) 在设置自动保存检查点文件后，手动保存仍然有效。

(2) 在运行一半后终止，再运行supervisor时会自动载入模型的参数，不需要手动调用saver.restore。

(3) 在session中，不需要再运行

*****ebook converter DEMO Watermarks*****

`tf.global_variables_initializer`函数。原因是supervisor在建立时会调用传入的`init_op`进行初始化，如果加了`sess.run (tf.global_variables_initializer ())`，则会导致所载入模型的变量被二次清空。

6. 建立worker文件

将文件复制两份，分别起名为“4-16 worker.py”与“4-17 worker2.py”，将角色名称修改成`worker`，并将“4-16 worker2.py”中的`task_index`设为1。

代码4-16 worker

```
.....  
#定义角色名称  
strjob_name = "worker"  
task_index = 0  
.....
```

代码4-17 worker2

```
.....  
#定义角色名称  
strjob_name = "worker"  
task_index = 1  
.....
```



注意：这个例子中使用了`summary`的一些方法将运行时态的数据保存起来，以便于使用*****ebook converter DEMO Watermarks*****

TensorBoard进行查看（见4.1.16节）。但在分布式部署时，使用该功能还需要注意以下几点：

(1) 不能使用`sv.summary_computed`，因为`worker2`不是`chief supervisors`，在`worker2`中是不会为`supervisor`对象构造默认`summary_writer`的（所有的`summary`信息都要通过该对象进行写入），所以即使调用`summary_computed`也无法执行下去，程序会报错。

(2) 手写控制`summary`与检查点文件保存时，需要将`chief supervisors`以外的`worker`全部去掉才可以。可以使用`supervisor`按时间间隔保存的形式来管理，这样用一套代码就可以解决了。

7. 部署运行

(1) 在Spyder中先将“4-15 ps.py”文件运行起来，选择菜单Consoles|Open an IPython console命令，新打开一个Consoles，如图4-7所示。

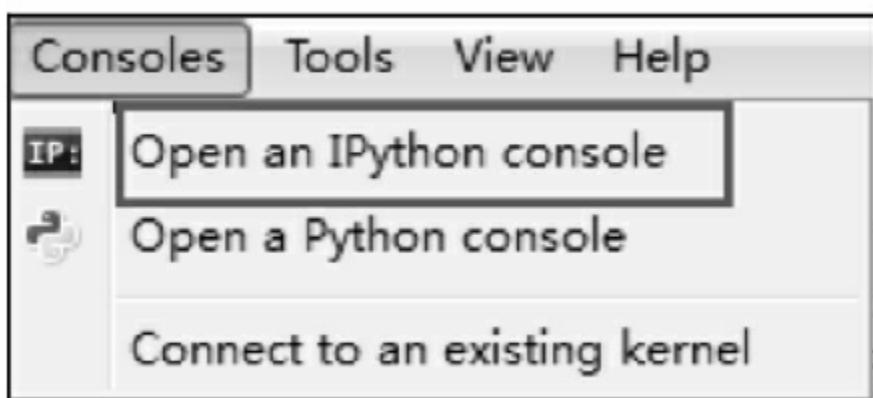
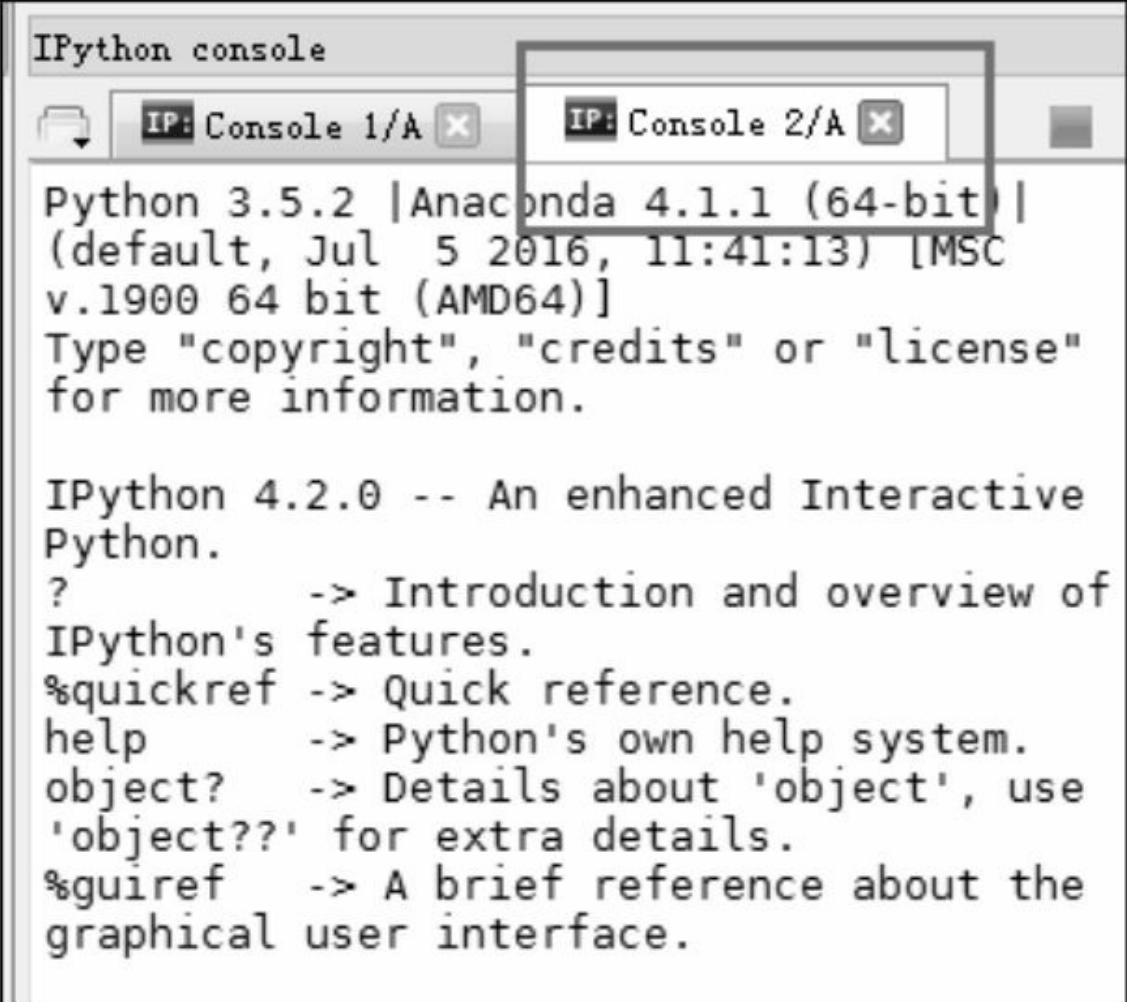


图4-7 Consoles菜单

(2) 在Spyder面板的右下角（见图2-13中的输出栏），可以看到在原有标题为“Console 1/A”标签旁边又多了一个“Console 2/A”标签（如图4-8所示），单击该标签，使其处于激活状态。



```
IPython console
IP: Console 1/A IP: Console 2/A
Python 3.5.2 |Anaconda 4.1.1 (64-bit)|
(default, Jul 5 2016, 11:41:13) [MSC
v.1900 64 bit (AMD64)]
Type "copyright", "credits" or "license"
for more information.

IPython 4.2.0 -- An enhanced Interactive
Python.
?          -> Introduction and overview of
IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use
'object??' for extra details.
%guiref   -> A brief reference about the
graphical user interface.
```

图4-8 Consoles 2/A标签

(3) 运行4-17 worker2.py文件。最后按照“4-17worker2.py”文件启动的方式，启动4-16 worker.py”文件，这时3个窗口的显示内容分别如*****ebook converter DEMO Watermarks*****

下：

- “4-16worker.py” 文件对应窗口显示正常的训练信息。
-

```
.....  
Epoch: 8000 cost= 0.0754263 W= [ 2.01029539] b= [-0.00388618  
Epoch: 8002 cost= 0.074845 W= [ 2.00651097] b= [ 0.00453186  
Epoch: 8003 cost= 0.0748089 W= [ 2.00529122] b= [ 0.00281144  
Epoch: 8005 cost= 0.0747555 W= [ 2.00324082] b= [ 0.00635108  
Epoch: 8007 cost= 0.075026 W= [ 2.00662613] b= [-0.00956773  
Epoch: 8009 cost= 0.0749311 W= [ 2.00585985] b= [-0.006533]  
Epoch: 8010 cost= 0.0748186 W= [ 2.00469637] b= [-0.00152521  
Epoch: 8011 cost= 0.0750369 W= [ 2.0065136] b= [-0.02676161  
Epoch: 8012 cost= 0.0758979 W= [ 2.0068512] b= [-0.02852018  
Epoch: 8013 cost= 0.0759059 W= [ 2.00671506] b= [-0.02870713  
Epoch: 8015 cost= 0.0753608 W= [ 2.0055182] b= [-0.01959283  
Epoch: 8018 cost= 0.0760464 W= [ 2.00559783] b= [-0.03230772  
Epoch: 8021 cost= 0.0758819 W= [ 2.00522184] b= [-0.02836083  
Epoch: 8023 cost= 0.0758949 W= [ 2.00778055] b= [-0.01191433  
Epoch: 8026 cost= 0.0752242 W= [ 2.00646138] b= [-0.01574964  
Epoch: 8028 cost= 0.0751021 W= [ 2.00708318] b= [-0.01172168  
Epoch: 8030 cost= 0.0749788 W= [ 2.0083425] b= [-0.00503741  
Epoch: 8034 cost= 0.0750521 W= [ 2.00837708] b= [-0.00846671  
Epoch: 8035 cost= 0.0750075 W= [ 2.01157689] b= [ 0.00467709  
Epoch: 8037 cost= 0.0751661 W= [ 2.01191807] b= [ 0.01593771  
Epoch: 8038 cost= 0.0750556 W= [ 2.01164842] b= [ 0.01059892  
Epoch: 8040 cost= 0.0753085 W= [ 2.01313496] b= [ 0.01954099  
Epoch: 8042 cost= 0.0753466 W= [ 2.01260543] b= [ 0.02123925  
.....
```

可以看到循环的次数并不是连续的，跳过的步骤被分配到worker2中去运算了。

- “4-17worker2.py” 文件对应窗口显示的信息如下：
-

```
INFO:tensorflow:Waiting for model to be ready. Ready_for_load  
*****ebook converter DEMO Watermarks*****
```

```
None, ready: Variables not initialized: weight, bias, global_step
INFO:tensorflow:Starting queue runners.

.....
Epoch: 8003 cost= 0.0977818 W= [ 2.00529122] b= [ 0.0028114]
Epoch: 8005 cost= 0.0979236 W= [ 2.00324082] b= [ 0.0063510]
Epoch: 8007 cost= 0.0978101 W= [ 2.0065136] b= [-0.01009204]
Epoch: 8012 cost= 0.0985371 W= [ 2.00671506] b= [-0.02870713]
Epoch: 8015 cost= 0.0981559 W= [ 2.0055182] b= [-0.01959283]
Epoch: 8017 cost= 0.0986897 W= [ 2.00519013] b= [-0.0299246]
Epoch: 8018 cost= 0.0987787 W= [ 2.00559783] b= [-0.0312844]
Epoch: 8020 cost= 0.0988223 W= [ 2.00550485] b= [-0.0290601]
Epoch: 8022 cost= 0.0985962 W= [ 2.00522184] b= [-0.0291886]
Epoch: 8024 cost= 0.0982481 W= [ 2.00616717] b= [-0.02256276]
Epoch: 8025 cost= 0.0977918 W= [ 2.00778055] b= [-0.0119143]
Epoch: 8026 cost= 0.0979684 W= [ 2.00646138] b= [-0.0157496]
Epoch: 8028 cost= 0.0978234 W= [ 2.00708318] b= [-0.01172168]
Epoch: 8030 cost= 0.0976372 W= [ 2.00842071] b= [-0.0048569]
Epoch: 8031 cost= 0.0976208 W= [ 2.00859952] b= [-0.0040868]
Epoch: 8032 cost= 0.0976431 W= [ 2.0083425] b= [-0.00503741]
Epoch: 8034 cost= 0.097557 W= [ 2.01164842] b= [ 0.01059892]
Epoch: 8039 cost= 0.0975473 W= [ 2.01065278] b= [ 0.00720035]
Epoch: 8040 cost= 0.0977502 W= [ 2.01313496] b= [ 0.01954095]
Epoch: 8042 cost= 0.0978443 W= [ 2.01260543] b= [ 0.02123925]

.....
```

显示结果中有警告输出，这是因为在构建 supervisor时没有填写local_init_op参数，该参数的意思是在创建worker实例时，初始化本地变量。由于例子中没有填，系统就会自动初始化，并给出警告提示。

从日志中可以看到worker2 与chief supervisors 的迭代序号近似互补，为什么没有绝对互补呢？可能与supervisor中的同步算法有关。

分布运算目的是为了提高整体运算速度，如果同步epoch的准确度需要以牺牲总体运算速度为代价，自然很不合适。所以更合理的推断是因为

*****ebook converter DEMO Watermarks*****

单机单次的运算太快迫使算法使用了更宽松的同步机制。

重要的一点是对于指定步数的学习参数 b 和 w 是一致的（如第8040步，学习参数是相同的，都为 $W = [2.01313496] b = [0.01954099]$ ），这表明两个终端是在相同的起点上进行运算的。

- 对于4-15ps.py文件，其对应窗口则是一直静默着只显示打印的那句wait，因为它只负责连接不参与运算。

4.6 动态图（Eager）

动态图是相对于静态图而言的。所谓的动态图是指在Python中代码被调用后，其操作立即被执行的计算。其与静态图最大的区别是不需要使用session来建立会话了。即，在静态图中，需要在会话中调用run方法才可以获得某个张量的具体值；而在动态图中，直接运行就可以得到具体值了。

动态图是在TensorFlow 1.3版本之后出现的。它使TensorFlow 的入门变得更简单，也使研发更直观。

启用动态图只需要在程序的最开始处加上两行代码：

```
import tensorflow.contrib.eager as tfe
tfe.enable_eager_execution()
```

这两行代码的作用就是开启动态图计算功能。例如，调用tf.matmul时，将会立即计算两个数相乘的值，而不是一个op。

Eager还处于一个试用阶段，也是TensorFlow大力推广的新特性，未来或许会成为趋势。想了解更多内容，可以参考如下网址：

<https://github.com/tensorflow/tensorflow/tree/master/tensorflow/python/framework>

在创建动态图的过程中， 默认也建立了一个session。所有的代码都在该session中进行，而且该session具有进程相同的生命周期。这表明一旦使用动态图就无法实现静态图中关闭session的功能。这便是动态图的不足之处：无法实现多session操作。如果当前代码只需要一个session来完成的话，建议优先选择动态图Eager来实现。

4.7 数据集 (tf.data)

TensorFlow中有3种数据输入模式：

- 直接使用feed_dict利用注入模式进行数据输入（见4.1.4节），适用于少量的数据集输入；
- 使用队列式管道（见11.5.3节），适用于大量的数据集输入；
- 性能更高的输入管道，适用于TensorFlow 1.4之后的版本，是为动态图（见4.6节）功能提供的大数据集输入方案（动态图的数据集输入只能使用该方法），当然也支持静态图。

关于第3种方式的更多介绍，请参考以下链接：

<https://github.com/tensorflow/tensorflow/blob/master/pipelin...>

第5章 识别图中模糊的手写数字 (实例21)

本章中将训练一个能够识别图片中手写数字的机器学习模型。这个模型很简单，仅使用了一个神经元——Softmax Regression。

学完本章，读者一方面可以巩固第4章所学的TensorFlow编程基础知识，另一方面对神经网络也有了一个大体的了解，还掌握了最简单的图像识别方法。

本章含有教学视频共13分14秒。

作者按照本章的内容结构，对主要内容进行了快速讲解，详细讲解了一个识别模糊手写数字图片的完整例子（重点是能够理解例子中的全部代码）。



本章实例中所用的图片来源于一个开源的训练数据集——MNIST。

实例描述

从MNIST数据集中选择一幅图，这幅图上有一个手写的数字，让机器模拟人眼来区分这个手写数字到底是几。

首先来介绍一下编写代码的相关步骤。

(1) 导入NMIST数据集。

(2) 分析MNIST样本特点定义变量。

(3) 构建模型。

*****ebook converter DEMO Watermarks*****

(4) 训练模型并输出中间状态参数。

(5) 测试模型。

(6) 保存模型。

(7) 读取模型。

下面我们就来一一操作。

5.1 导入图片数据集

首先来看看数据集是什么样的。

MNIST是一个入门级的计算机视觉数据集。当我们开始学习编程时，第一件事往往是学习打印Hello World。在机器学习入门的领域里，我们会用MNIST数据集来实验各种模型。

5.1.1 MNIST数据集介绍

MNIST里包含各种手写数字图片，如图5-1所示。

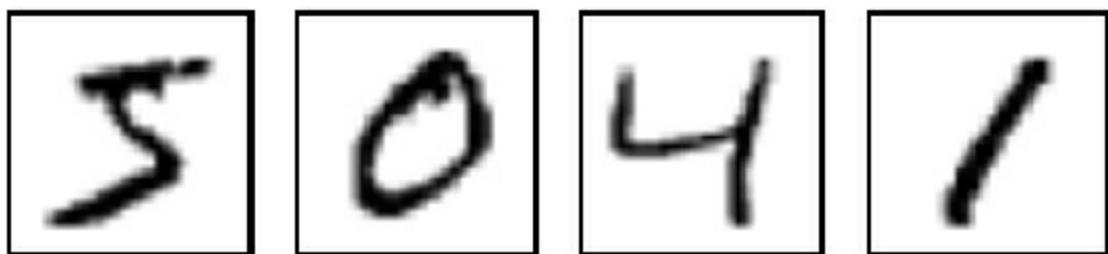


图5-1 MNIST中的数字

它也包含每一张图片对应的标签，告诉我们这个是数字几。例如，上面这4张图片的标签分别是5、0、4、1。

MNIST数据集的官网是<http://yann.lecun.com/exdb/mnist/>，读者可以在

这里面手动下载数据集，如图5-2所示。



图5-2 MNIST数据集下载

5.1.2 下载并安装MNIST数据集

介绍完MNIST数据集后，下面来演示一下如何通过代码来对其操作。

1. 利用TensorFlow代码下载MNIST

TensorFlow提供了一个库，可以直接用来自动下载与安装MNIST，见如下代码：

代码5-1 MNIST数据集

```
01 from tensorflow.examples.tutorials.mnist import input_data  
02 mnist = input_data.read_data_sets("MNIST_data/", one_hot=
```

运行上面的代码，会自动下载数据集并将文件解压到当前代码所在同级目录下的MNIST_data文件夹下。



注意： 代码中的one_hot=True，表示将样本标签转化为one_hot编码。

举例来解释one_hot编码：假如一共10类。0的one_hot为1000000000，1的one_hot为0100000000，2的one_hot为0010000000，3的one_hot为0001000000……依此类推。只有一个位为1，1所在的位置就代表着第几类。

MNIST数据集中的图片是 28×28 Pixel，所以，每一幅图就是1行784（ 28×28 ）列的数据，括号中的每一个值代表一个像素。

- 如果是黑白的图片，图片中黑色的地方数值为0；有图案的地方，数值为0~255之间的数字，代表其颜色的深度。
- 如果是彩色的图片，一个像素会由3个值来表示RGB（红、黄、蓝）。在后面讲解其他数据集时会具体讲到。

接下来通过几行代码将MNIST里面的信息打印出来，看看它的具体内容。

代码5-1 MNIST数据集（续）

```
03 print ('输入数据:',mnist.train.images)
04 print ('输入数据打shape:',mnist.train.images.shape)
05 import pylab
06 im = mnist.train.images[1]
07 im = im.reshape(-1,28)
08 pylab.imshow(im)
09 pylab.show()
```

运行上面的代码，输出信息如下：

```
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
输入数据: [[ 0.  0.  0. ...,  0.  0.  0.]
 [ 0.  0.  0. ...,  0.  0.  0.]
 [ 0.  0.  0. ...,  0.  0.  0.]
 .....
 [ 0.  0.  0. ...,  0.  0.  0.]
 [ 0.  0.  0. ...,  0.  0.  0.]
 [ 0.  0.  0. ...,  0.  0.  0.]]
输入数据打shape: (55000, 784)
```

输出结果如图5-3所示

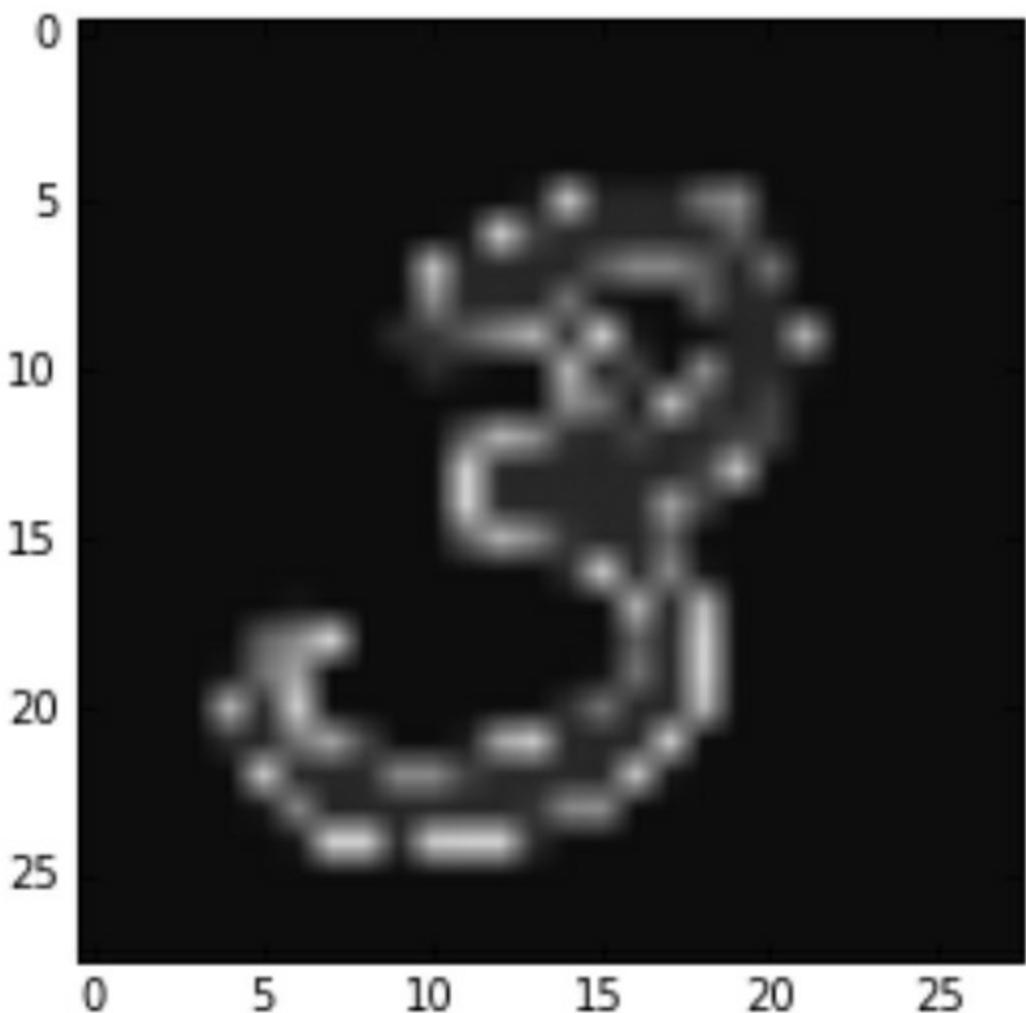


图5-3 输出结果

刚开始的打印信息是解压数据集的意思。如果是第一次运行，还会显示下载数据的相关信息。

接着打印出来的是训练集的图片信息，是一个55000行、784列的矩阵。即，训练集里有55000张图片。

2. MNIST数据集组成

在MNIST训练数据集中，`mnist.train.images`是一个形状为[55000, 784]的张量。其中，第1个维度数字用来索引图片，第2个维度数字用来索引每张图片中的像素点。此张量里的每一个元素，都表示某张图片里的某个像素的强度值，值介于0 ~ 255之间。

MNIST里包含3个数据集：第一个是训练数据集，另外两个分别是测试数据集（`mnist.test`）和验证数据集（`mnist.validation`）。可使用如下命令查看里面的数据信息：

代码5-1 MNIST数据集（续）

```
10 print ('输入数据打shape:',mnist.test.images.shape)
11 print ('输入数据打shape:',mnist.validation.images.shape)
```

运行完上面的命令，可以发现在测试数据集里有10000条样本图片，验证数据集里有5000个图片。

在实际的机器学习模型设计时，样本一般分为3部分：

- 一部分用于训练；
- 一部分用于评估训练过程中的准确度（测试数据集）；

· 一部分用于评估最终模型的准确度（验证数据集）。

训练过程中，模型并没有遇到过验证数据集中的数据，所以利用验证数据集可以评估出模型的准确度。这个准确度越高，代表模型的泛化能力越强。

另外，这3个数据集还有分别对应的3个文件（标签文件），用来标注每个图片上的数字是几。把图片和标签放在一起，称为“样本”。通过样本来就可以实现一个有监督信号的深度学习模型。

相对应的，MNIST数据集的标签是介于0~9之间的数字，用来描述给定图片里表示的数字。标签数据是“one-hot vectors”：一个one-hot向量，除了某一位的数字是1外，其余各维度数字都是0。例如，标签0将表示为([1, 0, 0, 0, 0, 0, 0, 0, 0, 0])。因此，mnist.train.labels是一个[55000, 10]的数字矩阵。

5.2 分析图片的特点，定义变量

由于输入图片是个 550000×784 的矩阵，所以先创建一个[None, 784]的占位符x和一个[None, 10]的占位符y，然后使用feed机制将图片和标签输入进去。具体代码如下。

代码5-2 MNIST分类

```
01 import tensorflow as tf # 导入tensorflow库
02 from tensorflow.examples.tutorials.mnist import input_data
03 mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)
04 import pylab
05
06 tf.reset_default_graph()
07 # 定义占位符
08 x = tf.placeholder(tf.float32, [None, 784]) # MNIST数据集中的
# 28×28=784
09 y = tf.placeholder(tf.float32, [None, 10]) # 数字0~9，共10类
```

代码中第8行的None，表示此张量的第一个维度可以是任何长度的。x就代表能够输入任意数量的MNIST图像，每一张图展平成784维的向量。

5.3 构建模型

样本完成后就可以构建模型了。下面列出了构建模型的相关步骤。

5.3.1 定义学习参数

模型也需要权重值和偏置量，它们被统一叫做学习参数。在TensorFlow里，使用Variable来定义学习参数。

一个Variable代表一个可修改的张量，定义在TensorFlow的图（一个执行任务）中，其本身也是一种变量。使用Variable定义的学习参数可以用于计算输入值，也可以在计算中被修改。

代码5-2 MNIST分类（续）

```
10 w = tf.Variable(tf.random_normal(([784, 10])))
11 b = tf.Variable(tf.zeros([10]))
```

在这里赋予tf.Variable不同的初值来创建不同的参数。一般将W设为一个随机值，将b设为0。



注意： W的维度是[784, 10]，因为想要用784维的图片向量乘以它，以得到一个10维的证
*****ebook converter DEMO Watermarks*****

据值向量，每一位对应不同数字类。b的形状是[10]，所以可以直接把它加到输出上面。

5.3.2 定义输出节点

有了输入和模型参数，接着便可以将它们串起来构建成真正的模型。

代码5-2 MNIST分类（续）

```
12 pred = tf.nn.softmax(tf.matmul(x, w) + b) # Softmax分类
```

首先，用`tf.matmul(x, w)`表示x乘以w，这里x是一个二维张量，拥有多个输入。然后再加上b，把它们的和输入到`tf.nn.softmax`函数里。

至此就构建好了正向传播的结构。也就是表明，只要模型中的参数合适，通过具体的数据输入，就能得到我们想要的分类。

5.3.3 定义反向传播的结构

下面定义一个反向传播的结构，编译训练模型，以得到合适的参数。

这里涉及一个“学习率”的概念。学习率，是指每次改变学习参数的大小。在这里读者只要先

有个概念即可，后面章节还会详细介绍。

先看下面代码。

代码5-2 MNIST分类（续）

```
13 # 损失函数
14 cost=tf.reduce_mean(-tf.reduce_sum(y*tf.log(pred),reduc
15
16 # 定义参数
17 learning_rate = 0.01
18 # 使用梯度下降优化器
19 optimizer=tf.train.GradientDescentOptimizer(learning_ra
```

上面的代码可以这样来理解：

- (1) 将生成的pred与样本标签y进行一次交叉熵的运算，然后取平均值。
- (2) 将这个结果作为一次正向传播的误差，通过梯度下降的优化方法找到能够使这个误差最小化的b和W的偏移量。
- (3) 更新b和W，使其调整为合适的参数。

整个过程就是不断地让损失值（误差值cost）变小。因为损失值越小，才能表明输出的结果跟标签数据越相近。当cost小到我们的需求时，这时的b和W就是训练出来的合适值。

5.4 训练模型并输出中间状态参数

现在开始真正地训练模型了，先定义训练相关的参数。

下面代码中，第20行中，`training_epochs`代表要把整个训练样本集迭代25次；第21行中，`batch_size`代表在训练过程中一次取100条数据进行训练；第22行中，`display_step`代表每训练一次就把具体的中间状态显示出来。



注意： `batch_size`参数代表的意义很关键，在深度学习中，都是将数据按批次地向里面放的。在后面章节中还会详细介绍这么做的目的。

参数定义好后，启动一个`session`就可以开始训练过程了。`session`中有两个`run`，第一个`run`是运行初始化，第二个`run`是运行具体的运算模型。模型运算之后便将里面的状态打印出来。

代码5-2 MNIST分类（续）

```
20 training_epochs = 25
21 batch_size = 100
22 display_step = 1
23
24 # 启动session
```

```
25 with tf.Session() as sess:  
26     sess.run(tf.global_variables_initializer())# Initial:  
27  
28     # 启动循环开始训练  
29     for epoch in range(training_epochs):  
30         avg_cost = 0.  
31         total_batch = int(mnist.train.num_examples/batch_  
32         # 循环所有数据集  
33         for i in range(total_batch):  
34             batch_xs, batch_ys = mnist.train.next_batch(1)  
35             # 运行优化器  
36             _, c = sess.run([optimizer, cost], feed_dict=  
37  
38             # 计算平均loss值  
39             avg_cost += c / total_batch  
40             # 显示训练中的详细信息  
41             if (epoch+1) % display_step == 0:  
42                 print ("Epoch:", '%04d' % (epoch+1), "cost=",  
43                     format(avg_cost))  
44     print( " Finished!")
```

执行上面的代码，会输出如下信息：

```
Epoch: 0001 cost= 9.923389743  
Epoch: 0002 cost= 4.695022035  
Epoch: 0003 cost= 3.076164273  
Epoch: 0004 cost= 2.417567778  
Epoch: 0005 cost= 2.052902991  
Epoch: 0006 cost= 1.816404106  
Epoch: 0007 cost= 1.649224558  
Epoch: 0008 cost= 1.523894480  
Epoch: 0009 cost= 1.425924496  
Epoch: 0010 cost= 1.346838083  
Epoch: 0011 cost= 1.281203090  
Epoch: 0012 cost= 1.225851107  
Epoch: 0013 cost= 1.178292338  
Epoch: 0014 cost= 1.136689923  
Epoch: 0015 cost= 1.100095906  
Epoch: 0016 cost= 1.067396342  
Epoch: 0017 cost= 1.038121746  
Epoch: 0018 cost= 1.011435861  
Epoch: 0019 cost= 0.987299248  
Epoch: 0020 cost= 0.965228878  
Epoch: 0021 cost= 0.944723253
```

*****ebook converter DEMO Watermarks*****

```
Epoch: 0022 cost= 0.925947570
Epoch: 0023 cost= 0.908483106
Epoch: 0024 cost= 0.892120825
Epoch: 0025 cost= 0.877055534
Finished!
```

这里输出的中间状态是cost损失值。读者也可以把自己关心的内容打印出来。可以看到，从第1次迭代到第25次迭代的损失值在逐渐减小，最终的误差只有0.8。

5.5 测试模型

还记得MNIST里面有测试数据吗？现在我们使用测试数据来测试一下训练完的模型吧。

与前面的过程类似，也是先将计算测试的网络结构建立起来，然后通过最终节点的eval将测试值运算出来。



注意：这个过程仍然是在session里进行的。

测试错误率的算法是：直接判断预测的结果与真实的标签是否相同，如是相同的就表明是正确的，如是不相同的就表示是错误的。然后将正确的个数除以总个数，得到的值即为正确率。由于是onehot编码，这里使用了tf.argmax函数返回onehot编码中数值为1的那个元素的下标。下面是具体代码。

代码5-2 MNIST分类（续）

```
45 # 测试 model
46     correct_prediction = tf.equal(tf.argmax(pred, 1), tf.
47         # 计算准确率
48         accuracy = tf.reduce_mean(tf.cast(correct_prediction,
49             print ("Accuracy:", accuracy.eval({x: mnist.test.images,
test.labels}))
```

上面代码执行后，显示信息如下：

```
Accuracy: 0.8316
```

测试正确率的算法与损失值的算法略有差别，但代表的意义却很类似。当然，也可以直接拿计算损失值的交叉熵结果来代表模型测试的错误率。



注意： (1) 并不是所有模型的测试错误率和训练时的最后一次损失值都很接近，这取决于训练样本和测试样本的分布情况，也取决于模型本身的拟合质量。关于拟合质量问题，将在后面章节详细介绍。

(2) 读者自己运行时，得到的值可能和本书中的值不一样。甚至每次运行时，得到的值也不一样。原因是每次初始的权重 w 都是随机的。由于初始权重不同，而且每次训练的批次数据也不同，所以最终生成的模型也不会完全相同。但如果核心算法保持一致，则会保证最终的结果不会有太大的偏差。

5.6 保存模型

下面开始讲解如何保存模型。

首先要建立一个saver和一个路径，然后通过调用save，自动将session中的参数保存起来，见如下代码。

代码5-2 MNIST分类（续）

```
50     # 保存模型
51     save_path = saver.save(sess, model_path)
52     print("Model saved in file: %s" % save_path)
```

上面代码的作用是保存模型，并将模型保存的路径打印出来。当然，在这段代码运行之前，需要添加saver和model_path的定义。来到前面代码段的第30行（也就是session创建之前）添加如下代码：

代码5-2 MNIST分类（续）

```
53 saver = tf.train.Saver()
54 model_path = "log/521model.ckpt"
```

执行上述的全部代码后，会在代码文件的同级目录下找到log文件夹，其中有4个文件，如图

5-4所示。

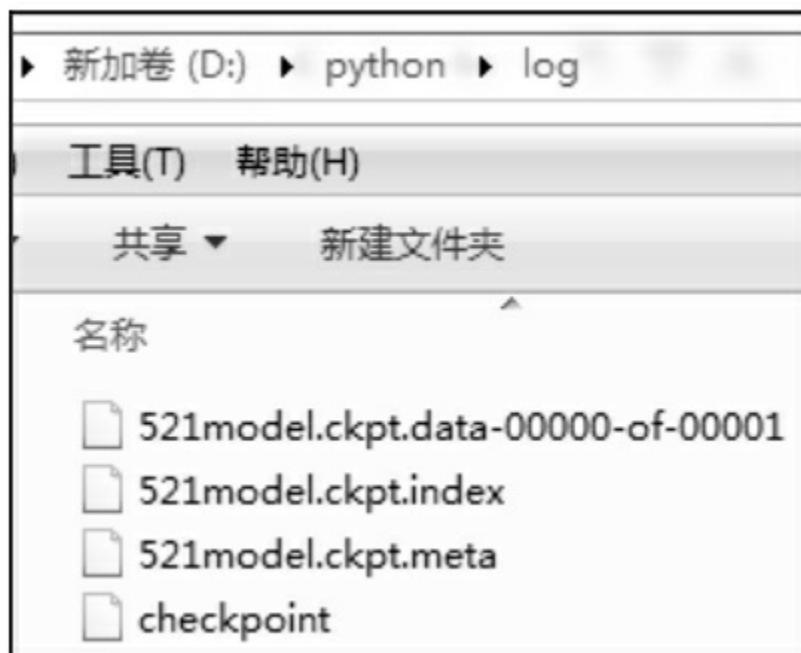


图5-4 模型文件位置

5.7 读取模型

将模型存储好后，下面来做一个实验：读取模型并将两张图片放进去让模型预测结果，然后将两张图片极其对应的标签一并显示出来。

在整个代码执行过程中，对于网络模型的定义不变，只是重新建立一个session而已，所有的操作都在这个新的session中完成。具体细节见代码。

代码5-2 MNIST分类（续）

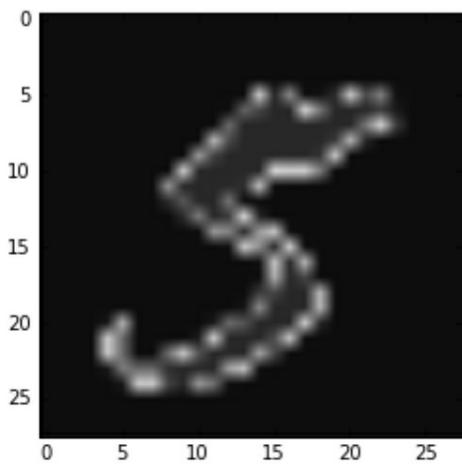
```
55 print("Starting 2nd session...")
56 with tf.Session() as sess:
57     # 初始化变量
58     sess.run(tf.global_variables_initializer())
59     # 恢复模型变量
60     saver.restore(sess, model_path)
61
62     # 测试 model
63     correct_prediction = tf.equal(tf.argmax(pred, 1), tf.
64     # 计算准确率
65     accuracy = tf.reduce_mean(tf.cast(correct_prediction,
66     print ("Accuracy:", accuracy.eval({x: mnist.test.images,
test.labels})) )
67
68     output = tf.argmax(pred, 1)
69     batch_xs, batch_ys = mnist.train.next_batch(2)
70     outputval, predv = sess.run([output, pred], feed_dict={x:
71     print(outputval, predv, batch_ys)
72
73     im = batch_xs[0]
74     im = im.reshape(-1, 28)
75     pylab.imshow(im)
76     pylab.show()
77
```

```
78     im = batch_xs[1]
79     im = im.reshape(-1, 28)
80     pylab.imshow(im)
81     pylab.show()
```

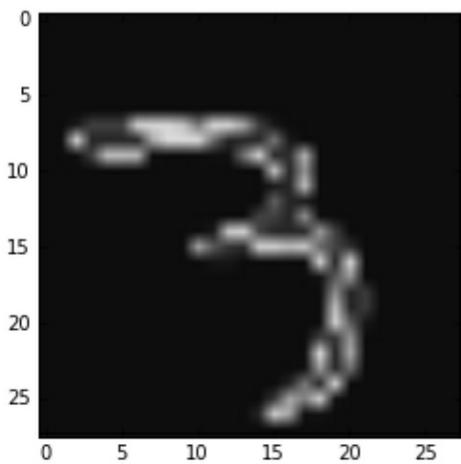
以上代码可以替代原来的session（从第30行到最后），也可以直接放到代码后面，将前面的session注释掉。

运行后可以看到如下信息，结果如图5-5所示。

```
Accuracy: 0.8316
[[ 3.26058798e-05    3.89398069e-09    2.60637262e-06    2.675
   6.77738354e-09    9.70463872e-01    1.54175677e-08    6.382
   1.79426873e-03    3.15453537e-04]
 [ 3.65457054e-10    9.57760785e-04    5.34406379e-02    8.836
   5.11178478e-05    1.06539410e-05    7.34308742e-06    1.402
   1.56633689e-07    4.78818417e-02]]
[[ 0.  0.  0.  0.  0.  1.  0.  0.  0.  0.]
 [ 0.  0.  0.  1.  0.  0.  0.  0.  0.]]
```



a) 预测结果为 5



b) 预测结果为 3

图5-5 运行结果

第一行是模型的准确率，接下来是3个数组。

- 第一个数组是输出的预测结果。
- 第二个大的数组比较大，是预测出来的真实输出值。
- 第三个大的数组元素都是0和1，是标签值onehot编码表示的5和3。



注意： 这里是恰巧举了一个全部正确的例子，因为还有0.17的错误率，所以有时也会有预测错误的情况。

到此我们已经通过两个模型的例子，大体了解了神经网络的作用。那么为什么神经网络会产生这样的效果呢？具体的原理将在后面的章节中一一介绍。

第2篇 深度学习基础——神经网络

本篇将从神经网络中的最基础单元——单个神经元开始，由浅入深地分别介绍各种类型的神经网络，包括多层神经网络、卷积神经网络、循环神经网络和自编码网络。

第6章 单个神经元

第7章 多层神经网络——解决非线性问题

第8章 卷积神经网络——解决参数太多问题

第9章 循环神经网络——具有记忆功能的网络

第10章 自编码网络——能够自学习样本特征的网络

第6章 单个神经元

前面的章节中介绍了TensorFlow框架的基本使用方法。从本章开始，我们将真正进入深度学习理论知识系统。神经网络是由多个神经元组成，所以本章先从一个神经元开始讲起。一个神经元由以下几个关键知识点组成：

- 激活函数；
- 损失函数；
- 梯度下降。

本章含有教学视频共14分56秒。

作者按照本章的内容结构，对主要内容进行了快速讲解，特别是对单个神经元的各个组成部分，以及每个部分的具体实现方法进行了重点讲解（掌握二分类、多分类及非互斥的多分类的实现方法为本章的重点）。



在详细介绍之前，有必要先讲讲神经元的拟合原理。

6.1 神经元的拟合原理

在第5章的代码“5-2 MNIST分类.py”文件中，建立的模型是一个单个神经元组成的网络模型。单个神经元的网络模型如图6-1所示。

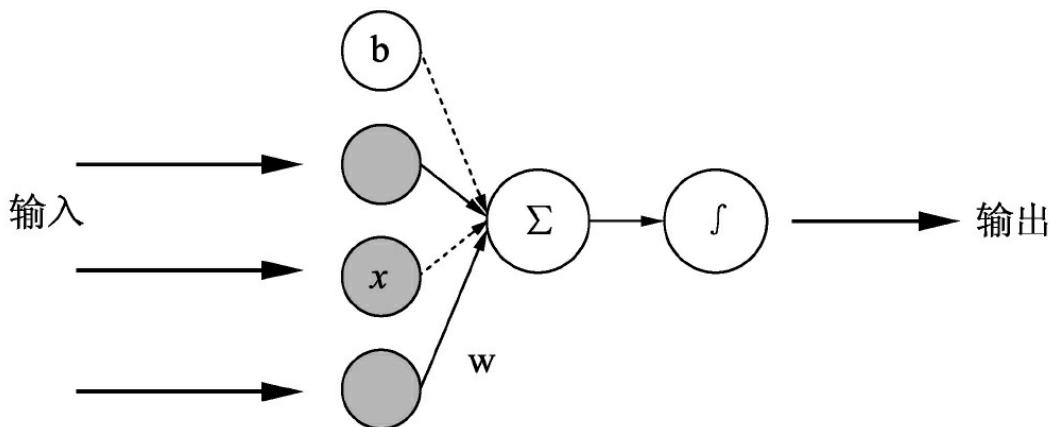


图6-1 单个神经元网络模型

其计算公式如式（6-1）所示。

$$z = \sum_{i=1}^n w_i \times x_i + b = w \cdot x + b \quad \text{式 (6-1)}$$

式（6-1）中：z为输出的结果；x为输入；w为权重；b为偏执值。w和b可以理解为两个变量。

模型每次的学习都是为了调整w和b从而得到一个合适的值，最终由这个值配合运算公式所形成的逻辑就是神经网络的模型。

其实这个模型是根据仿生学得来的。我们看一下大脑细胞里的神经突出如图6-2所示。

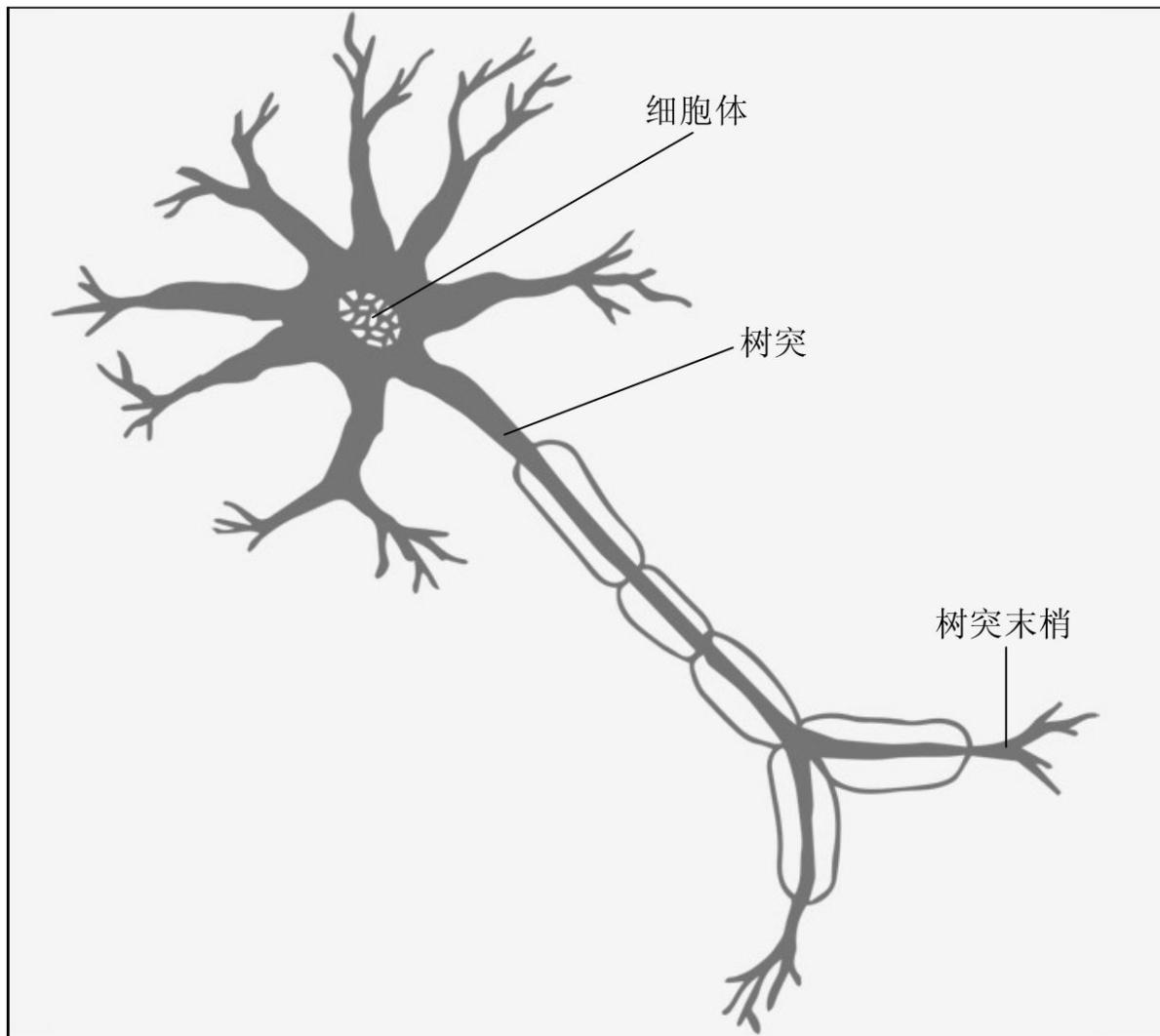


图6-2 神经细胞

是不是与我们建立的模型有点神似？

(1) 大脑神经细胞是靠生物电来传递信号的，可以理解成经过模型里的具体数值。

(2) 仔细观察发现神经细胞相连的连接树突有粗有细，显然通过不同粗细连接的生物电信号，也会有不同的影响。这就好比权重 w ，因为每个输入节点都会与相关连接的 w 相乘，也就实现了对信号的放大、缩小处理。

(3) 这里唯独不透明的就是中间的细胞体，于是我们将所有输入的信号经过 w 变换之后，再添加一个额外的偏执量 b ，把它们加在一起求合，然后再选择一个模拟细胞体处理的函数来实现整个过程的仿真。这个函数称其为激活函数。

我们把 w 和 b 赋予合适的值时，再配合合适的激活函数，就会发现它可以产生很好的拟合效果。

6.1.1 正向传播

前文描述的过程叫做正向传播，数据是从输入到输出的流向传递过来的。当然，它是在一个假设有合适的 w 和 b 的基础上，才可以实现对现实环境的正确拟合。但是，在实际过程中我们无法得知 w 和 b 的值具体是多少才算是正常的。

于是我们加入了一个训练过程，通过反向误差传递的方法让模型自动来修正，最终产生一个合适的权重。

6.1.2 反向传播

反向传播的意义很明确——告诉模型我们需要将 w 和 b 调整到多少。在刚开始没有得到合适的

权重时，正向传播生成的结果与实际的标签是有误差的，反向传播就是要把这个误差传递给权重，让权重做适当地调整来达到一个合适的输出。

在实际训练过程中，很难一次将其调整到位，而是通过多次迭代一点一点的将其修正，最终直到模型的输出值与实际标签值的误差小于某个阀值为止。

如何将输出的误差转化为权重的误差，这里面使用的就是BP算法。

1. BP算法介绍

本书不阐述过多的算法，只讲原理，读者理解道理即可。

BP算法又称“误差反向传播算法”。我们最终的目的，是要让正向传播的输出结果与标签间的误差最小化，这就是反向传播的核心思想。

正向传播的模型是清晰的，所以很容易得出一个关于由 b 和 w 组成的对于输出的表达式。接着，也可以得出一个描述损失值的表达式（将输出值与标签直接相减，或是做平方差等运算）。

为了要让这个损失值变得最小化，我们运用数学知识，选择一个损失值的表达式让这个表达

式有最小值，接着通过对其求导的方式，找到最小值时刻的函数切线斜率（也就是梯度），从而让w和b的值沿着这个梯度来调整。

至于每次调整多少，我们引入一个叫做“学习率”的参数来控制，这样通过不断的迭代，使误差逐步接近最小值，最终达到我们的目标。

6.2 激活函数——加入非线性因素，解决线性模型缺陷

激活函数的主要作用就是用来加入非线性因素的，以解决线性模型表达能力不足的缺陷，在整个神经网络里起到至关重要的作用。

因为神经网络的数学基础是处处可微的，所以选取的激活函数要能保证数据输入与输出也是可微的。

在神经网络里常用的激活函数有Sigmoid、Tanh和relu等，下面逐一介绍。

6.2.1 Sigmoid函数

Sigmoid是常见的激活函数，一起看看它的样子。

1. 函数介绍

Sigmoid是常用的非线性的激活函数，其数学形式见式（6-2）。

$$f(x) = \frac{1}{1 + e^{-x}} \quad \text{式 (6-2)}$$

Sigmoid函数曲线如图6-3所示，其中， x 可以是正无穷到负无穷，但是对应的 y 却只有 $0\sim 1$ 的范围，所以，经过Sigmoid函数输出的函数都会落在 $0\sim 1$ 的区间里，即Sigmoid函数能够把输入的值“压缩”到 $0\sim 1$ 之间。

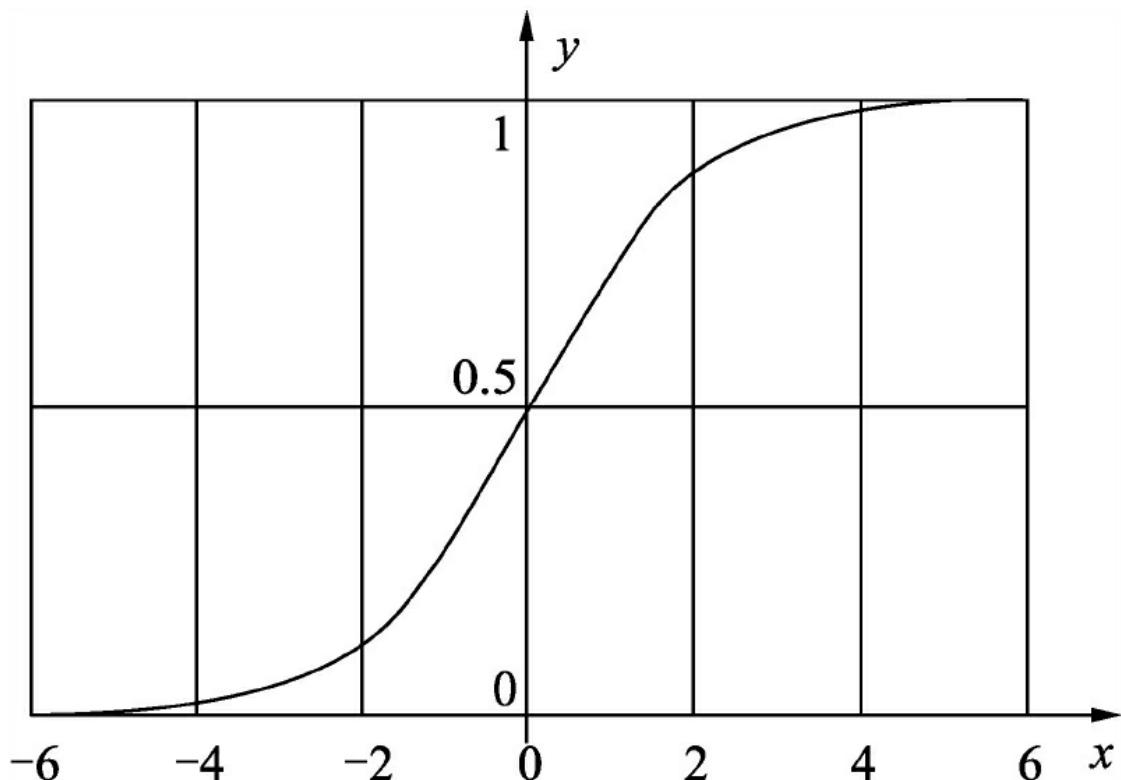


图6-3 Sigmoid函数曲线

2. 在TensorFlow中对应的函数

在TensorFlow中对应的函数为：

```
tf.nn.sigmoid(x, name=None)
```

从图像上看，随着 x 趋近正负无穷大， y 对应

的值越来越接近1或-1，这种情况叫做饱和。处于饱和态的激活函数意味着，当 $x = 100$ 和 $x = 1000$ 时的反映都是一样的，这样的特性转换相当于将1000大于100十倍这个信息给丢失了。

所以，为了能有效使用Sigmoid函数，从图6-3中看其极限也只能是-6~6之间，而在-3~3之间应该会有比较好的效果。

6.2.2 Tanh函数

Tanh函数可以说是Sigmoid函数的值域升级版，由Sigmoid函数的0~1之间升级到-1~1。但是Tanh函数也不能完全替代Sigmoid函数，在某些输出需要大于0的情况下，还是要用Sigmoid函数。

1. 函数介绍

Tanh函数也是常用的非线性激活函数，其数学形式见式（6-3）。

$$\tanh(x) = 2 \text{sigmoid}(2x) - 1 \quad \text{式 (6-3)}$$

Tanh函数曲线如图6-4所示，其x取值也是从正无穷到负无穷，对应的y值变为-1~1之间，相对于Sigmoid函数有更广的值域。

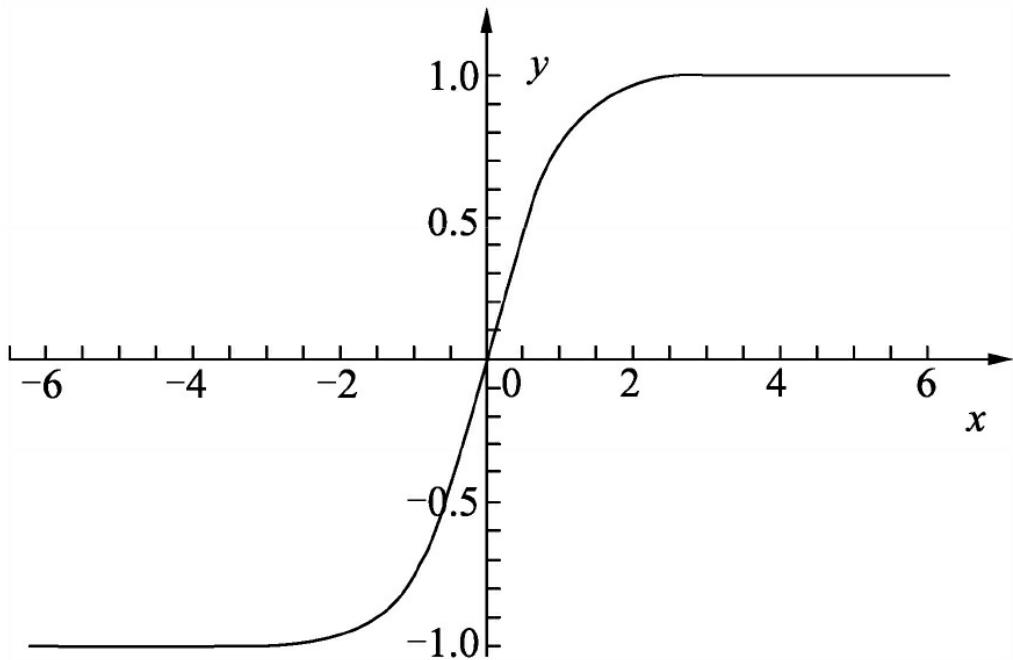


图6-4 Tanh函数曲线

2. 在TensorFlow中对应的函数

在TensorFlow中对应的函数

```
tf.nn.tanh(x, name=None)
```

显而易见，Tanh函数跟Sigmoid函数有一样的缺陷，也是饱和问题，所以在使用Tanh函数时，要注意输入值的绝对值不能过大，否则模型无法训练。

6.2.3 ReLU函数

1. 函数介绍

除了前面介绍的Sigmoid函数和Tanh函数之外，还有一个更为常用的激活函数（也称为Rectifier）。其数学形式见式（6-4）。

$$f(x) = \max(0, x) \quad \text{式 (6-4)}$$

该式非常简单，大于0的留下，否则一律为0，具体的图像如图6-5所示。ReLU函数应用的广泛性与它的优势是分不开的，这种对正向信号的重视，忽略了负向信号的特性，与我们人类神经元细胞对信号的反映极其相似。所以在神经网络中取得了很好的拟合效果。

另外由于ReLU函数运算简单，大大地提升了机器的运行效率，也是Relu函数一个很大的优点。

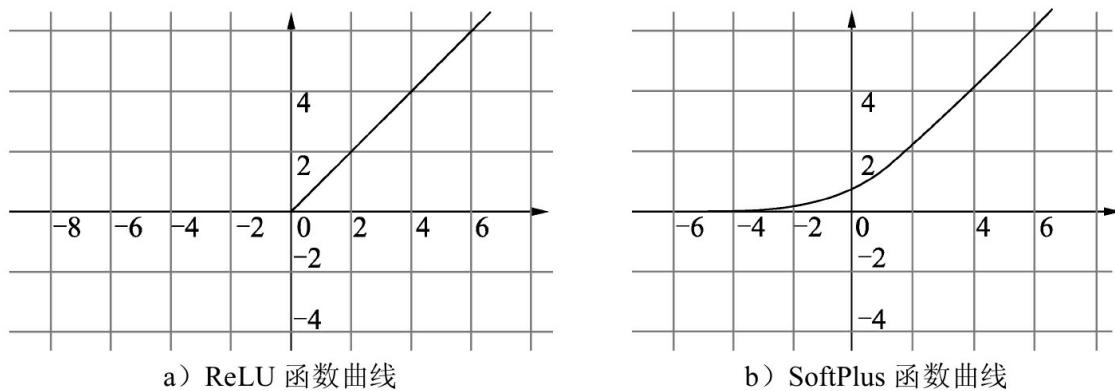


图6-5 ReLU函数和SoftPlus函数曲线

与ReLU函数类似的还有Softplus函数，如图6-5所示。二者的区别在于：Softplus函数会更加平滑，但是计算量很大，而且对于小于0的值保留*****ebook converter DEMO Watermarks*****

的相对更多一点。Softplus函数公式见式（6-5）。

$$f(x) = \ln(1 + e^x) \quad \text{式 (6-5)}$$

虽然ReLU函数在信号响应上有很多优势，但这仅仅在正向传播方面。由于其对负值的全部舍去，因此很容易使模型输出全零从而无法再进行训练。例如，随机初始化的w加入值中有个值是负值，其对应的正值输入值特征也就被全部屏蔽了，同理，对应负值输入值反而被激活了。这显然不是我们想要的结果。于是在基于ReLU的基础上又演化出了一些变种函数，举例如下：

· Noisy relus：为max中的x加了一个高斯分布的噪声，见式（6-6）。

$$f(x) = \max(0, x + Y), Y \in N(0, \sigma(x)) \quad \text{式 (6-6)}$$

· Leaky relus：在ReLU基础上，保留一部分负值，让x为负时乘0.01，即Leaky relus对负信号不是一味地拒绝，而是缩小。其数学形式见式（6-7）。

$$f(x) = \begin{cases} x & (x > 0) \\ 0.01x & (\text{otherwise}) \end{cases} \quad \text{式 (6-7)}$$

· 再进一步让这个0.01作为参数可调，于

是，当x小于0时，乘以a，a小于等于1。其数学形式见式（6-8）。

$$f(x) = \begin{cases} x & (x > 0) \\ ax & (\text{otherwise}) \end{cases} \rightarrow f(x) = \max(x, ax) \quad \text{式 (6-8)}$$

得到Leaky relus的公式 $\max(x, ax)$

· Elus：当x小于0时，做了更复杂的变换，见式（6-9）。

$$f(x) = \begin{cases} x & (x \geq 0) \\ a(e^x - 1) & (\text{otherwise}) \end{cases} \quad \text{式 (6-9)}$$

Elus函数激活函数与ReLU函数一样都是不带参数的，而且收敛速度比ReLU函数更快，使用Elus函数时，不使用批处理比使用批处理能够获得更好的效果，同时Elus函数不使用批处理的效果比ReLU函数加批处理的效果要好。

2. 在TensorFlow中对应的函数

在TensorFlow中，关于ReLU函数的实现，有以下两个对应的函数：

· `tf.nn.relu (features, name=None)` : 是一般的ReLU函数，即 $\max(features, 0)$ ；

· `tf.nn.relu6 (features, name=None)` : 是以

6为阈值的ReLU函数，即 $\min(\max(\text{features}, 0), 6)$ 。



注意： relu6存在的原因是防止梯度爆炸，当节点和层数特别多而且输出都为正时，它们的加和会是一个很大的值，尤其在经历几层变换之后，最终的值可能会离目标值相差太远。误差太大，会导致对参数调整修正值过大，这会导致网络抖动得较厉害，最终很难收敛。

在TensorFlow中，Softplus函数对应的函数如下：

```
tf.nn.softplus(features, name=None);
```

在TensorFlow中，Elus函数对应的函数如下：

```
tf.nn.elu(features, name=None)
```

在TensorFlow中，Leaky relus公式没有专门的函数，不过可以利用现有函数组成而得到：

```
tf.maximum(x, leak*x, name = name) #leak 为传入的参数，可以设为
```

6.2.4 Swish函数

Swish函数是谷歌公司发现的一个效果更优于Relu的激活函数。经过测试，在保持所有的模型参数不变的情况下，只是把原来模型中的ReLU激活函数修改为Swish激活函数，模型的准确率均有提升。其公式见式6-10

$$f(x) = x \times \text{sigmoid}(\beta x) \quad \text{式 (6-10)}$$

其中 β 为x的缩放参数，一般情况取默认值1即可。在使用了BN算法（见8.9.3节）的情况下，还需要对x的缩放值 β 进行调节。

在TensorFlow的低版本中，没有单独的Swish函数，可以手动封装，代码如下：

```
def Swish(x, beta=1):
    return x * tf.nn.sigmoid(x*beta)
```

6.2.5 激活函数总结

神经网络中，运算特征是不断进行循环计算，所以在每代循环过程中，每个神经元的值也是在不断变化的。这就导致了Tanh函数在特征相差明显时的效果会很好，在循环过程中其会不断扩大特征效果并显示出来。

但有时当计算的特征间的相差虽比较复杂却没有明显区别，或是特征间的相差不是特别大时，就需要更细微的分类判断，这时Sigmoid函数的效果就会更好一些。

后来出现的ReLU激活函数的优势是，经过其处理后的数据有更好的稀疏性。即，将数据转化为只有最大数值，其他都为0。这种变换可以近似程度地最大保留数据特征，用大多数元素为0的稀疏矩阵来实现。

实际上，神经网络在不断反复计算中，就变成了ReLU函数在不断尝试如何用一个大多数为0的矩阵来表达数据特征。以稀疏性数据来表达原有数据特征的方法，使得神经网络在迭代运算中能够取得又快又好的效果，所以目前大多用 $\max(0, x)$ 来代替Sigmoid函数。

6.3 softmax算法——处理分类问题

softmax基本上可以算是分类任务的标配。在本节中需要学会softmax为什么能分类，以及如何使用softmax来分类。如果需要比较哪个更重要，当然是学会如何使用会更重要。

6.3.1 什么是softmax

对于前面讲的激活函数，其输出值只有两种（0、1，或-1、1，或0、x），而现实生活中需要对某一问题进行多种分类，例如前面的图片分类例子，这时就需要使用softmax算法。

softmax，看名字就知道，就是如果判断输入属于某一个类的概率大于属于其他类的概率，那么这个类对应的值就逼近于1，其他类的值就逼近于0。该算法的主要应用就是多分类，而且是互斥的，即只能属于其中的一个类。与sigmoid类的激活函数不同的是，一般的激活函数只能分两类，所以可以理解成Softmax是Sigmoid类的激活函数的扩展，其算法见式（6-11）。

$$\text{soft max} = \exp(\logits) / \text{reduce_sum}(\exp(\logits), \text{dim}) \quad \text{式 (6-11)}$$

把所有值用e的n次方计算出来，求和后算每个值占的比率，保证总和为1，一般就可以认为

softmax得出的就是概率。

这里的 $\exp(\text{logits})$ 指的就是 e^{logits} 。



注意：对于要生成的多个类任务中不是互斥关系的任务，一般会使用多个二分类来组成。

6.3.2 softmax原理

softmax原理很简单，如图6-6所示为一个简单的Softmax网络模型，输入 X_1 和 X_2 ，要准备生成 Y_1 、 Y_2 和 Y_3 三个类。

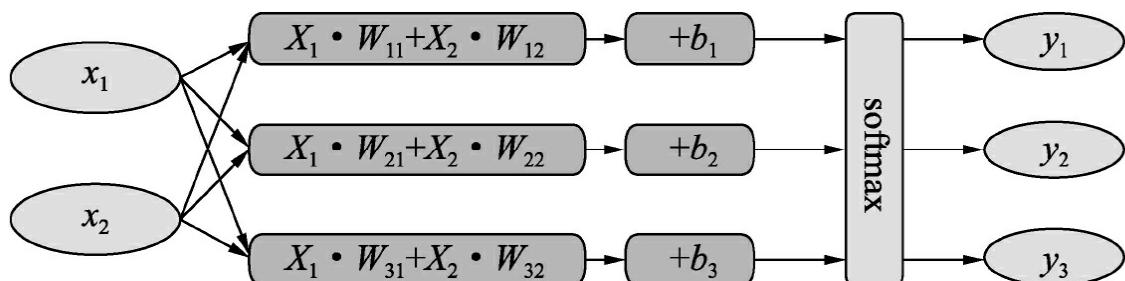


图 6-6 softmax 网络模型

对于属于 y_1 类的概率，可以转化成输入 x_1 满足某个条件的概率，与 x_2 满足某个条件的概率的乘积。

在网络模型里把等式两边都取 \ln 。这样， \ln

后的属于 y_1 类的概率就可以转化成， \ln 后的 x_1 满足某个条件的概率加上 \ln 后的 x_2 满足某个条件的概率。这样 $y_1 = x_1 w_{11} + x_2 w_{12} = \ln$ 后 y_1 的概率了。这也是softmax公式中要进行一次 e 的logits次方的原因。



注意： 等式两边取 \ln 是神经网络中常用的技巧，主要用来将概率的乘法转变成加法，即 $\ln(x^*y) = \ln x + \ln y$ 。然后在后续计算中再将其转为 e 的 x 次方，还原成原来的值。

了解完 e 的 n 次方的意义后，softmax就变得简单至极了。

举例：某个样本经过生成的值 y_1 为5， y_2 为3， y_3 为2。那么对应的概率就为 $y_1 = 5/10 = 0.5$ ， $y_2 = 3/10$ ， $y_3 = 2/10$ ，于是取最大的值 y_1 为最终的分类。

softmax在机器学习中有非常广泛的应用，前面介绍过MNIST的每一张图片都表示一个数字，从0到9。我们希望得到给定图片代表每个数字的概率。例如，训练的模型可能推测一张包含9的图片代表数字9的概率是80%，但是判断它是8的概率是5%（因为8和9都有上半部分相似的小圆），判断它代表其他数字的概率值更小。于是取最大

概率的对应数值，就是这个图片的分类了。这是一个使用softmax回归（softmax regression）模型的经典案例。



注意： 在实际使用中，softmax伴随的分类标签都为one_hot编码，而且这里还有个小技巧，在softmax时需要将目标分成几类，就在最后这层放几个节点。

6.3.3 常用的分类函数

如表6-1中列出了常用的分类函数。

表6-1 常用的分类函数

操作	描述
<code>tf.nn.softmax(logits, name=None)</code>	计算softmax
<code>tf.nn.log_softmax(logits, name=None)</code>	对softmax取对数 $\text{logsoftmax}[i, j] = \text{logits}[i, j] - \log(\sum(\exp(\text{logits}[i])))$

6.4 损失函数——用真实值与预测值的距离来指导模型的收敛方向

损失函数是绝对网络学习质量的关键。在学到后面章节就会发现，无论什么样的网络结构，如果使用的损失函数不正确，最终都将难以训练出正确的模型。这里先介绍几个常见的loss函数，针对不同的网络结构还会有更多的loss函数，在后面章节会伴随不同的网络模型来介绍。

6.4.1 损失函数介绍

损失函数的作用前面已经说过了，用于描述模型预测值与真实值的差距大小。一般有两种比较常见的算法——均值平方差（MSE）和交叉熵。下面来分别介绍每个算法的具体内容。

1. 均值平方差

均值平方差（Mean Squared Error, MSE），也称“均方误差”，在神经网络中主要是表达预测值与真实值之间的差异，在数理统计中，均方误差是指参数估计值与参数真值之差平方的期望值。公式定义见式（6-12），主要是对每一个真实值与预测值相减的平方取平均值：

$$MSE = \frac{1}{n} \sum_{t=1}^n (observe_t - predicted_t)^2 \quad \text{式 (6-12)}$$

均方误差的值越小，表明模型越好。类似的损失算法还有均方根误差RMSE（将MSE开平方）、平均绝对值误差MAD（对一个真实值与预测值相减的绝对值取平均值）等。



注意： 在神经网络计算时，预测值要与真实值控制在同样的数据分布内，假设将预测值经过Sigmoid激活函数得到取值范围在0~1之间，那么真实值也归一化成0~1之间。这样在做loss计算时才会有较好的效果。

2. 交叉熵

交叉熵（crossentropy）也是loss算法的一种，一般用在分类问题上，表达的意识为预测输入样本属于某一类的概率。其表达式见式（6-13），其中y代表真实值分类（0或1），a代表预测值。

$$c = -\frac{1}{n} \sum_x [y \ln a + (1-y) \ln(1-a)] \quad \text{式 (6-13)}$$

交叉熵也是值越小，代表预测结果越准。



注意： 这里用于计算的 a 也是通过分布统一化处理的（或者是经过Sigmoid函数激活的），取值范围在0~1之间。如果真实值和预测值都是1，前面一项 $y * \ln(a)$ 就是 $1 * \ln(1)$ 等于0，后一项 $(1-y) * \ln(1-a)$ 也就是 $0 * \ln(0)$ 等于0，loss为0，反之loss函数为其他数。

3. 总结：损失算法的选取

损失函数的选取取决于输入标签数据的类型：如果输入的是实数、无界的值，损失函数使用平方差；如果输入标签是位矢量（分类标志），使用交叉熵会更适合。

6.4.2 TensorFlow中常见的loss函数

下面看看TensorFlow中都有哪些常见的loss函数。

1. 均值平方差

在TensorFlow没有单独的MSE函数，不过由于公式比较简单，往往开发者都会自己组合，而且也可以写出n种写法，例如：

```
MSE=tf.reduce_mean(tf.pow(tf.sub(logits, outputs), 2.0))
MSE=tf.reduce_mean(tf.square(tf.sub(logits, outputs)))
MSE=tf.reduce_mean(tf.square(logits- outputs))
```

代码中logits代表标签值，outputs代表预测值。

同样也可以组合其他类似loss，例如：

```
Rmse= tf.sqrt(tf.reduce_mean(tf.pow(tf.sub(logits, outputs),  
mad= tf.reduce_mean (tf.complex_abs(tf.sub(logits, outputs))
```

2. 交叉熵

在TensorFlow中常见的交叉熵函数有：

- Sigmoid 交叉熵；
- softmax 交叉熵；
- Sparse 交叉熵；
- 加权Sigmoid 交叉熵。

在TensorFlow里常用的损失函数如表6-2所示。

表6-2 TensorFlow 中的交叉熵

操作	描述
<code>tf.nn.sigmoid_cross_entropy_with_logits (logits,targets , name=None)</code>	计算输入logits和targets的交叉熵
<code>tf.nn.softmax_cross_entropy_with_logits (logits, labels, name=None)</code>	计算logits和labels的softmax交叉熵 Logits和labels必须为相同的shape与数据类型
<code>tf.nn.sparse_softmax_cross_entropy_with_logits (logits, labels, name=None)</code>	计算 logits 和 labels 的 softmax 交叉熵，与 softmax_cross_entropy_with_logits 功能一样，区别在于 sparse_softmax_cross_entropy_with_logits 的样本真实值与预测结果不需要 one-hot 编码，但是要求分类的个数一定要从 0 开始。假如分 2 类，那么标签的预测值只有 0 和 1 这两个数。如果是 5 类，就是 0 1 2 3 4 这 5 个数
<code>tf.nn.weighted_cross_entropy_with_logits (logits, targets, pos_weight, name=None)</code>	在交叉熵的基础上给第一项乘以一个系数（加权），是增加或减少正样本在计算交叉熵时的损失值

当然，也可以像MSE那样使用自己组合的公式计算交叉熵，举例，对于softmax后的结果logits我们可以对其使用公式-

`tf.reduce_sum (labels*tf.log (logits) , 1)`，就等同于softmax_cross_entropy_with_logits得到的结果。

6.5 softmax算法与损失函数的综合应用

在神经网络中使用softmax计算loss时对于初学者常常会犯很多错误，下面通过具体的实例代码来演示需要注意的关键地方与具体的用法。

6.5.1 实例22：交叉熵实验

交叉熵这个比较生僻的术语，在深度学习领域中却是最常见的。由于其常用性，在TensorFlow中会被封装成多个版本，有的公式里直接带了交叉熵，有的需要自己单独求出，而在构建模型时，如果读者对这块知识不扎实，出现问题时会很难分析是模型的问题还是交叉熵的使用问题。因此这里有必要通过几个小实例将其弄得更明白一些。

实例描述

下面一段代码，假设有一个标签labels和一个网络输出值logits。

这个实例就是以这两个值来进行以下3次实验。

(1) 两次softmax实验：将输出值logits分别进行1次和2次softmax，观察两次的区别及意义。

(2) 观察交叉熵：将步骤（1）中的两个值分别进行softmax_cross_entropy_with_logits，观察它们的区别。

(3) 自建公式实验：将做两次softmax的值放到自建组合的公式里得到正确的值。

代码6-1 softmax应用

```
01 import tensorflow as tf
02
03 labels = [[0,0,1],[0,1,0]]
04 logits = [[2, 0.5, 6],
05            [0.1, 0, 3]]
06 logits_scaled = tf.nn.softmax(logits)
07 logits_scaled2 = tf.nn.softmax(logits_scaled)
08
09 result1 = tf.nn.softmax_cross_entropy_with_logits(labels=
09           logits=logits)
10 result2 = tf.nn.softmax_cross_entropy_with_logits(labels=
10           logits=logits_scaled)
11 result3 = -tf.reduce_sum(labels*tf.log(logits_scaled),1)
12
13 with tf.Session() as sess:
14     print ("scaled=",sess.run(logits_scaled))
15     print ("scaled2=",sess.run(logits_scaled2))
#经过第二次的softmax后，分布概率会有变化
16
17     print ("rel1=",sess.run(result1),"\n") #正确的方式
18     print ("rel2=",sess.run(result2),"\n")
#如果将softmax变换完的值放进去会，就相当于算第二次softmax的loss，所
19     print ("rel3=",sess.run(result3))
```

运行上面代码，输出结果如下：

```
scaled= [[ 0.01791432  0.00399722  0.97808844]
[ 0.04980332  0.04506391  0.90513283]]
scaled2= [[ 0.21747023  0.21446465  0.56806517]
```

*****ebook converter DEMO Watermarks*****

```
[ 0.2300214  0.22893383  0.54104471]  
rel1= [ 0.02215516  3.09967351]  
rel2= [ 0.56551915  1.47432232]  
rel3= [ 0.02215518  3.09967351]
```

可以看到：logits里面的值原本加和都是大于1的，但是经过softmax之后，总和变成了1。样本中第一个是跟标签分类相符的，第二与标签分类不符，所以第一个的交叉熵比较小，是0.02215516，而第二个比较大，是3.09967351。

下面开始验证下前面所说的实验：

- 比较scaled和scaled2可以看到：经过第二次的softmax后，分布概率会有变化，而scaled才是我们真实转化的softmax值。
- 比较rel1和rel2可以看到：传入softmax_cross_entropy_with_logits的logits是不需要进行softmax的。如果将softmax后的值scaled传入softmax_cross_entropy_with_logits就相当于进行了两次的softmax转换。

对于已经用softmax转换过的scaled，在计算loss时就不能在用TensorFlow里面的softmax_cross_entropy_with_logits了。读者可以自己写一个loss函数，参见rel3的生成，通过自己组合的函数实现了softmax_cross_entropy_with_logits一样的结果。

6.5.2 实例23：one_hot实验

输入的标签也可以不是标准的one-hot。下面用一组总和也是1但是数组中每个值都不等于0或1的数组来代替标签，看看效果。

实例描述

对非one-hot编码为标签的数据进行交叉熵的计算，比较其与one-hot编码的交叉熵之间的差别。

接上述代码，将标签换为[[0.4, 0.1, 0.5], [0.3, 0.6, 0.1]]与原始的[[0, 0, 1], [0, 1, 0]]代表的分类意义等价，将这个标签代入交叉熵。

代码6-1 softmax应用（续）

```
20 #标签总概率为1
21 labels = [[0.4, 0.1, 0.5], [0.3, 0.6, 0.1]]
22 result4 = tf.nn.softmax_cross_entropy_with_logits(labels=
logits=logits)
23 with tf.Session() as sess:
24     print ("rel4=", sess.run(result4), "\n")
```

运行上面的代码，生成结果如下：

```
rel4= [ 2.17215538  2.76967359]
```

比较前面的rel1发现，对于正确分类的交叉熵和错误分类的交叉熵，二者的结果差别没有标准one-hot那么明显。

6.5.3 实例24：sparse交叉熵的使用

下面再举个例子看一下
sparse_softmax_cross_entropy_with_logits函数的用法，它需要使用非one-hot的标签，所以，要把前面的标签换成具体数值[2, 1]，具体代码如下。

实例描述

使用sparse_softmax_cross_entropy_with_logits函数，对非one-hot的标签进行交叉熵计算，比较其与one-hot标签在使用上的区别。

代码6-1 softmax应用（续）

```
25 #sparse 标签
26 labels = [2, 1] #表明labels中总共分为3个类： 0 、 1、 2。 [2,1] ^_
 编码中的001与010
27 result5 = tf.nn.sparse_softmax_cross_entropy_with_logits(
 logits=logits)
28 with tf.Session() as sess:
29     print ("rel5=", sess.run(result5), "\n")
```

运行代码，生成结果如下：

```
rel5= [ 0.02215516  3.09967351]
```

*****ebook converter DEMO Watermarks*****

发现rel5与前面的rel1结果完全一样。

6.5.4 实例25：计算loss值

在真正的神经网络中，得到代码6-1中的一个数组并不能满足要求，还需要对其求均值，使其最终变成一个具体的数值。

实例描述

演示通过分别对前面交叉熵结果result1与softmax后的结果logits_scaled计算loss，验证如下结论：

- (1) 对于softmax_cross_entropy_with_logits后的结果求loss直接取均值。
- (2) 对于softmax后的结果使用-
`tf.reduce_sum (labels * tf.log (logits_scaled))` 求 loss。
- (3) 对于softmax后的结果使用-
`tf.reduce_sum (labels*tf.log (logits_scaled) , 1)` 等同于softmax_cross_entropy_with_logits结果。
- (4) 由 (1) 和 (3) 可以推出对 (3) 进行

求均值也可以得出正确的loss值，合并起来的公式为：
tf.reduce_sum (-
tf.reduce_sum (labels*tf.log (logits_scaled) ,
1)) =loss (该结论是由前面的验证推导出来，
有兴趣的读者可以自行验证)

代码6-1 softmax应用（续）

```
30 loss=tf.reduce_sum(result1)
31     with tf.Session() as sess:
32         print ("loss=",sess.run(loss))
```

运行上面的代码，生成结果如下：

```
loss= 3.12183
```

这便是我们最终要得到的损失值了。

而对于rel3这种已经求得softmax的情况求loss，可以把公式进一步简化成：

```
loss2 = -tf.reduce_sum(labels * tf.log(logits_scaled))
```

接着添加示例代码。

代码6-1 softmax应用（续）

```
33 labels = [[0,0,1],[0,1,0]]  
34 loss2 = -tf.reduce_sum(labels * tf.log(logits_scaled))  
35 with tf.Session() as sess:  
36     print ("loss2=",sess.run(loss2))
```

运行上面代码，输出结果如下：

```
loss2= 3.12183
```

与loss的值完全吻合。

6.5.5 练习题

试着将上一章的代码（5-2minist分类.py）改成使用sparse_softmax_cross_entropy_with_logits函数来运算交叉熵。

答案请参考本书源代码中的代码“6-2 sparesoftmaxwithminist.py”。

6.6 梯度下降——让模型逼近最小偏差

前面的例子中都提到了梯度下降，但不系统。本节将更详细地介绍梯度下降的作用及常用技巧。

6.6.1 梯度下降的作用及分类

梯度下降法是一个最优化算法，通常也称为最速下降法，常用于机器学习和人工智能中递归性地逼近最小偏差模型，梯度下降的方向也就是用负梯度方向为搜索方向，沿着梯度下降的方向求解极小值。

在训练过程中，每次的正向传播后都会得到输出值与真实值的损失值，这个损失值越小，代表模型越好，于是梯度下降的算法就用在这里，帮助寻找最小的那个损失值，从而可以反推出对应的学习参数 b 和 w ，达到优化模型的效果。

常用的梯度下降方法可以分为：批量梯度下降、随机梯度下降和小批量梯度下降。

· 批量梯度下降：遍历全部数据集算一次损失函数，然后算函数对各个参数的梯度和更新梯度。这种方法每更新一次参数，都要把数据集里的所有样本看一遍，计算量大，计算速度慢，不
*****ebook converter DEMO Watermarks*****

支持在线学习，称为Batch gradient descent，批梯度下降。

- 随机梯度下降：每看一个数据就算一下损失函数，然后求梯度更新参数，这称为stochastic gradient descent，随机梯度下降。这个方法速度比较快，但是收敛性能不太好，可能在最优点附近晃来晃去，命中不到最优点。两次参数的更新也有可能互相抵消，造成目标函数震荡比较剧烈。

- 小批量梯度下降：为了克服上面两种方法的缺点，一般采用一种折中手段——小批的梯度下降。这种方法把数据分为若干个批，按批来更新参数，这样一批中的一组数据共同决定了本次梯度的方向，下降起来就不容易跑偏，减少了随机性。另一方面因为批的样本数与整个数据集相比小了很多，计算量也不是很大。

6.6.2 TensorFlow中的梯度下降函数

下面重点介绍在TensorFlow中进行随机梯度下降优化的函数。

在TensorFlow中是通过一个叫做Optimizer的优化器类进行训练优化的。对于不同算法的优化器，在TensorFlow中会有不同的类，如表6-3所示。

表6-3 梯度下降优化器

操作	描述
<code>tf.train.GradientDescentOptimizer(learning_rate,use_locking=False, name='GradientDescent')</code>	一般的梯度下降算法的Optimizer
<code>tf.train.AdadeltaOptimizer(learning_rate=0.001,rho=0.95, epsilon=1e-08, use_locking=False, name='Adadelta')</code>	创建Adadelta优化器
<code>tf.train.AdagradOptimizer(learning_rate,initial_accumulator_value=0.1, use_locking=False, name='Adagrad')</code>	创建Adagrad优化器
<code>tf.train.MomentumOptimizer(learning_rate,momentum,use_locking=False,name='Momentum',use_nesterov=False)</code>	创建momentum优化器 momentum: 动量,一个Tensor或者浮点值
<code>tf.train.AdamOptimizer(learning_rate=0.001,beta1=0.9, beta2=0.999, epsilon=1e-08, use_locking=False, name='Adam')</code>	创建Adam优化器
<code>tf.train.FtrlOptimizer(learning_rate,learning_rate_power=-0.5, initial_accumulator_value=0.1,l1_regularization_strength=0.0, l2_regularization_strength=0.0, use_locking=False, name='Ftrl')</code>	创建FTRL算法优化器
<code>tf.train.RMSPropOptimizer(learning_rate,decay=0.9,momentum=0.0, epsilon=1e-10, use_locking=False, name='RMSProp')</code>	创建RMSProp算法优化器

在训练过程中，先实例化一个优化函数如 `tf.train.GradientDescentOptimizer`，并基于一定的学习率进行梯度优化训练：

```
optimizer = tf.train.GradientDescentOptimizer(learning_rate)
```

接着使用一个 `minimize()` 的操作，里面传入损失值节点 `loss`，再启动一个外层的循环，优化器就会按照循环的次数一次次沿着 `loss` 最小值的方向优化参数了。

整个过程中的求导和反向传播操作，都是在优化器里自动完成的。目前比较常用的优化器为 Adam 优化器。关于 Adam 的算法不在本书的介绍范围之内，有兴趣的读者可以参考相关资料扩充知识。

6.6.3 退化学习率——在训练的速度与精度之间找到平衡

前面介绍的每个优化器的第一个参数 learning_rate 就是代表学习率。

设置学习率的大小，是在精度和速度之间找到一个平衡：

- 如果学习率的值比较大，则训练速度会提升，但结果的精度不够；
- 如果学习率的值比较小，精度虽然提升了，但训练会耗费太多的时间。

下面就来介绍设置学习率的方法——退化学习率。

退化学习率又叫学习率衰减，它的本意是希望在训练过程中对于学习率大和小的优点都能够为我们所用，也就是当训练刚开始时使用大的学习率加快速度，训练到一定程度后使用小的学习率来提高精度，这时可以使用学习率衰减的方法：

```
def exponential_decay(learning_rate, global_step, decay_steps,
                      staircase=False, name=None):
```

学习率的衰减速度是由global_step和decay_steps来决定的。具体的计算公式如下：

```
decayed_learning_rate = learning_rate * decay_rate ^ (global_
```

staircase值默认为False。当为True时，将没有衰减功能，只是使用上面的公式初始化一个学习率的值而已。

例如下面的代码：

```
learning_rate = tf.train.exponential_decay(starter_learning_step, 100000, 0.96)
```

这种方式定义的学习率就是退化学习率，它的意思是当前迭代到global_step步，学习率每一步都按照每10万步缩小到0.96%的速度衰退。

有时还需要对已经训练好的模型进行微调，可以指定不同层使用不同的学习率，这个在后面章节中会详细介绍。



注意： 通过增大批次处理样本的数量也可以起到退化学习率的效果。但是这种方法要求训练时的最小批次要与实际应用中的最小批次一致。一旦满足该条件时，建议优先选择增大批次数量的方法，因为这样会省去一些开发量和训练

*****ebook converter DEMO Watermarks*****

中的计算量。

6.6.4 实例26：退化学习率的用法举例

本例主要是演示学习率衰减的使用方法。

本例中使用迭代循环计数变量global_step来标记循环次数，初始学习率为0.1，令其以每10次衰减0.9的速度来进行退化。

实例描述

定义一个学习率变量，将其衰减系数设置好，并设置好迭代循环的次数，将每次迭代运算的次数与学习率打印出来，观察学习率按照次数退化的现象。

代码6-3 退化学习率

```
import tensorflow as tf
global_step = tf.Variable(0, trainable=False)
initial_learning_rate = 0.1 #初始学习率
learning_rate = tf.train.exponential_decay(initial_learning_
                                             global_step=global_
                                             decay_steps=10, de
opt = tf.train.GradientDescentOptimizer(learning_rate)
add_global = global_step.assign_add(1) #定义一个op，令global_
with tf.Session() as sess:
    tf.global_variables_initializer().run()
    print(sess.run(learning_rate))
    for i in range(20):
        g, rate = sess.run([add_global, learning_rate])
        print(g,rate)
```

上面代码运行如下：

```
0.1  
1 0.1  
2 0.0989519  
3 0.0979148  
4 0.0968886  
5 0.0958732  
6 0.0948683  
7 0.093874  
8 0.0928902  
9 0.0919166  
10 0.0909533  
11 0.09  
12 0.0890567  
13 0.0881234  
14 0.0871998  
15 0.0862858  
16 0.0853815  
17 0.0844866  
18 0.0836011  
19 0.082725  
20 0.0818579
```

第1个数是迭代的次数，第2个输出是学习率。可以看到学习率在逐渐变小，在第11次由原来的0.1变为了0.09。



注意：这是一种常用的训练策略，在训练神经网络时，通常在训练刚开始时使用较大的 learning rate，随着训练的进行，会慢慢减小 learning rate。在使用时，一定要把当前迭代次数 global_step 传进去，否则不会有退化的功能。

6.7 初始化学习参数

在定义学习参数时可以通过get_variable和Variable两个方式，对于一个网络模型，参数不同的初始化情况，对网络的影响会很大，所以在TensorFlow提供了很多具有不同特性的初始化函数。在使用get_variable时，get_variable的定义如下：

```
def get_variable(name,
                 shape=None,
                 dtype=None,
                 initializer=None,
                 regularizer=None,
                 trainable=True,
                 collections=None,
                 caching_device=None,
                 partitioner=None,
                 validate_shape=True,
                 use_resource=None,
                 custom_getter=None)
```

其中，参数initializer就是初始化参数，可以取表6-4中列出的相关函数。

表6-4 初始话函数

操作	描述
tf.constant_initializer(value)	初始化一切所提供的值
tf.random_uniform_initializer(a, b)	从a到b均匀初始化
tf.random_normal_initializer(mean, stddev)	用所给平均值和标准差初始化均匀分布
tf.constant_initializer(value=0, dtype=tf.float32)	初始化常量

(续)

操作	描述
<code>tf.random_normal_initializer(mean=0.0,stddev=1.0, seed=None, dtype=tf.float32)</code>	正太分布随机数，均值mean,标准差stddev
<code>tf.truncated_normal_initializer(mean=0.0,stddev=1.0, seed=None, dtype=tf.float32)</code>	截断正态分布随机数，均值mean,标准差stddev， 不过只保留[mean-2*stddev,mean+2*stddev]范围 内的随机数
<code>tf.random_uniform_initializer(minval=0,maxval=None, seed=None, dtype=tf.float32)</code>	均匀分布随机数，范围为[minval,maxval]
<code>tf.uniform_unit_scaling_initializer(factor=1.0, seed=None, dtype=tf.float32)</code>	满足均匀分布，但不影响输出数量级的随机值
<code>tf.zeros_initializer(shape,dtype=tf.float32, partition_info=None)</code>	初始化为0
<code>tf.ones_initializer(dtype=tf.float32,partition_info=None)</code>	初始化为1
<code>tf.orthogonal_initializer(gain=1.0,dtype=tf.float32, seed=None)</code>	生成正交矩阵的随机数 当需要生成的参数是二维时，这个正交矩阵是 由均匀分布的随机数矩阵经过SVD分解而来

另外，在`tf.contrib.layers`函数中还有个`tf.contrib.layers.xavier_initializer`初始化函数，用来在所有层中保持梯度大体相同。尤其在深度神经网络里会经常使用（后面卷积内容的章节中还会提到该函数）。

对于`Variable`定义的变量，可以使用表4-7中的相关函数进行初始化。



注意：一般常用的初始化函数为`tf.truncated_normal`函数，因为该函数有截断功能，可以生成相对比较温和的初始值。

6.8 单个神经元的扩展——Maxout网络

在早期，单个神经元出现之后，为了得到更好的拟合效果，又出现了一种Maxout网络，下面具体介绍。

6.8.1 Maxout介绍

Maxout网络可以理解为单个神经元的扩展，主要是扩展单个神经元里面的激活函数，正常的单个神经元如图6-7所示。

Maxout是将激活函数变成一个网络选择器，原理就是将多个神经元并列地放在一起，从它们的输出结果中找到最大的那个，代表对特征响应最敏感，然后取这个神经元的结果参与后面的运算，如图6-8所示。

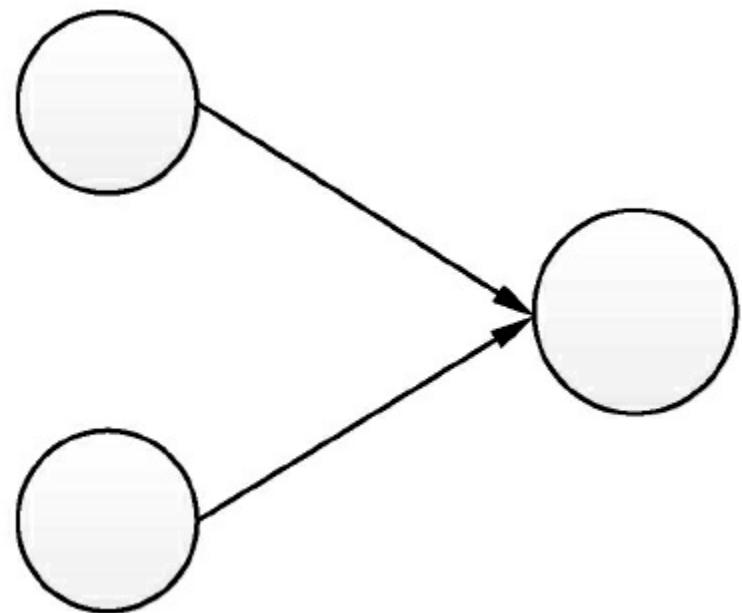


图6-7 单个神经元

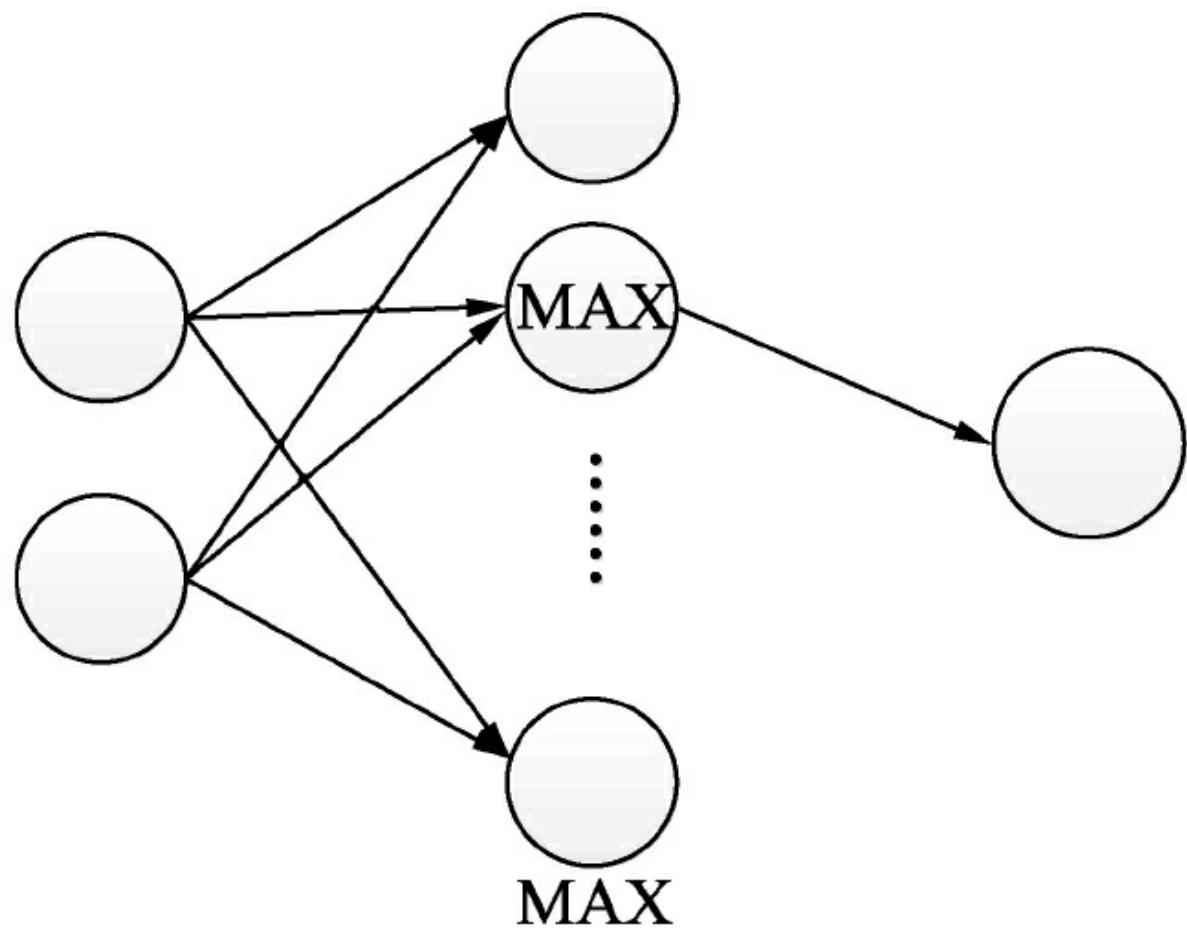


图6-8 Maxout网络

它的公式可以理解成：

```
z1=w1*x+b1  
z2=w2*x+b2  
z3=w3*x+b3  
z4=w4*x+b4  
z5=w5*x+b5  
.....  
out=max(z1,z2,z3,z4,z5.....)
```

为什么要这样做呢？在前面我们学习了一个神经元的作用，类似人类的神经细胞，不同的神经元会因为输入的不同而产生不同的输出，即不同的细胞关心的信号不同。依赖于这个原理，现在的做法就是相当于同时使用多个神经元放在一起，哪个有效果就用哪个。所以这样的网络会有更好的拟合效果。

6.8.2 实例27：用Maxout网络实现MNIST分类

本例主要是演示Maxout网络的构建方法。

本例中以6.5节的练习题答案来修改代码，在本书源代码中的代码“6-2 sparesoftmaxwithminist.py”文件里做如下改动。

实例描述

Maxout网络的构建方法：通过reduce_max函数对多个神经元的输出来计算Max值，将Max值

当作输入按照神经元正反传播方向进行计算。

通过上述方法构建Maxout网络，实现MNIST分类。

代码6-4 Maxout网络实现MNIST分类

```
....  
z= tf.matmul(x, w) + b  
  
maxout = tf.reduce_max(z, axis= 1, keep_dims=True)  
# 设置学习参数  
W2 = tf.Variable(tf.truncated_normal([1, 10], stddev=0.1))  
b2 = tf.Variable(tf.zeros([1]))  
# 构建模型  
pred = tf.nn.softmax(tf.matmul(maxout, W2) + b2)  
  
....  
learning_rate = 0.04  
#使用一般梯度下降方法的优化器  
optimizer = tf.train.GradientDescentOptimizer(learning_rate)  
(cost)  
  
training_epochs = 200  
batch_size = 100  
display_step = 1  
....
```

在网络模型部分，添加一层Maxout，然后将Maxout作为maxsoft的交叉熵输入。学习率设为0.04，迭代次数设为200。运行代码，得到如下结果：

```
Epoch: 0001 cost= 5.160553925  
Epoch: 0002 cost= 1.797463597  
....  
Epoch: 0198 cost= 0.290569865  
Epoch: 0199 cost= 0.290143878
```

```
Epoch: 0200 cost= 0.289847674
Finished!
```

可以看到损失值下降到0.28，随着迭代次数的增加还会继续下降。有兴趣的读者可以自己接着优化。

Maxout的拟合功能很强大，但是也会有节点过多、参数过多、训练过慢的缺点。在第7章中还会学习一种类似于Maxout的全连接网络，会更深刻地讨论拟合的方法及意义。

6.9 练习题

在了解这么多比较零散的知识点以后，最重要的是熟练掌握它们，读者可以将前面讲过的例子拿出来，通过调节学习率、改变激活函数、调节最小批次的方法，试着去改变模型，看看会得到什么不同的结果。

第7章 多层神经网络——解决非线性问题

第6章通过实验证明了单层神经网络的拟合功能。但是在实际环境中，发现这种拟合的效果极其有限。对于某些样本，即便是Maxout也无法解决问题。追究根本，源于样本本身的特性，即单层神经网络只能解决对线性可分的问题。

本章将介绍如何使用多层神经网络来解决非线性问题。

本章含有教学视频共6分35秒。

作者按照本章的内容结构，对主要内容进行了快速讲解，包括多层神经网络与单层神经网络的结构区分及功能能区分、线性与非线性的概念、拟合与过拟合的效果（重点掌握多层网络的拟合原理及训练方法）等。

深度学习之TensorFlow

入门、原理与进阶实战

第7章 多层神经网络 ——解决非线性问题

配套视频



代码医生

qq群: 40016981

Blog: <http://blog.csdn.net/ljjin6249>



字幕



*****ebook converter DEMO Watermarks*****

7.1 线性问题与非线性问题

“线性问题”与“非线性问题”是神经网络中的常用术语。为了能够更准确地解释它们，咱们先从一个例子入手。

7.1.1 实例28：用线性单分逻辑回归分析肿瘤是良性还是恶性的

在介绍线性逻辑回归例子之前，我们先利用第6章所学的知识，做下面的这个分类任务。

实例描述

假设某肿瘤医院想用神经网络对已有的病例数据进行分类，数据的样本特征包括病人的年龄和肿瘤的大小，对应的标签为该病例是良性肿瘤还是恶性肿瘤。

1. 生成样本集

对于这个任务，大家可能迫不及待地想用我们所学的模型试试了吧。这里因为没有医院的病例数据，为了方便演示，先用Python生成一些模拟数据来代替样本，它应该是个二维的数组“病人的年纪，肿瘤的大小”。代码7-1中，`generate`为生成模拟样本的函数，意思是按照指定的均值和方

差生成固定数量的样本。

代码7-1 线性逻辑回归

```
01 def generate(sample_size, mean, cov, diff, regression):
02     num_classes = 2
03     samples_per_class = int(sample_size/2)
04
05     X0 = np.random.multivariate_normal(mean, cov, samples_per_class)
06     Y0 = np.zeros(samples_per_class)
07
08     for ci, d in enumerate(diff):
09         X1 = np.random.multivariate_normal(mean+d, cov, samples_per_class)
10         Y1 = (ci+1)*np.ones(samples_per_class)
11
12         X0 = np.concatenate((X0,X1))
13         Y0 = np.concatenate((Y0,Y1))
14
15     if regression==False: #one-hot编码, 将0转成1 0
16         class_ind = [Y==class_number for class_number in
17                     classes]
18         Y = np.asarray(np.hstack(class_ind), dtype=np.float)
19
20     X, Y = shuffle(X0, Y0)
21
22     return X, Y
```

下面代码是调用generate函数生成1000个数据，并将它们图示化。

- 定义随机数的种子值（这样可以保证每次运行代码时生成的随机值都一样），见代码21行。
- 定义生成类的个数num_classes=2，见代码22行。

- 代码25行中的3.0是表明两类数据的x和y差距3.0。传入的最后一个参数regression =True表明使用非one-hot的编码标签。

代码7-1 线性逻辑回归（续）

```
21 np.random.seed(10)
22 num_classes =2
23 mean = np.random.randn(num_classes)
24 cov = np.eye(num_classes)
25 X, Y = generate(1000, mean, cov, [3.0],True)
26 colors = ['r' if l == 0 else 'b' for l in Y[:]]
27 plt.scatter(X[:,0], X[:,1], c=colors)
28 plt.xlabel("Scaled age (in yrs)")
29 plt.ylabel("Tumor size (in cm)")
30 plt.show()
31 lab_dim = 1
```

运行上面的代码，得到的结果如图7-1所示。

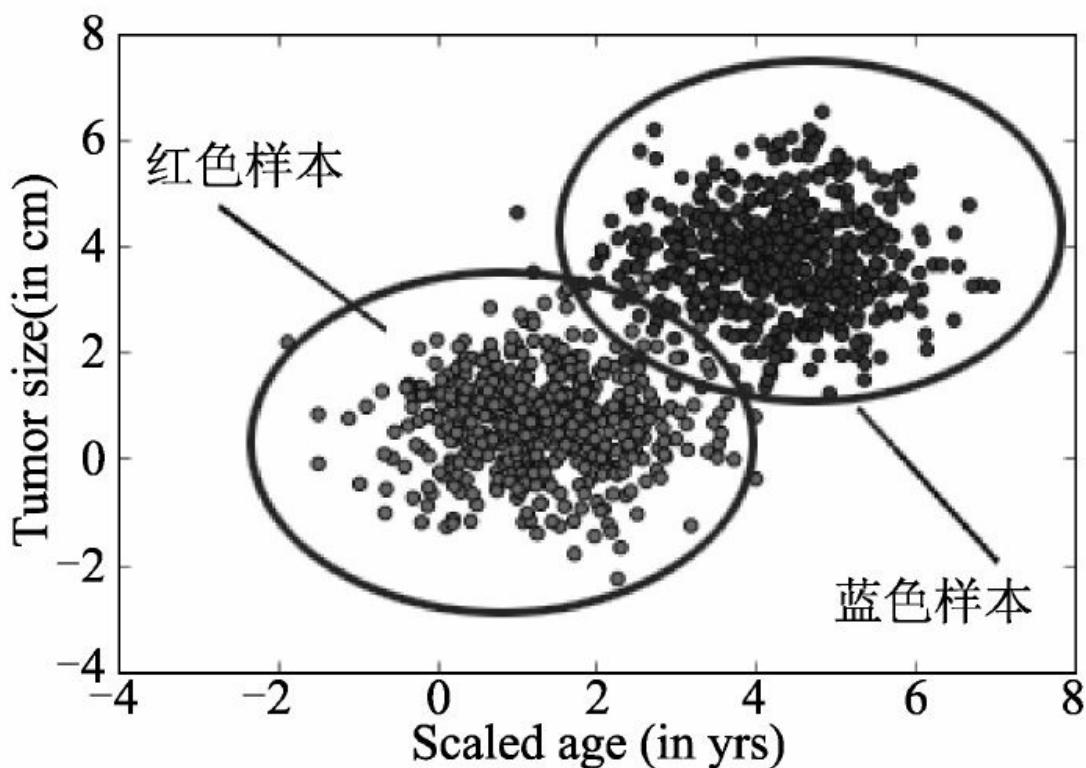


图7-1 模拟数据集

图7-1中左下是红色的样本数据，右上是蓝色的样本数据。

2. 构建网络结构

下面开始构建网络模型，见下方代码。

使用前面刚刚学过的一个神经元，先定义输入、输出两个占位符，然后是w和b的权重。

- 激活函数使用的是Sigmoid，见代码第38行。
- 损失函数loss仍然使用交叉熵，见代码第41行，里面又加了一个平方差函数，用来评估模型的错误率。
- 优化器使用AdamOptimizer，见代码第43行。

代码7-1 线性逻辑回归（续）

```
32 input_features = tf.placeholder(tf.float32, [None, input_
33 input_labels = tf.placeholder(tf.float32, [None, lab_dim]
34 # 定义学习参数
35 w = tf.Variable(tf.random_normal([input_dim, lab_dim]), name="weight")
36 b = tf.Variable(tf.zeros([lab_dim]), name="bias")
37
38 output = tf.nn.sigmoid(tf.matmul(input_features, w) + b)
39 cross_entropy = -(input_labels * tf.log(output) + (1 - i
```

```
* tf.log(1 - output))
40 ser= tf.square(input_labels - output)
41 loss = tf.reduce_mean(cross_entropy)
42 err = tf.reduce_mean(ser)
43 optimizer = tf.train.AdamOptimizer(0.04)
#尽量用这个，因其收敛快，会动态调节梯度
44 train = optimizer.minimize(loss)
```

3. 设置参数进行训练

令整个数据集迭代50次，每次的minibatchsize取25条。

代码7-1 线性逻辑回归（续）

```
45 maxEpochs = 50
46 minibatchSize = 25
47
48 # 启动session
49 with tf.Session() as sess:
50     sess.run(tf.global_variables_initializer())
51
52     # 向模型输入数据
53     for epoch in range(maxEpochs):
54         sumerr=0
55         for i in range(np.int32(len(Y)/minibatchSize)):
56             x1 = X[i*minibatchSize:(i+1)*minibatchSize,:]
57             y1 = np.reshape(Y[i*minibatchSize:(i+1)*minibatchSize], [-1,1])
58             tf.reshape(y1, [-1,1])
59             _,lossval, outputval,errval = sess.run([train_op, loss, output, err], feed_dict={input_features: x1, input_labels: y1})
60             sumerr =sumerr+errval
61
62         print ("Epoch:", '%04d' % (epoch+1), "cost=", "{:.9f}".format(lossval),
63             "err=",sumerr/minibatchSize)
```

每一次的计算都会将err错误值累加起来，数

*****ebook converter DEMO Watermarks*****

据集迭代完一次会将err的错误率进行一次平均，平均值再输出来。运行上面的代码，生成如下信息：

```
Epoch: 0001 cost= 0.937670827 err= 0.857066742182
Epoch: 0002 cost= 0.576581895 err= 0.474182988405
Epoch: 0003 cost= 0.326138794 err= 0.273106197715
.....
Epoch: 0048 cost= 0.028127037 err= 0.019222549072
Epoch: 0049 cost= 0.027764326 err= 0.0191760268845
Epoch: 0050 cost= 0.027415426 err= 0.0191324938528
```

经过50次的迭代，得到了错误率为0.019的模型。

4. 数据可视化

为了直观地解释线性可分，下面将模型结果和样本以可视化的方式显示出来，前一部分是先取100个测试点，在图像上显示出来，接着将模型以一条直线的方式显示出来。

代码7-1 线性逻辑回归（续）

```
64      train_X, train_Y = generate(100, mean, cov, [3.0], True)
65      colors = ['r' if l == 0 else 'b' for l in train_Y[:]]
66      plt.scatter(train_X[:,0], train_X[:,1], c=colors)
67      x = np.linspace(-1,8,200)
68      y=-x*(sess.run(W)[0]/sess.run(W)[1])-sess.run(b)/sess.run(W)[0]
69      plt.plot(x,y, label='Fitted line')
70      plt.legend()
71      plt.show()
```

如上代码，模型生成的z用公式可以表示为 $z=x_1w_1+x_2*w_2+b$ ，如果将 x_1 和 x_2 映射到直角坐标系中的x和y坐标，那么z就可以被分为小于0和大于0两部分。当 $z=0$ 时，就代表直线本身，令上面的公式中z等于零，就可以将模型转化成如下直线方程：

$$x_2=-x_1 * w_1/w_2 - b/w_2, \text{ 即: } y=-x * (w_1/w_2) - (b/w_2)$$

其中， w_1 、 w_2 、 b 都是模型中的学习参数，带到公式中用plot显示出来。运行代码，生成结果如图7-2所示。

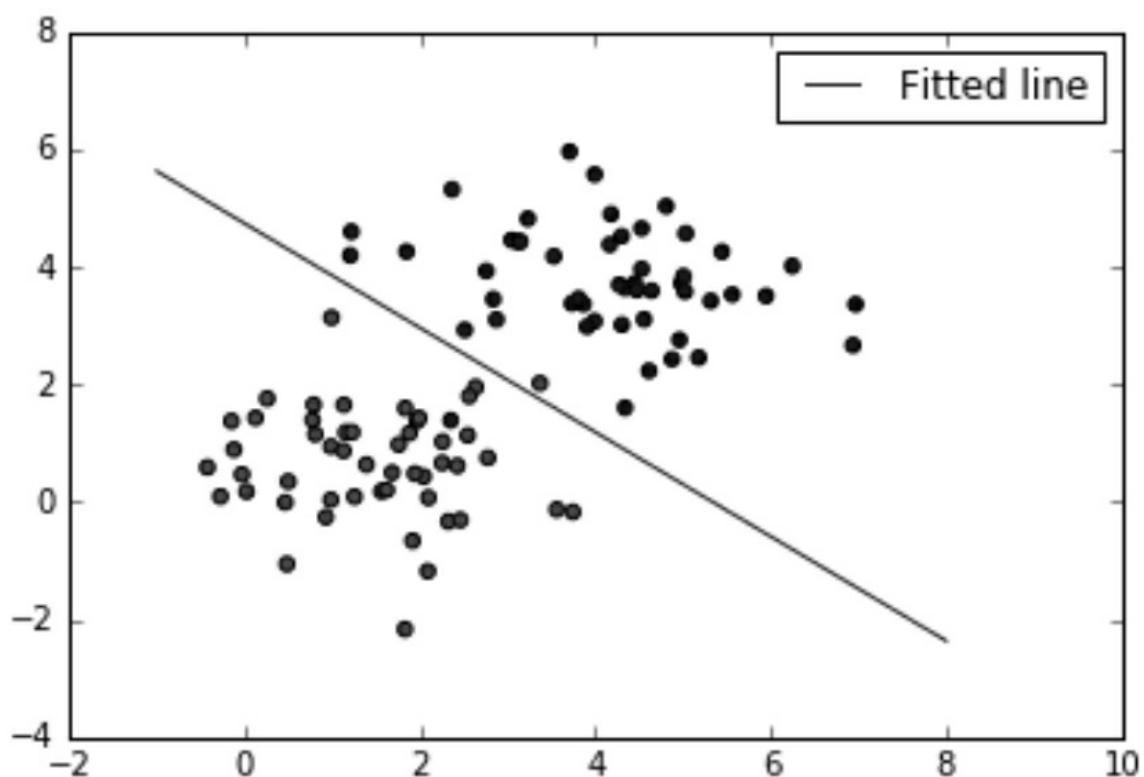


图7-2 线性逻辑回归

5. 线性可分概念

如图7-2所示这种情况，可以用直线分割的方式解决问题，则可以说这个问题是线性可分的。同理，类似这样的数据集就可以被称为线性可分数据集合。凡是使用这种方法来解决的问题就叫做线性问题。

7.1.2 实例29：用线性逻辑回归处理多分类问题

还是接着前面的例子，这次在数据集中再添加一类样本，可以使用多条直线将数据分成多类。

实例描述

构建网络模型完成将3类样本分开的任务。

在实现过程中先生成3类样本模拟数据，构造神经网络，通过softmax分类的方法计算神经网络的输出值，并将其分开。

1. 生成样本集

这里还使用上面代码中的generate函数，这次不同的是生成了2000个点、3类数据，并且使用one_hot编码。

代码7-2 线性多分类

```
01 np.random.seed(10)
02
03 input_dim = 2
04 num_classes =3
05 X, Y = generate(2000,num_classes, [[3.0],[3.0,0]],False)
06 aa = [np.argmax(l) for l in Y]
07 colors =['r' if l == 0 else 'b' if l==1 else 'y' for l in aa]
08 #将具体的点依照不同的颜色显示出来
09 plt.scatter(X[:,0], X[:,1], c=colors)
10 plt.xlabel("Scaled age (in yrs)")
11 plt.ylabel("Tumor size (in cm)")
12 plt.show()
```

进行上面的代码，生成的结果如图7-3所示。

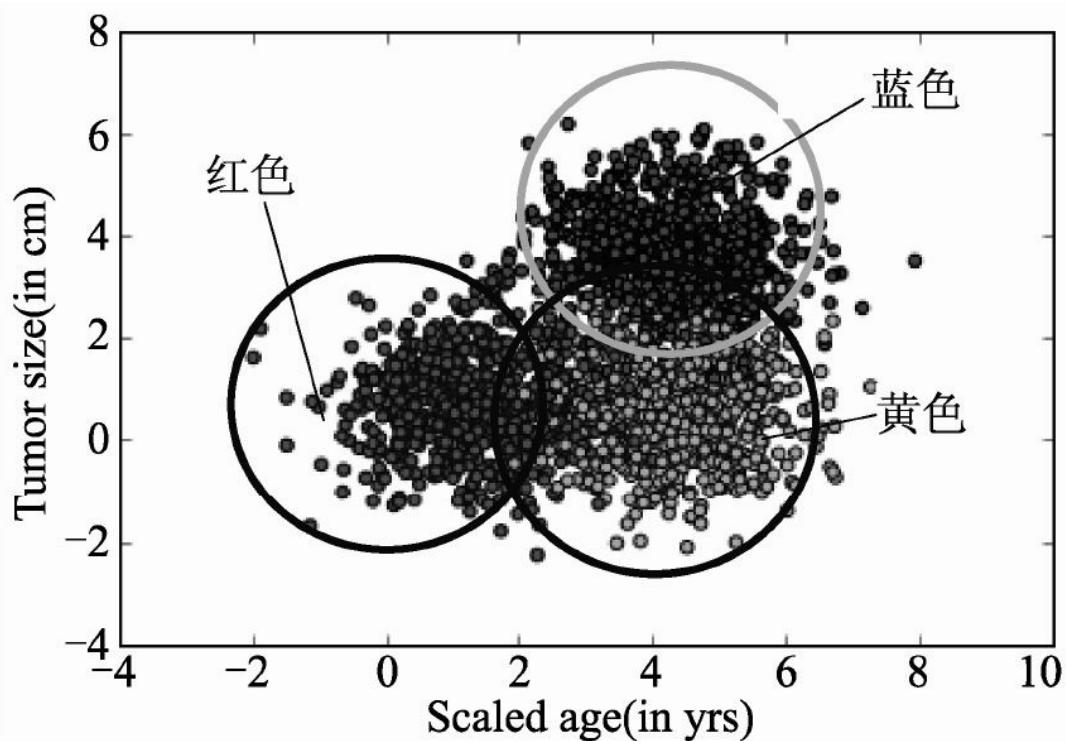


图7-3 三分类模拟数据集

在图7-3中，红色是原始的点，黄色点是在红

色点的基础上将 $x+3.0$ 后的变化，而蓝色的点是在红色点的基础上将 x 和 y 各加3.0。

2. 构建网络结构

下面开始构建网络模型，这次使用了softmax分类，损失函数loss仍然使用交叉熵，对于错误率评估部分换成了取one_hot结果里面不相同的个数，优化器使用AdamOptimizer。

代码7-2 线性多分类（续）

```
13 lab_dim = num_classes
14 # 定义占位符
15 input_features = tf.placeholder(tf.float32, [None, input_
16 input_labels = tf.placeholder(tf.float32, [None, lab_dim]
17 # 定义学习参数
18 W = tf.Variable(tf.random_normal([input_dim, lab_dim]), na
19 b = tf.Variable(tf.zeros([lab_dim]), name="bias")
20 output = tf.matmul(input_features, W) + b
21
22 z = tf.nn.softmax( output )
23
24 a1 = tf.argmax(tf.nn.softmax( output ), axis=1) #按行找出最
25 b1 = tf.argmax(input_labels, axis=1)
26 err = tf.count_nonzero(a1-b1) #两个数组相减，不为0的就是错误个
27
28 cross_entropy = tf.nn.softmax_cross_entropy_with_logits( la
input_labels, logits=output)
29 loss = tf.reduce_mean(cross_entropy) #对交叉熵取均值很有必要
30
31 optimizer = tf.train.AdamOptimizer(0.04) #尽量用Adam算法的1
32 train = optimizer.minimize(loss)
```

3. 设置参数进行训练

本次同样设置数据集迭代50次，每次的minibatchSize取25条。

代码7-2 线性多分类（续）

```
33 maxEpochs = 50
34 minibatchSize = 25
35
36 # 启动session
37 with tf.Session() as sess:
38     sess.run(tf.global_variables_initializer())
39
40     for epoch in range(maxEpochs):
41         sumerr=0
42         for i in range(np.int32(len(Y)/minibatchSize)):
43             x1 = X[i*minibatchSize:(i+1)*minibatchSize,:]
44             y1 = Y[i*minibatchSize:(i+1)*minibatchSize,:]
45
46             _,lossval, outputval,errval = sess.run([train_
47             err], feed_dict={input_features: x1, input_labels:
48             y1})
49             sumerr =sumerr+(errval/minibatchSize)
50
51         print ("Epoch:", '%04d' % (epoch+1), "cost=", "{:.9f}"
52             .format(lossval),"err=",sumerr/minibatchSize)
```

在迭代训练时对错误率的收集与前面的代码一致，每一次的计算都会将err错误值累加起来，数据集迭代完一次会将err的错误率进行一次平均，然后再输出平均值。运行上面的代码生成如下信息：

```
Epoch: 0001 cost= 0.408920079 err= 0.8544
Epoch: 0002 cost= 0.337683767 err= 0.3648
Epoch: 0003 cost= 0.321038276 err= 0.3328
Epoch: 0004 cost= 0.319500208 err= 0.32
.....
Epoch: 0048 cost= 0.422929078 err= 0.2784
```

```
Epoch: 0049 cost= 0.423131853 err= 0.2784
Epoch: 0050 cost= 0.423317522 err= 0.2784
```

4. 数据可视化

接下来一起看看对于三分类问题，线性可分是怎么分的。先取200个测试的点，在图像上显示出来，接着将模型中 x_1 、 x_2 的映射关系以一条直线的方式显示出来。因为输出端有3个节点，所以相当于3条直线。

代码7-2 线性多分类（续）

```
50     train_X, train_Y = generate(200, num_classes, [[3.0],
51         False)
52     aa = [np.argmax(l) for l in train_Y]
53     colors =['r' if l == 0 else 'b' if l==1 else 'y' for
54     plt.scatter(train_X[:,0], train_X[:,1], c=colors)
55     x = np.linspace(-1,8,200)
56
57     y=-x*(sess.run(W)[0][0]/sess.run(W)[1][0])-sess.run(t
      run(W)[1][0]
58     plt.plot(x,y, label='first line',lw=3)
59
60     y=-x*(sess.run(W)[0][1]/sess.run(W)[1][1])-sess.run(t
      run(W)[1][1]
61     plt.plot(x,y, label='second line',lw=2)
62
63     y=-x*(sess.run(W)[0][2]/sess.run(W)[1][2])-sess.run(t
      run(W)[1][2]
64     plt.plot(x,y, label='third line',lw=1)
65
66     plt.legend()
67     plt.show()
68     print(sess.run(W),sess.run(b))
```

运行上面的代码，输出如下，得到结果如图7-4所示。

```
[[ -1.29152238  1.68322766  1.79681265]
 [-0.55652267  2.47718096 -0.54918939]] [ 6.61509657 -8.4415
```

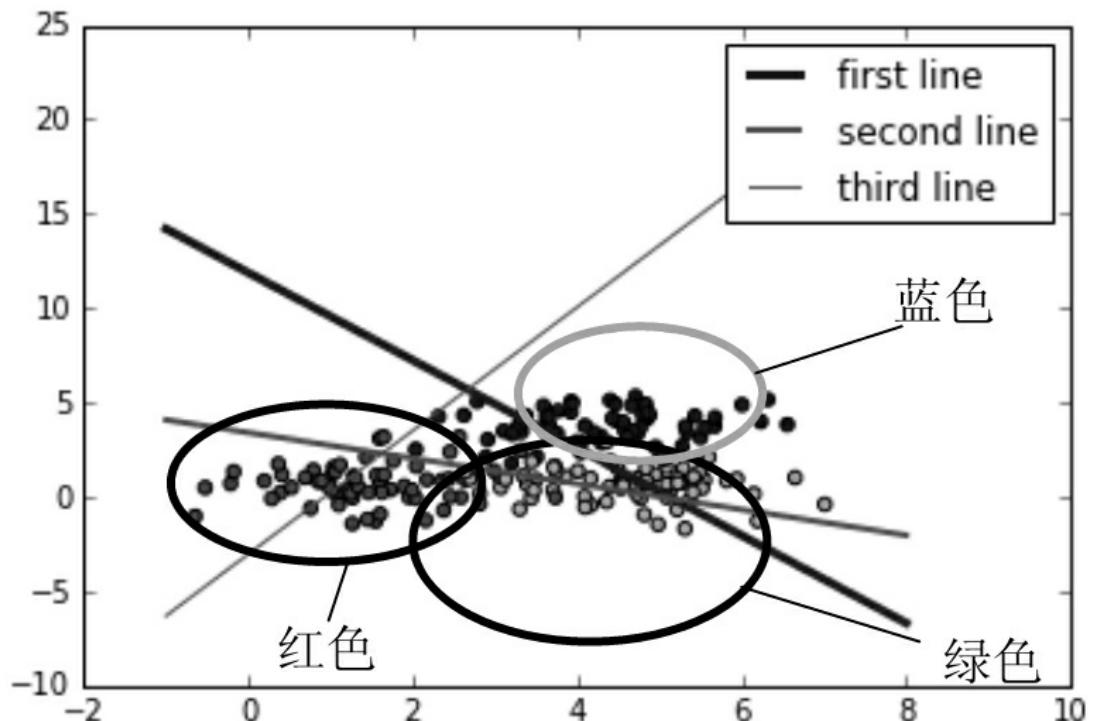


图7-4 三分类线性模型

图7-4中，3个权重分别代表了3条线。还原成模型就是模型里3个输出的分类节点：

- 第1个输出节点代表分类0，红色，蓝线(first line)。
- 第2个输出节点代表分类1，蓝色，绿线(second line)。

· 第3个输出节点代表分类2，黄色，红线（third line）。

这3条直线的斜率和截距是由神经网络的学习参数转化而来的。在神经网络里，一个样本通过这3个公式会得到3个结果，这3个结果可以理解成3个类的特征值。其中哪个值最大，则表示该样本具有哪种类别的特征最强烈，即属于哪一类。可以在横轴随便找一个值，分别带到3条直线的公式里，哪条直线得出的y值最大，则说明该点属于哪一类。这3条线也没有把集合点分开，这是因为它们的分类规则是不一样的。下面回顾一下直线公式：

$$y = -x^* \cdot (w_1/w_2) - (b/w_2)$$

正常来讲：如果一个点在直线上，等式成立；如果点在直线的上方，那么左边的y值就大；如果点在直线的下方，那么右边的算式值就大。

但放到模型里对应的图像上并不是这样的，还取决于 w_1 的正负取值，当 w_1 为负时正好是相反的情况。从上例中的输出结果里可以看到，只有第一条线（蓝线）的 w_1 是负数，所以蓝线是取其下面的点，红线和绿线是取其上方的点。举例：取一点红色的数据，如图7-5所示。

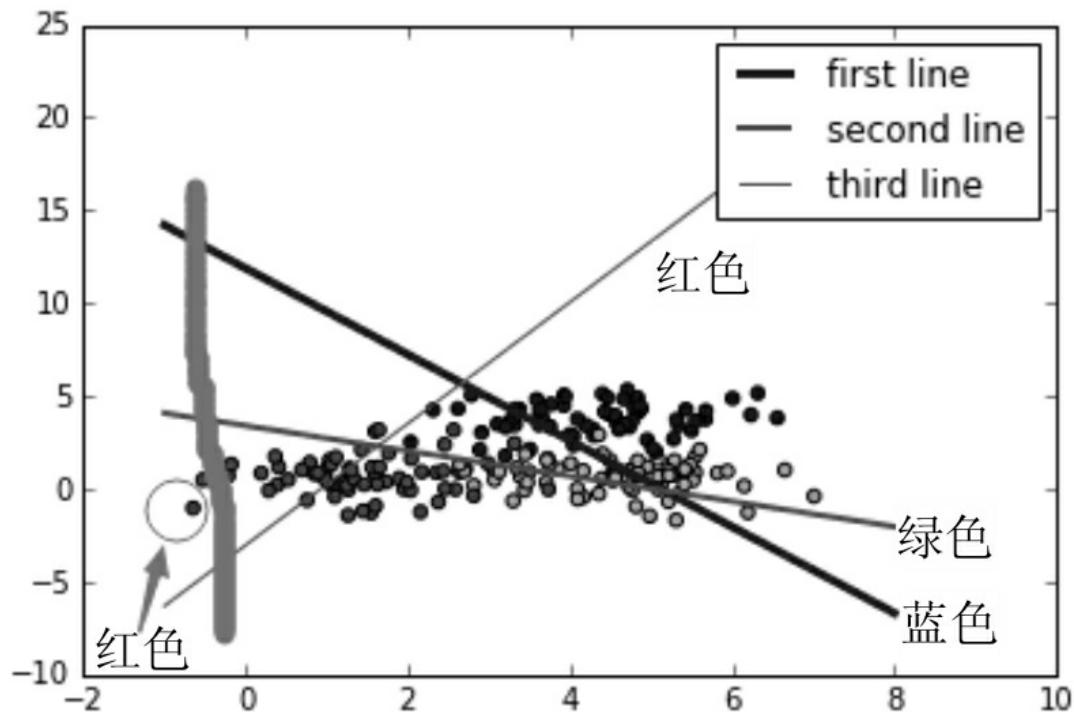


图7-5 三分类线性模型分析

沿着y轴的方向平行画一条线经过该点，可以看到它在第一条线（蓝线）的下方，并且离蓝线的距离是最远的，所以它就属于第一条线对应的红色分类。说明对于这类的数据集，仍然可以使用线性可分的方法将其分开。本例也展示了线性可分在多分类问题上的应用与原理。

5. 模型可视化

前面介绍了线性与模型的关系，现在把整个坐标系放到模型里，会得到一个更直观的模型分类可视化。

为了方便演示，还是在图像上生成200个点并显示出来。然后按照坐标系的排列，把x1, x2
*****ebook converter DEMO Watermarks*****

放到模型里，见如下代码。

代码7-2 线性多分类（续）

```
69     train_X, train_Y = generate(200, num_classes, [[3.0],  
70         False])  
70     aa = [np.argmax(l) for l in train_Y]  
71     colors =['r' if l == 0 else 'b' if l==1 else 'y' for  
72         plt.scatter(train_X[:,0], train_X[:,1], c=colors)  
73  
74     nb_of_xs = 200  
75     xs1 = np.linspace(-1, 8, num=nb_of_xs)  
76     xs2 = np.linspace(-1, 8, num=nb_of_xs)  
77     xx, yy = np.meshgrid(xs1, xs2) # 创建网格  
78     # 初始化和填充 classification plane  
79     classification_plane = np.zeros((nb_of_xs, nb_of_xs))  
80     for i in range(nb_of_xs):  
81         for j in range(nb_of_xs):  
82  
83             classification_plane[i,j] = sess.run(a1, feed_dict={  
84                 features: [[xx[i,j], yy[i,j]]]})  
85  
85     # 创建 color map 用于显示  
86     cmap = ListedColormap([  
87         colorConverter.to_rgba('r', alpha=0.30),  
88         colorConverter.to_rgba('b', alpha=0.30),  
89         colorConverter.to_rgba('y', alpha=0.30)])  
90     # 图示各个样本边界  
91     plt.contourf(xx, yy, classification_plane, cmap=cmap)  
92     plt.show()
```

上面代码的运行结果如图7-6所示。

图7-6中将三分类模型用了不同的颜色区域进行区分，这样就是符合人眼规律的一个比较直观的可视化图样了。

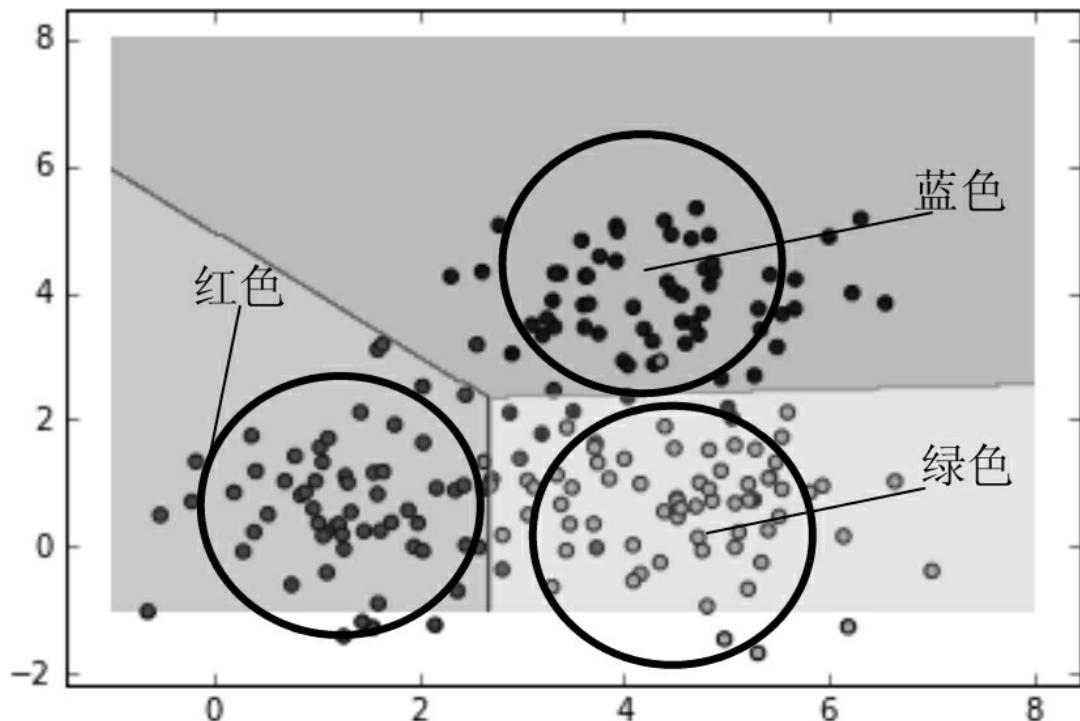


图7-6 三分类模型可视化

7.1.3 认识非线性问题

在明白了线性问题之后，接着介绍非线性问题。

非线性问题，就是用直线分不开的问题。为了解释这个概念，先来看一个数据集异或形态的数据，如图7-7所示。

图7-7中只有4个点，蓝色为一类，红色为一类，蓝色两个点的连线与红色两个点的连线会相交。对于这样的数据，你会发现无法使用一条直线将红色和蓝色两种类型的点分开，这就是非线性数据。

对于这样的数据，有一种笨方法，即对原始数据变形，使其变为线性分布。例如，将数据 x_1 、 x_2 进行一次绝对值运算，这时数据就会变为如图7-8所示的样子。

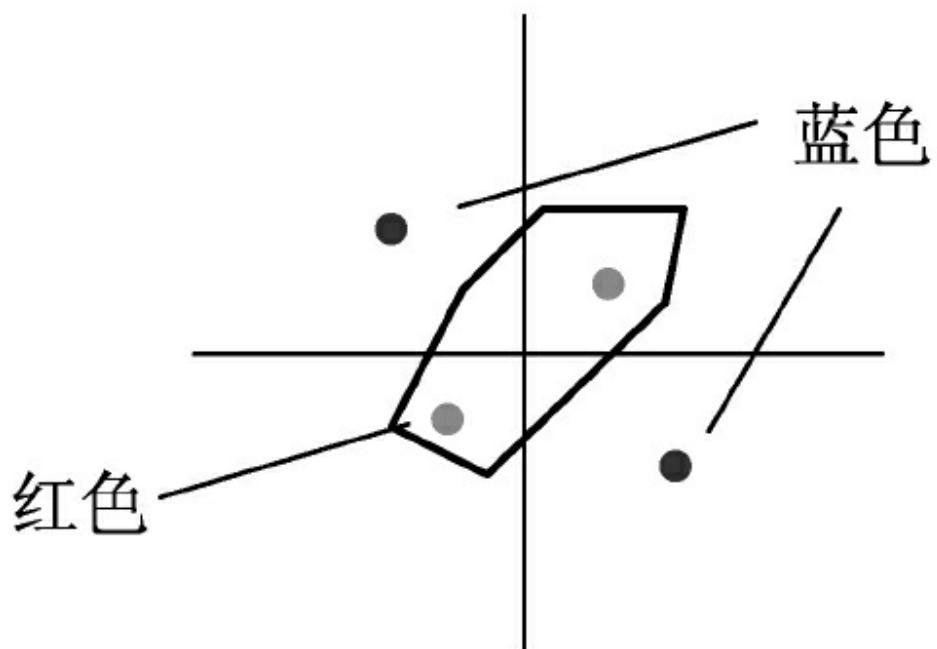


图7-7 异或形态的数据

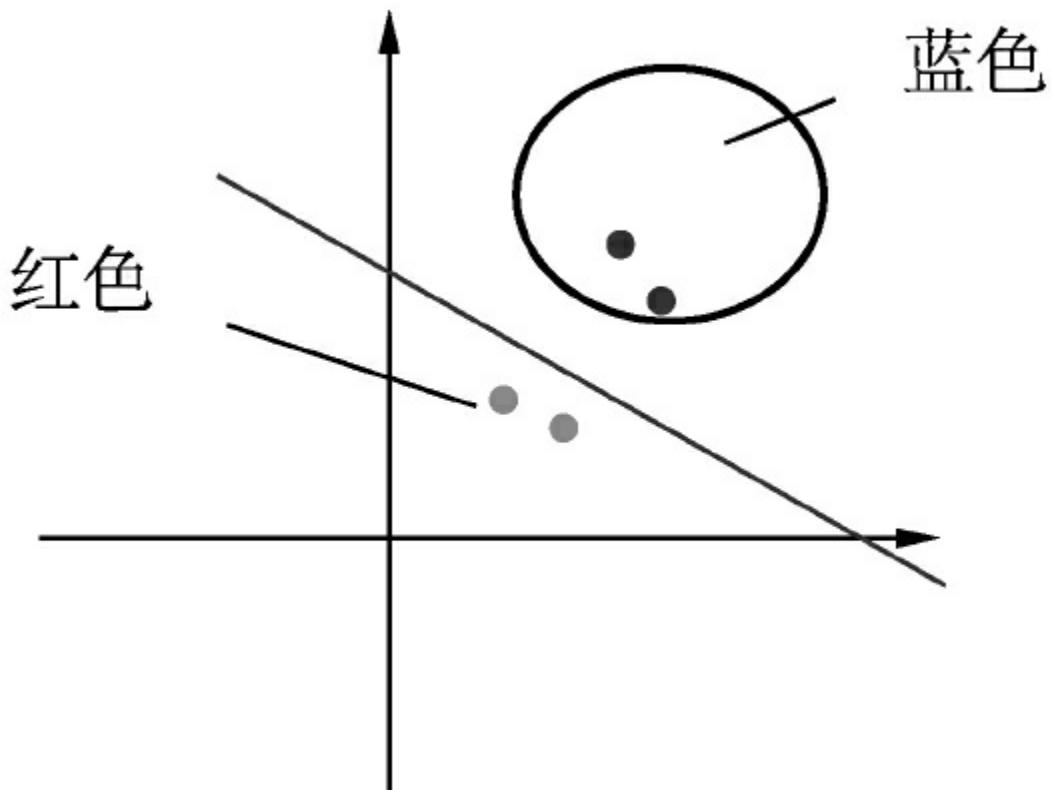


图7-8 改变输入数据

类似图7-8的方法有很多种，还可以将其进行一次平方运算。但这一切都是在人们肉眼看到模型分布后，通过分析得来的。而实际应用中会遇到更复杂的非线性数据集（如图7-9所示），或者有时数据维度太大，根本无法可视化。这时就需要用一种新方法——多层神经网络来解决问题了。

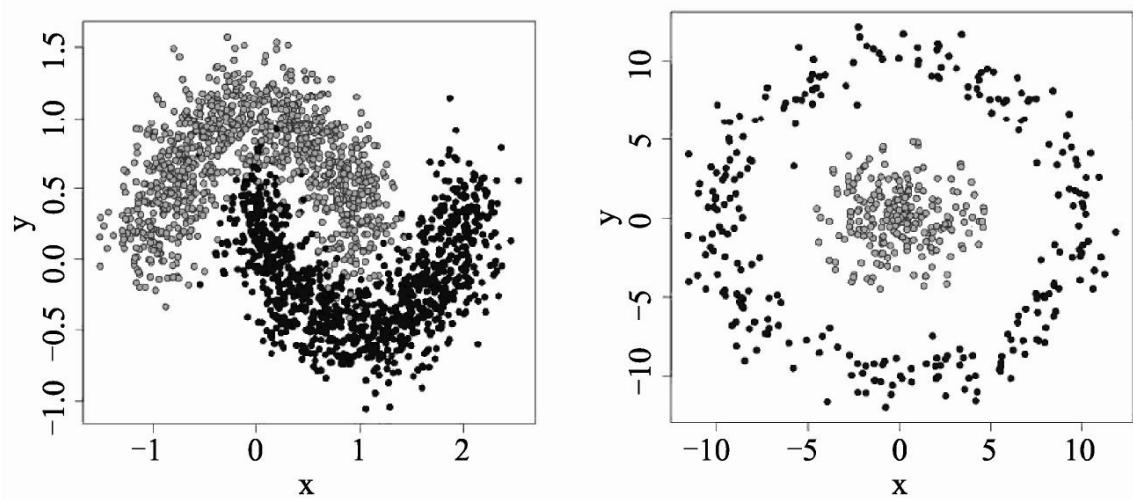


图7-9 非线性数据集

7.2 使用隐藏层解决非线性问题

多层神经网络非常好理解，就是在输入和输出中间多加些神经元，每一层可以加多个，也可以加很多层。下面通过一个例子将前面的异或数据进行分类。

7.2.1 实例30：使用带隐藏层的神经网络拟合异或操作

实例描述

通过构建符合异或规律的数据集作为模拟样本，构建一个简单的多层神经网络来拟合其样本特征完成分类任务。

1. 数据集介绍

所谓的“异或数据”是来源于异或操作，如图7-10所示。图7-10a为0、1操作，图7-10b为数据在直角坐标系上的展示。

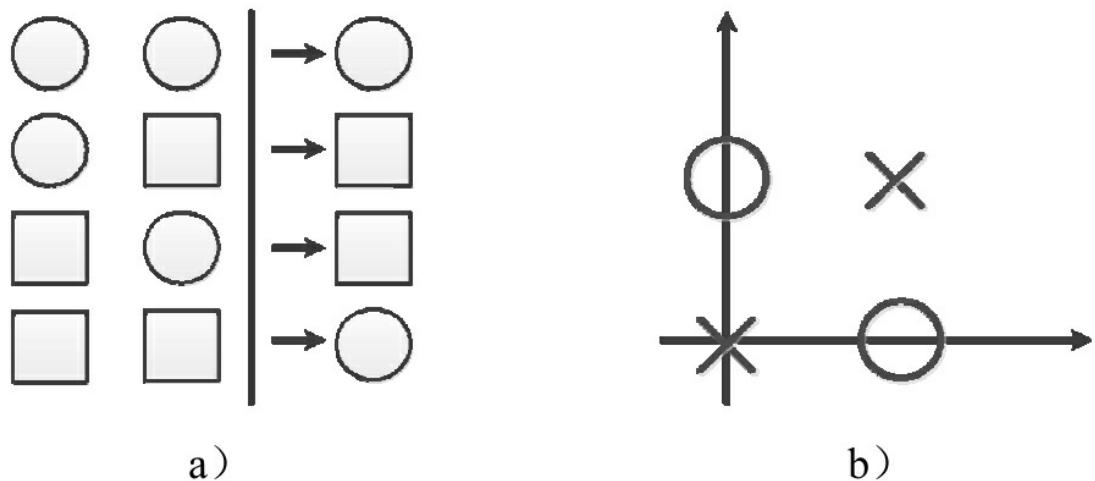


图7-10 异或数据介绍

从图7-10a中可以看出，当两个数相同时，输出为0，不相同时输出为1，这就是异或的规则。表示为两类数据就是 $(0, 0)$ 和 $(1, 1)$ 为一类， $(0, 1)$ 和 $(1, 0)$ 为一类。

2. 网络模型介绍

本例中使用了一个隐藏层来解决这个问题，如图7-11所示为要实现的网络结构。

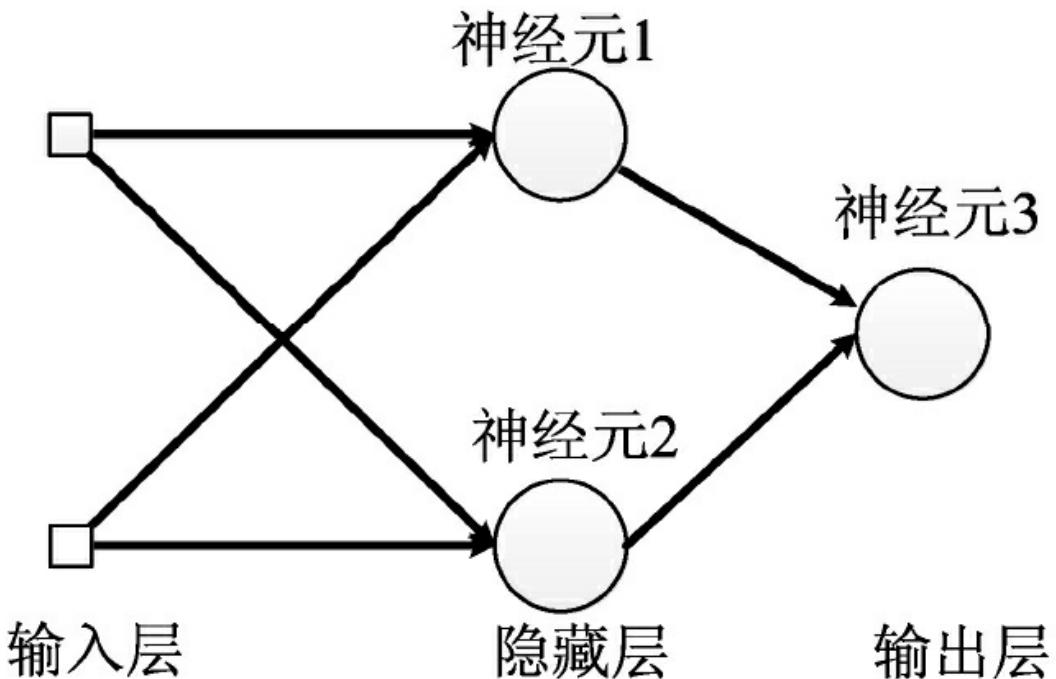


图7-11 隐藏层

3. 定义变量

下面开始编写代码。第一步定义变量，在网络参数的定义中，输入是“2”代表两个数，输出是“1”代表最终的结果，再放一个隐藏层，该隐藏层里有两个节点。输入占位符为x，输出为y，学习率为0.0001。

代码7-3 异或操作

```

01 import tensorflow as tf
02 import numpy as np
03
04 learning_rate = 1e-4
05 n_input  = 2                      #输入层节点个数
06 n_label  = 1
07 n_hidden = 2                      #隐藏层节点个数
08

```

```
09 x = tf.placeholder(tf.float32, [None, n_input])
10 y = tf.placeholder(tf.float32, [None, n_label])
```

4. 定义学习参数

这里以字典的方式定义权重w和b，里面的h1代表隐藏层，h2代表最终的输出层。

代码7-3 异或操作（续）

```
11 weights = {
12     'h1': tf.Variable(tf.truncated_normal([n_input, n_hidden],
13                             stddev=0.1)),
14     'h2': tf.Variable(tf.truncated_normal([n_hidden, n_label],
15                             stddev=0.1))
16 }
17 biases = {
18     'h1': tf.Variable(tf.zeros([n_hidden])),
19     'h2': tf.Variable(tf.zeros([n_label]))
20 }
```

5. 定义网络模型

该例中模型的正向结构入口为x，经过与第一层的w相乘再加上b，通过Relu函数进行激活转化，最终生成layer_1，再将layer_1代入第二层，使用Tanh激活函数生成最终的输出y_pred。

代码7-3 异或操作（续）

```
19 layer_1 = tf.nn.relu(tf.add(tf.matmul(x, weights['h1']),
20 ['h1']))
21 y_pred = tf.nn.tanh(tf.add(tf.matmul(layer_1, weights['h2']),
22 ['h2']))
```

*****ebook converter DEMO Watermarks*****

```
biases['h2']))  
21  
22 loss=tf.reduce_mean((y_pred-y)**2)  
23 train_step = tf.train.AdamOptimizer(learning_rate).minim:
```

模型的反向使用均值平方差（即对预测值与真实值的差取平均值）计算loss，最终使用AdamOptimizer进行优化。

6. 构建模拟数据

代码7-3 异或操作（续）

```
24 #生成数据  
25 X=[[0,0],[0,1],[1,0],[1,1]]  
26 Y=[[0],[1],[1],[0]]  
27 X=np.array(X).astype('float32')  
28 Y=np.array(Y).astype('int16')
```

手动建立X和Y数据集，形成对应的异或关系。

7. 运行session，生成结果

首先通过迭代10000次，将模型训练出来，然后将做好的X数据集放进去生成结果，接着再生成第一层的结果。

代码7-3 异或操作（续）

```
29 #加载session
```

*****ebook converter DEMO Watermarks*****

```
30 sess = tf.InteractiveSession()
31 sess.run(tf.global_variables_initializer())
32
33 #训练
34 for i in range(10000):
35     sess.run(train_step, feed_dict={x:X, y:Y} )
36
37 #计算预测值
38 print(sess.run(y_pred, feed_dict={x:X}))
39 #输出: 已训练100000次
40
41 #查看隐藏层的输出
42 print(sess.run(layer_1, feed_dict={x:X}))
```

运行上面的程序，得到如下结果：

```
[[ 0.10773809]
 [ 0.60417336]
 [ 0.76470393]
 [ 0.26959091]]
[[ 0.00000000e+00   2.32602470e-05]
 [ 7.25074887e-01   0.00000000e+00]
 [ 0.00000000e+00   9.64471161e-01]
 [ 2.06250161e-01   1.69421546e-05]]
```

第一个是4行1列的数组，用四舍五入法来取值，与我们定义的输出Y完全吻合。第二个为4行2列的数组，为隐藏层的输出。

7.2.2 非线性网络的可视化及其意义

接上例中的第二个输出是4行2列数组，其中第一列为隐藏层第一个节点的输出，第二列为隐藏层第二个节点的输出，将它们四舍五入取整显示如下：

$$\begin{bmatrix} [0 & 0] \\ [1 & 0] \\ [0 & 1] \\ [0 & 0] \end{bmatrix}$$

可以很明显地看出，最后一层其实是对隐藏层的AND运算，因为最终结果为[0, 1, 1, 0]。也可以理解成第一层将数据转化为线性可分的数据集，然后在输出层使用一个神经元将其分开。

1. 隐藏层神经网络相当于线性可分的高维扩展

在几何空间里，两个点可以定位一条直线，两条直线可以定位一个平面，两个平面可以定位一个三维空间，两个三维空间可以定位更高维的空间……

在线性可分问题上也可以这样扩展，线性可分是在一个平面里，通过一条线来分类，那么同理，如果线所在的平面升级到了三维空间，则需要通过一个平面将问题分类。如图7-12所示，把异或数据集的输入 x_1 、 x_2 当成平面的两个点，输出 y 当作三维空间里的 z 轴上的坐标，那么所绘制的图形就是这样的（见图7-12）。

很明显，这样的数据集是很好分开的。图7-12中，右面的比例尺指示的是纵坐标。0刻度往下，颜色由浅蓝逐渐变为深蓝；0刻度往上，颜色

由浅红逐渐变为深红。作一个平行于底平面、高度为0的平面，即可将数据分开，如图7-12中的虚平面。我们前面使用的隐藏层的两个节点，可以理解成定位中间平面的两条直线。其实，一个隐藏层的作用，就是将线性可分问题转化成平面可分问题。更多的隐藏层，就相当于转化成更高维度的空间可分问题。所以理论上通过升级空间可分的结构，是可以将任何问题分开的。

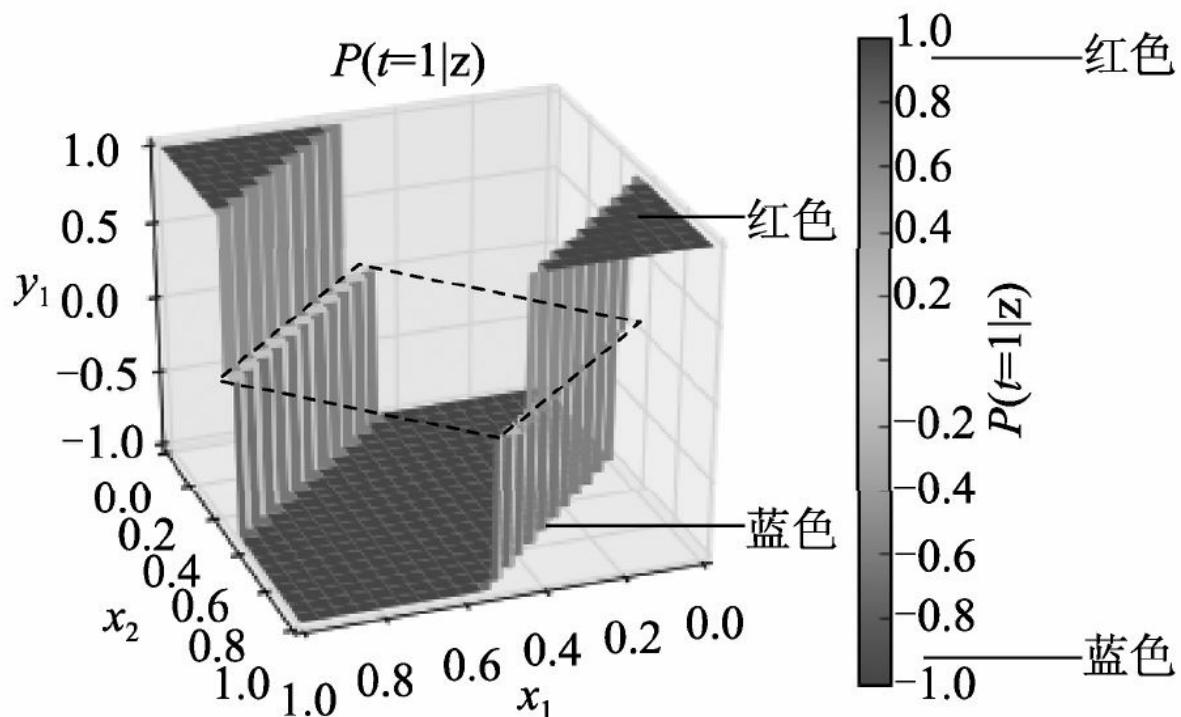


图7-12 异或集高维展示

2. 从逻辑门的角度来理解多层网络

对于多层网络，还可以通过逻辑门的角度来理解，如图7-13所示。将数据集映射到直角坐标系中，通过可视化图形可以看到，在直角坐标系

中有两条直线将其分开，对于两条直线的结果，可以再通过神经网络构建一个逻辑运算，即可将它们融合在一起并产生最终想要的结果。

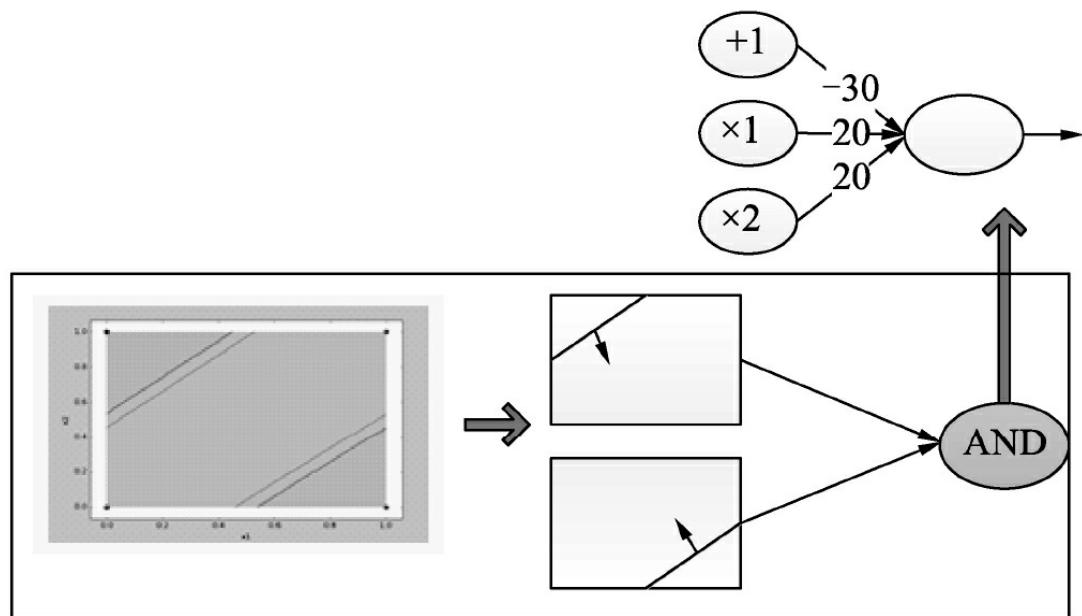


图7-13 隐藏层的意义

图7-13中，对两条直线的结果取AND运算，即可实现异或的效果，而构建AND逻辑的权重很容易实现，图中使用 $w[-30, 20]$, $b[20]$ 即实现了AND的逻辑。

类似这样的逻辑门还有很多，如图7-14中举例了神经元实现的AND、OR、NOT逻辑，最终通过这些逻辑门的运算，甚至不需要训练就可以搭建出一个异或的模型（如图7-14所示部分）。

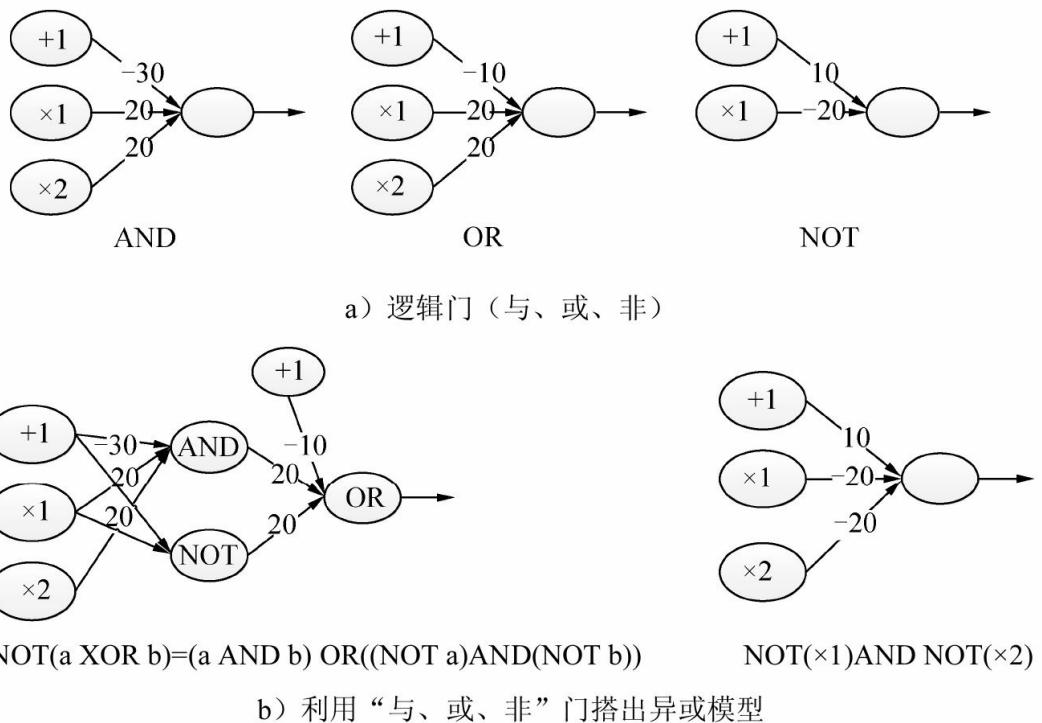


图7-14 逻辑门

看到这里，了解计算机原理的读者都知道，CPU的基础运算都是在构建逻辑门基础之上完成的，例如，用逻辑门组成最基本的加减乘除四则运算，再用四则运算组成更复杂的功能操作，最终可以实现操作系统并在其上进行各种操作。

神经网络的结构和功能，使其具有编程和实现各种高级功能的能力，只不过这个编程不需要人脑通过学习算法来拟合现实，而是使用模型学习的方式，直接从现实的表象中优化成需要的结构。

所以说，这种多层的结构只要层数足够多，每层的节点足够多，参数合理，就可以拟合世界上的任何问题，而放在神经网络里考验的则是，

模型的自学习功能是否足够高效和精准。

7.2.3 练习题

(1) 试着修改7.2.1节中的例子，调整最后一层的激活函数为Relu或是Sigmoid，看看会有什么结果（Sigmoid可以，但是Relu陷入了局部最优解，如果迭代次数增到20000，全0，即梯度丢失。于是可以使用Leaky relus，发现在10000、20000、30000时都会进入局部最优解，但再也不会出现梯度消失，将迭代次数变为40000时，得到了正确的模型）。

(2) 试着将7.2.1节中的例子的数据集修改成one_hot编码来进行拟合，利用所学的知识看看可以用几种方法来实现（可参考本书源代码中的代码“7-4 异或one_hot.py”文件）。

7.3 实例31：利用全连接网络将图片进行分类

本例使用全连接网络，将第5章中的例子重新实现一遍，将MNIST图像用多层神经网络来分类。

实例描述

构建一个简单的多层神经网络，以拟合MNIST样本特征完成分类任务。

1. 定义网络参数

在输入和输出之间使用两个隐藏层，每层各256个节点，学习率使用0.001。

代码7-5 MNIST多层分类

```
01 # 定义参数
02 learning_rate = 0.001
03 training_epochs = 25
04 batch_size = 100
05 display_step = 1
06
07 # 设置网络模型参数
08 n_hidden_1 = 256          # 第一个隐藏层节点个数
09 n_hidden_2 = 256          # 第二个隐藏层节点个数
10 n_input = 784             # MNIST 共784 (28×28)维
11 n_classes = 10            # MNIST 共10个类别 (0~9)
```

2. 定义网络结构

multilayer_perceptron函数为封装好的网络模型函数，第一层与第二层均使用Relu激活函数，loss使用softmax交叉熵。具体代码如下。

代码7-5 MNIST多层分类（续）

```
12 #定义占位符
13 x = tf.placeholder("float", [None, n_input])
14 y = tf.placeholder("float", [None, n_classes])
15
16 # 创建model
17 def multilayer_perceptron(x, weights, biases):
18     # 第一层隐藏层
19     layer_1 = tf.add(tf.matmul(x, weights['h1']), biases['b1'])
20     layer_1 = tf.nn.relu(layer_1)
21     # 第二层隐藏层
22     layer_2 = tf.add(tf.matmul(layer_1, weights['h2']), biases['b2'])
23     layer_2 = tf.nn.relu(layer_2)
24     # 输出层
25     out_layer = tf.matmul(layer_2, weights['out']) + biases['out']
26     return out_layer
27
28 # 学习参数
29 weights = {
30     'h1': tf.Variable(tf.random_normal([n_input, n_hidden_1])),
31     'h2': tf.Variable(tf.random_normal([n_hidden_1, n_hidden_2])),
32     'out': tf.Variable(tf.random_normal([n_hidden_2, n_classes]))
33 }
34 biases = {
35     'b1': tf.Variable(tf.random_normal([n_hidden_1])),
36     'b2': tf.Variable(tf.random_normal([n_hidden_2])),
37     'out': tf.Variable(tf.random_normal([n_classes])))
38 }
39
40 # 输出值
41 pred = multilayer_perceptron(x, weights, biases)
42
43 # 定义loss和优化器
44 cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(
    logits=pred, labels=y))
```

```
45 optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
minimize(cost)
```

3. 运行session输出结果

session运行的代码参见第5章实例，运行结果如下：

```
Epoch: 0001 cost= 166.257328408
Epoch: 0002 cost= 39.961055286
.....
Epoch: 0023 cost= 0.335092138
Epoch: 0024 cost= 0.289653350
Epoch: 0025 cost= 0.286943634
Finished!
Accuracy: 0.957
```

全连接网络可以成功地将图片进行分类，并且随着层数的增加和节点的增多，还能够得到更好的拟合效果。



注意：由于神经网络的学习算法限制，在实际情况中并不是层数越多、节点越多，效果就越好，因为在训练过程中使用的BP算法，会随着层数的逐渐增大其算出来的调整值会逐渐变小，直到其他层都感觉不到变化，即梯度消失的情况。

7.4 全连接网络训练中的优化技巧

随着科研人员在使用神经网络训练时不断的尝试，为我们留下了好多有用的技巧，合理地运用这些技巧可以使自己的模型得到更好的拟合效果。本节就来介绍下全连接网络在训练过程中的一些常用技巧。

7.4.1 实例32：利用异或数据集演示过拟合问题

全连接网络虽然在拟合问题上比较强大，但太强大的拟合效果也带来了其他的麻烦，这就是过拟合问题。什么是过拟合呢？

首先来看一个例子，这次将原有的4个异或数据扩充成上百个具有异或特征的数据集，通过全连接网络将它们进行分类。具体步骤如下：

实例描述

构建异或数据集模拟样本，再构建一个简单的多层神经网络来拟合其样本特征，观察其出现欠拟合的现象，接着通过增大网络复杂性的方式来优化欠拟合问题，使其出现过拟合现象。

1. 构建异或数据集

参照代码“7-2线性多分类.py”文件中的生成模拟数据代码，调用generate函数生成4类数据，然后将其中的两类数据合并。

代码7-6 异或集的过拟合

```
01 np.random.seed(10)
02
03 input_dim = 2
04 num_classes =4
05 X, Y = generate(320,num_classes, [[3.0,0],[3.0,3.0],[0,
06 Y=Y%2
07
08 xr=[]
09 xb=[]
10 for(l,k) in zip(Y[:,],X[:,]):
11     if l == 0.0 :
12         xr.append([k[0],k[1]])
13     else:
14         xb.append([k[0],k[1]])
15 xr =np.array(xr)
16 xb =np.array(xb)
17 plt.scatter(xr[:,0], xr[:,1], c='r',marker='+')
18 plt.scatter(xb[:,0], xb[:,1], c='b',marker='o')
19 plt.show()
```

运行上面的代码，可以看到数据集的结构如图7-15所示。

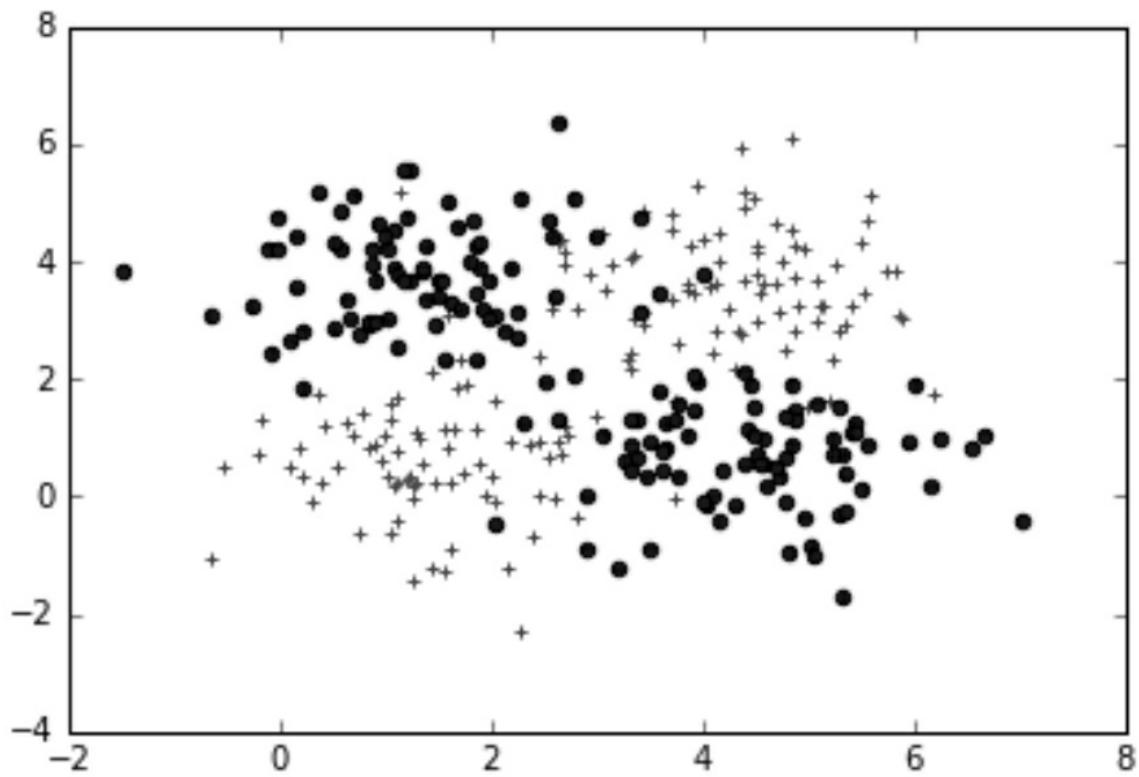


图7-15 异或数据集输出

可以看到，图7-15中的数据分为两类，其中左下和右上是一类（红色用+表示），左上和右下是另一类（蓝色用·表示）。

2. 修改定义网络模型

这里还沿用代码“7-3异或.py”文件里的代码，不用改动，只需要把原来的异或数据集注释掉即可（代码略）。

3. 添加可视化

这里生成120个点并放到模型里，然后将其在直角坐标系中显示出来。

代码7-6 异或集的过拟合（续）

```
20 xTrain, yTrain = generate(120, num_classes, [[3.0, 0], [3.0, 1], [0.0, 1], [0.0, 0]], True)
21 yTrain=yTrain%2
22 xr=[]
23 xb=[]
24 for(l,k) in zip(yTrain[:,],xTrain[:,]):
25     if l == 0.0 :
26         xr.append([k[0],k[1]])
27     else:
28         xb.append([k[0],k[1]])
29 xr =np.array(xr)
30 xb =np.array(xb)
31 plt.scatter(xr[:,0], xr[:,1], c='r',marker='+')
32 plt.scatter(xb[:,0], xb[:,1], c='b',marker='o')
33 yTrain=np.reshape(yTrain, [-1,1])
34 print ("loss:\n", sess.run(loss, feed_dict={x: xTrain, y: yTrain}))
35
36 nb_of_xs = 200
37 xs1 = np.linspace(-1, 8, num=nb_of_xs)
38 xs2 = np.linspace(-1, 8, num=nb_of_xs)
39 xx, yy = np.meshgrid(xs1, xs2) # 创建grid
40 # 初始化和填充 classification plane
41 classification_plane = np.zeros((nb_of_xs, nb_of_xs))
42 for i in range(nb_of_xs):
43     for j in range(nb_of_xs):
44         classification_plane[i,j] = sess.run(y_pred, feed_dict={x: [[xx[i,j], yy[i,j]]]})[0]
45         classification_plane[i,j] = int(classification_plane[i,j])
46
47 # 创建一个color map用来显示每一个格子的分类颜色
48 cmap = ListedColormap([
49     colorConverter.to_rgba('r', alpha=0.30),
50     colorConverter.to_rgba('b', alpha=0.30)])
51 # 图示样本的分类边界
52 plt.contourf(xx, yy, classification_plane, cmap=cmap)
53 plt.show()
```

运行上面的代码，得到如下信息：

```
Step: 0 Current loss: 0.50001
```

*****ebook converter DEMO Watermarks*****

```
Step: 1000 Current loss: 0.359438
.....
Step: 10000 Current loss: 0.204833
Step: 11000 Current loss: 0.204797
Step: 12000 Current loss: 0.204775
Step: 13000 Current loss: 0.204766
Step: 14000 Current loss: 0.204765
Step: 15000 Current loss: 0.204765
Step: 16000 Current loss: 0.204765
Step: 17000 Current loss: 0.204765
Step: 18000 Current loss: 0.204765
Step: 19000 Current loss: 0.204765
loss:
0.204765
```

可视化后生成的数据集结构如图7-16所示。

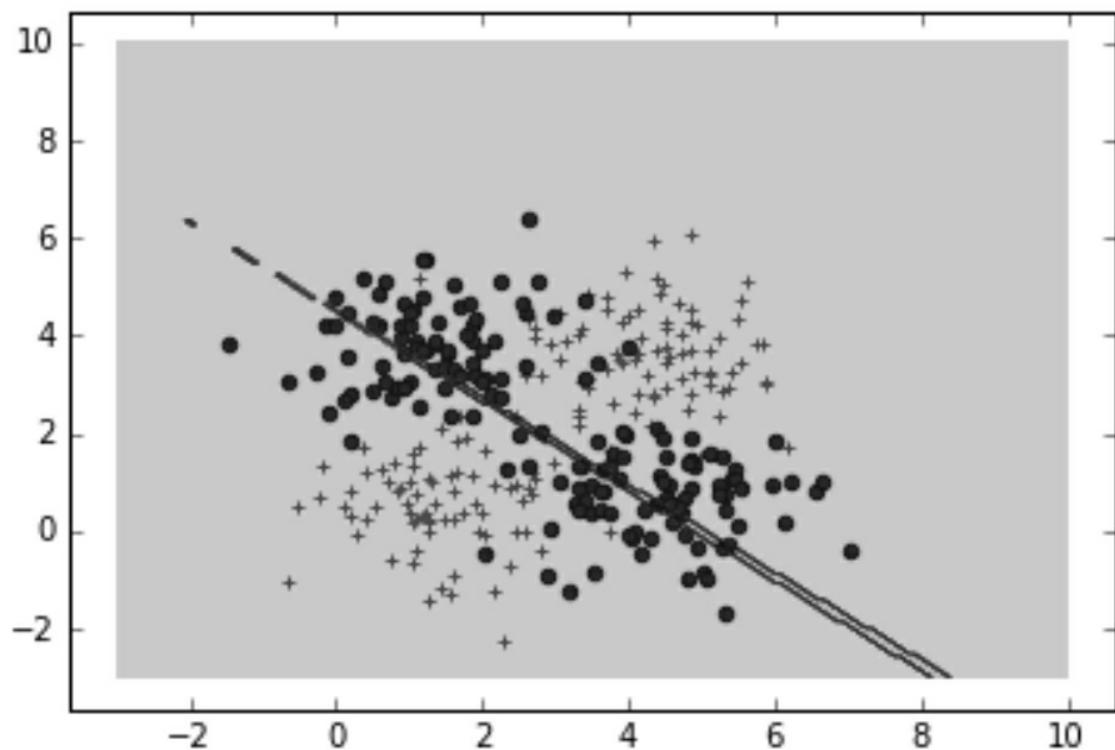


图7-16 失败模型

可以看到，模型在迭代训练10000次之后停止了梯度，而且loss值约为20%，准确率不高，所

可视化的图片也没有将数据完全分开。

4. 欠拟合定义

如图7-16所示的这种效果就叫做欠拟合，即没有完全拟合到想要得到的真实数据情况。

5. 修正模型提高拟合度

欠拟合的原因并不是模型不行，而是我们的学习方法无法更精准地学习到适合的模型参数。模型越薄弱，对训练的要求就越高。但是可以采用增加节点或增加层的方式，让模型具有更高的拟合性，从而降低模型的训练难度。

将隐藏层的节点提高到200，代码如下：

```
n_hidden = 200
```

运行代码后显示如下结果：

```
Step: 0 Current loss: 0.510105
Step: 1000 Current loss: 0.0951028
.....
Step: 15000 Current loss: 0.0477655
Step: 16000 Current loss: 0.0463676
Step: 17000 Current loss: 0.0451465
Step: 18000 Current loss: 0.043569
Step: 19000 Current loss: 0.0421998
```

可视化后生成的数据集结构如图7-17所示。

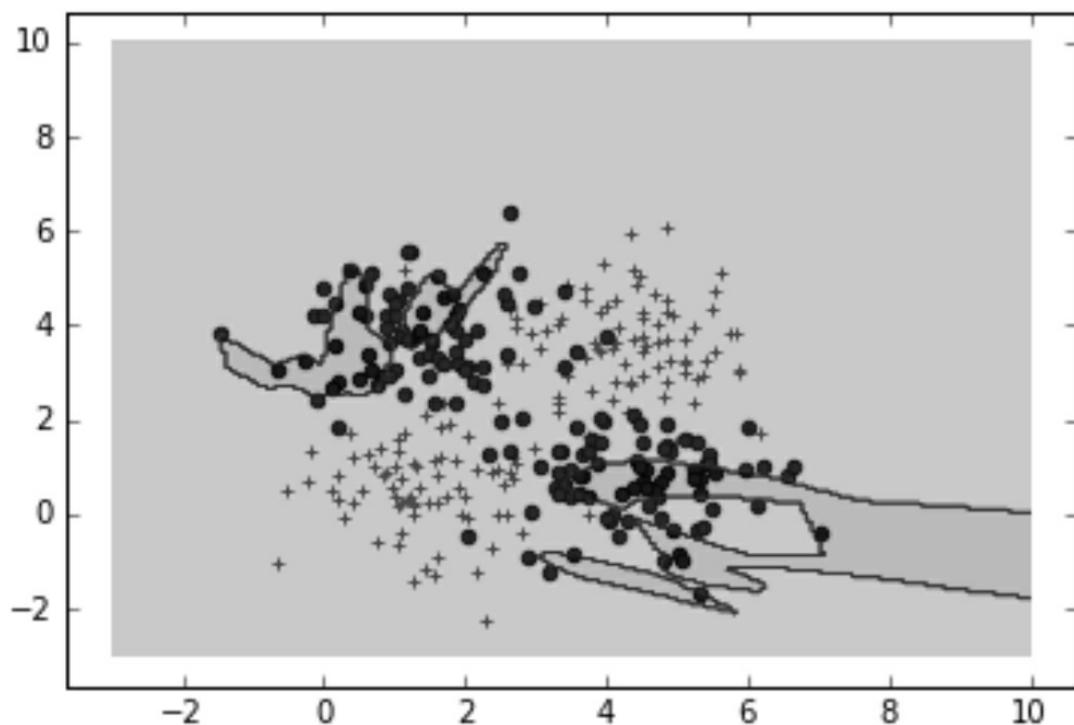


图 7-17 过拟合

从7-17中可以看到强大的全连接网络，仅仅通过一个隐藏层，使用200个点就可以将数据划分得这么细致。而loss值也在逐渐变小，20000次之后变为0.04。

6. 验证过拟合

那么对于上面的模型好不好呢？我们再取少量的数据（12个）放到模型里验证一下，然后用同样的方式在坐标系中可视化（可视化代码部分同上）。

代码7-6 异或集的过拟合（续）

*****ebook converter DEMO Watermarks*****

```

54 xTrain, yTrain = generate(12, num_classes, [[3.0, 0], [3.0, 1]], True)
55 yTrain=yTrain%2
56 xr=[]
57 xb=[]
58 for(l,k) in zip(yTrain[:,],xTrain[:,]):
59     if l == 0.0 :
60         xr.append([k[0],k[1]])
61     else:
62         xb.append([k[0],k[1]])
63 xr =np.array(xr)
64 xb =np.array(xb)
65 plt.scatter(xr[:,0], xr[:,1], c='r',marker='+')
66 plt.scatter(xb[:,0], xb[:,1], c='b',marker='o')
67 yTrain=np.reshape(yTrain,[-1,1])
68 print ("loss:\n", sess.run(loss, feed_dict={x: xTrain, y:yTrain}))
69 #可视化部分
70 .....

```

运行上面的代码，生成如下信息：

```

loss:
0.149396

```

可视化后生成的数据集结构如图7-18所示。

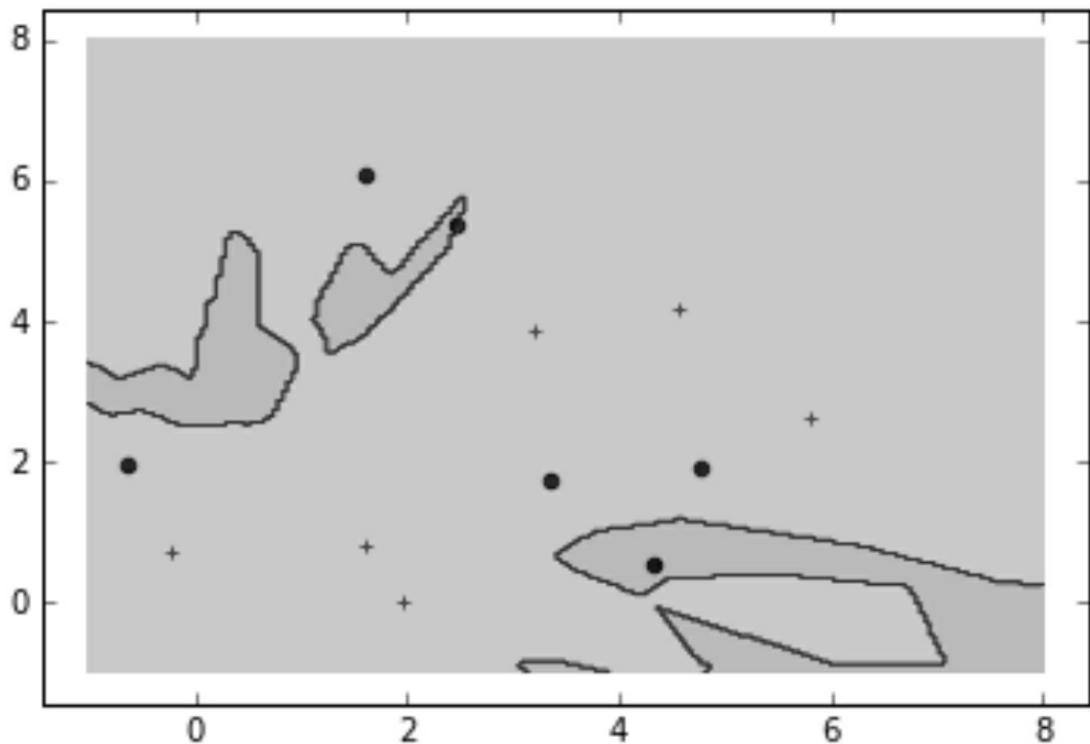


图7-18 过拟合验证

由图7-18可以看出，loss飙到了14%，并没有原来训练时那么好（4%），模型还是原来的模型，但是这次却框住了少量的样本。这种现象就是过拟合。它与欠拟合一样都是我们在训练模型中不愿意看到的现象，我们要的是真正的拟合在测试情况下能够表现出训练时的良好效果。

避免过拟合的方法有很多：常用的方法有early stopping、数据集扩增、正则化、dropout等。

- early stopping: 在发生过拟合之前提前结束训练。理论上是可以的，但是这个点不好把握。

- 数据集扩增 (data augmentation) : 就是让模型见到更多的情况，可以最大化地满足全样本，但实际应用中对于未来事件的预测却显得鞭长莫及。

- 正则化 (regularization) : 是通过引入范数的概念，增强模型的泛化能力，包括L1、L2 (L2 regularization也叫weight decay) 。

- dropout: 是网络模型中的一种方法，每次训练时舍去一些节点来增强泛化能力。

下面重点介绍一下后两种方法。

7.4.2 正则化

本节将开始学习正则化技巧。

1. 什么是正则化

所谓的正则化，其实就是在神经网络计算损失值的过程中，在损失后面再加一项。这样损失值所代表的输出与标准结果间的误差就会受到干扰，导致学习参数 w 和 b 无法按照目标方向来调整，实现模型无法与样本完全拟合的结果，从而达到防止过拟合的效果。

理解了原理之后，现在就来介绍如何添加这

个干扰项。干扰项一定要有这样的特性：

- 当欠拟合时，希望它对模型误差的影响越小越好，以便让模型快速拟合实际。
- 如果是过拟合时，希望它对模型误差的影响越大越好，以便让模型不要产生过拟合的情况。

由此引入了两个范数L1和L2：

- L1：所有学习参数w的绝对值的和。
- L2：所有学习参数w的平方和然后求平方根。

如果放到损失函数的公式里，会将其变形一下，如式（7-1）和式（7-2）所示，其中式（7-1）为L1，式（7-2）为L2。

$$less = less(0) + \lambda \sum_w^n |W| \quad \text{式 (7-1)}$$

$$less = less(0) + \frac{\lambda}{2} \sum_w^n W^2 \quad \text{式 (7-2)}$$

最终的loss为等式左边的结果， $less(0)$ 代表真实的loss值， $less(0)$ 后面的那一项就代表正则化了， λ 为一个可以调节的参数，用来控制正

则化对loss的影响。

对于L2，将其乘以1/2是为了反向传播时对其求导正好可以将数据规整。

2. TensorFlow中的正则化

对于上面的公式，读者了解一下就可以了。因为TensorFlow中已经有封装好的函数可以拿来直接使用。

L2的正则化函数为：

```
tf.nn.l2_loss(t, name=None)
```

L1的正则化函数目前在TensorFlow中没有现成的，可以自己组合为：

```
tf.reduce_sum(tf.abs(w))
```

7.4.3 实例33：通过正则化改善过拟合情况

了解完过拟合的解决方法后，现在就来给前面的代码“7-6异或集的过拟合.py”文件添加正则化的处理。代码非常简单，只需要在计算损失值时加上loss的正则化，例子中，使用的 λ 为0.01，添加的是L2_loss，代码如下。

实例描述

构建异或数据集模拟样本，使用多层神经网络将其分类，并使用正则化技术来改善过拟合情况。

代码7-7 异或集的L2_loss

```
01 .....
02 reg = 0.01           #l2_loss参数
03 loss=tf.reduce_mean((y_pred-y)**2)+tf.nn.l2_loss(weights|
*reg+tf.nn.l2_loss(weights['h2']))*reg
04 .....
```

其他的地方都不用动，运行代码，结果如下：

```
Step: 0 Current loss: 0.520193
.....
Step: 16000 Current loss: 0.0913737
Step: 17000 Current loss: 0.0913519
Step: 18000 Current loss: 0.0913312
Step: 19000 Current loss: 0.0913115

loss:
0.10637
```

可以看出，虽然训练的loss值增加了一些，变成了0.09，但是模型的测试loss却由0.15降到了0.1，比以前进步了不少。可视化后生成的模型如图7-19所示。

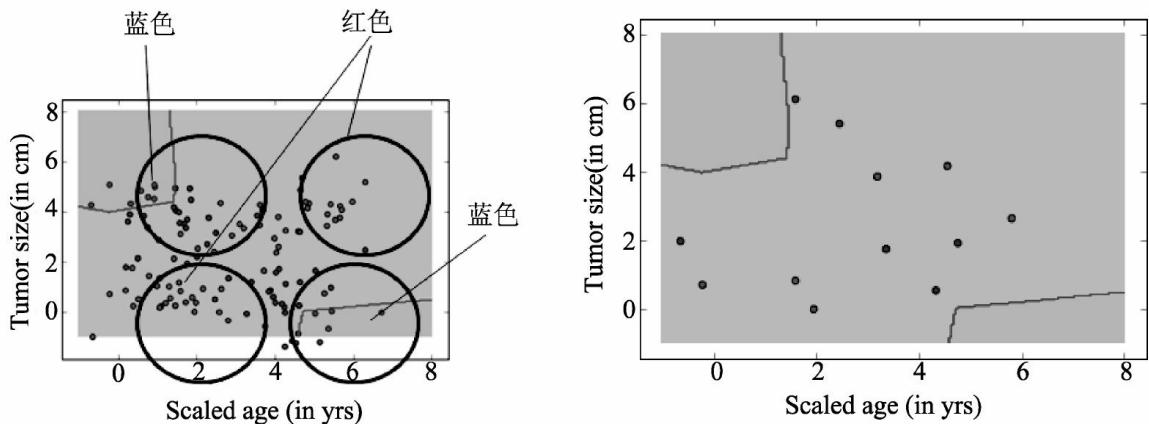


图7-19 正则化模型

图7-19中，左边为模型在训练时的结果，右边为测试时的结果。图中的蓝色区域比起前面的例子不再是单独封闭的区间了。

7.4.4 实例34：通过增大数据集改善过拟合

下面再试试通过增大数据集的方式来改善过拟合情况，这里不再生成一次样本，而是每次循环都生成1000个数据，来看看会发生什么。修改代码如下。

实例描述

构建异或数据集模拟样本，使用多层神经网络将其分类，并使用增大数据集的方法来改善过拟合情况。

在循环训练中，在for循环里的`sess.run`之前添加生成数据的代码，每次取1000个点。

代码7-7 异或集的L2_loss (续)

```
05 for i in range(20000):#生成异或数据集
06
07     X, Y = generate(1000, num_classes, [[3.0, 0], [3.0, 3.0],
08     True])
08     Y=Y%2
09     Y=np.reshape(Y, [-1,1])
10
11     _, loss_val = sess.run([train_step, loss], feed_dict=
```

其他地方都不用动，运行代码，生成如下信息：

```
Step: 0 Current loss: 0.399712
Step: 1000 Current loss: 0.141141
.....
Step: 17000 Current loss: 0.0992013
Step: 18000 Current loss: 0.0991972
Step: 19000 Current loss: 0.0991939
loss:
0.09075
```

这次得到的模型测试值直接降到了0.9，比训练时还低。所生成的模型可视化如图7-20所示。

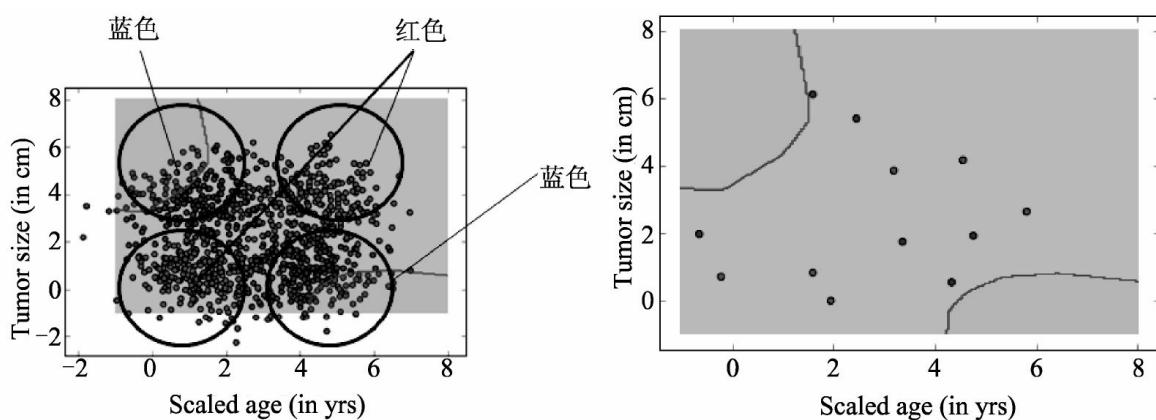


图7-20 增大数据集

如图7-20所示增加数据集之后，发现蓝色区域比之前变得更大了，泛化效果也有了明显的提示。

7.4.5 练习题

试着使用L1_loss来改善过拟合现象，看看效果。

7.4.6 dropout——训练过程中，将部分神经单元暂时丢弃

还有一种常用的手段叫做dropout，也是用来防止过拟合的。

1. dropout原理

还有一种常用的改善过拟合的方法dropout。dropout的意思是，在训练过程中，每次随机选择一部分节点不要去“学习”。

这样做的原理是什么呢？

因为从样本数据的分析来看，数据本身是不可能很纯净的，即任何一个模型不能100%把数据完全分开，在某一类中一定会有一些异常数据，

过拟合的问题恰恰是把这些异常数据当成规律来学习了。对于模型来讲，我们希望它能够有一定的“智商”，把异常数据过滤掉，只关心有用的规律数据。

异常数据的特点是，它与主流样本中的规律都不同，但是量非常少，相当于在一个样本中出现的概率比主流数据出现的概率低很多。我们就是利用这个特性，通过在每次模型中忽略一些节点的数据学习，将小概率的异常数据获得学习的机会降低，这样这些异常数据对模型的影响就会更小了。



注意：由于dropout让一部分节点不去“学习”，所以在增加模型的泛化能力的同时，会使学习速度降低，使模型不太容易“学成”，所以在使用的过程中需要合理地调节到底丢弃多少节点，并不是丢弃的节点越多越好。

2. TensorFlow中的dropout

在TensorFlow中dropout的函数原型如下：

```
def dropout(x, keep_prob, noise_shape=None, seed=None, name=
```

其中的参数意义如下。

- X：输入的模型节点。
- keep_prob：保持率。如果为1，则代表全部进行学习；如果为0.8，则代表丢弃20%的节点，只让80%的节点参与学习。
- noise_shape：代表指定x中，哪些维度可以使用dropout技术。为None时，表示所有维度都使用dropout技术。也可以将某个维度标志为1，来代表该维度使用dropout技术。例如：x的形状为[n, len, w, ch]，使用noise_shape为[n, 1, 1, ch]，这表明会对x中的第二维度len和第三维度w进行dropout。
- seed：随机选取节点的过程中随机数的种子值。



注意： dropout改变了神经网络的网络结构，它仅仅是属于训练时的方法，所以一般在进行测试时要将dropout的keep_prob变为1，代表不需要进行丢弃，否则会影响模型的正常输出。

7.4.7 实例35：为异或数据集模型添加dropout

本例在代码“7-7 异或集的L2_loss.py”文件的基础上进行修改，为了体现效果，把原来的l2_loss去掉（实际过程中可以两个方法一起使

用)。

实例描述

构建异或数据集模拟样本，使用多层神经网络将其分类，并使用dropout技术来改善过拟合情况。

如下代码，在layer_1后面添加一个dropout层，将dropout的keep_prob设为占位符，这样可以在运行时随时指定keep_prob，在session的run中指定keep_prob为0.6，这意味着每次训练将仅允许0.6的节点参与学习运算。由于学习速度慢了，所以要将学习率调大些，变成0.01，加快训练。

另外，在测试时别忘了一定要将keep_prob调成1。

代码7-8 异或集dropout

```
01 .....
02 learning_rate = 0.01#1e-4
03 .....
04 layer_1 = tf.nn.relu(tf.add(tf.matmul(x, weights['h1']),
05 ['h1']))
06 keep_prob = tf.placeholder("float")
07 layer_1_drop = tf.nn.dropout(layer_1, keep_prob)
08
09 #Leaky relus激活函数
10 layer2 = tf.add(tf.matmul(layer_1_drop, weights['h2']), bias)
11 y_pred = tf.maximum(layer2, 0.01*layer2)
12 .....
13 for i in range(20000):
```

```
14
15     X, Y = generate(1000, num_classes, [[3.0,0],[3.0,3.0]
16     True)
16     Y=Y%2
17     Y=np.reshape(Y, [-1,1])
18
19     _, loss_val = sess.run([train_step, loss], feed_dict=
20     keep_prob:0.6})
20
21     if i % 1000 == 0:
22         print ("Step:", i, "Current loss:", loss_val)
23 .....
24 yTrain=np.reshape(yTrain,[-1,1])
25 print ("loss:\n", sess.run(loss, feed_dict={x: xTrain, y:
26 prob:1.0}))
```

运行代码，输出如下：

```
Step: 0 Current loss: 0.503951
Step: 1000 Current loss: 0.0896698
Step: 2000 Current loss: 0.0923921
Step: 3000 Current loss: 0.0912758
Step: 4000 Current loss: 0.0885499
Step: 5000 Current loss: 0.0899685
Step: 6000 Current loss: 0.0923872
Step: 7000 Current loss: 0.0922362
Step: 8000 Current loss: 0.0920109
Step: 9000 Current loss: 0.0918544
Step: 10000 Current loss: 0.0894592
Step: 11000 Current loss: 0.0899565
Step: 12000 Current loss: 0.0939654
Step: 13000 Current loss: 0.0950037
Step: 14000 Current loss: 0.0922148
Step: 15000 Current loss: 0.0934821
Step: 16000 Current loss: 0.093902
Step: 17000 Current loss: 0.0913219
Step: 18000 Current loss: 0.0939114
Step: 19000 Current loss: 0.0912721
loss:
0.0604928
```

测试效果很不错！这次的模型测试loss比训练的loss值还要低，而且达到了0.06。这就是dropout的效果。

7.4.8 实例36：基于退化学习率dropout技术来拟合异或数据集

从上面的结果可以看到，损失值在10000时是0.08，后来又涨到了0.09，尤其在最后几次，出现了抖动的现象，这表明后期的学习率有点大了。读者还记得前面学过的退化学习率吗？下面我们就在上面的例子中添加退化学习率，让开始的学习率很大，后面逐渐变小。

实例描述

构建异或数据集模拟样本，使用多层神经网络将其分类，并使用dropout配合退化学习率的技术来改善过拟合情况。

在使用优化器的代码部分添加decaylearning_rate，设置总步数为20000，每执行1000步，学习率衰减0.9，见如下代码。

代码7-8 异或集dropout（续）

```
27 global_step = tf.Variable(0, trainable=False)
28 decaylearning_rate = tf.train.exponential_decay(learning_
global_step, 1000, 0.9)
```

```
29 #train_step = tf.train.AdamOptimizer(learning_rate).minimize(loss)
30 train_step = tf.train.AdamOptimizer(decaylearning_rate).minimize(loss, global_step=global_step)
```

运行上面代码，输出如下：

```
Step: 0 Current loss: 0.42503
Step: 1000 Current loss: 0.0930188
Step: 2000 Current loss: 0.0894333
Step: 3000 Current loss: 0.0918793
Step: 4000 Current loss: 0.0913094
Step: 5000 Current loss: 0.0913863
Step: 6000 Current loss: 0.0875175
Step: 7000 Current loss: 0.0903373
Step: 8000 Current loss: 0.0899588
Step: 9000 Current loss: 0.0899196
Step: 10000 Current loss: 0.0901761
Step: 11000 Current loss: 0.0887947
Step: 12000 Current loss: 0.0891289
Step: 13000 Current loss: 0.0883277
Step: 14000 Current loss: 0.0908775
Step: 15000 Current loss: 0.0866709
Step: 16000 Current loss: 0.0907037
Step: 17000 Current loss: 0.0897186
Step: 18000 Current loss: 0.0889717
Step: 19000 Current loss: 0.0901095
loss: 0.0568894
```

可以看到，整个loss的趋势是在减小的，而且loss值变成了0.56，比原来更低了，虽然也有些波动，但那是因为dropout随机时受到了异常数据运算结果的影响。

来看一下最终这次模型的可视化效果，如图7-21所示，红色用+表示，蓝色用·表示。

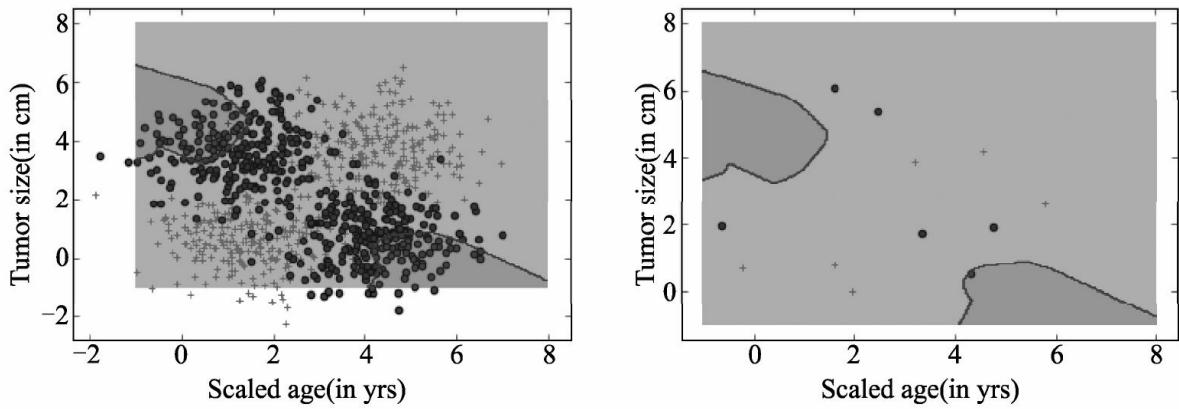


图7-21 dropout+退化学习率

7.4.9 全连接网络的深浅关系

全连接神经网络是一个通用的近似框架。只要有足够多的神经元，即使只有一层隐藏层的神经网络，利用常用的Sigmoid, reLU等激活函数，就可以无限逼近任何连续函数。

在实际中，如果想使用浅层神经网络来拟合复杂非线性函数，就需要靠增加的神经元个数来实现。神经元过多意味着需要训练的参数过多，这会增加网络的学习难度，并影响网络的泛化能力。因此，在搭建网络结构时，一般倾向于使用更深的模型，来减少网络中所需要神经元的数量，使网络有更好的泛化能力。

7.5 练习题

在本书的源代码里有3个关于异或问题的代码文件“7-9 xorerr1.py、7-10 xorerr2.py、7-11 xorerr3.py”，分别存在着不同的错误，试着修正它们，生成正确的模型。

第8章 卷积神经网络——解决参数太多问题

卷积神经网络是深度学习中最经典的模型之一。当今所有的深度学习经典模型中都能找到卷积神经网络的身影。它巧妙地利用很少的权重却达了全连接网络实现不了的效果。本章将进入卷积神经网络的学习。先看看与全连接网络相比，它能够解决哪些问题。

本章含有教学视频18分04秒。

作者按照本章的内容结构，对主要内容体系进行了概括性的讲解，包括卷积神经网络的作用，卷积操作和反卷积操作的实现及作用，以及关于卷积神经网络在训练中的一些优化技巧等（重点是对卷积及池化中的输入输出对应规则，以及卷积网络的优化技巧）。

深度学习之TensorFlow

入门、原理与进阶实战

第8章 卷积神经网络 ——解决参数太多问题

配套视频



代码医生

qq群: 40016981

<http://blog.csdn.net/lijin6249>



字幕



*****ebook converter DEMO Watermarks*****

8.1 全连接网络的局限性

在第7章的代码“7-5 mnist多层分类.py”的实例中，仅使用了一个 28×28 像素的小图片数据集就完成了分类任务。但在实际应用中要处理的图片像素一般都是1024，甚至更大。这么大的图片输入到全连接网络中后会有什么效果呢？我们可以分析一下。

如果只有两个隐藏层，每层各用了256个节点，则MNIST数据集所需要的参数是 $(28 \times 28 \times 256 + 256 \times 256 + 256 \times 10)$ 个w，再加上 $(256 + 256 + 10)$ 个b。

1. 图像变大导致色彩数变多，不好解决

如果换为1000像素呢？仅一层就需要 $1000 \times 1000 \times 256 \approx 2$ 亿个w（可以把b都忽略）。这只是灰度图，如果是RGB的真彩色图呢？再乘上3后则约等于6亿。如果想要得到更好的效果，再加几个隐藏层……可以想象，需要的学习参数量将是非常多的，不仅消耗大量的内存，同时也需要大量的运算，这显然不是我们想要的结果。

2. 不便处理高维数据

对于比较复杂的高维数据，如按照全连接的

方法，则只能通过增加节点、增加层数的方式来解决。而增加节点会引起参数过多的问题。因为由于隐藏层神经网络使用的是Sigmoid或Tanh激活函数，其反向传播的有效层数也只能在4~6层左右。所以，层数再多只会使反向传播的修正值越来越小，网络无法训练。

而卷积神经网络使用了参数共享的方式，换了一个角度来解决问题，不仅在准确率上大大提升，也把参数降了下来。下面就来学习一下卷积神经网络。

8.2 理解卷积神经网络

卷积神经网络避免了对参数的过度依赖，相比全连接神经网络，能更好地识别高维数据（即超大图片）。它是什么样的一个东西呢？先来理解一下sobel算子吧。如图8-1这就是sobel算子对图片处理后的效果，它可以把图片的轮廓显示出来。



a)



b)

图8-1 sobel算子示例

不要被它的名字吓到，它其实是个很简单的矩阵计算，其方法见图8-2所示的卷积过程。

图8-2a的 5×5 矩阵可以理解为图8-1a（即原始图片），经过卷积操作后，变为图8-1b（即轮廓图）。

整个过程如图8-2所示，具体步骤如下。

(1) 在外面补了一圈0，这个过程叫做padding，目的是为了变换后生成同样大小的矩阵。

(2) 将图8-2a左上角的 3×3 矩阵中的每个元素分别与中间的 3×3 矩阵对应位置上的元素相乘，然后再相加，这样得到的值作为图8-2b的第一个元素。

(3) 中间的 3×3 矩阵就是sobel算子。

(4) 把图8-2a中左上角的 3×3 矩阵向右移动一个格，这可以理解为步长为1。

(5) 将图8-2a矩阵中的每个元素分别与中间的 3×3 矩阵对应位置上的元素相乘然后进行加和运算，算出的值填到图8-2b的第二个元素里。

(6) 一直重复上述操作，直到将图8-2b中的值都填满，整个这个过程就叫做卷积。

sobel矩阵可以理解为卷积神经网络里的卷积核（也可以叫“滤波器”，filter），它里面的值也可以理解为权重w。在sobel中，这些w是固定的，就相当于一个训练好的模型，只要通过里面的值变换后的图片，就会产生具有轮廓的效果。这个变换后的图片，在卷积神经网络里称为

feature map。

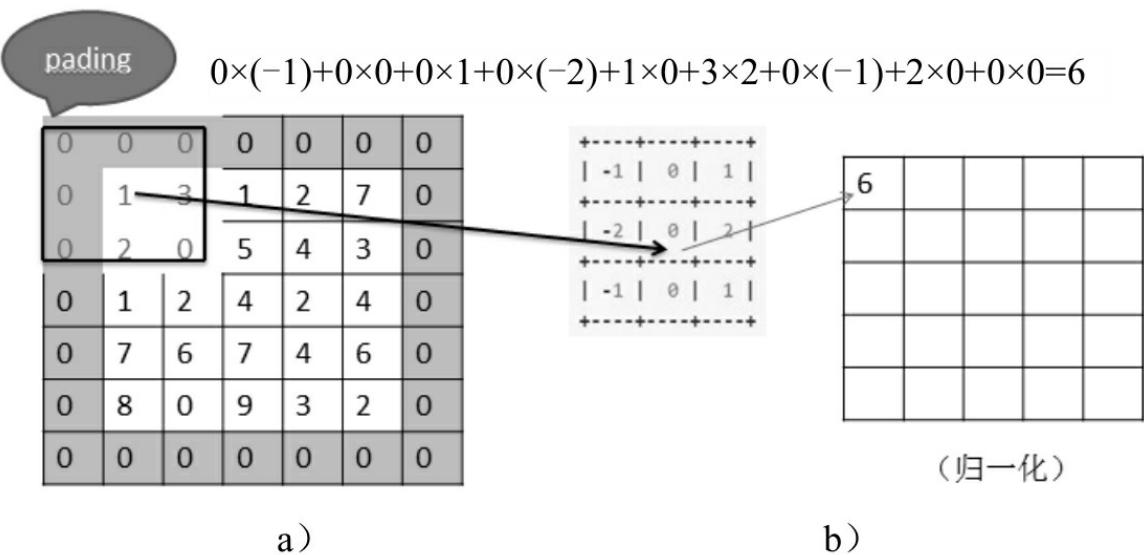


图8-2 卷积过程

注意：新生成的图片里面的每个像素值并不能保证在0~256之间。对于在区间外的像素点会导致灰度图无法显示，所以还需要做一次归一化，然后每个值都乘以256，再将所有的值映射到这个区间内。

归一化算法： $x = (x - \text{Min}) / (\text{Max} - \text{Min})$ 。

其中，Max与Min为整体数据里的最大值和最小值， x 是当前要转换的像素值。归一化之后可以保证每个 x 都在[0, 1]的区间内。

8.3 网络结构

卷积神经网络的结构与全连接网络相比复杂得多。它的网络结构主要包括卷积层、池化层。细节又可以分为滤波器、步长、卷积操作、池化操作等。

8.3.1 网络结构描述

前面讲述的是一个基本原理，实际的卷积操作会复杂一些，对于一幅图片一般会使用多个卷积核（滤波器），将它们统一放到卷积层里来操作，这一层中有几个滤波器，就会得出几个 feature map，接着还要经历一个池化层（pooling），将生成的 feature map 缩小（降维），池化层会在下面的文章中介绍。图8-3所示为神经网络中一个标准的卷积操作组合。

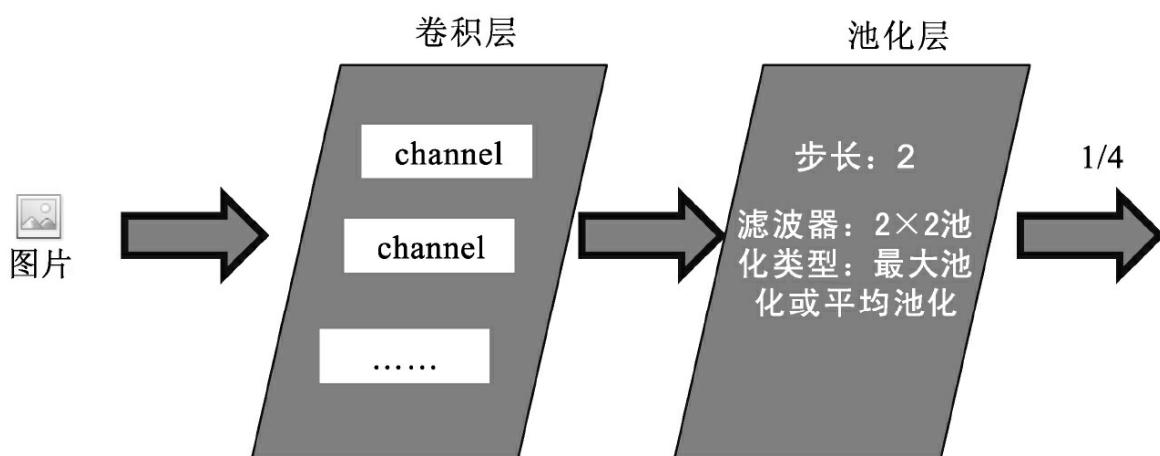


图8-3 卷积结构

图8-3中卷积层里面channel的个数代表卷积层的深度。池化层中则只有一个滤波器（fileter），主要参数是尺寸大小（即步长大小）。

下面先来看一个卷积网络的完整结构，如图8-4所示。

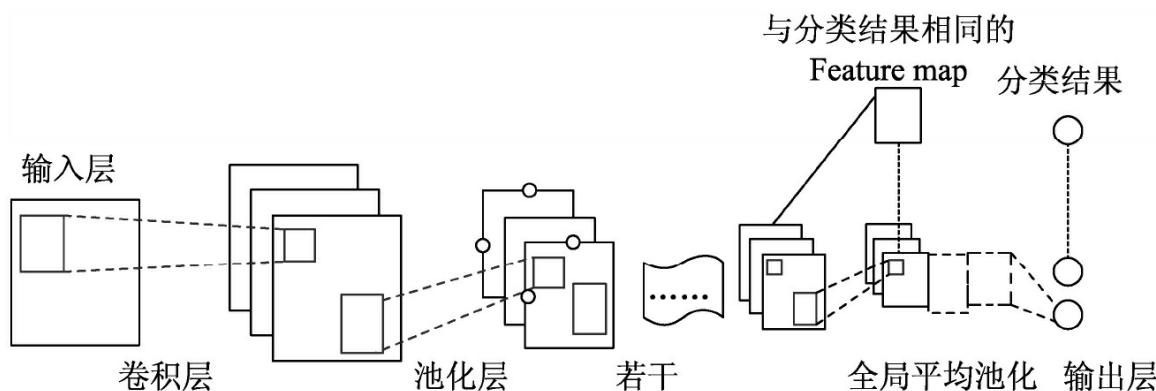


图8-4 卷积网络的完整结构

一个卷积神经网络里包括5部分——输入层、若干个卷积操作和池化层结合的部分、全局平均池化层、输出层：

- 输入层：将每个像素代表一个特征节点输入进来。
- 卷积操作部分：由多个滤波器组合的卷积层。
- 池化层：将卷积结果降维。
- 全局平均池化层：对生成的feature map取

全局平均值。

· 输出层：需要分成几类，相应的就会有几个输出节点。每个输出节点都代表当前样本属于的该类型的概率。

输入层、输出层在前面章节已有介绍，下面重点讲讲卷积操作和池化层。



注意： 全局平均池化层是后出的新技术，在以前的大部分教材里，这个位置通过是使用1~3个全连接层来代替的。全连接层的劣势在于会产生大量的计算，需要大量的参数，但在效果上却和全局平均池化层一样。所以，在这里请读者忘掉全连接层，直接使用效率更高的全局平均池化层。

8.3.2 卷积操作

前面的8.2节中采用sobel因子对图片的操作，可以理解成一次卷积操作。下面来系统地了解卷积操作。卷积分为窄卷积、全卷积和同卷积。

在一一介绍这些卷积类型之前，首先介绍一下步长的概念。

1. 步长

步长是卷积操作的核心。通过步长的变换，可以得到想要的不同类型的卷积操作。先以窄卷积为例，看看它的操作及相关术语，如图8-5所示。

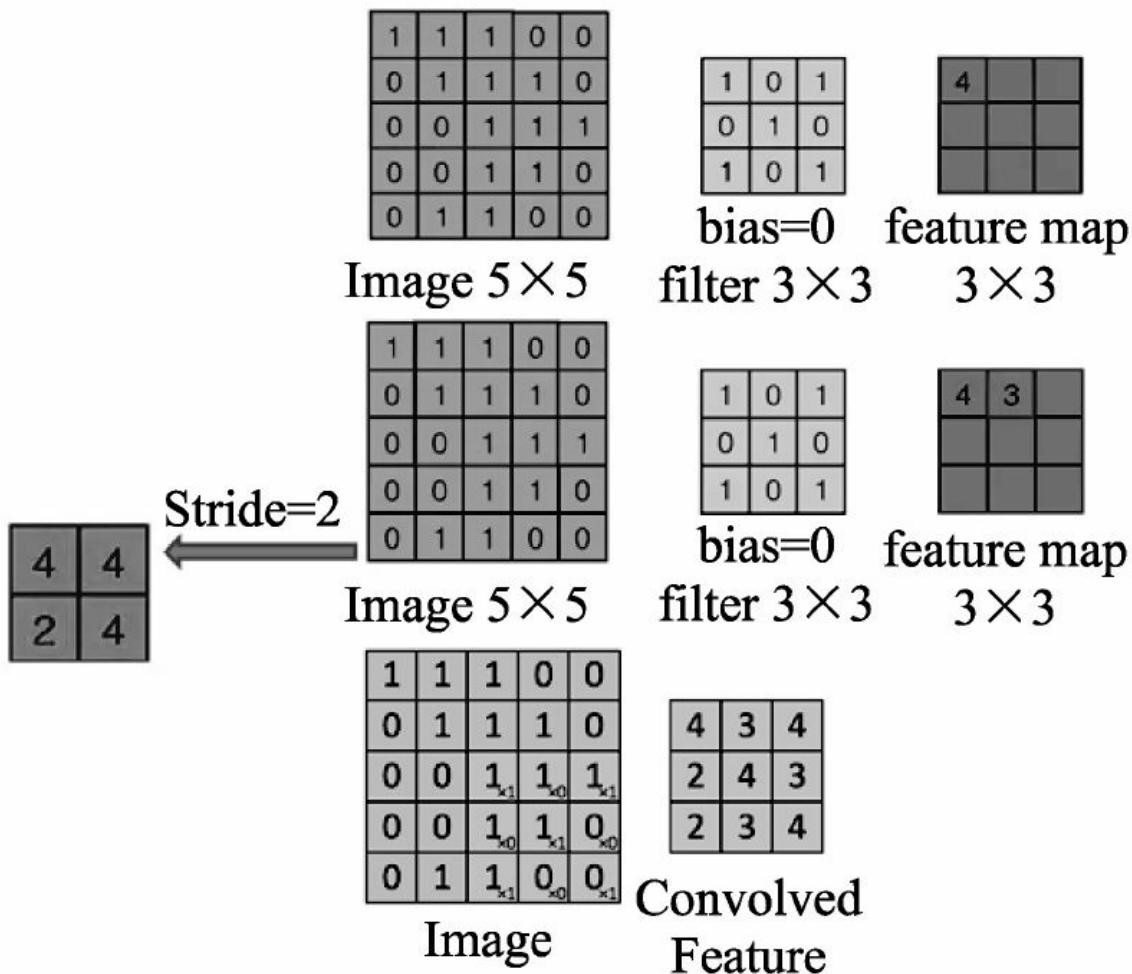


图8-5 卷积细节

图8-5中， 5×5 大小的矩阵代表图片，每个图片右侧的 3×3 矩阵代表卷积核，最右侧的 3×3 矩阵为计算完的结果feature map。

卷积操作仍然是将卷积核（filter）对应的图片（image）中的矩阵数据一一相乘，再相加。图

*****ebook converter DEMO Watermarks*****

8-5中，第一行feature map中的第一个元素，是由image块中前3行3列中的每个元素与filter中的对应元素相乘再相加得到的

$$(4=1\times 1+1\times 0+1\times 1+0\times 0+1\times 1+1\times 0+0\times 1+0\times 0+1\times 1)$$

步长（stride）表示卷积核在图片上移动的格数。

· 当步长为1的情况下，如图8-5中，第二行右边的feature map块里的第二个元素3，是由卷积核计算完第一个元素4，右移一格后计算得来的，相当于图片中的前3行和第1到第4列围成的 3×3 矩阵与卷积核各对应元素进行相乘相加操作 $(3=1\times 1+1\times 0+0\times 1+1\times 0+1\times 1+1\times 0+0\times 1+1\times 0+1)$

· 当步长为2的情况下，就代表每次移动2个格，最终会得到一个如图8-5中第二行左边的 2×2 矩阵块的结果。

2. 窄卷积

窄卷积（valid卷积），从字面上也可以很容易理解，即生成的feature map比原来的原始图片小，它的步长是可变的。假如滑动步长为S，原始图片的维度为 $N_1\times N_1$ ，那么卷积核的大小为 $N_2\times N_2$ ，卷积后的图像大小 $(N_1-N_2)/S+1 \times (N_1-N_2)/S+1$ 。

3. 同卷积

同卷积（same卷积），代表的意思是卷积后的图片尺寸与原始图片的尺寸一样大，同卷积的步长是固定的，滑动步长为1。一般操作时都要使用padding技术（外围补一圈0，以确保生成的尺寸不变）。

4. 全卷积

全卷积（full卷积），也叫反卷积，就是把原始图片里的每个像素点都用卷积操作展开。如图8-6所示，白色的块是原始图片，浅色的是卷积核，深色的是正在卷积操作的像素点。反卷积操作的过程中，同样需要对原有图片进行padding操作，生成的结果会比原有的图片尺寸大。

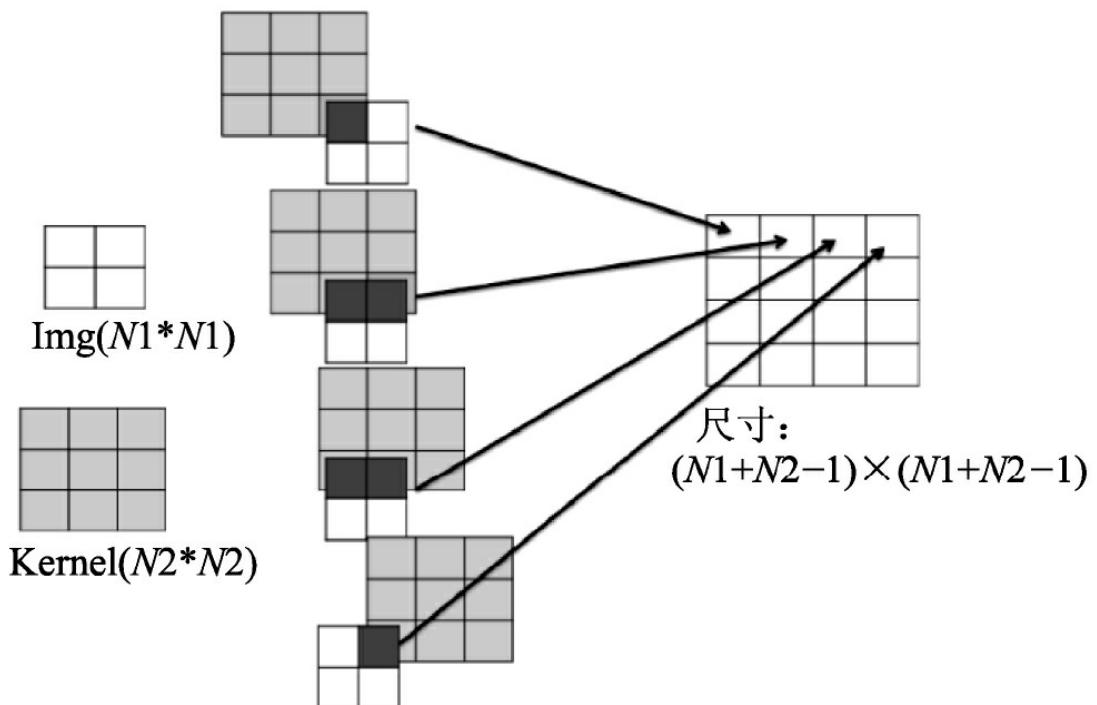


图8-6 反卷积

全卷积的步长也是固定的，滑动步长为1，假如原始图片的维度为 $N_1 \times N_1$ ，那么卷积核的大小为 $N_2 \times N_2$ ，卷积后的图像大小，即 $N_1 + N_2 - 1 \times N_1 + N_2 - 1$

前面的窄卷积和同卷积都是卷积网络里常用的技术，然而全卷积（full卷积）却相反，它更多地用在反卷积网络中，关于反卷积网络的内容，将在后面的章节进行介绍。

5. 反向传播

因为反向传播在框架里已经封装好，不需要对其进行编码修改，所以对于反向传播方面的知识，这里只简单介绍下基本原理，读者知道大概意思即可。

反向传播的核心步骤主要有两步：

- (1) 反向将误差传到前面一层。
- (2) 根据当前的误差对应的学习参数表达式，计算出其需要更新的差值。

对于第(2)步，与前面的反向求导是一样的，仍然使用链式求导法则，找到使误差最小化的梯度，再配合学习率算出更新的差值。将生成的feature map做一次padding后，与转置后的卷积核做一次卷积操作即可得到输入端的误差，从而

实现误差的反向传递。

这里只介绍个概念，具体的计算规则请读者参看后面的反卷积部分，这里不再赘述。

6. 多通道的卷积

通道（Channel），是指图片中每个像素由几个数来表示，这几个数一般指的就是色彩。比如一个灰度图的通道就是1，一个彩色图的通道就是3（红、黄、蓝）。前面介绍的都是单通道的卷积计算，那么对于多通道的卷积计算是什么样的呢？

在卷积神经网络里，通道又分输入通道和输出通道。

· 输入通道：就是前面刚介绍的图片的通道。如是彩色图片，起始的输入通道就是3。如是中间层的卷积，输入通道就是上一层的输出通道个数，计算方法是，每个输入通道的图片都使用同一个卷积核进行卷积操作，生成与输入通道匹配的feature map（比如彩色图片就是3个），然后再把这几张feature map相同位置上的值加起来，生成一张feature map。

· 输出通道：很好理解了，想要输出几个feature map，就放几个卷积核，就是几个输出通道。

8.3.3 池化层

池化的主要目的是降维，即在保持原有特征的基础上最大限度地将数组的维数变小。

池化的操作外表跟卷积很像，只是算法不同：

- 卷积是将对应像素上的点相乘，然后再相加。

- 池化中只关心滤波器的尺寸，不考虑内部的值。算法是，滤波器映射区域内的像素点取平均值或最大值。

池化步骤也有步长，这一点与卷积是一样的。

1. 均值池化

这个很好理解，就是在图片上对应出滤波器大小的区域，对里面的所有不为0的像素点取均值。这种方法得到的特征数据会对背景信息更敏感一些。



注意：一定是不为0的像素点，这个很重要。如果把带0的像素点加上，则会增加分母，从而使整体数据变低。

2. 最大池化

同理，最大池化就是在图片上对应出滤波器大小的区域，将里面的所有像素点取最大值。这种方法得到的特征数据会对纹理特征的信息更敏感一些。

3. 反向传播

池化的反向传播要比卷积容易理解。对于最大池化，直接将其误差还原到对应的位置，其他用0填入；对于均值池化，则是将其误差全部填入该像素对应的池化区域。该部分的详细算法也与反池化算法完全相同，读者可以参看反池化部分的介绍。

8.4 卷积神经网络的相关函数

在TensorFlow中，使用tf.nn.conv2d来实现卷积操作，使用tf.nn.max_pool进行最大池化操作。通过传入不同的参数，来实现各种不同类型的卷积与池化操作。下面介绍这两个函数中各参数的具体意义。

8.4.1 卷积函数tf.nn.conv2d

TensorFlow里使用tf.nn.conv2d函数来实现卷积，其格式如下。

```
tf.nn.conv2d(input, filter, strides, padding, use_cudnn_on_
```

除去参数name参数用以指定该操作的name，与方法有关的共有5个参数。

- input：指需要做卷积的输入图像，它要求是一个Tensor，具有[batch, in_height, in_width, in_channels]这样的形状（shape），具体含义是“训练时一个batch的图片数量，图片高度，图片宽度，图像通道数”，注意这是一个四维的Tensor，要求类型为float32和float64其中之一。

- filter：相当于CNN中的卷积核，它要求是

一个Tensor，具有[filter_height, filter_width, in_channels, out_channels]这样的shape，具体含义是“卷积核的高度，滤波器的宽度，图像通道数，滤波器个数”，要求类型与参数input相同。有一个地方需要注意，第三维in_channels，就是参数input的第四维。

- strides：卷积时在图像每一维的步长，这是一个一维的向量，长度为4。
- padding：定义元素边框与元素内容之间的空间。string类型的量，只能是SAME和VALID其中之一，这个值决定了不同的卷积方式，padding的值为'VALID'时，表示边缘不填充，当其为'SAME'时，表示填充到滤波器可以到达图像边缘。
- use_cudnn_on_gpu：bool类型，是否使用cudnn加速，默认为true。
- 返回值：tf.nn.conv2d函数结果返回一个Tensor，这个输出就是常说的feature map。



注意： 在卷积函数中，padding参数是最容易引起歧义的，该参数仅仅决定是否要补0，因此一定要清楚padding设为SAME的真正含义。在设为SAME的情况下，只有在步长为1时生成的

feature map才会与输入值相等。

8.4.2 padding规则介绍

padding属性的意义是定义元素边框与元素内容之间的空间。

在tf.nn.conv2d函数中，当变量padding为VALID和SAME时，函数具体是怎么计算的呢？其实是有公式的。为了方便演示，先来定义几个变量：

- 输入的尺寸中高和宽定义成in_height、in_width。
- 卷积核的高和宽定义成filter_height、filter_width。
- 输出的尺寸中高和宽定义成output_height、output_width。
- 步长的高宽方向定义成strides_height、strides_width。

1. VALID情况

输出宽和高的公式代码分别为：

```
output_width=(in_width-filter_width + 1)/strides_width (结果  
output_height=(in_height-filter_height+1)/strides_height (结
```

2. SAME情况

输出的宽和高将与卷积核没有关系，具体公式代码如下：

```
out_height = in_height / strides_height (结果向上取整)  
out_width = in_width / strides_width (结果向上取整)
```

这里有一个很重要的知识点——补零的规则，见如下代码：

```
pad_height=max((out_height-1)*strides_height +filter_height·  
pad_width = max((out_width-1)*strides_width +filter_width ·  
pad_top = pad_height / 2  
pad_bottom = pad_height - pad_top  
pad_left = pad_width / 2  
pad_right = pad_width - pad_left
```

上面代码中

- pad_height：代表高度方向要填充0的行数。
- pad_width：代表宽度方向要填充0的列数。
- pad_top、pad_bottom、pad_left、

pad_right: 分别代表上、下、左、右这4个方向填充0的行、列数。

3. 规则举例

下面通过例子来理解一下padding规则。

假设用一个一维数据来举例，输入是13，filter是6，步长是5，对于padding的取值有如下表示：

'VALID'相当于padding，生成的宽度为 $(13-6+1) / 5 = 2$ （向上取整）个数
inputs: 1 2 3 4 5 6 7 8 9 10 11 12 13
 |_____| |_____|
 dropped

'SAME'=相当于padding，生成的宽度为
 $13/5=3$ （向上取整）个数字。

Padding的方式可以如下计算：

Pad_width = $(3-1) \times 5 + 6 - 13 = 3$
Pad_left = pad_width/2 = $3/2 = 1$
Pad_right = pad_width - pad_left = 2

在左边补一个0，右边补2个0。

inputs: pad | 0 1 2 3 4 5 6 7 8 9 10 11 12 :
 |_____| |_____|

*****ebook converter DEMO Watermarks*****

8.4.3 实例37：卷积函数的使用

下面通过一个例子来介绍卷积函数的用法。

实例描述

通过手动生成一个 5×5 的矩阵来模拟图片，定义一个 2×2 的卷积核，来测试tf.nn.conv2d函数里的不同参数，验证其输出结果。

在这个例子中，分为如下几个步骤来写代码。

- (1) 定义输入变量。
- (2) 定义卷积核变量。
- (3) 定义卷积操作。
- (4) 运行卷积操作。

下面就来一一操作。

1. 定义输入变量

定义3个输入变量用来模拟输入图片，分别是 5×5 大小1个通道的矩阵、 5×5 大小2个通道的矩
*****ebook converter DEMO Watermarks*****

阵、 4×4 大小1个通道的矩阵，并将里面的值统统赋为1。

代码8-1 卷积函数使用

```
01 import tensorflow as tf
02
03 # [batch, in_height, in_width, in_channels] [训练时一个批次
图片高度, 图片宽度, 图像通道数]
04 input = tf.Variable(tf.constant(1.0, shape = [1, 5, 5, 1])
05 input2 = tf.Variable(tf.constant(1.0, shape = [1, 5, 5, 2])
06 input3 = tf.Variable(tf.constant(1.0, shape = [1, 4, 4, 1])
```

2. 定义卷积核变量

定义5个卷积核，每个卷积核都是 2×2 的矩阵，只是输入、输出的通道数有差别，分别为1ch输入、1ch输出，1ch输入、2ch输出，1ch输入、3ch输出，2ch输入、2ch输出，2ch输入、1ch输出，并分别在里面填入指定的数值：

代码8-1 卷积函数使用（续）

```
07 # [filter_height, filter_width, in_channels, out_channels]
(卷积核的高度, 卷积核的宽度, 图像通道数, 卷积核个数)
08 filter1 = tf.Variable(tf.constant([-1.0,0,0,-1],shape =
09 filter2 = tf.Variable(tf.constant([-1.0,0,0,-1,-1.0,0,0,
[2, 2, 1, 2]))
10 filter3 = tf.Variable(tf.constant([-1.0,0,0,-1,-1.0,0,0,
0,0,-1],shape = [2, 2, 1, 3]))
11 filter4 = tf.Variable(tf.constant([-1.0,0,0,-1,
12
13
14
15 filter5 = tf.Variable(tf.constant([-1.0,0,0,-1,-1.0,0,0,
```

```
[2, 2, 2, 1]))
```

3. 定义卷积操作

将步骤1的输入与步骤2的卷积核组合起来，建立8个卷积操作，看看生成的内容与前面所讲述的规则是否一致。

代码8-1 卷积函数使用（续）

```
16 # padding的值为'VALID'，表示边缘不填充； 当其为'SAME'时，表示填  
达图像边缘  
17 op1 = tf.nn.conv2d(input, filter1, strides=[1, 2, 2, 1],  
18 op2 = tf.nn.conv2d(input, filter2, strides=[1, 2, 2, 1],  
#1个通道输入，生成2个feature map  
19 op3 = tf.nn.conv2d(input, filter3, strides=[1, 2, 2, 1],  
20  
21 op4 = tf.nn.conv2d(input2, filter4, strides=[1, 2, 2, 1],  
'SAME') # 2个通道输入，生成2个feature  
22 op5 = tf.nn.conv2d(input2, filter5, strides=[1, 2, 2, 1],  
'SAME') # 2个通道输入，生成1个feature map  
23  
24 vop1 = tf.nn.conv2d(input, filter1, strides=[1, 2, 2, 1],  
'VALID') # 5*5 对于padding不同而不同  
25 op6 = tf.nn.conv2d(input3, filter1, strides=[1, 2, 2, 1],  
'SAME')  
26 vop6 = tf.nn.conv2d(input3, filter1, strides=[1, 2, 2, 1]  
'VALID') #4*4与padding无关
```

这么多卷积操作看着有点混乱，按照演示的目的将其分类一下，分别介绍。

（1）演示padding补0的情况

如上文代码，op1使用了padding= SAME的一

一个通道输入、一个通道输出的卷积操作，步长为 2×2 ，按前面的函数介绍，这种情况TensorFlow会对input补0。通过前面的公式计算，会生成 3×3 大小的矩阵，并且在右侧和下侧各补一圈0，由 5×5 矩阵变成 6×6 矩阵，如图8-7所示。

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

图8-7 padding例子

(2) 演示多通道输出时的内存排列

op2示例了1个通道生成2个输出，op3示例了1个通道生成3个输出，可以看下它们在内存中的排列样子。

(3) 演示卷积核对多通道输入的卷积处理

op4示例了2个通道生成2个输出，op5示例了2个通道生成1个输出，比较下对于2个通道的卷积结果，观察是多通道的结果叠加，还是每个通道单独对应一个卷积核进行输出。

(4) 验证不同尺寸下的输入受到padding为SAME和VALID的影响

op1和vop1示例了 5×5 尺寸输入在padding为SAME和VALID时的变化情况，op6 和vop6示例了 4×4 尺寸输入在padding为SAME和VALID下的变化情况。

4. 运行卷积操作

在本步操作之前，读者可以把前面的规则熟记一下，然后试着自己推导一下，比较得到的输出结果。下面把这些结果打印出来，看看与你推导的是否一致。

代码8-1 卷积函数使用（续）

```
27 init = tf.global_variables_initializer()
28 with tf.Session() as sess:
29     sess.run(init)
30
31     print("op1:\n", sess.run([op1, filter1]))      #1-1 后面
32     print("-----")
33
34     print("op2:\n", sess.run([op2, filter2]))      #1-2多卷积核
35     print("op3:\n", sess.run([op3, filter3]))      #1-3一个输入
36     print("-----")
37
38     print("op4:\n", sess.run([op4, filter4]))      #2-2通道叠加
39     print("op5:\n", sess.run([op5, filter5]))      #2-1两个输入
40     print("-----")
41
42     print("op1:\n", sess.run([op1, filter1]))      #1-1一个输入
43     print("vop1:\n", sess.run([vop1, filter1]))
44     print("op6:\n", sess.run([op6, filter1]))
45     print("vop6:\n", sess.run([vop6, filter1]))
```

下面分别是介绍这段代码的执行结果。

(1) 执行代码8-1中的31和32行代码，对应的输出如下：（为了看起来方便，将格式进行了整理）

```
op1:  
[array([[[[-2.],[-2.],[-1.]],  
        [[-2.],[-2.],[-1.]],  
        [[-1.],[-1.],[-1.]]], dtype=float32),  
 array([[[[-1.],[[ 0.]]],  
       [[[ 0.],[[-1.]]]], dtype=float32)]
```

上面输出中 5×5 矩阵通过卷积操作生成了 3×3 矩阵，对padding的补0情况是在后面和下面补0，所以会在矩阵的右边和下边生成-1。

(2) 执行代码8-1中的34~36行，对应的输出如下：

```
op2:  
[array([[[[-2.,-2.],[-2.,-2.],[-2.,0.]],  
        [[-2.,-2.],[-2.,-2.],[-2.,0.]],  
        [[-1.,-1.],[-1.,-1.],[-1.,0.]]], dtype=float32),  
 array([[[[-1.,0.],[[ 0.,-1.]]],  
       [[[ -1.,0.],[[ 0.,-1.]]]], dtype=float32)]  
op3:  
[array([[[[-2.,-2.,-2.],[-2.,-2.,-2.],[-1.,-1.,-1.]],  
        [[-2.,-2.,-2.],[-2.,-2.,-2.],[-1.,-1.,-1.]],  
        [[-2.,-1.,0.],[-2.,-1.,0.],[-1.,0.,0.]]], dtype=float32),  
 array([[[[-1.,0.,0.],[[-1.,-1.,0.]]],  
       [[[ 0.,-1.,-1.],[[ 0.,0.,-1.]]]], dtype=float32)]
```

上面输出中，生成的多通道的输出，是按列排列的（每一个feature map为一列）。为了方便理解，以op2为例，其剖析图如图8-8所示。

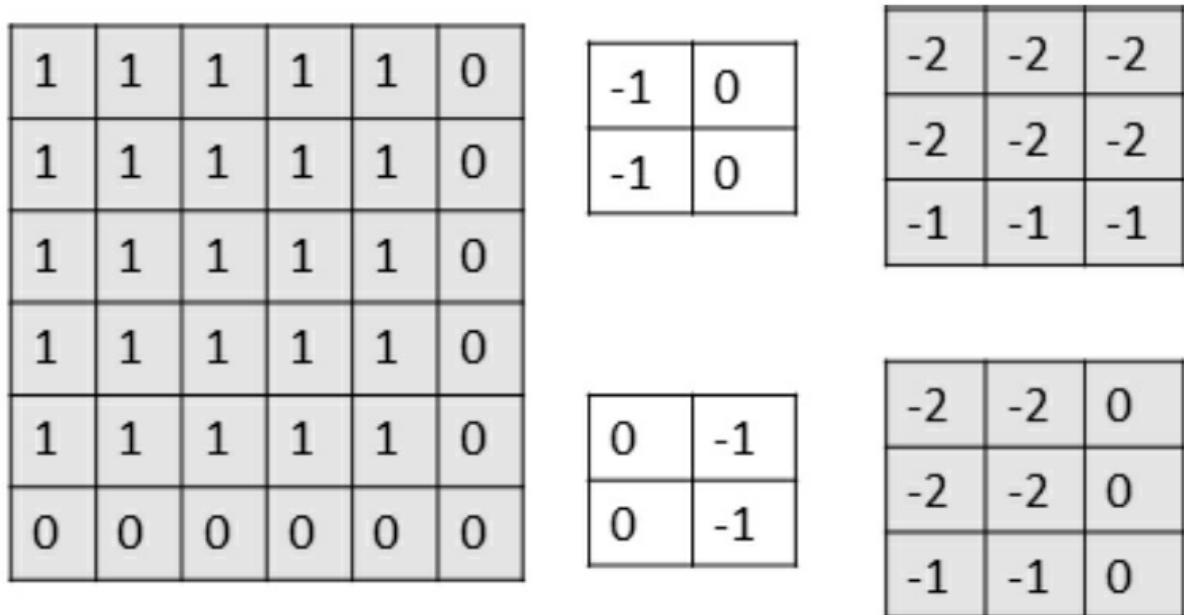


图8-8 卷积示例

(3) 执行代码8-1中的33~39行，对应的输出如下：

```
op4:  
[array([[[[-4., -4.], [-4., -4.], [-2., -2.]],  
        [[-4., -4.], [-4., -4.], [-2., -2.]],  
        [[-2., -2.], [-2., -2.], [-1., -1.]]]], dtype=float32),  
 array([[[[-1., 0.], [0., -1.]], [[-1., 0.], [0., -1.]]],  
       [[[1., 0.], [0., -1.]], [[1., 0.], [0., -1.]]]], dtype=float32)]  
op5:  
[array([[[[-4.], [-4.], [-2.]],  
        [[-4.], [-4.], [-2.]],  
        [[-2.], [-2.], [-1.]]]], dtype=float32),  
 array([[[[-1.], [0.]], [[0.], [-1.]]],  
       [[[1.], [0.]], [[0.], [-1.]]]], dtype=float32)]
```

卷积核对多通道输入的卷积处理，是多通道的结果叠加。以op5为例展开。图8-9所示为将每个通道的feature map叠加生成了最终的结果。

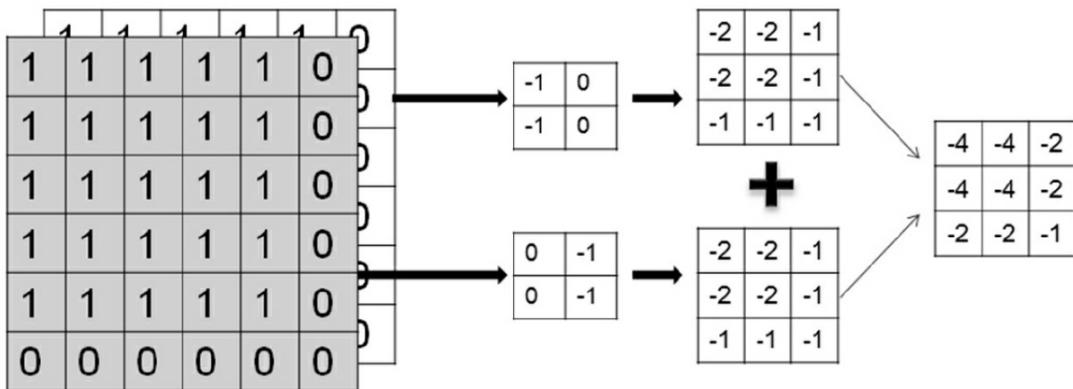


图8-9 多通道卷积

(4) 执行代码8-1中的42~45行，对应的输出如下，是不同尺寸输入分别为SAME和VALID时的比较。

```

op1:
[array([[[[-2.],[-2.],[-1.]],
         [[-2.],[-2.],[-1.]],
         [[-1.],[-1.],[-1.]]]],dtype=float32),
array([[[[-1.],[[0.]]],
       [[[0.],[[-1.]]]]],dtype=float32)]
vop1:
[array([[[[-2.],[-2.]],
         [[-2.],[-2.]]]],dtype=float32),
array([[[[-1.],[[0.]]],
       [[[0.],[[-1.]]]]],dtype=float32)]
op6:
[array([[[[-2.],[-2.]],
         [[-2.],[-2.]]]],dtype=float32),
array([[[[-1.],[[0.]]],
       [[[0.],[[-1.]]]]],dtype=float32)]
vop6:
[array([[[[-2.],[-2.]],
         [[-2.],[-2.]]]],dtype=float32),

```

```
array([[[[-1.]], [[0.]]],  
      [[[0.]], [[-1.]]]], dtype=float32)]
```

通过上面的结果可以看出：

- 对于op1和vop1的比较可以看出 5×5 矩阵在padding为'SAME'时生成的是 3×3 矩阵，而在'VALID'时生成的是 2×2 。
- 而在op6和vop6的例子中，对于 4×4 矩阵在padding为'SAME'和'VALID'下都会生成 2×2 的矩阵，这是因为 4×4 的输入对于 2×2 的卷积核步长为2的情况下，正好可以把所有数据处理完，本身在'SAME'的情况下就不需要补0。

通过卷积函数可以实现8.4.2节卷积操作中的窄卷积和同卷积（步长唯一并且补零操作的卷积），但不能实现全卷积。TensorFlow中有单独的反卷积函数，会在后面会讲到。



注意：本节特意用了很多篇幅来解释卷积的操作细节，表明这部分内容非常重要，是卷积神经网络的重点。将卷积操作的细节理解透彻，会使你在实际编程过程中少走弯路。因为在自己搭建网络的过程中，必须对输入、输出的具体维度有个清晰的计算，这样才能保证网络结构的正确性，才能使网络运行下去。

8.4.4 实例38：使用卷积提取图片的轮廓

通过8.4.3节的练习，相信读者已经掌握了卷积操作的细节。下面来做一个实际的例子，通过卷积操作来实现本章开篇所讲的sobel算子。

实例描述

通过卷积操作来实现本章开篇所讲的sobel算子，将彩色的图片生成带有边缘化信息的图片。

本例中先载入一个图片，然后使用一个“3通道输入，1通道输出的 3×3 卷积核”（即sobel算子），最后使用卷积函数输出生成的结果。

1. 载入图片并显示

首先将图片放到代码的同级目录下，通过imread载入，然后将其显示并打印出来。

代码8-2 sobel

```
01 import matplotlib.pyplot as plt # plt 用于显示图片
02 import matplotlib.image as mpimg # mpimg 用于读取图片
03 import numpy as np
04 import tensorflow as tf
05
06 myimg = mpimg.imread('img.jpg') # 读取和代码处于同一目录下的图
07 plt.imshow(myimg) # 显示图片
08 plt.axis('off') # 不显示坐标轴
09 plt.show()
10 print(myimg.shape)
```

运行上面代码，得到输出如下，输出图片如图8-10所示。



图8-10 图片显示

(3264, 2448, 3)

可以看到，载入的图片维度为 3264×2448 大小，3个通道。

2. 定义占位符、卷积核、卷积op

这里需要手动将sobel算子填入到卷积核里。使用tf.constant函数可以将常量直接初始化到Variable中，因为是3通道，所以sobel卷积核的每个元素都扩成了3个。



注意：sobel算子处理过的图片不保证每个像素都在0~255之间，所以要做一次归一化操作（即用每个值减去最小值的结果，再除以最大值与最小值的差），让生成的值都在[0, 1]之间，然后再乘以255。

代码8-2 sobel (续)

```
11 full=np.reshape(myimg,[1,3264,2448,3])
12 inputfull = tf.Variable(tf.constant(1.0,shape = [1, 3264,
13
14 filter = tf.Variable(tf.constant([[-1.0,-1.0,-1.0], [0,0,
15                               [-2.0,-2.0,-2.0], [0,0,0],
16                               [-1.0,-1.0,-1.0],[0,0,0],[1
17                               shape = [3, 3, 3, 1]]))
18 op= tf.nn.conv2d(inputfull, filter, strides=[1, 1, 1, 1],
#3个通道输入，生成1个feature ma
19 o=tf.cast( ((op-tf.reduce_min(op))/(tf.reduce_max(op)-t1
(op)) ) *255 ,tf.uint8)
```

上面的代码中，卷积op的步长为 1×1 ，padding为SAME表明这是个同卷积的操作。

3. 运行卷积操作并显示

现在就可以建立session然后运行程序了。具体代码如下。

代码8-2 sobel (续)

```
20 with tf.Session() as sess:  
21     sess.run(tf.global_variables_initializer() )  
22  
23     t,f=sess.run([o,filter],feed_dict={ inputfull:  
24         full})  
25     t=np.reshape(t,[3264,2448])  
26  
27     plt.imshow(t,cmap='Greys_r') #显示图片  
28     plt.axis('off')           #不显示坐标轴  
29     plt.show()
```

上述代码执行后输出结果如图8-11所示。

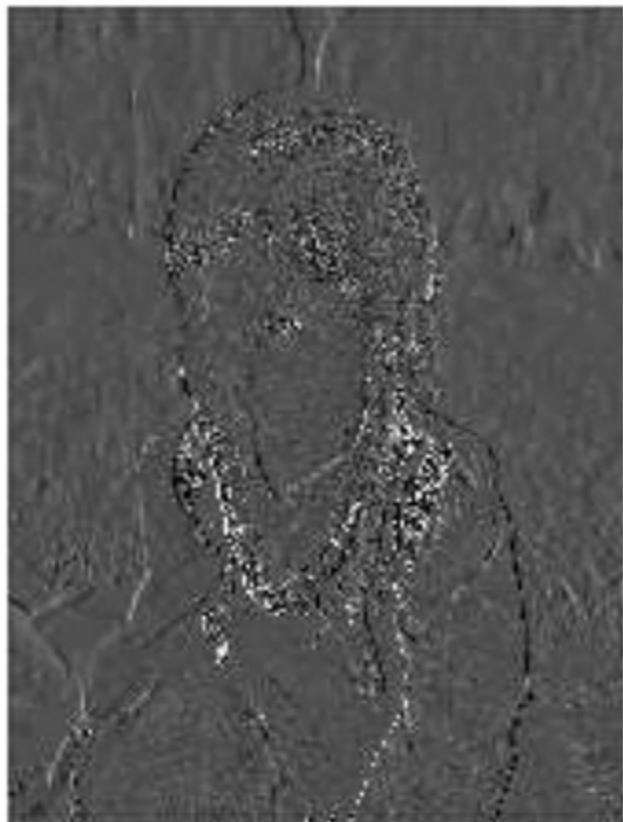


图8-11 边缘化

可以看出，sobel的卷积操作之后，提取到了一张含有轮廓特征的图像。

8.4.5 池化函数tf.nn.max_pool (avg_pool)

TensorFlow里的池化函数如下：

```
tf.nn.max_pool(input, ksize, strides, padding, name=None)
tf.nn.avg_pool(input, ksize, strides, padding, name=None)
```

这两个函数中的4个参数和卷积参数很类似，具体说明如下。

- value：需要池化的输入，一般池化层接在卷积层后面，所以输入通常是feature map，依然是[batch, height, width, channels]这样的shape。
- ksize：池化窗口的大小，取一个四维向量，一般是[1, height, width, 1]，因为我们不想在batch和channels上做池化，所以这两个维度设为了1。
- strides：和卷积参数含义类似，窗口在每一个维度上滑动的步长，一般也是[1, stride, stride, 1]。
- padding：和卷积参数含义一样，也是取VALID或者SAME，VALID是不padding操作，SAME是padding操作。
- 返回一个Tensor，类型不变，shape仍然是

[batch, height, width, channels]这种形式。

8.4.6 实例39：池化函数的使用

下面通过一个例子来介绍池化函数的用法。

实例描述

通过手动生成一个 4×4 的矩阵来模拟图片，定义一个 2×2 的滤波器，通过几个在卷积神经网络中常用的池化操作来测试池化函数里的参数，并验证输出结果。

1. 定义输入变量

定义1个输入变量用来模拟输入图片、 4×4 大小的2通道矩阵，并将其赋予指定的值。2个通道分别为：4个0到3 3 3 3组成的矩阵，4个4到7 7 7 7组成的矩阵。

代码8-3 池化函数使用

```
01 import tensorflow as tf
02
03 img=tf.constant([
04     [[0.0,4.0],[0.0,4.0],[0.0,4.0],[0.0,4.0]],
05     [[1.0,5.0],[1.0,5.0],[1.0,5.0],[1.0,5.0]],
06     [[2.0,6.0],[2.0,6.0],[2.0,6.0],[2.0,6.0]],
07     [[3.0,7.0],[3.0,7.0],[3.0,7.0],[3.0,7.0]]
08 ])
09 img=tf.reshape(img,[1,4,4,2])
```

2. 定义池化操作

这里定义了4个池化操作和一个取均值操作。前两个操作是最大池化操作，接下来是两个均值池化操作，最后一个是对均值操作。

代码8-3 池化函数使用（续）

```
10 pooling=tf.nn.max_pool(img,[1,2,2,1],[1,2,2,1],padding='\\
11 pooling1=tf.nn.max_pool(img,[1,2,2,1],[1,1,1,1],padding=
12 pooling2=tf.nn.avg_pool(img,[1,4,4,1],[1,1,1,1],padding=
13 pooling3=tf.nn.avg_pool(img,[1,4,4,1],[1,4,4,1],padding=
14 nt_hpool2_flat = tf.reshape(tf.transpose(img), [-1, 16])
15 pooling4=tf.reduce_mean(nt_hpool2_flat,1) #1表示对行求均值
0表示对列求均值
```

3. 运行池化操作

在本步骤操作之前，读者可以把前面的规则熟记一下，然后试着自己推导一下，比较得到的输出结果。下面把这些结果打印出来，看看与你推导的是否一致。

代码8-3 池化函数使用（续）

```
16 with tf.Session() as sess:
17     print("image:")
18     image=sess.run(img)
19     print (image)
20     result=sess.run(pooling)
21     print ("reslut:\n",result)
22     result=sess.run(pooling1)
23     print ("reslut1:\n",result)
24     result=sess.run(pooling2)
```

*****ebook converter DEMO Watermarks*****

```
25     print ("reslut2:\n", result)
26     result=sess.run(pooling3)
27     print ("reslut3:\n", result)
28     flat,result=sess.run([nt_hpool2_flat,pooling4])
29     print ("reslut4:\n", result)
30     print("flat:\n",flat)
```

执行上面的代码，得到如下输出（为了方便读者观看，这里将格式进行了整理）：

```
image:
[[[[ 0.  4.]
   [ 0.  4.]
   [ 0.  4.]
   [ 0.  4.]]

[[ 1.  5.]
 [ 1.  5.]
 [ 1.  5.]
 [ 1.  5.]]

[[ 2.  6.]
 [ 2.  6.]
 [ 2.  6.]
 [ 2.  6.]]

[[ 3.  7.]
 [ 3.  7.]
 [ 3.  7.]
 [ 3.  7.]]]]
```

通过上面的输出可以看出，img与我们设置的初始值是一样的，即第一个通道为[[0 0 0 0], [1 1 1 1], [2 2 2 2], [3 3 3 3]]；第二个通道为[[4 4 4 4], [5 5 5 5], [6 6 6 6], [7 7 7 7]]。

```
reslut:
[[[[ 1.  5.]
```

*****ebook converter DEMO Watermarks*****

```
[ 1.  5.]  
[[ 3.  7.]  
[ 3.  7.]]]]
```

这个操作在卷积神经网络中是最常用的，一般步长都会设成与池化滤波器尺寸一致（池化的卷积尺寸为 2×2 ，所以步长也是2），生成2个通道的 2×2 矩阵。矩阵的内容是从原始输入中取最大值，由于池化filter中对应通道的维度是1，所以结果仍然保持源通道数。

```
reslut1:  
[[[[ 1.  5.]  
[ 1.  5.]  
[ 1.  5.]])  
  
[[ 2.  6.]  
[ 2.  6.]  
[ 2.  6.]])  
  
[[ 3.  7.]  
[ 3.  7.]  
[ 3.  7.]]]]  
reslut2:  
[[[[ 1.  5. ]  
[ 1.  5. ]  
[ 1.  5. ]  
[ 1.  5. ]])  
  
[[ 1.5  5.5]  
[ 1.5  5.5]  
[ 1.5  5.5]  
[ 1.5  5.5]])  
  
[[ 2.  6. ]  
[ 2.  6. ]  
[ 2.  6. ]  
[ 2.  6. ]])  
  
[[ 2.5  6.5]]
```

*****ebook converter DEMO Watermarks*****

```
[ 2.5  6.5]  
[ 2.5  6.5]  
[ 2.5  6.5]]]
```

result1和result2分别演示了VALID和SAME的两种padding的取值。

- VALID中使用的filter为 2×2 , 步长为 1×1 , 生成了 2×2 大小的矩阵。
- 在SAME中使用的 4×4 的filter, 步长仍然为 1×1 , 生成了 4×4 的矩阵, padding之后在计算avg_pool时, 是将输入矩阵与filter对应尺寸内的元素总和除以这些元素中非零的个数(而不是filter的总个数)。

```
reslut3:  
[[[[ 1.5  5.5]]]]  
reslut4:  
[ 1.5  5.5]  
flat:  
[[ 0.  1.  2.  3.  0.  1.  2.  3.  0.  1.  2.  3.  0.  1.  
 [ 4.  5.  6.  7.  4.  5.  6.  7.  4.  5.  6.  7.  4.  5. ]
```

result3是常用的操作手法, 也叫全局池化法, 就是使用一个与原有输入同样尺寸的filter进行池化, 一般放在最后一层, 用于表达图像通过卷积网络处理后的最终特征。而result4是一个均值操作, 可以看到将数据转置后的均值操作得到的值与全局池化平均值是一样的结果。

8.5 使用卷积神经网络对图片分类

本节练习使用卷积网络对CIFAR数据集进行分类。在前面接触到了MNIST数据集，它是一堆手写图片，CIFAR也是一堆图片，会比MNIST更为复杂。卷积神经网络最擅长的就是图像数据的处理，所以在CIFAR数据集上做图像识别的练习会更有意思。下面就先从CIFAR开始介绍。

8.5.1 CIFAR介绍

CIFAR由 Alex Krizhevsky、Vinod Nair和 Geoffrey Hinton收集而来，起初的数据集共分为10类，分别为飞机、汽车、鸟、猫、鹿、狗、青蛙、马、船、卡车，所以CIFAR数据集常以CIFAR-10命名。CIFAR共包含60 000张 32×32 的彩色图像（包含50 000张训练图片，10 000张测试图片），其中没有任何类型重叠的情况。

因为是彩色图像，所以这个数据集是三通道的，分别是R，G，B 3个通道。后来CIFAR又出了一个分类更多的版本叫CIFAR-100，从名字也可以看出共有100类，将图片分得更细，当然对神经网络图像识别是更大的挑战了。有了这些数据，我们可以把精力全部投在网络优化上。

CIFAR的官网

*****ebook converter DEMO Watermarks*****

为<http://www.cs.toronto.edu/~kriz/cifar.html>，不同于MNIST数据集，它的数据集是已经打包好的文件（如图8-12所示），分别为Python、MATLIB、二进制bin文件包，以方便不同的程序读取。



图8-12 CIFAR数据集

8.5.2 下载CIFAR数据

与MNIST类似，TensorFlow中同样有一个下载和导入CIFAR数据集的代码文件，不同的是，自从TensorFlow1.0之后，将里面的Models模块分离了出来。下载和导入CIFAR数据集的代码在models里面，所以要先去TensorFlow的GitHub网

*****ebook converter DEMO Watermarks*****

站将其下载下来。

如果你使用Git，可以直接用下面的命令下载：

```
git clone https://github.com/tensorflow/models.git
```

如果没有使用Git，可直接复制上面的网址，在右下角单击clone or download按钮，在下方单击Download ZIP按钮下载代码的压缩包，如图8-13所示。

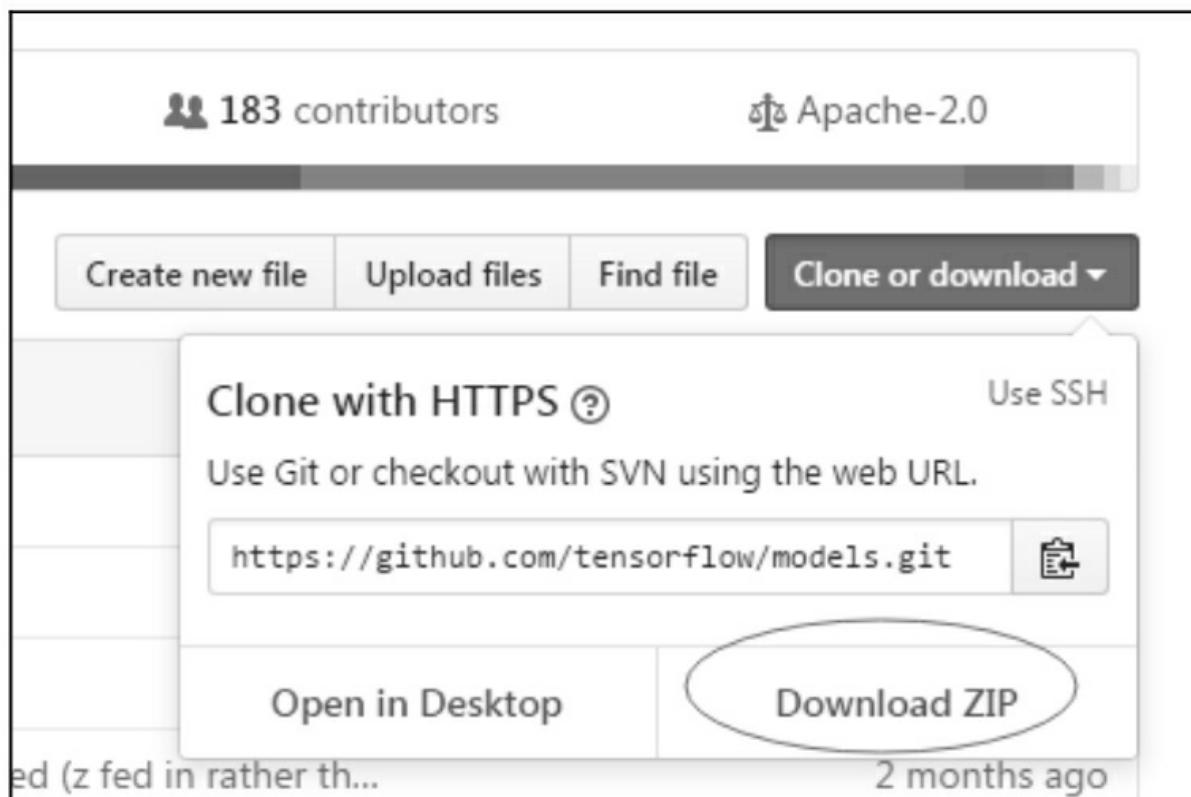


图8-13 model代码下载

代码下载后，将其解压，将里面

models/tutorials/image/路径下的CIFAR10复制到本地的Python工作区即可。

现在可以在CIFAR10文件夹下新建Python文件，用来下载和导入CIFAR10图片了。与MNIST不同的是，CIFAR数据集代码不是很方便，下载和导入时都需要单独调用，所以本节的例子代码第一个步会有一个独立的代码文件。

将如下代码文件放到cifar10文件夹下（确保import cifar10能找到对应文件），引入CIFAR10，使用函数maybe_download_and_extract即可完成数据的下载和解压。

代码8-4 CIFAR下载

```
import cifar10
cifar10.maybe_download_and_extract()
```

上面的代码会自动将CIFAR10的bin文件ZIP包下载到\tmp\cifar10_data路径下（如果是Windows就是本地磁盘下的这个路径，如D:\tmp\cifar10_data），然后自动解压到\tmp\cifar10_data\cifar-10-batches-bin路径下。

以上这个环节也可以手动下载，然后解压到指定路径里。



注意： cifar10.py也可以单独运行，但主要功能并不是下载和解压，所以以库的形式引入到代码里，在Anaconda里面不是太友好，运行第二次时会报错，需要重启console才可以。不过好在我们只运行一次，下载并解压后就不需要了。

在两行代码之后，会看到对应路径下生成的相关文件，如图8-14所示。

其中

- batches.meta.txt： 标签说明文件。
- data_batch_x.bin： 是训练文件，一共有5个，每个10 000条。
- test.batch.bin： 10 000条测试文件。

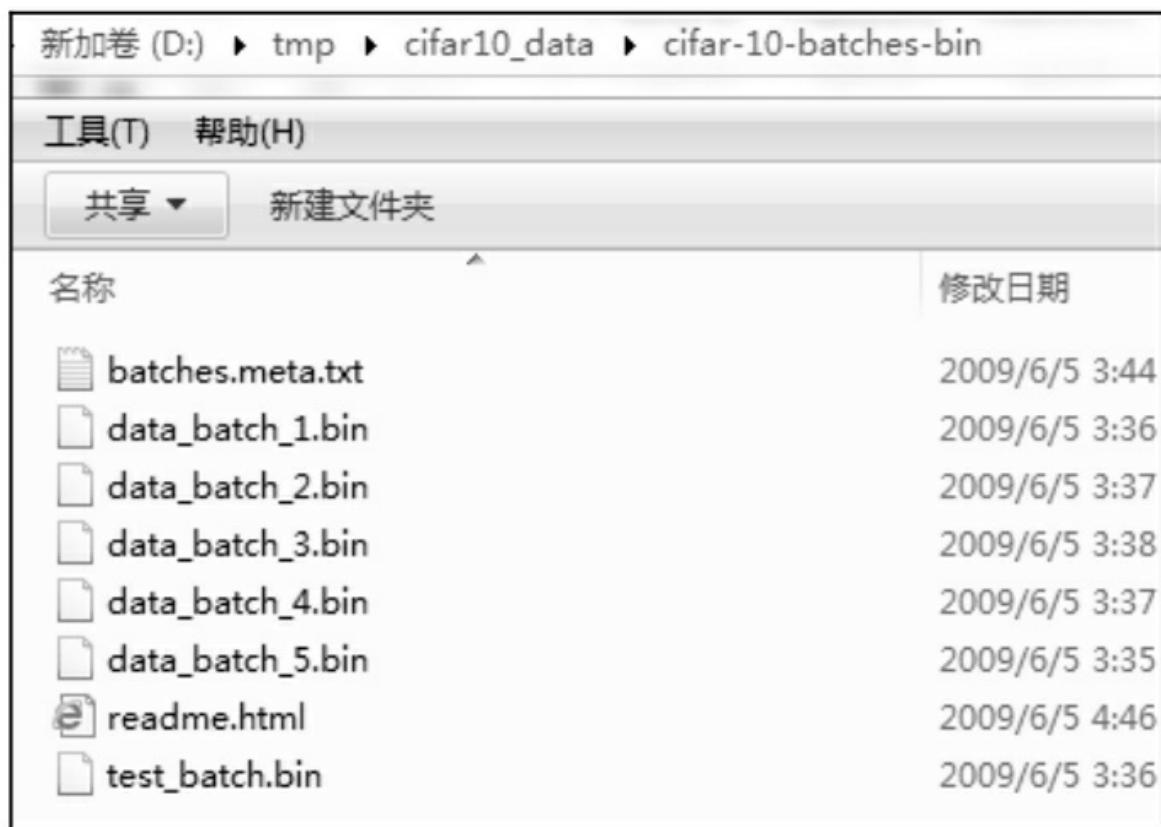


图8-14 生成CIFAR文件

8.5.3 实例40：导入并显示CIFAR数据集

这里通过import cifar10_input来导入CIFAR数据集，cifar10_input.py里定义了这获取数据的函数，具体调用见代码。

代码8-5 CIFAR

```
01 import cifar10_input  
02 import tensorflow as tf  
03 import pylab  
04  
05 #取数据  
06 batch_size = 128  
07 data_dir = '/tmp/cifar10_data/cifar-10-batches-bin'
```

*****ebook converter DEMO Watermarks*****

```
08 images_test, labels_test = cifar10_input.inputs(eval_data=True)
dir = data_dir, batch_size = batch_size)
```

cifar10_input.inputs是专门获取数据的函数，返回数据集和对应的标签，但是cifar10_input.inputs函数会将图片裁剪好，由原来的 $32 \times 32 \times 3$ ，变成了 $24 \times 24 \times 3$ 。该函数默认是使用测试数据集，如果使用训练数据集，可以将第一个参数传入eval_data=False。

另外，再将batch_size和dir传入，就可以得到dir下面的batch_size个数据了。



注意： 这里所获得的图片并不是原始图片，是经过了两次变换，首先将 32×32 尺寸裁剪成了 24 尺寸，然后又进行了一次图片标准化（减去均值像素，并除以像素方差）。这样做的好处是，使所有的输入都在一个有效的数据分布之内，便于特征的分类处理，会使梯度下降算法的收敛更快。

cifar10_input.py中除了对图像进行了一些预处理，还提供了一个读取大数据的方法示例，即使用queue的方法示例。queue是TensorFlow里常用的方法，尤其是在使用大数据样本做训练时。

关于队列方面的内容会在8.5.8节详细介绍。现在我们将cifar10_input.inputs函数得到的内容显

*****ebook converter DEMO Watermarks*****

示出来。

代码8-5 CIFAR（续）

```
09 sess = tf.InteractiveSession()
10 tf.global_variables_initializer().run()
11 tf.train.start_queue_runners()
12 image_batch, label_batch = sess.run([images_test, labels_
13 print("__\n",image_batch[0])
14 print("__\n",label_batch[0])
15 pylab.imshow(image_batch[0])
16 pylab.show()
```

运行上面的代码，主要显示内容如下：（部分内容略去）

```
[[[1.24836731  0.04940184 -1.49835348]
 [ 1.117571    0.02760247 -1.56375158]
 [ 1.24836731  0.18019807 -1.41115606]
 .....
 [-1.58555102 -0.40838495  0.5943861 ]
 [-1.82534409 -0.58277994  0.46358991]
 [-1.56375158 -0.23398998  0.89957732]]]
```

—
3

代码中，session用的是tf.InteractiveSession函数，原因在后面讲队列时会讲，又额外使用了一个train.start_queue_runners函数，是运行队列的意思。上面代码的输出是图片像素数据和标签数据。可以看到，读取的数据都是进过标准化处理的（变成了均值为0，方差为1的数据分布），所以输出的图片就是乱的（如图8-15所示）。

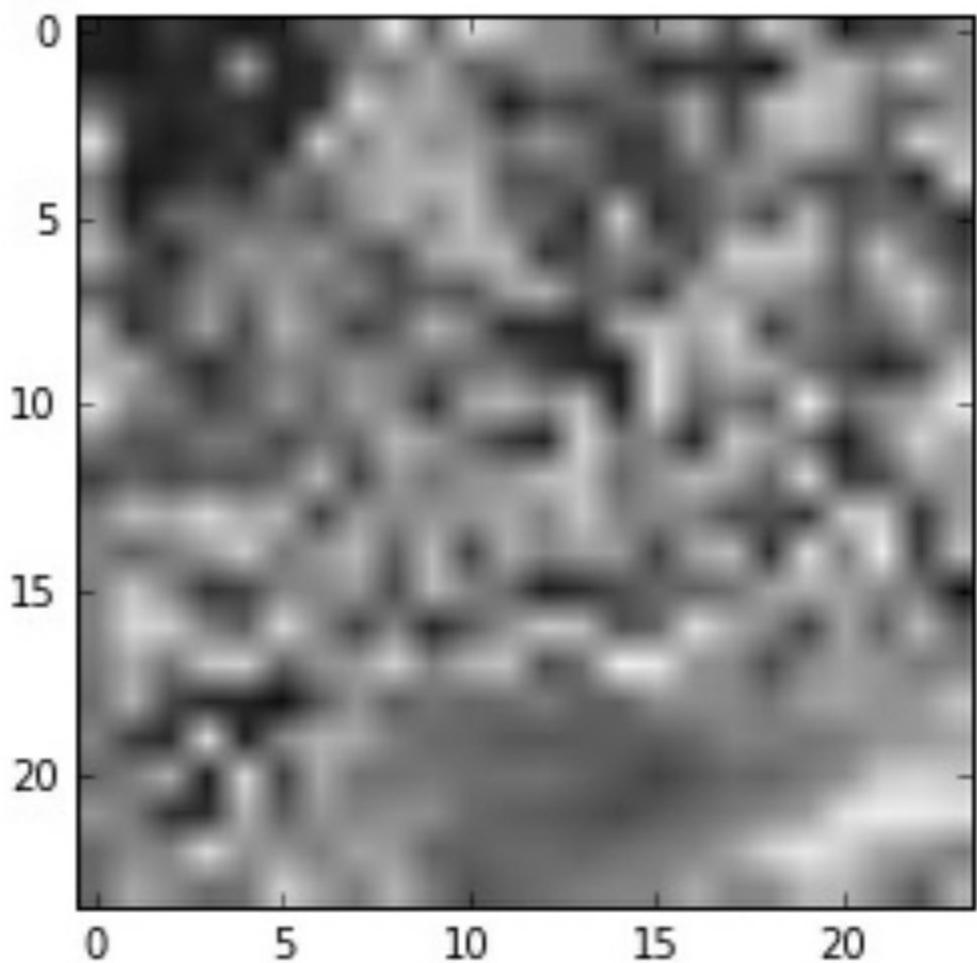


图8-15 CIFAR归一化输出

8.5.4 实例41：显示CIFAR数据集的原始图片

如果希望看到正常的数据怎么办呢？有两种方式：

- 修改cifar10_input.py文件，先让它不去标准化。
- 手动读取数据并显示。

先来看第一种方式。直接在cifar10_input.py文件里做如下修改：在240行后添加一行代码，并随后将243行代码的内容改为注释。

```
.....  
float_image = resized_image  
# Subtract off the mean and divide by the variance of the  
#float_image = tf.image.per_image_standardization(resized_
```

再次运行代码“8-5 cifar.py”文件，输出如图8-16所示。

可以看出是一只松鼠，但图片仍然是被裁剪过的尺寸 $24 \times 24 \times 3$ 。

另一种方式是自己手动编写代码，参见下面的具体内容：

代码8-6 cifar手动读取

```
import numpy as np  
from scipy.misc import imsave  
  
filename = '/tmp/cifar10_data/cifar-10-batches-  
bin/test_batch.bin'  
  
bytestream = open(filename, "rb")  
buf = bytestream.read(10000 * (1 + 32 * 32 * 3))  
bytestream.close()  
  
data = np.frombuffer(buf, dtype=np.uint8)  
data = data.reshape(10000, 1 + 32*32*3)  
labels_images = np.hsplit(data, [1])  
labels = labels_images[0].reshape(10000)  
images = labels_images[1].reshape(10000, 32, 32, 3)
```

*****ebook converter DEMO Watermarks*****

```
img = np.reshape(images[0], (3, 32, 32)) #导出第一幅图
img = img.transpose(1, 2, 0)

import pylab
print(labels[0])
pylab.imshow(img)
pylab.show()
```

运行上面的代码，显示内容如图8-17所示。

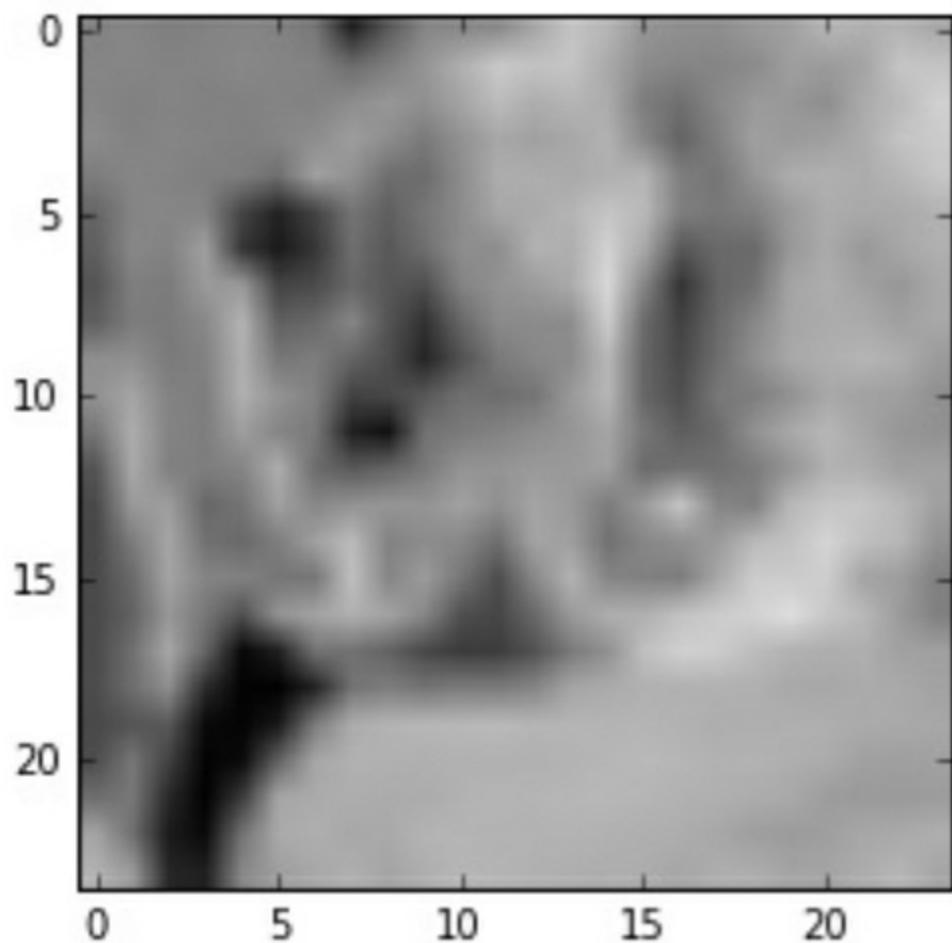


图8-16 CIFAR图片输出1

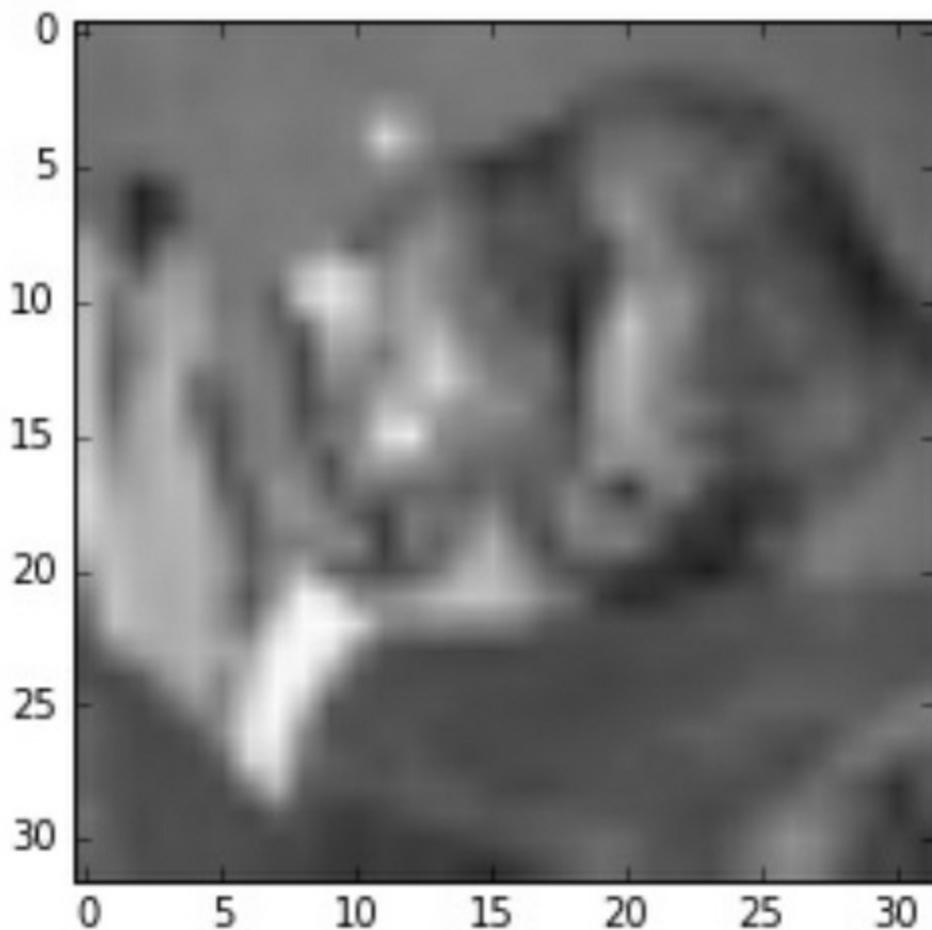


图8-17 CIFAR图片输出2

这次得到的是真实的原始图片，尺寸为 $32 \times 32 \times 3$ 。

8.5.5 cifar10_input的其他功能

cifar10_input.py文件里还有个功能更强大的数据——distorted_inputs，可以在代码里找到其实现。它是针对train数据的，对train数据进行了变形处理，起到一个数据增广的作用。在数据集比较小、数据量远远不够的情况下，可以对图片进

行翻转、随机剪切等操作以增加数据，制造出更加多的样本，提高对图片的利用率。

这部分功能的核心代码在cifar10_input.py文件的第169~183行。具体代码如下：

```
# Randomly crop a [height, width] section of the image.  
distorted_image = tf.random_crop(reshaped_image, [height,  
  
# Randomly flip the image horizontally.  
distorted_image = tf.image.random_flip_left_right(distorted_image)  
  
# Because these operations are not commutative, consider  
# the order their operation.  
distorted_image = tf.image.random_brightness(distorted_image,  
                                             max_delta=63)  
distorted_image = tf.image.random_contrast(distorted_image,  
                                         lower=0.2, upper=1.0)  
  
# Subtract off the mean and divide by the variance of the  
float_image = tf.image.per_image_standardization(distorted_image)
```

上述代码中分别调用了不同的函数对图片进行不同的变换，具体解释如下。

- `tf.random_crop`: 为图片随机裁剪。
- `tf.image.random_flip_left_right`: 随机左右翻转。
- `tf.image.random_brightness`: 随机亮度变化。
- `tf.image.random_contrast`: 随机对比度变

化。

- `tf.image.per_image_standardization`: 减去均值像素，并除以像素方差（图片标准化）。



注意：这些函数都是增加数据的好方法，读者可以积累起来，在自己的训练样本中使用。

8.5.6 在TensorFlow中使用queue

TensorFlow提供了一个队列机制，通过多线程将读取数据与计算数据分开。因为在处理海量数据集的训练时，无法把数据集一次全部载入到内存中，需要一边从硬盘中读取，一边进行训练计算。

对于建立队列读取文件部分的代码，已经在 `cifar10_input.py` 里实现了。因为这部分不是本书的重点，所以不做太多介绍，有兴趣的读者可以看下 `cifar10_input.py` 里面的源码。

在这里主要讲解内部机制及如何使用，这里分为以下3个知识点。

1. 队列线程启动及挂起机制

还记得8.5.4节中的例子代码（“8-5 CIFAR.py”），在session里面有这么一句：

```
tf.train.start_queue_runners()
```

可以试着将其注释掉，然后运行一下看下效果——程序不动了，这时处于一个挂起状态，`start_queue_runners`的作用是启动线程，向队列里面读数据。那么为什么会挂起呢？源于下面的这句代码：

```
image_batch, label_batch = sess.run([images_test, labels_test])
```

这句话的意思是从队列里拿出指定批次的数据。但是队列里没有数据，所以程序进入挂起等待状态。

2. 在session内部的退出机制

接下来可以把代码“8-5 CIFAR.py”文件中的session部分改成with语法，如下：

```
with tf.Session() as sess:  
    tf.global_variables_initializer().run()  
    tf.train.start_queue_runners()  
    image_batch, label_batch = sess.run([images_test, labels_test])  
    print("__\n", image_batch[0])  
  
    print("__\n", label_batch[0])  
    pylab.imshow(image_batch[0])
```

*****ebook converter DEMO Watermarks*****

```
pylab.show()
```

再次运行程序，发现虽然程序能够正常运行，但是结束后会报错，输出如下信息：

```
ERROR:tensorflow:Exception in QueueRunner: Run call was cancel
ERROR:tensorflow:Exception in QueueRunner: Run call was cancel
ERROR:tensorflow:Exception in QueueRunner: Run call was cancel
ERROR:tensorflow:Exception in QueueRunner: Session has been closed
ERROR:tensorflow:Exception in QueueRunner: Run call was cancel
ERROR:tensorflow:Exception in QueueRunner: Run call was cancel
ERROR:tensorflow:Exception in QueueRunner: Enqueue operation
....
```

原因就是带with语法的session是自动关闭的。当运行结束后session自动关闭的同时会把里面所有的操作都关掉，而此时的队列还在等待另一个进程往里写数据，所以就会出现错误。最简单的解决方法就是如代码“8-5 CIFAR.py”文件中的session创建方式，使用sess = tf.InteractiveSession来实现。或者，也可以在原来代码中去掉with语句（将上面代码的第1行改后下面代码的第1行），但后面的操作都要指定属于哪个session（将上一段代码的第1~3行改后下面代码的第1~3行）。改完之后的代码如下：

```
sess = tf.Session()
tf.global_variables_initializer().run(session=sess)
tf.train.start_queue_runners(sess=sess)
image_batch, label_batch = sess.run([images_test, labels_test])
print("__\n", image_batch[0])

print("__\n", label_batch[0])
*****ebook converter DEMO Watermarks*****
```

```
pylab.imshow(image_batch[0])
pylab.show()
```

上面的代码在单例程序中没什么问题，资源会随着程序关闭而整体销毁。但如果在复杂的代码中，需要某个线程自动关闭，而不是依赖进程的结束而销毁，这种情况下需要使用tf.train.Coordinator函数来创建一个协调器，以信号量的方式来协调线程间的关系，完成线程间的同步。

8.5.7 实例42：协调器的用法演示

下面来看一下协调器的用法。

在本例子中，先建立一个100大小的队列。主线程使用计数器不停地加1，队列线程再把主线程里的计数器放到队列里。当队列为空时，主线程在sess.run(queue.dequeue())语句位置挂起，当队列线程写入对列中时，主线程的计数器同步开始工作。整个操作都是在使用with语法的session中进行的，由于使用了Coordinator，当session要关闭之前会进行coord.request_stop函数将所有线程关闭，之后才会关闭session。

代码8-7 queue

```
import tensorflow as tf
```

*****ebook converter DEMO Watermarks*****

```
#创建长度为100的队列
queue = tf.FIFOQueue(100, "float")

c = tf.Variable(0.0)                      #计数器
#加1操作
op = tf.assign_add(c, tf.constant(1.0))
#操作:将计数器的结果加入队列
enqueue_op = queue.enqueue(c)

#创建一个队列管理器QueueRunner，用这两个操作向q中添加元素。目前我们只
qr = tf.train.QueueRunner(queue, enqueue_ops=[op, enqueue_op])

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())

    coord = tf.train.Coordinator()

    # 启动入队线程，Coordinator是线程的参数
    enqueue_threads = qr.create_threads(sess, coord = coord,
                                         # 启动入队线程

                                         # 主线程
    for i in range(0, 10):
        print ("-----")
        print(sess.run(queue.dequeue()))

coord.request_stop() #通知其他线程关闭 其他所有线程关闭之后,
```

运行以上代码，输出如下信息：（可以看到
并没有报错）

```
3.0
-----
405.0
-----
410.0
-----
413.0
-----
420.0
-----
475.0
```

*****ebook converter DEMO Watermarks*****

```
478.0
```

```
-----  
896.0  
-----
```

```
902.0  
-----
```

```
904.0
```

这里还可以使用
coord.join (enqueue_threads) 指定等待某个进程
结束。

8.5.8 实例43：为session中的队列加上协调器

这里将上例中的coord放到启动队列里即可。

实例描述

在with tf.Session函数中加入启动队列，并通
过加入coord协调器的方式使session close时同步内
部线程一起退出。

修改“8-5cifar”代码如下。

代码8-8 cifar队列协调器（部分代码）

```
with tf.Session() as sess:  
    tf.global_variables_initializer().run()  
    #定义协调器  
    coord = tf.train.Coordinator()  
    threads = tf.train.start_queue_runners(sess, coord)
```

```
image_batch, label_batch = sess.run([images_test, labels])
print("\n", image_batch[0])

print("\n", label_batch[0])
pylab.imshow(image_batch[0])
pylab.show()
coord.request_stop()
```

8.5.9 实例44：建立一个带有全局平均池化层的卷积神经网络

现在正式开始卷积神经网络的示例。在本示例中，使用了全局平均池化层来代替传统的全连接层，使用了3个卷积层的同卷积操作，滤波器为 5×5 ，每个卷积层后面都会跟个步长为 2×2 的池化层，滤波器为 2×2 。2层的卷积加池化后是输出为10个通道的卷积层，然后对这10个feature map进行全局平均池化，得到10个特征，再对这10个特征进行softmax计算，其结果来代表最终分类。

实例描述

通过一个带有全局平局池化层的卷积神经网络对CIFAR数据集分类。

具体步骤如下：

1. 导入头文件引入数据集

这步骤与前面的代码相似，还是使用cifar10_input里面的代码，导入这种被切割后的*****ebook converter DEMO Watermarks*****

24×24尺寸图片。每次取128个图片进行运算。
在“cifar10”文件夹下建立“8-9cifar卷积.py”文件，
编写如下代码。

代码8-9 cifar卷积

```
01 import cifar10_input  
02 import tensorflow as tf  
03 import numpy as np  
04  
05 batch_size = 128  
06 data_dir = '/tmp/cifar10_data/cifar-10-batches-bin'  
07 print("begin")  
08 images_train, labels_train = cifar10_input.inputs(eval_data=False,  
data_dir = data_dir, batch_size = batch_size)  
09 images_test, labels_test = cifar10_input.inputs(eval_data=True,  
dir = data_dir, batch_size = batch_size)  
10 print("begin data")
```

2. 定义网络结构

对于权重w的定义，统一使用函数truncated_normal来生成标准差为0.1的随机数为其初始化。对于权重b的定义，统一初始化为0.1。

卷积操作的函数中，统一进行同卷积操作，即步长为1，padding='SAME'。

池化层有两个函数：

- 一个是放在卷积后面，取最大值的方法，步长为2，padding='SAME'，即将原尺寸的长和宽各除以2。

· 另一个是用来放在最后一层，取均值的方法，步长为最终生成的特征尺寸 6×6 (24×24 经过两次池化变成了 6×6)，filter也为 6×6 。

倒数第二层是没有最大池化的卷积层，因为共有10类，所以卷积输出的是10个通道，并使其全局平均池化为10个节点。

代码8-9 cifar卷积（续）

```
11 def weight_variable(shape):
12     initial = tf.truncated_normal(shape, stddev=0.1)
13     return tf.Variable(initial)
14
15 def bias_variable(shape):
16     initial = tf.constant(0.1, shape=shape)
17     return tf.Variable(initial)
18
19 def conv2d(x, w):
20     return tf.nn.conv2d(x, w, strides=[1, 1, 1, 1], padding='SAME')
21
22 def max_pool_2x2(x):
23     return tf.nn.max_pool(x, ksize=[1, 2, 2, 1],
24                           strides=[1, 2, 2, 1], padding='SAME')
25
26 def avg_pool_6x6(x):
27     return tf.nn.avg_pool(x, ksize=[1, 6, 6, 1],
28                           strides=[1, 6, 6, 1], padding='SAME')
29
30 # 定义占位符
31 x = tf.placeholder(tf.float32, [None, 24, 24, 3]) # cifar
32 y = tf.placeholder(tf.float32, [None, 10]) # 0~9 数字分类
33
34 W_conv1 = weight_variable([5, 5, 3, 64])
35 b_conv1 = bias_variable([64])
36
37 x_image = tf.reshape(x, [-1, 24, 24, 3])
38
39 h_conv1 = tf.nn.relu(conv2d(x_image, W_conv1) + b_conv1)
40 h_pool1 = max_pool_2x2(h_conv1)
41
```

```
42 W_conv2 = weight_variable([5, 5, 64, 64])
43 b_conv2 = bias_variable([64])
44
45 h_conv2 = tf.nn.relu(conv2d(h_pool1, W_conv2) + b_conv2)
46 h_pool2 = max_pool_2x2(h_conv2)
47
48 W_conv3 = weight_variable([5, 5, 64, 10])
49 b_conv3 = bias_variable([10])
50 h_conv3 = tf.nn.relu(conv2d(h_pool2, W_conv3) + b_conv3)
51
52 nt_hpool3=avg_pool_6x6(h_conv3)#10
53 nt_hpool3_flat = tf.reshape(nt_hpool3, [-1, 10])
54 y_conv=tf.nn.softmax(nt_hpool3_flat)
55
56 cross_entropy = -tf.reduce_sum(y*tf.log(y_conv))
57
58 train_step = tf.train.AdamOptimizer(1e-4).minimize(cross_
59
60 correct_prediction = tf.equal(tf.argmax(y_conv,1), tf.arg_
61 accuracy = tf.reduce_mean(tf.cast(correct_prediction, "f_
```

对于梯度优化算法，还是和多分类问题一样，我们使用AdamOptimizer函数，学习率使用0.0001。

3. 运行session进行训练

启动session，迭代15000次数据集，这里记着要启动队列，同时读出来的label还要转成onehot编码。

代码8-9 cifar卷积（续）

```
62 sess = tf.Session()
63 sess.run(tf.global_variables_initializer())
64 tf.train.start_queue_runners(sess=sess)
65 for i in range(15000):#20000
66     image_batch, label_batch = sess.run([images_train, lab_
```

*****ebook converter DEMO Watermarks*****

```
67     label_b = np.eye(10,dtype=float)[label_batch] #one hot
68
69     train_step.run(feed_dict={x:image_batch, y: label_b}, session=sess)
70
71     if i%200 == 0:
72         train_accuracy = accuracy.eval(feed_dict={
73             x:image_batch, y: label_b}, session=sess)
74         print("step %d, training accuracy %g"%(i, train_accuracy))
75
```

4. 评估结果

从测试数据集里将数据取出，放到模型里运行，查看模型的正确率。

代码8-9 cifar卷积（续）

```
76 image_batch, label_batch = sess.run([images_test, labels_test])
77 label_b = np.eye(10,dtype=float)[label_batch] #one hot
78 print ("finished! test accuracy %g"%accuracy.eval(feed_dict={
79             x:image_batch, y: label_b}, session=sess))
```

运行代码后，输出如下：

```
#begin
#begin data
#step 0, training accuracy 0.15625
#step 200, training accuracy 0.3125
#step 400, training accuracy 0.359375
#step 600, training accuracy 0.3125
#step 800, training accuracy 0.382812
#step 1000, training accuracy 0.273438
.....
#step 14400, training accuracy 0.554688
#step 14600, training accuracy 0.601562
#step 14800, training accuracy 0.5625
#finished! test accuracy 0.632812
```

可以看出，识别效果得到了收敛，正确率在0.6左右，这个正确率不算很高，因为模型相对简单，只是用了两层的卷积操作。接下来还将介绍更多的优化方法来提升准确率。



注意：例子中对于卷积和池化的使用也表明了一种习惯，一般在卷积过程中都会设为步长为1的same卷积，即大小不变，需要降维时则是全部通过池化来完成的。

8.5.10 练习题

(1) 使用前面所学的知识，试着将MNIST图片集进行分类（见代码“8-10 MNIST卷积.py”文件）。

```
01 def max_pool_with_argmax(net, stride):
02     _, mask = tf.nn.max_pool_with_argmax( net, ksize=[1, :
03                                         1], strides=[1, stride, stride, 1], padding='SAME')
04     mask = tf.stop_gradient(mask)
05     net = tf.nn.max_pool(net, ksize=[1, stride, stride, :
06                           stride, stride, 1], padding='SAME')
07     return net, mask
```

(2) 将CIFAR卷积分类的例子中最后一层改成全连接网络，试试看会有什么效果（见代码“8-11 Cifar全连接卷积.py”）

8.6 反卷积神经网络

反卷积是指，通过测量输出和已知输入重构未知输入的过程。在神经网络中，反卷积过程并不具备学习的能力，仅仅是用于可视化一个已经训练好的卷积网络模型，没有学习训练的过程。

如图8-18所示为VGG 16反卷积神经网络的结构，展示了一个卷积网络与反卷积网络结合的过程。VGG 16是一个深度神经网络模型，在后面会专门介绍。它的反卷积就是将中间的数据，按照前面卷积、池化等变化的过程，完全相反地做一遍，从而得到类似原始输入的数据。

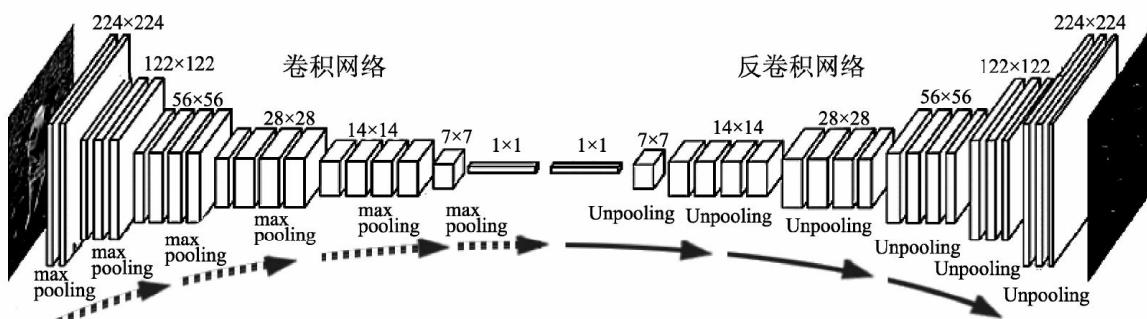


图8-18 VGG 16反卷积结构

8.6.1 反卷积神经网络的应用场景

由于反卷积网络的特性，导致它有许多特别的应用，一般可以用于信道均衡、图像恢复、语音识别、地震学、无损探伤等未知输入估计和过

程辨识方面的问题。

在神经网络的研究中，反卷积更多的是充当可视化的作用。对于一个复杂的深度卷积网络，通过每层若干个卷积核的变换，我们无法知道每个卷积核关注的是什么，变换后的特征是什么样子。通过反卷积的还原，可以对这些问题有个清晰的可视化，以各层得到的特征图作为输入，进行反卷积，得到反卷积结果，用以验证显示各层提取到的特征图。

8.6.2 反卷积原理

反卷积，可以理解为卷积操作的逆操作。这里千万不要当成反卷积操作可以复原卷积操作的输入值，反卷积并没有那个功能，它仅仅是将卷积变换过程中的步骤反向变换一次而已，通过将卷积核转置，与卷积后的结果再做一遍卷积，所以它还有个名字叫是转置卷积。

虽然它不能还原出原来卷积的样子，但是在作用上具有类似的效果，可以将带有小部分缺失的信息最大化地恢复，也可以用来恢复被卷积生成后的原始输入。

反卷积的具体操作比较复杂，具体步骤如下。

(1) 首先是将卷积核反转（并不是转置，而是上下左右方向进行递序操作）。

(2) 再将卷积结果作为输入，做补0的扩充操作，即往每一个元素后面补0。这一步是根据步长来的，对每一个元素沿着步长的方向补（步长-1）个0。例如，步长为1就不用补0了。

(3) 在扩充后的输入基础上再对整体补0。以原始输入的shape作为输出，按照前面介绍的卷积padding规则，计算padding的补0位置及个数，得到的补0位置要上下和左右各自颠倒一下。

(4) 将补0后的卷积结果作为真正的输入，反转后的卷积核为filter，进行步长为1的卷积操作。



注意： 计算padding按规则补0时，统一按照padding='SAME'、步长为 1×1 的方式来计算。

如图8-19所示，以一个[1, 4, 4, 1]的矩阵为例，进行filter为 2×2 ，步长为 2×2 的卷积操作（如图8-19a所示），其对应的反卷积操作步骤如图8-19b所示。

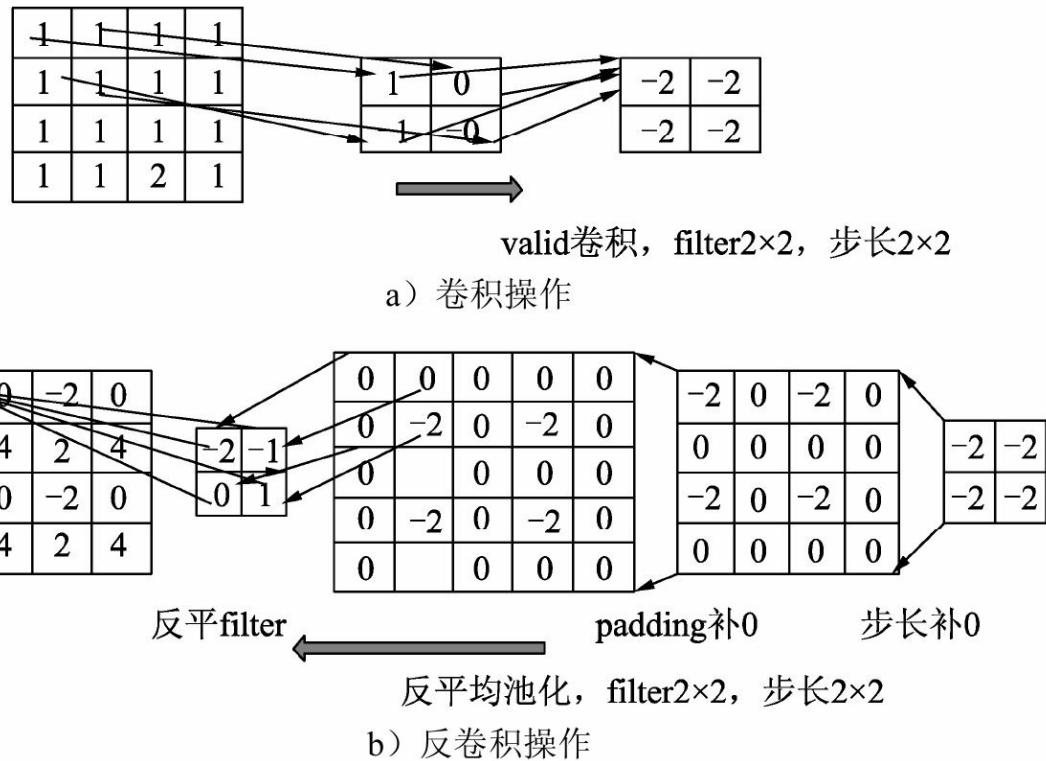


图8-19 卷积与反卷积操作

在反卷积过程中，首先将 2×2 矩阵通过步长补0的方式变成 4×4 ，再通过padding反方向补0，然后与反转后的filter使用步长为 1×1 的卷积操作，最终得出了结果。但是这个结果已经与原来的全1矩阵不等了，说明转置卷积只能恢复部分特征，无法百分百地恢复原始数据。

8.6.3 实例45：演示反卷积的操作

在编写反卷积代码时，心中想着一个正向的卷积过程会很有帮助。在TensorFlow中反卷积是通过函数tf.nn.conv2d_transpose来实现的，其定义如下：

```
def conv2d_transpose(value,
                      filter,
                      output_shape,
                      strides,
                      padding="SAME",
                      data_format="NHWC",
                      name=None):
```

具体参数说明如下。

- value: 代表通过卷积操作之后的张量，一般用NHWC类型。
- filter: 代表卷积核。
- output_shape: 代表输出的张量形状也是个四维张量。
- strides: 代表步长。
- padding: 代表原数据生成value时使用的方式，是用来检查输入形状和输出形状是否合规的。
- return: 反卷积后的结果，按照output_shape指定的形状。



注意： NHWC类型是神经网络中在处理图像方面常用的类型，4个字母分别代表4个意思，即N-个数、H-高、W-宽、C-通道数。也就是

我们常见的四维张量。



注意： `output_shape`并不是一个随便填写的形状，它必须是能够生成`value`参数的原数据的形状，如果输出形状不对，函数会报错。

跟进TensorFlow的源码中可以看到，反卷积操作其实是使用了`gen_nn_ops.conv2d_backprop_input`函数来最终实现的，相当于TensorFlow中利用了卷积操作在反向传播的处理函数中做反卷积操作，即卷积操作的反向传播就是反卷积操作。

下面通过例子将前面图示的数据演示出来，并且比较一下SAME和VALID下对应卷积和反卷积的影响。

实例描述

通过对模拟数据进行卷积与反卷积的操作，来比较卷积与反卷积中`padding`在SAME、VALID下的变化。

代码8-12 反卷积操作

```
import numpy as np
import tensorflow as tf
#模拟数据
img = tf.Variable(tf.constant(1.0, shape = [1, 4, 4, 1]))
```

*****ebook converter DEMO Watermarks*****

```
filter = tf.Variable(tf.constant([1.0, 0, -1, -2], shape = [2,
#分别进行VALID与SAME操作
conv = tf.nn.conv2d(img, filter, strides=[1, 2, 2, 1], padd:
cons = tf.nn.conv2d(img, filter, strides=[1, 2, 2, 1], padd:
print(conv.shape)
print(cons.shape)
#再进行反卷积
contv= tf.nn.conv2d_transpose(conv, filter, [1,4,4,1],strides
1], padding='VALID')
conts = tf.nn.conv2d_transpose(cons, filter, [1,4,4,1],strides
1], padding='SAME')

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer() )

    print("conv:\n",sess.run([conv,filter]))
    print("cons:\n",sess.run([cons]))
    print("contv:\n",sess.run([contv]))
    print("conts:\n",sess.run([conts]))
```

先定义一个[1, 4, 4, 1]的矩阵，矩阵里的值全为1，进行filter为 2×2 、步长为 2×2 的卷积操作，分别使用padding为SAME和VALID的两种情况生成卷积数据，然后将结果放到tf.nn.conv2d_transpose里，再次使用padding为SAME和VALID的两种情况生成数据，运行上面代码，输出如下：

```
(1, 2, 2, 1)
(1, 2, 2, 1)
conv:
[array([[[[-2.],
           [-2.]],
          [[-2.],
           [-2.]]]], dtype=float32), array([[[[ 1.]],
          [[[ 0.]]],
          [[[ -1.]],
           [[-2.]]]], dtype=float32)]
cons:
```

*****ebook converter DEMO Watermarks*****

```
[array([[[[-2.],
          [-2.]],
         [[-2.],
          [-2.]]]], dtype=float32)]
contv:
[array([[[[-2.],
          [ 0.],
          [-2.],
          [ 0.]],
         [[ 2.],
          [ 4.],
          [ 2.],
          [ 4.]],
         [[-2.],
          [ 0.],
          [-2.],
          [ 0.]],
         [[ 2.],
          [ 4.],
          [ 2.],
          [ 4.]]]], dtype=float32)]
conts:
[array([[[[-2.],
          [ 0.],
          [-2.],
          [ 0.]],
         [[ 2.],
          [ 4.],
          [ 2.],
          [ 4.]],
         [[-2.],
          [ 0.],
          [-2.],
          [ 0.]],
         [[ 2.],
          [ 4.],
          [ 2.],
          [ 4.]]]], dtype=float32)]
```

可以看到，输出的结果与图8-19是一样的，并且也验证了当padding为SAME并且不需要补0时，卷积和反卷积对于padding是SAME和VALID都是相同的。

8.6.4 反池化原理

反池化是属于池化的逆操作，是无法通过池化的结果还原出全部的原始数据。因为池化的过程就是只保留主要信息，舍去部分信息。如想从池化后的这些主要信息恢复出全部信息，则存在着信息缺失，这时只能通过补位来实现最大程度的信息完整。

池化有两种最大池化和平均池化，其反池化也需要与其对应。

· 平均池化的操作比较简单。首先还原成原来的大小，然后将池化结果中的每个值都填入其对应于原始数据区域中的相应位置即可，如图8-20所示。

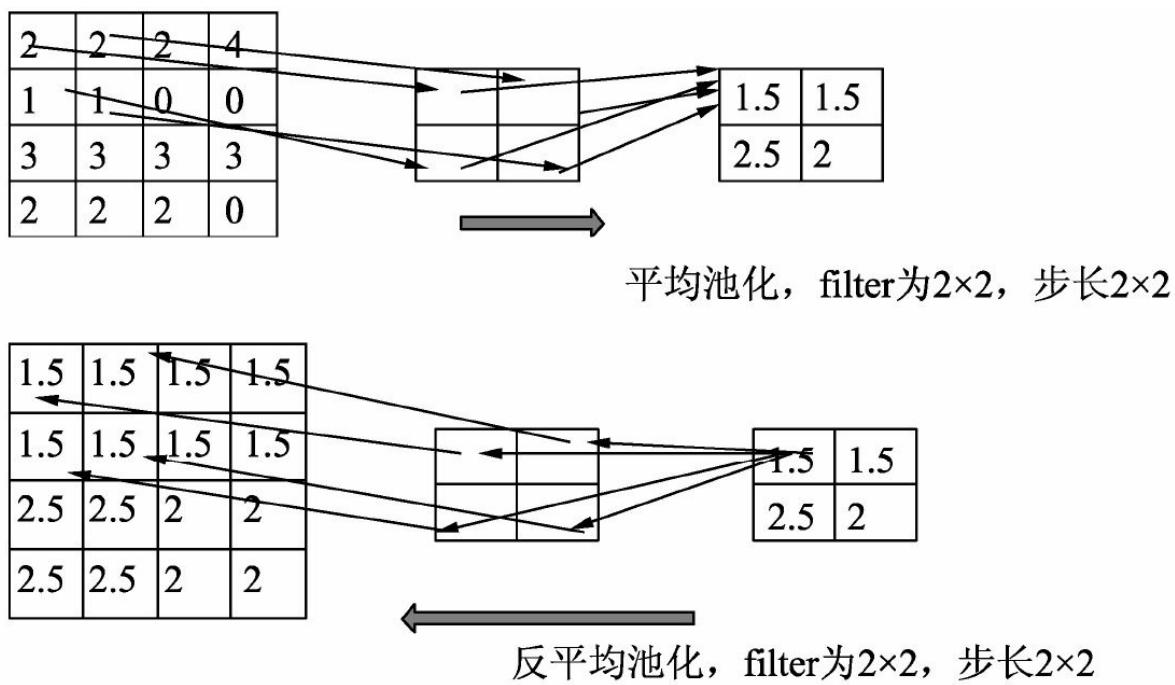


图8-20 反平均池化

· 最大池化的反池化会复杂一些。要求在池化过程中记录最大激活值的坐标位置，然后在反池化时，只把池化过程中最大激活值所在位置坐标的值激活，其他的值置为0。当然，这个过程只是一种近似。因为在池化的过程中，除了最大值所在的位置，其他的值也是不为0的。如图8-21所示。

Input	Max-pooling [3,3]	Unpooling																																								
<table border="1"><tr><td>0.8</td><td>0</td><td>3.3</td><td>0</td><td>4</td><td>1</td></tr><tr><td>1.2</td><td>1.2</td><td>0</td><td>2.1</td><td>0</td><td>0</td></tr><tr><td>0</td><td>3</td><td>0.2</td><td>3.1</td><td>5.2</td><td>1</td></tr></table>	0.8	0	3.3	0	4	1	1.2	1.2	0	2.1	0	0	0	3	0.2	3.1	5.2	1	→ Value: <table border="1"><tr><td>3.3</td><td>5.2</td></tr></table> Position: <table border="1"><tr><td>(0,2)</td><td>(2,1)</td></tr></table>	3.3	5.2	(0,2)	(2,1)	→ <table border="1"><tr><td>0</td><td>0</td><td>3.3</td><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>5.2</td><td>0</td></tr></table>	0	0	3.3	0	0	0	0	0	0	0	0	0	0	0	0	0	5.2	0
0.8	0	3.3	0	4	1																																					
1.2	1.2	0	2.1	0	0																																					
0	3	0.2	3.1	5.2	1																																					
3.3	5.2																																									
(0,2)	(2,1)																																									
0	0	3.3	0	0	0																																					
0	0	0	0	0	0																																					
0	0	0	0	5.2	0																																					

图8-21 反最大池化

8.6.5 实例46：演示反池化的操作

TensorFlow中目前还没有反池化操作的函数。对于最大池化，也不支持输出最大激活值的位置，但是同样有个池化的反向传播函数tf.nn.max_pool_with_argmax。该函数可以输出位置，需要开发者利用这个函数做一些改动，自己封装一个最大池化操作，然后再根据mask写出反池化函数，下面以反最大池化为例。

实例描述

定义一个数组作为模拟图片，将其进行最大池化，接着再进行反池化，比较原始数据与反池化后的数据。

首先重新定义最大池化函数，代码如下。

代码8-13 反池化操作

```
01 def max_pool_with_argmax(net, stride):
02     _, mask = tf.nn.max_pool_with_argmax( net, ksize=[1, 1],
03                                         strides=[1, stride, stride, 1], padding='SAME')
04     mask = tf.stop_gradient(mask)
05     net = tf.nn.max_pool(net, ksize=[1, stride, stride, 1],
06                          strides, stride, 1], padding='SAME')
07     return net, mask
```

在上面代码里，先调用 `tf.nn.max_pool_with_argmax` 函数获得每个最大值的位置 `mask`，再将反向传播的 `mask` 梯度计算停止（后面会有关于梯度停止的介绍），接着再用 `tf.nn.max_pool` 函数计算最大池化操作，然后将 `mask` 和池化结果一起返回。



注意： `tf.nn.max_pool_with_argmax` 的方法只支持 GPU 操作，所以利用这个方法目前还不能在 CPU 机器上使用。

接下来定义一个数组，并使用最大池化函数对其进行池化操作，比较一下与自带的

tf.nn.max_pool函数是否一样，看看输出的mask是什么效果。

代码8-13 反池化操作（续）

```
06 img=tf.constant([
07     [[0.0,4.0],[0.0,4.0],[0.0,4.0],[0.0,4.0]],
08     [[1.0,5.0],[1.0,5.0],[1.0,5.0],[1.0,5.0]],
09     [[2.0,6.0],[2.0,6.0],[2.0,6.0],[2.0,6.0]],
10     [[3.0,7.0],[3.0,7.0],[3.0,7.0],[3.0,7.0]]
11 ])
12
13 img=tf.reshape(img,[1,4,4,2])
14 pooling2=tf.nn.max_pool(img,[1,2,2,1],[1,2,2,1],padding=
15 encode, mask = max_pool_with_argmax(img, 2)
16 with tf.Session() as sess:
17     print("image:")
18     image=sess.run(img)
19     print (image)
20     result=sess.run(pooling2)
21     print ("pooling2:\n",result)
22     result,mask2=sess.run([encode, mask])
23     print ("encode:\n",result,mask2)
```

代码运行后，输出如下：

```
image:
[[[[ 0.  4.]
   [ 0.  4.]
   [ 0.  4.]
   [ 0.  4.]]

[[ 1.  5.]
   [ 1.  5.]
   [ 1.  5.]
   [ 1.  5.]]

[[ 2.  6.]
   [ 2.  6.]
   [ 2.  6.]]]
```

```
[ 2.  6.]]  
[[[ 3.  7.]  
[ 3.  7.]  
[ 3.  7.]  
[ 3.  7.]]]]  
pooling2:  
[[[[ 1.  5.]  
[ 1.  5.]]]  
[[[ 3.  7.]  
[ 3.  7.]]]]  
encode:  
[[[[ 1.  5.]  
[ 1.  5.]]]  
[[[ 3.  7.]  
[ 3.  7.]]]] [[[ 8  9]  
[12 13]]]  
[[[24 25]  
[28 29]]]]
```

可以看到，定义的最大池化与原来的版本输出是一样的。mask的值是将整个数组flat（扁平化）后的索引，但却保持与池化结果一致的shape。

了解这些信息后，就可以接着写代码，定义一个反最大池化的操作了。

代码8-13 反池化操作（续）

```
24 def unpool(net, mask, stride):  
25  
26     ksize = [1, stride, stride, 1]  
27     input_shape = net.get_shape().as_list()  
28     # 计算new shape  
29     output_shape = (input_shape[0], input_shape[1] * ksize[1], ksize[2], ksize[3])  
30  
31     indices = tf.reshape(tf.where(tf.equal(mask, 1)), [-1, 4])  
32     values = tf.reshape(net, [-1])  
33  
34     new_shape = tf.stack([output_shape[0], output_shape[1],  
35                          output_shape[2], output_shape[3]])  
36  
37     return tf.scatter_nd(indices, values, new_shape)
```

*****ebook converter DEMO Watermarks*****

```
shape[2] * kszie[2], input_shape[3])
30     # 计算索引
31     one_like_mask = tf.ones_like(mask)
32     batch_range = tf.reshape(tf.range(output_shape[0], dtype=tf.int64), shape=[input_shape[0], 1, 1, 1])
33     b = one_like_mask * batch_range
34     y = mask // (output_shape[2] * output_shape[3])
35     x = mask % (output_shape[2] * output_shape[3]) // output_shape[3]
36     feature_range = tf.range(output_shape[3], dtype=tf.int64)
37     f = one_like_mask * feature_range
38     # 转置索引
39     updates_size = tf.size(net)
40     indices = tf.transpose(tf.reshape(tf.stack([b, y, x, updates_size])))
41     values = tf.reshape(net, [updates_size])
42     ret = tf.scatter_nd(indices, values, output_shape)
43     return ret
```

上面代码的大概思路是找到mask对应的索引，将max的值填到指定地方。这里不做过多解释，读者可以将反向传播函数当成一个工具，以后直接使用即可。

下面调用反池化函数，并将结果打印出来。

代码8-13 反池化操作（续）

```
44 img2 = unpool(encode, mask, 2)
45 with tf.Session() as sess:
46     .....
47     print ("encode:\n", result, mask2)
48     result=sess.run(img2)
49     print ("reslut:\n", result)
```

代码运行后，输出结果如下：

```
reslut:  
[[[[ 0.  0.  
[ 0.  0.  
[ 0.  0.  
[ 0.  0.]  
  
[[ 1.  5.  
[ 0.  0.  
[ 1.  5.  
[ 0.  0.]  
  
[[ 0.  0.  
[ 0.  0.  
[ 0.  0.  
[ 0.  0.]  
  
[[ 3.  7.  
[ 0.  0.  
[ 3.  7.  
[ 0.  0.]]]]
```

可以看到，最大值已经填入对应的位置，其他地方的值为0。

8.6.6 实例47：演示gradients基本用法

这部分内容本来是要放在前面章节讲的，考虑到读者直接了解梯度相关的知识有些生硬，而且一时也用不上，所以就将这部分内容移到了本节中，这样，通过例子中引出的知识点，会使读者学习起来更加通顺。

实例描述

通过定义两个矩阵变量相乘来演示使用gradients求梯度。

*****ebook converter DEMO Watermarks*****

在反向传播过程中，神经网络需要对每一个 loss 对应的学习参数求偏导，算出的这个值也叫梯度，用来乘以学习率然后更新学习参数使用的。它是通过 tf.gradients 函数来实现的。

tf.gradients 函数里第一个参数为求导公式的结 果，第二个参数为指定公式中的哪个变量来求偏 导。下面通过例子介绍 tf.gradients 函数的用法。

代码8-14 gradients0

```
import tensorflow as tf

w1 = tf.Variable([[1, 2]])
w2 = tf.Variable([[3, 4]])

y = tf.matmul(w1, [[9], [10]])
grads = tf.gradients(y, [w1])          #求w1的梯度

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    gradval = sess.run(grads)
    print(gradval)
```

运行后输出结果如下：

```
[array([[ 9, 10]])]
```

上面例子中，由于 y 是由 $w1$ 与 $[[9], [10]]$ 相乘 而来，所以其导数也就是 $[[9], [10]]$ （即斜率）。



注意： 如果求梯度的式子中没有要求偏

*****ebook converter DEMO Watermarks*****

导的变量，系统会报错。例如，写成`grads = tf.gradients(y, [w1, w2])`。

8.6.7 实例48：使用gradients对多个式子求多变量偏导

`tf.gradients`函数还可以同时对多个式子求关于多个变量的偏导，见如下代码

实例描述

有两个OP，4个参数，演示使用`gradients`同时为两个式子4个参数求梯度。

代码8-15 gradients1

```
import tensorflow as tf

tf.reset_default_graph()
w1 = tf.get_variable('w1', shape=[2])
w2 = tf.get_variable('w2', shape=[2])

w3 = tf.get_variable('w3', shape=[2])
w4 = tf.get_variable('w4', shape=[2])

y1 = w1 + w2+ w3
y2 = w3 + w4
# grad_ys求梯度的输入值
gradients = tf.gradients([y1,y2], [w1,w2,w3,w4], grad_ys=[tf.
tensor([1.,2.]),tf.convert_to_tensor([3.,4.])])

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    print(sess.run(gradients))
```

运行代码，输出结果如下：

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

这里使用了tf.gradients函数的第三个参数，即给定公式结果的值，来求参数偏导，这里相当于y1为[1., 2.]、y2为[3., 4.]。对于y1来讲，求关于w1的偏导时，会认为w2和w3为常数，所以w1和w2的导数为0，即w1的梯度就为[1., 2.]。同理可以得出w2和w3均为[1., 2.]，接着求y2的偏导数，得到w3与w4均为[3., 4.]。然后将两个式子中的w3结果累加起来，所以w3就为[4., 6.]。

8.6.8 实例49：演示梯度停止的实现

实例描述

演示梯度停止的用法，并观察当变量设置梯度停止后，对其求梯度的结果。

对于反向传播过程中某种特殊情况需要停止梯度的运算时，在TensorFlow中提供了一个tf.stop_gradient函数，被它定义过的节点将没有梯度运算功能。

例如，在前面代码中加入y3结点。通过gradients2来计算其相关变量的梯度。

代码8-16 gradients2

```
01 .....
02 a = w1+w2
03 a_stoped = tf.stop_gradient(a)          #令a梯度停止
04 y3= a_stoped+w3
05
06 gradients = tf.gradients([y1,y2],[w1,w2,w3,w4], grad_ys=
to_tensor([1.,2.]),
07     tf.convert_to_tensor([3.,4.]))
08
09 gradients2 = tf.gradients(y3,[w1,w2,w3], grad_ys=tf.conve
([1.,2.]))
10 print(gradients2)
11
12 with tf.Session() as sess:
13     sess.run(tf.global_variables_initializer())
14     print(sess.run(gradients))
15     print(sess.run(gradients2))
```

运行代码，结果如下：

```
[None, None, <tf.Tensor 'gradients_1/add_4_grad/Reshape_1:0'
dtype=float32>]
Traceback (most recent call last):
....
```

可以看到程序运行出错，并且在出错前显示了gradients2的内容，w1和w2对应的位置都为None，这是由于梯度被停止了。后面的程序试图去求一个None的梯度，所以报错了。

再定义一个gradients3，只求存在的梯度，同时将print (sess.run (gradients2))注释掉，代码

如下。

代码8-16 gradients2（续）

```
16 gradients3 = tf.gradients(y3, [ w3], grad_ys=tf.convert_to_tensor([1.,2.]))
17
18 with tf.Session() as sess:
19     sess.run(tf.global_variables_initializer())
20     print(sess.run(gradients))
21     #print(sess.run(gradients2))    #程序试图去求一个None的梯
22     print(sess.run(gradients3))
```

运行代码，输出结果如下：

```
[None, None, <tf.Tensor 'gradients_1/add_4_grad/Reshape_1:0' type=float32>]
[array([ 1.,  2.], dtype=float32), array([ 1.,  2.], dtype=float32),
 array([ 4.,  6.], dtype=float32), array([ 3.,  4.], dtype=float32),
 [array([ 1.,  2.], dtype=float32)]]
```

这时就可以正常运算了。

8.7 实例50：用反卷积技术复原卷积网络各层图像

在了解了反卷积神经网络之后，下面通过一个例子将前面的卷积神经网络里的卷积层可视化出来，看看每一层到底学到了什么信息。

实例描述

将代码“8-9 cifar卷积.py”文件中的每层卷积结果进行反卷积并输出，通过tensorboard观察其结果。

改写代码代码“8-9 cifar卷积.py”，将每层的卷积内容可视化并在tensroboard中查看，具体步骤如下。

1. 替换Maxpool池化函数

这里不再使用自己定义的max_pool_2x2池化函数，改成新加入的带mask返回值的max_pool_with_argmax函数，具体代码如下。

代码8-17 cifar反卷积

```
01 ....  
02 x_image = tf.reshape(x, [-1, 24, 24, 3])  
03
```

```
04 h_conv1 = tf.nn.relu(conv2d(x_image, w_conv1) + b_conv1)
05 h_pool1, mask1 = max_pool_with_argmax(h_conv1, 2)
06
07 w_conv2 = weight_variable([5, 5, 64, 64])
08 b_conv2 = bias_variable([64])
09
10 h_conv2 = tf.nn.relu(conv2d(h_pool1, w_conv2) + b_conv2)
11 #####
12 h_pool2, mask = max_pool_with_argmax(h_conv2, 2) #(128, 6,
13 print(h_pool2.shape)
```



注意： 上面代码的最后一行是将 `h_pool2` 的形状打印出来，这也是在组建网络结构时常用的一种调试方法。反卷积和反池化对形状都很敏感（尤其层数太多时），这种方法可以让我们不用花费太多精力来推导到底当前的输入是什么形状。

2. 反卷积第二层卷积结果

以第二池化输出的变量 `h_pool2` 为开始部分，沿着 `h_pool2` 生成的方式反向操作一层一层推导，直到生成原始图 `t1_x_image`。

如图8-22所示，上半部分是 `h_pool2` 卷积的过程，下半部分为反卷积过程。为了便于分析，下半部分的名字与代码中的变量一致。

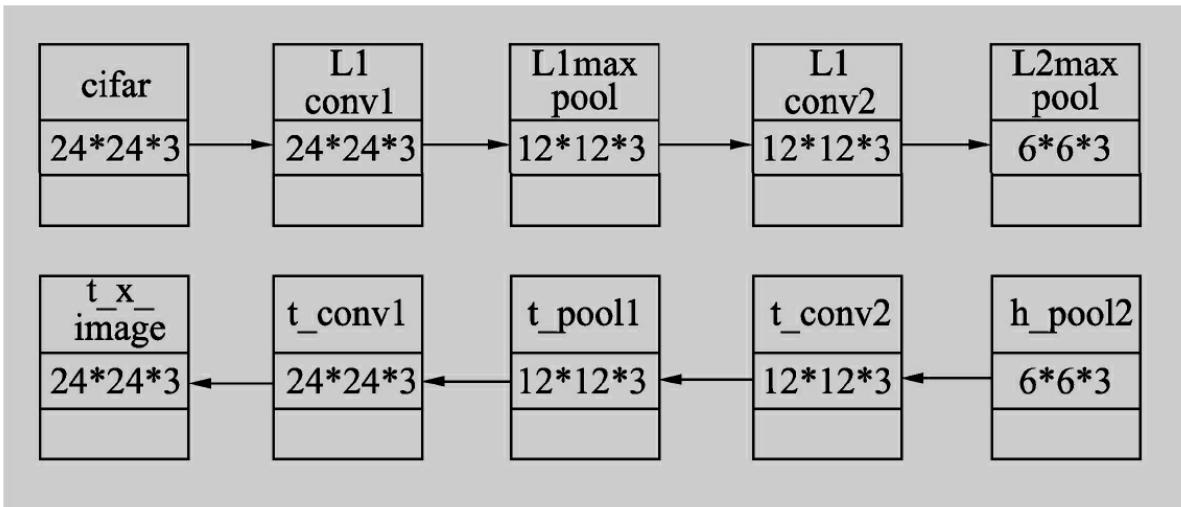


图8-22 反卷积例子

因为在卷积过程中，每个卷积后都要加上权重b，所以在反卷积过程中就要将b减去。由于Relu函数基本上是恒等变化（除了小于0的部分），所以在反向时不需要可逆操作，可以直接略去。

代码8-17 cifar反卷积（续）

```

14 t_conv2 = unpool(h_pool2, mask, 2) #(128, 12, 12, 64)
15 t_pool1 = tf.nn.conv2d_transpose(t_conv2-b_conv2, w_conv2
shape, [1,1,1,1]) #(128, 24, 24, 64)
16 print(t_conv2.shape,h_pool1.shape,t_pool1.shape)
17 t_conv1 = unpool(t_pool1, mask1, 2)
18 t_x_image = tf.nn.conv2d_transpose(t_conv1-b_conv1, w_conv1
shape, [1,1,1,1])

```

3. 反卷积第一层卷积结果

参考第二层的反卷积，第一层会更为简单，代码如下。

代码8-17 cifar反卷积（续）

```
19 #第一层卷积还原  
20 t1_conv1 = unpool(h_pool1, mask1, 2)  
21 t1_x_image = tf.nn.conv2d_transpose(t1_conv1-b_conv:  
image.shape, [1,1,1,1])
```

4. 合并还原结果，并输出给TensorBoard输出

这次是将结果通过TensorBoard进行展示，所以将生成的第一层图片和第二层图片与原始图片合在一起，统一放入tf.summary.image里，这样在TensorBoard的image里就能看到了。

代码8-17 cifar反卷积（续）

```
22 # 生成最终图像  
23 stitched_decodings = tf.concat((x_image, t1_x_image, t_x_:  
24 decoding_summary_op = tf.summary.image('source/cifar', s1  
decodings)
```

5. session中写入log

按照前面介绍过的TensorBoard步骤，在session中建立一个summary_writer，然后在代码结尾处通过session.run运行前面的tf.summary.image操作，使用summary_writer将得出的结果写入log。

代码8-17 cifar反卷积（续）

```
25 ....
26 cross_entropy = -tf.reduce_sum(y*tf.log(y_conv)) +(tf.nn.
(w_conv1)+tf.nn.l2_loss(w_conv2)+tf.nn.l2_loss(w_conv3))
27
28 train_step = tf.train.AdamOptimizer(1e-4).minimize(cross_
29
30 correct_prediction = tf.equal(tf.argmax(y_conv,1), tf.arg_
31 accuracy = tf.reduce_mean(tf.cast(correct_prediction, "f_
32 sess = tf.Session()
33 sess.run(tf.global_variables_initializer())
34 summary_writer = tf.summary.FileWriter('./log/', sess.gra_
35 ....
36 decoding_summary = sess.run(decoding_summary_op, feed_dict_
batch, y: label_b})
37 summary_writer.add_summary(decoding_summary)
```

这里在计算cross_entropy时，对所有的w权重用了一次Loss2正则化。

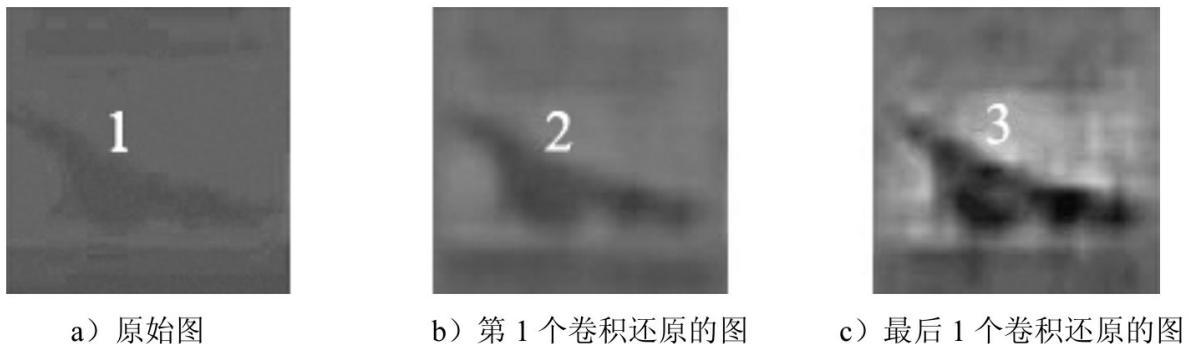
6. Tensorboard中查看结果

运行以上代码后，就可以在TensorBoard中查看结果了。

上面的log是写在本代码同级目录下的log文件夹内，启动TensorBoard的步骤可以参考前面的介绍，这里不再多讲（一定要把路径要找对）。

上面的代码中，image定义的路径是source/cifar，所以在TensorBoard中单击image就会看到source，点开后就能看到如图8-23所示的图片。

*****ebook converter DEMO Watermarks*****



a) 原始图

b) 第 1 个卷积还原的图

c) 最后 1 个卷积还原的图

图8-23 反卷积结果1

图8-23中的数字是后面标注的。第1幅是原始图片，其很不清晰的原因是在cifar10_input.inputs代码中，将图片做的归一化（变成-1~1之间的数）。第2幅是第一个卷积层还原的图片，第3幅是最后一个卷积层还原的图片。可以看到，最后的卷积输出对图像的主要特征响应更强烈。

为了让图片看得更清晰，我们去掉归一化的操作，使用原始图片在模型中“跑”一下。来到“cifar10_input.py”文件中将第241行代码修改如：

```
float_image = resized_image
# Subtract off the mean and divide by the variance of the
#float_image = tf.image.per_image_standardization(resized_
```

再次运行代码，在TensorBoard中如图8-24所示。这时1、2、3分别代表原图、1层卷积后和2层卷积后的图片。

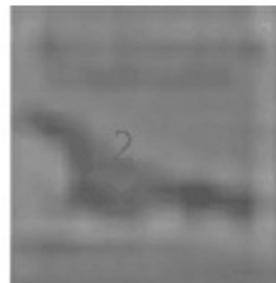
与归一化的效果对比，显然归一化后的图片

*****ebook converter DEMO Watermarks*****

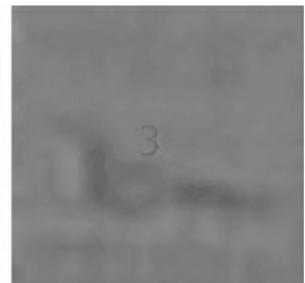
卷积效果特征会更加明显，这也是为什么做归一化的原因。



a) 原始图



b) 第 1 层卷积后的图



c) 第 2 层卷积后的图

图8-24 反卷积结果2

8.8 善用函数封装库

本节讲一下TensorFlow中的一个封装好的高级库，里面有前面讲过的很多函数的高级封装，使用这个高级库来开发程序将会提高效率。那么这个高级库具体好在哪里？请看下面的例子。

8.8.1 实例51：使用函数封装库重写CIFAR卷积网络

改写代码代码“8-9 cifar卷积.py”，将网络结构中的全连接、卷积和池化全部用 tensorflow.contrib.layers 改写。

实例描述

将“代码8-9：cifar卷积.py”中的代码使用 tf.contrib.layers 重构。

1. 改写代码

卷积函数使用 tf.contrib.layers.conv2d，池化使用 tf.contrib.layers.max_pool2d 和 tf.contrib.layers.avg_pool2d，这次使用全连接来作为输出层，并演示全连接函数 tf.contrib.layers.fully_connected 的使用。

代码8-18 cifar简洁代码

```
.....  
#定义占位符  
x = tf.placeholder(tf.float32,[None, 24,24,3]) #CIFAR数据集的  
y = tf.placeholder(tf.float32,[None, 10]) # 0-9 数字分类=> 10  
  
x_image = tf.reshape(x, [-1,24,24,3])  
  
h_conv1 =tf.contrib.layers.conv2d(x_image,64,5,1,'SAME',act:  
h_pool1 = tf.contrib.layers.max_pool2d(h_conv1,[2,2],stride:  
'SAME')  
  
h_conv2=tf.contrib.layers.conv2d(h_pool1,64,[5,5],1,'SAME',  
fn=tf.nn.relu)  
h_pool2 = tf.contrib.layers.max_pool2d(h_conv2,[2,2],stride:  
'SAME')  
  
nt_hpool2 = tf.contrib.layers.avg_pool2d(h_pool2,[6,6],strid  
'SAME')  
  
nt_hpool2_flat = tf.reshape(nt_hpool2, [-1, 64])  
  
y_conv = tf.contrib.layers.fully_connected(nt_hpool2_flat,10  
fn=tf.nn.softmax)  
  
cross_entropy = -tf.reduce_sum(y*tf.log(y_conv))  
train_step = tf.train.AdamOptimizer(1e-4).minimize(cross_ent  
  
correct_prediction = tf.equal(tf.argmax(y_conv,1), tf.argmax  
accuracy = tf.reduce_mean(tf.cast(correct_prediction, "float  
  
sess = tf.Session()  
.....
```

这里只修改“8-9cifar卷积.py”中间的代码，代码的运行不会受到影响。可以看到整个代码段变得简洁了，这就是使用tf.contrib.layers的好处。尤其在深层网络结构中，大量的重复代码会使代码可读性越来越差，所以使用tf.contrib.layers不失为

一个好办法。

2. tf.contrib.layers中的具体函数介绍

看似简单的函数，幕后却做了很多事情，在上面的代码中，没有定义权重，没有初始化，没有过多的参数，这些都是tf.contrib.layers帮我们封装好的。

下面以最复杂的卷积为例进行介绍，其他函数与之相似，不再展开介绍。

tf.contrib.layers.conv2d的函数定义如下：

```
def convolution(inputs,
                num_outputs,
                kernel_size,
                stride=1,
                padding='SAME',
                data_format=None,
                rate=1,
                activation_fn=nn.relu,
                normalizer_fn=None,
                normalizer_params=None,
                weights_initializer=initializers.xavier_init,
                weights_regularizer=None,
                biases_initializer=init_ops.zeros_initializer,
                biases_regularizer=None,
                reuse=None,
                variables_collections=None,
                outputs_collections=None,
                trainable=True,
                scope=None):
```

常用的参数说明如下。

- inputs：代表输入。
- num_outputs：代表输出几个channel。这里不需要再指定输入的channel了，因为函数会根据inputs的shape去判断。
- kernel_size：卷积核大小，不需要带上batch和channel，只需输入尺寸即可。[5, 5]就代表 5×5 大小的卷积核。如果长、宽都一样，也可以直接写一个数5。
- stride：步长，默认是长、宽都相等的步长。卷积时，一般都用1，所以默认值也是1，如果长、宽的步长都不同，也可以用一个数组[1, 2]。
- padding：与前面的padding规则一样。
- activation_fn：输出后的激活函数。
- weights_initializer：权重的初始化，默认为initializers.xavier_initializer函数，参见第6章的说明。biases_initializer同理，不再赘述。
- weights_regularizer：正则化项。可以加入正则函数，biases_regularizer同理，不再赘述。
- trainable：是否可训练，如作为训练节点，必须设为True。默认即可。

对于全连接层等其他函数的使用，会在后续代码中找到相应的演示例子，这里就不一一介绍了。

8.8.2 练习题

任选一个前面章节的例子，将其改用使用
`tf.contrib.layers`库来实现。

8.9 深度学习的模型训练技巧

下面看看卷积神经网络的训练有哪些技巧。

8.9.1 实例52：优化卷积核技术的演示

在实际的卷积训练中，为了加快速度，常常把卷积核裁开。比如一个 3×3 的过滤器，可以裁成 3×1 和 1×3 两个过滤器，分别对原有输入做卷积操作，这样可以大大提升运算的速度。

原理：在浮点运算中乘法消耗的资源比较多，我们目的就是尽量减小乘法运算。

- 比如对一个 5×2 的原始图片进行一次 3×3 的同卷积，相当于生成的 5×2 像素中每一个都要经历 3×3 次乘法，那么一共是90次。

- 同样是这个图片，如果先进行一次 3×1 的同卷积需要30次运算，再进行一次 1×3 的同卷积还是30次，一共才60次。

这仅仅是一个很小的数据张量，而且随着张量维度的增大，层数的增多，减少的运算会更多。那么运算量减少了，运算效果会等价吗？答案是肯定的。因为有公式来做保证 3×1 的矩阵乘上 1×3 的矩阵会正好生成 3×3 的矩阵。所以这个技

巧在卷积网络中很常见。

下面我们把这个技巧用在实例中，改写代码“代码8-9 cifar卷积.py”如下。

实例描述

使用优化卷积核技术将代码“8-9 cifar卷积.py”中的代码重构，并观察效果。

代码8-19 cifar卷积核优化（片段）

```
.....  
x_image = tf.reshape(x, [-1, 24, 24, 3])  
  
h_conv1 = tf.nn.relu(conv2d(x_image, w_conv1) + b_conv1)  
h_pool1 = max_pool_2x2(h_conv1)  
w_conv21 = weight_variable([5, 1, 64, 64])  
b_conv21 = bias_variable([64])  
h_conv21 = tf.nn.relu(conv2d(h_pool1, w_conv21) + b_conv21)  
  
w_conv2 = weight_variable([1, 5, 64, 64])  
b_conv2 = bias_variable([64])  
h_conv2 = tf.nn.relu(conv2d(h_conv21, w_conv2) + b_conv2)  
  
h_pool2 = max_pool_2x2(h_conv2)  
.....
```

上面代码中，将原来的第二层 5×5 的卷积操作conv2注释掉，换成两个 5×1 和 1×5 的卷积操作，代码运行后可以看到准确率没有变化，但是速度快了一些。

8.9.2 实例53：多通道卷积技术的演示

*****ebook converter DEMO Watermarks*****

这里介绍的多通道卷积，可以理解为一种新型的CNN网络模型，在原有的卷积模型基础上的扩展。

- 原有的卷积层中是使用单个尺寸的卷积核对输入数据卷积操作（如图8-25中的上半部分），生成若干个feature map。
- 而多通道卷积的变化就是，在单个卷积层中加入若干个不同尺寸的过滤器（如图8-25中的下半部分），这样会使生成的feature map特征更加多样性。

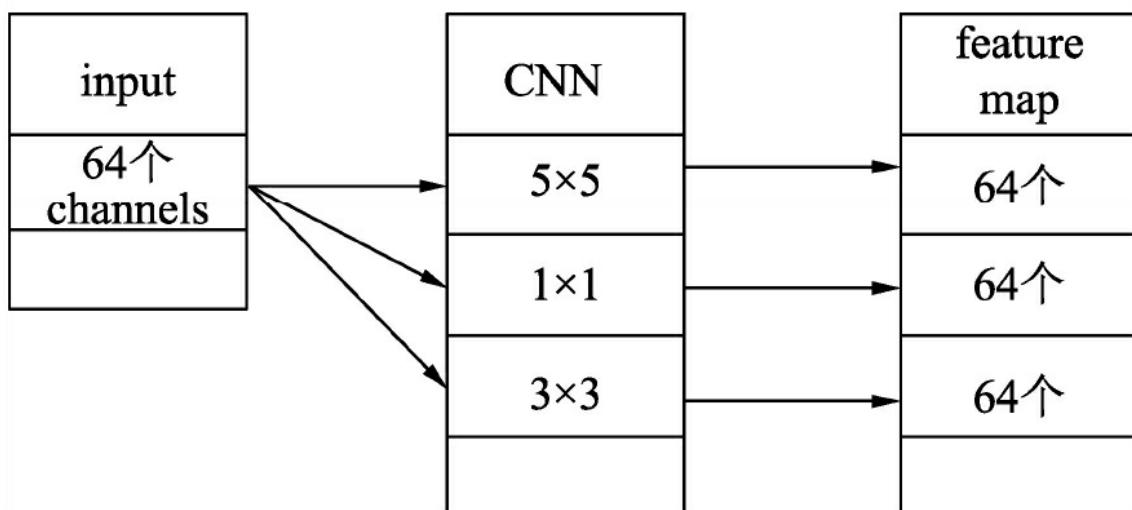
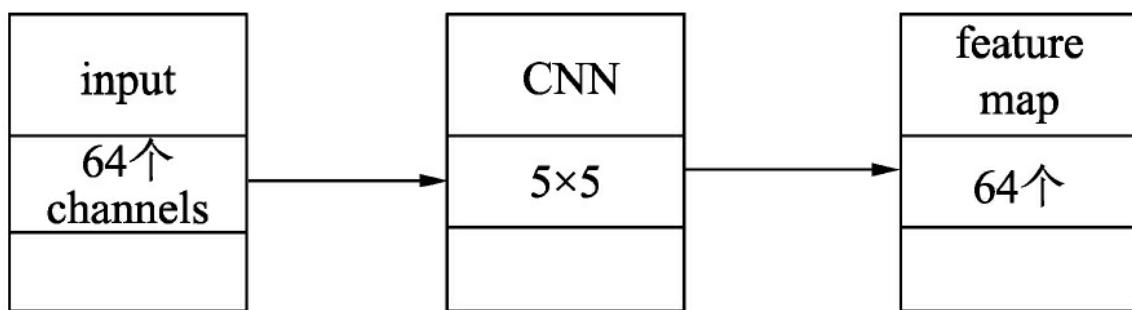


图8-25 多通道卷积

同样还是在代码“8-9 cifar卷积.py”中修改，为网络的卷积层增加不同尺寸的卷积核。这里将原有的 5×5 卷积，扩展到 7×7 卷积、 1×1 卷积、 3×3 卷积，并将它们的输出通过concat函数并在一起。

实例描述

使用多通道技术将代码“8-9 cifar卷积.py”中的代码重构，并观察效果。

代码8-20 cifar多通道卷积

```
.....  
x_image = tf.reshape(x, [-1, 24, 24, 3])  
  
h_conv1 = tf.nn.relu(conv2d(x_image, w_conv1) + b_conv1)  
h_pool1 = max_pool_2x2(h_conv1)  
#####多卷积  
w_conv2_5x5 = weight_variable([5, 5, 64, 64])  
b_conv2_5x5 = bias_variable([64])  
w_conv2_7x7 = weight_variable([7, 7, 64, 64])  
b_conv2_7x7 = bias_variable([64])  
  
w_conv2_3x3 = weight_variable([3, 3, 64, 64])  
b_conv2_3x3 = bias_variable([64])  
  
w_conv2_1x1 = weight_variable([3, 3, 64, 64])  
b_conv2_1x1 = bias_variable([64])  
  
h_conv2_1x1 = tf.nn.relu(conv2d(h_pool1, w_conv2_1x1) + b_conv2_1x1)  
h_conv2_3x3 = tf.nn.relu(conv2d(h_pool1, w_conv2_3x3) + b_conv2_3x3)  
h_conv2_5x5 = tf.nn.relu(conv2d(h_pool1, w_conv2_5x5) + b_conv2_5x5)  
h_conv2_7x7 = tf.nn.relu(conv2d(h_pool1, w_conv2_7x7) + b_conv2_7x7)  
h_conv2 = tf.concat([h_conv2_5x5, h_conv2_7x7, h_conv2_3x3, h_conv2_1x1], 3)
```

```
h_pool2 = max_pool_2x2(h_conv2)
#####
W_conv3 = weight_variable([5, 5, 256, 10])
b_conv3 = bias_variable([10])
h_conv3 = tf.nn.relu(conv2d(h_pool2, W_conv3) + b_conv3)

nt_hpool3=avg_pool_6x6(h_conv3)#10
nt_hpool3_flat = tf.reshape(nt_hpool3, [-1, 10])
y_conv=tf.nn.softmax(nt_hpool3_flat)

.....
```

上面代码中， 1×1 、 3×3 、 5×5 、 7×7 的卷积操作输入都是`h_pool1`，每个卷积操作后都生成了64个feature map，再用concat函数将它们合在一起变成一个[batch、12, 12, 256]大小的数据（4个64channels=256个channels）。

代码运行后输出如下：

```
step 0, training accuracy 0.0859375
step 200, training accuracy 0.296875
step 400, training accuracy 0.445312
step 600, training accuracy 0.414062
step 800, training accuracy 0.4375
step 1000, training accuracy 0.484375
step 1200, training accuracy 0.5
.....
step 14000, training accuracy 0.671875
step 14200, training accuracy 0.671875
step 14400, training accuracy 0.59375
step 14600, training accuracy 0.695312
step 14800, training accuracy 0.609375
finished! test accuracy 0.664062
```

如果上面的concat函数会让你迷惑的话，可以参考第4章中关于concat的说明。

8.9.3 批量归一化

还有一种应用十分广泛的优化方法——批量归一化（简称BN算法）。一般用在全连接或卷积神经网络中。这个里程碑式技术的问世，使得整个神经网络的识别准确度上升了一个台阶，下面就来介绍下其具体内容。

1. 批量归一化介绍

先来看下面的例子：

假如有一个极简的网络模型，每一层只有一个节点，没有偏置。那么如果这个网络有三层的话，可以用如下式子表示其输出值：

$$Z = x \times w_1 \times w_2 \times w_3$$

假设有两个神经网络，学习出了两套权重（ $w_1: 1, w_2: 1, w_3: 1$ ）和（ $w_1: 0.01, w_2: 10000, w_3: 0.01$ ），它们对应的输出 z 都是相同的。现在让它们训练一次，看看会发生什么。

(1) 反向传播：假设反向传播时计算出的损失值 Δy 为1，那么对于这两套权重的修正值将变为（ $\Delta w_1: 1, \Delta w_2: 1, \Delta w_3: 1$ ）和（ $\Delta w_1: 100, \Delta w_2: 0.0001, \Delta w_3: 100$ ）。

(2) 更新权重：这时更新过后的两套权重就变成了（w1: 2, w2: 2, w3: 2）和（w1: 100.01, w2: 10000.0001, w3: 100.01）。

(3) 第二次正向传播：假设输入样本是1，第一个神经网络的值为：

$$Z=1 \times 2 \times 2 \times 2 = 8$$

第二个神经网络的值为：

$$Z=1 \times 100.01 \times 10000.0001 \times 100.01 = 100000000$$

看到这里，读者是不是已经感觉到两个网络的输出值差别巨大？如果再往下进行，这时计算出的loss值会变得更大，使得网络无法计算，这种现象也叫做梯度爆炸。产生梯度爆炸的原因就是因为网络的内部协变量转移（Internal Covariate Shift），即正向传播时的不同层的参数会将反向训练计算时所参照的数据样本分布改变。

这就是引入批量正则化的目的。它的作用是要最大限度地保证每次的正向传播输出在同一分布上，这样反向计算时参照的数据样本分布就会与正向计算时的数据分布一样了。保证了分布统一，对权重的调整才会更有意义。

了解了原理之后，再来看批量正则化的做法就会变得很简单，即将每一层运算出来的数据都

归一化成均值为0方差为1的标准高斯分布。这样就会在保留样本分布特征的同时，又消除了层与层间的分布差异。



提示： 在实际应用中，批量归一化的收敛非常快，并且具有很强的泛化能力，某种情况下可以完全代替前面讲过的正则化、Dropout。

2. 批量归一化定义

先来看看TensorFlow中自带的BN函数定义：

```
tf.nn.batch_normalization(x, mean, variance, offset, scale, variance_epsilon=0.001, name=None)
```

它的参数很简单，各参数说明如下。

- x：代表输入。
- mean：代表样本的均值。
- variance：代表方差。
- offset：代表偏移，即相加一个转化值，后面我们会用激活函数来转换，所以这里不需要再转化，直接使用0。
- scale：代表缩放，即乘以一个转化值，同

理，一般都用1。

· variance_epsilon：是为了避免分母为0的情况，给分母加一个极小值。默认即可。

要想使用这个函数，必须由另一个函数配合——tf.nn.moments，由它来计算均值和方差，然后就可以使用BN了。tf.nn.moments 定义如下：

```
tf.nn.moments(x, axes, name=None, keep_dims=False)
```

axes主要是指定哪个轴来求均值与方差。



注意： axes在使用过程中经常容易犯错。这里提供一个小技巧，为了求样本的均值和方差，一般都会设为保留最后一个维度，对于x来讲可以直接使用公式axis = list (range (len (x.get_shape ())) - 1)) 即可。例如，[128, 3, 3, 12] axes就为[0, 1, 2]，输出的均值方差维度为[12]

有了上面的两个函数还不够，为了有更好的效果，我们希望使用平滑指数衰减的方法来优化每次的均值与方差，于是就用到了tf.train.ExponentialMovingAverage函数。它的作用是让上一次的值对本次的值有个衰减后的影响，从而使每次的值连起来后会相对平滑一些。展开

*****ebook converter DEMO Watermarks*****

后可以用下面的代码来表示：

```
shadow_variable = decay * shadow_variable + (1 - decay) * va
```

各参数说明如下。

- decay：代表衰减指数，是在 ExponentialMovingAverage 中指定的，比如 0.9。
- variable：代表本批次样本中的值。
- 等式右边的 shadow_variable：代表上次总样本的值。
- 等式左边 shadow_variable：代表计算出来的本次总样本的值。

3. 批量归一化的简单用法

上面的函数虽然参数不多，但需要几个函数联合起来使用，于是 TensorFlow 中的 layers 模块里又实现了一次 BN 函数，相当于把几个函数合并到了一起，使用起来更加简单。下面来介绍一下，在使用时需要引入头文件：

```
from tensorflow.contrib.layers.python.layers import batch_nor
```

函数的定义如下：

*****ebook converter DEMO Watermarks*****

```
def batch_norm(inputs,
               decay=0.999,
               center=True,
               scale=False,
               epsilon=0.001,
               activation_fn=None,
               param_initializer=None,
               param_regularizer=None,
               updates_collections=ops.GraphKeys.UPDATE_OPS,
               is_training=True,
               reuse=None,
               variables_collections=None,
               outputs_collections=None,
               trainable=True,
               batch_weights=None,
               fused=False,
               data_format=DATA_FORMAT_NHWC,
               zero_debias_moving_mean=False,
               scope=None,
               renorm=False,
               renorm_clipping=None,
               renorm_decay=0.99):
```

虽然使用简单，但由于其中的参数较多，也增大了学习难度，因此这里列出一些常用的参数及使用习惯。

- inputs：代表输入。
- decay：代表移动平均值的衰败速度，是使用了一种叫做平滑指数衰减的方法更新均值方差，一般会设为0.9；值太小会导致均值和方差更新太快，而值太大又会导致几乎没有衰减，容易出现过拟合，这种情况一般需要把值调小点。
- scale：是否进行变化（通过乘一个gamma

值进行缩放），我们常习惯在BN后面接着一个线性的变化，如Relu。所以scale一般都会设为False。因为后面有对数据的转化处理，因此这里就不用再处理了。

- epsilon: 是为了避免分母为0的情况，给分母加一个极小值。一般默认即可。
- is_training: 当它为True时，代表是训练过程，这时会不断更新样本集的均值与方差。当测试时，要设成False，这样就会使用训练样本集的均值与方差。
- updates_collections: 其变量默认是tf.GraphKeys.UPDATE_OPS，在训练时提供了一种内置的均值方差更新机制，即通过图（一个计算任务）中的tf.GraphKeys.UPDATE_OPS变量来更新。但它是在每次当前批次训练完成后才更新均值和方差，这样导致当前数据总是使用前一次的均值和方差，没有得到最新的更新。所以一般都会将其设成None，让均值和方差即时更新。这样做虽然相比默认值在性能上稍慢点，但是对模型的训练还是有很大帮助的。
- reuse: 支持共享变量，与下面的scope参数联合使用
- scope: 指定变量的作用域variable_scope。

8.9.4 实例54：为CIFAR图片分类模型添加BN

本例将演示BN函数的使用方法，同样是在原有的代码“8-9 cifar卷积.py”例子中修改，具体步骤如下。

实例描述

使用BN算法将代码“8-9 cifar卷积.py”中的代码重构，并观察其效果。

1. 添加BN函数

改写代码“8-9 cifar卷积.py”，在池化函数后面加入BN函数。

代码8-21 cifarBN

```
01 .....
02 def avg_pool_6x6(x):
03     return tf.nn.avg_pool(x, ksize=[1, 6, 6, 1],
04                           strides=[1, 6, 6, 1], padding='SAME')
05 def batch_norm_layer(value, train = None, name = 'batch_norm'):
06     if train is not None:
07         return batch_norm(value, decay = 0.9, updates_collections =
08                           [train], is_training = True)
09     else:
10         return batch_norm(value, decay = 0.9, updates_collections =
11                           [tf.GraphKeys.UPDATE_OPS], is_training = False)
```

2. 为BN函数添加占位符参数

由于BN里面需要设置是否为训练状态，所以这里定义一个train将训练状态当成一个占位符来传入。

代码8-21 cifarBN（续）

```
10 x = tf.placeholder(tf.float32, [None, 24, 24, 3]) #CIFAR数据集  
    为24×24×3  
11 y = tf.placeholder(tf.float32, [None, 10]) # 10 类  
12 train = tf.placeholder(tf.float32)
```

3. 修改网络结构添加BN层

在第一层h_conv1与第二层h_conv2的输出之前卷积之后加入BN层。

代码8-21 cifarBN（续）

```
13 .....  
14 h_conv1 = tf.nn.relu(batch_norm_layer((conv2d(x_image, w_  
b_conv1),train))  
15 h_pool1 = max_pool_2x2(h_conv1)  
16  
17 W_conv2 = weight_variable([5, 5, 64, 64])  
18 b_conv2 = bias_variable([64])  
19  
20 h_conv2 = tf.nn.relu(batch_norm_layer((conv2d(h_pool1, w_  
b_conv2),train))  
21 h_pool2 = max_pool_2x2(h_conv2)  
22 .....
```

4. 加入退化学习率

将原来的学习率改成退化学习率，使用0.04的初始值，让其每1000次退化0.9。

代码8-21 cifarBN（续）

```
23 .....
24 cross_entropy = -tf.reduce_sum(y*tf.log(y_conv))
25 global_step = tf.Variable(0, trainable=False)
26 decaylearning_rate = tf.train.exponential_decay(0.04, global_step,
1000, 0.9)
27
28 train_step = tf.train.AdamOptimizer(decaylearning_rate).minimize(cross_entropy, global_step=global_step)
29 correct_prediction = tf.equal(tf.argmax(y_conv, 1), tf.argmax(y_, 1))
30 accuracy = tf.reduce_mean(tf.cast(correct_prediction, "float"))
31 .....
```

5. 在运行session中添加训练标志

在session中找到循环的部分，为占位符train添加数值1，表明当前是训练状态。其他的地方都不用改动，因为在第一步的BN函数里设定好train为None时，已经认为是测试状态。

代码8-21 cifarBN（续）

```
32 .....
33 for i in range(20000):
34     image_batch, label_batch = sess.run([images_train, labels_train])
35     label_b = np.eye(10, dtype=float)[label_batch] #one hot
36
37     train_step.run(feed_dict={x:image_batch, y: label_b, train=True},
38                     session=sess)
39 .....
```

运行代码，得到如下输出：

```
begin
begin data
step 0, training accuracy 0.210938
step 200, training accuracy 0.484375
step 400, training accuracy 0.601562
step 600, training accuracy 0.617188
.....
step 18400, training accuracy 0.921875
step 18600, training accuracy 0.921875
step 18800, training accuracy 0.921875
step 19000, training accuracy 0.953125
step 19200, training accuracy 0.9375
step 19400, training accuracy 0.914062
step 19600, training accuracy 0.96875
step 19800, training accuracy 0.9375
finished! test accuracy 0.71875
```

可以看到，准确率有了明显提升，训练时达到了90%以上，测试时模型的准确率下降了不少。有兴趣的读者可以通过对原样本变形的方式来增大数据集（使用cifar10_input中的distorted_inputs来获取数据），或采用前面讲的一些过拟合的方法继续优化。

8.9.5 练习题

(1) 搭建神经网络，使用多通道卷积，将MNIST数据集进行分类（代码见“8-23多通道mnist.py”。）

(2) 再接着练习(1) 将多通道卷积加入批

量正则化处理（代码见“8-22带BN的多通道mnist.py”，可将准确率提高到99%）。

第9章 循环神经网络——具有记忆功能的网络

前面讲的内容可以理解为静态数据的处理，也就是样本是单次的，彼此之间没有关系。然而人工智能对计算机的要求不仅仅是单次的运算，还需要让计算机像人一样具有记忆功能。这一节我们就来学习循环神经网络（RNN），它是一个具有记忆功能的网络。这种网络最适合解决连续序列的问题，善于从具有一定顺序意义的样本与样本间学习规律。

本章含有教学视频共10分53秒。

本章的内容比较多，作者按照书中的内容结构做了快速讲解。在视频内容中主要对RNN的作用、原理和结构做了清晰的讲解，同时对本章内容中各个技术点的学习方法及后面的实例通用性做了补充说明。

深度学习之TensorFlow

入门、原理与进阶实战

第9章 循环神经网络 ——具有记忆功能的网络

配套视频



代码医生

qq群: 40016981

<http://blog.csdn.net/lijin6249>



字幕



*****ebook converter DEMO Watermarks*****

9.1 了解RNN的工作原理

在解释RNN原理之前，我们先看看人脑是怎么处理的。

9.1.1 了解人的记忆原理

如果你身边有2岁或3岁的孩子，可以仔细观察一下，他说话时虽然能表达出具体的意思，但是听起来总会觉得怪怪的。比如笔者的孩子，在刚开始说话时，把“我要”说成了“要我”，一看到喜欢吃的小零食，就会用手指着小零食对你大喊“要我，要我……”。

类似这样的话为什么我们听起来会感觉很别扭呢？这是因为我们的大脑受刺激时对一串后续的字有预测功能。如果从神经网络的角度来理解，大脑中的语音模型在某一场景下一定是对这两个字有先后顺序区分的。比如，第一个字是“我”，后面跟着“要”，人们就会觉得正常，而使用“要我”，来匹配“我要”的意思在生活中很少遇到，于是人们就会觉得很奇怪。

当获得“我来找你玩游”信息后，大脑的语言模型会自动预测后一个字为“戏”，而不是“乐、泳”等其他字，如图9-1所示。

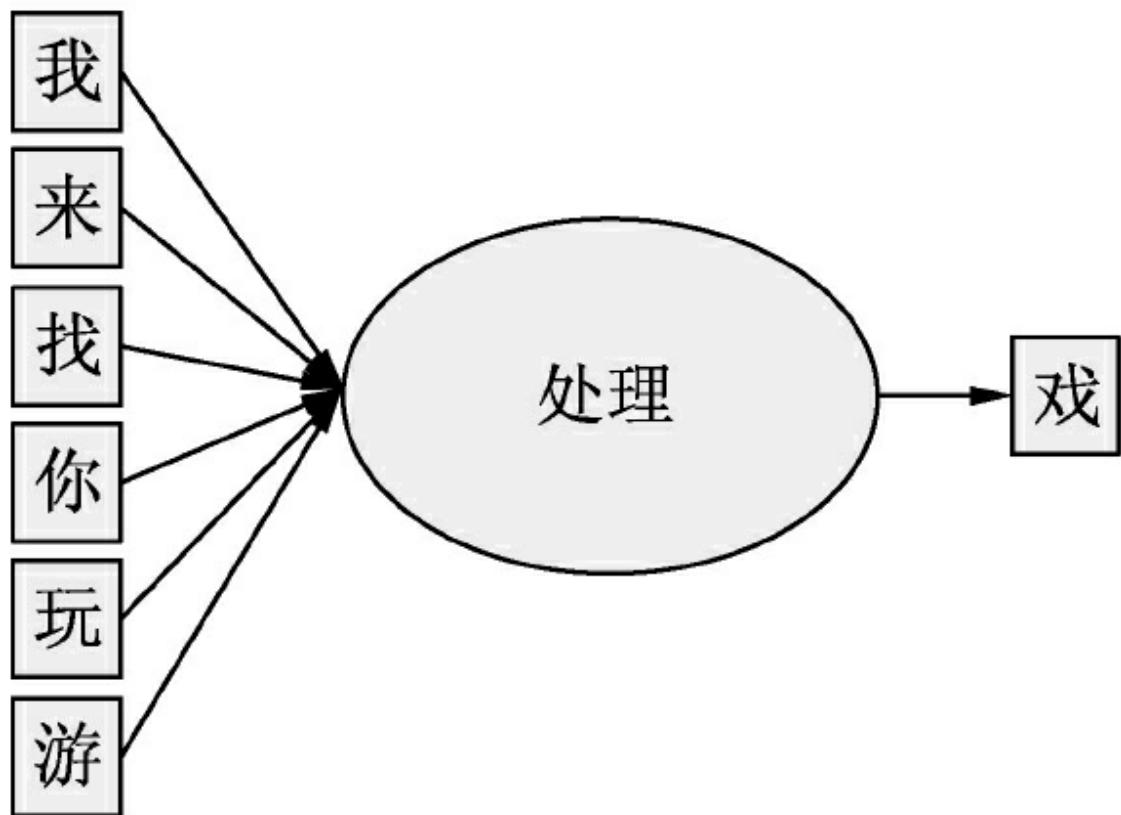


图9-1 人脑处理文字举例

图9-1中的逻辑并不是在说完“我来找你玩游”之后进入大脑来处理的，而是每个字都在脑子里进行着处理，将图9-1中的每个字分别裁开，在语言模型中就形成了一个循环神经网络，图9-1中的逻辑可以用下面的伪码表示：

```
(input我+ empty-input) ->output我  
(input来+ output我) ->output来  
(input找+ output来) ->output找  
(input你+ output找) ->output你  
.....
```

即，每个预测的结果都会放到下一个输入里进行运算，与下一次的输入一起来生成下一次的

结果。如图9-2所示的网络模型可以很好地表达我们见到的现象。

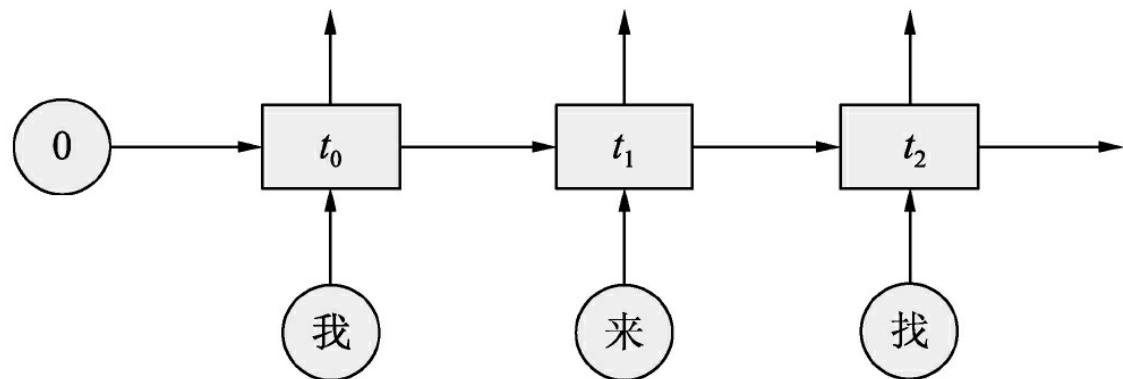


图9-2 RNN结构

图9-2也可以看成是一个链式的结构，如何理解链式结构呢？举个例子：后来我的孩子上了幼儿园，学习了《三字经》，而且可以背很长的内容，背得很熟练，于是我想考考他，就问了一个中间的句子“名俱扬”下一句是啥，他很快说了出来，马上又问他上一句是啥，他想了半天，从头背了一遍，背到“名俱扬”时才知道上一句是“教五子”。这种现象可以理解为我们大脑并不是简单的存储，而是链式的、有顺序的存储。

这种“链式的、有顺序存储”很节省空间，对于中间状态的序列，我们的大脑没有选择直接记住，而是存储计算方法。当我们需要取值时，直接将具体的数据输入，通过计算得出来相应的结果。这种解决方法在很多具体问题时都会用到。

例如：程序员常常会使用一个递归的函数来

求阶乘 $n! = n \times (n-1) \times \dots \times 1$ 。

函数的代码如下：

```
long ff(int n) {
    long f;
    if(n<0) printf("n<0, input error");
    else if(n==0||n==1) f=1;
    else f=ff(n-1)*n;
    return(f);
}
```

还有，我们在计算加法时的进位过程。

23+17的加法过程是：先个位加个位，再算十位加十位；然后将个位的结果状态（是否有进位）送到十位的运算中去，则十位是“2+1+个位的进位数（1）”等于4。

9.1.2 RNN网络的应用领域

对于序列化的特征的任务，都适合采用RNN网络来解决。细分起来可以有情感分析（Sentiment Analysis）、关键字提取（Key Term Extraction）、语音识别（Speech Recognition）、机器翻译（Machine Translation）和股票分析等。

9.1.3 正向传播过程

RNN结构如图9-3左侧图所示，A代表网络， x_t 代表t时刻输入的x， h_t 代表网络生成的结果，A中间又画出了一条线指向自己，是表明上一时刻的输出接着输入到了A里面。

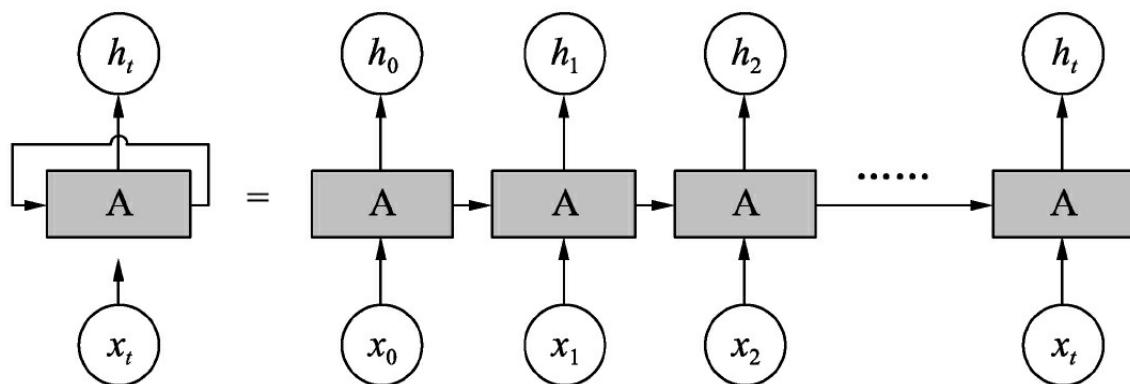


图9-3 RNN正向传播

当有一系列的x输入到图9-3左侧结构中后，展开就变成了右侧的样子，其实就是一个含有隐藏层的网络，只不过隐藏层的输出变成了两份，一份传到下一个节点，另一份传给本身节点。其时序图如图9-4所示。

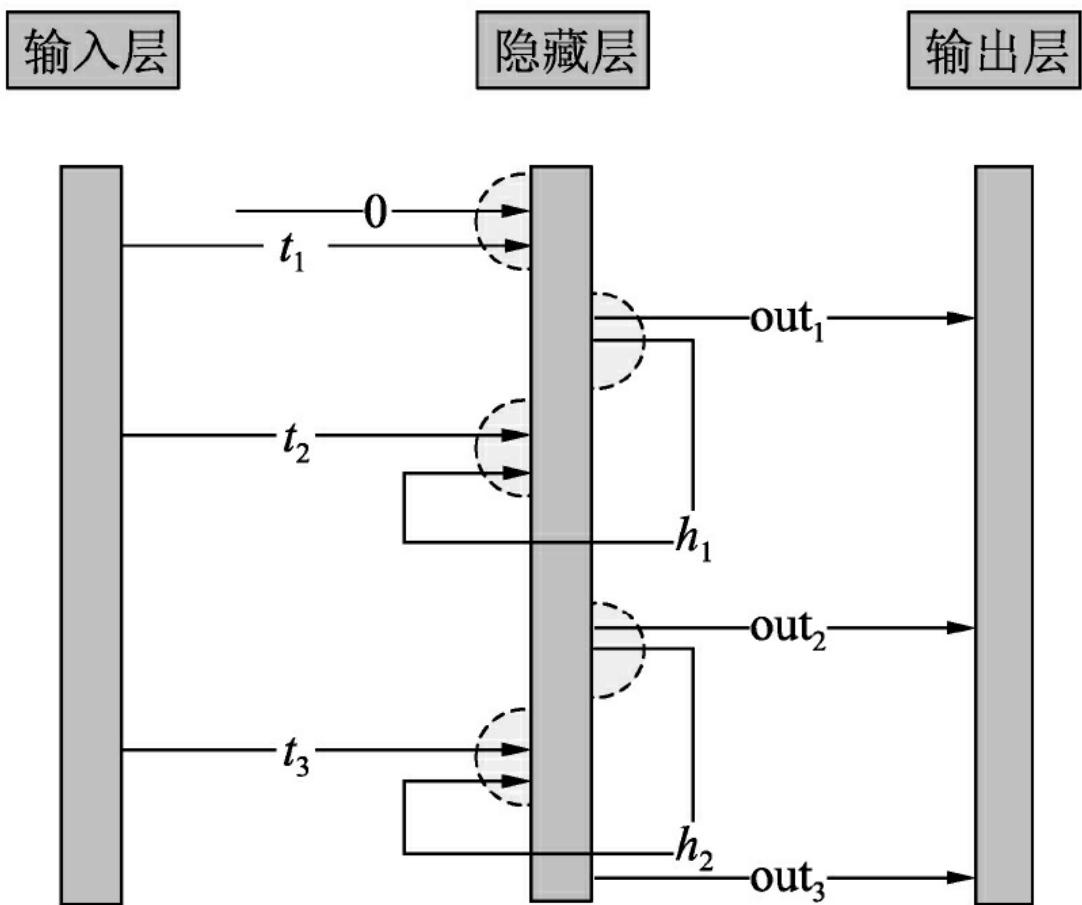


图9-4 RNN正向传播时序

假设有3个时序 t_1 、 t_2 、 t_3 ，如图9-4所示，在RNN中可以分解成以下3个步骤。

(1) 开始时 t_1 通过自己的输入权重和0作为输入，生成了 out_1 。

(2) out_1 通过自己的权重生成了 h_1 ，然后和 t_2 经过输入权重转化后一起作为输入，生成了 out_2 。

(3) out_2 通过同样的隐藏层权重生成了 h_2

，然后和 t_3 经过输入权重转化后一起作为输入，生成了 out_2 。

9.1.4 随时间反向传播

与单神经元相似，RNN也需要反向传播误差来调整自己的参数。RNN网络使用随时间反向传播（BackPropagation Through Time，BPTT）的链式求导算法来反向传播误差。

先来回顾一下反向传播的BP算法，如图9-5所示。

这是一个含有一个隐藏层的网络结构。隐藏层只有一个节点。具体的过程如下：

(1) 有一个批次含有3个数据A、B、C，批次中每个样本有两个数(x_1 、 x_2)通过权重(w_1 、 w_2)来到隐藏层H并生成批次h，如图9-5中 w_1 、 w_2 两条直线所在方向。

(2) 该批次的h通过隐藏层权重 p_1 生成最终的输出结果y。

(3) y与最终的标签p比较，生成输出层误差less(y, p)。

(4) less(y, p)与生成y的导数相乘，得

到Del_y。Del_y为输出层所需要的修改值。

(5) 将h的转置与del_y相乘得到del_p1。这是源于h与p1相等得到的y，见第(2)步。

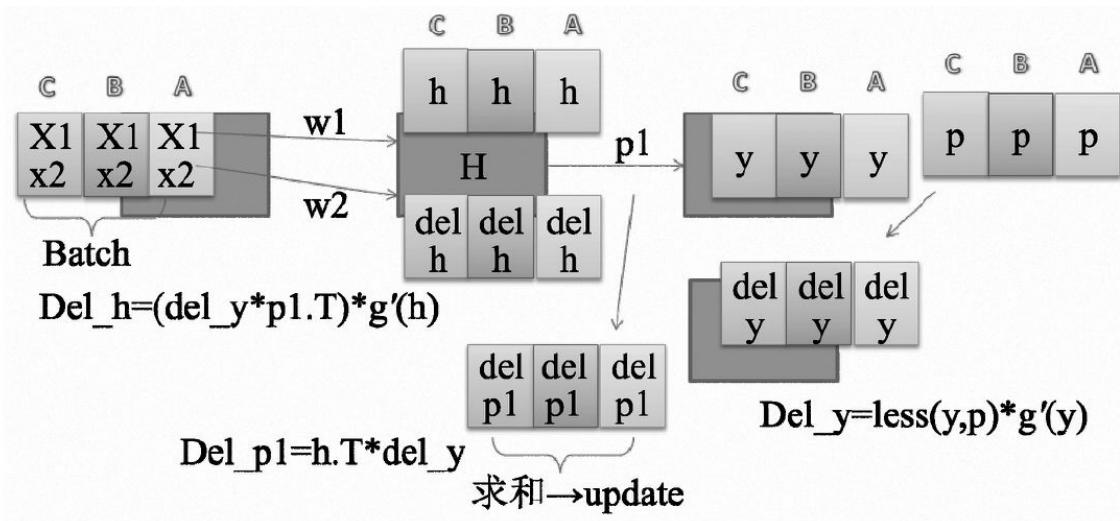


图9-5 BP反向传播1

(6) 最终将该批次的del_p1求和并更新到p1上。

(7) 同理，再将误差反向传递到上一层：计算Del_h。得到Del_h后再计算del_w1、del_w2并更新。

若BP的算法读者已经理解了，下面再来比较一下BPTT算法，如图9-6所示。

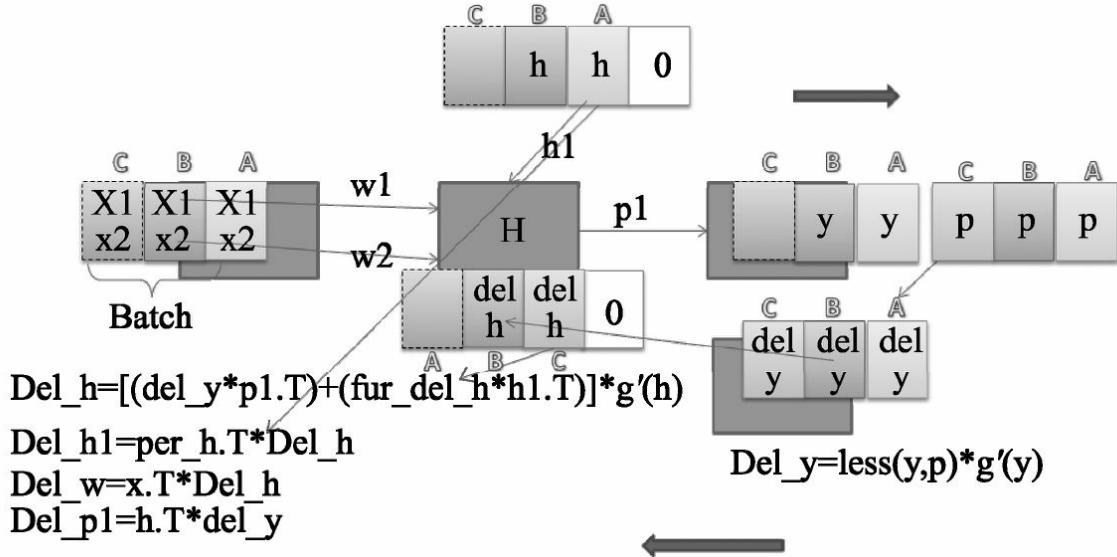


图9-6 RNN 反向传播

图9-6中，同样是一个批次的数据A、B、C，按顺序进入循环神经网络。正向传播的实例是B正在进入神经网络的过程，可以看到A的h参与进来并一起经过P1生成了B的y。因为C还没有进入，为了清晰，这里用灰色表示。

当所有块都进入之后，会将p标签与输出进行 Del_y 的运算，由于C块的y是最后生成的，所以我们先从C块开始对h的输出传递误差 Del_h 。

图9-6中的反向传播是表示C块已经反向传播完成，开始B块反向传播的状态，可以看到B块 Del_h 是由B块的 del_y 和C块的 del_h 通过计算得来的。这就是与BP算法不同的地方（BP中 Del_h 直接与自己的 Del_y 相关，不会与其他的值有联系）。

作为一个批次的数据，正向传播时是沿着ABC的顺序，当反向传播时，就按照正向传播的相反顺序，即每个节点的CBA挨个计算并传递梯度。

9.2 简单RNN

了解完RNN的原理后，下面一起来实现一个简单的RNN网络。

9.2.1 实例55：简单循环神经网络实现——裸写一个退位减法器

本例将把前面所讲述的内容用代码实现一遍。如果前面的描述读者还不明白，可以通过这个例子，加深对前面内容的理解。

本例是一个纯手写的代码例子，使用Python手动搭建一个简单的RNN网络，让它来拟合一个退位减法。退位减法也具有RNN的特性，即输入的两个数相减时，一旦发生退位运算，需要将中间状态保存起来，当高位的数传入时将退位标志一并传入参与运算。

下面就来用代码实现RNN拟合减法，具体步骤如下。

实例描述

使用Ptyhon编写简单循环神经网络拟合一个退位减法的操作，观察其反向传播过程。

1. 定义基本函数

首先手动写一个Sigmoid函数及其导数（导数用于反向传播）。

代码9-1 subtraction

```
01 import copy, numpy as np
02 np.random.seed(0)          #固定随机数生成器的种子，可以每次
03 def sigmoid(x):           #激活函数
04     output = 1/(1+np.exp(-x))
05     return output
06
07 def sigmoid_output_to_derivative(output): #激活函数的导数
08     return output*(1-output)
```

2. 建立二进制映射

定义的减法最大值限制在256之内，即8位二进制的减法，定义int与二进制之间的映射数组int2binary。

代码9-1 subtraction（续）

```
09 int2binary = {}          #整数到其二进制表示的映射
10 binary_dim = 8           #暂时制作256以内的减法
11 ## 计算0~256的二进制表示
12 largest_number = pow(2,binary_dim)
13 binary = np.unpackbits(
14     np.array([range(largest_number)],dtype=np.uint8).T,a
15 for i in range(largest_number):
16     int2binary[i] = binary[i]
```

3. 定义参数

定义学习参数：隐藏层的权重synapse_0、循环节点的权重synapse_h（输入节点16、输出节点16）、输出层的权重synapse_1（输入16节点，输出1节点）。为了减小复杂度，这里只设置w权重，b被忽略。

代码9-1 subtraction（续）

```
17 # 参数设置
18 alpha = 0.9                         #学习速率
19 input_dim = 2                         #输入的维度是2，减数和被减
20 hidden_dim = 16
21 output_dim = 1                        #输出维度为1
22
23 # 初始化网络
24 synapse_0 = (2*np.random.random((input_dim,hidden_dim)) ·
#维度为2*16, 2是输入维度, 16是隐藏层维度
25 synapse_1 = (2*np.random.random((hidden_dim,output_dim)))
26 synapse_h = (2*np.random.random((hidden_dim,hidden_dim)))
27 # => [-0.05, 0.05),
28
29 # 用于存放反向传播的权重更新值
30 synapse_0_update = np.zeros_like(synapse_0)
31 synapse_1_update = np.zeros_like(synapse_1)
32 synapse_h_update = np.zeros_like(synapse_h)
```

synapse_0_update在前面很少见到，是因为它被隐含在优化器里了。这里全部“裸写”（不使用Tensor Flow库函数），需要定义一组变量，用于反向优化参数时存放参数需要调整的调整值，对应于前面的3个权重synapse_0、synapse_1和synapse_h。

4. 准备样本数据

大致是这样的过程：

(1) 建立循环生成样本数据，先生成两个数a和b。如果a小于b，就交换位置，保证被减数大。

(2) 计算出相减的结果c。

(3) 将3个数转换成二进制，为模型计算做准备。

将上面过程一一实现，代码如下。

代码9-1 subtraction (续)

```
33 # 开始训练
34 for j in range(10000):
35
36     #生成一个数字a
37     a_int = np.random.randint(largest_number)
38     #生成一个数字b, b的最大值取的是largest_number/2, 作为被减数
39     b_int = np.random.randint(largest_number/2)
40     #如果生成的b大了, 那么交换一下
41     if a_int<b_int:
42         tt = a_int
43         b_int = a_int
44         a_int=tt
45
46     a = int2binary[a_int] # 二进制编码
47     b = int2binary[b_int] # 二进制编码
48     # 正确的答案
49     c_int = a_int - b_int
50     c = int2binary[c_int]
```

5. 模型初始化

初始化输出值为0，初始化总误差为0，定义layer_2_deltas存储反向传播过程中的循环层的误差，layer_1_values为隐藏层的输出值，由于第一个数据传入时，没有前面的隐藏层输出值来作为本次的输入，所以需要为其定义一个初始值，这里定义为0.1。

代码9-1 subtraction（续）

```
51      # 存储神经网络的预测值
52      d = np.zeros_like(c)
53      overallError = 0                      #每次把总误差清零
54
55      layer_2_deltas = list()                #存储每个时间点输出层的误差
56      layer_1_values = list()                #存储每个时间点隐藏层的值
57
58      layer_1_values.append(np.ones(hidden_dim)*0.1) # 一开始
```

6. 正向传播

循环遍历每个二进制位，从个位开始依次相减，并将中间隐藏层的输出传入下一位的计算（退位减法），把每一个时间点的误差导数都记录下来，同时统计总误差，为输出准备。

代码9-1 subtraction（续）

```
59 for position in range(binary_dim):          #循环遍历每一个二进制位
60
61      # 生成输入和输出
```

*****ebook converter DEMO Watermarks*****

```
62      X = np.array([[a[binary_dim - position - 1],b[bir
63          position - 1]]]) #从右到左，每次取两个输入数字的一个bit位
64          y = np.array([[c[binary_dim - position - 1]]]).T
65          # hidden layer (input ~+ prev_hidden)
66          layer_1 = sigmoid(np.dot(X,synapse_0) + np.dot(layer_1[-1],synapse_h))# (输入层 + 之前的隐藏层) -> 新的隐藏层，神经网络的最核心的地方
67          # output layer (new binary representation)
68          layer_2 = sigmoid(np.dot(layer_1,synapse_1))    #
69          layer_2_error = y - layer_2
70          layer_2_deltas.append((layer_2_error)*sigmoid_output_
derivative(layer_2)) #把每一个时间点的误差导数都记录下来
71          overallError += np.abs(layer_2_error[0])
72
73          d[binary_dim - position - 1] = np.round(layer_2[0])
74
75          # 将隐藏层保存起来。下个时间序列便可以使用
76          layer_1_values.append(copy.deepcopy(layer_1))
77
78          future_layer_1_delta = np.zeros(hidden_dim)
```

最后一行代码是为了反向传播准备的初始化。同正向传播一样，反向传播是从最后一次往前反向计算误差，对于每一个当前的计算都需要有它的下一次结果参与。

反向计算是从最后一次开始的，它没有后一次的输出，所以需要初始化一个值作为其后一次的输入，这里初始化为0。

7. 反向训练

初始化之后，开始从高位往回遍历，一次对每一位的所有层计算误差，并根据每层误差对权重求偏导，得到其调整值，最终将每一位算出的各层权重的调整值加在一起乘以学习率，来更新

*****ebook converter DEMO Watermarks*****

各层的权重，完成一次优化训练。

代码9-1 subtraction（续）

```
79 #反向传播，从最后一个时间点到第一个时间点
80     for position in range(binary_dim):
81
82         X = np.array([[a[position],b[position]]]) #最后一个
83         layer_1 = layer_1_values[-position-1]      #当前时间
84         prev_layer_1 = layer_1_values[-position-2] #前一个
85
86         layer_2_delta = layer_2_deltas[-position-1] #当前
87         # 通过后一个时间点（因为是反向传播）的隐藏层误差和当前时间
88         #计算当前时间点的隐藏层误差
89         layer_1_delta = (future_layer_1_delta.dot(synapse_
90             2_delta.dot(synapse_1.T)) * sigmoid_output_to_deriva
91             (layer_1))
92
93         # 等完成了所有反向传播误差计算，才会更新权重矩阵，先暂时把更
94         synapse_1_update += np.atleast_2d(layer_1).T.dot(
95             delta)
96         synapse_h_update += np.atleast_2d(prev_layer_1).T.
97             dot(delta)
98         synapse_0_update += X.T.dot(layer_1_delta)
99
100        future_layer_1_delta = layer_1_delta
101
102        # 完成所有反向传播之后，更新权重矩阵，并把矩阵变量清零
103        synapse_0 += synapse_0_update * alpha
104        synapse_1 += synapse_1_update * alpha
105        synapse_h += synapse_h_update * alpha
106        synapse_0_update *= 0
107        synapse_1_update *= 0
108        synapse_h_update *= 0
```

更新完后会将中间变量值清零。

8. 输出结果

每运行800次将结果输出，代码如下。

*****ebook converter DEMO Watermarks*****

代码9-1 subtraction (续)

```
104 # 打印输出过程
105     if(j % 800 == 0):
106
107         print("总误差:" + str(overallError))
108         print("Pred:" + str(d))
109         print("True:" + str(c))
110         out = 0
111         for index,x in enumerate(reversed(d)):
112             out += x*pow(2,index)
113         print(str(a_int) + " - " + str(b_int) + " = " +
114         print("-----")
```

运行代码，输出结果如下：

```
总误差:[ 3.97242498]
Pred:[0 0 0 0 0 0 0 0]
True:[0 0 0 0 0 0 0 0]
9 - 9 = 0
-----
总误差:[ 2.1721182]
Pred:[0 0 0 0 0 0 0 0]
True:[0 0 0 1 0 0 0 1]
17 - 0 = 0
-----
.....
-----
总误差:[ 0.04588656]
Pred:[1 0 0 1 0 1 1 0]
True:[1 0 0 1 0 1 1 0]
167 - 17 = 150
-----
总误差:[ 0.08098026]
Pred:[1 0 0 1 1 0 0 0]
True:[1 0 0 1 1 0 0 0]
204 - 52 = 152
-----
总误差:[ 0.03262333]
Pred:[1 1 0 0 0 0 0 0]
True:[1 1 0 0 0 0 0 0]
209 - 17 = 192
```


可以看到，刚开始还不准，随着迭代次数增加，到后来已经可以完全拟合退位减法了。

9.2.2 实例56：使用RNN网络拟合回声信号序列

本例使用TensorFlow中的函数来演示搭建一个简单RNN网络，使用一串随机的模拟数据作为原始信号，让RNN网络来拟合其对应的回声信号。详细介绍如下。

样本数据为一串随机的由0、1组成的数字，将其当成发射出去的一串信号。当碰到阻挡被反弹回来时，会收到原始信号的回音。

如果步长为3，那么输入和输出的序列如图9-7所示。

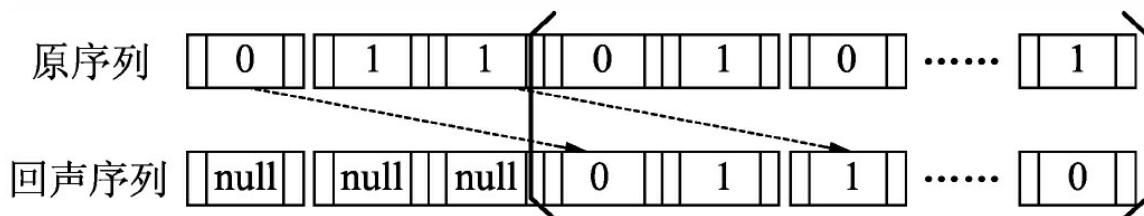


图9-7 回声序列

如图9-7所示，回声序列的前3项是null，原序列的第一个信号0，对应的是回声序列的第4项，即回声序列的每一个数都会比原序列滞后3个时

序。本例的任务就是将序列截取出来，对于每个原序列来预测它的回声序列。

构建的网络结构如图9-8所示。

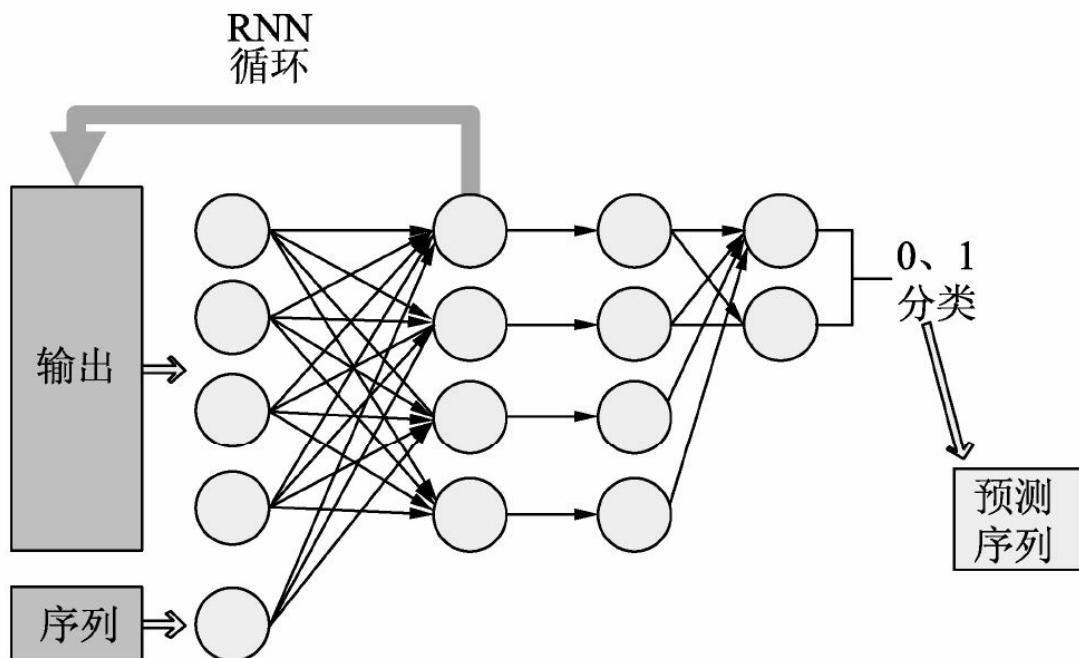


图9-8 echo例子网络结构

图9-8中，初始的输入有5个，其中4个是中间状态，1个是x的序列值。通过一层具有4个节点的RNN网络，再接一个全连接层输出0、1分类。这样序列中的每个x都会有一个对应的预测分类值，最终将整个序列x生成了预测序列。具体步骤如下。

实例描述

构建一组序列，生成其对应的模拟回声序列。使用TensorFlow创建一个简单循环神经网络

*****ebook converter DEMO Watermarks*****

拟合这个回声序列。

1. 定义参数生成样本数据

在了解前面样本的规则后，开始编写代码制作样本。

导入Python库，定义相关参数，取50000个序列样本数据，每个测试数据截取15个序列，回声序列的步长为3，最小批次为5。定义生成样本函数generateData，在函数里先随机生成50000个0、1数据的数组x，作为原始的序列，令x里的数据向右循环移动3个位置，生成数据y，作为x的回声序列。因为回声步长是3，表明回声y是从x的第3个数据开始才出现，所以将y的前3个数据清零。

代码9-2 echo模拟

```
01 import numpy as np
02 import tensorflow as tf
03 import matplotlib.pyplot as plt
04
05 num_epochs = 5
06 total_series_length = 50000
07 truncated_backprop_length = 15
08 state_size = 4
09 num_classes = 2
10 echo_step = 3
11 batch_size = 5
12 num_batches = total_series_length//batch_size//truncated_
length
13
14 def generateData():
15     x = np.array(np.random.choice(2, total_series_length,
0.5)) #在0 和1 中选择total_series_length个数
```

```
16     y = np.roll(x, echo_step)#向右循环移位, 将【1111000】变为
17     y[0:echo_step] = 0
18
19     x = x.reshape((batch_size, -1)) # 5,10000
20     y = y.reshape((batch_size, -1))
21
22     return (x, y)
```

2. 定义占位符处理输入数据

定义3个占位符，输入的batchX_placeholder原始序列，回声batchY_placeholder作为标签，循环节点的初始值state。如前面介绍的网络结构，x的原始序列是逐个输入网络的，所以需要将输进去的数据打散，按照时间序列变成15个数组，每个数组有batch_size个元素，进行统一处理。

代码9-2 echo模拟（续）

```
23 batchX_placeholder = tf.placeholder(tf.float32,[batch_size,
backprop_length])
24 batchY_placeholder = tf.placeholder(tf.int32,[batch_size,
backprop_length])
25 init_state = tf.placeholder(tf.float32, [batch_size, stat
26
27 # 将batchX_Placeholder沿维度为1的轴方向进行拆分
28 inputs_series = tf.unstack(batchX_placeholder, axis=1)
#truncated_backprop_length个序列
29 labels_series = tf.unstack(batchY_placeholder, axis=1)
```

3. 定义网络结构

按照图9-8中的网络结构，定义一层循环网络与一层全连接网络。由于数据是一个数组序列，

*****ebook converter DEMO Watermarks*****

所以需要通过循环将输入数据按照原有序列逐个输入网络，并输出对应的predictions序列。同样的，对于每个序列值都要对其输出做loss计算，在loss中使用了

sparse_softmax_cross_entropy_with_logits函数，因为label的最大值正好是1，而且是一位的，就不需要再转成one_hot编码了（具体细节见本书6.5.3节），最终将所有的loss取均值放入优化器中。

代码9-2 echo模拟（续）

```
30 current_state = init_state
31 predictions_series = []
32 losses = []
33 #使用一个循环，按照序列逐个输入
34 for current_input, labels in zip(inputs_series, labels_series):
35     current_input = tf.reshape(current_input, [batch_size, 1])
36     #加入初始状态
37     input_and_state_concatenated = tf.concat([current_input, state], 1)
38
39     next_state = tf.contrib.layers.fully_connected(input_and_state_concatenated, state_size, activation_fn=tf.tanh)
40     current_state = next_state
41     logits = tf.contrib.layers.fully_connected(next_state, classes, activation_fn=None)
42     loss = tf.nn.sparse_softmax_cross_entropy_with_logits(labels, logits)
43     losses.append(loss)
44     predictions = tf.nn.softmax(logits)
45     predictions_series.append(predictions)
46
47
48 total_loss = tf.reduce_mean(losses)
49 train_step = tf.train.AdagradOptimizer(0.3).minimize(total_loss)
```

4. 建立session训练数据

建立session，总样本循环10次进行迭代。将初始化循环神经网络的状态设为0，在总样本中循环读取15个序列作为批次中的一个样本。

代码9-2 echo模拟（续）

```
50 with tf.Session() as sess:  
51     sess.run(tf.global_variables_initializer())  
52     plt.ion()  
53     plt.figure()  
54     plt.show()  
55     loss_list = []  
56  
57     for epoch_idx in range(num_epochs):  
58         x,y = generateData()  
59         _current_state = np.zeros((batch_size, state_size))  
60  
61         print("New data, epoch", epoch_idx)  
62  
63         for batch_idx in range(num_batches):#50000/ 5 /15  
64             start_idx = batch_idx * truncated_backprop_length  
65             end_idx = start_idx + truncated_backprop_length  
66  
67             batchX = x[:,start_idx:end_idx]  
68             batchY = y[:,start_idx:end_idx]  
69  
70             _total_loss, _train_step, _current_state, _pi  
series = sess.run(  
71                 [total_loss, train_step, current_state, pi  
series],  
72                 feed_dict={  
73                     batchX_placeholder:batchX,  
74                     batchY_placeholder:batchY,  
75                     init_state:_current_state  
76                 })  
77  
78         loss_list.append(_total_loss)
```

5. 测试模型及可视化

每循环100次，将打印数据并调用plot函数生成图像。

代码9-2 echo模拟（续）

```
79         if batch_idx%100 == 0:
80             print("Step",batch_idx, "Loss", _total_lo
81             plot(loss_list, _predictions_series, batc
82
83 plt.ioff()
84 plt.show()
```

plot函数定义如下：

```
85 def plot(loss_list, predictions_series, batchX, batchY):
86     plt.subplot(2, 3, 1)
87     plt.cla()
88     plt.plot(loss_list)
89
90     for batch_series_idx in range(batch_size):
91         one_hot_output_series = np.array(predictions_ser
series_idx, :]
92         single_output_series = np.array([(1 if out[0] < (
out in one_hot_output_series)])
93
94         plt.subplot(2, 3, batch_series_idx + 2)
95         plt.cla()
96         plt.axis([0, truncated_backprop_length, 0, 2])
97         left_offset = range(truncated_backprop_length)
98         left_offset2 = range(echo_step,truncated_backprop
step)
99
100        label1 = "past values"
101        label2 = "True echo values"
102        label3 = "Predictions"
103        plt.plot(left_offset2, batchX[batch_series_idx,
"o--b", label=label1)
104        plt.plot(left_offset, batchY[batch_series_idx,
"x--b", label=label2)
105        plt.plot(left_offset, single_output_series*0.2-
```

```
    label=label3)
106
107     plt.legend(loc='best')
108     plt.draw()
109     plt.pause(0.0001)
```

函数中将输入的x序列、回声y序列和预测的序列同时打印到图像中。按照批次的个数生成图像。为了让3个序列看起来更明显，将其缩放0.2，并且调节每个图像的高度。同时将第一个原始序列的x在显示中滞后echo_step个序列，将3个图像放在同一序列顺序中比较。

运行代码，生成如下结果，如图9-9所示。

```
New data, epoch 0
Step 0 Loss 0.760327
Step 100 Loss 0.462219
Step 200 Loss 0.364076
.....
New data, epoch 4
Step 0 Loss 0.324354
Step 100 Loss 0.103451
Step 200 Loss 0.0894693
Step 300 Loss 0.0940791
Step 400 Loss 0.09462
Step 500 Loss 0.10184
Step 600 Loss 0.0910746
```

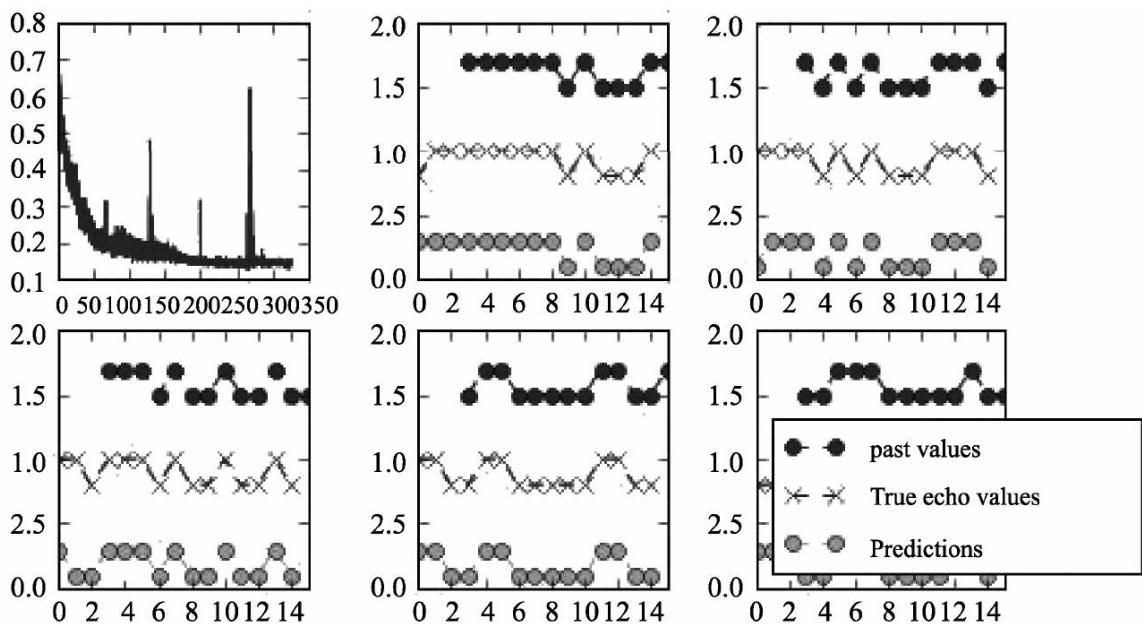


图9-9 RNN回声实例结果

最下面的点为预测的序列，中间的为回声序列，从图像中可以看到预测序列和回声序列几乎相同，表明RNN网络已经完全可以学到回声的规则。

9.3 循环神经网络（RNN）的改进

9.2节中演示的代码看似功能很强大，但也仅限于简单的逻辑和样本。对于相对较复杂的问题，这种RNN便会显出其缺陷，原因还是出在激活函数。通常来讲，激活函数在神经网络里最多只能6层左右，因为它的反向误差传递会随着层数的增加，传递的误差值越来越小，而在RNN中，误差传递不仅存在于层与层之间，也在存于每一层的样本序列间，所以RNN无法去学习太长的序列特征。

于是，神经网络学科中又演化了许多RNN网络的变体版本，使得模型能够学习更长的序列特征。接下来一起看看循环神经网络RNN的各种演化版本及内部原理与结构。

9.3.1 LSTM网络介绍

长短记忆的时间递归神经网络（Long Short Term Memory，LSTM）可以算是RNN网络的代表，其结构同样也非常复杂，下面一起来学习一下。

1. 整体介绍

LSTM是一种RNN特殊的类型，可以学习长

*****ebook converter DEMO Watermarks*****

期依赖信息。LSTM通过刻意的设计来避免长期依赖问题，其结构示意如图9-10所示。

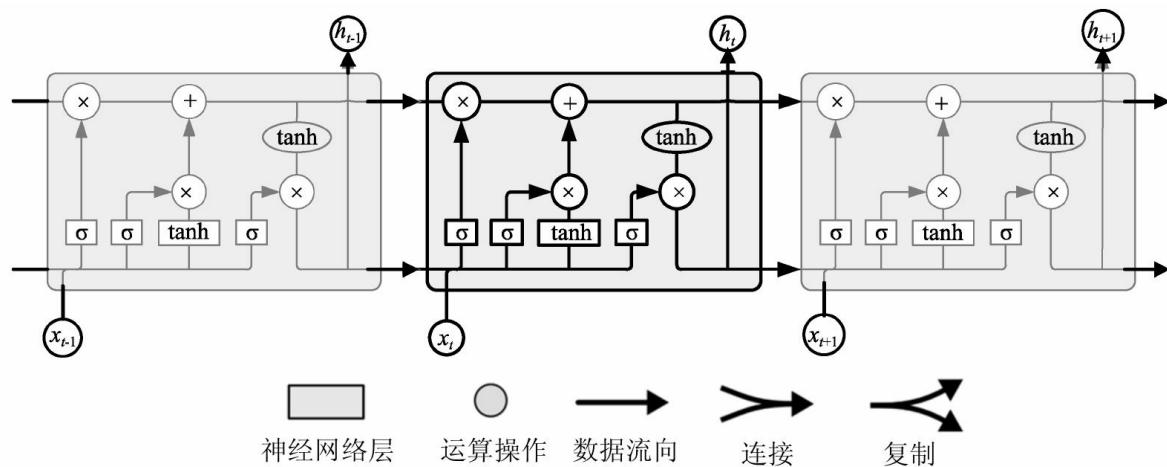


图9-10 LSTM结构示意

图9-10中，每一条黑线传输着一整个向量，从一个节点的输出到其他节点的输入。方框上方的圆圈代表运算操作（如向量的和），而中间的方框就是学习到的神经网络层。合在一起的线表示向量的连接，分开的线表示内容被复制，然后分发到不同的位置。

将其简化成图9-11，就与之前所说的结构一样了（这里的激活函数使用的是Tanh）。

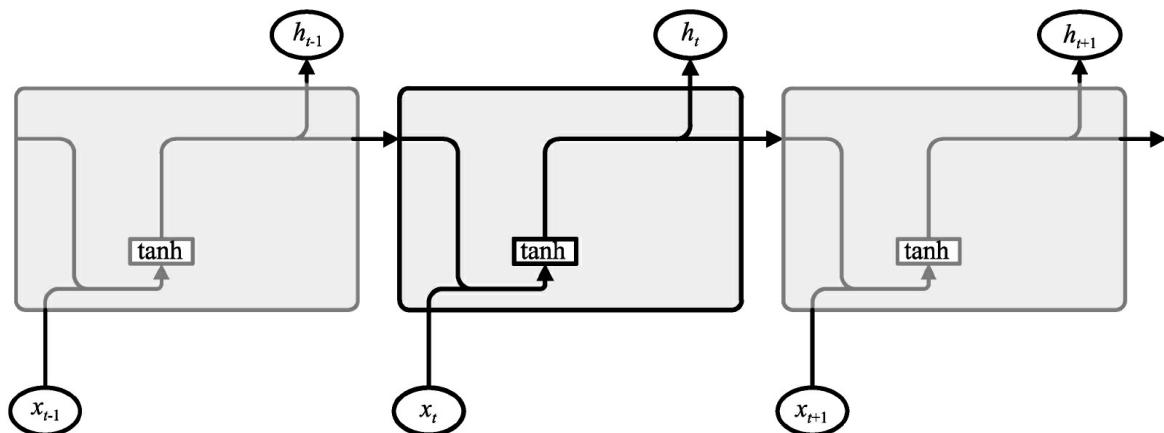


图9-11 LSTM2

这种结构的核心思想是引入了一个叫做细胞状态的连接，这个细胞状态用来存放想要记忆的东西（对应于简单RNN中的 h ，只不过这里面不再只存放上一次的状态了，而是通过网络学习存放那些有用的状态）。同时在里面加入3个门。

- 忘记门：决定什么时候需要把以前的状态忘记。
- 输入门：决定什么时候加入新的状态。
- 输出门：决定什么时候需要把状态和输入放在一起输出。

从字面意思可以看出，简单RNN只是把上一次的状态当成本次的输入一起输出。而LSTM在状态的更新和状态是否参与输入都做了灵活的选择，具体选什么，则一起交给神经网络的训练机制来训练。

现在分别介绍一下这三个门的结构和作用。

2. 忘记门

如图9-12所示为忘记门。该门决定模型会从细胞状态中丢弃什么信息。

该门会读取 h_{t-1} 和 x_t ，输出一个在0~1之间的数值给每个在细胞状态 C_{t-1} 中的数字。1表示“完全保留”，0表示“完全舍弃”。

例如一个语言模型的例子，假设细胞状态会包含当前主语的性别，于是根据这个状态便可以选择正确的代词。当我们看到新的主语时，应该把新的主语在记忆中更新。该门的功能就是先去记忆中找到以前那个旧的主语（并没有真正忘掉操作，只是找到而已）。

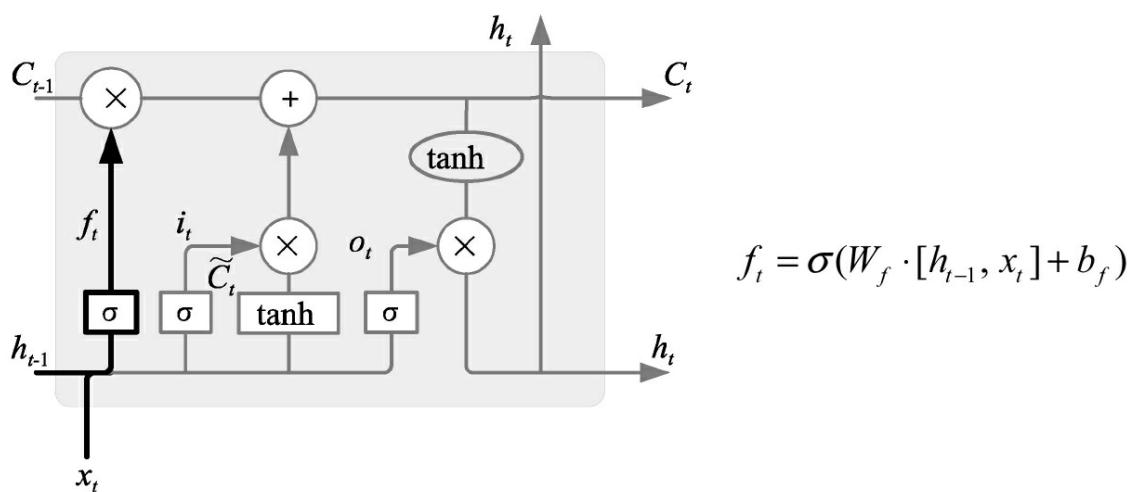


图9-12 LSTM忘记门

3. 输入门

输入门其实可以分成两部分功能，如图9-13所示。一部分是找到那些需要更新的细胞状态，另一部分是把需要更新的信息更新到细胞状态里。

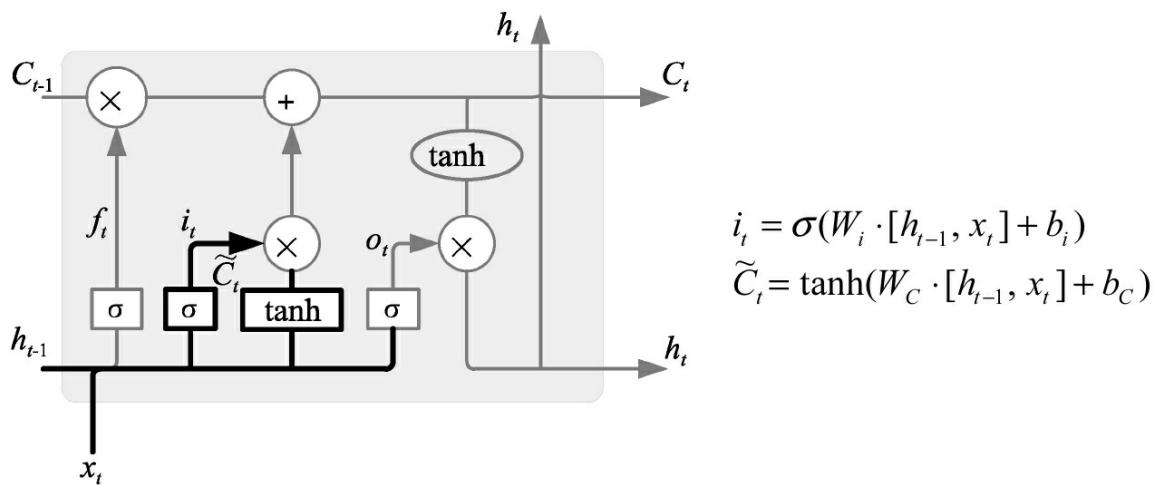


图9-13 输入门

其中， \tanh 层就是要创建一个新的细胞状态值向量—— C_t ，会被加入到状态中。

忘记门找到了需要忘掉的信息 f_t 后，再将它与旧状态相乘，丢弃掉确定需要丢弃的信息。再将结果加上 $i_t \times C_t$ 使细胞状态获得新的信息，这样就完成了细胞状态的更新，如图9-14所示。

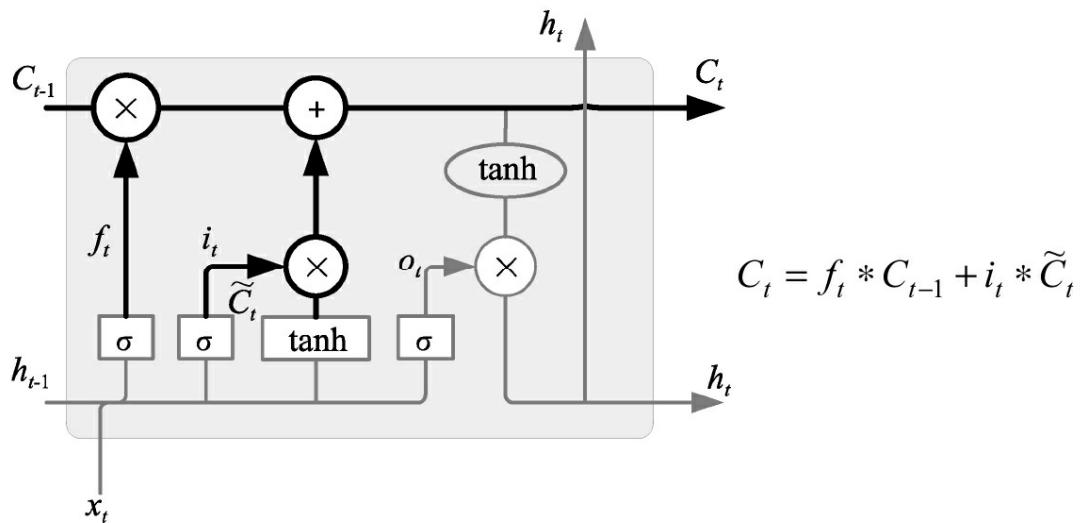


图9-14 输入门更新

4. 输出门

图9-15所示，在输出门中，通过一个Sigmoid层来确定哪部分的信息将输出，接着把细胞状态通过Tanh进行处理（得到一个在-1~1之间的值）并将它和Sigmoid门的输出相乘，得出最终想要输出的那部分，例如在语言模型中，假设已经输入了一个代词，便会计算出需要输出一个与动词相关的信息。

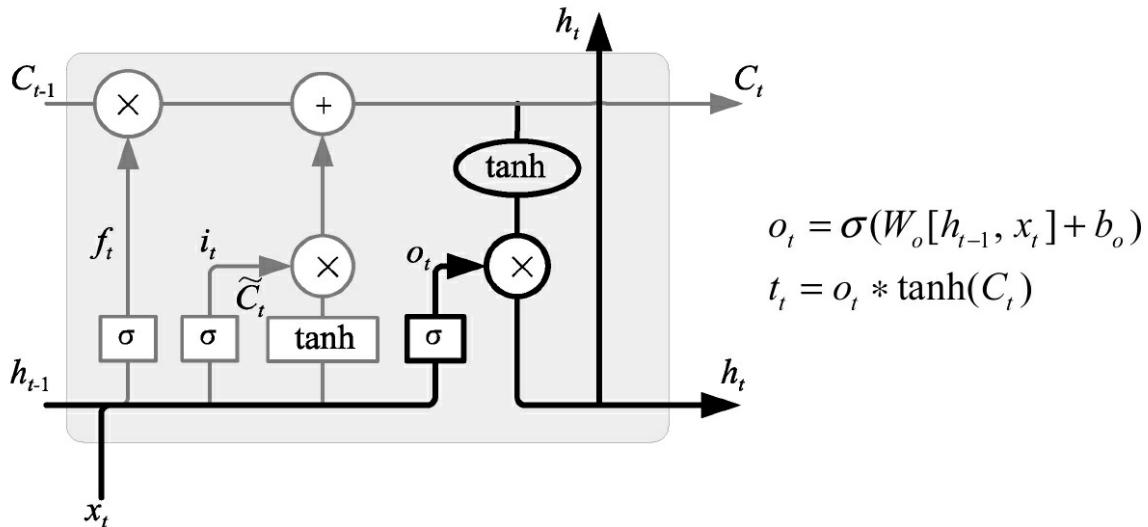


图9-15 输出门

9.3.2 窥视孔连接（Peephole）

窥视孔连接（Peephole）的出现是为了弥补忘记门一个缺点：当前cell的状态不能影响到Input Gate, Forget Gate在下一时刻的输出，使整个cell对上个序列的处理丢失了部分信息。所以增加了Peephole connections，如图9-16所示虚线部分。计算的顺序为：

(1) 上一时刻从cell输出的数据，随着本次时刻的数据一起输入Input Gate和Forget Gate。

(2) 将输入门和忘记门的输出数据同时输入cell中。

(3) cell出来的数据输入到当前时刻的Output Gate，也输入到下一时刻的input gate，

forget gate。

(4) Forget Gate输出的数据与cell激活后的数据一起作为整个Block的输出。

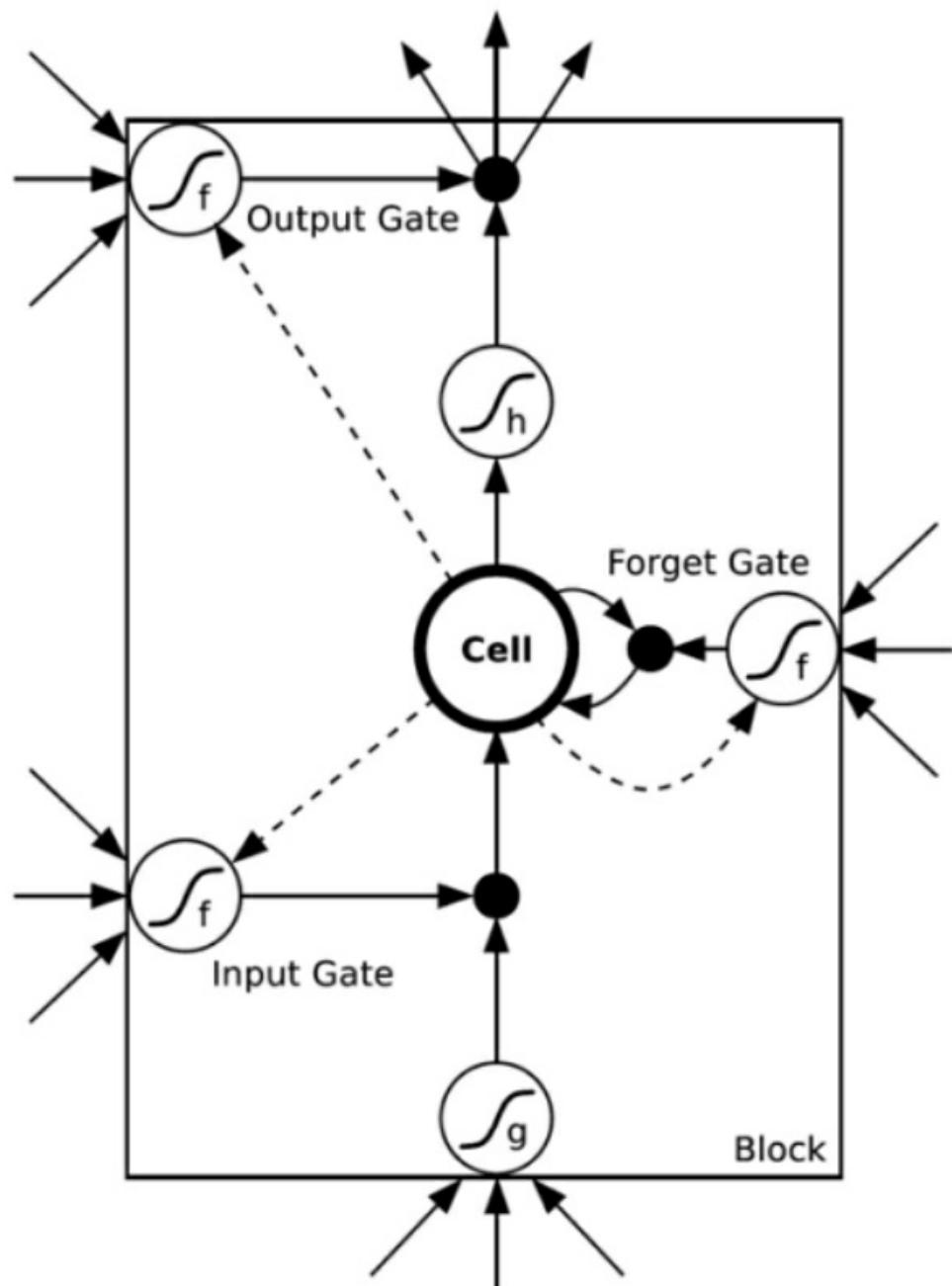


图9-16 Peephole逻辑

如图9-17所示为Peephole的详细结构。通过这样的结构，将Gate的输入部分增加了一个来源——Forget Gate，Input Gate的输入来源增加了cell前一时刻的输出，Output Gate的输入来源增加了cell当前时刻的输出，使cell对序列记忆增强。

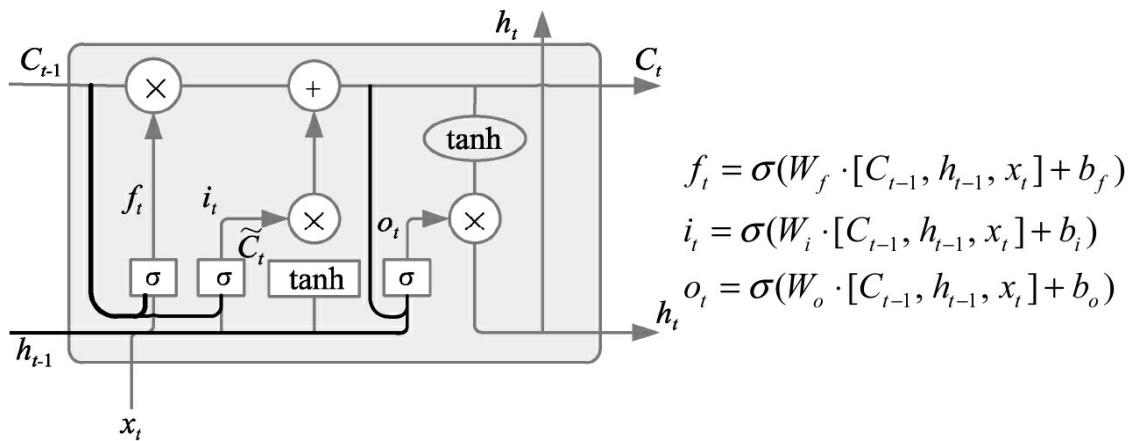


图9-17 Peephole的详细结构

9.3.3 带有映射输出的STMP

带有映射的LSTM (lstm with recurrent projection layer)，在原有LSTM基础之上增加了一个映射层 (projection layer)，并将这个layer连接到LSTM的输入，该映射层是通过全连接网络来实现的，可以通过改变其输出维度调节总的参数量，起到模型压缩的作用。

9.3.4 基于梯度剪辑的cell

基于梯度剪辑的cell (Clipping cell) 源于这

个问题：LSTM的损失函数是每一个时间点的RNN的输出和标签的交叉熵（cross-entropy）之和。这种loss在使用Backpropagation through time（BPTT）梯度下降法的训练过程中，可能会出现剧烈的抖动。

当参数值在较为平坦的区域更新时，由于该区域梯度值比较小，此时的学习率一般会变得较大，如果突然到达了陡峭的区域，梯度值陡增，再与此时较大的学习率相乘，参数就有很大幅度的更新，因此学习过程非常不稳定。

Clipping cell方法的使用可以优化这个问题，具体做法是：为梯度设置阈值，超过该阈值的梯度值都会被cut，这样参数更新的幅度就不会过大，因此容易收敛。

从原理上可以理解为：RNN和LSTM的记忆单元的相关运算是不同的，RNN中每一个时间点的记忆单元中的内容（隐藏层结点）都会更新，而LSTM则是使用忘记门机制将记忆单元中的值与输入值相加（按某种权值）再更新（cell状态），记忆单元中的值会始终对输出产生影响（除非Forget Gate完全关闭），因此梯度值易引起爆炸，所以Clipping功能是很有必要的。

9.3.5 GRU网络介绍

GRU是与LSTM功能几乎一样的另一个常用的网络结构，它将忘记门和输入门合成了一个单一的更新门，同样还混合了细胞状态和隐藏状态及其他一些改动。最终的模型比标准的LSTM模型要简单，如图9-18所示。

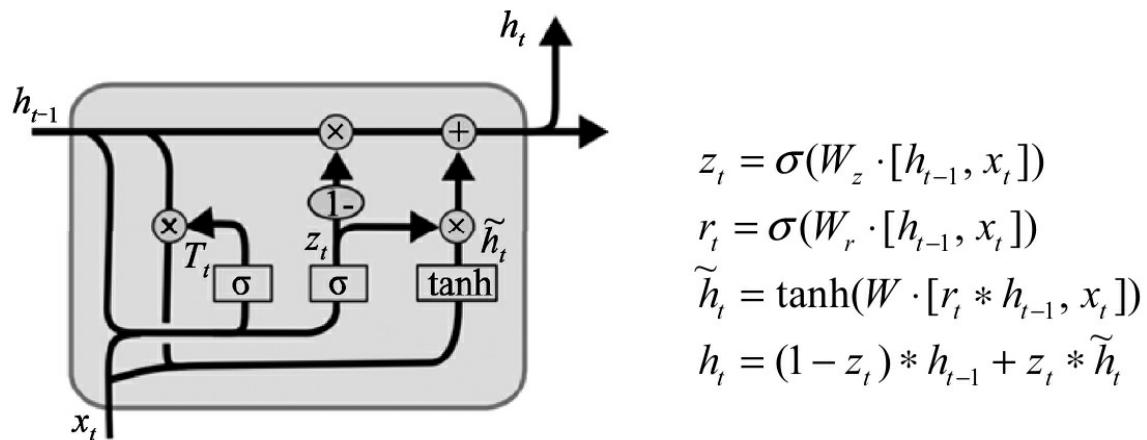


图9-18 GRU模型

当然，基于LSTM的变体不止GRU一个，并且经过一些专业人士的测试，它们在性能和准确度上几乎没什么差别，只是在具体的某些业务上会有略微不同。

由于GRU比LSTM少一个状态输出，效果几乎一样，因此在编码时使用GRU可以让代码更为简单一些。

9.3.6 Bi-RNN网络介绍

Bi-RNN又叫双向RNN，是采用了两个方向的RNN网络。

RNN网络擅长的是对于连续数据的处理，既然是连续的数据规律，我们不仅可以学习它的正向规律，还可以学习它的反向规律。这样将正向和反向结合的网络，会比单向的循环网络有更高的拟合度。例如，预测一个语句中缺失的词语，则需要根据上下文来进行预测。

双向RNN的处理过程与单向的RNN非常类似，就是在正向传播的基础上再进行一次反向传播，而且这两个都连接着一个输出层。这个结构提供给输出层输入序列中，每一个点完整的过去和未来的上下文信息。图9-19所示为一个沿着时间展开的双向循环神经网络。

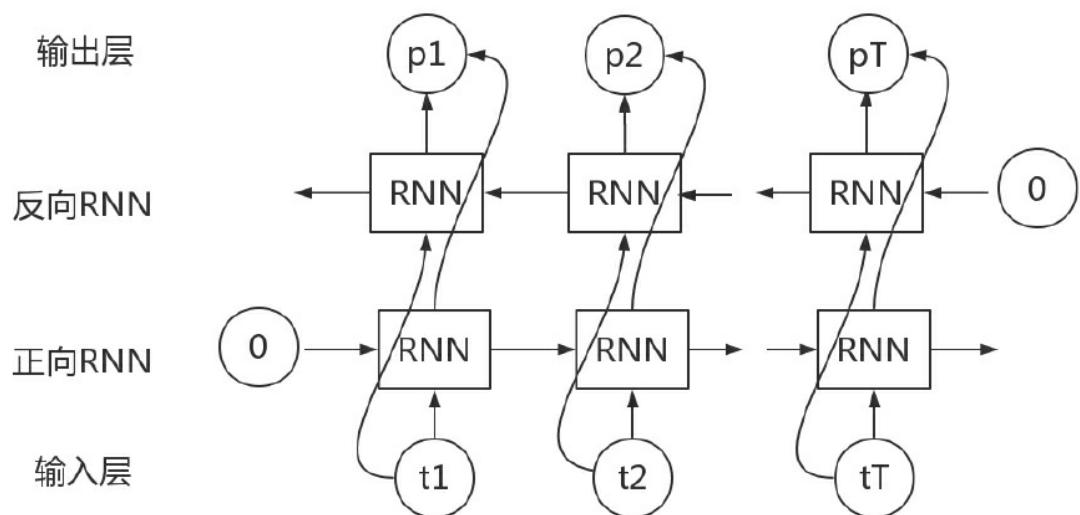


图9-19 一个沿着时间展开的双向循环神经网络

双向RNN会比单向RNN多一个隐藏层，6个独特的权值在每一个时步被重复利用，6个权值分别对应输入到向前和向后隐含层（ w_1, w_3 ），隐含层到隐含层自己（ w_2, w_5 ），向前和向后隐含

*****ebook converter DEMO Watermarks*****

层到输出层（w4， w6）。

双向PNN时序在神经网络里的时序步骤如图9-20所示。

在按照时间序列正向运算完之后，网络又从时间的最后一项反向地运算一遍，即把t3时刻的输入与默认值0一起生成反向的out3，把反向out3当成t2时刻的输入与原来的t2时刻输入一起生成反向out2；依此类推，直到第一个时序数据。



注意： 双向循环网络的输出是2个，正向一个，反向一个。最终会把输出结果通过concat并联在一起，然后交给后面的层来处理。例如，数据输入[batch, nhidden]，输出就会变成[batch, nhidden×2]。

在大多数应用里，基于时间序列与上下文有关的、类似NLP中自动回答类的问题，一般都是使用双向LSTM+LSTM/RNN横向扩展来实现的，效果非常好。

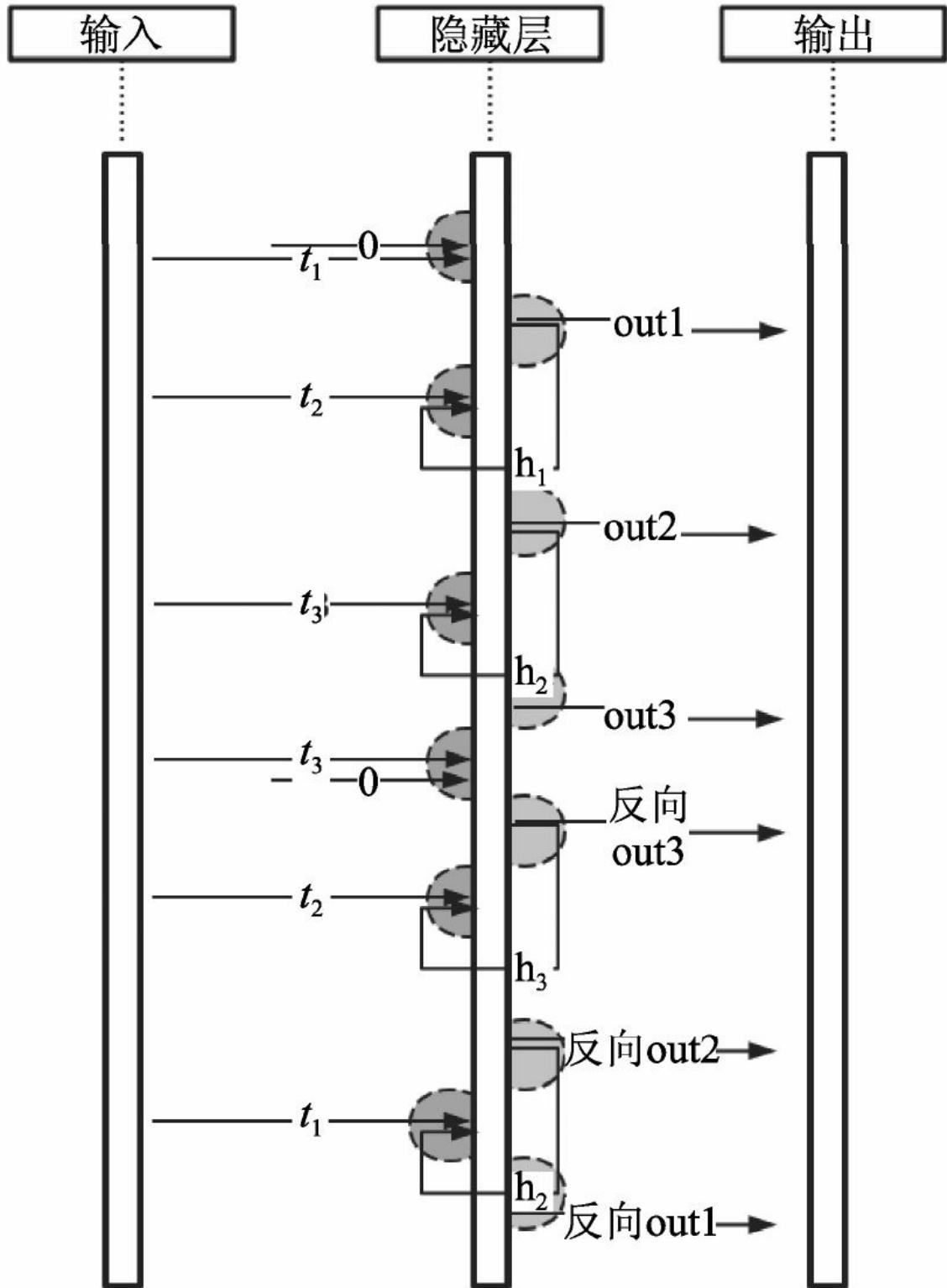


图9-20 双向RNN时序

9.3.7 基于神经网络的时序类分类CTC

*****ebook converter DEMO Watermarks*****

CTC (Connectionist Temporal Classification) 是语音辨识中的一个关键技术，通过增加一个额外的Symbol代表NULL来解决叠字问题。

RNN的优势是在处理连续的数据，在基于连续的时间序列分类任务中，常常会使用CTC的方法。

该方法主要体现在处理loss值上，通过对序列对不上的label添加blank（空label）的方式，将预测的输出值与给定的label值在时间序列上对齐，通过交叉熵的算法求出具体损失值。

比如在语音识别的例子中，对于一句语音有它的序列值及对应的文本，可以使用CTC的损失函数求出模型输出与label之间的loss，再通过优化器的迭代训练让损失值变小的方式将模型训练出来。

关于ctc_loss的算法细节，这里不做展开，后文还会有例子演示ctc_loss的真正用法。

9.4 TensorFlow实战RNN

在了解了RNN原理及类型之后，本节开始讲解在TensorFlow中如何构建RNN网络。

9.4.1 TensorFlow中的cell类

TensorFlow中定义了5个关于cell的类，具体定义如表9-1所示。

表9-1 cell类

cell 类	描述
BasicRNNCell def __init__(self, num_units, input_size=None, activation=tanh, reuse=None)	最基本的RNN类实现。 <ul style="list-style-type: none">• num_units: 包含cell的个数• input_size: 废弃
BasicLSTMCell def __init__(self, num_units, forget_bias=1.0, input_size=None, state_is_tuple=True, activation=tanh, reuse=None)	LSTM实现的一个basic版本： <ul style="list-style-type: none">• num_units: 包含cell的个数• state_is_tuple: 由于细胞状态状态ct和输出ht是分开记录，当为True时放在一个tuple中(c=array([[]]), h=array([[]]))，当为False时两个状态就按列连接起来，成为[batch, 2n]。一般建议都用True，该参数将被废弃• forget_bias: 添加到forget门的偏置• input_size: 被废弃的参数• reuse: 在一个scope里是否重用
LSTMCell def __init__(self, num_units, input_size=None, use_peepholes=False, cell_clip=None, initializer=None, num_proj=None, proj_clip=None, num_unit_shards=None, num_proj_shards=None, forget_bias=1.0, state_is_tuple=True, activation=tanh, reuse=None)	LSTM实现的一个高级版本： <ul style="list-style-type: none">• use_peepholes: 默认False, True表示启用Peephole连接• cell_clip: 是否在输出前对cell状态按照给定值进行截断处理• initializer: 指定初始化函数• num_proj: 通过projection层进行模型压缩的输出维度• proj_clip: 将num_proj按照给定的proj_clip截断
GRUCell def __init__(self, num_units, input_size=None, activation=tanh, reuse=None)	GRU类定义： num_units: 包含cell的个数 input_size: 废弃
MultiRNNCell def __init__(self, cells, state_is_tuple=True)	多层RNN的实现 <ul style="list-style-type: none">• cells: 一个cell列表，将列表中的cell一个个堆叠起来，如果使用cells=[cell1, cell2]，就是一共有2层，数据经过cell1后还要经过cell2

(续)

cell 类	描述
MultiRNNCell def __init__(self, cells, state_is_tuple=True)	<ul style="list-style-type: none">• state_is_tuple: 如果True则返回的是 n-tuple, 即, 将cell的输出值与cell的输出状态组成了一个tuple。其中, 输出值的结构为c=[batch_size, num_units], 输出状态的结构为h=[batch_size, num_units]

 **注意：** 在使用MultiRNNCell时，有些习惯写法是cells参数中直接用[cell] × n来代表创建n层的cell，这种写法如果不使用作用域隔离，则会报编译错误，或者使用一个外层循环将cell一个个

append进去来解决命名冲突。

9.4.2 通过cell类构建RNN

定义好cell类之后，还需要将它们连接起来构成RNN网络。TensorFlow中有几种现成的构建网络模式，是封装好的函数，直接调用即可，具体介绍如下。

1. 静态RNN构建

TensorFlow中提供了一个构建静态RNN的函数static_rnn，定义如下：

```
def static_rnn(cell, inputs, initial_state=None, dtype=None,
               length=None, scope=None):
```

具体参数说明如下。

- cell：生成好的cell类对象。
- inputs：输入数据，一定是list或者二维张量，list的顺序就是时间序列。元素就是每一个序列的值。
- initial_state：初始化cell状态。见9.4.14的详细介绍。

- dtype: 期望输出和初始化state的类型。
- sequence_length: 每一个输入的序列长度。
- scope: 命名空间。
 - 返回值有两个，一个是结果，一个是cell状态，我们只关注结果即可，结果也是一个list。输入是多少个时序，list里面就会输出多少个元素。



注意： TensorFlow中的这种定义很不友好，初学者极易出错。在输入时，一定要将我们习惯使用的张量改成list。另外，在得到输出时也要取结果中的最后一个元素参与后面的运算。

2. 动态RNN构建

关于动态RNN函数dynamic_rnn的定义如下：

```
def dynamic_rnn (cell, inputs, sequence_length=None, initial_state=None, parallel_iterations=None, swap_memory=False, time_major=False, scope=None):
```

具体参数说明如下。

- cell: 生成好的cell类对象。
- inputs: 输入数据，是一个张量，一般是三

维张量，[batch_size, max_time, ...]。其中batch_size表示一次的批次数量，max_time表示时间序列总数，后面是具体数据。

- initial_state：初始化cell状态。见9.4.14的详细介绍。
- dtype：期望输出和初始化state的类型。
- sequence_length：每一个输入的序列长度。
- time_major：为默认值False时，input的shape为[batch_size, max_time, ...]。如果是True，shape为[max_time, batch_size, ...]。
- scope：命名空间。
- 返回值：一个是结果，一个是cell状态，结果是以[batch_size, max_time, ...]形式的张量。



注意： 动态RNN也存在很多容易出错的地方，尤其在输出部分，它是以批次优先的矩阵。因为我们需要取最后一个时序的输出，所以需要转置成时间优先的形式。

3. 双向RNN构建

双向RNN作为一个可以学习正、反向规律的

循环神经网络，在TensorFlow中有4个函数可以使用，如表9-2所示。

表9-2 双向RNN函数

函 数	说 明
<code>tf.nn.bidirectional_dynamic_rnn (cell_fw,cell_bw, inputs, sequence_length=None, initial_state_fw= None,initial_state_bw=None,dtype=None, parallel_ iterations=None, swap_memory=False, time_ major=False,scope=None)</code>	<p>其中：</p> <ul style="list-style-type: none">• <code>cell_fw, cell_bw</code>: 前向和反向的rnn cell• <code>inputs</code>: 输入序列，一个张量输入，形状为`[batch_size, max_time, `...`], 或nested tuple等元素• <code>sequence_length</code>: 序列长度• <code>initial_state_fw, initial_state_bw</code>: 前向rnn_cell的初始状态, 反向rnn_cell的初始状态• 返回值: 是一个tuple (<code>outputs, output_state_fw, output_state_bw</code>), <code>outputs</code>也是tuple, (<code>output_fw,output_bw</code>),每一个值为一个张量 [batch_size, max_time, layers_output], 如果需要总的结果, 可以将前向后项的layers_output使用 <code>tf.concat</code>连接起来

函 数	说 明
<code>tf.contrib.rnn.static_bidirectional_rnn (cell_fw, cell_bw, inputs, initial_state_ fw=None, initial_state_bw=None, dtype=None, sequence_length=None, scope=None):</code>	<ul style="list-style-type: none"> • <code>cell_fw, cell_bw</code>: 这两个参数是实例化之后的cell，代表前向和后向，两个cell的结构必须一样 • <code>inputs</code>: 一个长度为t的输入列表，每一个都是一个形状的张量，形状为[batch_size, input_size]，或嵌套元组等元素 • <code>initial_state_fw, initial_state_bw</code>: 前向、后向的细胞状态初始化，默认为0 • <code>dtype</code>: 可以为自定义cell初始状态指定类型 • <code>sequence_length</code>: 传入的序列长度 • 返回值: 是一个tuple (outputs, output_state_fw, output_state_bw)，outputs为一个长度为t的list，每一个元素都包含有正、反向的输出
<code>tf.contrib.rnn.stack_bidirectional_rnn(cells_fw, cells_bw, inputs, initial_states_fw=None, initial_states_bw=None, dtype=None, sequence_length=None, scope=None)</code>	<p>创建一个多层双向网络。输出作为下一层的输入，前向和后向的输入大小必须一致，两个层之间是独立的，不能共享信息。</p> <ul style="list-style-type: none"> • <code>cells_fw, cells_bw</code>: 前向和后向 实例化之后的cell列表，正、反向的list长度必须相同（即具有同样深度），输入必须相同 • <code>inputs</code>: 一个长度t的输入列表，每一个都是一个形状的张量 • [batch_size, input_size]，或嵌套元组等元素; • <code>initial_states_fw, initial_states_bw</code>: 前向和后向的cell 初始化状态 • 返回值: 是一个tuple (outputs, output_state_fw, output_state_bw)，outputs为一个长度为t的list，每一个元素都包含有正、反向的输出
<code>tf.contrib.rnn.stack_bidirectional_dynamic_rnn (cells_fw, cells_bw, inputs, initial_states_fw=None, initial_states_bw=None, dtype=None, sequence_length=None, parallel_iterations=None, scope=None)</code>	<p>创建一个动态的多层双向RNN网络。输出作为下一层的输入，前向和后向的输入大小必须一致，两个层之间是独立的，不能共享信息。</p> <ul style="list-style-type: none"> • <code>cells_fw, cells_bw</code>: 前向和后向实例化之后的cell列表，正、反向的list长度必须相同（即具有同样深度），输入必须相同 • <code>inputs</code>: 一个张量输入，形状为: '[batch_size, max_time, ...]'，或nested tuple等元素 • <code>initial_states_fw, initial_states_bw</code>: 前向和后向的cell 初始化状态 • <code>parallel_iterations</code>: 要并行的迭代次数(默认为32)。对于没有任何时间依赖性的操作可以并行计算，并通过这个参数进行时间与空间的权衡。当该参数大于1时使用更多的内存，但占用的时间更少。相反取较小的值时，使用更少的内存，但是计算要花费更长的时间。 • 返回值: 是一个tuple (outputs, output_state_fw, output_state_bw)，outputs为一个张量[batch_size, max_time, layers_output]，layers_output包含tf.concat之后的正向和反向的输出

表9-2中，第一个函数是建立一个简单的双向RNN网络，两个方向各一个cell。第二个函数是

*****ebook converter DEMO Watermarks*****

建立多层的双向RNN，每个方向都是一个多层次cell。最后一个函数与第二个函数相同，只不过输入和输出是张量的形式。有了前面多层网络结构及卷积的基础之后，再理解LSTM将变得很容易。下面例子中仍然是对MNIST进行分类，这里只列出了核心部分，其他部分与原来一样，不再赘述。



注意： 在单层、多层次、双向RNN函数的介绍中，都有动态和静态之分。静态的意思就是按照样本的时间序列个数（n）展开，在图中创建（n）个序列的cell或cell中；动态的意思是只创建样本中一个序列的RNN，其他的序列数据都会通过循环来进入该RNN来运算。

通过静态生成的RNN网络，生成过程所需的时间会更长，网络所占有的内存会更多，导出的模型会更大。模型中会带有每个序列中间态的信息，利于调试。在使用时必须与训练时的样本序列个数相同。通过动态生成的RNN网络，所占用的内存较少，导出的模型较小。模型中只会有最后的状诚。在使用时还能支持不同的序列个数。

4. 使用动态RNN处理变长序列

动态RNN还有个更高级的功能就是可以处理变长序列，方法就是：在准备样本的同时，将样

本对应的长度也作为初始化参数，一起创建动态RNN。示例代码如下：

```
import tensorflow as tf
import numpy as np
tf.reset_default_graph()
# 创建输入数据
X = np.random.randn(2, 4, 5)

# 第二个样本长度为3
X[1,1:] = 0
seq_lengths = [4, 1]
#分别建立一个lstm与GRU的cell, 比较输出的状态
cell = tf.contrib.rnn.BasicLSTMCell(num_units=3, state_is_tuple=True)
gru = tf.contrib.rnn.GRUCell(3)

# 如果没有 initial_state, 必须指定 a dtype
outputs, last_states = tf.nn.dynamic_rnn(cell,X, seq_lengths)
gruoutputs, grulast_states = tf.nn.dynamic_rnn(gru,X,seq_lengths)
sess = tf.InteractiveSession()
sess.run(tf.global_variables_initializer())
result,sta ,gruout,grusta=sess.run([outputs,last_states,gruoutputs,grulast_states])

print("全序列: \n", result[0]) #对于全序列则输出正常长度的值
print("短序列: \n", result[1]) #对于短序列, 会为多余的序列长度补0
print('LSTM的状态: ',len(sta),'\n',sta[1]) #在初始化中设置了state_is_tuple为true, 所以lstm的状态, 为 (状态, 输出值)
print('GRU的短序列: \n',gruout[1])
print('GRU的状态: ',len(grusta),'\n',grusta[1]) #Gru没有状态输出
最终结果, 因为批次为两个, 所以输出为2
```

这种变成序列在运算之后，对于短序列会在输出结果后面补0，同时会把补0之前的最后输出放到状态里。例如上面代码执行后，会有如下输出：

全序列:

*****ebook converter DEMO Watermarks*****

```
[[ -0.01654044  0.01401587 -0.09957964]
 [-0.02326733  0.05380562 -0.00796815]
 [-0.01326877  0.26243431 -0.20821182]
 [-0.02425857  0.04418174 -0.2551933 ]]
```

短序列:

```
[[ 0.01152199  0.00987599  0.05193869]
 [ 0.          0.          0.          ]
 [ 0.          0.          0.          ]
 [ 0.          0.          0.          ]]
```

LSTM的状态: 2

```
[[ -0.02425857  0.04418174 -0.2551933 ]
 [ 0.01152199  0.00987599  0.05193869]]
```

GRU的短序列:

```
[[ 0.34744831 -0.0745199   0.04048231]
 [ 0.          0.          0.          ]
 [ 0.          0.          0.          ]
 [ 0.          0.          0.          ]]
```

GRU的状态: 2

```
[ 0.34744831 -0.0745199   0.04048231]
```

在源码中，批次的值为2，里面放了一个全序列样本与一个短序列样本。在输出的结果中，使用result[0]将全序列的结果输出，result[1]将短序列的结果输出。全序列与短序列两部分的输出均为4行3列的数组，其中3是由于有3个RNN单元，而4是源于全序列的长度为4。可以看到由于短序列长度为1，其输出结果中其他的3个序列自动补上了0。

动态RNN会将真实长度的最后输出放到状态里，直接从状态取值即可拿到结果。这里需要区分一下LSTM与GRU的状态取值方法。

- LSTM的状态：一般是一个元组（取决于state_is_tuple初始化时的参数设置），内容为（状态，输出值），取值时需要选择输出值对应的索

*****ebook converter DEMO Watermarks*****

引。

· GRU的状态：因为GRU本身没有状态输出，所以状态值即为输出值。如上面的代码通过打印grusta[1]的值（最后一行），直接可以得到短序列的最终输出值并在屏幕上打印出来。

9.4.3 实例57：构建单层LSTM网络对MNIST数据集分类

这里的输入 x 当成28个时间段，每段内容为28个值，使用unstack将原始的输入 28×28 调整成具有28个元素的list，每个元素为 1×28 的数组。这28个时序一次送入RNN中，如图9-21所示。

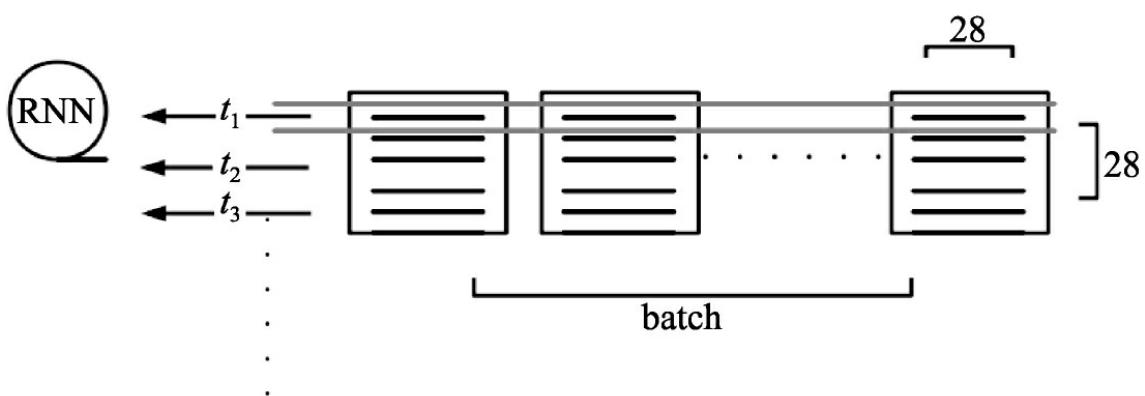


图9-21 LSTM例子

由于是批次操作，所以每次都取该批次中所有图片的一行作为一个时间序列输入。

理解了这个转换之后，构建网络就变得很容易了，先建立一个包含128个cell的类lstm_cell，

*****ebook converter DEMO Watermarks*****

然后将变形后的x1放进去生成节点outputs，最后通过全连接生成pred，最后使用softmax进行分类。

实例描述

演示使用单层LSTM网络对MNIST数据集分类。

代码9-3 LSTM Mnist

```
import tensorflow as tf
# 导入MINST data
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("/data/", one_hot=True)

n_input = 28          #MNIST data 输入(img shape: 28*28)
n_steps = 28          #序列个数
n_hidden = 128         #隐藏层个数
n_classes = 10          #MNIST 分类个数 (0~9 digits)
#定义占位符
x = tf.placeholder("float", [None, n_steps, n_input])
y = tf.placeholder("float", [None, n_classes])

x1 = tf.unstack(x, n_steps, 1)
lstm_cell = tf.contrib.rnn.BasicLSTMCell(n_hidden, forget_bias=1.0)
outputs, states = tf.contrib.rnn.static_rnn(lstm_cell, x1,
                                            dtype=tf.float32)
pred = tf.contrib.layers.fully_connected(outputs[-1], n_classes,
                                         activation_fn = None)
.....
```

运行上面代码，结果如下：

```
Extracting /data/train-images-idx3-ubyte.gz
Extracting /data/train-labels-idx1-ubyte.gz
Extracting /data/t10k-images-idx3-ubyte.gz
```

*****ebook converter DEMO Watermarks*****

```
Extracting /data/t10k-labels-idx1-ubyte.gz
Iter 1280, Minibatch Loss= 1.957660, Training Accuracy= 0.35
Iter 2560, Minibatch Loss= 1.633594, Training Accuracy= 0.46
.....
Iter 98560, Minibatch Loss= 0.156201, Training Accuracy= 0.94
Iter 99840, Minibatch Loss= 0.170062, Training Accuracy= 0.94
Finished!
Testing Accuracy: 0.945
```

本例中用到了BasicLSTMCell类，还可以使用LSTMCell类，将类名换一下即可，见本书附带资源中的代码“9-4 LSTMCell.py”文件。

9.4.4 实例58：构建单层GRU网络对MNIST数据集分类

GRU的实现与LSTM几乎一样，修改该前面的代码“9-3 LSTMMnist.py”文件，将LSTMCell换成GRUCell，同时去掉参数和返回值。

实例描述

演示使用单层GRU网络对MNIST数据集分类。

代码9-5 gru

```
.....
gru = tf.contrib.rnn.GRUCell(n_hidden)
outputs = tf.contrib.rnn.static_rnn(gru, x1, dtype=tf.float32)
.....
```

由于GRU只有一个输出，所以创建起来没有state_is_tuple参数。

9.4.5 实例59：创建动态单层RNN网络对MNIST数据集分类

本例中将静态RNN函数改成动态RNN函数即可，将上面的代码“9-5 gru.py”修改如下。

实例描述

演示使用单层动态RNN网络对MNIST数据集分类。

代码9-6 创建动态RNN

```
....  
gru = tf.contrib.rnn.GRUCell(n_hidden)  
  
# 创建动态RNN  
outputs,_ = tf.nn.dynamic_rnn(gru,x,dtype=tf.float32)  
outputs = tf.transpose(outputs, [1, 0, 2])  
pred = tf.contrib.layers.fully_connected(outputs[-1],n_classes,  
activation_fn = None)  
....
```

上面代码中，输入不再是转成list的x1，而是x，输出的outputs也通过transpose做了一次转置。

transpose中的第二参数[1, 0, 2]的意思是将[batch_size, max_time,]中的第1维

batch_size放在前面，第0维max_time放在后面，而第2维的数据不变。按照这些要求，在数据集变为[max_time, batch_size,]之后，取最后一个时间序列时得到的就是[batch_size,]了。



注意：对于输出是张量形式的RNN对结果处理先转置，再取最后一条，这是一个常用的技巧。

多层RNN在创建过程中，需要使用到前面介绍的MultiRNNCell类，这个类的实例化需要通过单层的cell对象输入。

与前面的例子类似，先创建单层的cell，然后再创建MultiRNNCell对象，在创建好MultiRNNCell后，可以通过静态或动态的RNN网络建立方式将网络组合起来。

9.4.6 实例60：静态多层LSTM对MNIST数据集分类

修改该前面的代码“9-3 LSTMmnist.py”例子代码如下：通过一个循环来建立3个LSTM的cell并放在list列表变量stacked_rnn里，然后实例化MultiRNNCell对象得到mcell。用unstack将输入的x转成list，输入到static_rnn函数里，返回值的结果再接一个全连接层进行softmax分类。

实例描述

演示使用静态多层LSTM网络对MNIST数据集分类。

代码9-7 McellMNIST

```
.....  
x = tf.placeholder("float", [None, n_steps, n_input])  
y = tf.placeholder("float", [None, n_classes])  
  
stacked_rnn = []  
for i in range(3):  
    stacked_rnn.append(tf.contrib.rnn.LSTMCell(n_hidden))  
mcell = tf.contrib.rnn.MultiRNNCell(stacked_rnn)  
  
x1 = tf.unstack(x, n_steps, 1)  
outputs, states = tf.contrib.rnn.static_rnn(mcell, x1, dtype=tf.float32)  
pred = tf.contrib.layers.fully_connected(outputs[-1], n_classes,  
activation_fn = None)  
  
learning_rate = 0.001  
# 定义loss和优化器  
cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(  
logits=pred, labels=y))  
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)  
minimize(cost)  
  
# 评估模型节点  
correct_pred = tf.equal(tf.argmax(pred,1), tf.argmax(y,1))  
accuracy = tf.reduce_mean(tf.cast(correct_pred, tf.float32))  
  
training_iters = 100000  
display_step = 10  
  
# 启动session  
with tf.Session() as sess:  
    sess.run(tf.global_variables_initializer())  
.....
```

9.4.7 实例61：静态多层RNN-LSTM连接GRU对MNIST数据集分类

MultiRNNCell类的功能就是将多个RNN连接在一起，在前面的例子中将3个一样的LSTM连在了一起，其中这些RNN可以是不同的类型。这个就相当于前面讲过的MLP（多层神经网络）中的神经元节点一样。

下面的例子就要将LSTM连接到GRU网络上输出。代码如下。

实例描述

演示使用静态多层LSTM网络对MNIST数据集分类。

代码9-8 mcellLSTMGRU

```
gru = tf.contrib.rnn.GRUCell(n_hidden*2)
lstm_cell = tf.contrib.rnn.LSTMCell(n_hidden)
mcell = tf.contrib.rnn.MultiRNNCell([lstm_cell, gru])
```

上面的代码只是把循环生成的LSTM换成由LSTM与GRU组成的list即可，为了演示两个cell的无关性，特意将GRU的cell设成了 $n_hidden \times 2$ 个，LSTM的cell设成 n_hidden 个，当然最终输出以最后一个节点为主，就是一个具有28个元素的list，

每个元素为[batch_size, n_hidden×2]。如果想要生成更多层的网络结构，直接在list里添加RNN的cell即可。

9.4.8 实例62：动态多层RNN对MNIST数据集分类

本例与动态单层一样使用dynamic_rnn函数，改写上面代码如下。

实例描述

演示使用动态多层RNN网络对MNIST数据集分类。

代码9-9 动态多层

```
outputs,states = tf.nn.dynamic_rnn(mcell,x,dtype=tf.float32  
256)  
outputs = tf.transpose(outputs, [1, 0, 2])  
#(28, ?, 256) 28个时序，取最后一个时序outputs[-1]=(?,256)  
pred = tf.contrib.layers.fully_connected(outputs[-1],n_classes,  
activation_fn = None)
```

将输入改成x，同时将输出的结果进行tf.transpose(outputs, [1, 0, 2])的转置处理，取outputs[-1]放到下一层里参与运算。

9.4.9 练习题

*****ebook converter DEMO Watermarks*****

本书附带资源中有4个代码文件——“9-10 LSTM改错.py”“9-11 lstm改错1.py”“9-12 GRU 改错2.py”“9-13 LSTM改错3.py”，分别有不同的错误，请将这些错误找出来使程序正常运行。

9.4.10 实例63：构建单层动态双向RNN对MNIST数据集分类

先建立两个包含128个正反向cell的类lstm_fw_cell、lstm_bw_cell，然后使用tf.nn.bidirectional_dynamic_rnn函数将x放进去生成节点outputs，由于bidirectional_dynamic_rnn的输出结果与状态是分离的，所以需要手动将结果合并起来并进行转置，然后通过全连接生成pred，再使用softmax进行分类。代码如下。

实例描述

演示使用单层动态双向RNN网络对MNIST数据集分类。

代码9-14 BiRNNMnist

```
import tensorflow as tf
from tensorflow.contrib import rnn
# 导入MINST data
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("/data/", one_hot=True)

# 定义参数
learning_rate = 0.001
```

*****ebook converter DEMO Watermarks*****

```

training_iters = 100000
batch_size = 128
display_step = 10

# 网络模型参数设置
n_input = 28                      # MNIST data 输入(img shape: 28x28)
n_steps = 28                         # 序列个数
n_hidden = 128                       # 隐藏层节点个数
n_classes = 10                        # MNIST 分类数 (0~9 digits)

tf.reset_default_graph()

# 定义占位符
x = tf.placeholder("float", [None, n_steps, n_input])
y = tf.placeholder("float", [None, n_classes])

x1 = tf.unstack(x, n_steps, 1)
lstm_fw_cell = rnn.BasicLSTMCell(n_hidden, forget_bias=1.0)
# 反向cell
lstm_bw_cell = rnn.BasicLSTMCell(n_hidden, forget_bias=1.0)
outputs, output_states = tf.nn.bidirectional_dynamic_rnn(lstm_fw_cell,
lstm_bw_cell, x,
                                         dtype=tf.float32)
print(len(outputs), outputs[0].shape, outputs[1].shape)
outputs = tf.concat(outputs, 2)
outputs = tf.transpose(outputs, [1, 0, 2])

pred = tf.contrib.layers.fully_connected(outputs[-1], n_classes,
activation_fn = None)
.....
```

运行代码后，输出的outputs类型如下：

```
2 (?, 28, 128) (?, 28, 128)
```

可以再次证明，输出的outputs是前向和后向分开的。这种方法最原始也最灵活，但要注意，一定要把两个输出结果进行融合（也可以不用concat）。因为后面实例的方法输出的都是concat

*****ebook converter DEMO Watermarks*****

之后的结果，不需要再额外考虑融合操作。

9.4.11 实例64：构建单层静态双向RNN对MNIST数据集分类

静态双向RNN的建立是使用static_bidirectional_rnn函数，先建立两个包含128个正反向cell的类lstm_fw_cell、lstm_bw_cell，然后将变形后的x1放进去生成节点outputs，再通过全连接生成pred，最后使用softmax进行分类。代码如下。

实例描述

演示使用单层静态双向RNN网络对MNIST数据集分类。

代码9-15 单层静态双向rnn

```
import tensorflow as tf
from tensorflow.contrib import rnn
# 输入MINST data
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("/data/", one_hot=True)

# 定义参数
learning_rate = 0.001
training_iters = 1000000
batch_size = 128
display_step = 10

# 网络模型参数设置
n_input = 28                      #MNIST数据输入 (img shape:
n_steps = 28                         #步骤序列
```

*****ebook converter DEMO Watermarks*****

```
n_hidden = 128 #隐藏层个数
n_classes = 10 #MNIST总类别(0~9 digits)

tf.reset_default_graph()

#定义占位符
x = tf.placeholder("float", [None, n_steps, n_input])
y = tf.placeholder("float", [None, n_classes])

x1 = tf.unstack(x, n_steps, 1)
lstm_fw_cell = rnn.BasicLSTMCell(n_hidden, forget_bias=1.0)
# 反向cell
lstm_bw_cell = rnn.BasicLSTMCell(n_hidden, forget_bias=1.0)
outputs, _, _ = rnn.static_bidirectional_rnn(lstm_fw_cell, lstm_bw_cell,
                                             x1, sequence_length=n_steps,
                                             dtype=tf.float32)
print(outputs[0].shape, len(outputs))
pred = tf.contrib.layers.fully_connected(outputs[-1], n_classes,
                                         activation_fn = None)
.....
```

运行代码，输出结果如下：

```
Extracting /data/train-images-idx3-ubyte.gz
Extracting /data/train-labels-idx1-ubyte.gz
Extracting /data/t10k-images-idx3-ubyte.gz
Extracting /data/t10k-labels-idx1-ubyte.gz
(?, 256) 28
Iter 1280, Minibatch Loss= 2.142399, Training Accuracy= 0.30
Iter 2560, Minibatch Loss= 1.830110, Training Accuracy= 0.31
Iter 3840, Minibatch Loss= 1.613333, Training Accuracy= 0.40
.....
Iter 96000, Minibatch Loss= 0.114890, Training Accuracy= 0.95
Iter 97280, Minibatch Loss= 0.159568, Training Accuracy= 0.95
Iter 98560, Minibatch Loss= 0.168179, Training Accuracy= 0.95
Iter 99840, Minibatch Loss= 0.089507, Training Accuracy= 0.95
Finished!
Testing Accuracy: 0.992188
```

在输出过程中，我们将outputs的shape打印了出来，可以看到是个长度为28的list，每个元素为[batch_size, 2×n_hidden]。双向RNN将输出两倍

*****ebook converter DEMO Watermarks*****

的结果。

9.4.12 实例65：构建多层双向RNN对MNIST数据集分类

修改该前面的“9-15单层静态双向rnn.py”例子：将static_bidirectional_rnn换成stack_bidirectional_rnn，并将前、后向中的lstm_fw_cell和lstm_bw_cell用中括号扩起来。这样就用stack_bidirectional_rnn生成了正反各带有一层RNN的双向RNN网络。如果想再增加层，需要在中括号里接着添加即可。代码如下。

实例描述

演示使用多层双向RNN网络对MNIST数据集分类。

代码9-16 多层双向RNN

```
outputs, _, _ = rnn.stack_bidirectional_rnn([lstm_fw_cell], [lstm_bw_cell],  
                                         inputs, sequence_length, initial_state_fw=  
                                         initial_state_bw, dtype=tf.float32)
```

也可以用循环方式生成多个RNN放到list里，代码如下。

代码9-17 list多层次双向RNN

*****ebook converter DEMO Watermarks*****