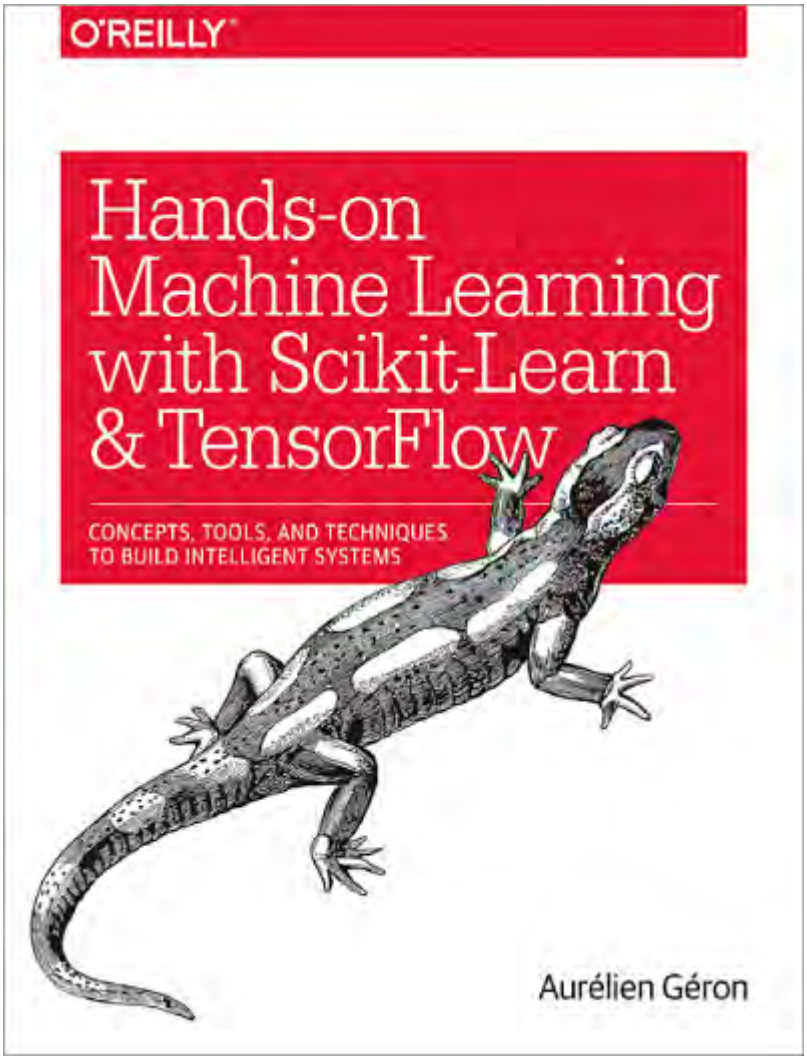
Scikit-Learn 与 TensorFlow 机器学习实用指南 中文精要



译者：

- 1：SeanCheney
- 2-8：fjl_csdn
- 9-14：akon_wang_hkbu

下载本书和代码：https://www.jianshu.com/p/9efbae6dbf8e

本书自2017年4月9日出版，便长期占据美国亚马逊Computer Science前三，评分4.6，几乎80%是给的5星好评。CS类里面其实有许多科普书，真正的技术类书籍里面，就是这本和《Deep Learning》分享第一和第二。

作者Aurélien Géron，法国人，毕业于AgroParisTech（1994-1997），曾任Google Youtube视频分类项目组负责人，创建过多家公司并担任CTO，也曾在AgroParisTech担任讲师。

一、机器学习概览

大多数人听到“机器学习”，往往会在脑海中勾勒出一个机器人：一个可靠的管家，或是一个可怕的终结者，这取决于你问的是谁。但是机器学习并不是未来的幻想，它已经来到我们身边了。事实上，一些特定领域已经应用机器学习几十年了，比如光学字符识别 （Optical Character Recognition，OCR）。但是直到1990年代，第

一个影响了数亿人的机器学习应用才真正成熟，它就是垃圾邮件过滤器（spam filter）。虽然并不是一个有自我意识的天网系统（Skynet），垃圾邮件过滤器从技术上是符合机器学习的（它可以很好地进行学习，用户几乎不用再标记某个邮件为垃圾邮件）。后来出现了更多的数以百计的机器学习产品，支撑了更多你经常使用的产品和功能，从推荐系统到语音识别。

机器学习的起点和终点分别是什么呢？确切的讲，机器进行学习是什么意思？如果我下载了一份维基百科的拷贝，我的电脑就真的学会了什么吗？它马上就变聪明了吗？在本章中，我们首先会澄清机器学习到底是什么，以及为什么你要使用它。

然后，在我们出发去探索机器学习新大陆之前，我们要观察下地图，以便知道这片大陆上的主要地区和最明显的地标：监督学习vs非监督学习，线上学习vs批量学习，基于实例vs基于模型学习。然后，我们会学习一个典型的机器学习项目的工作流，讨论可能碰到的难点，以及如何评估和微调一个机器学习系统。

这一章介绍了大量每个数据科学家需要牢记在心的基础概念（和习语）。第一章只是概览（唯一不含有代码的一章），相当简单，但你要确保每一点都搞明白了，再继续进行学习本书其余章节。端起一杯咖啡，开始学习吧！

> 提示：如果你已经知道了机器学习的所有基础概念，可以直接翻到第2章。如果你不确信，可以尝试回答本章末尾列出的问题，然后再继续。

什么是机器学习？

机器学习是通过编程让计算机从数据进行学习的科学（和艺术）。

下面是一个更广义的概念：

机器学习是让计算机具有学习的能力，无需进行明确编程。 —— 亚瑟·萨缪尔，1959

和一个工程性的概念：

计算机程序利用经验E学习任务T，性能是P，如果针对任务T的性能P随着经验E不断增长，则称为机器学习。—— 汤姆·米切尔，1997

例如，你的垃圾邮件过滤器就是一个机器学习程序，它可以根据垃圾邮件（比如，用户标记的垃圾邮件）和普通邮件（非垃圾邮件，也称作ham）学习标记垃圾邮件。用来进行学习的样例称作训练集。每个训练样例称作训练实例（或样本）。在这个例子中，任务T就是标记新邮件是否是垃圾邮件，经验E是训练数据，性能P需要定义：例如，可以使用正确分类的比例。这个性能指标称为准确率，通常用在分类任务中。

如果你下载了一份维基百科的拷贝，你的电脑虽然有了很多数据，但不会马上变得聪明起来。因此，这不是机器学习。

为什么使用机器学习？

思考一下，你会如何使用传统的编程技术写一个垃圾邮件过滤器（图1-1）：

1. 你先观察下垃圾邮件一般都是什么样子。你可能注意到一些词或短语（比如4U、credit card、free、amazing）在邮件主题中频繁出现，也许还注意到发件人名字、邮件正文的格式，等等。
2. 你为观察到的规律写了一个检测算法，如果检测到了这些规律，程序就会标记邮件为垃圾邮件。
3. 测试程序，重复第1步和第2步，直到满足要求。

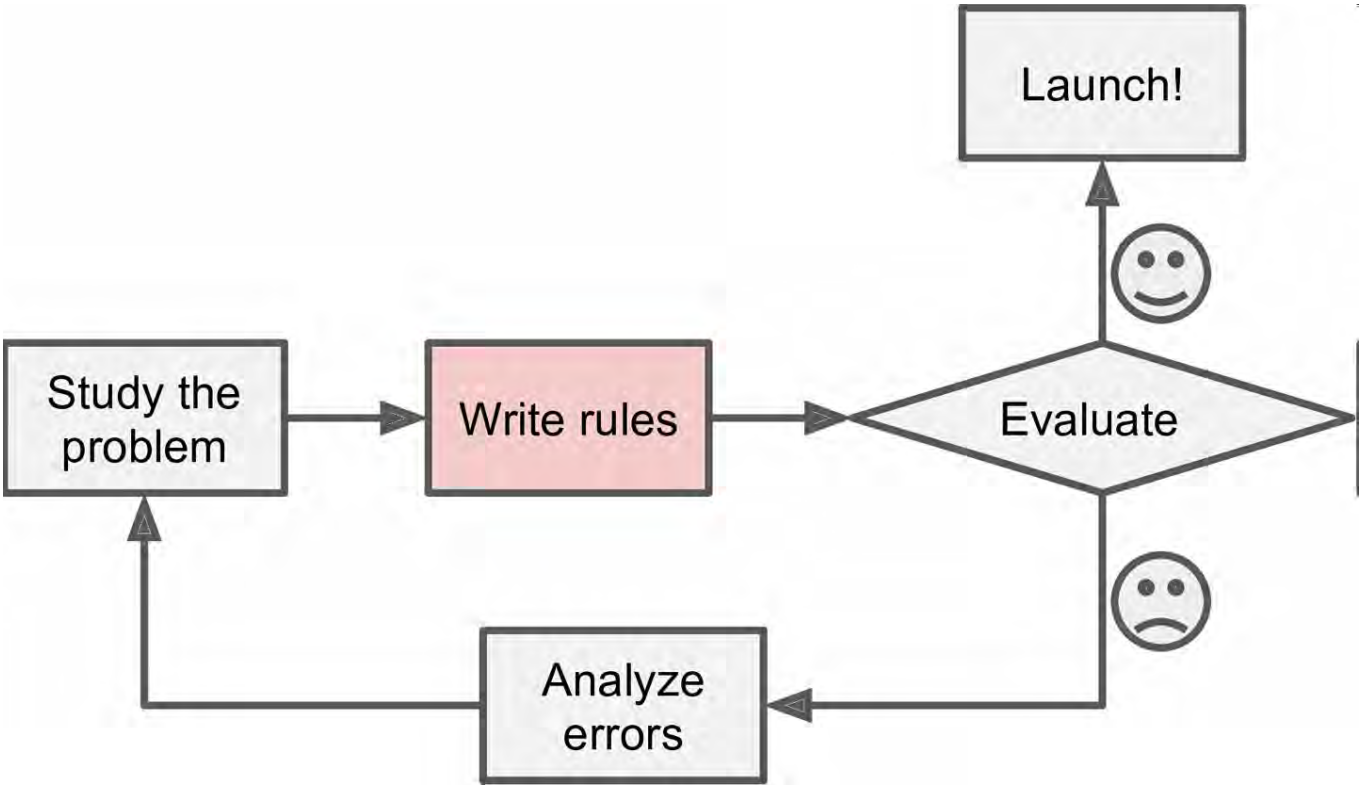


图1-1 传统方法

这个问题并不简单，你的程序很可能会变成一长串复杂的规则—— 这样就会很难维护。

相反的，基于机器学习技术的垃圾邮件过滤器会自动学习哪个词和短语是垃圾邮件的预测值，通过与普通邮件比较，检测垃圾邮件中反常频次的词语格式（图1-2）。这个程序短得多，更易维护，也更精确。

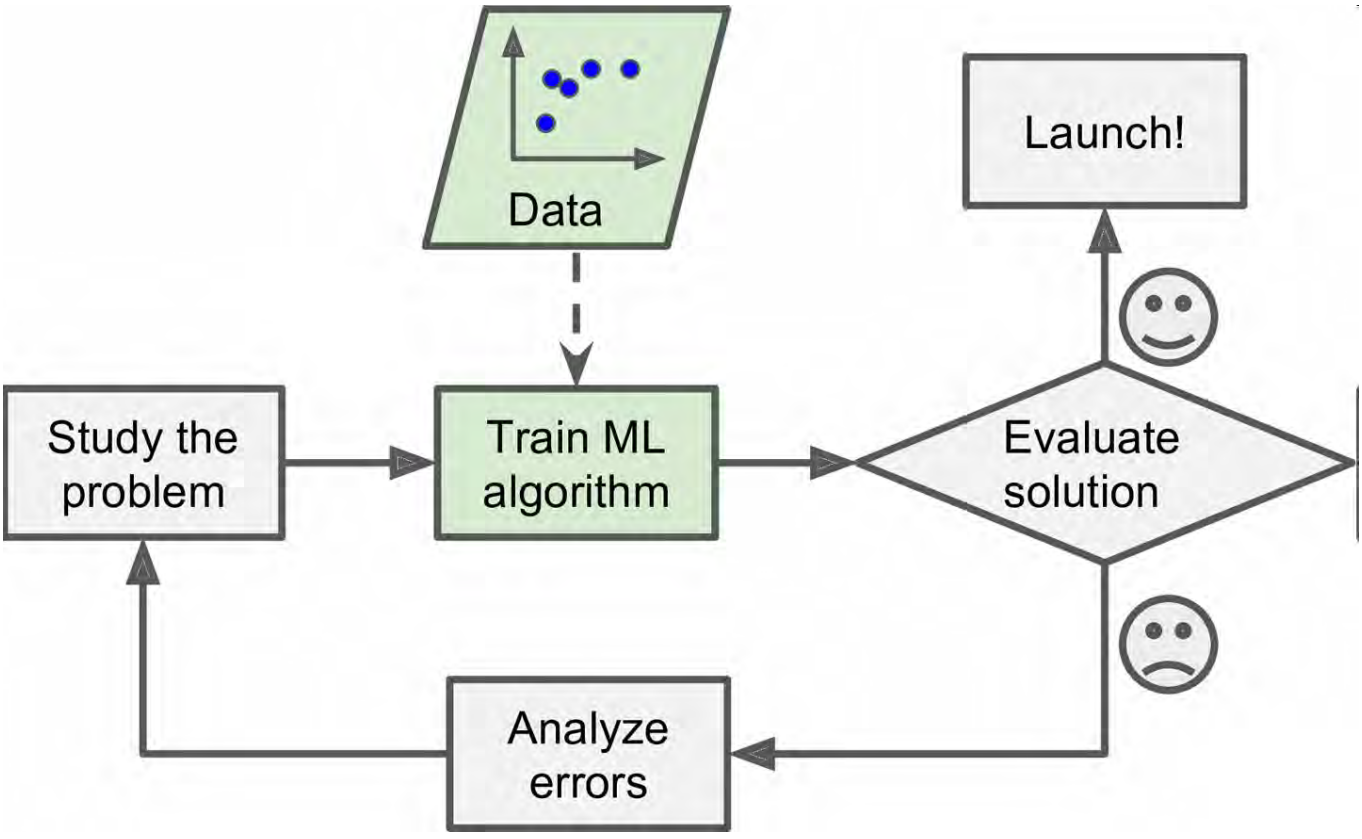


图1-2 机器学习方法

进而，如果发送垃圾邮件的人发现所有包含“4U”的邮件都被屏蔽了，可能会转而使用“For U”。使用传统方法的垃圾邮件过滤器需要更新以标记“For U”。如果发送垃圾邮件的人持续更改，你就需要迟勋写入新规则。

相反的，基于机器学习的垃圾邮件过滤器会自动注意到“For U”在用户手动标记垃圾邮件中的反常频繁性，然后就能自动标记垃圾邮件而无需干预了（图1-3）。

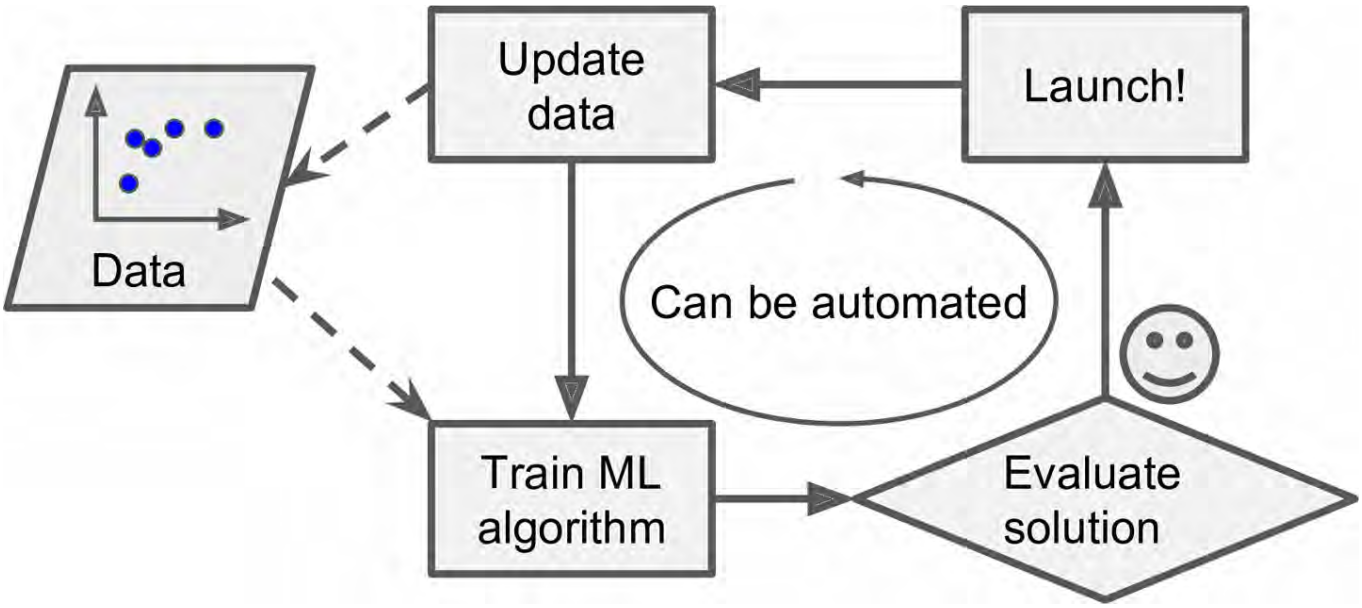


图1-3 自动适应改变

机器学习的另一个优点是善于处理对于传统方法太复杂或是没有已知算法的问题。例如，对于语言识别：假如想写一个可以识别“one”和“two”的简单程序。你可能注意到“two”起始是一个高音（“T”），所以可以写一个可以测量高音强度的算法，用它区分one和two。很明显，这个方法不能推广到嘈杂环境下的数百万人的数千词汇、数十种语言。（现在）最佳的方法是根据大量单词的录音，写一个可以自我学习的算法。

最后，机器学习可以帮助人类进行学习（图1-4）：可以检查机器学习算法已经掌握了什么（尽管对于某些算法，这样做会有点麻烦）。例如，当垃圾邮件过滤器被训练了足够多的垃圾邮件，就可以用它列出垃圾邮件预测值的单词和单词组合列表。有时，可能会发现不引人关注的关联或新趋势，有助于对问题更好的理解。

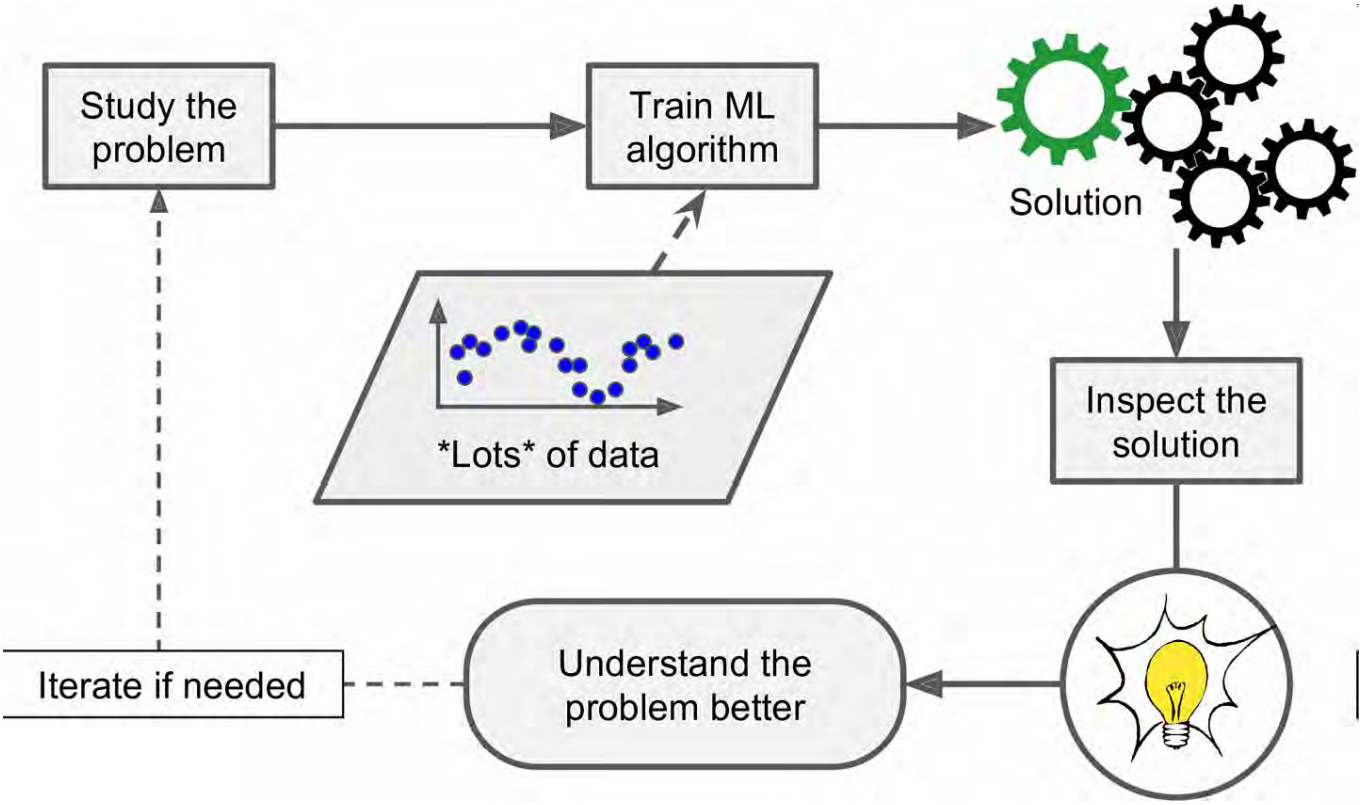


图1-4 机器学习可以帮助人类学习

使用机器学习方法挖掘大量数据，可以发现并不显著的规律。这称作数据挖掘。

总结一下，机器学习善于：

- 需要进行大量手工调整或长串规则的问题：机器学习算法通常可以简化代码、提高性能。
- 问题复杂，传统方法难以解决：最好的机器学习方法可以找到解决方案。
- 环境有波动：机器学习算法可以适应新数据。
- 洞察复杂问题和大量数据。

机器学习系统的类型

机器学习有多种类型，可以根据如下规则进行分类：

- 是否在人类监督下进行训练（监督，非监督，半监督和强化学习）
- 是否可以动态渐进学习（线上学习vs批量学习）
- 它们是否只是通过简单地比较新的数据点和已知的数据点，或者在训练数据中进行模式识别，以建立一个预测模型，就像科学家所做的那样（基于实例学习vs基于模型学习）

规则并不仅限于以上的，你可以将他们进行组合。例如，一个先进的垃圾邮件过滤器可以使用神经网络模型动态进行学习，用垃圾邮件和普通邮件进行训练。这就让它成了一个线上、基于模型、监督学习系统。

下面更仔细地学习这些规则。

监督/非监督学习

机器学习可以根据训练时监督的量和类型进行分类。主要有四类：监督学习、非监督学习、半监督学习和强化学习。

监督学习

在监督学习中，用来训练算法训练数据包括了答案，称为标签（图1-5）。

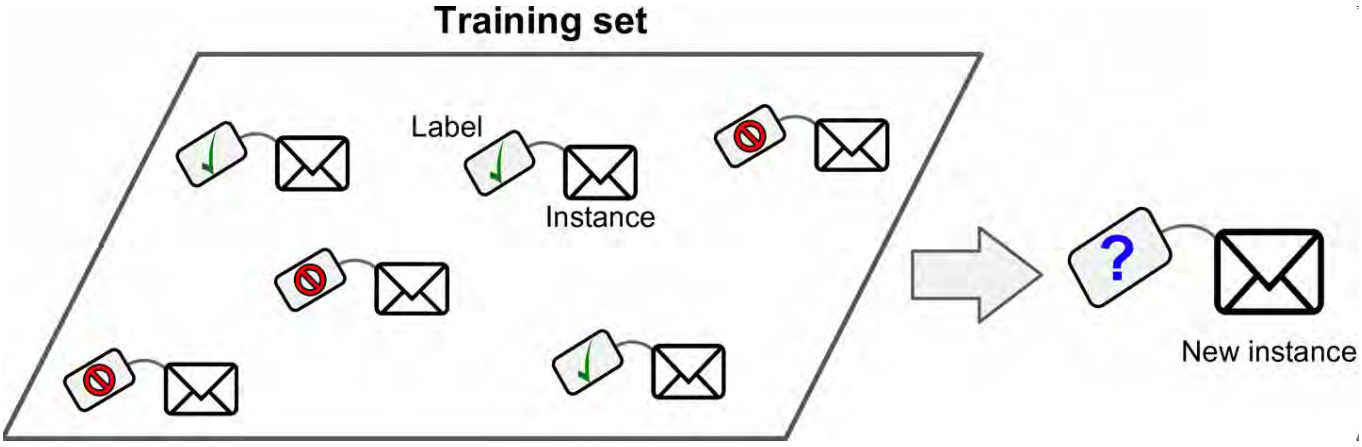


图1-5 用于监督学习（比如垃圾邮件分类）的加了标签的训练集

一个典型的监督学习任务是分类。垃圾邮件过滤器就是一个很好的例子：用许多带有归类（垃圾邮件或普通邮件）的邮件样本进行训练，过滤器必须还能对新邮件进行分类。

另一个典型任务是预测目标数值，例如给出一些特征（里程数、车龄、品牌等等）称作预测值，来预测一辆汽车的价格。这类任务称作回归（图1-6）。要训练这个系统，你需要给出大量汽车样本，包括它们的预测值和

标签（即，它们的价格）。

> 笔记：在机器学习中，一个属性就是一个数据类型（例如，“里程数”），取决于具体问题一个特征会有多个含义，但通常是属性加上它的值（例如，“里程数=15000”）。许多人是不区分地使用属性和特征。

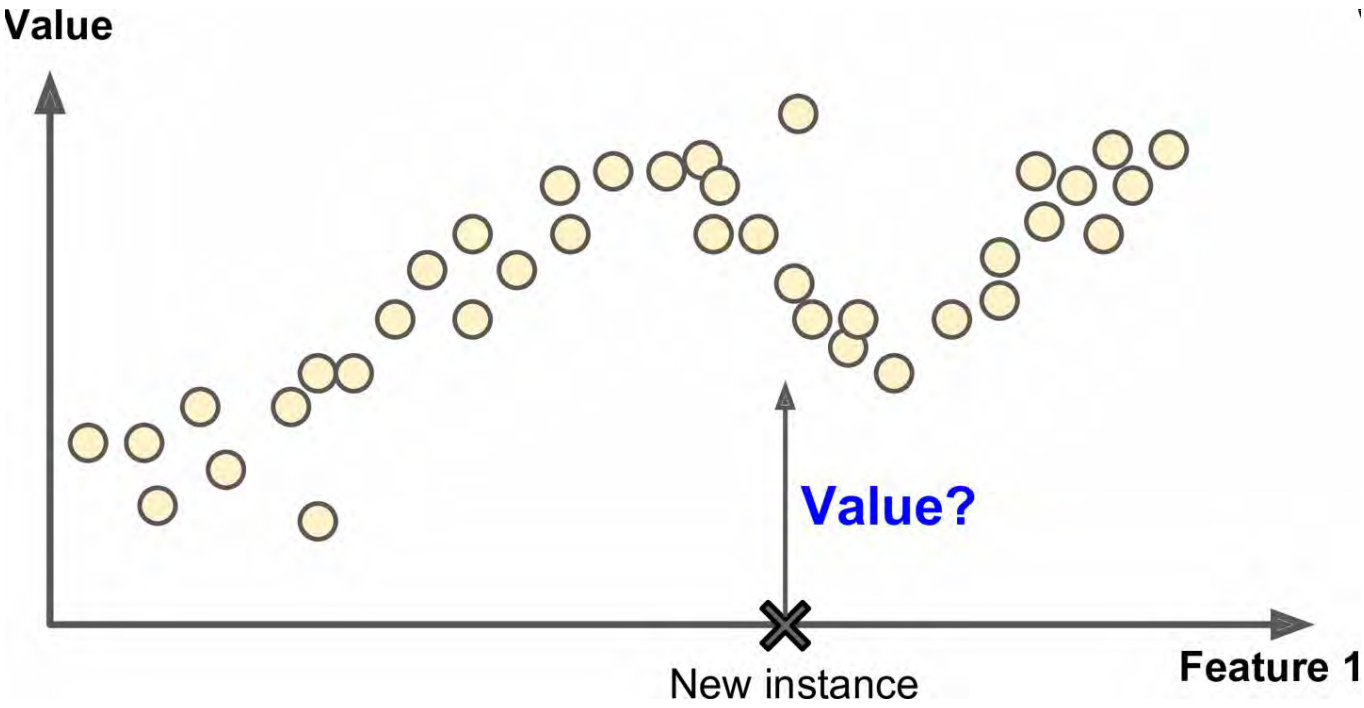


图1-6 回归

注意，一些回归算法也可以用来进行分类，反之亦然。例如，逻辑回归通常用来进行分类，它可以生成一个归属某一类的可能性的值（例如，20%几率为垃圾邮件）。

下面是一些重要的监督学习算法（本书都有介绍）：

- K近邻算法
- 线性回归
- 逻辑回归
- 支持向量机（SVM）
- 决策树和随机森林
  -神经网络

非监督学习

在非监督学习中，你可能猜到了，训练数据是没有加标签的（图1-7）。系统在没有老师的条件下进行学习。

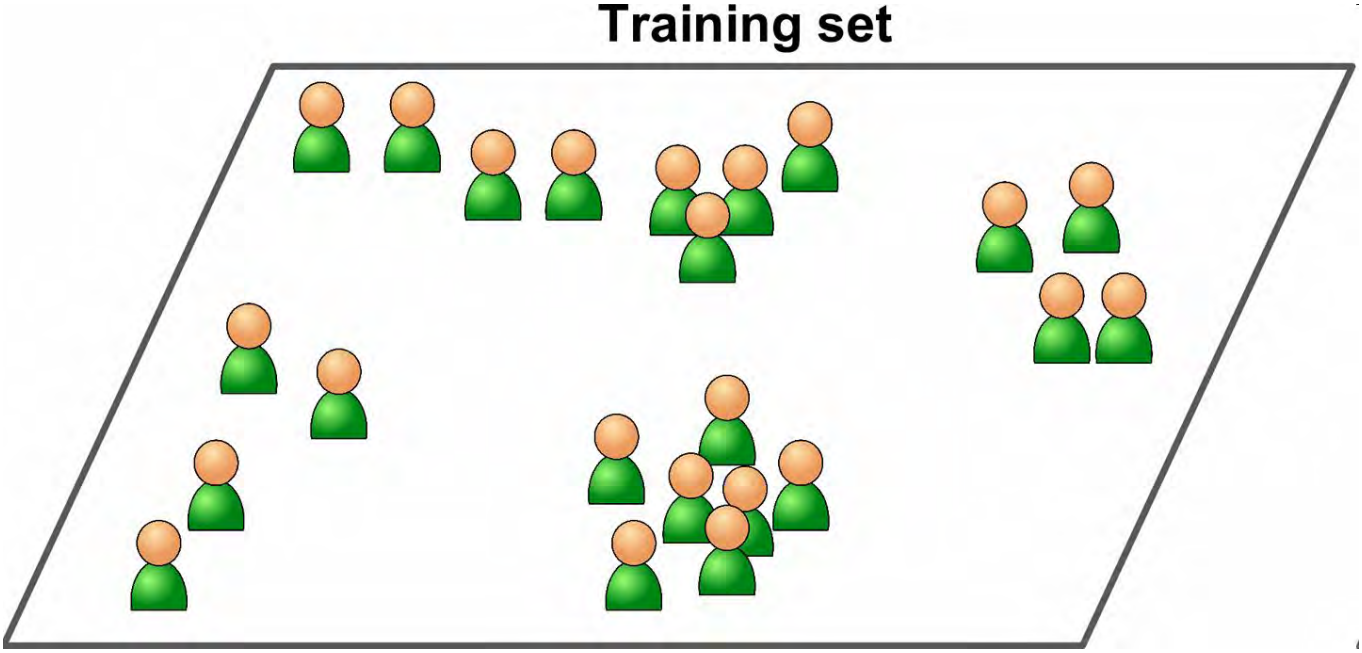


图1-7 非监督学习的一个不加标签的训练集

下面是一些最重要的非监督学习算法（我们会在第8章介绍降维）：

- 聚类
  k-均值
  层次聚类分析（Hierarchical Cluster Analysis，HCA）
  期望最大值
- 可视化和降维
  主成分分析（Principal Component Analysis，PCA）
  核主成分分析
  局部线性嵌入（Locally-Linear Embedding，LLE）
  t-分布邻域嵌入算法（t-distributed Stochastic Neighbor Embedding，t-SNE）
- 关联性规则学习
  Apriori算法
  Eclat算法

例如，假设你有一份关于你的博客访客的大量数据。你想运行一个聚类算法，检测相似访客的分组（图1-8）。你不会告诉算法某个访客属于哪一类：它会自己找出关系，无需帮助。例如，算法可能注意到40%的访客是喜欢漫画书的男性，通常是晚上访问，20%是科幻爱好者，他们是在周末访问等等。如果你使用层次聚类分析，它可能还会细分每个分组为更小的组。这可以帮助你为每个分组定位博文。

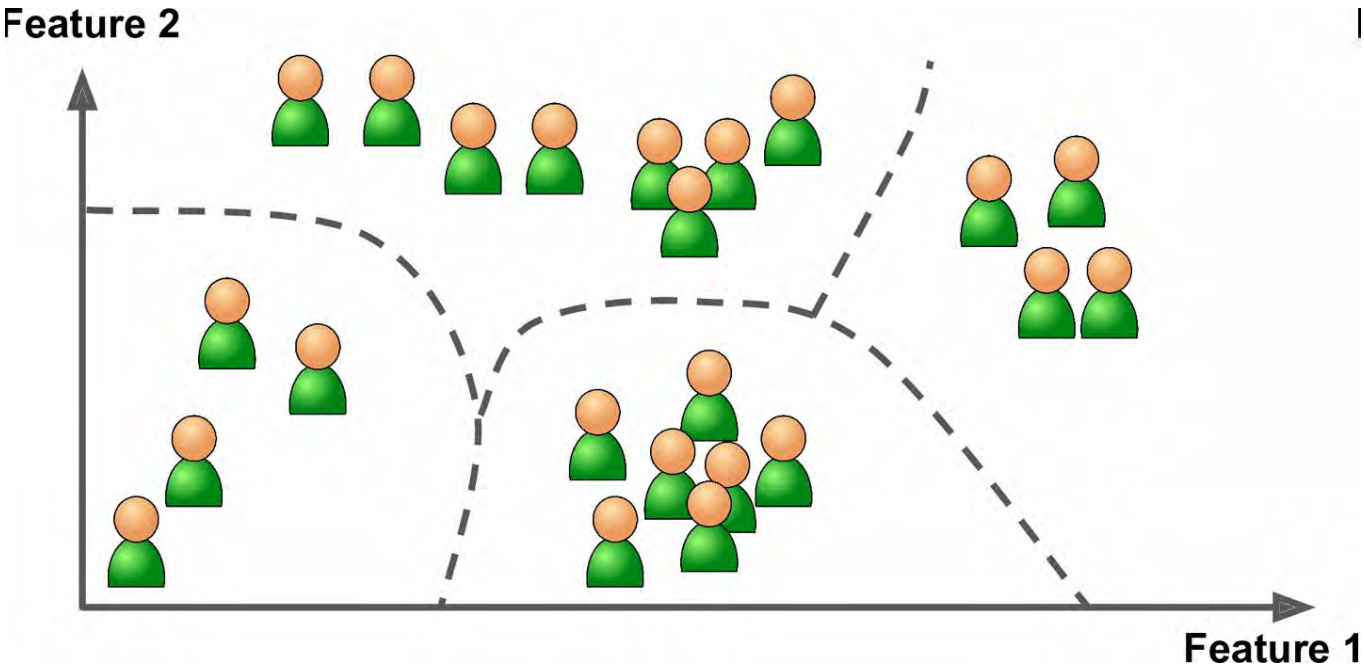


图1-8 聚类

可视化算法也是极佳的非监督学习案例：给算法大量复杂的且不加标签的数据，算法输出数据的2D或3D图像（图1-9）。算法会试图保留数据的结构（即尝试保留输入的独立聚类，避免在图像中重叠），这样就可以明白数据是如何组织起来的，也许还能发现隐藏的规律。

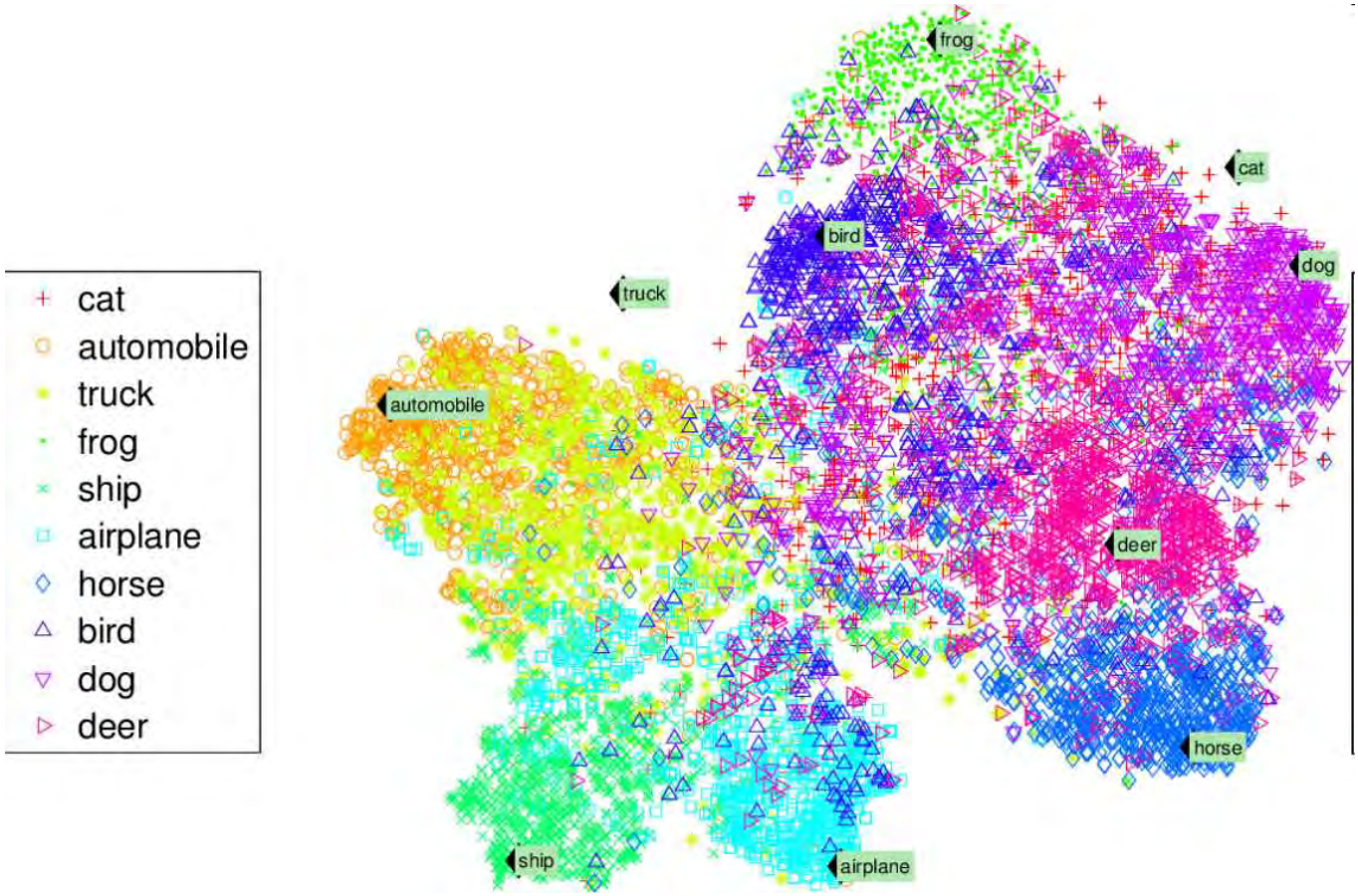


图1-9 t-SNE可视化案例，突出了聚类（注：注意动物是与汽车分开的，马与鹿很近、与鸟院，以此类推）

与此有关联的任务是降维，降维的目的是简化数据、但是不能失去大部分信息。做法之一是合并若干相关的特征。例如，汽车的里程数与车龄高度相关，降维算法就会将它们合并成一个，表示汽车的磨损。这叫做特征提取。

> 提示：在用训练集训练机器学习算法（比如监督学习算法）时，最好对训练集进行降维。这样可以运行的更快，占用的硬盘和内存空间更少，有些情况下性能也更好。

另一个重要的非监督任务是异常检测（anomaly detection） —— 例如，检测异常的信用卡转账以防欺诈，检测制造缺陷，或者在训练之前自动从训练数据集去除异常值。异常检测的系统使用正常值训练的，当它碰到一个新实例，它可以判断这个新实例是否像正常值或异常值（图1-10）。

图1-10 异常检测

最后，另一个常见的非监督任务是关联规则学习，它的目标是挖掘大量数据以发现属性间有趣的关系。例如，假设你拥有一个超市。在销售日志上运行关联规则，可能发现买了烧烤酱和薯片的人也会买牛排。因此，你可以将这些商品放在一起。

半监督学习

一些算法可以处理部分带标签的训练数据，通常是大量不带标签数据加上小部分带标签数据。这称作半监督学习（图1-11）。

一些图片存储服务，比如Google Photos，是半监督学习的好例子。一旦你上传了所有家庭相片，它就能自动识别相同的人A出现了相片1、5、11中，另一个人B出现在了相片2、5、7中。这是算法的非监督部分（聚类）。现在系统需要的就是你告诉这两个人是谁。只要给每个人一个标签，算法就可以命名每张照片中的每个人，特别适合搜索照片。

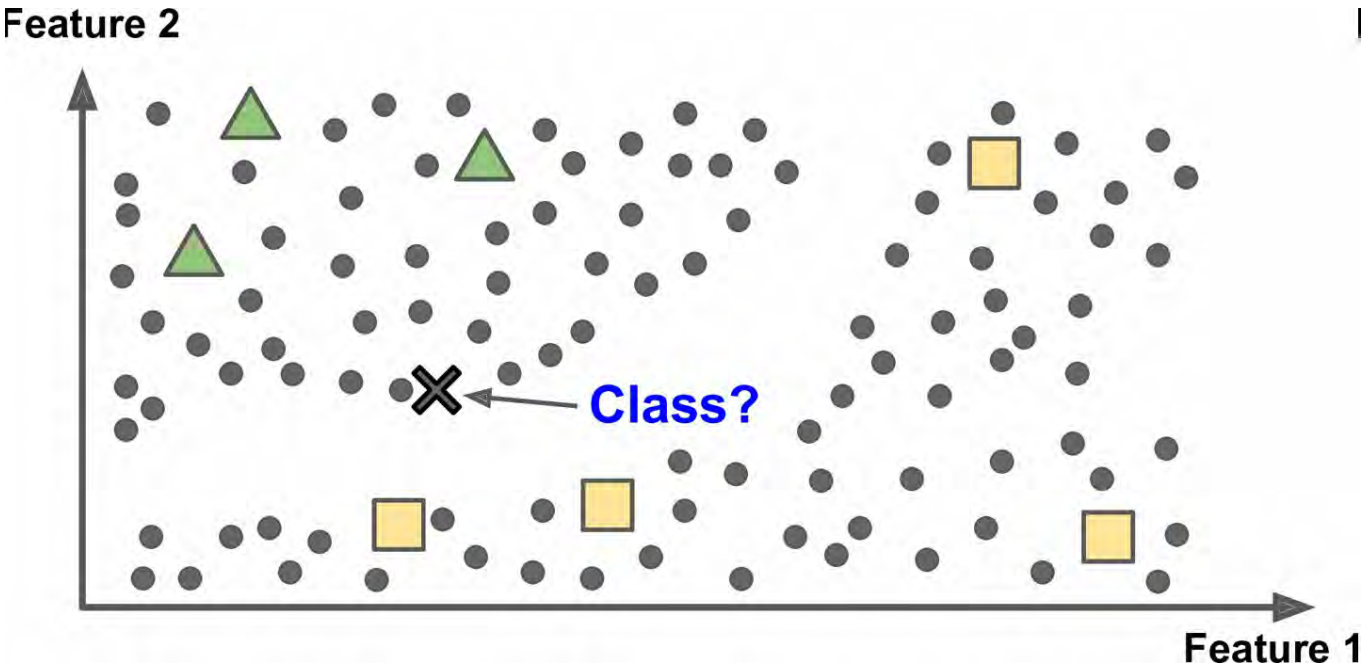


图1-11 半监督学习

多数半监督学习算法时非监督和监督算法的结合。例如，深度信念网络（deep belief networks）是基于被称为互相叠加的受限玻尔兹曼机（restricted Boltzmann machines，RBMs）的非监督组件。RBMs是先用非监督方法进行训练，再用监督学习方法进行整个系统微调。

强化学习

强化学习非常不同。学习系统在这里被称为执行者agent，可以对环境进行观察，选择和执行动作，获得奖励（负奖励是惩罚，见图1-12）。然后它必须自己学习哪个是最佳方法（称为策略policy），以得到长久的最大奖励。策略决定了执行者在给定情况下应该采取的行动。

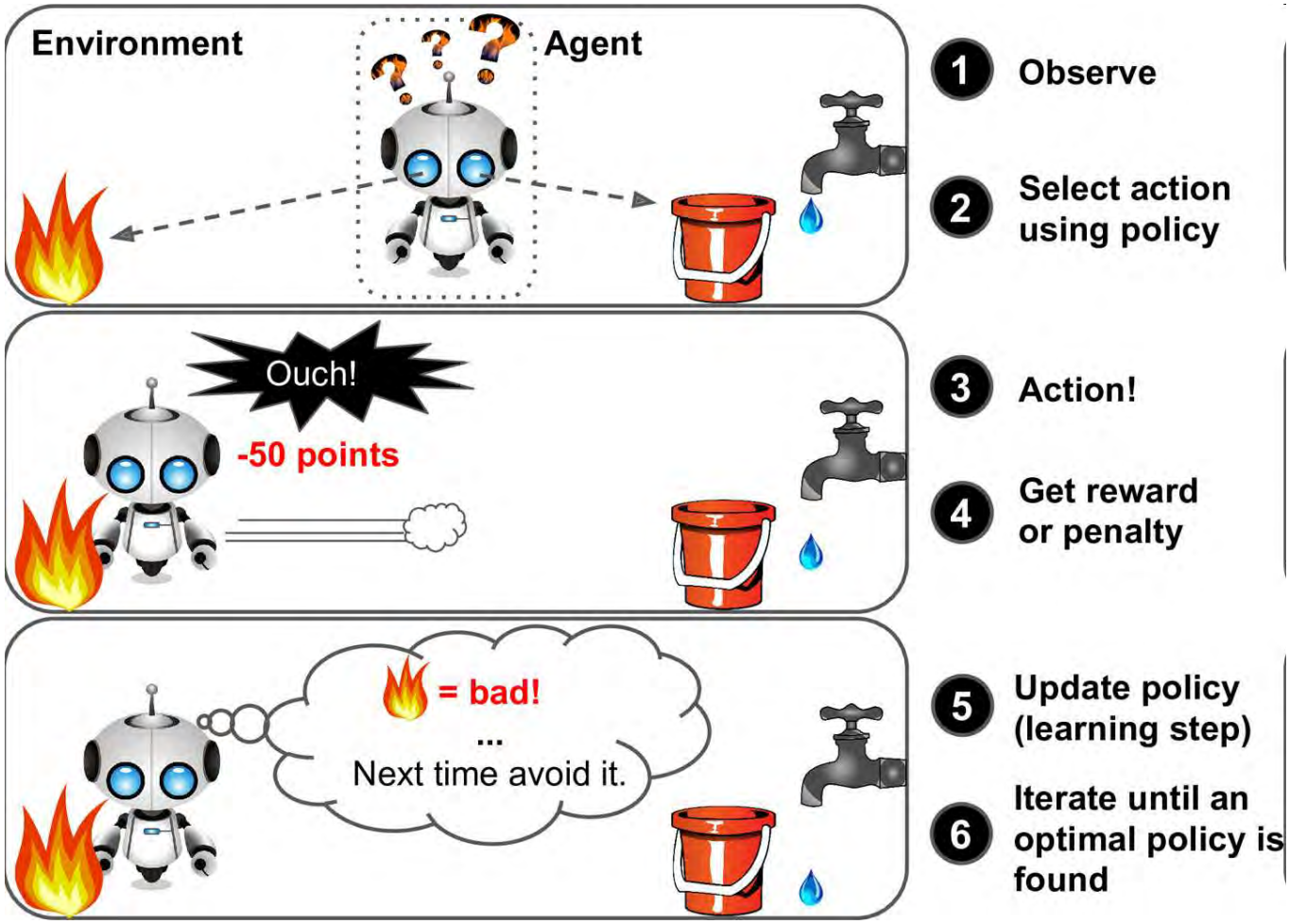


图1-12 强化学习

例如，许多机器人运行强化学习算法以学习如何行走。DeepMind的AlphaGo也是强化学习的例子：它在2016年三月击败了世界围棋冠军李世石（译者注：2017年五月，AlphaGo又击败了世界排名第一的柯洁）。它是通过分析数百万盘棋局学习制胜策略，然后自己和自己下棋。要注意，在比赛中机器学习是关闭的；AlphaGo只是使用它学会的策略。

批量和线上学习

另一个用来分类机器学习的准则是，它是否能从导入的数据流进行持续学习。

批量学习

在批量学习中，系统不能进行持续学习：必须用所有可用数据进行训练。这通常会占用大量时间和计算资源，所以一般是线下做的。首先是进行训练，然后部署在生产环境且不带学习，它只是应用学到的。这称为离线学习。

如果你想让一个批量学习系统明白新数据（例如垃圾邮件的新类型），就需要从头训练一个系统的新版本，使用全部数据集（不仅有新数据也有老数据），然后停掉老系统，换上新系统。

幸运的是，训练、评估、部署一套机器学习的系统的整个过程可以自动进行（见图1-3），所以即便是批量学习也可以适应改变。只要有需要，就可以方便地更新数据、训练一个新版本。

这个方法很简单，通常可以满足需求，但是用全部数据集进行训练会花费大量时间，所以一般是每24小时或每周训练一个新系统。如果系统需要快速适应变化的数据（比如，预测股价变化），就需要一个响应更及时的方案。

另外，用全部数据训练需要大量计算资源（CPU、内存空间、磁盘空间、磁盘I/O、网络I/O等等）。如果你有大量数据，并让系统每天自动从头开始训练，就会开销很大。如果数据量巨大，甚至无法使用批量学习算法。

最后，如果你的系统需要自动学习，但是资源有限（比如，一台智能手机或火星车），携带大量训练数据、每天花费数小时的大量资源进行训练是不实际的。

幸运的是，对于上面这些情况，还有一个更佳的方案可以进行持续学习。

线上学习

在线上学习中，是用数据实例持续地进行训练，可以一次一个或一次几个实例（称为小批量）。每个学习步骤都很快且廉价，所以系统可以动态地学习到达的新数据（见图1-13）。

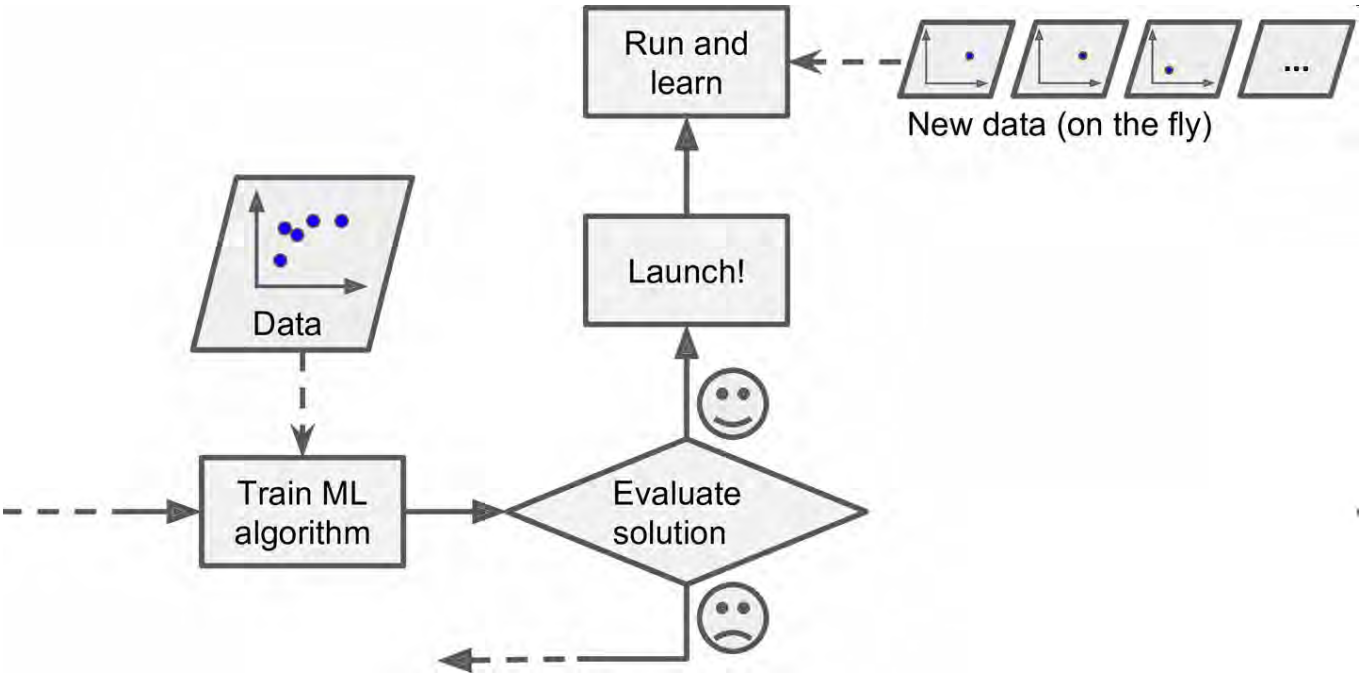


图1-13 线上学习

线上学习很适合系统接收连续流的数据（比如，股票价格），且需要自动对改变作出调整。如果计算资源有限，线上学习是一个不错的方案：一旦线上学习系统学习了新的数据实例，它就不再需要这些数据了，所以扔掉这些数据（除非你想滚回到之前的一个状态，再次使用数据）。这样可以节省大量的空间。

线上学习算法也可以当机器的内存存不下大量数据集时，用来训练系统（这称作核外学习， out-of-core learning）。算法加载部分的数据，用这些数据进行训练，重复这个过程，直到用所有数据都进行了训练（见图1-14）。

> 警告：这个整个过程通常是离线完成的（即，不在部署的系统上），所以线上学习这个名字会让人疑惑。可以把它想成持续学习。

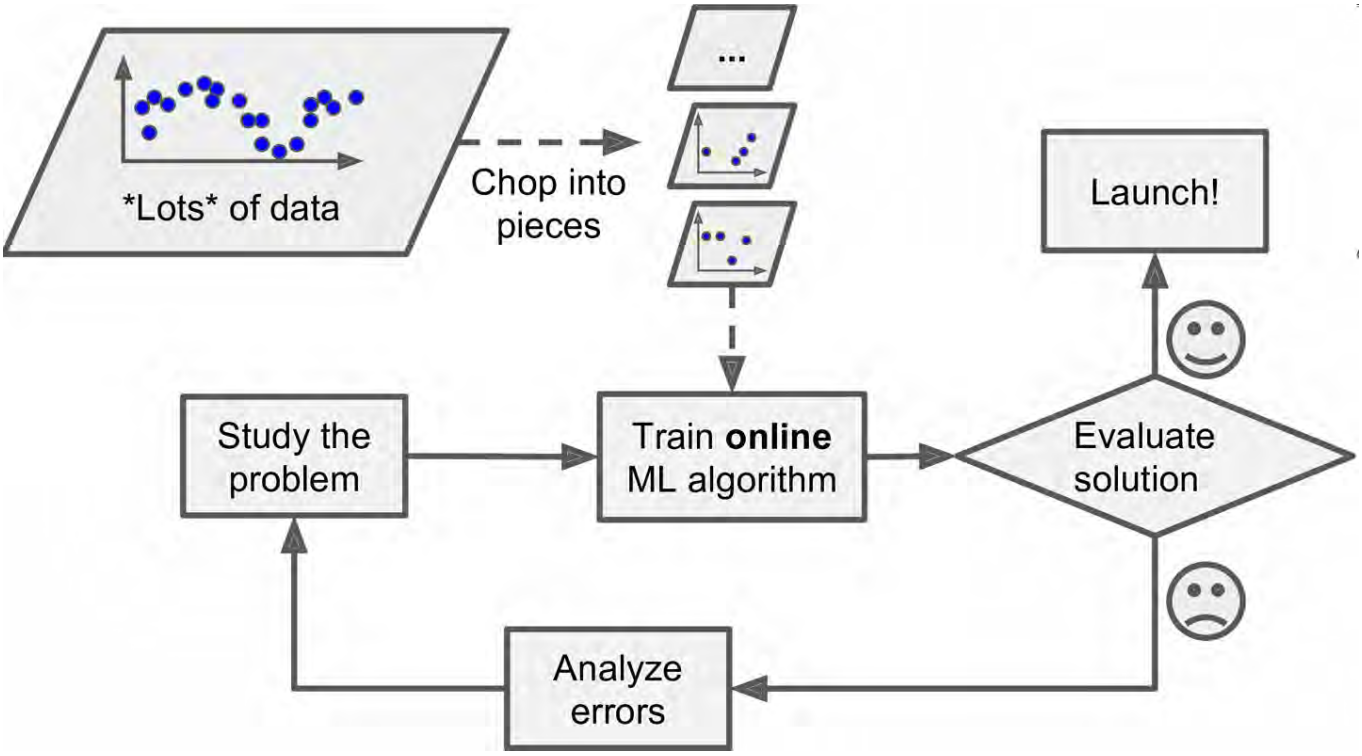


图1-14 使用线上学习处理大量数据集

线上学习系统的一个重要参数是，它们可以多快地适应数据的改变：这被称为学习速率。如果你设定一个高学习速率，系统就可以快速适应新数据，但是也会快速忘记老书记（你可不想让垃圾邮件过滤器只标记最新的垃圾邮件种类）。相反的，如果你设定的学习速率低，系统的惰性就会强：即，它学的更慢，但对新数据中的噪声或没有代表性的数据点结果不那么敏感。

线上学习的挑战之一是，如果坏数据被用来进行训练，系统的性能就会逐渐下滑。如果这是一个部署的系统，用户就会注意到。例如，坏数据可能来自失灵的传感器或机器人，或某人向搜索引擎传入垃圾信息以提高搜索排名。要减小这种风险，你需要密集监测，如果检测到性能下降，要快速关闭（或是滚回到一个之前的状态）。你可能还要监测输入数据，对反常数据做出反应（比如，使用异常检测算法）。

基于实例vs基于模型学习

另一种分类机器学习的方法是判断它们是如何进行归纳推广的。大多机器学习任务是关于预测的。这意味着给定一定数量的训练样本，系统需要能推广到之前没见到过的样本。对训练数据集有很好的性能还不够，真正的目标是对新实例的性能。

有两种主要的归纳方法：基于实例学习和基于模型学习。

基于实例学习

也许最简单的学习形式就是用记忆学习。如果用这种方法做一个垃圾邮件检测器，只需标记所有和用户标记的垃圾邮件相同的邮件 —— 这个方法不差，但肯定不是最好的。

不仅能标记和已知的垃圾邮件相同的邮件，你的垃圾邮件过滤器也要能标记类似垃圾邮件的邮件。这就需要测量两封邮件的相似性。一个（简单的）相似度测量方法是统计两封邮件包含的相同单词的数量。如果一封邮件含有许多垃圾邮件中的词，就会被标记为垃圾邮件。

这被称作基于实例学习：系统先用记忆学习案例，然后使用相似度测量推广到新的例子（图1-15）。

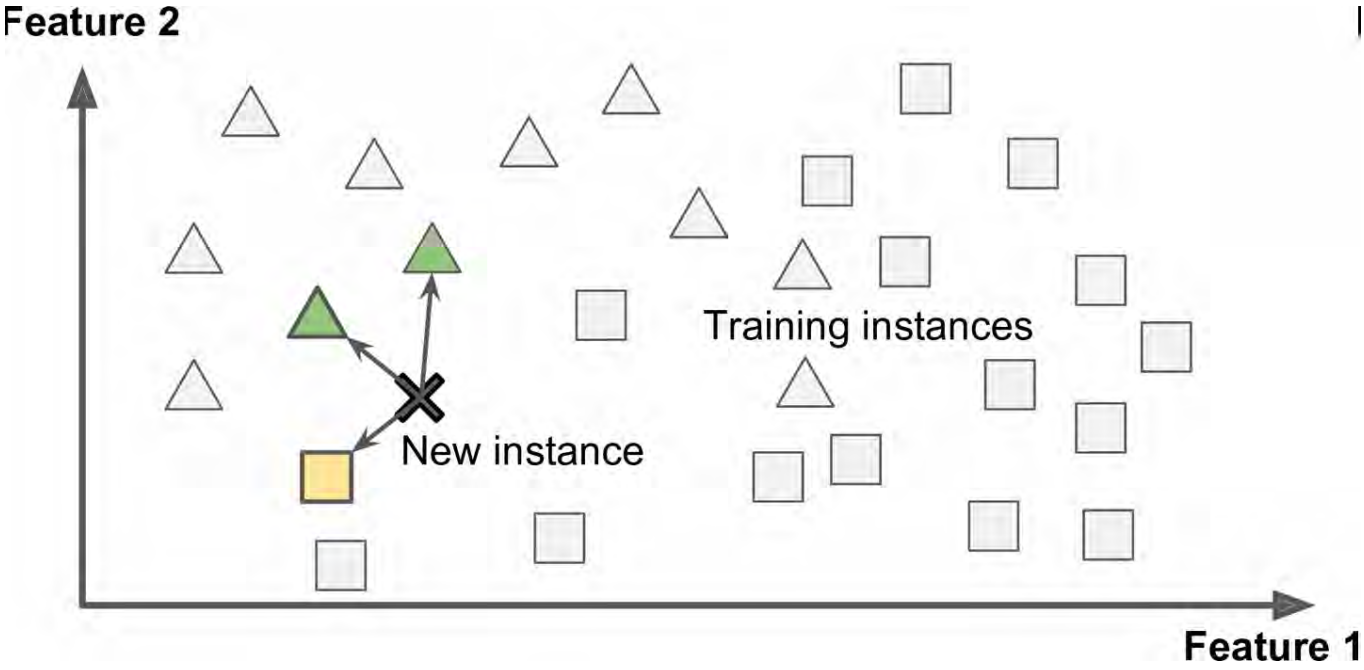


图1-15 基于实例学习

基于模型学习

另一种从样本集进行归纳的方法是建立这些样本的模型，然后使用这个模型进行预测。这称作基于模型学习（图1-16）。

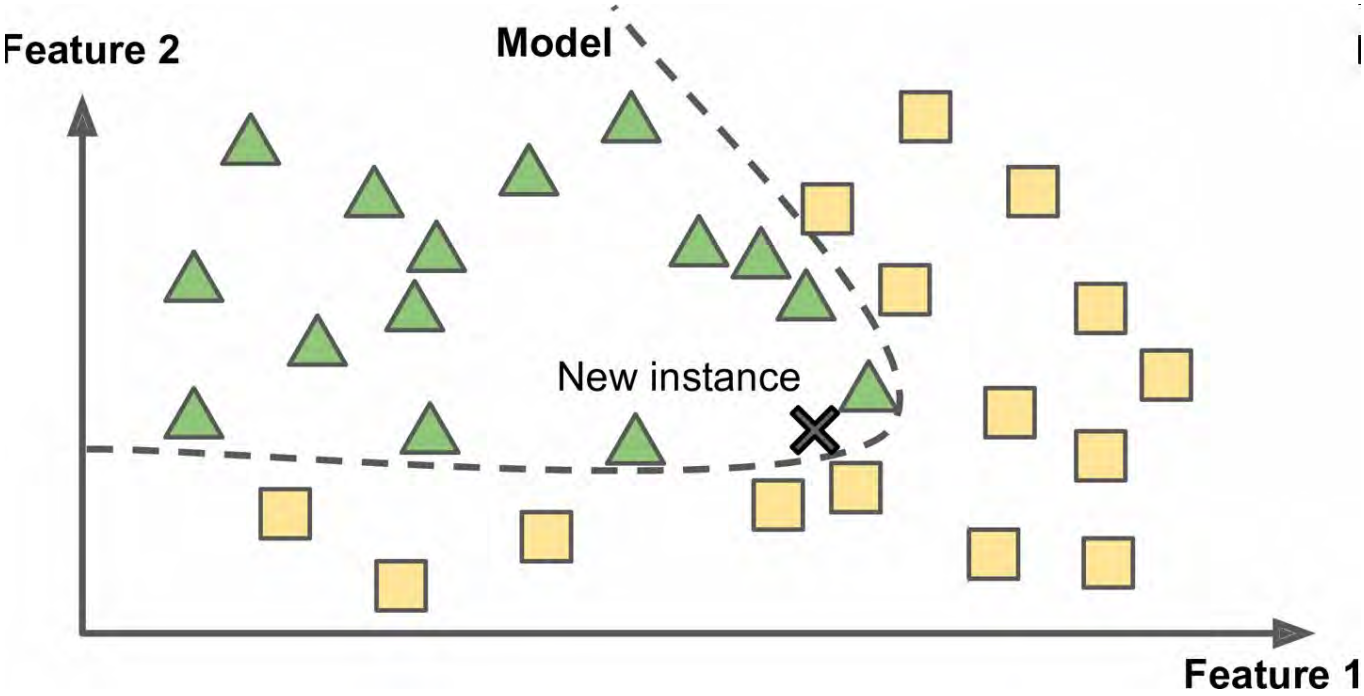


图1-16 基于模型学习

例如，你想知道钱是否能让人快乐，你从OECD网站（http://stats.oecd.org/index.aspx?DataSetCode=BLI）下载了Better Life Index指数数据，还从IMF（http://goo.gl/j1MSKe，这个短链接需要翻墙才能打开，长链接又太长）下载了人均GDP数据。表1-1展示了摘要。

国家	人均 GDP（美元）	生活满意度
匈牙利	12240	4.9
韩国	27195	5.8
发过	37675	6.5
澳大利亚	50962	7.3
美国	55805	7.2

表1-1 钱会使人幸福吗?

用一些国家的数据画图（图1-17）。

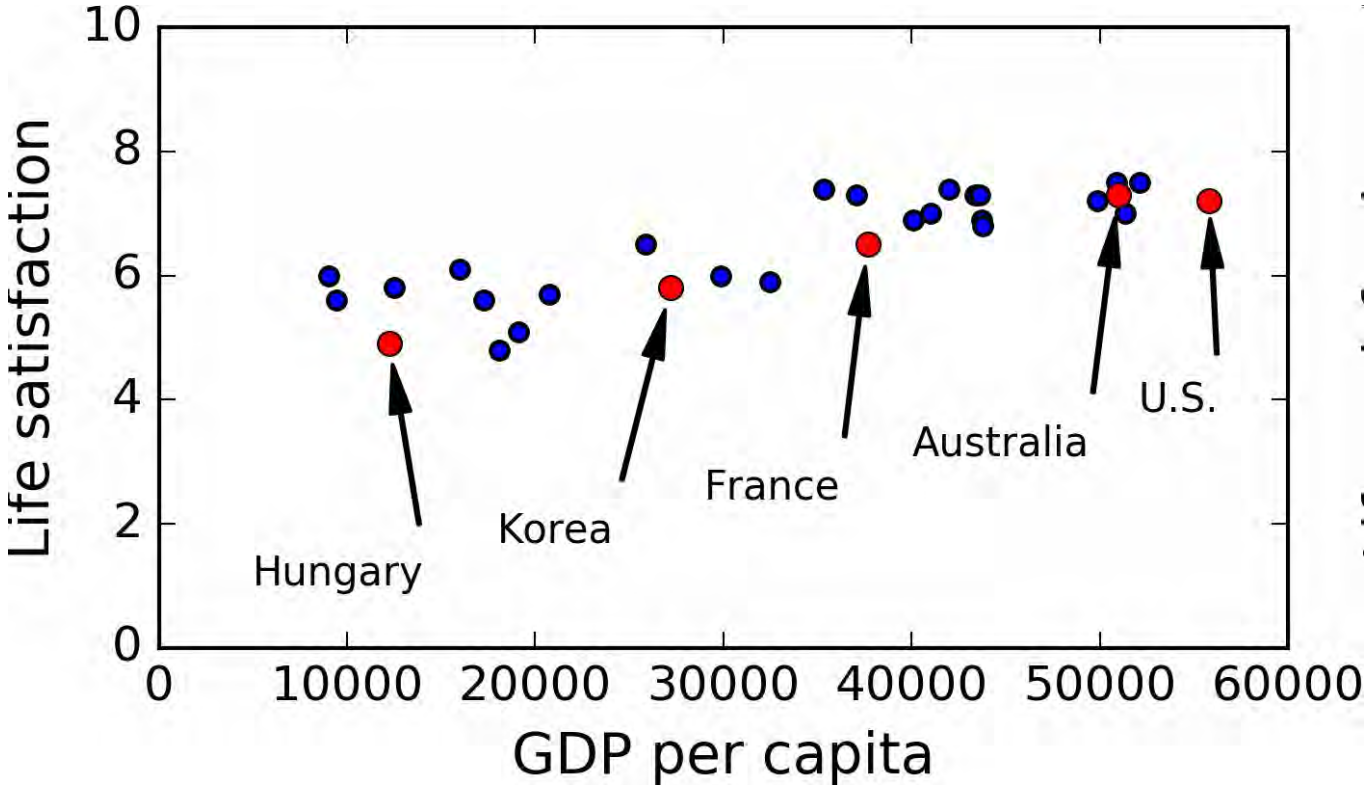


图1-17 你看到趋势了吗?

确实能看到趋势！尽管数据有噪声（即，部分随机），看起来生活满意度是随着人均GDP的增长线性提高的。所以，你决定生活满意度建模为人均GDP的线性函数。这一步称作模型选择：你选一个生活满意度的线性模型，只有一个属性，人均GDP（等式1-1）。

$$life\_satisfaction = \theta_0 + \theta_1 \times GDP\_per\_capita$$

等式1-1 一个简单的线性模型

这个模型有两个参数θ_0和θ_1。通过调整这两个参数，你可以使你的模型表示任何线性函数，见图1-18。

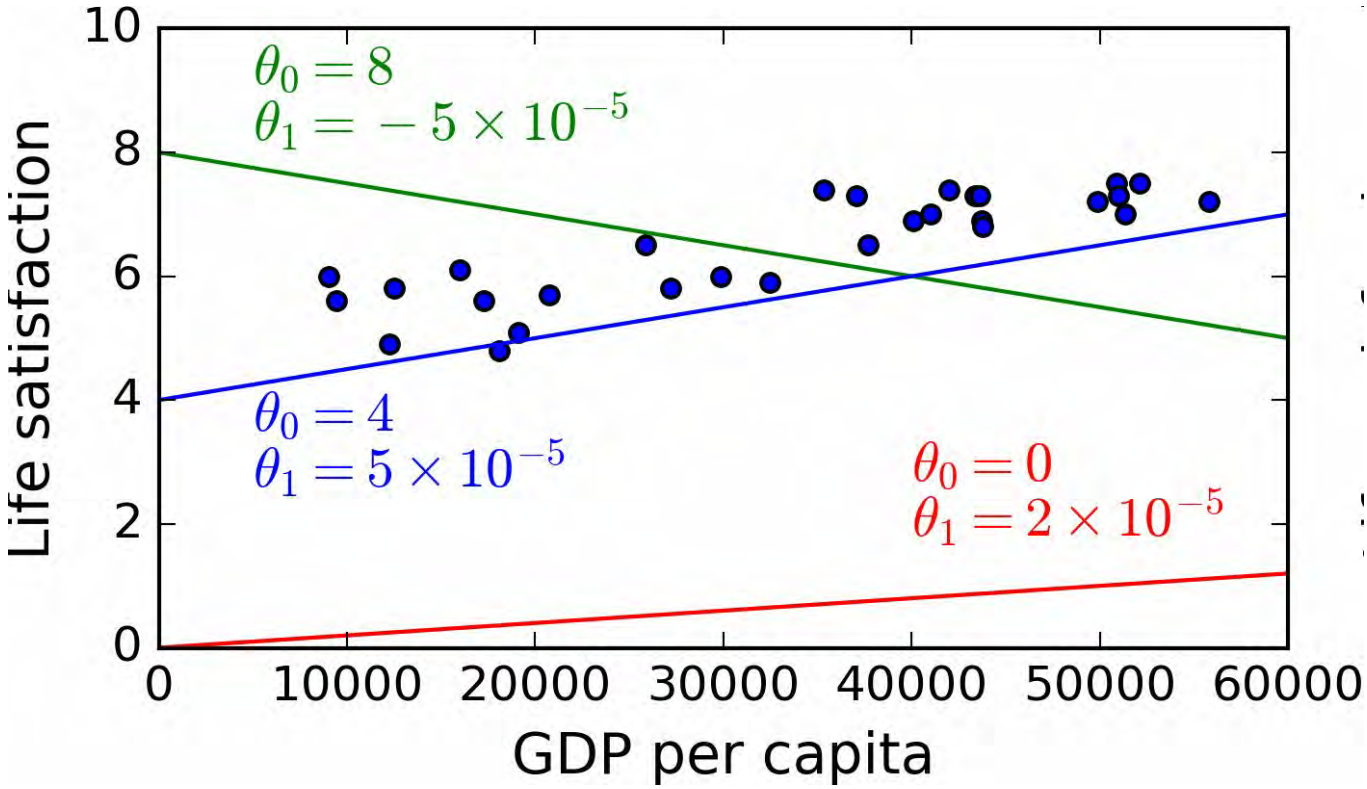


图-18 几个可能的线性模型

在使用模型之前，你需要确定θ_0和θ_1。如何能知道哪个值可以使模型的性能最佳呢？要回答这个问题，你需要指定性能的量度。你可以定义一个实用函数（或拟合函数）用来测量模型是否够好，或者你可以定义一个代价函数来测量模型有多差。对于线性回归问题，人们一般是用代价函数测量线性模型的预测值和训练样本的距离差，目标是使距离差最小。

接下来就是线性回归算法，你用训练样本训练算法，算法找到使线性模型最拟合数据的参数。这称作模型训练。在我们的例子中，算法得到的参数值是θ_0=4.85和θ_1=4.91×10^{-5}。

现在模型已经最紧密地拟合到训练数据了，见图1-19。

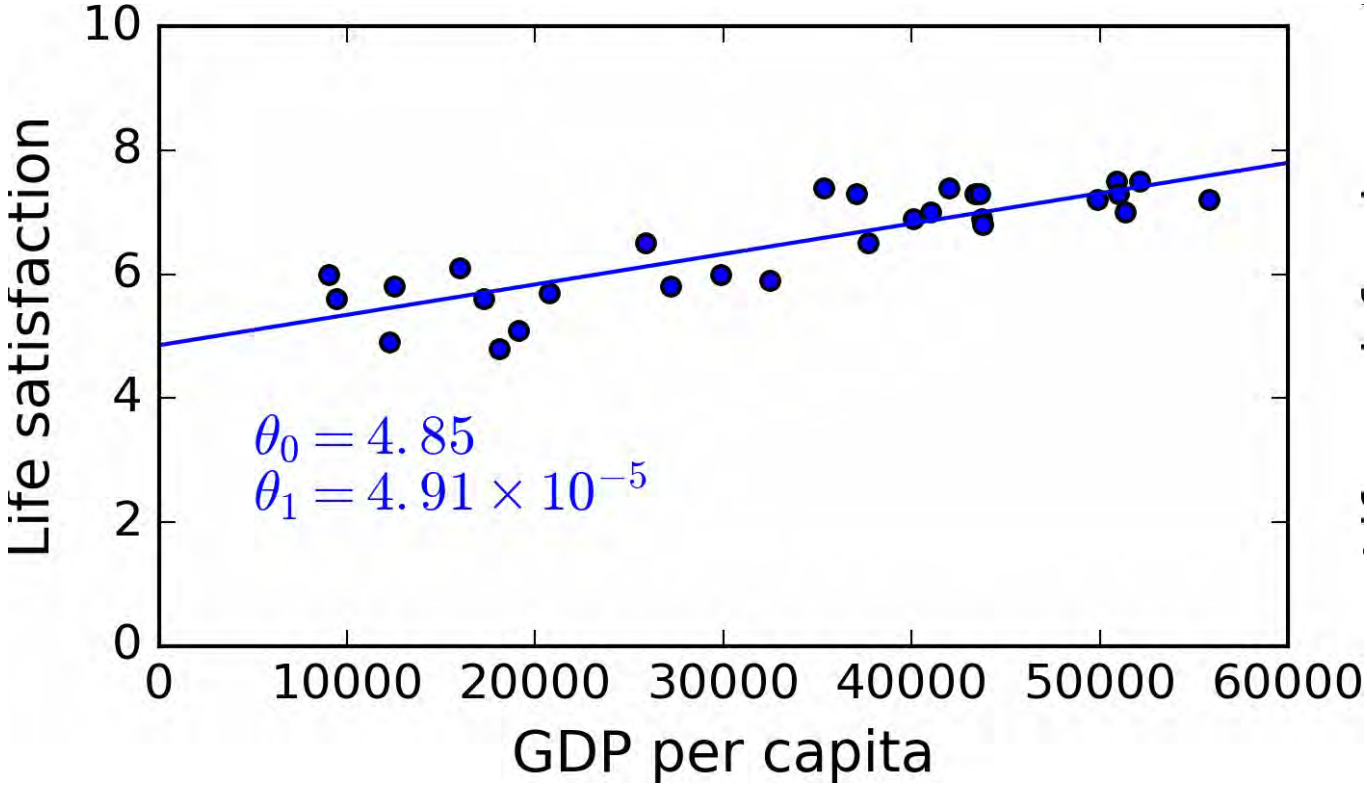


图1-19 最佳拟合训练数据的线性模型

最后，可以准备运行模型进行预测了。例如，假如你想知道塞浦路斯人有多幸福，但OECD没有它的数据。幸运的是，你可以用模型进行预测：查询塞浦路斯的人均GDP，为22587美元，然后应用模型得到生活满意度，后者的值在$4.85 + 22,587 \times 4.91 \times 10^{-5} = 5.96$左右。

为了激起你的兴趣，案例1-1展示了加载数据、准备、创建散点图的Python代码，然后训练线性模型并进行预测。

案例1-1，使用Scikit-Learn训练并运行线性模型。

```
import matplotlib
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import sklearn

# 加载数据
oecd_bli = pd.read_csv("oecd_bli_2015.csv", thousands=',')
gdp_per_capita = pd.read_csv("gdp_per_capita.csv",thousands=',',delimiter='\t',
                             encoding='latin1', na_values="n/a")

# 准备数据
country_stats = prepare_country_stats(oecd_bli, gdp_per_capita)
X = np.c_[country_stats["GDP per capita"]]
y = np.c_[country_stats["Life satisfaction"]]

# 可视化数据
country_stats.plot(kind='scatter', x="GDP per capita", y='Life satisfaction')
plt.show()

# 选择线性模型
lin_reg_model = sklearn.linear_model.LinearRegression()

# 训练模型
lin_reg_model.fit(X, y)

# 对塞浦路斯进行预测
X_new = [[22587]]  # 塞浦路斯的人均GDP
print(lin_reg_model.predict(X_new)) # outputs [[ 5.96242338]]
```

> 笔记：如果你之前接触过基于实例学习算法，你会发现斯洛文尼亚的人均GDP（20732美元）和塞浦路斯差距很小，OECD数据上斯洛文尼亚的生活满意度是5.7，就可以预测塞浦路斯的生活满意度也是5.7。如果放大一下范围，看一下接下来两个临近的国家，你会发现葡萄牙和西班牙的生活满意度分别是5.1和6.5。对这三个值进行平均得到5.77，就和基于模型的预测值很接近。这个简单的算法叫做k近邻回归（这个例子中，k=3）。
>
> 在前面的代码中替换线性回归模型为k近邻模型，只需更换下面一行：
>
> ```
> clf = sklearn.linear_model.LinearRegression()
> ```

> 为：
>
> ```
> clf = sklearn.neighbors.KNeighborsRegressor(n_neighbors=3)
> ```

如果一切顺利，你的模型就可以作出好的预测。如果不能，你可能需要使用更多的属性（就业率、健康、空气污染等等），获取更多更好的训练数据，或选择一个更好的模型（比如，多项式回归模型）。

总结一下：

- 研究数据
- 选择模型
- 用训练数据进行训练（即，学习算法搜寻模型参数值，使代价函数最小）
- 最后，使用模型对新案例进行预测（这称作推断），但愿这个模型推广效果不差

这就是一个典型的机器学习项目。在第2章中，你会第一手地接触一个完整的项目。

我们已经学习了许多关于基础的内容：你现在知道了机器学习是关于什么的，为什么它这么有用，最常见的机器学习的分类，典型的项目工作流。现在，让我们看一看会发生什么错误，导致不能做出准确的预测。

机器学习的主要挑战

简而言之，因为你的主要任务是选择一个学习算法并用一些数据进行训练，会导致错误的两件事就是“错误的算法”和“错误的数据”。我们从错误的数据开始。

训练数据量不足

要让一个蹒跚学步的孩子知道什么是苹果，需要做的就是指着一个苹果说“苹果”（可能需要重复这个过程几次）。现在这个孩子就能认识所有形状和颜色的苹果。真是个天才！

机器学习还达不到这个程度；需要大量数据，才能让多数机器学习算法正常工作。即便对于非常简单的问题，一般也需要数千的样本，对于复杂的问题，比如图像或语音识别，你可能需要数百万的样本（除非你能重复使用部分存在的模型）。

> 数据不合理的有效性
>
> 在一篇2001年发表的著名论文（goo.gl/R5enIE）中，微软研究员Michele Banko和Eric Brill展示了不同的机器学习算法，包括非常简单的算法，一旦有了大量数据进行训练，在进行去除语言歧义的测试中几乎有相同的性能（见图1-20）。

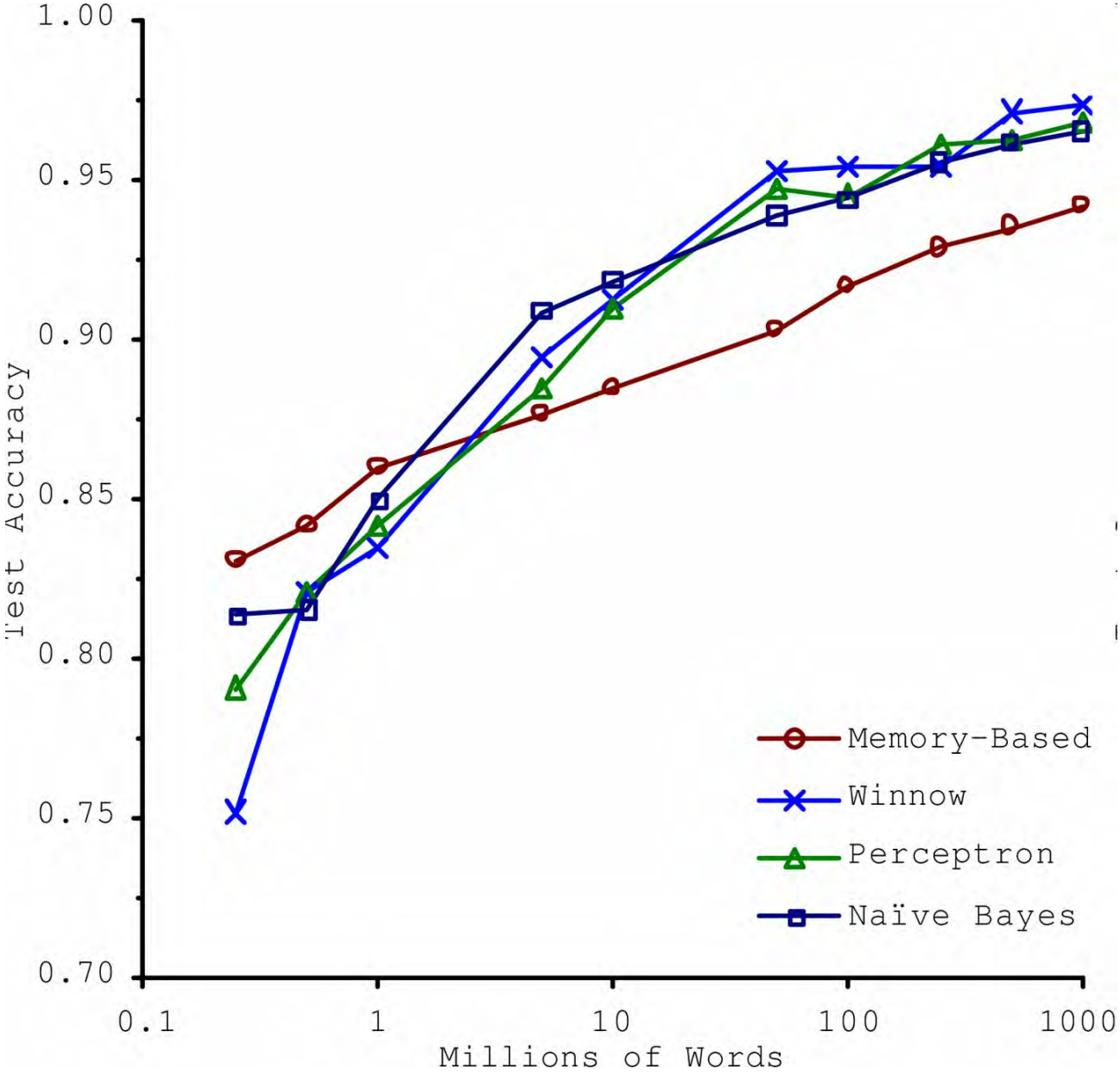


图1-20 数据和算法的重要性对比

论文作者说：“结果说明，我们可能需要重新考虑在算法开发vs语料库发展上花费时间和金钱的取舍。”

对于复杂问题，数据比算法更重要的主张在2009年由Norvig发表的论文《The Unreasonable Effectiveness of Data》得到了进一步的推广。但是，应该注意到，小型和中型的数据集仍然是非常常见的，获得额外的训练数据并不总是轻易和廉价的，所以不要抛弃算法。

没有代表性的训练数据

为了更好地进行归纳推广，让训练数据对新数据具有代表性是非常重要的。无论你用的是基于实例学习或基于模型学习，这点都很重要。

例如，我们之前用来训练线性模型的国家集合不够具有代表性：缺少了一些国家。图1-21展示了添加这些缺失国家之后的数据。

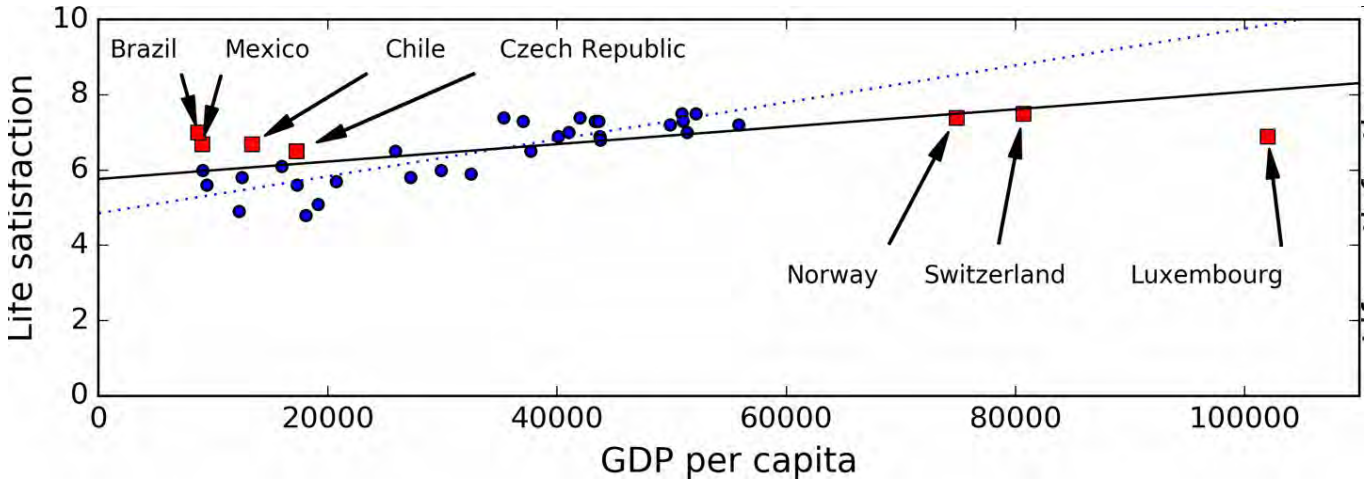


图1-21 一个更具代表性的训练样本

如果你用这份数据训练线性模型，得到的是实线，旧模型用虚线表示。可以看到，添加几个国家不仅可以显著地改变模型，它还说明如此简单的线性模型可能永远不会达到很好的性能。貌似非常富裕的国家没有中等富裕的国家快乐（事实上，非常富裕的国家看起来更不快乐），相反的，一些贫穷的国家看上去比富裕的国家还幸福。

使用了没有代表性的数据集，我们训练了一个不可能得到准确预测的模型，特别是对于非常贫穷和非常富裕的国家。

使用具有代表性的训练集对于推广到新案例是非常重要的。但是做起来比说起来要难：如果样本太小，就会有样本噪声（即，会有一定概率包含没有代表性的数据），但是即使是非常大的样本也可能没有代表性，如果取样方法错误的话。这叫做样本偏差。

> 一个样本偏差的著名案例
>
> 也许关于样本偏差最有名的案例发生在1936年兰登和罗斯福的美国大选：《文学文摘》做了一个非常大的民调，给1000万人邮寄了调查信。得到了240万回信，非常有信心地预测兰登会以57%赢得大选。然而，罗斯福赢得了62%的选票。错误发生在《文学文摘》的取样方法：
>
> - 首先，为了获取发信地址，《文学文摘》使用了电话黄页、杂志订阅用户、俱乐部会员等相似的列表。所有这些列表都偏向于富裕人群，他们都倾向于投票给共和党（即兰登）。
> - 第二，只有25%的回答了调研。这就又一次引入了样本偏差，它排除了不关心政治的人、不喜欢《文学文摘》的人，和其它关键人群。这种特殊的样本偏差称作无应答偏差。
>
> 下面是另一个例子：假如你想创建一个能识别疯克音乐视频的系统。建立训练集的方法之一是在YouTube上搜索“疯克音乐”，使用搜索到的视频。但是这样就假定了YouTube的搜索引擎返回的视频集，是对YouTube上的所有疯克音乐有代表性的。事实上，搜索结果会偏向于人们歌手（如果你居住在巴西，你会得到许多“疯克卡瑞欧卡”视频，它们和James Brown的截然不同）。另一方面，怎么还能得到一个大的训练集呢?

低质量数据

很明显，如果训练集中的错误、异常值和噪声（错误测量引入的）太多，系统检测出潜在规律的难度就会变大，性能就会降低。花费时间对训练数据进行清理是十分重要的。事实上，大多数据科学家的一大部分时间是做清洗工作的。例如：

- 如果一些实例是明显的异常值，最好删掉它们或尝试手工修改错误；

- 如果一些实例缺少特征（比如，你的5%的顾客没有说明年龄），你必须决定是否忽略这个属性、忽略这些实例、填入缺失值（比如，年龄中位数），或者训练一个含有这个特征的模型和一个不含有这个特征的模型，等等。

不相关的特征

俗语说：进来的是垃圾，出去的也是垃圾。你的系统只有在训练数据包含足够相关特征、非相关特征不多的情况下，才能进行学习。机器学习项目成功的关键之一是用好的特征进行训练。这个过程称作特征工程，包括：

- 特征选择：在所有存在的特征中选取最有用的特征进行训练。
- 特征提取：组合存在的特征，生成一个更有用的特征（如前面看到的，可以使用降维算法）。
- 收集新数据创建新特征。

现在，我们已经看过了许多坏数据的例子，接下来看几个坏算法的例子。

过拟合训练数据

如果你在外国游玩，当地的出租车司机多收了你的钱。你可能会说这个国家所有的出租车司机都是小偷。过度归纳是我们人类经常做的，如果我们不小心，机器也会犯同样的错误。在机器学习中，这称作过拟合：意思是说，模型在训练数据上表现很好，但是推广效果不好。

图1-22展示了一个高阶多项式生活满意度模型，它大大过拟合了训练数据。即使它比简单线性模型在训练数据上表现更好，你会相信它的预测吗？

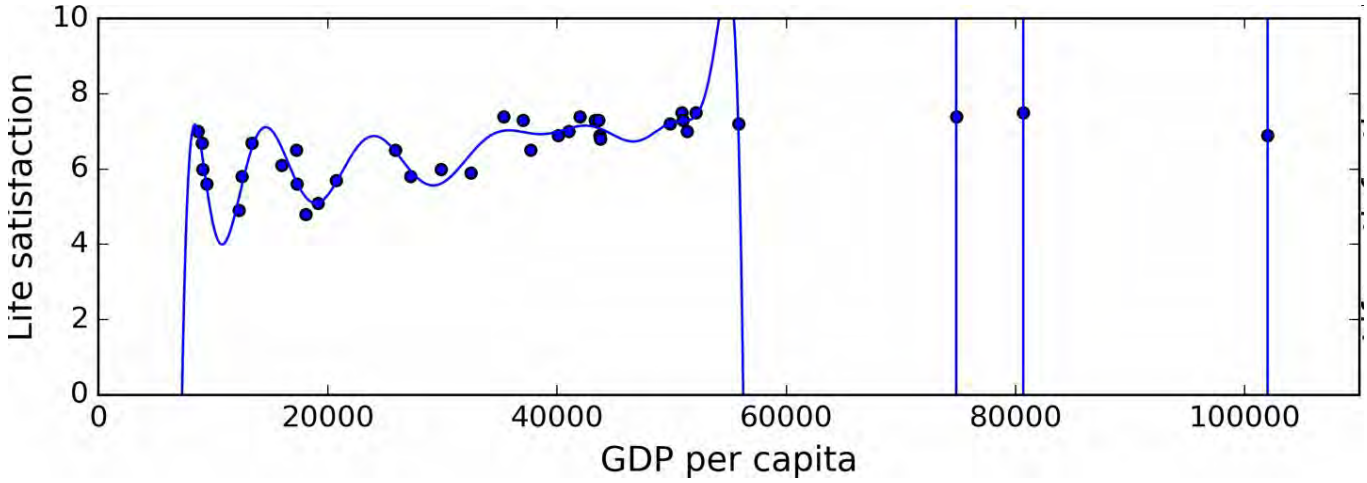


图1-22 过拟合训练数据

复杂的模型，比如深度神经网络，可以检测数据中的细微规律，但是如果训练集有噪声，或者训练集太小（太小会引入样本噪声），模型就会去检测噪声本身的规律。很明显，这些规律不能推广到新实例。例如，假如你用更多的属性训练生活满意度模型，包括不包含信息的属性，比如国家的名字。如此一来，负责的模型可能会检测出训练集中名字有w字母的国家的生活满意度大于7：新西兰（7.3），挪威（7.4），瑞典（7.2）和瑞士（7.5）。你能相信这个W-满意度法则推广到卢旺达和津巴布韦吗？很明显，这个规律只是训练集数据中偶然出现的，但是模型不能判断这个规律是真实的、还是噪声的结果。

> 警告：过拟合发生在相对于训练数据的量和噪声，模型过于复杂的情况。可能的解决方案有：
>
> - 简化模型，可以通过选择一个参数更少的模型（比如使用线性模型，而不是高阶多项式模型）、减少训练数据的属性数、或限制一下模型
> - 收集更多的训练数据
> - 减小训练数据的噪声（比如，修改数据错误和去除异常值）

限定一个模型以让它更简单，降低过拟合的风险被称作正规化（regularization）。例如，我们之前定义的线性模型有两个参数，θ_0和θ_1。它给了学习算法两个自由度以让模型适应训练数据：可以调整截距θ_0和斜率θ_1。如果强制$\theta_1=0$，算法就只剩一个自由度，拟合数据就会更为困难：能做的只是将线上下移动，尽可能地靠近训练实例，结果会在平均值附近。这就是一个非常简单的模型！如果我们允许算法可以修改θ_1，但是只能在一个很小的范围内修改，算法的自由度就会介于1和2之间。它要比两个自由度的模型简单，比1个自由度的模型要复杂。你的目标是在完美拟合数据和保持模型简单性上找到平衡，确保算法的推广效果。

图1-23展示了三个模型：虚线表示用缺失部分国家的数据训练的原始模型，短划线是我们的第二个用所有国家训练的模型，实线模型的训练数据和第一个相同，但进行了正规化限制。你可以看到正规化强制模型有一个小的斜率，它对训练数据的拟合不是那么好，但是对新样本的推广效果好。

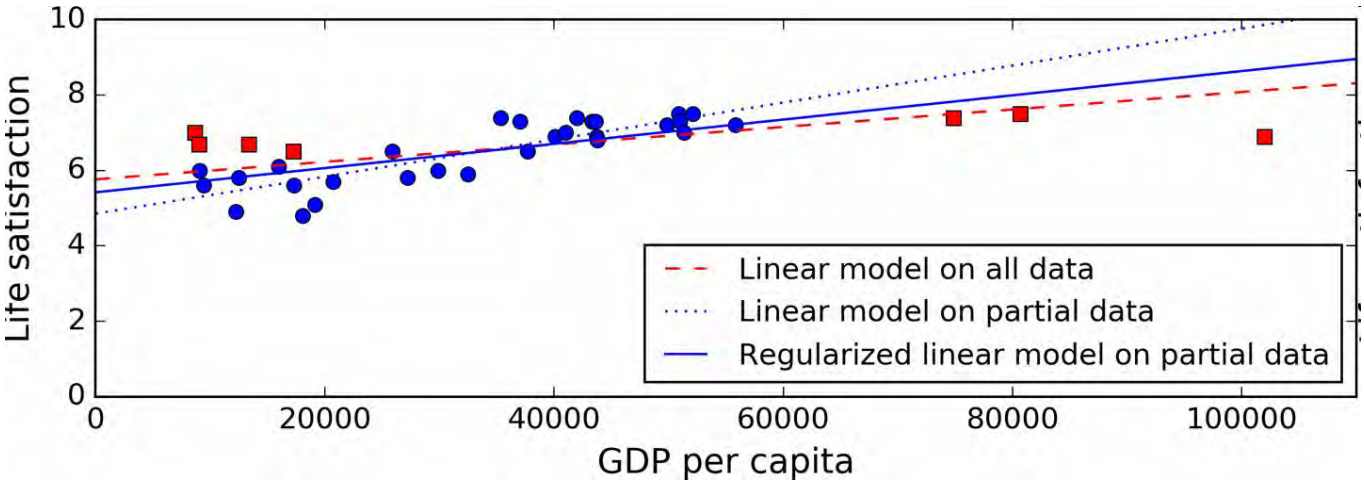


图1-23 正规化降低了过度拟合的风险

正规化的度可以用一个超参数（hyperparameter）控制。超参数是一个学习算法的参数（而不是模型的）。这样，它是不会被学习算法本身影响的，它的优于训练数据，在训练中是保持不变的。如果你设定的超参数非常大，就会得到一个几乎是平的模型（斜率接近于0）；学习算法几乎肯定不会过拟合训练数据，但是也很难得到一个好的解。调节超参数是创建机器学习算法非常重要的一部分（下一章你会看到一个详细的例子）。

欠拟合训练数据

你可能猜到了，欠拟合是和过拟合相对的：当你的模型过于简单时就会发生。例如，生活满意度的线性模型倾向于欠拟合；现实要比这个模型复杂的多，所以预测很难准确，即使在训练样本上也很难准确。

解决这个问题的选项包括：

- 选择一个更强大的模型，带有更多参数
- 用更好的特征训练学习算法（特征工程）
- 减小对模型的限制（比如，减小正规化超参数）

回顾

现在，你已经知道了很多关于机器学习的知识。然而，学过了这么多概念，你可能会感到有些迷失，所以让我们退回去，回顾一下重要的：

- 机器学习是让机器通过学习数据对某些任务做得更好，而不使用确定的代码规则。
- 有许多不同类型的机器学习系统：监督或非监督，批量或线上，基于实例或基于模型，等等。
- 在机器学习项目中，训练集来收集数据，然后对学习算法进行训练。如果算法是基于模型的，就调节一些参数，让模型拟合到训练集（即，对训练集本身作出好的预测），然后希望它对新样本也能有好预测。如果算法是基于实例的，就是用记忆学习样本，然后用相似度推广到新实例。

- 如果训练集太小、数据没有代表性、含有噪声、或掺有不相关的特征（垃圾进，垃圾出），系统的性能不会好。最后，模型不能太简单（会发生欠拟合）或太复杂（会发生过拟合）。

还差最后一个主题要学习：训练完了一个模型，你不只希望将它推广到新样本。你想评估它，作出必要的微调。一起来看一看。

测试和确认

要知道一个模型推广到新样本的效果，唯一的办法就是真正的进行试验。一种方法是将模型部署到生产环境，观察它的性能。这么做可以，但是如果模型的性能很差，就会引起用户抱怨 —— 这不是最好的方法。

更好的选项是将你的数据分成两个集合：训练集合测试集。正如它们的名字，用训练集进行训练，用测试集进行测试。对新样本的错误率称作推广错误（或样本外错误），通过模型对测试集的评估，你可以预估这个错误。这个值可以告诉你，你的模型对新样本的性能。

如果训练错误率低（即，你的模型在训练集上错误不多），但是推广错误率高，意味着模型对训练数据过拟合。

> 提示：一般使用80%的数据进行训练，保留20%用于测试。

因此，评估一个模型很简单：只要使用测试集。现在假设你在两个模型之间犹豫不决（比如一个线性模型和一个多项式模型）：如何做决定呢？一种方法是两个都训练，，然后比较在测试集上的效果。

现在假设线性模型的效果更好，但是你想做一些正规化以避免过拟合。问题是：如何选择正规化超参数的值？一种选项是用100个不同的超参数训练100个不同的模型。假设你发现最佳的超参数的推广错误率最低，比如只有5%。然后就选用这个模型作为生产环境，但是实际中性能不佳，误差率达到了15%。发生了什么呢？

答案在于，你在测试集上多次测量了推广误差率，调整了模型和超参数，以使模型最适合这个集合。这意味着模型对新数据的性能不会高。

这个问题通常的解决方案是，再保留一个集合，称作验证集合。用测试集和多个超参数训练多个模型，选择在验证集上有最佳性能的模型和超参数。当你对模型满意时，用测试集再做最后一次测试，以得到推广误差率的预估。

为了避免“浪费”过多训练数据在验证集上，通常的办法是使用交叉验证：训练集分成互补的子集，每个模型用不同的子集训练，再用剩下的子集验证。一旦确定模型类型和超参数，最终的模型使用这些超参数和全部的训练集进行训练，用测试集得到推广误差率。

> 没有免费午餐公理
>
> 模型是观察的简化版本。简化意味着舍弃无法进行推广的表面细节。但是，要确定舍弃什么数据、保留什么数据，必须要做假设。例如，线性模型的假设是数据基本上是线性的，实例和模型直线间的距离只是噪音，可以放心忽略。
>
> 在一篇1996年的著名论文（goo.gl/3zaHIZ）中，David Wolpert证明，如果完全不对数据做假设，就没有理由选择一个模型而不选另一个。这称作没有免费午餐（NFL）公理。对于一些数据集，最佳模型是线性模型，而对其它数据集是神经网络。没有一个模型可以保证效果更好（如这个公理的名字所示）。确信的唯一方法就是测试所有的模型。因为这是不可能的，实际中就必须要做一些对数据合理的假设，只评估几个合理的模型。例如，对于简单任务，你可能是用不同程度的正规化评估线性模型，对于复杂问题，你可能要评估几个神经网络模型。

练习

本章中，我们学习了一些机器学习中最为重要的概念。下一章，我们会更加深入，并写一些代码。开始下章之前，确保你能回答下面的问题：

1. 如何定义机器学习？
2. 机器学习可以解决的四类问题？
3. 什么是带标签的训练集？
4. 最常见的两个监督任务是什么？
5. 指出四个常见的非监督任务？
6. 要让一个机器人能在各种未知地形行走，你会采用什么机器学习算法？
7. 要对你的顾客进行分组，你会采用哪类算法？
8. 垃圾邮件检测是监督学习问题，还是非监督学习问题？
9. 什么是线上学习系统？
10. 什么是核外学习？
11. 什么学习算法是用相似度做预测？
12. 模型参数和学习算法的超参数的区别是什么？
13. 基于模型学习的算法搜寻的是什么？最成功的策略是什么？基于模型学习如何做预测？
14. 机器学习的四个主要挑战是什么？
15. 如果模型在训练集上表现好，但推广到新实例表现差，问题是什么？给出三个可能的解决方案。
16. 什么是测试集，为什么要使用它？
17. 验证集的目的是什么？
18. 如果用测试集调节超参数，会发生什么？
19. 什么是交叉验证，为什么它比验证集好？

练习答案见附录A。

二、端到端的机器学习

一、简介

Scikit-learn集成了很多机器学习需要使用的函数，学习Scikit-learn能简洁、快速写出机器学习程序。本文章主要是对真实数据进行实战，手把手带你走一遍使用机器学习对真实数据进行处理的全过程。并且通过代码更加深入的了解机器学习模型，学习如何处理数据，如何选择模型，如何选择和调整模型参数。

二、配置必要的环境

1、推荐安装Anaconda（集成Python和很多有用的Package）

2、编辑器：Spyder 或 Pycharm 或 Jupyter Notebook

三、开始实战（处理CSV表格数据）

1、下载数据

数据集为房屋信息housing，代码运行后，会下载一个tgz文件，然后用tarfile解压，解压后目录中会有一个housing.scv文件（可以自行用excel打开看看），下载代码为：

```
import os
import tarfile
```

```
from six.moves import urllib
DOWNLOAD_ROOT = "https://raw.githubusercontent.com/ageron/handson-ml/master/"
HOUSING_PATH = "datasets/housing"
HOUSING_URL = DOWNLOAD_ROOT + HOUSING_PATH + "/housing.tgz"
def fetch_housing_data(housing_url=HOUSING_URL, housing_path=HOUSING_PATH):
    if not os.path.isdir(housing_path):
        os.makedirs(housing_path)
        tgz_path = os.path.join(housing_path, "housing.tgz")
    urllib.request.urlretrieve(housing_url, tgz_path)
    housing_tgz = tarfile.open(tgz_path)
    housing_tgz.extractall(path=housing_path)
    housing_tgz.close()
fetch_housing_data()
```

2、读入数据

通过panda库读取csv文件。

```
import pandas as pd
def load_housing_data(housing_path=HOUSING_PATH):
    csv_path = os.path.join(housing_path, "housing.csv")
    return pd.read_csv(csv_path)
housing = load_housing_data()
```

3、观察数据

载入数据以后，首先就是要观察数据是否成功导入，是否存在缺失值，是否存在异常值，数据的特征呈现何种分布等。

head()输出前5个数据和表头

head()可以查看数据是否成功导入，并可以查看数据包含哪些特征以及特征的形式大概是怎么样的。

```
housing.head()
```

输出结果

```
In [7]: housing.head()
Out[7]:
   longitude  latitude  housing_median_age  total_rooms  total_bedrooms  \
0    -122.23     37.88                41.0        880.0           129.0
1    -122.22     37.86                21.0       7099.0          1106.0
2    -122.24     37.85                52.0       1467.0           190.0
3    -122.25     37.85                52.0       1274.0           235.0
4    -122.25     37.85                52.0       1627.0           280.0

   population  households  median_income  median_house_value ocean_proximity
0       322.0       126.0         8.3252            452600.0        NEAR BAY
1      2401.0      1138.0         8.3014            358500.0        NEAR BAY
2       496.0       177.0         7.2574            352100.0        NEAR BAY
3       558.0       219.0         5.6431            341300.0        NEAR BAY
4       565.0       259.0         3.8462            342200.0        NEAR BAY
```

info() 输出每个特征的元素总个数以及类型信息等

info()可以查看每个特征的元素总个数，因此可以查看某个特征是否存在缺失值。还可以查看数据的类型以及内存占用情况。

```
housing.info()
```

输出结果

```
In [8]: housing.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 20640 entries, 0 to 20639
Data columns (total 10 columns):
longitude             20640 non-null float64
latitude              20640 non-null float64
housing_median_age    20640 non-null float64
total_rooms           20640 non-null float64
total_bedrooms        20433 non-null float64
population            20640 non-null float64
households            20640 non-null float64
median_income         20640 non-null float64
median_house_value    20640 non-null float64
ocean_proximity       20640 non-null object
dtypes: float64(9), object(1)
memory usage: 1.6+ MB
```

可以看到total_bedrooms特征总个数为20433，而不是20640，所以存在缺失值。除了ocean_proximity为object类型（一般为一些文字label）以外，其余特征都为浮点型(float64)

value_counts() 统计特征中每个元素的总个数

value_counts()一般用在统计有有限个元素的特征（如标签label，地区等）

```
housing["ocean_proximity"].value_counts()
```

输出结果

```
In [9]: housing["ocean_proximity"].value_counts()
Out[9]:
<1H OCEAN     9136
INLAND        6551
NEAR OCEAN    2658
NEAR BAY      2290
ISLAND           5
Name: ocean_proximity, dtype: int64
```

可以看到ocean_proximity特征元素分为5类，以及每一类的总个数。

describe() 可以看实数特征的统计信息

describe()可以看实数特征的最大值、最小值、平均值、方差、总个数、25%，50%，75%小值。

```
housing.describe()
```

输出结果

Out[10]:

	longitude	latitude	housing_median_age	total_rooms	\
count	20640.000000	20640.000000	20640.000000	20640.000000	
mean	-119.569704	35.631861	28.639486	2635.763081	
std	2.003532	2.135952	12.585558	2181.615252	
min	-124.350000	32.540000	1.000000	2.000000	
25%	-121.800000	33.930000	18.000000	1447.750000	
50%	-118.490000	34.260000	29.000000	2127.000000	
75%	-118.010000	37.710000	37.000000	3148.000000	
max	-114.310000	41.950000	52.000000	39320.000000	

	total_bedrooms	population	households	median_income	\
count	20433.000000	20640.000000	20640.000000	20640.000000	
mean	537.870553	1425.476744	499.539680	3.870671	
std	421.385070	1132.462122	382.329753	1.899822	
min	1.000000	3.000000	1.000000	0.499900	
25%	NaN	787.000000	280.000000	2.563400	
50%	NaN	1166.000000	409.000000	3.534800	
75%	NaN	1725.000000	605.000000	4.743250	
max	6445.000000	35682.000000	6082.000000	15.000100	

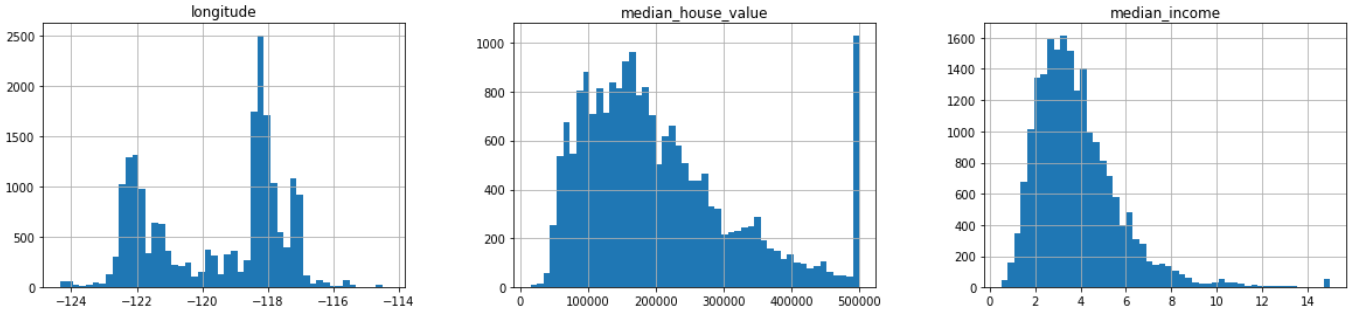
其中count为总个数，mean为平均值，std为标准差，min为最小值，max为最大值，25%，50%，75%为第25%，50%，75%的最小值。

hist() 输出实数域的直方图

同过hist()生成直方图，能够查看实数特征元素的分布情况。

```
import matplotlib.pyplot as plt
housing.hist(bins=50, figsize=(20,15))
plt.show()
```

输出结果



可以看到第一个图的分布为两个峰；第二、三个图近似为长尾分布（Long-tailed distributions）。

需要注意：hist()函数需要配合matplotlib包使用

4、分开训练和测试集

为了最终验证模型是否具有推广泛化能力，需要分开训练集于测试集，假设将数据集分为80%训练，20%测试。下面为一种普遍的分开数据集的代码：

```
import numpy as np
def split_train_test(data, test_ratio):
    shuffled_indices = np.random.permutation(len(data))
    test_set_size = int(len(data) * test_ratio)
    test_indices = shuffled_indices[:test_set_size]
```

```
    train_indices = shuffled_indices[test_set_size:]
    return data.iloc[train_indices], data.iloc[test_indices]

train_set, test_set = split_train_test(housing, 0.2)
print(len(train_set), "train +", len(test_set), "test")
```

这虽然能正确的分开训练、测试集，但是如果重新运行程序，训练和测试集会不一样。假设在原来模型的基础上继续训练，则不能保证测试集没有被模型训练过，因此不能验证模型效果。下面有两种方案：

方案一：使用在shuffle之前（即permutation），调用np.random.seed(42)，则每次运行shuffle的结果一样（即训练、测试集一样）。但是如果新增加了一些数据集，则这个方案将不可用。

方案二：为了解决方案一的问题，采用每个样本的识别码（可以是ID，可以是行号）来决定是否放入测试集，例如计算识别码的hash值，取hash值得最后一个字节（0~255），如果该值小于一个数（20% * 256）则放入测试集。这样，这20%的数据不会包含训练过的样本。具体代码如下：

```
def test_set_check(identifier, test_ratio, hash):
    return hash(np.int64(identifier)).digest()[-1] < 256 * test_ratio
def split_train_test_by_id(data, test_ratio, id_column, hash=hashlib.md5):
    ids = data[id_column]
    in_test_set = ids.apply(lambda id_: test_set_check(id_, test_ratio, hash))
    return data.loc[~in_test_set], data.loc[in_test_set]

housing_with_id = housing.reset_index() # adds an `index` column
train_set, test_set = split_train_test_by_id(housing_with_id, 0.2, "index")
```

如果用行号作识别码，需要保证新的数据放在之前的数据以后，而且没有行被删除。如果没有办法做到以上两条准则，则可以应该使用更加稳定的特征作为识别码，例如一个地区的经纬度（longitude 和 latitude）。

```
housing_with_id["id"] = housing["longitude"] * 1000 + housing["latitude"]
train_set, test_set = split_train_test_by_id(housing_with_id, 0.2, "id")
```

简洁、方便的Scikit-Learn 也提供了相关的分开训练和测试集的函数。

```
from sklearn.model_selection import train_test_split
train_set, test_set = train_test_split(housing, test_size=0.2, random_state=42)
```

参数和之前几乎相同，random_state为0或没有时为每次随机的情况，42时为seed的情况。

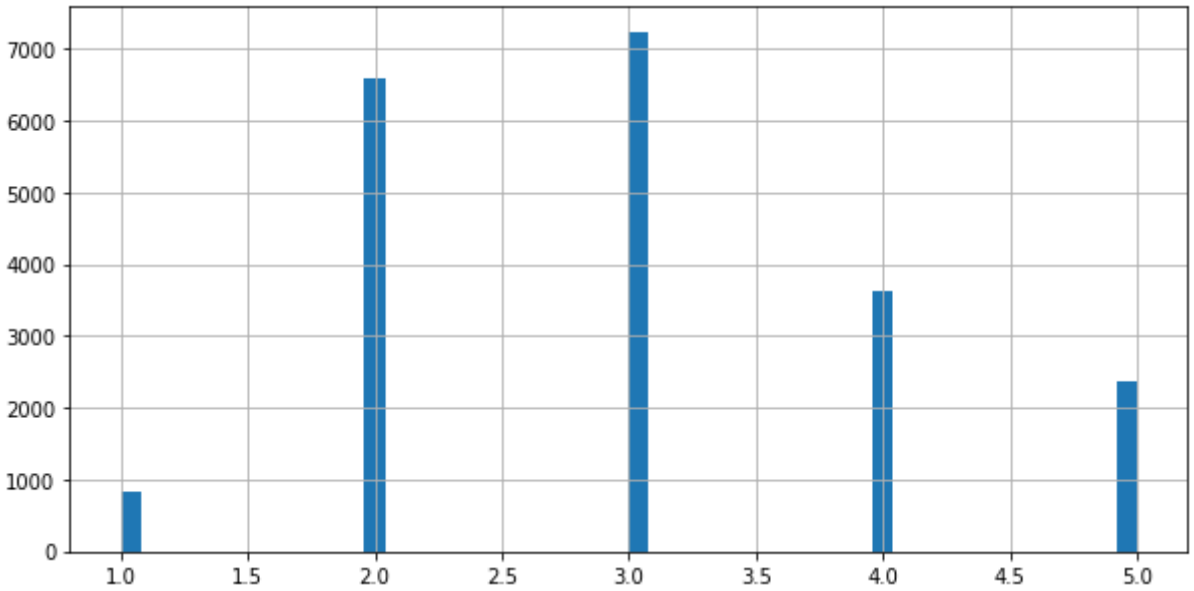
需要注意：如果没有找到model_selection，请将sklearn更新到最新版本（pip install -U sklearn）

由于上面的情况都只是考虑纯随机采样，如果样本很大，则能表现良好，如果样本比较少，则会有采样偏差的风险。比如要对1000个人做问卷调查，社会人口男女比例为51.3%和48.7%，则采样人数按照这个比例，则应该为513和487，这就是分层采样。如果纯随机采样（即上述按行号，识别码的随机采样），则有12%的可能测试集中女性少于49%或男性多于54%。这样的话就会产生采样偏差。因此sklearn提供了另一个函数StratifiedShuffleSplit（分层随机采样）

继续从真实数据来看，假设专家告诉你median_income 是用于预测median housing price一个很重要的特征，则你想把median_income作为划分的准则来观察不同的median_income对median housing price的影响。但是可以看到median_income是连续实数值。所以需要把median_income变为类别属性。

根据之前显示的图标表，除以1.5界分为5类，除了以后大于5的归为5，下面图片可以上述说过的hist()函数画出来看看，对比一下原来的median_income的分布，看是否相差较大，如果较大，则界需要调整。

```
housing["income_cat"] = np.ceil(housing["median_income"] / 1.5)
housing["income_cat"].where(housing["income_cat"] < 5, 5.0, inplace=True)
```



接下来就可以根据上面分号层的”income_cat”使用StratifiedShuffleSplit函数作分层采样，其中n_splits为分为几组样本（如果需要交叉验证，则n_splits可以取大于1，生成多组样本），其他参数和之前相似。

```
from sklearn.model_selection import StratifiedShuffleSplit
split = StratifiedShuffleSplit(n_splits=1, test_size=0.2, random_state=42)
for train_index, test_index in split.split(housing, housing["income_cat"]):
    strat_train_set = housing.loc[train_index]
    strat_test_set = housing.loc[test_index]
```

最后我们来比较一下分层采样和随机采样的结果比例情况

	Overall	Random	Stratified	Rand. %error	Strat. %error
1.0	0.039826	0.040213	0.039738	0.973236	-0.219137
2.0	0.318847	0.324370	0.318876	1.732260	0.009032
3.0	0.350581	0.358527	0.350618	2.266446	0.010408
4.0	0.176308	0.167393	0.176399	-5.056334	0.051717
5.0	0.114438	0.109496	0.114369	-4.318374	-0.060464

从表格中可以看到纯随机采样产生的采样偏差还是比较大的。

由于income_cat特征只是我们用于划分的特征，对训练没有任何作用，所以最后需要将加入的income_cat删除

```
for set in (strat_train_set, strat_test_set):
    set.drop(["income_cat"], axis=1, inplace=True)
```

下一篇：机器学习实战(用Scikit-learn和TensorFlow进行机器学习)(二)

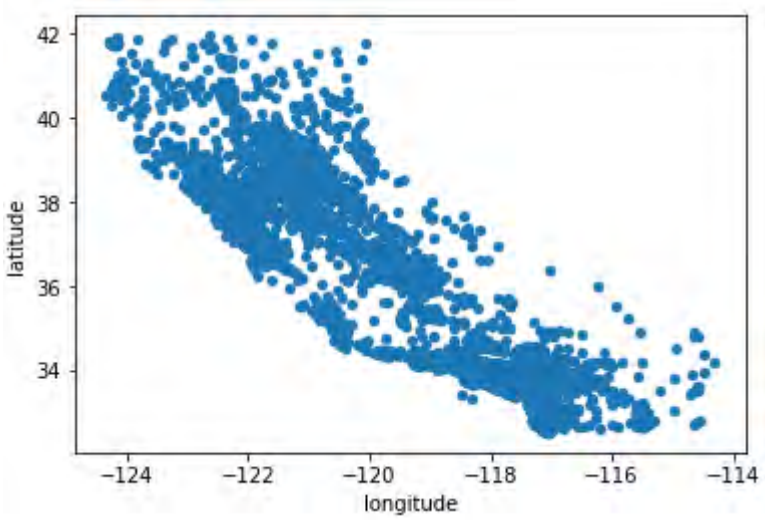
5、查看训练集的特征图像信息以及特征之间的相关性

上一节粗略地查看了数据的统计信息，接下来需要从训练样本中得到更多的信息，从而对数据进行一些处理。

查看训练集的特征图像信息

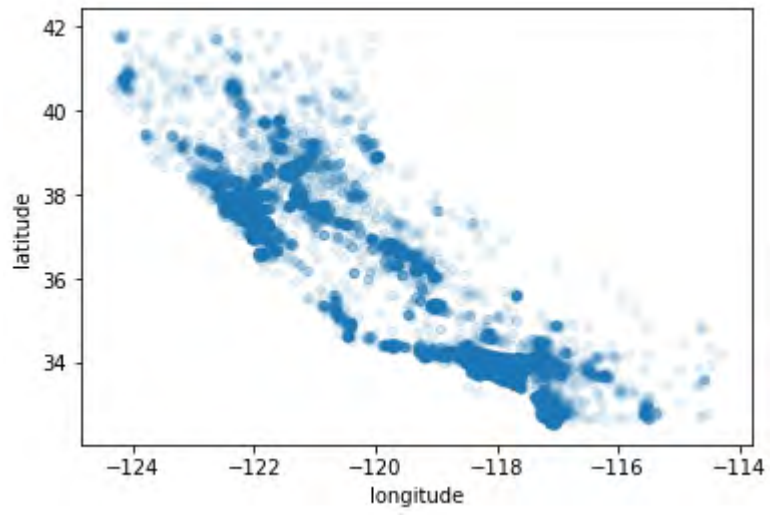
为了防止误操作在查看的时候修改了训练集，所以先复制一份进行操作。对longitude和latitude（经纬度）以散点（scatter）的形式输出，看数据的地区分布。

```
train_housing = strat_train_set.copy()
train_housing.plot(kind="scatter", x="longitude", y="latitude")
```



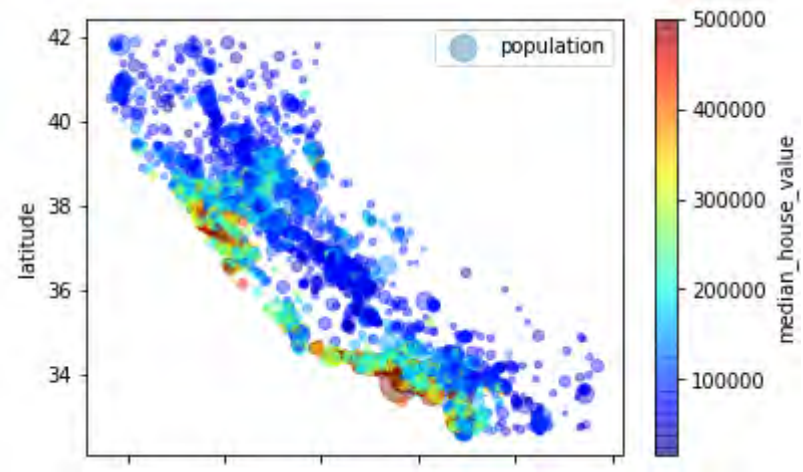
若加上参数alpha=0.1可以看到数据高密度的区域，若alpha设为1，则和上面一样，alpha越靠近0则只加深越高密度的地方。

```
train_housing.plot(kind="scatter", x="longitude", y="latitude", alpha=0.1)
```



为了查看目标median_house_value的在各个地区的分布情况，所以可以增加参数c（用于对某参数显示颜色），参数s（表示某参数圆的半径），cmap表示一个colormap，jet为一个蓝到红的颜色，colorbar为右侧颜色条

```
train_housing.plot(kind="scatter", x="longitude", y="latitude", alpha=0.4,
s=housing["population"]/100, label="population",
c="median_house_value", cmap=plt.get_cmap("jet"), colorbar=True,
)
plt.legend()
```



查看特征之间的相关性

查看median_house_value与其他变量的线性相关性，并排序输出，数据越靠近1则越相关，靠近-1则越负相关，接近0为不相关。

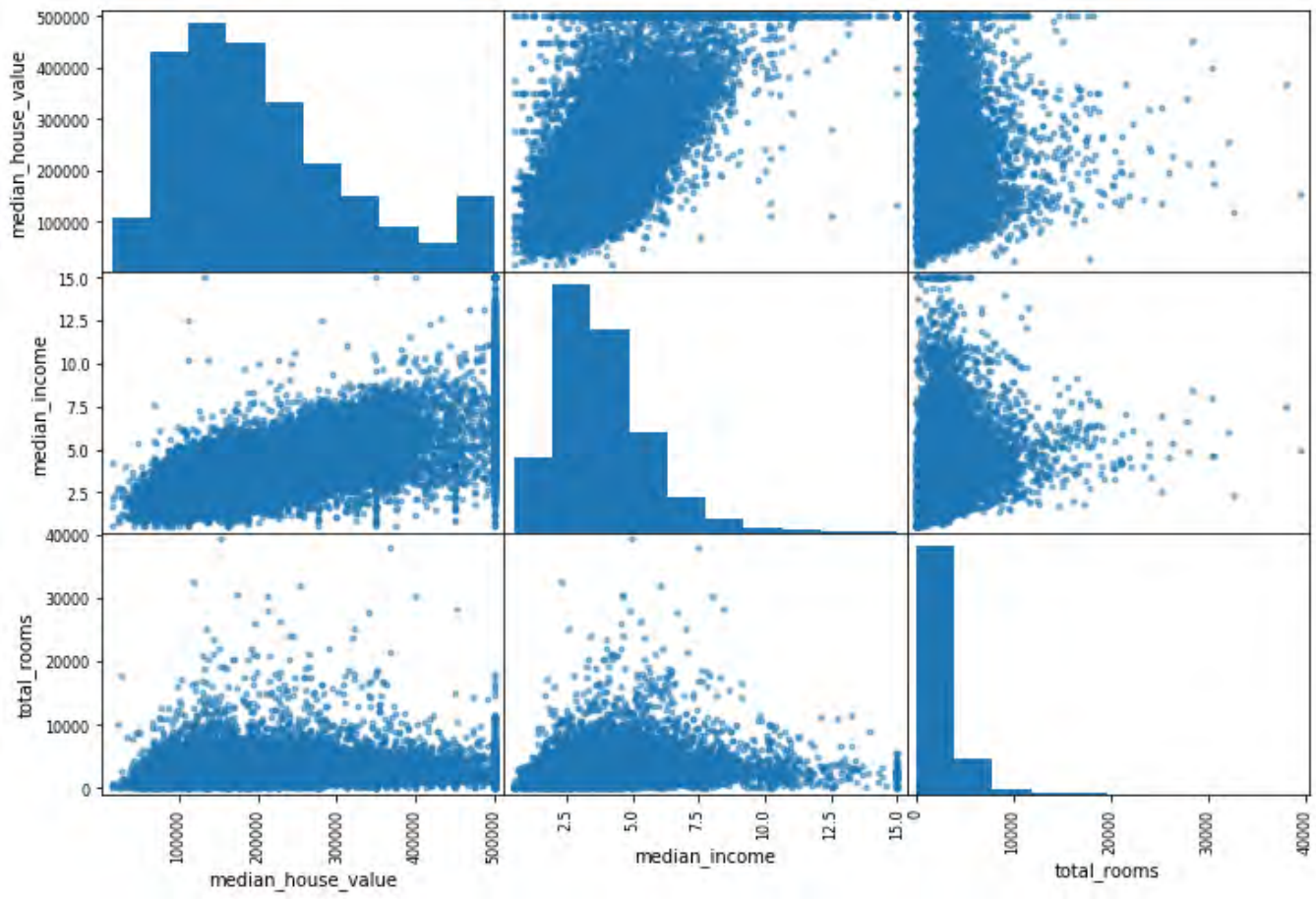
```
corr_matrix = train_housing.corr()
corr_matrix["median_house_value"].sort_values(ascending=False)
```

```
median_house_value    1.000000
median_income         0.687160
total_rooms           0.135097
housing_median_age    0.114110
households            0.064506
total_bedrooms        0.047689
population           -0.026920
longitude            -0.047432
latitude             -0.142724
Name: median_house_value, dtype: float64
```

可以看到除了与自己最相关以外，和median_income线性相关性很强。

然而上面只是计算了线性相关性，而特征之间可能是非线性的关系，因此需要画出图来看一下变量之间是否相关。（代码中只取其中的几个来看）

```
from pandas.tools.plotting import scatter_matrix
attributes = ["median_house_value", "median_income", "total_rooms"]
scatter_matrix(housing[attributes], figsize=(12, 8))
```



从第一行第二幅图，可以看到median_house_value和median_income的正线性相关性还是比较强的，但是还是看到一些问题，比如大于500000后的点可能在收集资料或预处理时设立的边界，使得变为一条直线一样；而且还有右下角一些奇异的点。为了让算法不学习到这些有问题的点，你可以去除这些相关区域的点。

特征之间的组合

两个特征对目标的相关性都不强，但是组合起来可能有较大的提升。最后还可以尝试一下特征的组合（不是特别重要）

```
train_housing["rooms_per_household"] = housing["total_rooms"]/housing["households"]
train_housing["bedrooms_per_room"] = housing["total_bedrooms"]/housing["total_rooms"]
train_housing["population_per_household"]=housing["population"]/housing["households"]
corr_matrix = train_housing.corr()
corr_matrix["median_house_value"].sort_values(ascending=False)
```

```
median_house_value          1.000000
median_income               0.687160
rooms_per_household         0.146285
total_rooms                 0.135097
housing_median_age          0.114110
households                  0.064506
total_bedrooms              0.047689
population_per_household   -0.021985
population                 -0.026920
longitude                  -0.047432
latitude                   -0.142724
bedrooms_per_room          -0.259984
Name: median_house_value, dtype: float64
```

经过特征组合，可以看到新特征bedrooms_per_room对median_house_value的影响比较大（-0.2599），呈一定的负相关，即每个房子的卧室越少，价格反而越贵。

6、准备数据（数据预处理）

首先分开特征（feature）和目标标签（label），以median_house_value作为标签，其他作为特征。

```
train_housing = strat_train_set.drop("median_house_value", axis=1)
train_housing_labels = strat_train_set["median_house_value"].copy()
```

数据清洗

在第一节我们得知total_bedrooms存在一些缺失值，对于缺失值的处理有三种方案：

1、去掉含有缺失值的个体（dropna）

2、去掉含有缺失值的整个特征（drop）

3、给缺失值补上一些值（0、平均数、中位数等）（fillna）

```
#train_housing.dropna(subset=["total_bedrooms"]) # option 1
#train_housing.drop("total_bedrooms", axis=1) # option 2
median = train_housing["total_bedrooms"].median()
train_housing["total_bedrooms"].fillna(median) # option 3
```

为了得到更多可利用的数据，在这里我们选择方案3。

当然非常方便的Scikit-Learn也存在对缺失值处理的类Imputer。我们打算对所有地方的缺失值都补全，以防运行模型时发生错误。使用Imputer函数需要先定义一个补缺失值的策略（如median），由于median策略只能对实数值有效，所以需要将文本属性先去除，然后再补缺失值。最后使用fit方法对变量执行相应操作。

```
from sklearn.preprocessing import Imputer
imputer = Imputer(strategy="median")
housing_num = train_housing.drop("ocean_proximity", axis=1)
imputer.fit(housing_num)
```

对数据缺失值补全以后，一般需要转化为Numpy的矩阵格式，方便模型的输入，所以可以调用Imputer的transform()方法，当然fit和transform也可以合起来使用，即fit_transform()，这个函数会比分开调用要快一些。

```
X = imputer.transform(housing_num)
#X = imputer.fit_transform(housing_num)

#也可以将numpy格式的转换为pd格式
#housing_tr = pd.DataFrame(X, columns=housing_num.columns)
```

处理类别文本特征

由于文本属性不能作median等操作，所以需要将文本特征编码为实数特征，对应Scikit-Learn中的类为LabelEncoder，通过调用LabelEncoder类，再使用fit_transform()方法自动将文本特征编码

```
from sklearn.preprocessing import LabelEncoder
encoder = LabelEncoder()
housing_cat = train_housing["ocean_proximity"]
housing_cat_encoded = encoder.fit_transform(housing_cat)
print(housing_cat_encoded)
print(encoder.classes_)
```

输出的数字编码与编码对应的类型为：

```
[0 0 4 ..., 1 0 3]
['<1H OCEAN' 'INLAND' 'ISLAND' 'NEAR BAY' 'NEAR OCEAN']
```

由于0到1的距离比0到3的距离要近，所以这种数字编码暗含了0和1的相似性比0到3的相似性要强，然而事实上并非如此，每个元素的相似性应趋于相等。如果该数字编码作为label，则只是一个标签，没有什么影响。但是如果用于特征，则这种数字编码不适用，应该采用one hot编码（形式可以看下面的图），对应Scikit-Learn中的类为OneHotEncoder

```
from sklearn.preprocessing import OneHotEncoder
encoder = OneHotEncoder()
housing_cat_1hot = encoder.fit_transform(housing_cat_encoded.reshape(-1,1))
```

默认的输出结果为稀疏矩阵Scipy（sparse matrix），而不是Numpy，由于矩阵大部分为0，浪费空间，所以使用稀疏矩阵存放，如果想看矩阵的具体样子，则用toarray（）方法变为dense matrix（Numpy）。

```
housing_cat_1hot.toarray()
```

```
array([[ 1.,  0.,  0.,  0.,  0.],
       [ 1.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  1.],
       ...,
       [ 0.,  1.,  0.,  0.,  0.],
       [ 1.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  1.,  0.]])
```

上述文本编码先经过数字编码再转为one hot编码用了两步，当然也可以一步到位，直接从文本编码到one hot，对应Scikit-Learn中的类为LabelBinarizer

```
from sklearn.preprocessing import LabelBinarizer
encoder = LabelBinarizer()
#encoder = LabelBinarizer(sparse_output=True)
housing_cat_1hot = encoder.fit_transform(housing_cat)
```

要注意的是，不设参数sparse_output=True的话，默认输出的是Numpy矩阵。

自定义Transformer

由于Scikit-Learn中的函数中提供的Transformer方法并不一定适用于真实情形，所以有时候需要自定义一个Transformer，与Scikit-Learn能够做到“无缝结合”，比如pineline（以后会说到）。定义类时需要加入基础类：BaseEstimator（必须），以及TransformerMixin（用于自动生成fit_transformer()方法）。下面是一个例子：用于增加组合特征的Trainsformer

```
from sklearn.base import BaseEstimator, TransformerMixin
rooms_ix, bedrooms_ix, population_ix, household_ix = 3, 4, 5, 6

class CombinedAttributesAdder(BaseEstimator, TransformerMixin):
    def __init__(self, add_bedrooms_per_room = True): # no *args or **kargs
        self.add_bedrooms_per_room = add_bedrooms_per_room
    def fit(self, X, y=None):
        return self # nothing else to do
    def transform(self, X, y=None):
        rooms_per_household = X[:, rooms_ix] / X[:, household_ix]
        population_per_household = X[:, population_ix] / X[:, household_ix]
        if self.add_bedrooms_per_room:
            bedrooms_per_room = X[:, bedrooms_ix] / X[:, rooms_ix]
            return np.c_[X, rooms_per_household, population_per_household,
                         bedrooms_per_room]
        else:
            return np.c_[X, rooms_per_household, population_per_household]

attr_adder = CombinedAttributesAdder(add_bedrooms_per_room=False)
housing_extra_attribs = attr_adder.transform(housing.values)
```

在该代码中，init 中设置参数，可以调整是否加入该元素，用于尝试确定加入该元素是否对模型效果提升，方便修改，节省时间。

特征缩放

由于机器学习算法在不同尺度范围的特征之间表现的不好，比如total number of rooms范围是6-39320，而median_incomes范围是0-15。因此需要对特征的范围进行缩放，对应Scikit-Learn中的类为：

1、MinMaxScaler：将特征缩放到0-1之间，但异常值对这个影响比较大，比如异常值为100，缩放0-15为0-0.15;

2、feature_range：可以自定义缩放的范围，不一定是0-1;

3、StandardScaler：标准化（减均值，除方差），对异常值的影响较小，但可能会不符合某种范围

需要注意：每次缩放只能针对训练集或只是测试集，而不能是整个数据集，这是由于测试集（或新数据）不属于训练范围。

Transformation Pipelines

可以看到，上述有非常多的转换操作，并按一定的顺序执行，但是再次处理其他数据（如测试数据）时需要重新调用执行众多步骤，代码看起来过于繁琐。所以Scikit-Learn提供了Pineline类来帮助这种一系列的转换，把这些转换封装为一个转换。下面是一个简单的例子。

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
num_pipeline = Pipeline([
('imputer', Imputer(strategy="median")),
('attribs_adder', CombinedAttributesAdder()),
('std_scaler', StandardScaler()),
```

```
])
housing_num_tr = num_pipeline.fit_transform(housing_num)
```

Pipeline是由（name(名字)，Estimator(类)）对组成，但最后一个必须为transformer，这是因为要形成fit_transform()方法

上面的pineline只是用于处理实数特征的，对于处理类别特征的还有另一个Pineline，这就可以使用FearureUnion类来结合多个pineline，多个Pineline可以并行处理，最后将结果拼接在一起输出。

由于Scikit-Learn没有处理Pandas数据的DataFrame，因此需要自己自定义一个如下：

```
from sklearn.base import BaseEstimator, TransformerMixin
class DataFrameSelector(BaseEstimator, TransformerMixin):
    def __init__(self, attribute_names):
        self.attribute_names = attribute_names
    def fit(self, X, y=None):
        return self
    def transform(self, X):
        return X[self.attribute_names].values
```

然后就可以通过FeatureUnion类结合两个Pineline

```
from sklearn.pipeline import FeatureUnion
num_attribs = list(housing_num)
cat_attribs = ["ocean_proximity"]
num_pipeline = Pipeline([
('selector', DataFrameSelector(num_attribs)),
('imputer', Imputer(strategy="median")),
('attribs_adder', CombinedAttributesAdder()),
('std_scaler', StandardScaler()),
])
cat_pipeline = Pipeline([
('selector', DataFrameSelector(cat_attribs)),
('label_binarizer', LabelBinarizer()),
])
full_pipeline = FeatureUnion(transformer_list=[
("num_pipeline", num_pipeline),
("cat_pipeline", cat_pipeline),
])
housing_prepared = full_pipeline.fit_transform(train_housing)
```

需要注意：在scikit-learn == 0.18.0及以前版本LabelBinarizer()用在Pineline没有问题；而在0.19.0版本则会报错，因此需要自己定义一个新的LabelBinarizer_new()，代码如下；0.20.0版本以后可以使用新的类CategoricalEncoder()

```
from sklearn.base import BaseEstimator, TransformerMixin
class LabelBinarizer_new(TransformerMixin, BaseEstimator):
    def fit(self, X, y = 0):
        self.encoder = None
        return self
    def transform(self, X, y = 0):
```

```
        if(self.encoder is None):
            print("Initializing encoder")
            self.encoder = LabelBinarizer();
            result = self.encoder.fit_transform(X)
        else:
            result = self.encoder.transform(X)
        return result
```

7、选择及训练模型

首先尝试训练一个线性回归模型（LinearRegression ）

```
from sklearn.linear_model import LinearRegression
lin_reg = LinearRegression()
lin_reg.fit(train_housing_prepared, train_housing_labels)
```

训练完成，然后评估模型，计算训练集中的均方误差（RMSE）

```
from sklearn.metrics import mean_squared_error
housing_predictions = lin_reg.predict(train_housing_prepared)
lin_mse = mean_squared_error(train_housing_labels, housing_predictions)
lin_rmse = np.sqrt(lin_mse)
lin_rmse
```

```
Out[32]: 68628.198198489219
```

可以看到线性回归模型的训练集均方误差为68626

再试试看更强大的模型，决策树模型（DecisionT reeRegressor ）

```
from sklearn.tree import DecisionTreeRegressor
tree_reg = DecisionTreeRegressor()
tree_reg.fit(train_housing_prepared, train_housing_labels)
housing_predictions = tree_reg.predict(train_housing_prepared)
tree_mse = mean_squared_error(train_housing_labels, housing_predictions)
tree_rmse = np.sqrt(tree_mse)
tree_rmse
```

```
Out[150]: 0.0
```

可以看到决策树回归模型的的训练集均方误差竟然为0。比线性回归模型的的训练集均方误差小太多太多。

但这是否说明了决策树回归模型比线性回归模型在此问题上好很多，当然不是，训练误差小的模型并不代表为好模型，这是因为模型可能过度地学习了训练集的数据，只是在训练集上的表现好（即过拟合），一旦测试新的数据表现就会很差。

因此在训练的时候需要将部分的训练数据提取出来作为验证集，验证该模型是否对此问题适用。其中比较常用的就是交叉验证法。

交叉验证法

交叉验证的基本思想是将训练数据集分为k份，每次用k-1份训练模型，用剩余的1份作为验证集。按顺序训练k次后，计算k次的平均误差来评价模型（改变参数后即为另一个模型）的好坏。（具体做法可以看百度百科）

在Scikit-Learn中交叉验证对应的类为cross_val_score，下面是线性回归模型与决策树回归模型的交叉验证实例：

```
from sklearn.model_selection import cross_val_score
tree_scores = cross_val_score(tree_reg, train_housing_prepared, train_housing_labels,
scoring="neg_mean_squared_error", cv=10)
lin_scores = cross_val_score(lin_reg, train_housing_prepared, train_housing_labels,
scoring="neg_mean_squared_error", cv=10)
tree_rmse_scores = np.sqrt(-tree_scores)
lin_rmse_scores = np.sqrt(-lin_scores)

def display_scores(scores):
    print("Scores:", scores)
    print("Mean:", scores.mean())
    print("Standard deviation:", scores.std())

display_scores(tree_rmse_scores)
display_scores(lin_rmse_scores)
```

其中参数scoring为选择一个指标，代码中选的为均方误差；参数cv是交叉验证划分的个数，这里划为为10份。

需要注意：这里经过交叉验证求均方误差的结果为负值，所以后面求平方根前需要加负号。

```
('Scores:', array([ 69400.57083425,  67485.6968196 ,  69740.69826221,  69929.6630072 ,
        71316.34292745,  74830.37927816,  70704.04344147,  71197.76523943,
        76383.24879494,  70646.07722087]))
('Mean:', 71163.44858255799)
('Standard deviation:', 2477.4183500597069)
('Scores:', array([ 66756.89288533,  66963.71866078,  70347.95244419,  74752.93602822,
        68031.13388938,  71204.4762852 ,  64967.81468897,  68273.02578122,
        71552.91566558,  67661.80150892]))
('Mean:', 69051.266783778803)
('Standard deviation:', 2737.8553258407578)
```

可以看到决策树回归模型的交叉验证平均误差为71163，而线性回归模型的交叉验证平均误差为69051，这说明决策树回归模型明显是过拟合，实际上比线性回归模型要差一些。

除了这两个简单的模型以外，还应该试验不同的模型（如随机森林，不同核的SVM，神经网络等），最终选择2-5个候选的模型。（也可以写到同一个文件下，方便以后直接调用）

保存模型

最后介绍一下如何保存模型到本地（硬盘）与重新加载本地模型，可以使用Pickle库，也可以使用scikit-learn中的joblib库，具体代码如下：

```
from sklearn.externals import joblib
joblib.dump(my_model, "my_model.pkl") #保存模型
# and later...
my_model_loaded = joblib.load("my_model.pkl") #加载模型
```

8、模型调参

现在已经有一些候选的模型，你需要对模型的参数进行微调，使模型表现的更好。下面介绍几种调参方法

网格搜索（Grid Search ）

scikit-learn中提供函数GridSearchCV用于网格搜索调参，网格搜索就是通过自己对模型需要调整的几个参数设定一些可行值，然后Grid Search会排列组合这些参数值，每一种情况都去训练一个模型，经过交叉验证今后输出结果。下面为随机森林回归模型（RandomForestRegression）的一个Grid Search的例子。

```
from sklearn.model_selection import GridSearchCV
param_grid = [
{'n_estimators': [3, 10, 30], 'max_features': [2, 4, 6, 8]},
{'bootstrap': [False], 'n_estimators': [3, 10], 'max_features': [2, 3, 4]},
]
forest_reg = RandomForestRegressor()
grid_search = GridSearchCV(forest_reg, param_grid, cv=5,
scoring='neg_mean_squared_error')
grid_search.fit(train_housing_prepared, train_housing_labels)
```

例子中首先调第一行的参数为n_estimators和max_features，即有3*4=12种组合，然后再调第二行的参数，即2*3=6种组合，具体参数的代表的意思以后再讲述。总共组合数为12+6=18种组合。每种交叉验证5次，即18*5=90次模型计算，虽然运算量比较大，但运行完后能得到较好的参数。

输出最好的参数

```
grid_search.best_params_
```

```
Out[37]: {'max_features': 6, 'n_estimators': 30}
```

可以看到最好参数中30是选定参数的边缘，所以可以再选更大的数试验，可能会得到更好的模型，还可以在6附近选定参数，也可能会得到更好的模型。

输出最好参数的模型

```
grid_search.best_params_
```

```
Out[38]:
RandomForestRegressor(bootstrap=True, criterion='mse', max_depth=None,
           max_features=6, max_leaf_nodes=None, min_impurity_decrease=0.0,
           min_impurity_split=None, min_samples_leaf=1,
           min_samples_split=2, min_weight_fraction_leaf=0.0,
           n_estimators=30, n_jobs=1, oob_score=False, random_state=None,
           verbose=0, warm_start=False)
```

也可以看看每一个组合分别的交叉验证的结果

```
cvres = grid_search.cv_results_
... for mean_score, params in zip(cvres["mean_test_score"], cvres["params"]):
... print(np.sqrt(-mean_score), params)
```

```
    ....
(63158.137805988255, {'max_features': 2, 'n_estimators': 3})
(55928.355873156768, {'max_features': 2, 'n_estimators': 10})
(53030.410636141038, {'max_features': 2, 'n_estimators': 30})
(60519.664187847149, {'max_features': 4, 'n_estimators': 3})
(53089.064295580465, {'max_features': 4, 'n_estimators': 10})
(50857.207290481238, {'max_features': 4, 'n_estimators': 30})
(59039.065914604776, {'max_features': 6, 'n_estimators': 3})
(52385.452116908295, {'max_features': 6, 'n_estimators': 10})
(50184.124445302681, {'max_features': 6, 'n_estimators': 30})
(58114.455728804052, {'max_features': 8, 'n_estimators': 3})
(52053.125622224776, {'max_features': 8, 'n_estimators': 10})
(50204.164076117413, {'max_features': 8, 'n_estimators': 30})
(63113.297736484572, {'max_features': 2, 'n_estimators': 3, 'bootstrap': False})
(54535.087787284763, {'max_features': 2, 'n_estimators': 10, 'bootstrap': False})
(60054.16923547937, {'max_features': 3, 'n_estimators': 3, 'bootstrap': False})
(52604.769545660616, {'max_features': 3, 'n_estimators': 10, 'bootstrap': False})
(59162.440481902668, {'max_features': 4, 'n_estimators': 3, 'bootstrap': False})
(51712.192650442601, {'max_features': 4, 'n_estimators': 10, 'bootstrap': False})
```

随机搜索（Randomized Search ）

由于上面的网格搜索搜索空间太大，而机器计算能力不足，则可以通过给参数设定一定的范围，在范围内使用随机搜索选择参数，随机搜索的好处是能在更大的范围内进行搜索，并且可以通过设定迭代次数n_iter，根据机器的计算能力来确定参数组合的个数，是下面给出一个随机搜索的例子。

```
from sklearn.model_selection import RandomizedSearchCV
param_ran={'n_estimators':range(30,50),'max_features': range(3,8)}
forest_reg = RandomForestRegressor()
random_search = RandomizedSearchCV(forest_reg,param_ran,cv=5,scoring='neg_mean_squared_error',n_iter=
random_search.fit(train_housing_prepared, train_housing_labels)
```

分析最好的模型每个特征的重要性

假设现在调参以后得到最好的参数模型，然后可以查看每个特征对预测结果的贡献程度，根据贡献程度，可以删减减少一些不必要的特征。

```
feature_importances = grid_search.best_estimator_.feature_importances_
extra_attribs = ["rooms_per_hhold", "pop_per_hhold", "bedrooms_per_room"]
cat_one_hot_attribs = list(encoder.classes_)
```

```
attributes = num_attribs + extra_attribs + cat_one_hot_attribs
sorted(zip(feature_importances, attributes), reverse=True)
```

```
[(0.33374399821409401, 'median_income'),
 (0.14756390646053333, 'INLAND'),
 (0.099905702526762369, 'pop_per_hhold'),
 (0.082634797509351529, 'longitude'),
 (0.070612572851175712, 'latitude'),
 (0.07010459850497204, 'rooms_per_hhold'),
 (0.061502964618674529, 'bedrooms_per_room'),
 (0.041527496479690222, 'housing_median_age')
 (0.018005712718636224, 'population'),
 (0.017839021914494053, 'total_rooms'),
 (0.017020336161886099, 'total_bedrooms'),
 (0.015779980803186672, '<1H OCEAN'),
 (0.015410483053514289, 'households'),
 (0.0054839924604509933, 'NEAR OCEAN'),
 (0.0027446327556356113, 'NEAR BAY'),
 (0.00011980296694234748, 'ISLAND')]
```

可以看到ocean_proximity中的4个特征中只有一个特征是有用的，其他3个几乎没有用，所以可以考虑去除其他3个特征。

在测试集中评估

经过努力终于得到了最终的模型，现在就差在测试集上验证这个模型的泛化能力以及准确性。测试集中的操作和训练集中的操作基本相同，唯一不同的是不需要fit()，只需要transform()就可以了，这是因为测试集不是用来训练模型，所以不用fit()，所以将fit_transform()改为transform()。

```
final_model = grid_search.best_estimator_
X_test = strat_test_set.drop("median_house_value", axis=1)
y_test = strat_test_set["median_house_value"].copy()
X_test_prepared = full_pipeline.transform(X_test)
final_predictions = final_model.predict(X_test_prepared)
final_mse = mean_squared_error(y_test, final_predictions)
final_rmse = np.sqrt(final_mse)
```

```
Out[57]: 47372.925352208949
```

可以发现，结果和交叉验证以后的结果比较相似，说明经过交叉验证后，在新的数据集上也能达到类似的效果。

需要注意：在测试集中补缺失值，标准化等用到的值都是训练集上的中值，平均值等，而不是测试集上的。因为必须把数据放缩到同一尺度。

最后还可以分析这个模型学习到了什么，没做到什么，作出了什么假设，有什么局限性，得到了什么结论（比如median income是最影响结果的）

三、分类问题

下面将使用新的具有代表性的数据集MNIST（手写体数字数据集），数据集总共有70000个小图片，每个小图片为一个手写的数字，（数据中0代表白，1代表黑），数据中把28*28个像素拉成一个向量作为特征，写的数字作为label。

1、关于MNIST数据集

Scikit-learn提供了MNIST数据的下载，如果下载不了也可以自行网站上下载。

```
from sklearn.datasets import fetch_mldata
mnist = fetch_mldata('MNIST original')
```

下载完成后可以输入mnist自行查看一下数据的结构，还可以使用matplotlib输出一张图片看看。

下面需要划分训练集和测试集，MNIST数据集已经帮我们划分好（前60000个为训练集，后10000个位测试集）

```
X, y = mnist["data"], mnist["target"]
X_train, X_test, y_train, y_test = X[:60000], X[60000:], y[:60000], y[60000:]
```

虽然MNIST数据集中已经把训练测试集分好，但是还未打乱（shuffle），所以需要对训练集进行打乱。

```
import numpy as np
shuffle_index = np.random.permutation(60000)
X_train, y_train = X_train[shuffle_index], y_train[shuffle_index]
```

2、二分类

假设现在分类是否为数字5，则分类两类（是5或不是5），训练一个SGD分类器（该分类器对大规模的数据处理较快）。

```
#划分数据
y_train_5 = (y_train == 5)
y_test_5 = (y_test == 5)
#训练模型
from sklearn.linear_model import SGDClassifier
sgd_clf = SGDClassifier(random_state=42)
sgd_clf.fit(X_train, y_train_5)
#交叉验证
from sklearn.model_selection import cross_val_predict
y_train_pred = cross_val_predict(sgd_clf, X_train, y_train_5, cv=3)
```

查准率和查全率（Precision and Recall ）以及F1指标

与回归问题计算损失函数不同，二分类特有的一种评价指标为查准率和查全率（Precision and Recall）以及F1指标。

Precision就是预测为正类的样本有多少比例的样本是真的正类，TP/(TP+FP)；Recall就是所有真正的正类样本有多少比例被预测为正类，TP/（TP+FN）。其中TP为真正类被预测为正类，FP为负类被预测为正类，FN为真正类被预测为负类。Scikit-learn也有对应的函数

```
from sklearn.metrics import precision_score, recall_score
precision_score(y_train_5, y_train_pred)
recall_score(y_train_5, y_train_pred)
```

由于Precision和Recall有两个数，如果一大一下的话不好比较两个模型的好坏，F1指标就是结合两者，求调和平均

$$\mathrm{F1} = 2 * \frac{\mathrm{Precision} * \mathrm{Recall}}{\mathrm{Precision} + \mathrm{Recall}}$$

```
from sklearn.metrics import f1_score
f1_score(y_train_5, y_train_pred)
```

对于F1值，暗含了Precision和Recall是同等重要的，然而对于现实问题并不一定同等重要，比如推荐电影，给观众推荐一部很好的电影比很多电影更重要，因此Precision更加重要；而对于检查安全问题，宁可多次去核查也不能出现一点错误，因此Recall更重要。所以对于实际问题，应该适当权衡Precision和Recall。

Precision/Recall 的权衡

虽然我们没有办法改变Scikit-learn里面的predict()函数来改变分类输出，但我们能够通过decision_function()方法来得到输出的得分情况，得分越高意味着越有把握分为这一类。因此可以通过对得分设一个界（threshold），得分大于threshod的分为正类，否则为负类，以此来调整Precision和Recall。

然而这个threhold应该怎么确定，scikit-learn中提供相关的函数precision_recall_curve()来帮助我们确定。

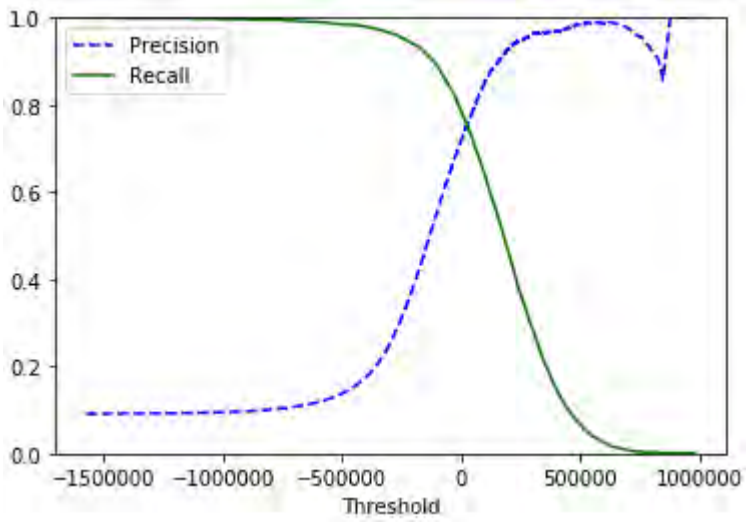
需要注意：cross_val_predict计算交叉验证后验证集部分的得分(而不是类别)。

```
#计算得分
from sklearn.model_selection import cross_val_predict
y_scores = cross_val_predict(sgd_clf, X_train, y_train_5, cv=3,
method="decision_function")
#调用precision_recall_curve
from sklearn.metrics import precision_recall_curve
precisions, recalls, thresholds = precision_recall_curve(y_train_5, y_scores)
```

需要注意：得出的结果precisions和recalls的元素个数比threholds多一个，因此画图时需要回退一个。

将结果通过图展示出来

```
import matplotlib.pyplot as plt
def plot_precision_recall_vs_threshold(precisions, recalls, thresholds):
    plt.plot(thresholds, precisions[:-1], "b--", label="Precision")
    plt.plot(thresholds, recalls[:-1], "g-", label="Recall")
    plt.xlabel("Threshold")
    plt.legend(loc="upper left")
    plt.ylim([0, 1])
plot_precision_recall_vs_threshold(precisions, recalls, thresholds)
plt.show()
```



观察图，按需要选择合适的Threshold。

3、多分类

对于随机森林分类器(Random Forest classifiers)和朴素贝叶斯分类器(naive Bayes classifiers)，本来就能够处理多分类的问题。但是有一些算法如支撑向量机（Support Vector Machine classifier）和线性分类器(Linear classifiers)就是二分类器。将二分类器扩展到多分类器一般有两种做法。

1、OVA(one-versus-all)：比如分类数字（0-9），则训练10个分类器（是否为0的分类器，是否为1的分类器.，…，是否为9的分类器），每一个分类器最后会算出一个得分，判定为最高分的那一类

2、OVO(one-versus-one)：每个类之间训练一个分类器（比如0和1训练一个分类器，1-3训练一个分类器），这样总共有N*(N-1)/2个分类器，哪个类得分最高判定为那一类。

一般情况下，OVO训练速度比较快（因为训练多个小分类器比训练一个大分类器时间要快），而OVA的表现会更好，因此Scikit-learn中二分类器进行多分类默认为OVA，除了支撑向量机使用OVO以外（由于支撑向量机对大规模的数据表现不好）

下面是一个二分类器SGD分类器扩展为多分类器用作数字分类的例子

```
from sklearn.linear_model import SGDClassifier
sgd_clf = SGDClassifier(random_state=42)
sgd_clf.fit(X_train, y_train)
```

可以随便取一个数据看看得分情况

```
some_digit = X[36000]
sgd_clf.predict([some_digit])
some_digit_scores = sgd_clf.decision_function([some_digit])
```

```
Out[20]:
array([[-210210.64242799, -466605.94685249, -616267.64814787,
         -95872.88243383, -417860.94008745,  156031.25711853,
        -701276.72055114, -393255.50541873, -768114.61125831,
        -578388.98887504]])
```

可以看到第6个分数（代表数字5）最高，即分类为数字5。

这是OVA的情况，也可以强制变为OVO（OVA）的情况OneVsOneClassifier（OneVsRestClassifier）

4、评价分类器的好坏

对于回归任务，上一节评价模型的好坏用的是交叉验证法，对于分类任务，同样也可以采取交叉验证法，不同的是，误差不是均方误差，而是准确率（或者交叉熵）。和之前交叉验证的函数相同为cross_val_score，不过scoring为accuracy。

```
from sklearn.model_selection import cross_val_score
cross_val_score(sgd_clf, X_train, y_train, cv=3, scoring="accuracy")
```

结果为：

```
Out[25]: array([ 0.86362727,  0.85509275,  0.86032905])
```

可以看到准确率大概在86%，当然也可以采用预处理（标准化）来增加准确率。

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train.astype(np.float64))
cross_val_score(sgd_clf, X_train_scaled, y_train, cv=3, scoring="accuracy")
```

结果为：

```
Out[27]: array([ 0.90626875,  0.90984549,  0.91218683])
```

可以看到经过标准化后准确率上升到了90%

5、错误分析

当然，如果这是一个真正的项目，还需要尝试多个模型，选择最佳模型并微调超参数，现在假设已经找到了这个模型，并想进一步提升，其中一种方法是分析错误的类型是哪些。

混淆矩阵法

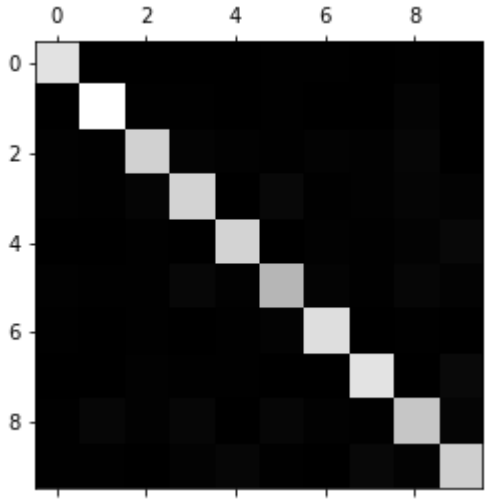
首先要使用cross_val_predict计算交叉验证后验证集部分的预测分类结果（而不是正确率），然后根据真正的标签和预测结果对比，画出混淆矩阵（如下图），第 i 行第 j 列的数字代表数字 i 被预测为数字 j 的个数总和，比如第5行第一个数表示数字4被误判为0的次数为75；对角线上即为正确分类的次数。

```
from sklearn.model_selection import cross_val_predict
from sklearn.metrics import confusion_matrix
y_train_pred = cross_val_predict(sgd_clf, X_train_scaled, y_train, cv=3)
conf_mx = confusion_matrix(y_train, y_train_pred)
conf_mx
```

```
array([[5735,    3,   18,    9,    8,   50,   48,    9,   40,    3],
       [   1, 6484,   43,   30,    6,   41,    9,    9,  107,   12],
       [  54,   40, 5318,  110,   71,   30,   97,   53,  170,   15],
       [  52,   43,  132, 5350,    2,  227,   34,   57,  145,   89],
       [  22,   26,   37,   10, 5358,   12,   54,   33,   77,  213],
       [  75,   45,   32,  183,   71, 4618,  108,   27,  171,   91],
       [  31,   22,   47,    1,   40,   95, 5628,    6,   47,    1],
       [  25,   20,   75,   31,   50,   10,    4, 5793,   18,  239],
       [  50,  160,   75,  155,   15,  161,   53,   31, 5022,  129],
       [  39,   36,   26,   89,  164,   38,    2,  213,   82, 5260]], dtype=int64)
```

然而矩阵中数据太多很复杂，所以可以通过图更加直观地表示出来

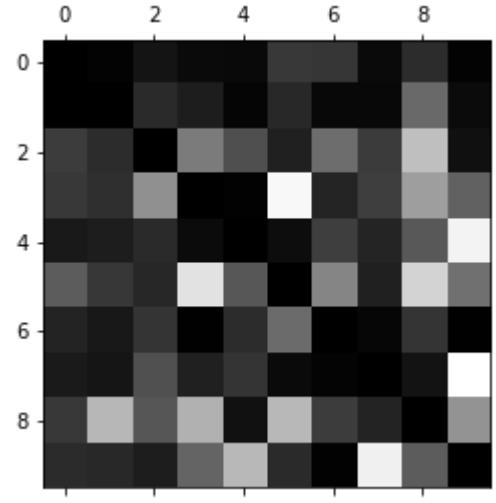
```
plt.matshow(conf_mx, cmap=plt.cm.gray)
plt.show()
```



从图中可以看到分类结果表现的还好，数字5比较暗，说明数字5被错分的比较多。

还可以去除正确的对角线的分布情况，只看错误的分布

```
row_sums = conf_mx.sum(axis=1, keepdims=True)
norm_conf_mx = conf_mx / row_sums.astype(np.float64)
np.fill_diagonal(norm_conf_mx, 0)
plt.matshow(norm_conf_mx, cmap=plt.cm.gray)
plt.show()
```



可以看到第6行第4列与第4行第6列种比较白，说明5错分为3，3错分为5的次数比较多。因此还可以把错误分为3和5的样本提取出来观察分析，从而改进算法。

6、多标签分类（Multilabel Classification ）

直到现在的例子都是将数据分为某一类，但有些时候想分为多个类（比如人脸识别分类器，如果一张图片上有多个人脸，那就不能只识别一个人，而是要识别出多个人），输出比如为[1，0，1]，则分为1和3类。对于这种输出多个二值分类标签的就是多标签分类。下面是一个简单的例子：

```
from sklearn.neighbors import KNeighborsClassifier
y_train_large = (y_train >= 7)
y_train_odd = (y_train % 2 == 1)
y_multilabel = np.c_[y_train_large, y_train_odd]
knn_clf = KNeighborsClassifier()
knn_clf.fit(X_train, y_multilabel)
knn_clf.predict([some_digit])
```

```
Out[58]: array([[False,  True]], dtype=bool)
```

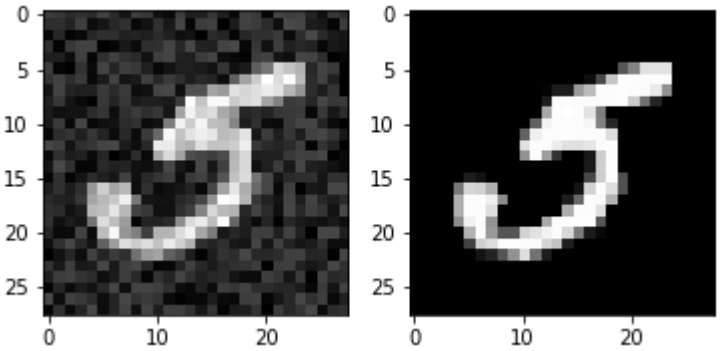
例子中的任务是：分类数据 是否大于等于7 以及 是否为奇数 这2个标签，使用k-近邻分类器（KNN）进行多标签分类（不是所有的分类器都能进行多标签分类）。

评价多标签分类模型的方法可以对每种标签求F1值，再求平均值。

7、多输出分类（Multioutput Classification ）

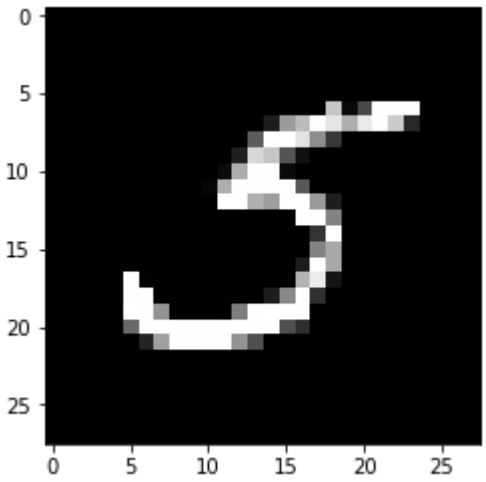
多输出分类是多标签分类更一般的一种形式，比如现在图像有噪声，需要将每个像素分类为0或1已达到去噪的目的（也可以多个标签）。下面就是一个例子，如下图，左图为的像素为特征，右图为标签。

```
#生成左边的噪声图
import numpy.random as rnd
noise1 = rnd.randint(0, 100, (len(X_train), 784))
noise2 = rnd.randint(0, 100, (len(X_test), 784))
X_train_mod = X_train + noise1
X_test_mod = X_test + noise2
y_train_mod = X_train
y_test_mod = X_test
import matplotlib.pyplot as plt
plt.subplot(1,2,1)
plt.imshow(X_train_mod[36000].reshape(28,28),cmap=plt.cm.gray)
plt.subplot(1,2,2)
plt.imshow(X_train[36000].reshape(28,28),cmap=plt.cm.gray)
```



接下来训练一个KNN模型实现多输出分类（去噪）

```
from sklearn.neighbors import KNeighborsClassifier
knn_clf = KNeighborsClassifier()
knn_clf.fit(X_train_mod, y_train_mod)
clean_digit = knn_clf.predict([X_train_mod[36000]])
plt.imshow(clean_digit.reshape(28,28),cmap=plt.cm.gray)
```



可以看到训练的模型能够对每个像素进行分类，从而实现去噪。

四、训练模型

1、线性模型

线性模型形如：

$$y = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots \theta_n x_n$$

为了得到使均方误差最小的θ，一般采用两种方法：

Normal Equation

使用数学方法推导能直接求得参数θ的方法为Normal Equation。而对于线性模型，数学方法为：最小二乘法。

通过最小二乘法能直接得到线性模型：

$$\hat{\theta} = (\mathrm{X}^T X)^{-1} X^T y$$

对应的Scikit-learn中的函数为LinearRegression，下面是一个例子y=4*x+3

```
import numpy as np
X = 2 * np.random.rand(100, 1)
y = 4 + 3 * X + np.random.randn(100, 1)

from sklearn.linear_model import LinearRegression
lin_reg = LinearRegression()
lin_reg.fit(X, y)
lin_reg.intercept_, lin_reg.coef_
```

```
Out[1]: (array([ 4.08580049]), array([[ 2.94946396]]))
```

可以看到得到的结果接近4和3，没有达到准确值是由于加入了随机噪声的原因。

虽然能够精确的得到结果，但是这个算法的复杂度很高，计算$\mathrm{X}^T X$ 的逆的算法复杂度就为$O(\mathrm{n}^{2.4}) - O(\mathrm{n}^3)$，如果训练样本非常大，则运行时间会很长。而且这只是只有一种特征的情况，则运行时间会更长。

因此这种方法适合数据量较小的情况，能够找到精确解。

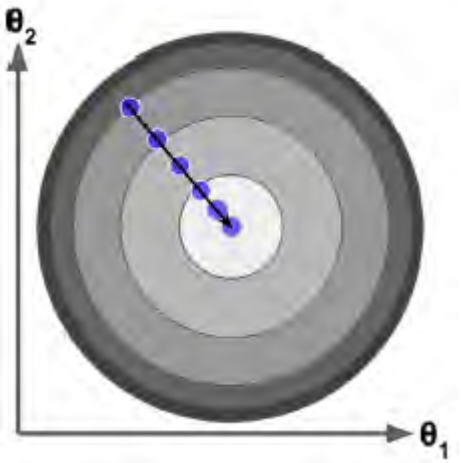
梯度下降法（Gradient Descent ）

为了处理训练样本数比较大数据，通常采用梯度下降法，得到一个近似解，但是速度较快。梯度下降法就是通过设置学习率eta与迭代次数，每一代计算目标函数的梯度来更新权值的方法。（具体不做介绍）

学习率太小会导致收敛太慢，学习率太大会导致偏离最优解，因此可以通过之前讲的grid search来调整参数。对于迭代次数可以一开始设置一个比较大的数，当梯度开始变得很小时，则可以停下来，记录迭代次数。

Batch Gradient Descent

Batch Gradient Descent即每次迭代使用所有样本计算梯度，取平均，来更新参数。可以看到这个方法能直接朝着最优解方向前进。

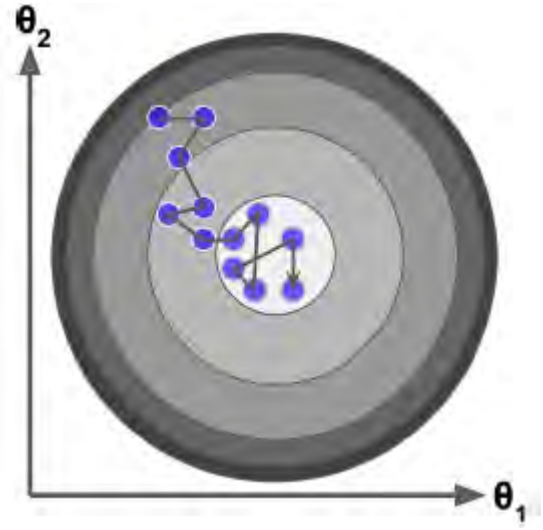



```
eta = 0.1 # learning rate
n_iterations = 1000
m = 100
theta = np.random.randn(2,1) # random initialization
X_b = np.c_[np.ones((100, 1)), X]
for iteration in range(n_iterations):
    gradients = 2/float(m) * X_b.T.dot(X_b.dot(theta) - y)
    theta = theta - eta * gradients
```

```
array([[ 4.1829176 ],
       [ 2.90391237]])
```

Stochastic Gradient Descent

由于Batch Gradient Descent每次迭代使用全部训练集，当训练集数据量较大时，训练速度则会很慢。因此与这个相反，随机梯度下降（Stochastic Gradient Descent）就是每一代随机选择一个样本，计算梯度，更新参数。由于每次只抽取一个样本，因此在大量数据也能很好的计算。



但是当接近最优解时（如图）始终无法收敛到最优解，这是由于每次根据一个样本更新参数，每个样本都有自己的梯度方向，所以会不断的在最优解附近振荡。为了解决这个问题，引入了learning_schedule参数，这个一个学习率衰减参数，在迭代的过程中，不断减少学习率，在迭代结束时学习率非常小，可以看作不再振荡，接近最优解。

随机梯度下降skicit-learn也有对应的函数：其中n_iter为迭代次数，eta0为学习率，penalty下面会说到。

```
from sklearn.linear_model import SGDRegressor
sgd_reg = SGDRegressor(n_iter=50, penalty=None, eta0=0.1)
sgd_reg.fit(X, y.ravel())
sgd_reg.intercept_, sgd_reg.coef_
```

```
(array([ 4.21280158]), array([ 2.94808826]))
```

Mini-batch Gradient Descent

结合Batch Gradient Descent和Stochastic Gradient Descent，每一代计算Mini-batch（远小于Batch）个样本的梯度，计算平均值，更新参数。这种方式的好处是能在数据量比较大的训练样本中进行矩阵运算，特别是可以在GPU上计算。（在深度学习中运用的比较多）

2、多项式模型

如果数据不是简单的一条直线，也可以通过线性模型来训练，其中一种比较简单的方法是通过给训练数据特征开n次方。下面是一个例子：

```
m = 100
X = 6 * np.random.rand(m, 1) - 3
y = 0.5 * X**2 + X + 2 + np.random.randn(m, 1)

from sklearn.preprocessing import PolynomialFeatures
poly_features = PolynomialFeatures(degree=2, include_bias=False)
X_poly = poly_features.fit_transform(X)
X[0],X_poly[0]
```

```
(array([ 0.13625783]), array([ 0.13625783,  0.0185662 ]))
```

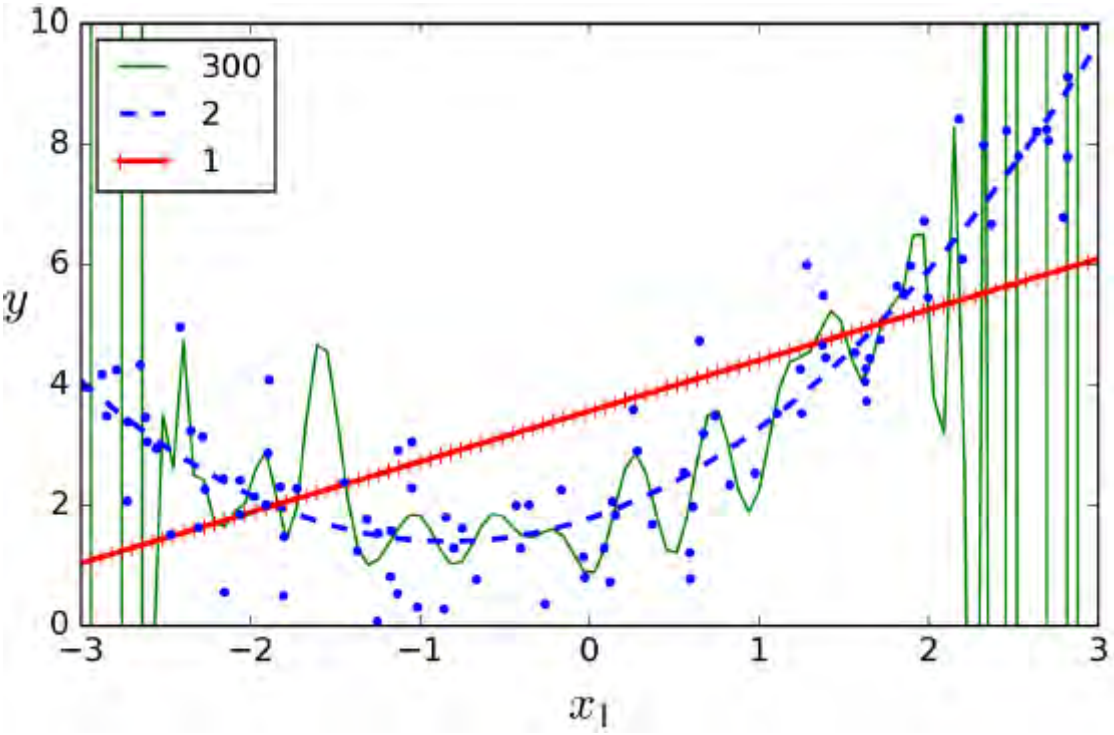
可以看到，通过PolynomialFeatures以后，得到新的特征(x, x^2)，再用这个新特征训练一个线性模型。

```
lin_reg = LinearRegression()
lin_reg.fit(X_poly, y)
```

注：若存在两个特征a和b，若degree=2时，不仅增加了a^2和b^2，还有ab

Learning Curves

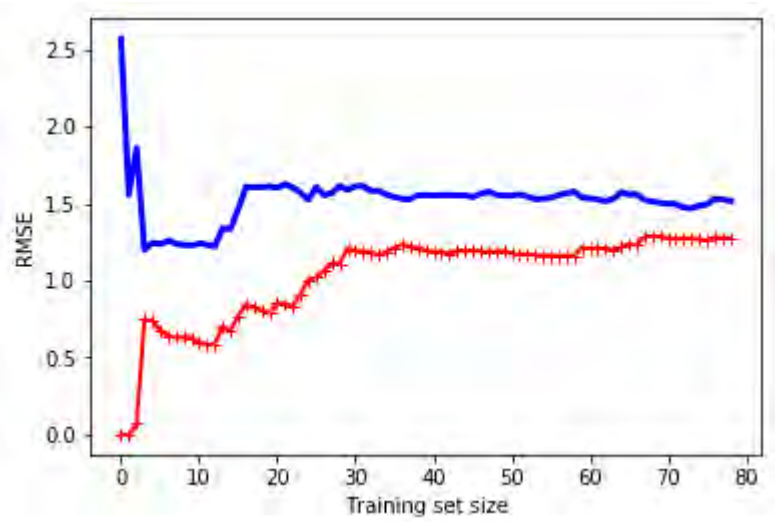
如果使用高degree创造非常多特征来作多项式模型，你能够很好的拟合训练数据。然而这种只对训练数据拟合的情况称为过拟合（如图），而对于线性模型则为欠拟合。



为了检测模型是否有效，在第3节已经讲述了其中一种方法，交叉验证法。下面再介绍一种方法Learning Curves。通过分开训练和验证集，不断调整训练集的数量，计算均方误差来画出曲线，观察训练集和验证集在训练中的情况。

```
import matplotlib.pyplot as plt
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split
def plot_learning_curves(model, X, y):
    X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2)
    train_errors, val_errors = [], []
    for m in range(1, len(X_train)):
        model.fit(X_train[:m], y_train[:m])
        y_train_predict = model.predict(X_train[:m])
        y_val_predict = model.predict(X_val)
        train_errors.append(mean_squared_error(y_train_predict, y_train[:m]))
        val_errors.append(mean_squared_error(y_val_predict, y_val))
    plt.plot(np.sqrt(train_errors), "r-+", linewidth=2, label="train")
    plt.plot(np.sqrt(val_errors), "b-", linewidth=3, label="val")
    plt.xlabel("Training set size")
    plt.ylabel("RMSE")
```

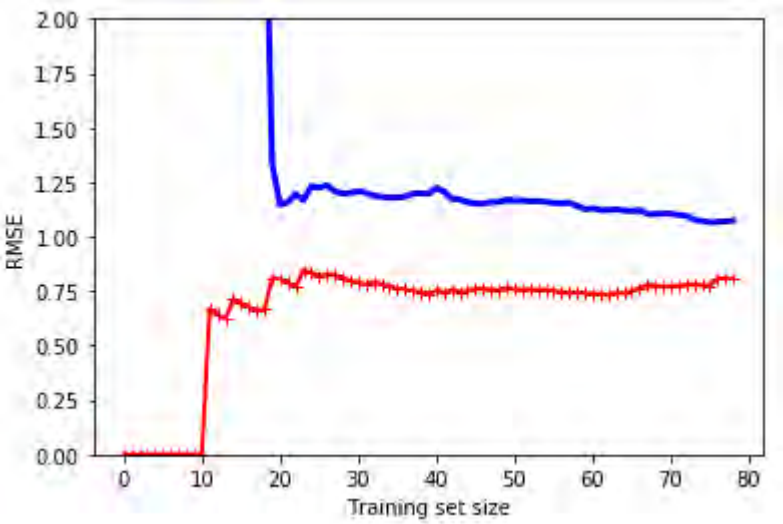
```
lin_reg = LinearRegression()
plot_learning_curves(lin_reg, X, y)
```



可以看到当训练集中只有一个或两个实例时，模型可以完美地拟合它们，这就是曲线从零开始的原因。但是，当新样本被添加到训练集时，模型不可能很好地拟合训练数据，这是由于样本是复杂的。再看看验证数据上模型的性能。当模型在很少的训练样本时，就不能很好的泛化推广，这就是为什么验证错误最初是相当大的。当新样本被添加到训练集时，通过学习，从而验证错误缓慢下降。误差最终会停留在一个类似高原的地方。

上面为degree=1的模型，再看看degree=10的模型

```
from sklearn.pipeline import Pipeline
polynomial_regression = Pipeline((
("poly_features", PolynomialFeatures(degree=10, include_bias=False)),
("sgd_reg", LinearRegression()),
))
plot_learning_curves(polynomial_regression, X, y)
plt.ylim(0,2)
```



可以看到degree=10的训练误差和验证误差都比degree=1的要小。所以该模型较好。我们还可以看到两条曲线中间的间隙，间隙越大，说明过拟合程度越大。

3、正则化线性模型（Regularized Linear Models ）

为了防止过拟合，可以给模型损失函数（如均方误差）增加正则项。

Ridge Regression （L2正则化）

Ridge Regression的损失函数为：

$$J(\theta) = MSE(\theta) + \alpha\frac{1}{2}\sum_{i=1}^{n}{\theta_i}^2$$

可以看到加入后面的正则项后会使θ靠近0，直观上可以理解为对$MSE(\theta)$（即模型）贡献越小的θ，惩罚越大，这样的θ越小，所以通过正则化能够减小这些无关特征带来的影响。

需要注意的是：在Ridge Regression之前要先对特征进行缩放（标准化或最大最小缩放），这是因为Ridge Regression对特征的尺度大小很敏感。

同样有两种方法计算最优解Normal Equation和梯度下降法。

对于Normal Equation结果为：

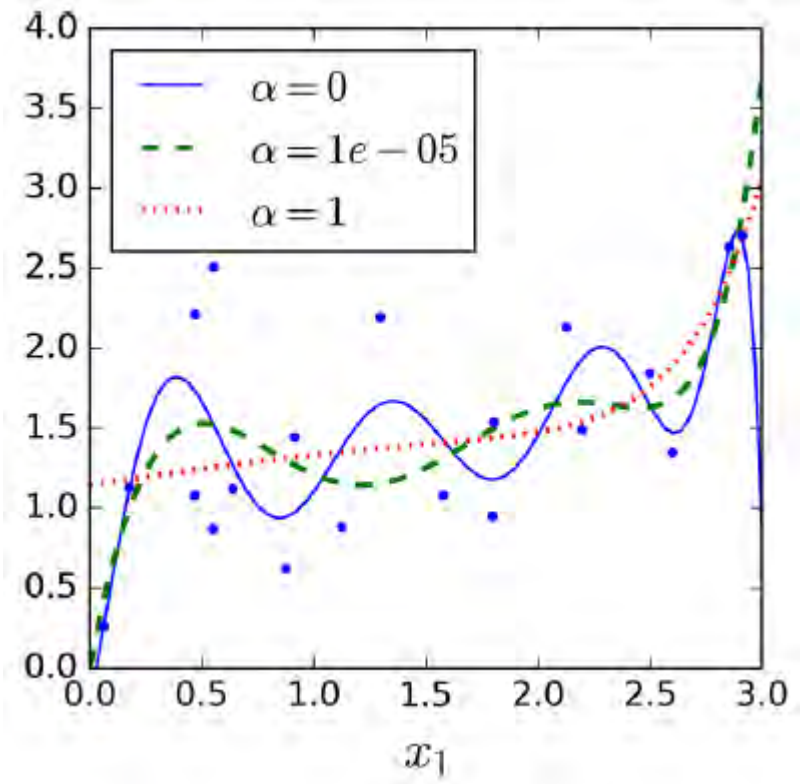
$$\hat{\theta} = (\mathrm{X}^T\mathrm{X} + \alpha\mathrm{A})^{-1}X^Ty$$

对应的代码为

```
from sklearn.linear_model import Ridge
ridge_reg = Ridge(alpha=1, solver="cholesky")
ridge_reg.fit(X, y)
```

随机梯度下降L2正则化对应的代码为

```
from sklearn.linear_model import SGDRegressor
sgd_reg = SGDRegressor(penalty="l2")
sgd_reg.fit(X, y.ravel())
```



通过设置不同的α值，可以看到通过正则化后曲线变得平缓，看起来比没有正则化的曲线有着更好的泛化推广能力。

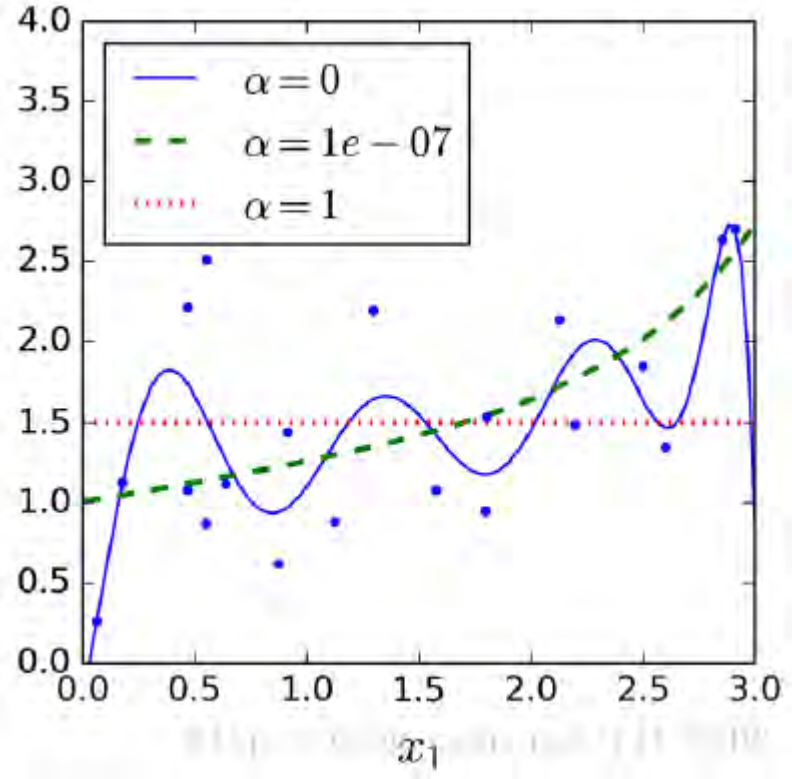
Lasso Regression　（L1正则化）

Lasso Regression的损失函数为：

$$J(\theta) = MSE(\theta) + \alpha \sum_{i=1}^{n} |\theta_i|$$

与L2正则化不同的是，正则项从二次方变为了一次放的绝对值，这就带来的一个特性，不同于L2正则化使得θ在原点附近（即大部分θ都靠近0），L1正则化使得θ更趋向于在坐标轴上（即大部分的θ等于零，少部分靠近零），相当于惩罚变得更大。

```
#最小二乘LASSO方法
from sklearn.linear_model import Lasso
lasso_reg = Lasso(alpha=0.1)
lasso_reg.fit(X, y)

#随机梯度下降正则化L2（线性）
from sklearn.linear_model import SGDRegressor
sgd_reg = SGDRegressor(penalty="l1")
sgd_reg.fit(X, y.ravel())
```



可以看到经过L1正则化后曲线变得更加平缓，更像一条直线。

需要注意的是：L1正则化后会导致在最优点附近震荡，因此要像随机梯度下降一样减小学习率。

Elastic Net

Elastic Net的损失函数为：

$$J(\theta) = MSE(\theta) + r\alpha \sum_{i=1}^{n} |\theta_i| + \frac{1-r}{2}\alpha \sum_{i=1}^{n} {\theta_i}^2$$

可以看到Elastic Net是L1正则化和L2正则化的结合，通过一个参数调整比例。

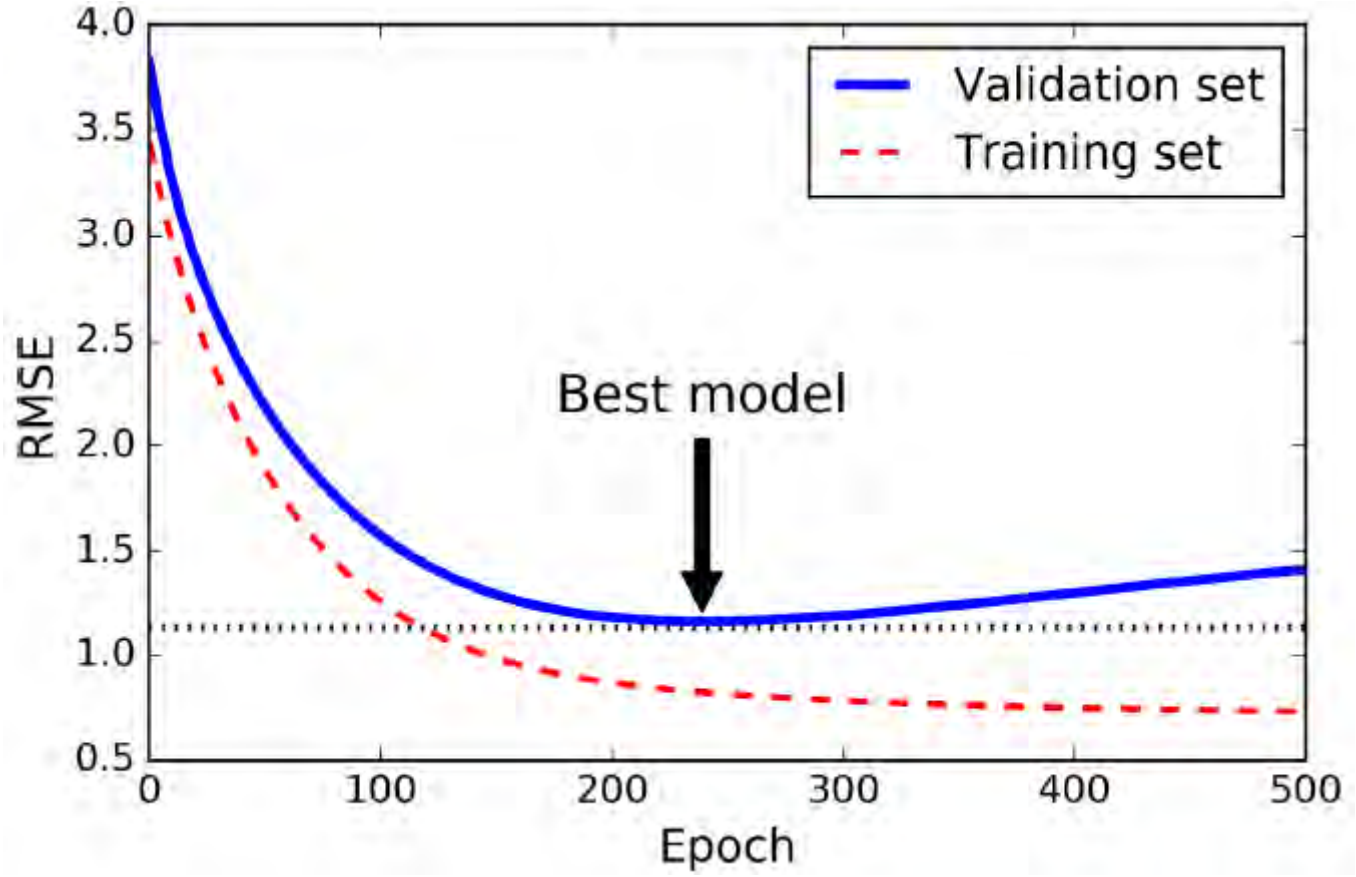
```
#最小二乘ElasticNet方法
from sklearn.linear_model import ElasticNet
elastic_net = ElasticNet(alpha=0.1, l1_ratio=0.5)
elastic_net.fit(X, y)

#随机梯度下降ElasticNet（线性）
from sklearn.linear_model import SGDRegressor
sgd_reg = SGDRegressor(penalty="elasticnet",l1_ratio=0.5)
sgd_reg.fit(X, y.ravel())
```

如何选择正则化

加入少量的正则化一般都会给模型带来一定的提升。一般情况下都会选择Ridge（L2正则化）；但是如果得知数据中只有少量的特征是有用的，那么推荐使用Lasso（L1正则化）或Elastic Net；一般来说Elastic Net会比Lasso效果要好，因为当遇到强相关性特征说特征数量大于训练样本时Lasso会表现的很奇怪。

提前中断训练（Early Stopping ）

通过调整迭代次数，在验证误差达到最小时立即停止训练。图中是用Batch Gradient Descent训练的，随着迭代次数的上升，验证集上的预测误差会下降。 但是，过了一段时间验证错误停止下降，反而往上开始回升。这表明该模型已经开始过度拟合训练数据。 一旦验证错误达到最小，立即停止训练。 这是一种简单而有效的正则化技术。



需要注意的是：对于Stochastic 和 Mini-batch Gradient Descent的曲线不会这么平滑。其中一种方案是当到达最低点时，再过一段时间（当你觉得不会再下降），停止，将参数调回到当时的最低点。

下面是一个例子，当warm_start=True时，调用fit()后继续训练模型而不是重新训练模型。

```
from sklearn.base import clone
sgd_reg = SGDRegressor(n_iter=1, warm_start=True, penalty=None,
learning_rate="constant", eta0=0.0005)
minimum_val_error = float("inf") #正无穷
best_epoch = None
best_model = None
for epoch in range(1000):
    sgd_reg.fit(X_train_poly_scaled, y_train) # 继续训练
    y_val_predict = sgd_reg.predict(X_val_poly_scaled)
    val_error = mean_squared_error(y_val_predict, y_val)
    if val_error < minimum_val_error:
        minimum_val_error = val_error
        best_epoch = epoch
        best_model = clone(sgd_reg) #保存模型
```

4、Logistic 回归（Logistic Regression ）

Logistic回归与线性回归模型比较相似，Logistic回归在线性回归模型的基础上增加了sigmoid函数$\sigma()$，即

$$h_\theta(x) = \sigma(\theta^T x)$$

其中

$$\sigma(t) = \frac{1}{1 + e^{-t}}$$

损失函数采用对数似然损失函数：

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}[y_i \log(h_\theta(x_i)) + (1 - y_i)\log(1 - h_\theta(x_i)]$$

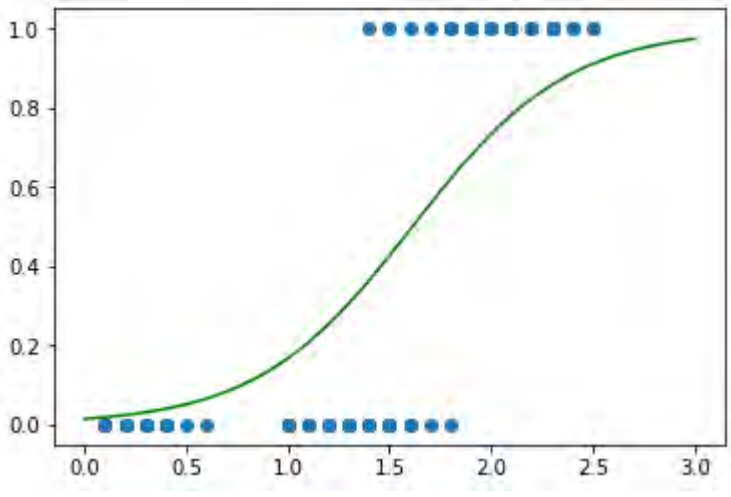
训练与线性回归模型相似，可以计算梯度，用梯度下降法更新参数。

下面利用Logistic回归对真实数据Iris进行分类。Iris数据为150个训练样本，包含4个特征，分为3类。

首先先用Logistic模型作一个二分类器。取出其中一种特征，判断是否为某一类。

```
#Iris
import numpy as np
from sklearn import datasets
iris = datasets.load_iris()
X = iris["data"][:, 3:] # 只读取最后一个特征
y = (iris["target"] == 2).astype(np.int) # 取出判断是否为第3类的label

#训练Logistic回归模型
from sklearn.linear_model import LogisticRegression
log_reg = LogisticRegression()
log_reg.fit(X, y)
```

对于一个二分类问题，还可以画出判别线



注：对于Logistic回归同样需要L1，L2正则化，而这个scikit-learn也默认设置了。

4、Softmax 回归（Softmax Regression ）

对于Logistic回归是一个二分类器，然而不需要像第4节所讲到了训练多个二分类器来实现多分类。Logistic回归可以直接扩展成一个多分类器Softmax回归。

与Logistic回归相似，Softmax回归计算每一类的一个概率，归为概率最大的一类。

Softmax函数为：

$$\hat{p}_k = \sigma(s(x))_k = \frac{\exp(s_k(x))}{\sum_{j=1}^{K} \exp(s_j(x))}$$

其中：

$$s_k(x) = {\theta_k}^T x$$

其中K为类别数，需要注意的是θ_k说明每个类别对应有自己的θ，所有θ_k组合起来是全部的参数。

对于Softmax回归使用交叉熵（cross entropy）函数作为损失函数：

$$J(\theta) = -\frac{1}{m} \sum_{i=1}^{m} \sum_{k=1}^{K} {y_k}^{(i)} \log({\hat{p}_k}^{(i)})$$

训练与线性回归模型相似，可以计算每一类的偏导数，用梯度下降法更新每一类的参数。

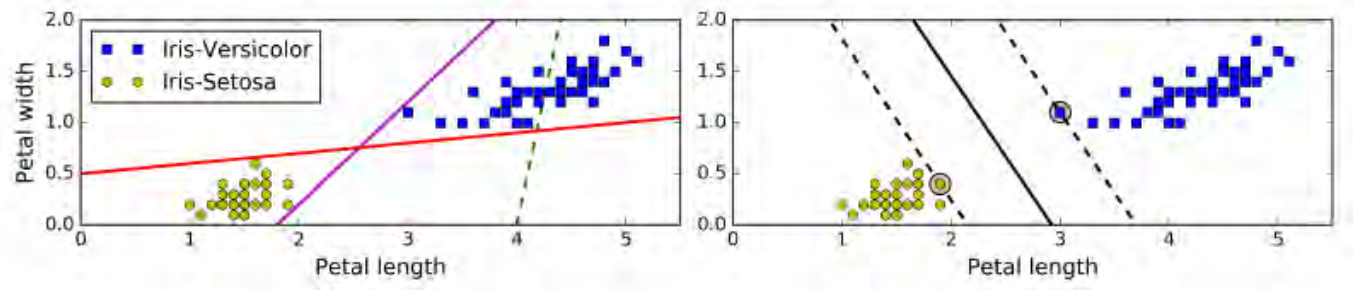
下面是一个实例，LogisticRegression默认使用（OVA），如果使用Softmax就把multi_class=”multinomial”

```
X = iris["data"][:, (2, 3)] # petal length, petal width
y = iris["target"]
softmax_reg = LogisticRegression(multi_class="multinomial",solver="lbfgs", C=10)
softmax_reg.fit(X, y)
```

五、支持向量机

支持向量机（SVM）是一种非常强大的机器学习模型，能够进行线性、非线性分类、回归问题，还能检测异常值。SVM特别适用于复杂但小型或中型的数据集的分类。

1、线性SVM分类（Linear SVM Classification ）



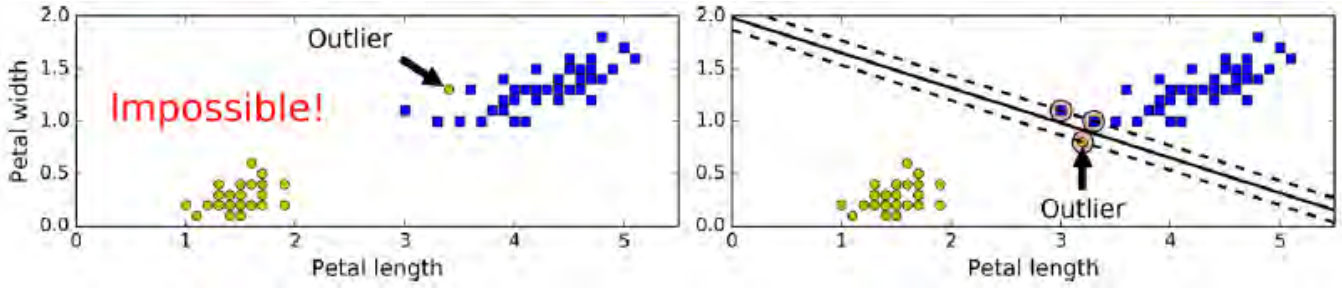
如图，从左图我们可以看到现在有两类数据，只需要1条直线就能把它们分开，其中红线和紫线都能把这两类数据完美的分开，但是两条直线都非常靠近样本，如果有新的样本加入，有比较大的可能会分类错误。

因此线性SVM的思想就是如右图的黑色实线，决策边界在正确分类的同时离最近的样本尽可能的远。而这些最近的样本（途中虚线上的点）即为支持向量(support vector)。因此只要没有点在这些点划分的区域之间，决策边界就只由这些支持向量所决定。

需要注意：SVM对特征之间的尺度比较敏感，因此要先对特征进行缩放（如标准化（StandardScaler））

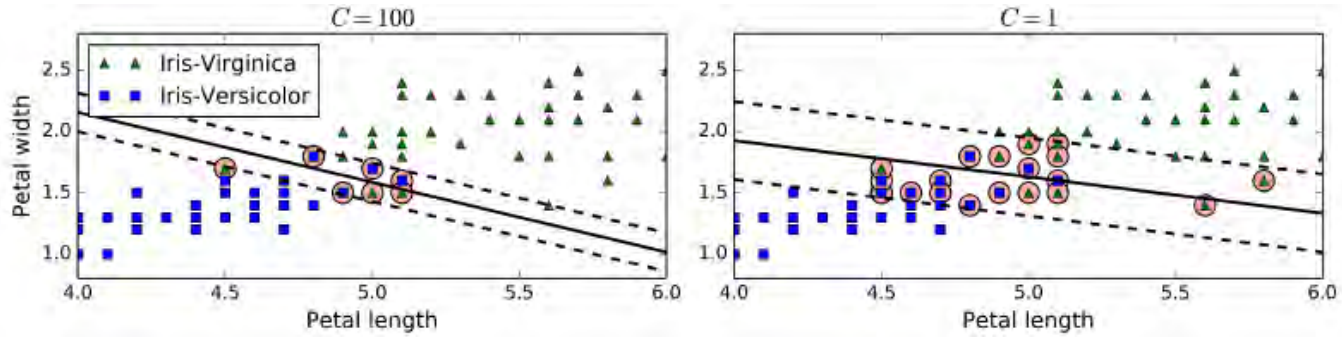
2、Soft Margin Classification

上面严格的把所有样本用一条直线分开为两类，叫做Hard margin classification。但是在实际情况中，可能存在一些异常点（如下图），左图中的异常点在另一类之中，没有办法用一条直线分开；右图中异常点离另一类很近，虽然能分开，但是决策线看起来非常不好。



为了解决这种问题，需要一个更加宽松灵活的模型，来权衡尽可能分开以及限制支持向量组成的区域里面点的个数，这就叫Soft Margin Classification

在Scikit-learn的SVM类中，可以通过调剂松弛因子C来权衡，C越大，分类越严格；C越小，在margin内的点越多。



需要注意：如果你的SVM模型过拟合了，可以尝试减小C

下面通过加载Iris数据，标准化后训练一个线性SVM模型（C=0.1），使用的损失函数为hinge loss：

$$\max(0, 1 - t)$$

```
import numpy as np
from sklearn import datasets
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.svm import LinearSVC
iris = datasets.load_iris()
X = iris["data"][:, (2, 3)] # petal length, petal width
y = (iris["target"] == 2).astype(np.float64) # Iris-Virginica
svm_clf = Pipeline((
("scaler", StandardScaler()),
("linear_svc", LinearSVC(C=1, loss="hinge")),
))
svm_clf.fit(X, y)

#预测
svm_clf.predict([[5.5, 1.7]])
```

需要注意：SVM分类器不像Logistic回归分类器一样有predict_proba()方法来计算得分（概率），因为SVM只是靠支持向量来构建决策线。而且loss函数一定要记得填”hinge”，因为默认不是”hinge”。

除了使用LinearSVC类来训练线性SVM外，还可以使用SVC类，参数为（kernel=“linear”，C=1）,不过这种方法会比LinearSVC慢很多。还可以用SGDClassifier类，参数为（loss=“hinge”，alpha=1/（m*C））， 这种使用随机梯度下降方法来训练SVM，虽然没有LinearSVC收敛快，但是能够处理数据量庞大的训练集，而且能够在线学习。

3、非线性SVM分类（Nonlinear SVM Classification）

由于现实数据中更多的是线性不可分的数据。为了能用线性分类器分类线性不可分的数据，其中一种方法就是如上一节的多项式分类想法一样，通过增加x^2, x^3等特征，使得数据能够线性可分。

多项式核（Polynomial Kernel ）

基于这个思想，可以增加多项式特征来进行非线性分类。但是如果degree设置的比较小，则比较难分类比较复杂的数据；如果degree设置的比较大，产生了大量模型，导致训练的非常慢。

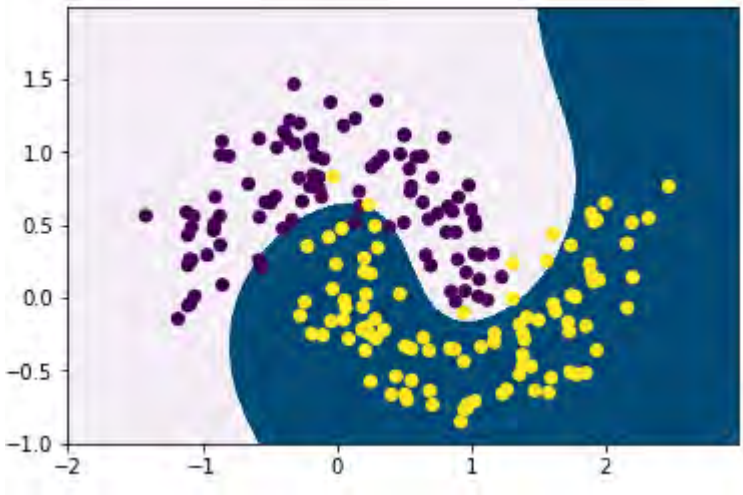
幸运的是，SVM能用运用一些数学的技巧，称为核技巧（Kernel trick）（后面会说到）。可以不真正的增加这些（如多项式）特征，来达到增加这些特征相同的效果。所以高degree并不会带来特征的急剧增加，因为并没有真正增加特征。这种方法可以在SVC类中使用。下面是一个例子：

```
from sklearn.datasets import make_moons
from sklearn.pipeline import Pipeline
from sklearn.svm import SVC
#构造球型数据集
(X,y)=make_moons(200,noise=0.2)
#使用SVC类中的多项式核训练
poly_kernel_svm_clf = Pipeline((
("scaler", StandardScaler()),
("svm_clf", SVC(kernel="poly", degree=3, coef0=1, C=5))
))
poly_kernel_svm_clf.fit(X, y)
```

其中参数coef0为高degree特征相比低degree特征对模型的影响程度。参数degree为选择多项式特征的维度，参数C为松弛因子。

还可以把决策线画出来看看效果如何：

```
import numpy as np
import matplotlib.pyplot as plt
xx, yy = np.meshgrid(np.arange(-2,3,0.01), np.arange(-1,2,0.01))
y_new=polynomial_svm_clf.predict(np.c_[xx.ravel(),yy.ravel()])
plt.contourf(xx, yy, y_new.reshape(xx.shape),cmap="PuBu")
plt.scatter(X[:,0],X[:,1],marker="o",c=y)
```



可以看到能够比较好的完成非线性分类。

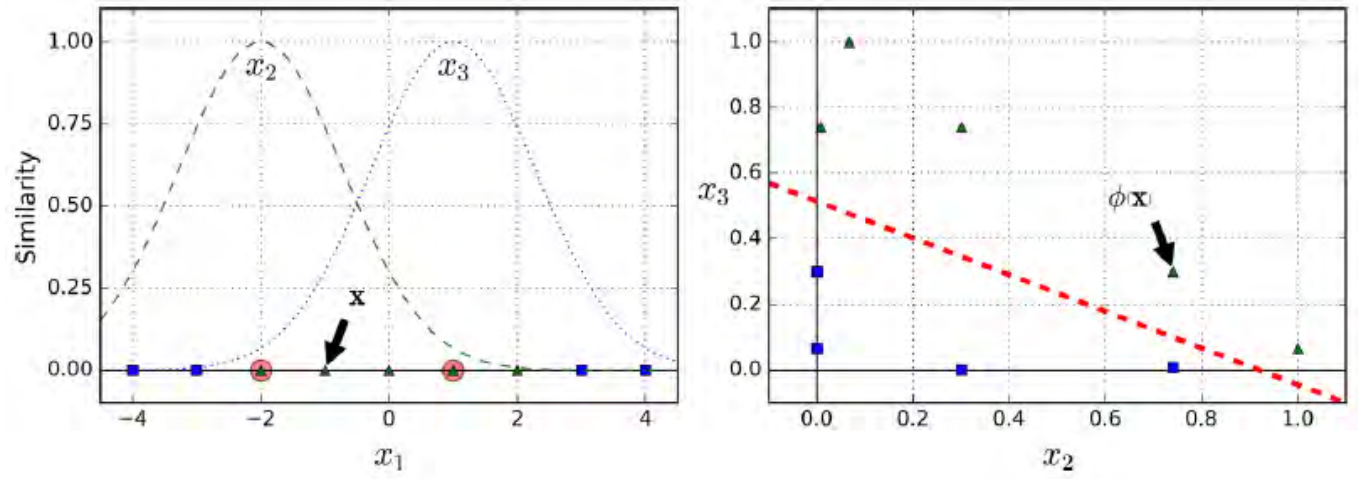
需要注意：模型的参数应该使用Grid search来调整到一个比较好的状态，如果过拟合（可以通过交叉验证来评价），则可以适当减小degree；如果欠拟合，则增加。

高斯径向基核（Gaussian RBF Kernel ）

除了通过多项式增加特征以外，还有别的增加特征的方式，比如通过Gaussian Radial Basis Function (RBF)增加相似特征。高斯RBF的表达式为：

$$\phi\gamma(x,l) = \exp(-\gamma||x-l||^2)$$

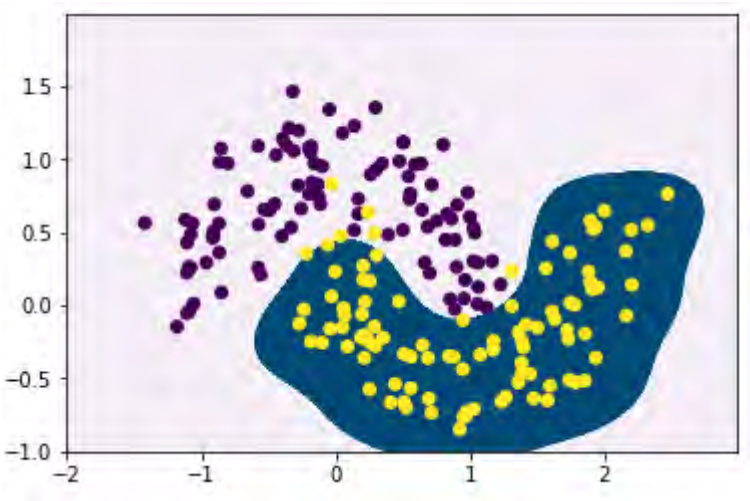
如下左图为一维线性不可分的情况，比如通过设置参数γ为0.3，参数l（landmark）为-2和1，可以看到RBF函数对应的函数曲线类似钟形。假设为$x_1 = -1$增加特征，$x_2 = \exp(-0.3 \times (-1-(-2))^2) \approx 0.74$, $x_2 = \exp(-0.3 \times (-1-(1))^2) \approx 0.30$。将所有点增加特征后如右图，可以发现增加特征后能够线性可分。



如何选择landmark。 最简单的方法是在数据集中的每个实例的位置创建一个landmark。这创建了许多特征，从而使得变换后的训练集将是线性可分的。缺点是具有m个实例和n个特征的训练集被转换成具有m个实例和m个特征的训练集（假设放弃了原始特征）。如果训练集非常大，则会得到大量的特征，影响计算速度。

然而对于SVM，正如上面所说可以并不真正增加特征而达到同样的效果，下面利用SVC类的RBF核来试验一下：

```
#训练RBF核SVM
rbf_kernel_svm_clf = Pipeline((
("scaler", StandardScaler()),
("svm_clf", SVC(kernel="rbf", gamma=5, C=0.001))
))
rbf_kernel_svm_clf.fit(X, y)
#画出决策线
import numpy as np
import matplotlib.pyplot as plt
xx, yy = np.meshgrid(np.arange(-2,3,0.01), np.arange(-1,2,0.01))
y_new=rbf_kernel_svm_clf.predict(np.c_[xx.ravel(),yy.ravel()])
plt.contourf(xx, yy, y_new.reshape(xx.shape),cmap="PuBu")
plt.scatter(X[:,0],X[:,1],marker="o",c=y)
```



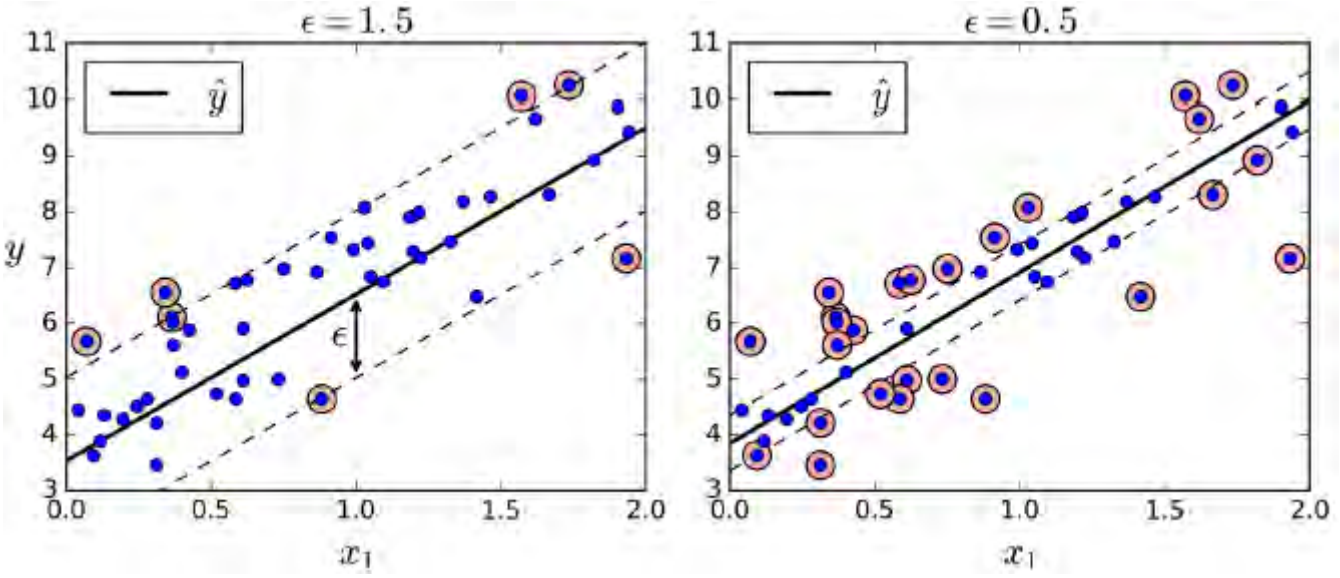
可以看到实现了非线性分类，然而决策边界范围很小，如果γ比较大，会使得决策线变窄，变得不规则。相反，小的γ使决策线变宽，边平滑。所以γ就像一个正则化参数：如果你的模型过拟合，可以适当减少它，如果它欠拟合，可以增加它（类似于C超参数）。

除了多项式核和高斯RBF核以外还有一些别的核，但都不太常用，一般用于一些特殊数据。

有这么多的内核可供选择，应该使用哪一个？一个经验法则，应该总是先尝试线性内核（LinearSVC比SVC（kernel =“linear”）快得多），特别是当训练集非常大或者有很多特征的时候。如果训练集不是太大，应该尝试高斯RBF核; 它在大多数情况下运行良好。如果你有空闲时间和计算能力，你还可以使用交叉验证和网格搜索来试验其他一些内核，特别是如果有专门用于你的训练集的数据结构的内核。

3、SVM回归（SVM Regression ）

SVM除了能进行分类任务以外还能做回归任务。与SVM分类任务尽量让点在margin以外，而SVM回归则是尽量让点在margin以内通过参数ε控制margin的大小，ε越大，margin越大，否则越小。（如下图）



下面为训练一个线性SVM回归模型的例子，对应的类为LinearSVR

```
from sklearn.svm import LinearSVR
svm_reg = LinearSVR(epsilon=1.5)
svm_reg.fit(X, y)
```

SVM非线性回归问题与分类问题类似，通过设置核来实现。

```
from sklearn.svm import SVR
svm_poly_reg = SVR(kernel="poly", degree=2, C=100, epsilon=0.1)
svm_poly_reg.fit(X, y)
```

六、决策树

1、训练决策树并其可视化

下面是决策树分类（DecisionTreeClassifier）用在Iris分类上的粒子。参数max_depth控制决策树的深度。

```
from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier
iris = load_iris()
X = iris.data[:, 2:] # petal length and width
y = iris.target
tree_clf = DecisionTreeClassifier(max_depth=2)
tree_clf.fit(X, y)
```

训练完以后可以预测分为每一类的概率，或最终结果

```
tree_clf.predict_proba([[5, 1.5]])
tree_clf.predict([[5, 1.5]])
```

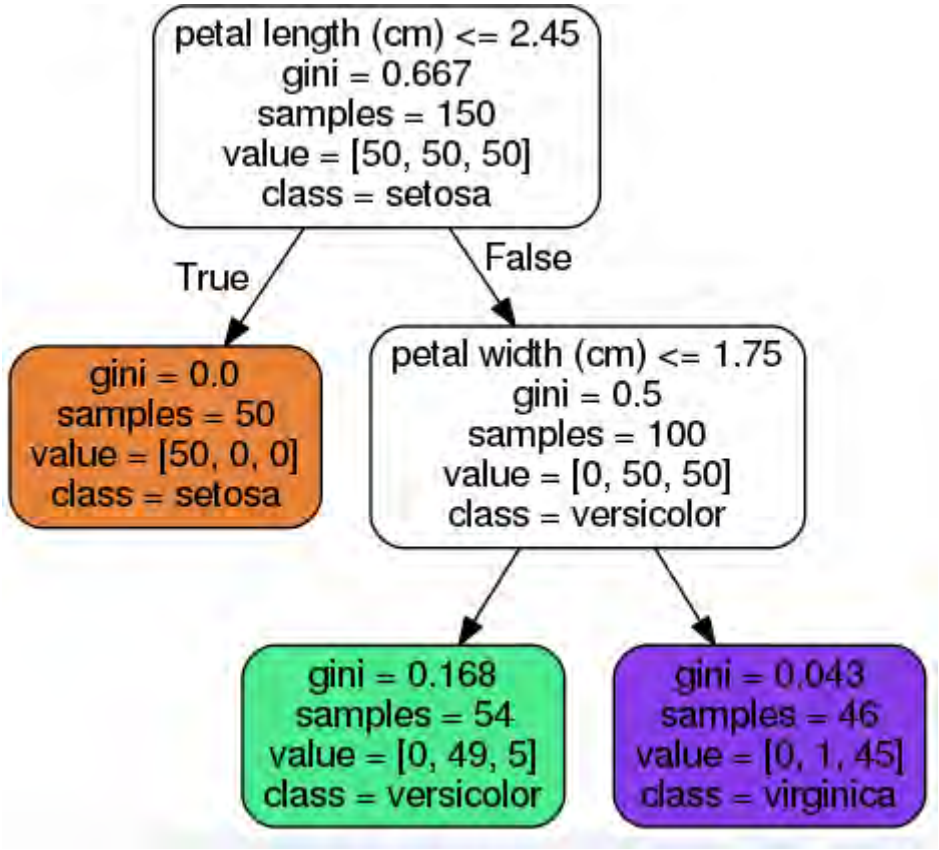
```
In [35]: tree_clf.predict_proba([[5, 1.5]])
Out[35]: array([[ 0.        ,  0.90740741,  0.09259259]])

In [36]: tree_clf.predict([[5, 1.5]])
Out[36]: array([1])
```

还可以用export_graphviz()方法把决策树画出来，输出的是.dot文件

```
from sklearn.tree import export_graphviz
export_graphviz(
tree_clf,
out_file=image_path("iris_tree.dot"),
feature_names=iris.feature_names[2:],
class_names=iris.target_names,
rounded=True,
filled=True
)
```

.dot文件可以转换为PDF或png格式查看更加直观清晰。（linux使用dot命令转换需要graphviz包）

```
dot -Tpng iris_tree.dot -o iris_tree.png #linux代码
```



图中白色框的第一行为划分条件，gini为划分比例，samples为此时总共有多少样本，value为此时每一类样本的总数，class为分为哪一类。

因此如果有一个新的样本，则从顶端开始根据划分条件，不断往下判断，最终预测为某一类。

需要注意：决策树只需要很少样本就可以生成，而且不需要对特征进行缩放。而且Scikit使用的为CART algorithm，即每次只生成两个分支，而ID3等算法可以产生多个分支。

2、正则化参数（Regularization Hyperparameters）

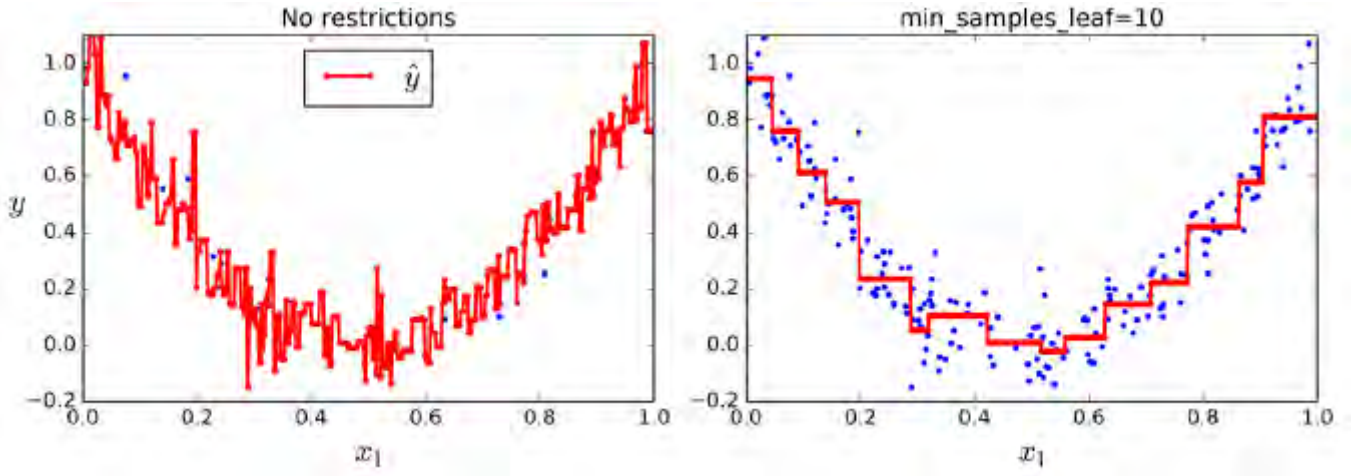
由于决策树算法对训练数据没有什么假设（相比线性模型假设决策线为一条线），这就对算法没有任何限制，因此很容易拟合训练数据，从而很容易导致过拟合。

因此为了防止对训练数据过拟合，需要增加一些参数来限制。最一般的设置应该设置最大深度（max_depth）；DecisionTreeClassifier类还有一些其他参数用来防止过拟合，节点被分开的最小样本数（min_samples_split）；叶子节点的最小样本数（min_samples_leaf）；和min_samples_leaf有点像，不过这个是分开节点变为叶子的最小比例，（min_weight_fraction_leaf）；叶子节点的最大样本数（max_leaf_nodes）；在每个节点分开时评估的最大特征数（max_features）。增加min_*，减小max_*都能正则化算法。

3、决策树回归（Decision Tree Regression）

决策树回归与决策树分类相似，不同的是决策树分类叶子节点最终预测的是类别，而决策树回归叶子节点最终预测的是一个值。

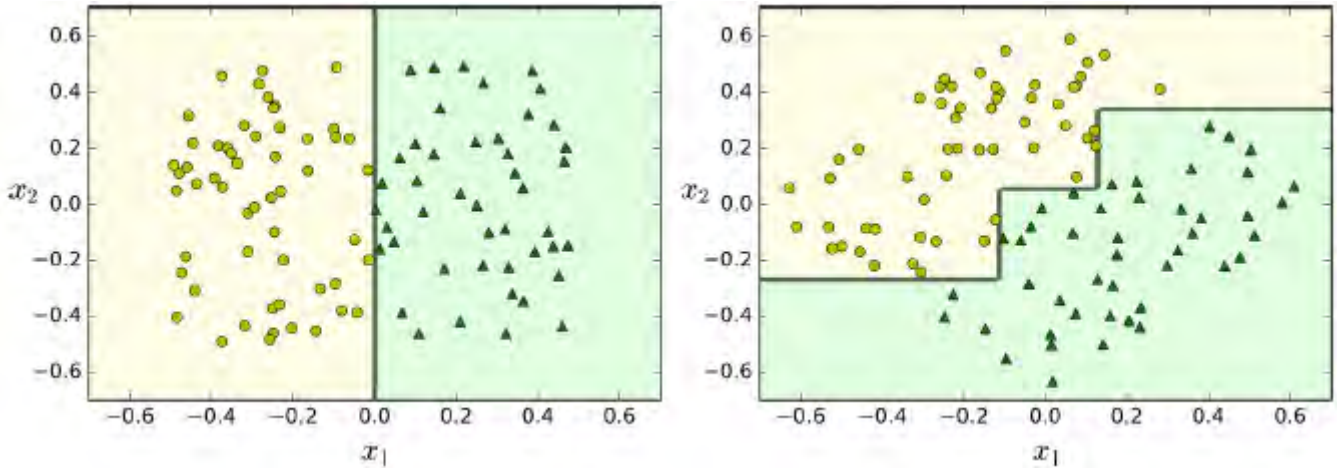
```
from sklearn.tree import DecisionTreeRegressor
tree_reg = DecisionTreeRegressor(max_depth=2)
tree_reg.fit(X, y)
```



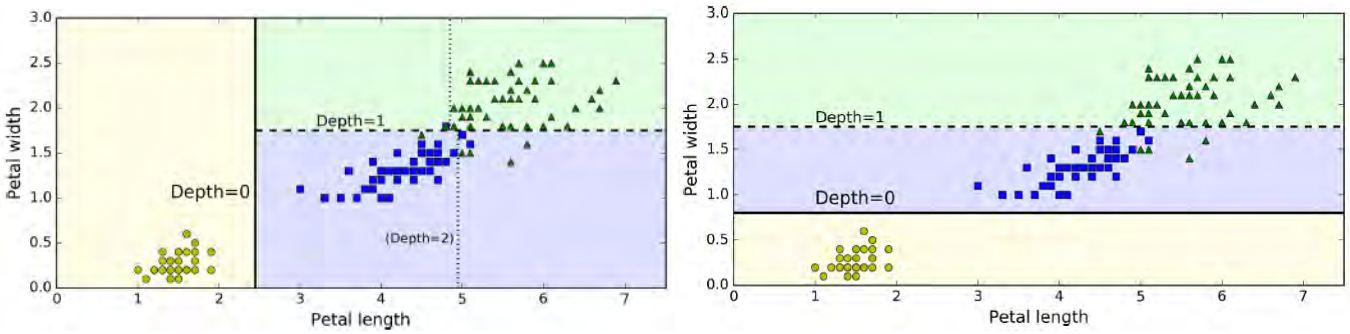
可以看到如果不加任何约束，则会产生过拟合。所以使用决策树回归时也要增加正则化参数。

4、局限性（不稳定性）

虽然决策树易于使用，易于展示与理解，但是存在一些局限性。你可能注意到决策树喜欢正交决策边界（垂直），这使得它们对训练集旋转很敏感。例如下图为一个简单的可线性分离的数据集：左边的决策树可以轻松地分割，而右边的数据集旋转45度后，决策边界看起来较为复杂。虽然这两个决策树都能拟合训练集，但是右边的模型很可能没有很好的泛化推广能力。解决这个问题的一种方法是使用PCA（后两节会说）来使训练数据更好的定位。



除了正交决策边界的缺陷以外，如果训练样本稍有变动，可能会导致决策线发生巨大改变，如左图删掉1个点以后决策线变为了右图，相差非常大，说明决策树算法非常不稳定。



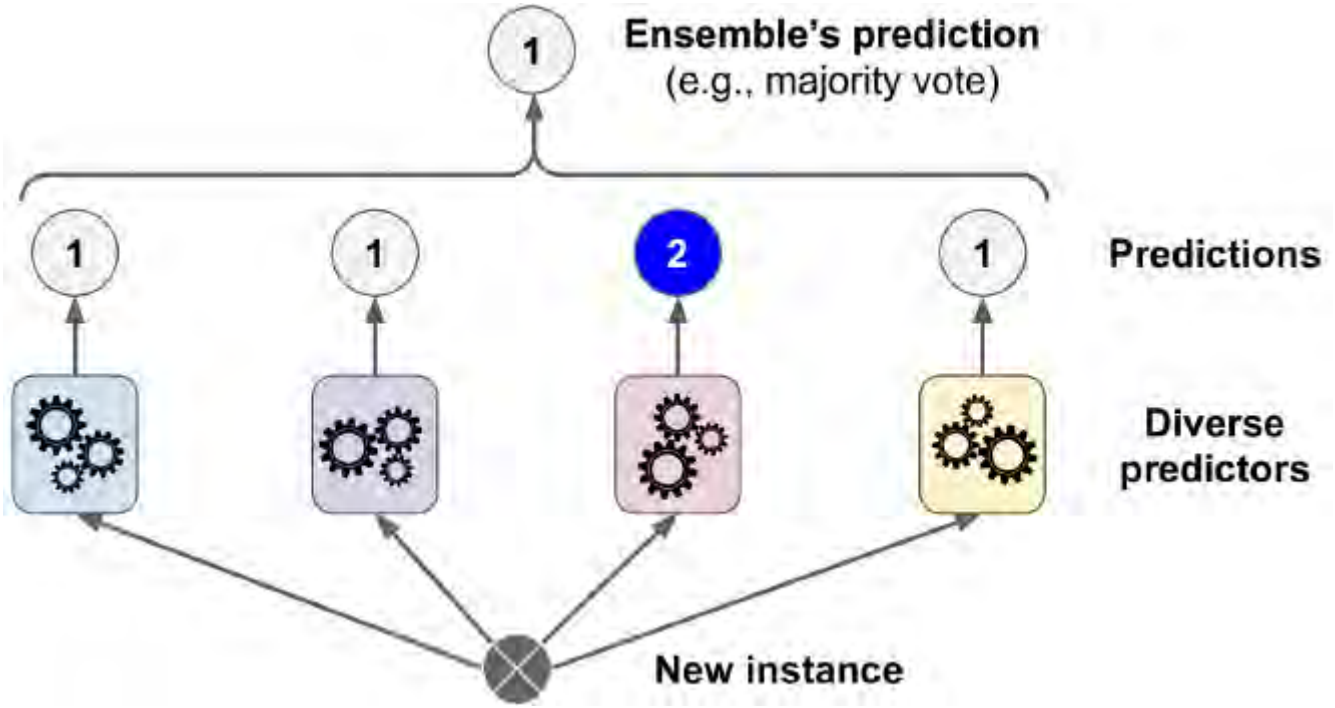
随机森林可以通过对多个决策树作平均来限制这种不稳定性，在下一节的集成算法我们看到。

七、集成学习与随机森林

假设要解决一个复杂的问题，让众多学生去回答，然后汇总他们的答案。在许多情况下，会发现这个汇总的答案比一个老师的答案要好。同样，如果汇总了一组预测变量（例如分类器或回归因子）的预测结果，则通常会得到比最佳个体预测变量得到更好的预测结果。这种技术被称为集成学习（Ensemble Learning）。

1、投票分类器（Voting Classifiers ）

创建集成分类器的一个非常简单的方法是聚合多个分类器（如Linear，Logistic，SVM，k-近邻、Ramdom forest等）的预测结果并预测得到最多选票的类。这个多数投票分类器被称为硬投票（hard voting）分类器。



这个投票分类器通常比集合中最好的分类器实现更高的准确性。事实上，即使每个分类器是一个弱学习器（weak learner）（意味着它只比随机猜测略好一点），但如果有足够的数量，集合仍然是一个强大的学习器（达到高精度）。比如每个学习器能到达51%的精度，1000个这样的分类器，通过投票取最大的分类，则能达到接近75%的精度。分类器越多精度更高。

需要注意：上面的情况只有每个分类器完全独立才能达到这种程度，相似的分类器很可能会犯同样的错误，所以会有大部分错误的选票，降低集体的准确性。因此获得不同分类器的一种方法是使用非常不同的算法来训练。这增加了会犯很多不同类型错误的机会，从而提高了整体的准确性。

下面是一个集成Logistic回归，SVM分类，Random forest分类的投票分类器，实验数据由moon产生，由于是硬投票，所以voting要设置为hard。

```
#产生moon数据并分开训练测试集
from sklearn.datasets import make_moons
from sklearn.model_selection import train_test_split
(X,y)=make_moons(1000,noise=0.5)
X_train, X_test, y_train, y_test = train_test_split(X,y,test_size=0.2, random_state=42)

#构造模型和集成模型
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import VotingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
log_clf = LogisticRegression()
rnd_clf = RandomForestClassifier()
svm_clf = SVC()
voting_clf = VotingClassifier(
estimators=[('lr', log_clf), ('rf', rnd_clf), ('svc', svm_clf)],
voting='hard'
)
voting_clf.fit(X_train, y_train)

#训练并预测
```

```
from sklearn.metrics import accuracy_score
for clf in (log_clf, rnd_clf, svm_clf, voting_clf):
    clf.fit(X_train, y_train)
    y_pred = clf.predict(X_test)
    print(clf.__class__.__name__, accuracy_score(y_test, y_pred))
```

```
('LogisticRegression', 0.81999999999999995)
('RandomForestClassifier', 0.79000000000000004)
('SVC', 0.81999999999999995)
('VotingClassifier', 0.82499999999999996)
```

从结果可以看到集成的投票分类器的正确率比其中最好的分类器要高（但并不是每次都会更好，不过一般也能接近最好）。

如果所有分类器都能够估计分为每一类的概率，即都有predict_proba()方法，那么可以对每个分类器的概率取平均，再预测具有最高类概率的类，这被称为软投票(soft voting)。它通常比硬投票(hard voting)取得更好的效果，因为它给予了高度信任的投票更多的权重。 因此把voting =“soft”替换voting =“hard”，并确保所有分类器都可以估计类的概率即可。

由于默认情况下SVC类没有predict_proba()方法，所以需要将它的probability参数设置为True（这将使SVC类使用交叉验证来估计类概率，减慢训练速度，并且会增加一个predict_proba（） 方法）。

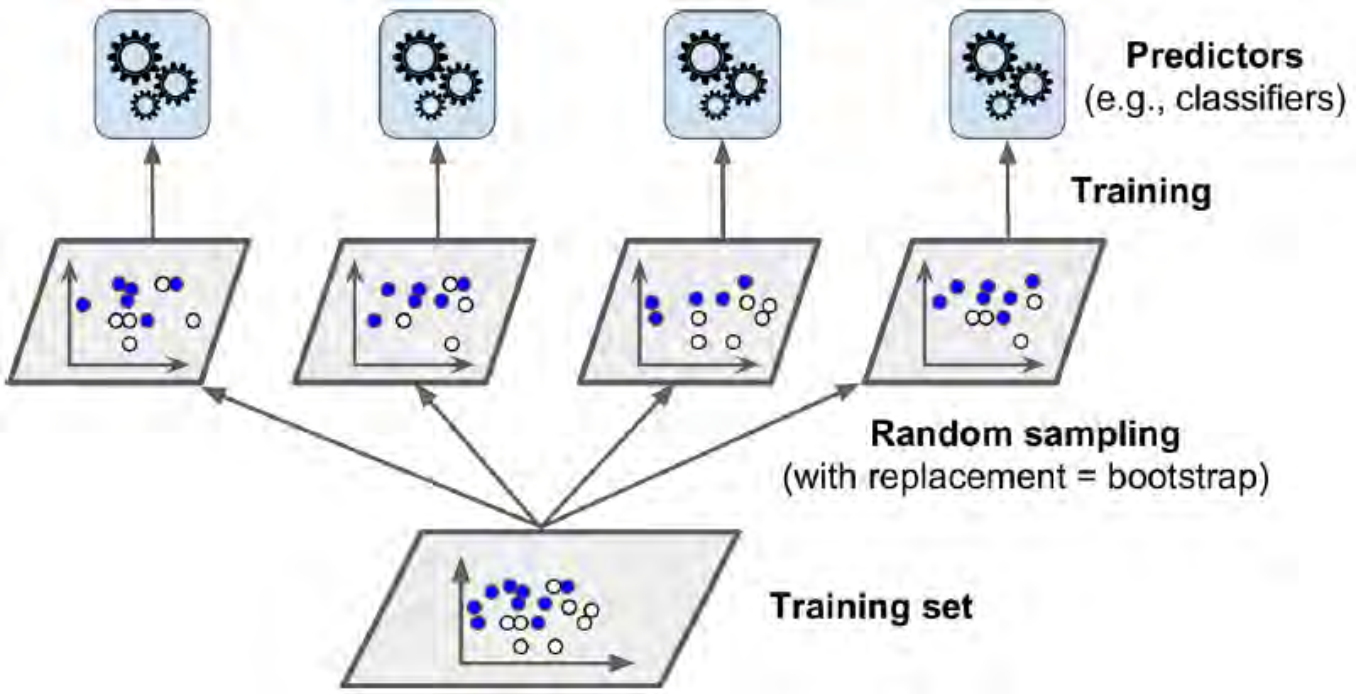
```
#修改SVC类使得有predict_proba()方法，并软投票
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import VotingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
log_clf = LogisticRegression()
rnd_clf = RandomForestClassifier()
svm_clf = SVC(probability=True)
voting_clf = VotingClassifier(
estimators=[('lr', log_clf), ('rf', rnd_clf), ('svc', svm_clf)],
voting='soft'
)
voting_clf.fit(X_train, y_train)

#训练并预测
from sklearn.metrics import accuracy_score
for clf in (log_clf, rnd_clf, svm_clf, voting_clf):
    clf.fit(X_train, y_train)
    y_pred = clf.predict(X_test)
    print(clf.__class__.__name__, accuracy_score(y_test, y_pred))
```

```
('LogisticRegression', 0.81999999999999995)
('RandomForestClassifier', 0.79000000000000004)
('SVC', 0.81999999999999995)
('VotingClassifier', 0.83499999999999996)
```

可以看到软投票比硬投票确实要好一点。

2、Bagging and Pasting

上述获得多种分类器的方式是使用不同的训练算法。另一种方法是对每个分类器使用相同的算法，但是要在训练集的不同随机子集上进行训练。如果抽样时有放回，称为Bagging；当抽样没有放回，称为Pasting。抽样与培训过程如图所示。



与投票分类器一样，最后结果预测为被预测最多的类（分类问题）或平均值（回归问题）。一般来说，相比在完整的训练数据上得到的结果，集成学习得到的结果具有类似的偏差和较低的方差。而且这种算法具有并行性。

下面使用bagging算法训练模型，选择决策树分类器作为训练算法；n_estimators表示产生分类器的数目；max_samples为每个分类器分得的样本数；bootstrap=True表示使用bagging算法，否则为pasting算法；n_jobs表示使用CPU核的数目，-1代表把能用的都用上。

```
from sklearn.ensemble import BaggingClassifier
from sklearn.tree import DecisionTreeClassifier
bag_clf = BaggingClassifier(
DecisionTreeClassifier(), n_estimators=500,
max_samples=100, bootstrap=True, n_jobs=-1
)
bag_clf.fit(X_train, y_train)
y_pred = bag_clf.predict(X_test)
print(bag_clf.__class__.__name__, accuracy_score(y_test, y_pred))
```

```
('BaggingClassifier', 0.83999999999999997)
```

可以看到算法比前面三个算法集成的投票分类器正确率还要高一点。相比pasting算法，bagging算法偏差会稍微高一些，但是方差更小，通常能取得不错的效果，所以通常情况下人们更愿意用bagging算法。

Out-of-Bag Evaluation

由于bagging算法采用有放回的抽样方式（自助采样法），假设训练集有m个样本，每次抽取一个后放回，直到抽到m个样本，那么样本始终没有被抽到的概率为$(1-\frac{1}{m})^m$，取极限得

$$\lim_{m\to\infty}(1-\frac{1}{m})^m=\frac{1}{e}\approx 0.368$$

这意味对于每一个分类器大约有36.8%的样本没有用于训练，这样的样本成为Out-of-Bag，所以可以使用这些样本得到结果的平均值来用于验证模型，而不需要划分训练验证集或交叉验证。在Scikit-learn中只需要设置参数oob_score=True即可使用这种方法估计。

```
bag_clf = BaggingClassifier(
DecisionTreeClassifier(), n_estimators=500,
bootstrap=True, n_jobs=-1, oob_score=True)
bag_clf.fit(X_train, y_train)
print("oob", bag_clf.oob_score_)

from sklearn.metrics import accuracy_score
y_pred = bag_clf.predict(X_test)
print("test", accuracy_score(y_test, y_pred))
```

```
('oob', 0.79249999999999998)
('test', 0.81000000000000005)
```

可以看到两个精度还是比较相似的，因此可以用与模型的验证。

3、Random Patches and Random Subspaces

除了能随机选样本创建多个子分类器以外还能够随机选择特征来创建多个子分类器，通过参数max_features和bootstrap_features实现，其含义与max_samples和bootstrap类似。对特征进行采样能够提升模型的多样性，增加偏差，减少方差。

当处理高维(多特征)数据（例如图像）时，这种方法比较有用。同时对训练数据和特征进行抽样称为Random Patches，只针对特征抽样而不针对训练数据抽样是Random Subspaces。

随机森林（Random forest）

随机森林算法是以决策树算法为基础，通过bagging算法采样训练样本，再抽样特征，3者组合成的算法。对应的scikit-learn中为RandomForestClassifier（RandomForestRegression），它有这决策树(DecisionTreeClassifier)的所有参数，以及bagging(BaggingClassifier)的所有参数。下面是一个例子：

```
from sklearn.ensemble import RandomForestClassifier
rnd_clf = RandomForestClassifier(n_estimators=500, max_leaf_nodes=16, n_jobs=-1)
rnd_clf.fit(X_train, y_train)
y_pred_rf = rnd_clf.predict(X_test)
```

Extra-Trees

随机森林算法每个分类器是从随机抽样部分特征，然后选择最优特征来划分。如果在此基础上使用随机的阈值分割这个最优特征，而不是最优的阈值，这就是Extra-Trees（Extremely Randomized Trees）。这会再次增加偏差，减少方差。一般来说，Extra-Trees训练速度优于随机森林，因为寻找最优的阈值比随机阈值耗时。对应scikit-learn的类为ExtraTreesClassifier（ExtraTreesRegressor），参数与随机森林相同。

Extra-tree和随机森林哪个更好不好比较，只能通过交叉验证两种算法都实验一次才能知道结果。

特征重要性(Feature Importance)

由于决策树算法根据最优特征分层划分的，即根部的特征更为重要，而底部的特征不重要（不出现的特征更不重要）。根据这个可以判断特征的重要程度，在Scikit-learn可以通过feature_importance获得特征的重要程度。下面iris数据训练一个随机森林模型，输出特征的重要程度。

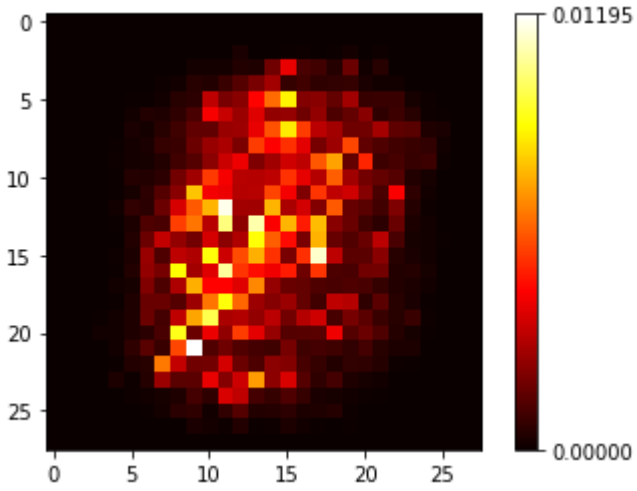
```
from sklearn.datasets import load_iris
iris = load_iris()
rnd_clf = RandomForestClassifier(n_estimators=500, n_jobs=-1)
rnd_clf.fit(iris["data"], iris["target"])
for name, score in zip(iris["feature_names"], rnd_clf.feature_importances_):
    print(name, score)
```

```
('sepal length (cm)', 0.095071713111187645)
('sepal width (cm)', 0.025182660618959785)
('petal length (cm)', 0.44343097254907315)
('petal width (cm)', 0.43631465372077932)
```

可以看到petal length的重要性为44%，petal width为43%；

随机森林还能对图像中像素（特征）的重要程度，下面以MNIST图像为例子。

```
import matplotlib.pyplot as plt
import matplotlib
from sklearn.datasets import fetch_mldata
mnist = fetch_mldata('MNIST original')
rnd_clf = RandomForestClassifier(random_state=42)
rnd_clf.fit(mnist["data"], mnist["target"])

importances = rnd_clf.feature_importances_.reshape(28, 28)
plt.imshow(importances, cmap = matplotlib.cm.hot)
plt.colorbar(ticks=[rnd_clf.feature_importances_.min(), rnd_clf.feature_importances_.max()])
```

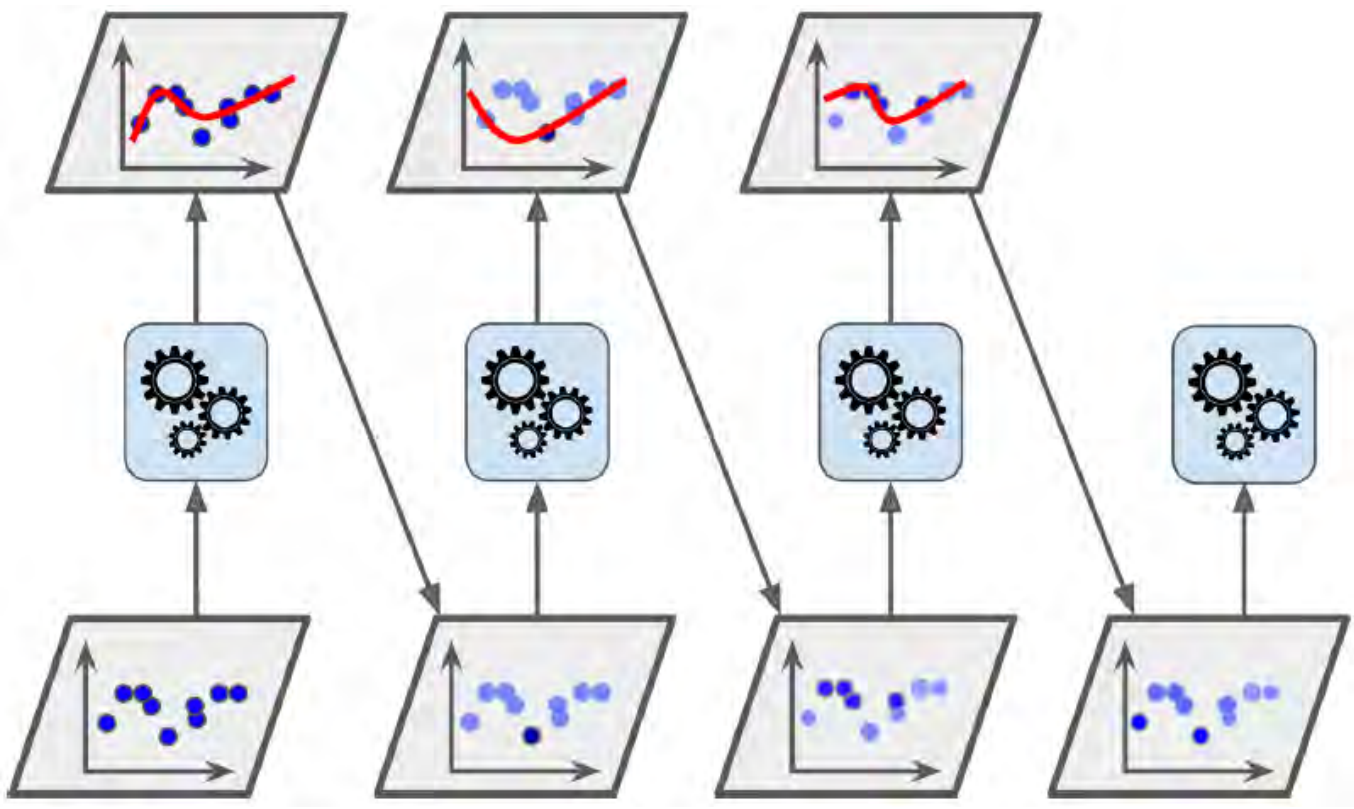


4、Boosting

Boosting是将弱学习器集成为强学习器的方法，主要思想是按顺序训练学习器，以尝试修改之前的学习器。Boosting的方法有许多，最为有名的方法为AdaBoost（Adaptive Boosting）和Gradient Boosting。

AdaBoost

一个新的学习器会更关注之前学习器分类错误的训练样本。因此新的学习器会越来越多地关注困难的例子。这种技术成为AdaBoost。



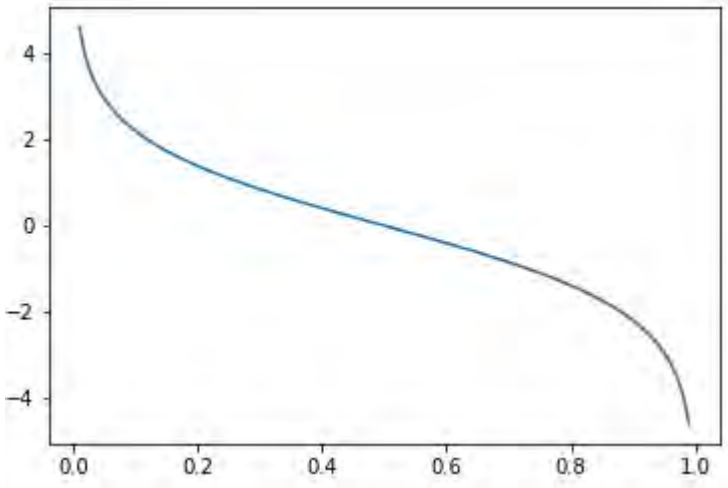
下面介绍AdaBoost算法是怎么工作的：

(1)假设有m个训练样本，初始化每个样本的权值$\mathrm{w}^{(i)}$为$\frac{1}{m}$，经过第j个学习器训练以后，对训练样本计算加权错误率r_{j}:

$$r_{\mathrm{j}}=\frac{\sum_{i=1}^{m} w^{(i)}}{\sum_{i=1}^{m} w^{(i)}}$$

(2)然后计算每个学习器对应的权值α_j。（如下图）当r_{j}比较小时，说明该学习器的准确率越高，比随机猜（0.5）越好，分配的权值也越大；如果随机猜（0.5），权值为0；如果小于随机猜，则为负值。其中η为学习率（和梯度下降法有点相似）

$$\alpha_j=\eta \log \frac{1-r_j}{r_j}$$



（3）更新样本权值$\mathrm{w}^{(i)}$，将没有预测出来的样本权值变大，以便后续的学习器重点训练。当然这个计算完以后需要归一化。

$$w^{(i)} = w^{(i)} \qquad \hat{y}_j^{(i)} = y^{(i)}$$
$$w^{(i)} = w^{(i)} \exp(\alpha_j) \qquad \hat{y}_j^{(i)} \neq y^{(i)}$$

（4）重复（1）（2）（3）步骤不断更新权值和训练新的学习器，直到学习器到一定的数目。

（5）最终得到N个学习器，计算每个学习器对样本的加权和，并预测为加权后最大的一类。

$$\hat{y}(x) = \arg\max_k \sum_{\substack{j=1 \\ \hat{y}_j(x)=k}}^{N} \alpha_j$$

可以看到上述的AdaBoost是二分类学习器，Scikit-learn中对应为AdaBoostClassifier类，如果要多分类，则可以设置参数algorithm=”SAMME”,如果需要predict_proba()方法，则设置参数algorithm=”SAMME.R”,下面为以决策树为基学习器，的多分类任务例子。

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import AdaBoostClassifier
ada_clf = AdaBoostClassifier(
                             DecisionTreeClassifier(max_depth=1), n_estimators=200,
                             algorithm="SAMME.R", learning_rate=0.5
                             )
ada_clf.fit(X_train, y_train)
```

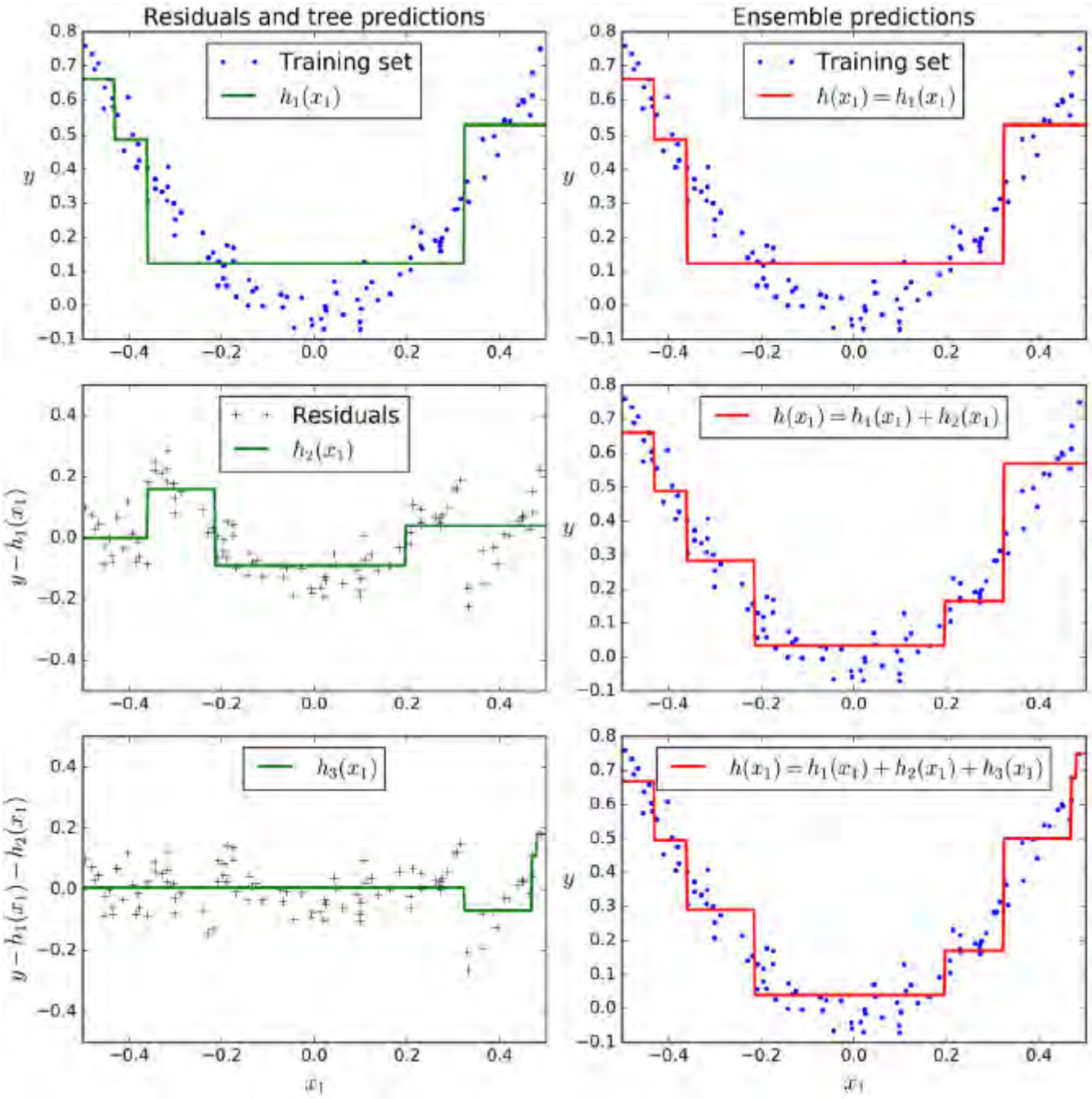
需要注意：SVM算法由于训练速度慢且不稳定，所以不适合AdaBoost的基算法；如果产生过拟合可以减少学习器的数目；AdaBoost的缺点为不能并行，由于每一个学习器依赖上一个学习器。

Gradient Boosting

和AdaBoost类似，Gradient Boosting也是逐个训练学习器，尝试纠正前面学习器的错误。不同的是，AdaBoost纠正错误的方法是更加关注前面学习器分错的样本，Gradient Boosting（适合回归任务）纠正错误的方法是拟合前面学习器的残差（预测值减真实值）。

下面训练以决策树回归为基算法，根据上一个学习器的残差训练3个学习器。数据使用二次加噪声数据。

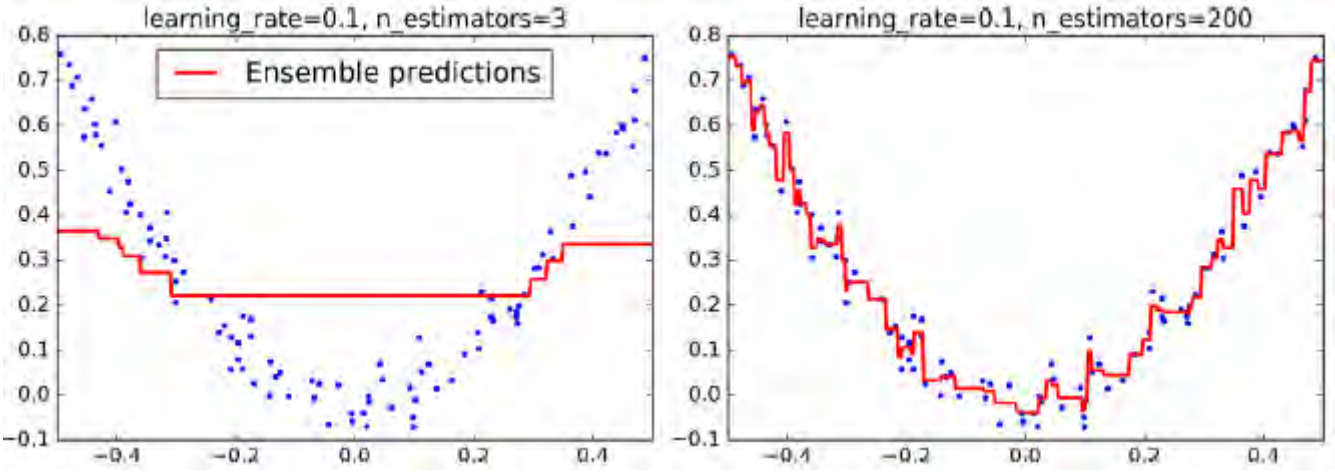
```
import numpy.random as rnd
from sklearn.tree import DecisionTreeRegressor
#准备数据
X = rnd.rand(200, 1) - 1
y = 3*X**2 + 0.05 * rnd.randn(200,1)
#训练第一个模型
tree_reg1 = DecisionTreeRegressor(max_depth=2)
tree_reg1.fit(X, y)
#根据上一个模型的残差训练第二个模型
y2 = y - tree_reg1.predict(X)
tree_reg2 = DecisionTreeRegressor(max_depth=2)
tree_reg2.fit(X, y2)
#再根据上一个模型的残差训练第三个模型
y3 = y2 - tree_reg2.predict(X)
tree_reg3 = DecisionTreeRegressor(max_depth=2)
tree_reg3.fit(X, y3)
#预测
X_new=0.5
y_pred = sum(tree.predict(X_new) for tree in (tree_reg1, tree_reg2, tree_reg3))
```



如果将结果画成图，我们可以看到左侧为每个学习器的决策线，有图为相加后的结果，可以发现拟合效果越来越好。对应Scikit-learn中的函数为GradientBoostingRegressor（决策树为基）。

```
from sklearn.ensemble import GradientBoostingRegressor
gbrt = GradientBoostingRegressor(max_depth=2, n_estimators=3, learning_rate=1.0)
gbrt.fit(X, y)
```

参数learning_rate表示每个学习器的贡献程度，如果设置learning_rate比较低，则需要较多的学习器拟合训练数据，但是通常会得到更好的效果。下图展示较低学习率的训练结果，左图学习器过少，欠拟合；右图学习器过多，过拟合。



为了找到最优学习器的数量，可以使用early stopping方法（在第5节讲到）。对应可以使用staged_predict()方法，该方法能够返回每增加一个学习器的预测结果。下面为一个实例：

```
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
X_train, X_val, y_train, y_val = train_test_split(X, y)
gbrt = GradientBoostingRegressor(max_depth=2, n_estimators=120)
gbrt.fit(X_train, y_train)
errors = [mean_squared_error(y_val, y_pred) for y_pred in gbrt.staged_predict(X_val)]
bst_n_estimators = np.argmin(errors)
gbrt_best = GradientBoostingRegressor(max_depth=2,n_estimators=bst_n_estimators)
gbrt_best.fit(X_train, y_train)
```

还可以与第5节中一样通过设置warm_start = True使模型继续训练，当认为不能再下降时停止，而不是训练完最大数目的学习器再找最小错误的。下面为连续迭代五次的错误没有改善时，停止训练：

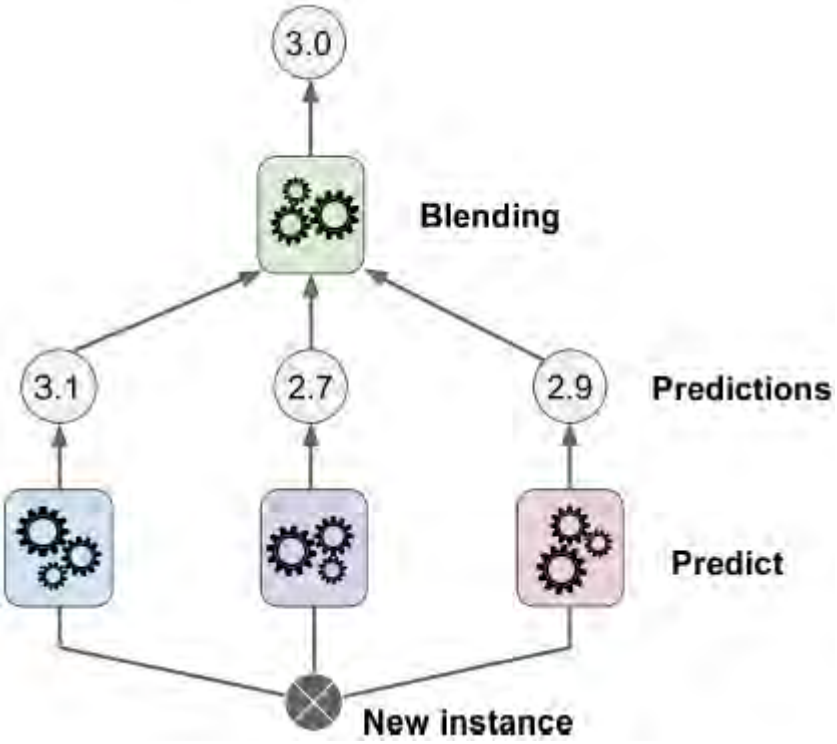
```
gbrt = GradientBoostingRegressor(max_depth=2, n_estimators=1, learning_rate=0.1, random_state=42, war
min_val_error = float("inf")
error_going_up = 0
for n_estimators in range(1, 120):
    gbrt.n_estimators = n_estimators
    gbrt.fit(X_train, y_train)
    y_pred = gbrt.predict(X_val)
    val_error = mean_squared_error(y_val, y_pred)
    if val_error < min_val_error:
        min_val_error = val_error
        error_going_up = 0
    else:
        error_going_up += 1
        if error_going_up == 5:
            break  # early stopping
print gbrt.n_estimators
```

GradientBoostingRegressor类可以通过设置参数subsample，来确定每棵树训练的样本比例，这如上面的bagging一样，可以获得较低的方差和高的偏差。但也大大加快训练速度。这种技术被称为Stochastic Gradient Boosting。

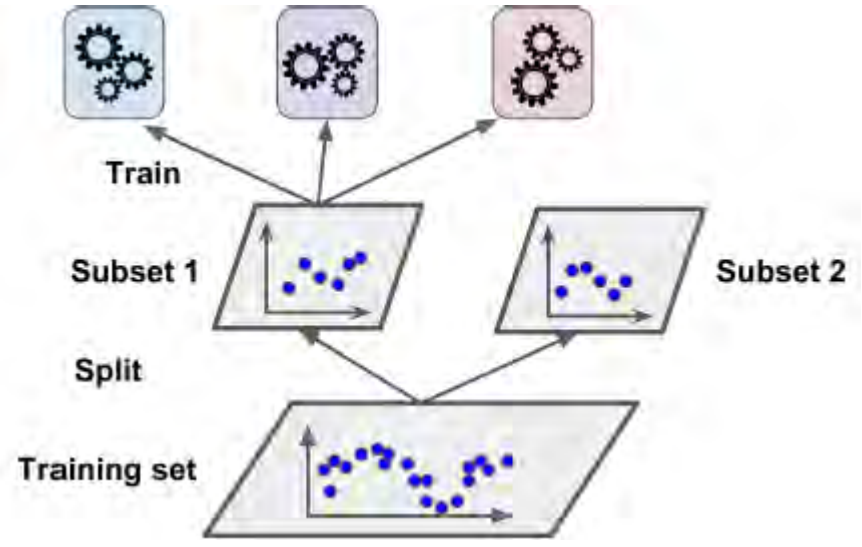
需要注意：Gradient Boosting还可以选择其他损失函数，通过设置参数loss来代替残差

5、Stacking

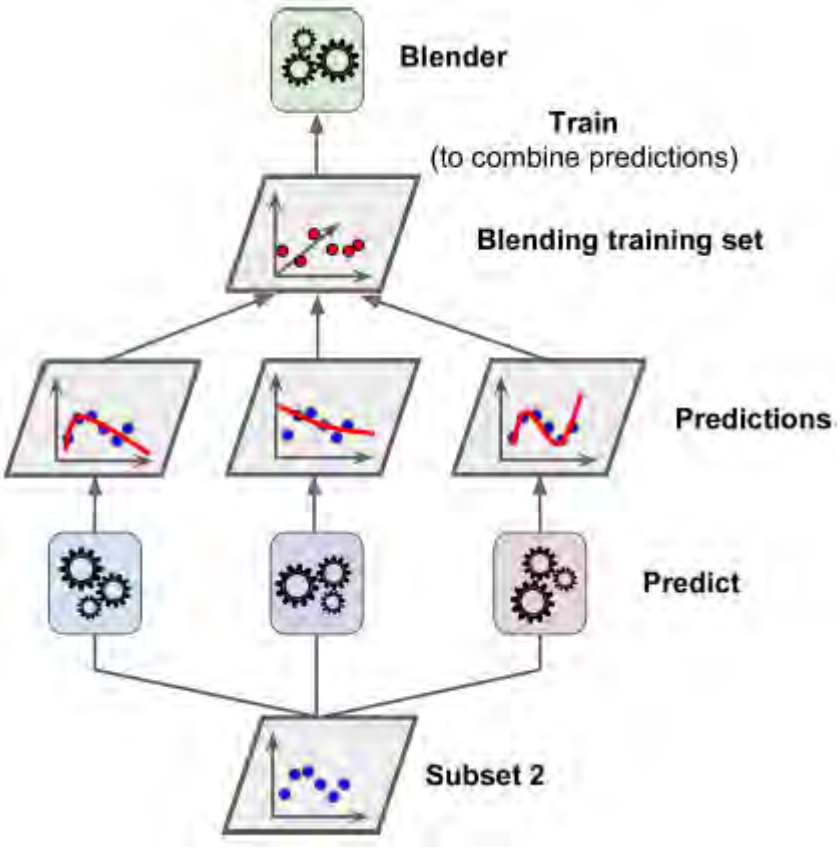
上述的模型都是通过训练多个学习器后分别得到结果后整合为最终结果，整合的过程为投票、求平均、求加权平均等统计方法。那为什么不把每个学习器得到的结果作为特征进行训练(Blend)，再预测出最后的结果，这就是Stacking的思想。



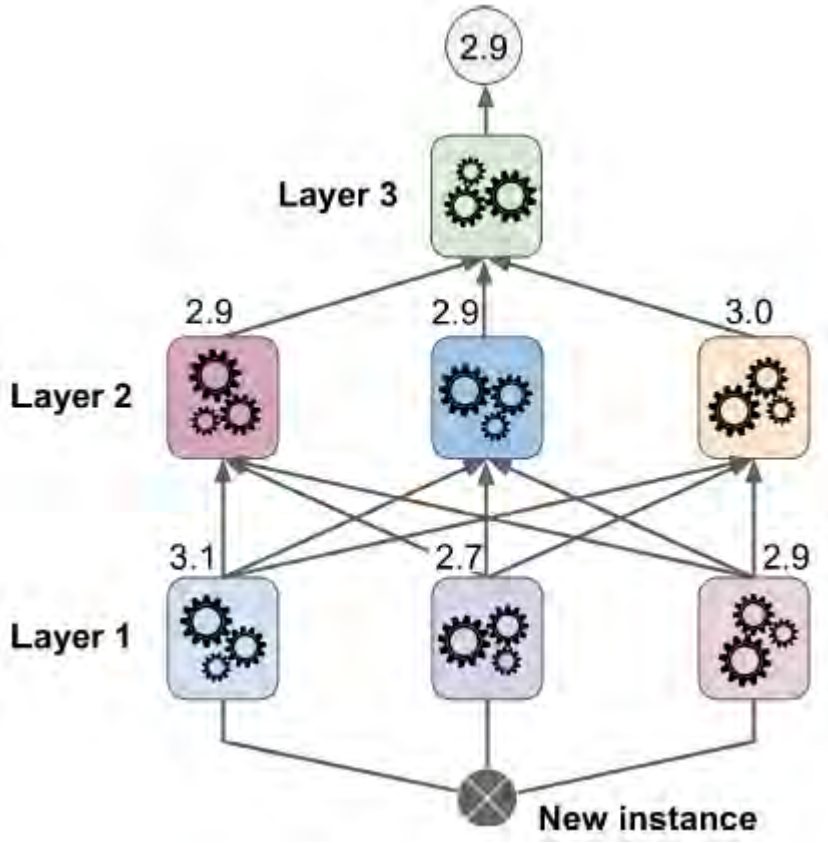
为了上述思想，需要将训练数据分为两部分。第1部分用于训练多个基学习器。



第2部分（hold-out set）用于训练blender。blender的输入为第2部分数据在第一部分数据训练好的多个模型的预测结果。



实际上可以训练多个blender（如一个Logistic回归，另一个Ramdomforest）实现这个思想的诀窍是将训练集分成三份，第一份用于训练多个基学习器，第二份用于训练第二个层（使用第一个层的预测器进行的预测作为输入），第三份用于训练第三层（使用第二层的预测器进行的预测作为输入）。如图所示。



不幸的是，scikit-learn没有提供stacking的实现，但是实现并不困难，可以参见（https://github.com/viisar/brew）。

八、降维

在现实生活中很多机器学习问题有上千维，甚至上万维特征，这不仅影响了训练速度，通常还很难找到比较好的解。这样的问题成为维数灾难（curse of dimensionality）

幸运的是，理论上降低维度是可行的。比如MNIST数据集大部分的像素总是白的，因此可以去掉这些特征；相邻的像素之间是高度相关的，如果变为一个像素，相差也并不大。

需要注意：降低维度肯定会损失一些信息，这可能会让表现稍微变差。因此应该先在原维度训练一次，如果训练速度太慢再选择降维。虽然有时候降为能去除噪声和一些不必要的细节，但通常不会，主要是能加快训练速度。

降维除了能训练速度以外，还能用于数据可视化。把高维数据降到2维或3维，然后就能把特征在2维空间（3维空间）表示出来，能直观地发现一些规则。

本节会将两种主要的降维方法(projection and Manifold Learning)，并且通过3中流行的降维技术：PCA，Kernel PCA和LLE。

1、维数灾难

在高维空间中，许多表现和我们在低维认识的差别很大。例如，在单位正方形（1×1）内随机选点，那么随机选的点离所有边界大于0.001（靠近中间位置）的概率为0.4%（$1 - 0.998^2$），但在一个10000维单位超立方体（1×1×…×1），这个概率大于99 999999%（$1 - 0.998^{10000}$）。因此高维超立方体中的大多数随机点都非常接近边界。

在高维空间中还有一个与认识不同的：如果在单位超立方体（1×1×…×1）随机选两个点，在2维空间，两个点之间平均距离约为0.52；在3维空间约为0.66，在1,000,000维空间，则约为408.25，可以看到在单位立方体中两个点距离竟然能相差这么大。因此在高维空间，这很可能会使得训练集和测试集相差很大，导致训练过拟合。

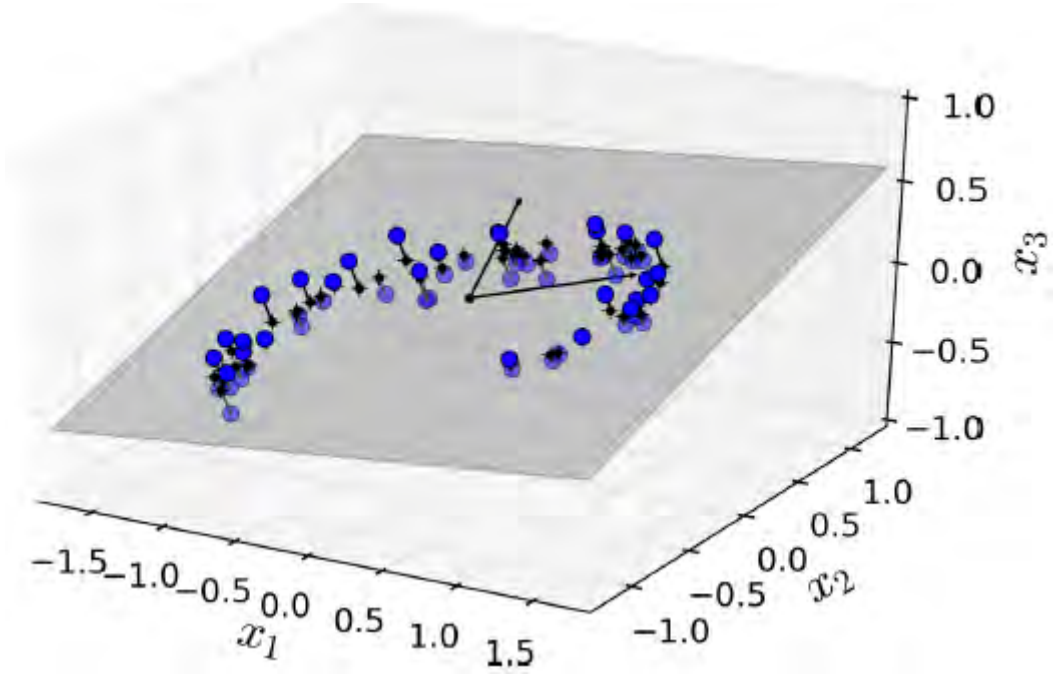
理论上，只要通过增加训练集，就能达到训练空间足够密集。但是随着维度的增加，需要的训练集是呈指数增长的。如果100维（比MNIST数据集还要小），要达到每个样本的距离为0.1（平均的分散开）的密度，则需要的样本为10^{100}个样本，可见需要的样本之多。

2、降维的主要方法

降为的方法主要为两种：projection 和 Manifold Learning。

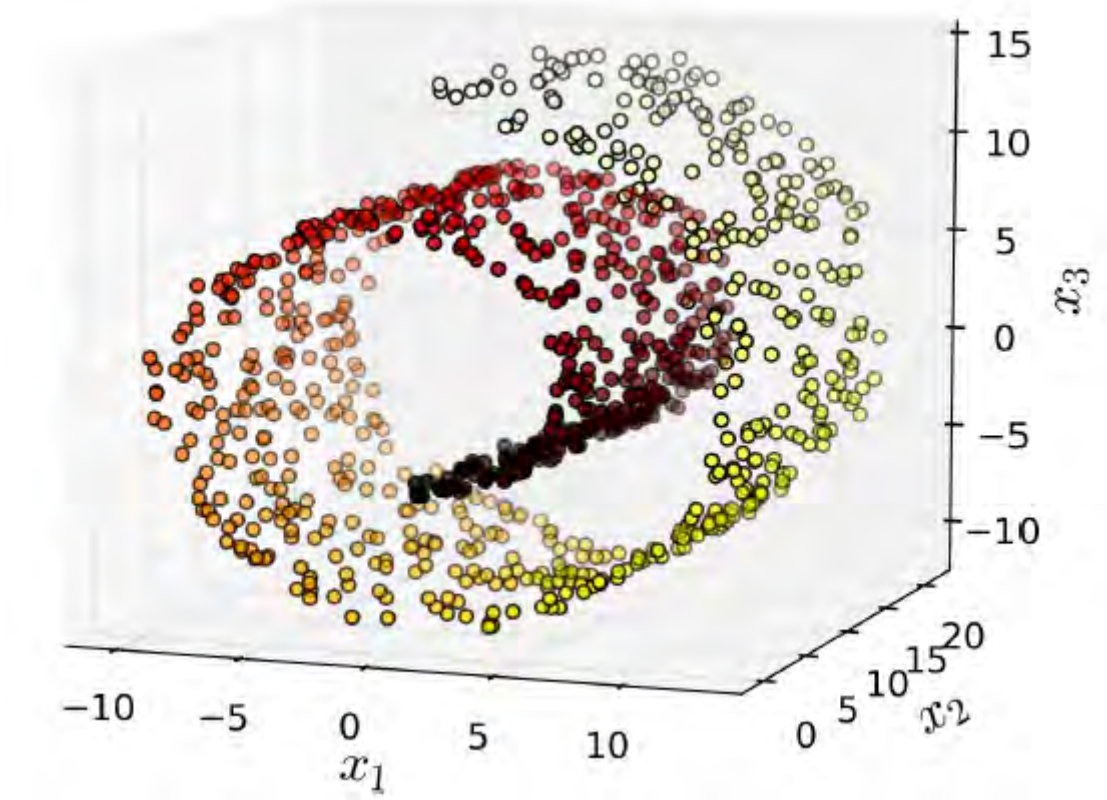
投影（Projection ）

在大多数的真实问题，训练样例都不是均匀分散在所有的维度，许多特征都是固定的，同时还有一些特征是强相关的。因此所有的训练样例实际上可以投影在高维空间中的低维子空间中，下面看一个例子。

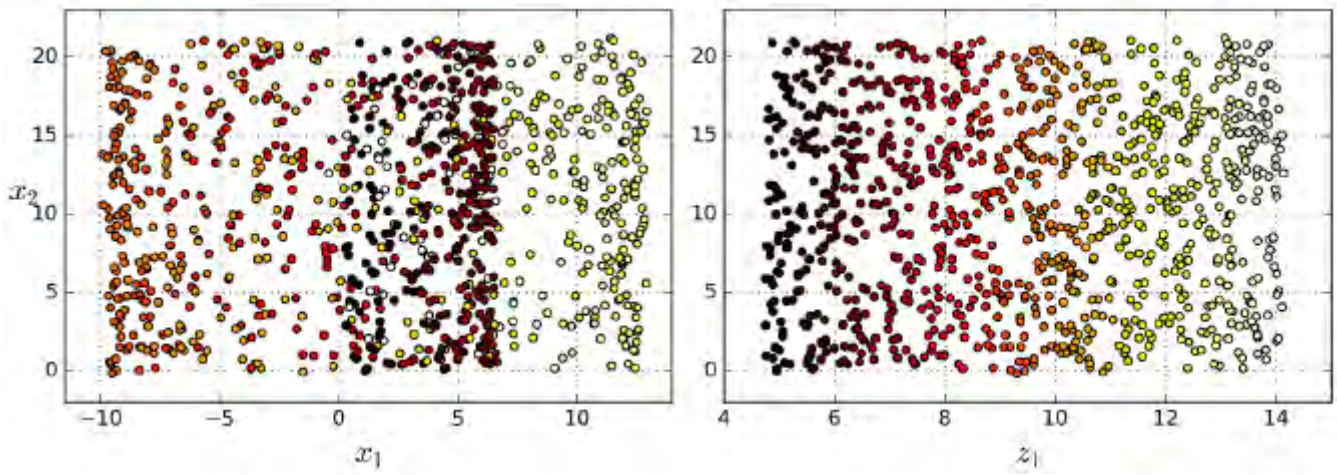


可以看到3维空间中的训练样例其实都分布在同一个2维平面，因此我们能够将所有样例都投影在2维平面。对于更高维的空间可能能投影到低维的子空间中。

然而投影（projection）不总是降维最好的方法在，比如许多情况下，空间可以扭转，如著名的瑞士卷（Swiss roll）数据。



如果简单的使用投影（project）降维（例如通过压平第3维），那么会变成如下左图的样子，不同类别的样例都混在了一起，而我们的预想是变成右下图的形式。



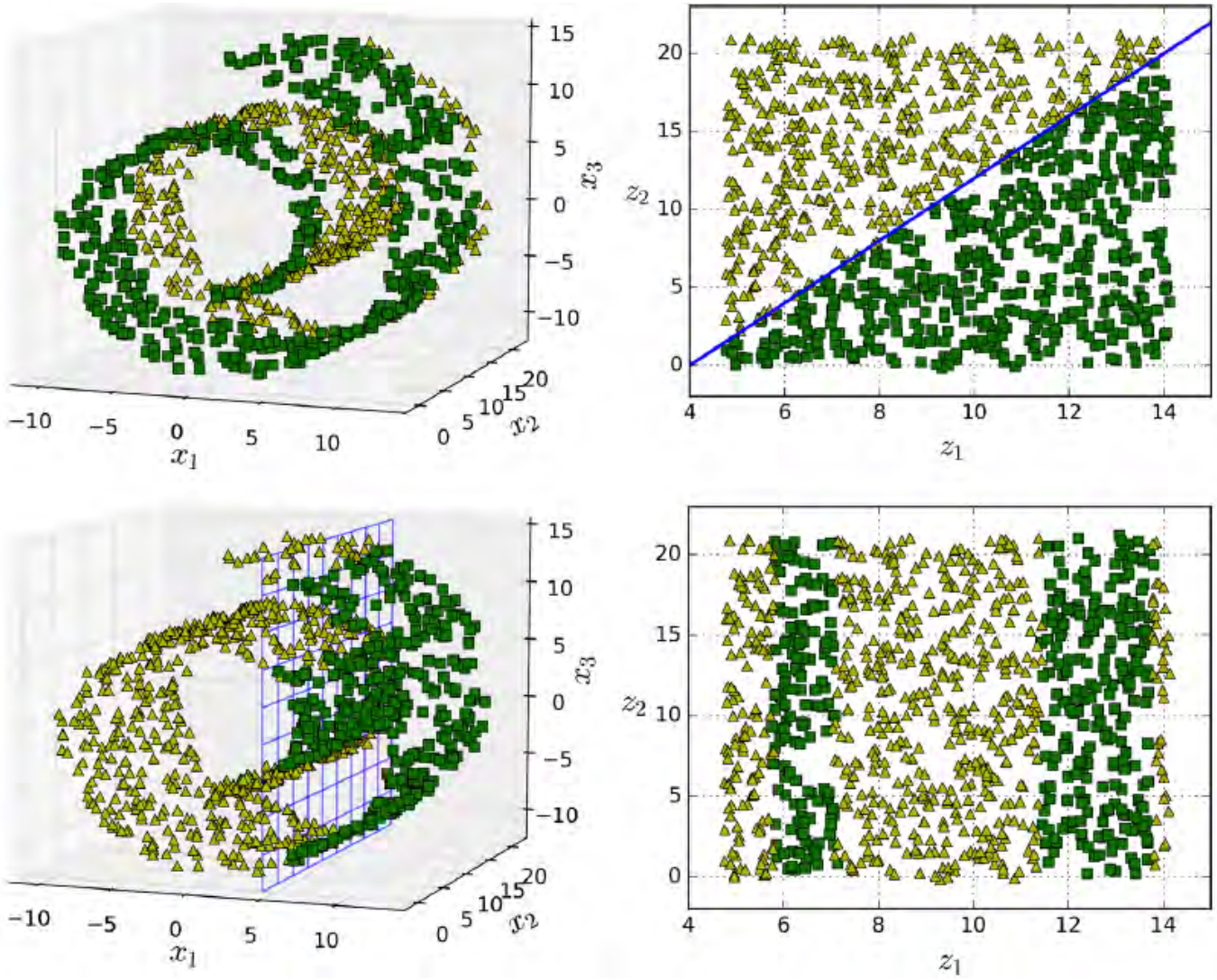
流行学习（Manifold Learning ）

瑞士卷（Swiss roll）是二维流形的例子。它可以在高维空间中弯曲。更一般地，一个d维流形在n维空间弯曲（其中d < n）。在瑞士卷的情况下，D = 2和n = 3。

基于流行数据进行建模的降维算法称为流形学习（Manifold Learning）。它假设大多数现实世界的高维数据集接近于一个低维流形。

流行假设通常隐含着另一个假设：通过流形在低维空间中表达，任务（例如分类或回归）应该变得简单。如下图第一行，Swiss roll分为两类，在3D的空间看起来很复杂，但通过流行假设到2D就能变得简单。

但是这个假设并不总是能成立，比如下图第二行，决策线为x=5，2D的的决策线明显比3D的要复杂。因此在训练模型之前先降维能够加快训练速度，但是效果可能会又增有减，这取决于数据的形式。



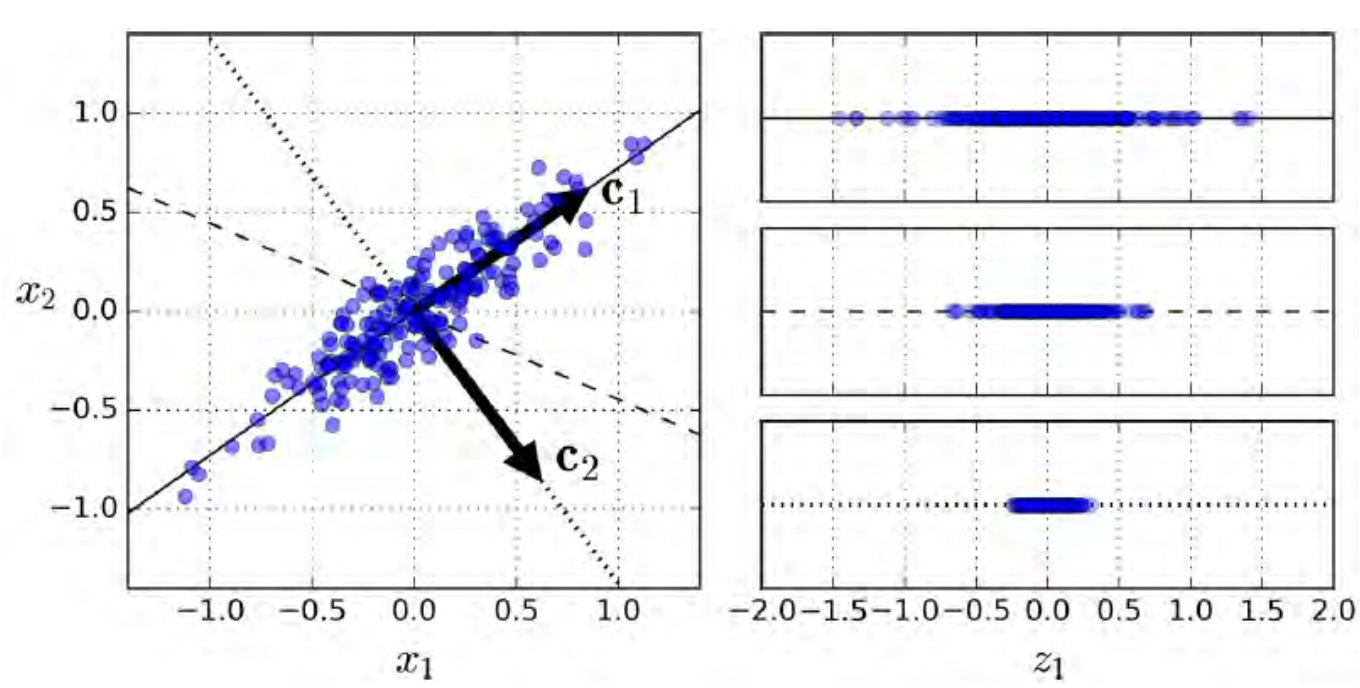
下面介绍几种著名的降维技术。

3、主成分分析（PCA）

主成分分析（PCA）是用的最出名的降维技术，它通过确定最接近数据的超平面，然后将数据投射(project)到该超平面上。

保留最大方差

首先需要选择一个好的超平面。先看下图的例子，需要将2D降为1D，选择不同的平面得到右图不一样的结果，第1个投影以后方差最大，第3个方差最小，选择最大方差的一个感觉上应该是比较合理的，因为这样能保留更多的信息。



另外一种判断的方式是：通过最小化原数据和投影后的数据之间的均方误差。

主成分（Principal Components ）

算法首先找到第一个主成分，使得投影后方差最大，如上图的c1。然后再找第二个主成分（要和第一个主成分正交）使得上面投影后的数据集再投影后方差最大，由于上图是2D平面，因此只有c2符合。如果是n维平面，则找第(t<=n）个主成分（要和1,2,…,t-1个主成分都正交），使得t-1投影后的数据再投影后方差最大。

需要注意：主成分的方向是不稳定的，如果稍微扰动数据，则可能使方向相反，不过仍会在同一个轴上。在某些情况下，一对主成分甚至可以旋转或交换，但它们定义的平面一般保持不变。

要如何才能在训练集中找到这些主成分？我们可以通过奇异值分解（SVD）来把X矩阵分解为三个矩阵$\mathrm{U}*\sum*V^{\mathrm{T}}$，其中$V^{\mathrm{T}}$矩阵的每一个列向量就是我们要找的主成分。

Numpy中提供svd()方法，下面为求主成分的例子

```
#产生数据
import numpy as np
x1=np.random.normal(0,1,100)
x2=x1*1+np.random.rand(100)
X=np.c_[x1,x2]
#svd分解求出主成分
X_centered = X - X.mean(axis=0)
U, s, V = np.linalg.svd(X_centered)
c1 = V.T[:, 0]
c2 = V.T[:, 1]
```

需要注意：PCA假设数据以原点为中心，因此要先减去均值再svd分解。Scikit-learn中的PCA类已经减去均值，但是如果在别的场景使用要记得先减去均值再求主成分。

投影到d维空间

得到主成分以后就能将数据降维，假设降到d维空间，则用数据矩阵与前d个主成分形成的矩阵相乘，得到降维后的矩阵。

$$X_d = X * W_d$$

对应的代码为：

```
d=1
Wd = V.T[:, :d]
XD = X_centered.dot(Wd)
```

使用Scikit-Learn

Scikit-learn提供了PCA类，n_components控制维数

```
from sklearn.decomposition import PCA
pca = PCA(n_components = 1)
XD = pca.fit_transform(X)
```

fit以后可以通过components_变量来输出主成分，还可以通过explained_variance_ratio_来查看每个主成分占方差的比率。

```
#查看主成分
pca.components_.T
#显示PCA主成分比率
print("主成分方差比率为：")
print(pca.explained_variance_ratio_)
```

```
主成分方差比率为：
[ 0.97005632]
```

在这个2D空间，可以看到第一个主成分占训练集方差的97%，即第二个主成分占训练集方差的3%，因此假设第二个主成分含有的信息比较少时比较合理的。

选择合理的维数

合理的选择维数而不是随机选择一个维数，我们可以通过设置一个合适的方差比率（如95%），计算需要多少个主成分的方差比率和能达到这个比率，就选择该维度，对应代码如下。（除非需要将数据降到2D或3D用于可视化等）

```
pca = PCA()
pca.fit(X)
cumsum = np.cumsum(pca.explained_variance_ratio_) #累加
d = np.argmax(cumsum >= 0.95) + 1
```

得到维度d后再次设置n_components进行PCA降维。当然还有更加简便的方法，直接设置n_components_=0.95，那么Scikit-learn能直接作上述的操作。

```
pca = PCA(n_components=0.95)
X_reduced = pca.fit_transform(X)
```

增量PCA（IPCA）

当数据量较大时，使用SVD分解会耗费很大的内存以及运算速度较慢。幸运的是，可以使用IPCA算法来解决。先将训练样本分为mini-batches，每次给IPCA算法一个mini-batch，这样就能处理大量的数据，也能实现在线学习（当有新的数据加入时）。

下面是使用Numpy的array_split()方法将MNIST数据集分为100份，再分别喂给IPCA，将数据降到154维。需要注意，这里对于mini-batch使用partial_fit()方法，而不是对于整个数据集的fit()方法。

```
#加载数据
from sklearn.datasets import fetch_mldata
mnist = fetch_mldata('MNIST original')
X = mnist["data"]
#使用np.array_split（）方法的IPCA
from sklearn.decomposition import IncrementalPCA
n_batches = 100
inc_pca = IncrementalPCA(n_components=154)
for X_batch in np.array_split(X, n_batches):
    inc_pca.partial_fit(X_batch)
X_mnist_reduced = inc_pca.transform(X)
```

还可以使用Numpy 的memmap类来操纵储存在硬盘上的二进制编码的大型数据，只有当数据被用到的时候才会将数据放入内存。由于IPCA每次只需将一部分数据，因此能通过memmap来控制内存。由于使用的是输入的是整个数据集，因此使用的是fit（）方法。

```
X_mm = np.memmap(filename, dtype="float32", mode="readonly", shape=(m, n))
batch_size = m // n_batches
inc_pca = IncrementalPCA(n_components=154, batch_size=batch_size)
inc_pca.fit(X_mm)
```

随机PCA

随机PCA是个随机算法，能快速找到接近前d个主成分，它的计算复杂度为$O(m * d^3) + O(d^3)$，而不是$O(m * n^3) + O(n^3)$，如果d<

```
rnd_pca = PCA(n_components=154, svd_solver="randomized")
X_reduced = rnd_pca.fit_transform(X_mnist)
```

4、核PCA(Kernel PCA)

在第六节的SVM中提到了核技巧，即通过数学方法达到增加特征类似的功能来实现非线性分类。类似的技巧还能用在PCA上，使得可以实现复杂的非线性投影降维，称为kPCA。该算法善于保持聚类后的集群(clusters)后投影，有时展开数据接近于扭曲的流形。下面是使用RBF核的例子。

```
#生成Swiss roll数据
from sklearn.datasets import make_swiss_roll
data=make_swiss_roll(n_samples=1000, noise=0.0, random_state=None)
X=data[0]
y=data[1]
#画3维图
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
```

```
ax = plt.subplot(111, projection='3d')
ax.scatter(X[:,0], X[:,1], X[:,2],c=y)
plt.show()
#kPCA
from sklearn.decomposition import KernelPCA
rbf_pca = KernelPCA(n_components = 2, kernel="rbf", gamma=0.04)
X_reduced = rbf_pca.fit_transform(X)
```

需要注意，此方法要使用大量内存，可能会使内存溢出。

选择合适的核与参数

由于kPCA是非监督算法，因此无法判断性能的好坏，因此需要结合分类或回归问题来判断。通过GridSearch来选择合适的核与参数，下面是一个例子：

```
from sklearn.datasets import fetch_mldata
mnist = fetch_mldata('MNIST original')
X,y = mnist["data"],mnist["target"]

from sklearn.decomposition import KernelPCA
from sklearn.model_selection import GridSearchCV
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import Pipeline
clf = Pipeline([
("kpca", KernelPCA(n_components=2)),
("log_reg", LogisticRegression())
])
param_grid = [{
"kpca__gamma": np.linspace(0.03, 0.05, 10),
"kpca__kernel": ["rbf", "sigmoid"]
}]
grid_search = GridSearchCV(clf, param_grid, cv=3)
grid_search.fit(X, y)
print(grid_search.best_params_)
```

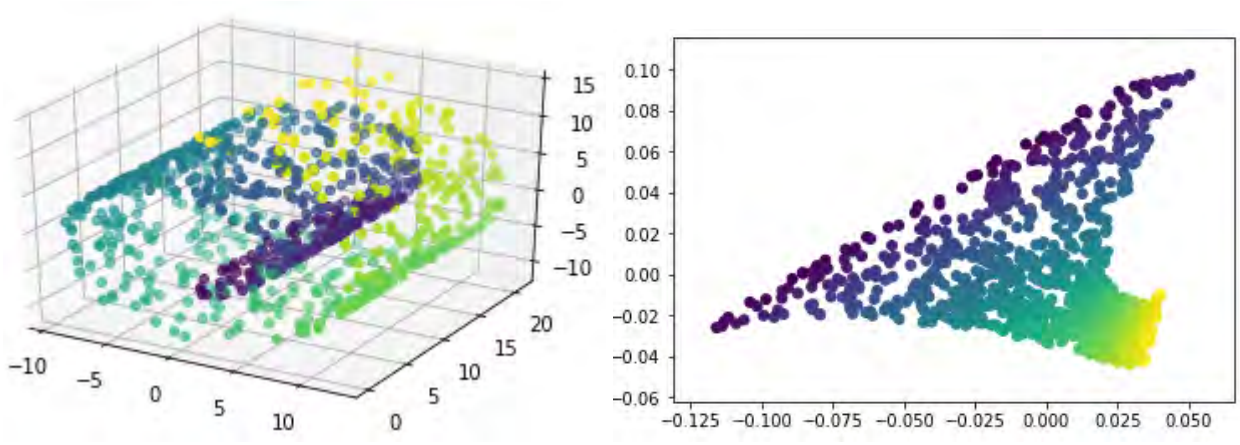
5、LLE

局部线性嵌入（Locally Linear Embedding）是另一种非线性降维技术。它基于流行学习而不是投影。LLE首先测量每个训练样例到其最近的邻居（CN）的线性关系，然后寻找一个低维表示使得这种相关性保持得最好（具体细节后面会说）。这使得它特别擅长展开扭曲的流形，特别是没有太多的噪音的情况。

下面是使用Scikit-learn中的LocallyLinearEmbedding类来对Swiss roll数据降维。

```
#生成Swiss roll数据
from sklearn.datasets import make_swiss_roll
data=make_swiss_roll(n_samples=1000, noise=0.0, random_state=None)
X=data[0]
y=data[1]
#画3维图
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
ax = plt.subplot(111, projection='3d')
```

```
ax.scatter(X[:,0], X[:,1], X[:,2],c=y)
plt.show()
#LLe降维
from sklearn.manifold import LocallyLinearEmbedding
lle = LocallyLinearEmbedding(n_components=2, n_neighbors=10)
#画出降为图
X_reduced = lle.fit_transform(X)
plt.scatter(X_reduced[:,0],X_reduced[:,1],c=y)
```



可以看到，Swiss roll展开后局部保持的距离比较好。然而，但是对大的整体距离并不是保持的很好（右下角部分被挤压，左上角伸展）。但是LLE算法还是在流形中表现的很好。

LLE原理

第一步：对于每一个训练样本$\mathrm{x}^{(i)}$，寻找离它最近的k个样本（如k=10），然后尝试重建$\mathrm{x}^{(i)}$与这些邻居的线性关系，即用这些邻居来线性表示$\mathrm{x}^{(i)}$。如下公式，最小化$\mathrm{x}^{(i)}$与这个线性表示的距离。其中非邻居的$w_{i,j}$=0。

$$\hat{W} = \arg\min_{W} \sum_{i=1}^{m} ||\mathrm{x}^{(i)} - \sum_{j=1}^{m} w_{i,j} x^{(j)}||^2$$

$$w_{i,j} = 0 \quad if \quad i,j \ is \ not \ neighbor$$

$$\sum_{j=1}^{m} w_{i,j} x^{(j)} = 1$$

第二步：将训练集降为d维(d < n)，由于上一步已经求出局部特征矩阵$\hat{W}$，因此在d维空间也要尽可能符合这个矩阵$\hat{W}$，假设样本$\mathrm{x}^{(i)}$降维变为$\mathrm{z}^{(i)}$，如下公式：

$$\hat{Z} = \arg\min_{Z} \sum_{i=1}^{m} ||z^{(i)} - \sum_{j=1}^{m} \hat{w}_{i,j} z^{(j)}||^2$$

计算k个邻居的计算复杂度为：$O(m\log(m)n\log(k))$；优化权值的复杂度为：$O(mnk^3)$；重建低维向量的复杂度为：$O(dm^2)$。如果训练样本数量过大，那么在最后一步重建m^2会很慢。

6、其它降维技术

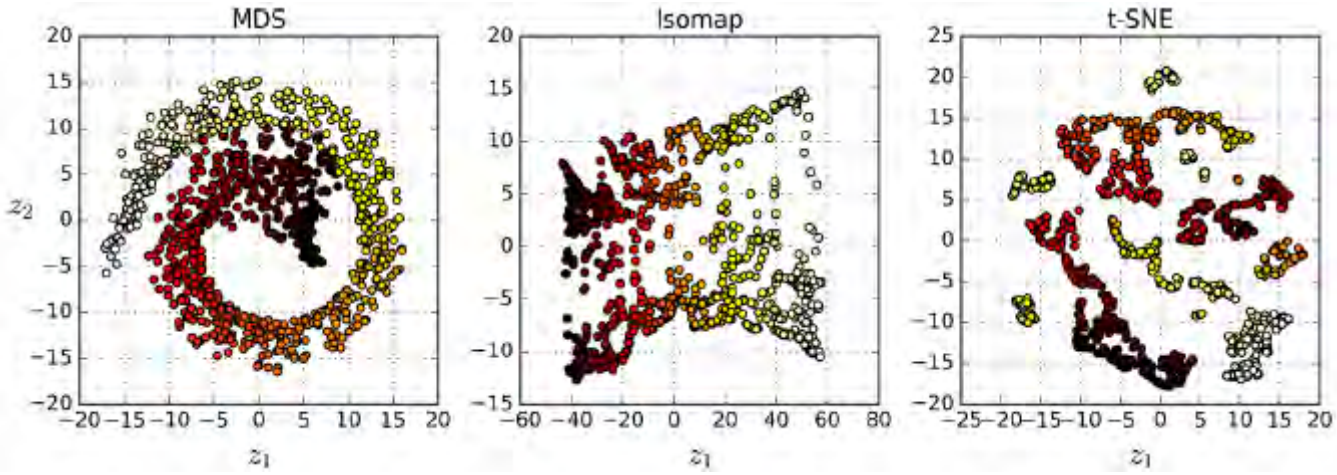
还有非常多降维的技术，有一些Scikit-learn也提供支持。下面简单列举一些比较有名的算法。

1、Multidimensional Scaling (MDS)降维的同时保留样本之间的距离，如下图。

2、Isomap通过连接每个样本和它的最近邻居来创建一个图，然后降低维的同时尝试保留样本间的测地距离(两个样本之间最少经过几个点)。

3、t-Distributed Stochastic Neighbor Embedding (t-SNE)，减少维度的同时试图保持相似的样本靠近和不同的样本分离。它主要用于可视化，特别是可视化高维空间中的聚类。

4、Linear Discriminant Analysis (LDA)，是一种分类算法，但是在训练定义了一个超平面来投影数据。投影使得同一类的样本靠近，不同一类的样本分开，所以在运行另一分类算法（如SVM分类器）之前，LDA是一种很好的减少维数的技术。



九、Up and Running with T ensorFlow

本篇文章是个人翻译的,如有商业用途,请通知本人谢谢.

创造第一个图谱,然后运行它

```
import tensorflow as tf
x = tf.Variable(3, name="x")
y = tf.Variable(4, name="y")
f = x*x*y + y + 2
```

这就是它的一切！ 最重要的是要知道这个代码实际上并不执行任何计算，即使它看起来像(尤其是最后一行）。 它只是创建一个计算图谱。 事实上，变量都没有初始化.要评估此图，您需要打开一个TensorFlow会话并使用它初始化变量并评估f。 TensorFlow会话负责处理在诸如CPU和GPU之类的设备上的操作并运行它们，并且它保留所有变量值.以下代码创建一个会话，初始化变量，并评估，然后f关闭会话（释放资源）：

```
# way1
sess = tf.Session()
sess.run(x.initializer)
sess.run(y.initializer)
result = sess.run(f)

print(result)
sess.close()
```

不得不每次重复sess.run（）有点麻烦，但幸运的是有一个更好的方法：

```
#way2
with tf.Session() as sess:
    x.initializer.run()
    y.initializer.run()
    result = f.eval()
print(result)
```

在with块中，会话被设置为默认会话。 调用x.initializer.run()等效于调用tf.get_default_session().run（x.initial）, f.eval（）等效于调用tf.get_default_session()。运行（F）。 这使得代码更容易阅读。 此外，会话在块的末尾自动关闭。

你可以使用global_variables_initializer（）函数,而不是手动初始化每个变量。 请注意，它实际上没有立即执行初始化，而是在图谱中创建一个当程序运行时所有变量都会初始化的节点：

```
#way3
# init = tf.global_variables_initializer()
# with tf.Session() as sess:
#     init.run()
#     result = f.eval()
# print(result)
```

在Jupyter内部或在Python shell中，您可能更喜欢创建一个InteractiveSession。 与常规会话的唯一区别是，当创建InteractiveSession时，它将自动将其自身设置为默认会话，因此您不需要使用模块（但是您需要在完成后手动关闭会话）：

```
#way4
init = tf.global_variables_initializer()
sess = tf.InteractiveSession()
init.run()
result = f.eval()
print(result)
sess.close()
```

TensorFlow程序通常分为两部分：第一部分构建计算图谱（这称为构造阶段），第二部分运行它（这是执行阶段）。 建设阶段通常构建一个表示ML模型的计算图谱,然后对其进行训练,计算。 执行阶段通常运行循环，重复地评估训练步骤（例如，每个mini-batch），逐渐改进模型参数。

管理图谱

您创建的任何节点都会自动添加到默认图形中：

```
>>> x1 = tf.Variable(1)
>>> x1.graph is tf.get_default_graph()
True
```

在大多数情况下，这是很好的，但有时您可能需要管理多个独立图形。 您可以通过创建一个新的图形并暂时将其设置为一个块中的默认图形，如下所示：

```
>>> graph = tf.Graph()
>>> with graph.as_default():
... x2 = tf.Variable(2)
```

```
...
>>> x2.graph is graph
True
>>> x2.graph is tf.get_default_graph()
False
```

在Jupyter（或Python shell）中，通常在实验时多次运行相同的命令。 因此，您可能会收到包含许多重复节点的默认图形。 一个解决方案是重新启动Jupyter内核（或Python shell），但是一个更方便的解决方案是通过运行tf.reset_default_graph（）来重置默认图。

节点值的生命周期

评估节点时，TensorFlow会自动确定所依赖的节点集，并首先评估这些节点。 例如，考虑以下代码：

```
# w = tf.constant(3)
# x = w + 2
# y = x + 5
# z = x * 3

# with tf.Session() as sess:
#     print(y.eval())
#     print(z.eval())
```

首先，这个代码定义了一个非常简单的图。然后，它启动一个会话并运行图来评估y：TensorFlow自动检测到y取决于w，这取决于x，所以它首先评估w，然后x，然后y，并返回y的值。最后，代码运行图来评估z。再次，TensorFlow检测到它必须首先评估w和x。重要的是要注意，它不会重用以前的w和x的评估结果。简而言之，前面的代码评估w和x两次。

所有节点值都在图运行之间删除，除了变量值，由会话跨图形运行维护（队列和读者也保持一些状态）。变量在其初始化程序运行时启动其生命周期，并且在会话关闭时结束。

如果要有效地评估y和z，而不像之前的代码那样评估w和x两次，那么您必须要求TensorFlow在一个图形运行中评估y和z，如下面的代码所示：

```
# with tf.Session() as sess:
#     y_val, z_val = sess.run([y, z])
#     print(y_val) # 10
#     print(z_val) # 15
```

在单进程TensorFlow中，多个会话不共享任何状态，即使它们重用同一个图（每个会话都有自己的每个变量的副本）。 在分布式TensorFlow中，变量状态存储在服务器上，而不是在会话中，因此多个会话可以共享相同的变量。

Linear Regression with T　ensorFlow

TensorFlow操作（也称为ops）可以采用任意数量的输入并产生任意数量的输出。 例如，加法运算和乘法运算都需要两个输入并产生一个输出。 常量和变量不输入（它们被称为源操作）。 输入和输出是称为张量的多维数组（因此称为“tensor flow”）。 就像NumPy数组一样，张量具有类型和形状。 实际上，在Python API中，张量简单地由NumPy ndarrays表示。 它们通常包含浮点数，但您也可以使用它们来传送字符串（任意字节数组）。

迄今为止的示例，张量只包含单个标量值，但是当然可以对任何形状的数组执行计算。例如，以下代码操作二维数组来对加利福尼亚房屋数据集进行线性回归（在第2章中介绍）。它从获取数据集开始;那么它会向所有训练实例添加一个额外的偏置输入特征（x0 = 1）（它使用NumPy进行，因此立即运行）;那么它创建两个TensorFlow常数节点X和y来保存该数据和目标，并且它使用TensorFlow提供的一些矩阵运算来定义theta。这些矩阵函数transpose（），matmul（）和matrix_inverse（） 是不言自明的，但是像往常一样，它们不会立即执行任何计算;相反，它们会在图形中创建在运行图形时执行它们的节点。您可以认识到θ的定义对应于方程$(\hat{\theta} = \mathbf{X}^T \cdot \mathbf{X})^{-1} \cdot \mathbf{X}^T \cdot \mathbf{y}$。

最后，代码创建asession并使用它来评估theta。

```
import numpy as np
from sklearn.datasets import fetch_california_housing
housing = fetch_california_housing()
m, n = housing.data.shape
#np.c_按colunm来组合array
housing_data_plus_bias = np.c_[np.ones((m, 1)), housing.data]

X = tf.constant(housing_data_plus_bias, dtype=tf.float32, name="X")
y = tf.constant(housing.target.reshape(-1, 1), dtype=tf.float32, name="y")
XT = tf.transpose(X)
theta = tf.matmul(tf.matmul(tf.matrix_inverse(tf.matmul(XT, X)), XT), y)
with tf.Session() as sess:
    theta_value = theta.eval()
print(theta_value)
```

该代码的主要优点是直接使用NumPy计算正态方程式，如果您有一个（如果您安装了支持GPU的TensorFlow，则TensorFlow将自动运行在GPU卡上），请参阅第12章了解更多详细信息）

其实这里就是用最小二乘法算θ

http://blog.csdn.net/akon_wang_hkbu/article/details/77503725

实现梯度下降

让我们尝试使用批量梯度下降（在第4章中介绍），而不是普通方程。 首先，我们将通过手动计算梯度来实现，然后我们将使用TensorFlow的自动扩展功能来使TensorFlow自动计算梯度，最后我们将使用几个TensorFlow的优化器。

当使用梯度下降时，请记住，首先要对输入特征向量进行归一化，否则训练可能要慢得多。 您可以使用TensorFlow，NumPy，ScikitLearn的StandardScaler或您喜欢的任何其他解决方案。 以下代码假定此规范化已经完成。

手动计算渐变

以下代码应该是相当不言自明的，除了几个新元素：

•random_uniform（）函数在图形中创建一个节点，它将生成包含随机值的张量，给定其形状和值范围，就像NumPy的rand（）函数一样。

•assign（）函数创建一个将为变量分配一个新值的节点。 在这种情况下，它实现了批次梯度下降步骤θ(next step)=θ-η∇θMSE（θ）。

•主循环一次又一次（n_epochs次）执行训练步骤，每100次迭代都打印出当前均方误差（mse）。 你应该看到MSE在每次迭代中都会下降。

```
housing = fetch_california_housing()
m, n = housing.data.shape
m, n = housing.data.shape
#np.c_按colunm来组合array
housing_data_plus_bias = np.c_[np.ones((m, 1)), housing.data]
scaled_housing_data_plus_bias = scale(housing_data_plus_bias)
n_epochs = 1000
learning_rate = 0.01
X = tf.constant(scaled_housing_data_plus_bias, dtype=tf.float32, name="X")
y = tf.constant(housing.target.reshape(-1, 1), dtype=tf.float32, name="y")
theta = tf.Variable(tf.random_uniform([n + 1, 1], -1.0, 1.0), name="theta")
y_pred = tf.matmul(X, theta, name="predictions")
error = y_pred - y
mse = tf.reduce_mean(tf.square(error), name="mse")
gradients = 2/m * tf.matmul(tf.transpose(X), error)
training_op = tf.assign(theta, theta - learning_rate * gradients)
init = tf.global_variables_initializer()
with tf.Session() as sess:
    sess.run(init)
    for epoch in range(n_epochs):
        if epoch % 100 == 0:
            print("Epoch", epoch, "MSE =", mse.eval())
        sess.run(training_op)
    best_theta = theta.eval()
```

Using autodiff

上述代码运行的不错，但它需要从数学上损失成本函数（MSE）的梯度。 在线性回归的情况下，这是相当容易的，但是如果你必须用深层神经网络来做这个事情，你会感到头痛：这将是乏味和容易出错的。 您可以使用symbolic di‰erentiation来为您自动找到偏导数的方程式，但结果代码不一定非常有效。

为了理解为什么，考虑函数f（x）= exp（exp（exp（x））） 。 如果你知道微积分，你可以计算出它的导数f'（x）= exp（x）×exp（exp（x））×exp（exp（exp（x））） 。如果您按照普通的计算方式分别去写f（x）和f'（x），您的代码将不会如此有效。 一个更有效的解决方案是写一个首先计算exp（x），然后exp（exp（x）），然后exp（exp（exp（x））） 的函数，并返回所有三个。这给你直接（第三项）f（x），如果你需要派生，你可以把这三个子式相乘，你就完成了。 通过传统的方法，您不得不将exp函数调用9次来计算f（x）和f'（x）。 使用这种方法，你只需要调用它三次。

当您的功能由某些任意代码定义时，它会变得更糟。 你可以找到方程（或代码）来计算以下函数的偏导数?

提示：不要尝试。

```
def my_func(a, b):
    z = 0
    for i in range(100):
    z = a * np.cos(z + i) + z * np.sin(b - i)
    return z
```

幸运的是，TensorFlow的自动计算梯度功能可以计算这个公式：它可以自动高效地为您梯度的计算。 只需用以下一行替换上一节中代码的gradients = ... 行，代码将继续工作正常：

```
gradients = tf.gradients(mse, [theta])[0]
```

gradients（）函数使用一个op（在这种情况下是mse）和一个变量列表（在这种情况下只是theta），它创建一个ops列表（每个变量一个）来计算op的梯度变量。 因此，梯度节点将计算MSE相对于theta的梯度向量。

自动计算梯度有四种主要方法。 它们总结在表9-2中。 TensorFlow使用反向模式，这是完美的（高效和准确），当有很多输入和少量的输出，如通常在神经网络的情况。 它只是在n(outputs) + 1图遍历中计算所有输出的偏导数。

Table 9-2. Main solutions to compute gradients automatically

Technique	Nb of graph traversals to compute all gradients	Accuracy	Supports arbitrary code	Comment
Numerical differentiation	$n_{inputs} + 1$	Low	Yes	Trivial to implement
Symbolic differentiation	N/A	High	No	Builds a very different graph
Forward-mode autodiff	n_{inputs}	High	Yes	Uses *dual numbers*
Reverse-mode autodiff	$n_{outputs} + 1$	High	Yes	Implemented by TensorFlow

使用优化器

所以TensorFlow为您计算梯度。 但它还有更好的方法：它还提供了一些可以直接使用的优化器，包括梯度下降优化器。您可以使用以下代码简单地替换以前的gradients = ... 和training_op = ...行，并且一切都将正常工作：

```
optimizer = tf.train.GradientDescentOptimizer(learning_rate=learning_rate)
training_op = optimizer.minimize(mse)
```

如果要使用其他类型的优化器，则只需要更改一行。 例如，您可以通过定义优化器来使用动量优化器（通常会比渐变渐变收敛得快得多;参见第11章）

```
optimizer = tf.train.MomentumOptimizer(learning_rate=learning_rate,
                                       momentum=0.9)
```

将数据提供给训练算法

我们尝试修改以前的代码来实现小批量梯度(Mini-batch Gradient Descent)下降。 为此，我们需要一种在每次迭代时用下一个小批量替换X和Y的方法。 最简单的方法是使用占位符节点(placeholder)。 这些节点是特别的，因为它们实际上并不执行任何计算，只是输出您在运行时输出的数据。 它们通常用于在培训期间将训练数据传递TensorFlow。 如果在运行时没有为占位符指定值，则会收到异常。

要创建占位符节点，您必须调用placeholder() 函数并指定输出张量的数据类型。 或者，您还可以指定其形状，如果要强制执行。 如果指定维度为“None”，则表示“任何大小”。例如，以下代码创建一个占位符节点A，还有一个节点B = A + 5.当我们评估B时，我们将一个feed_dict传递给eval ()方法指定A的值。注意，A必须具有2级（即它必须是二维的），并且必须有三列（否则引发异常），但它可以有任意数量的行。

```
>>> A = tf.placeholder(tf.float32, shape=(None, 3))
>>> B = A + 5
>>> with tf.Session() as sess:
... B_val_1 = B.eval(feed_dict={A: [[1, 2, 3]]})
... B_val_2 = B.eval(feed_dict={A: [[4, 5, 6], [7, 8, 9]]})
...
>>> print(B_val_1)
[[ 6. 7. 8.]]
>>> print(B_val_2)
[[ 9. 10. 11.]
[ 12. 13. 14.]]
```

您实际上可以提供任何操作的输出，而不仅仅是占位符。 在这种情况下，TensorFlow不会尝试评估这些操作;它使用您提供的值。

要实现小批量渐变下降，我们只需稍微调整现有的代码。 首先更改X和Y的定义，使其占位符节点：

```
X = tf.placeholder(tf.float32, shape=(None, n + 1), name="X")
y = tf.placeholder(tf.float32, shape=(None, 1), name="y")
```

然后定义批量大小并计算总批次数：

```
batch_size = 100
n_batches = int(np.ceil(m / batch_size))
```

最后，在执行阶段，逐个获取小批量，然后在评估依赖于X和y的值的任何一个节点时,通过feed_dict提供X和y的值。

```
def fetch_batch(epoch, batch_index, batch_size):
    [...] # load the data from disk
    return X_batch, y_batch

with tf.Session() as sess:
    sess.run(init)

    for epoch in range(n_epochs):
        for batch_index in range(n_batches):
            X_batch, y_batch = fetch_batch(epoch, batch_index, batch_size)
            sess.run(training_op, feed_dict={X: X_batch, y: y_batch})
    best_theta = theta.eval()
```

在评估theta时，我们不需要传递X和y的值，因为它不依赖于它们。

MINI-BATCH完整代码

```
import numpy as np
from sklearn.datasets import fetch_california_housing
import tensorflow as tf
from sklearn.preprocessing import StandardScaler

housing = fetch_california_housing()
m, n = housing.data.shape
```

```
print("数据集:{}行,{}列".format(m,n))
housing_data_plus_bias = np.c_[np.ones((m, 1)), housing.data]
scaler = StandardScaler()
scaled_housing_data = scaler.fit_transform(housing.data)
scaled_housing_data_plus_bias = np.c_[np.ones((m, 1)), scaled_housing_data]

n_epochs = 1000
learning_rate = 0.01

X = tf.placeholder(tf.float32, shape=(None, n + 1), name="X")
y = tf.placeholder(tf.float32, shape=(None, 1), name="y")
theta = tf.Variable(tf.random_uniform([n + 1, 1], -1.0, 1.0, seed=42), name="theta")
y_pred = tf.matmul(X, theta, name="predictions")
error = y_pred - y
mse = tf.reduce_mean(tf.square(error), name="mse")
optimizer = tf.train.GradientDescentOptimizer(learning_rate=learning_rate)
training_op = optimizer.minimize(mse)

init = tf.global_variables_initializer()

n_epochs = 10
batch_size = 100
n_batches = int(np.ceil(m / batch_size)) #ceil()方法返回x的值上限 - 不小于x的最小整数。

def fetch_batch(epoch, batch_index, batch_size):
    know = np.random.seed(epoch * n_batches + batch_index)  # not shown in the book
    print("我是know:",know)
    indices = np.random.randint(m, size=batch_size)  # not shown
    X_batch = scaled_housing_data_plus_bias[indices] # not shown
    y_batch = housing.target.reshape(-1, 1)[indices] # not shown
    return X_batch, y_batch

with tf.Session() as sess:
    sess.run(init)

    for epoch in range(n_epochs):
        for batch_index in range(n_batches):
            X_batch, y_batch = fetch_batch(epoch, batch_index, batch_size)
            sess.run(training_op, feed_dict={X: X_batch, y: y_batch})

    best_theta = theta.eval()

print(best_theta)
```

存储和回复模型

一旦你训练了你的模型，你应该把它的参数保存到磁盘，所以你可以随时随地回到它，在另一个程序中使用它，与其他模型比较，等等。 此外，您可能希望在培训期间定期保存检查点，以便如果您的计算机在训练过程中崩溃，您可以从上次检查点继续进行，而不是从头开始。

TensorFlow可以轻松保存和恢复模型。 只需在构造阶段结束（创建所有变量节点之后）创建一个Save节点; 那么在执行阶段，只要你想保存模型，只要调用它的save（）方法:

```
[...]
theta = tf.Variable(tf.random_uniform([n + 1, 1], -1.0, 1.0), name="theta")
[...]
init = tf.global_variables_initializer()
saver = tf.train.Saver()
with tf.Session() as sess:
    sess.run(init)

    for epoch in range(n_epochs):
        if epoch % 100 == 0: # checkpoint every 100 epochs
            save_path = saver.save(sess, "/tmp/my_model.ckpt")

        sess.run(training_op)
    best_theta = theta.eval()
    save_path = saver.save(sess, "/tmp/my_model_final.ckpt")
```

恢复模型同样容易：在构建阶段结束时创建一个Saver，就像之前一样，但是在执行阶段的开始，而不是使用init节点初始化变量，你可以调用restore（）方法 的Saver对象：

```
with tf.Session() as sess:
    saver.restore(sess, "/tmp/my_model_final.ckpt")
    [...]
```

默认情况下，Saver将以自己的名称保存并还原所有变量，但如果需要更多控制，则可以指定要保存或还原的变量以及要使用的名称。 例如，以下Saver将仅保存或恢复名称权重下的theta变量：

```
saver = tf.train.Saver({"weights": theta})
```

完整代码

```
 numpy as np
from sklearn.datasets import fetch_california_housing
import tensorflow as tf
from sklearn.preprocessing import StandardScaler

housing = fetch_california_housing()
m, n = housing.data.shape
print("数据集:{}行,{}列".format(m,n))
housing_data_plus_bias = np.c_[np.ones((m, 1)), housing.data]
scaler = StandardScaler()
scaled_housing_data = scaler.fit_transform(housing.data)
scaled_housing_data_plus_bias = np.c_[np.ones((m, 1)), scaled_housing_data]

n_epochs = 1000  # not shown in the book
learning_rate = 0.01  # not shown

X = tf.constant(scaled_housing_data_plus_bias, dtype=tf.float32, name="X")  # not shown
y = tf.constant(housing.target.reshape(-1, 1), dtype=tf.float32, name="y")  # not shown
theta = tf.Variable(tf.random_uniform([n + 1, 1], -1.0, 1.0, seed=42), name="theta")
y_pred = tf.matmul(X, theta, name="predictions")  # not shown
error = y_pred - y  # not shown
mse = tf.reduce_mean(tf.square(error), name="mse")  # not shown
```

```
optimizer = tf.train.GradientDescentOptimizer(learning_rate=learning_rate)  # not shown
training_op = optimizer.minimize(mse)  # not shown

init = tf.global_variables_initializer()
saver = tf.train.Saver()

with tf.Session() as sess:
    sess.run(init)

    for epoch in range(n_epochs):
        if epoch % 100 == 0:
            print("Epoch", epoch, "MSE =", mse.eval())  # not shown
            save_path = saver.save(sess, "/tmp/my_model.ckpt")
        sess.run(training_op)

    best_theta = theta.eval()
    save_path = saver.save(sess, "/tmp/my_model_final.ckpt") #找到tmp文件夹就找到文件了
```

使用TensorBoard展现图形和训练曲线

所以现在我们有一个使用Mini_batch 梯度下降训练线性回归模型的计算图谱，我们正在定期保存检查点。 听起来很复杂，不是吗？ 然而，我们仍然依靠print（）函数可视化训练过程中的进度。 有一个更好的方法：进入TensorBoard。 如果您提供一些训练统计信息，它将在您的网络浏览器中显示这些统计信息的良好交互式可视化（例如学习曲线）。 您还可以提供图形的定义，它将为您提供一个很好的界面来浏览它。 这对于识别图中的错误，找到瓶颈等是非常有用的。

第一步是调整程序，以便将图形定义和一些训练统计信息（例如，training_error（MSE））写入TensorBoard将读取的日志目录。 您每次运行程序时都需要使用不同的日志目录，否则TensorBoard将会合并来自不同运行的统计信息，这将会混乱可视化。 最简单的解决方案是在日志目录名称中包含时间戳。 在程序开头添加以下代码：

```
from datetime import datetime
now = datetime.utcnow().strftime("%Y%m%d%H%M%S")
root_logdir = "tf_logs"
logdir = "{}/run-{}/".format(root_logdir, now)
```

接下来，在建设阶段结束时添加以下代码：

```
mse_summary = tf.summary.scalar('MSE', mse)
file_writer = tf.summary.FileWriter(logdir, tf.get_default_graph())
```

第一行创建一个将评估MSE值并将其写入TensorBoard兼容的二进制日志字符串（称为摘要）中的节点。 第二行创建一个FileWriter，您将用于将日志文件的摘要写入日志目录中。 第一个参数指示日志目录的路径（在本例中为tf_logs / run-20160906091959 /，相对于当前目录）。 第二个（可选）参数是您想要可视化的图形。 创建时，文件写入器创建日志目录（如果需要），并将其定义在二进制日志文件（称为事件）中。

接下来，您需要更新执行阶段，以便在训练期间定期评估mse_summary节点（例如，每10个小批量）。 这将输出一个摘要，然后可以使用file_writer写入事件文件。 以下是更新的代码：

```
[...]
for batch_index in range(n_batches):
    X_batch, y_batch = fetch_batch(epoch, batch_index, batch_size)
```

```
    if batch_index % 10 == 0:
        summary_str = mse_summary.eval(feed_dict={X: X_batch, y: y_batch})
        step = epoch * n_batches + batch_index
        file_writer.add_summary(summary_str, step)
    sess.run(training_op, feed_dict={X: X_batch, y: y_batch})
[...]
```

避免在每一个培训阶段记录培训数据，因为这会大大减慢培训速度.

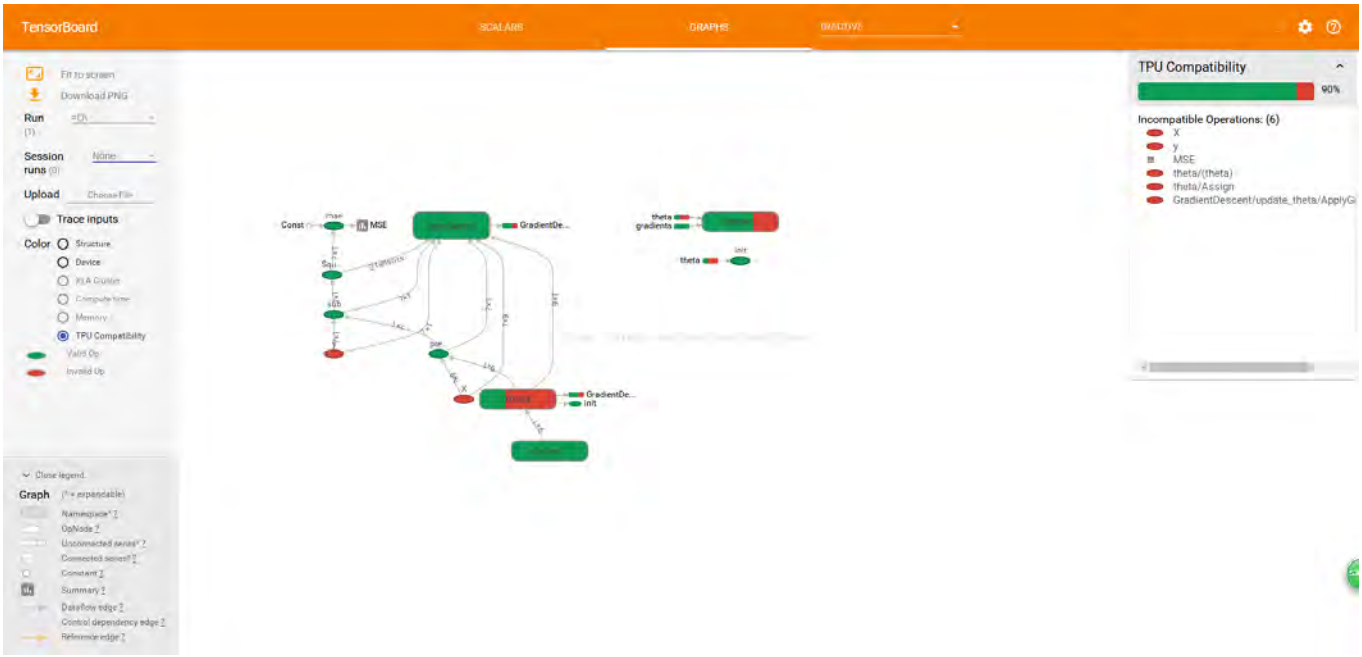
最后，要在程序结束时关闭FileWriter：

```
file_writer.close()
```

完整代码

```
import numpy as np
from sklearn.datasets import fetch_california_housing
import tensorflow as tf
from sklearn.preprocessing import StandardScaler

housing = fetch_california_housing()
m, n = housing.data.shape
print("数据集:{}行,{}列".format(m,n))
housing_data_plus_bias = np.c_[np.ones((m, 1)), housing.data]
scaler = StandardScaler()
scaled_housing_data = scaler.fit_transform(housing.data)
scaled_housing_data_plus_bias = np.c_[np.ones((m, 1)), scaled_housing_data]

from datetime import datetime

now = datetime.utcnow().strftime("%Y%m%d%H%M%S")
root_logdir = r"D://tf_logs"
logdir = "{}/run-{}/".format(root_logdir, now)
n_epochs = 1000
learning_rate = 0.01

X = tf.placeholder(tf.float32, shape=(None, n + 1), name="X")
y = tf.placeholder(tf.float32, shape=(None, 1), name="y")
theta = tf.Variable(tf.random_uniform([n + 1, 1], -1.0, 1.0, seed=42), name="theta")
y_pred = tf.matmul(X, theta, name="predictions")
error = y_pred - y
mse = tf.reduce_mean(tf.square(error), name="mse")
optimizer = tf.train.GradientDescentOptimizer(learning_rate=learning_rate)
training_op = optimizer.minimize(mse)

init = tf.global_variables_initializer()
mse_summary = tf.summary.scalar('MSE', mse)
file_writer = tf.summary.FileWriter(logdir, tf.get_default_graph())

n_epochs = 10
batch_size = 100
n_batches = int(np.ceil(m / batch_size))

def fetch_batch(epoch, batch_index, batch_size):
```

```
    np.random.seed(epoch * n_batches + batch_index)  # not shown in the book
    indices = np.random.randint(m, size=batch_size)  # not shown
    X_batch = scaled_housing_data_plus_bias[indices] # not shown
    y_batch = housing.target.reshape(-1, 1)[indices] # not shown
    return X_batch, y_batch

with tf.Session() as sess:                                                    # not shown
    sess.run(init)                                                            # not shown

    for epoch in range(n_epochs):                                             # not shown
        for batch_index in range(n_batches):
            X_batch, y_batch = fetch_batch(epoch, batch_index, batch_size)
            if batch_index % 10 == 0:
                summary_str = mse_summary.eval(feed_dict={X: X_batch, y: y_batch})
                step = epoch * n_batches + batch_index
                file_writer.add_summary(summary_str, step)
            sess.run(training_op, feed_dict={X: X_batch, y: y_batch})

    best_theta = theta.eval()
file_writer.close()
print(best_theta)
```



名称范围

当处理更复杂的模型（如神经网络）时，该图可以很容易地与数千个节点混淆。 为了避免这种情况，您可以创建名称范围来对相关节点进行分组。 例如，我们修改以前的代码来定义名为“loss”的名称范围内的错误和mse操作：

```
with tf.name_scope("loss") as scope:
    error = y_pred - y
    mse = tf.reduce_mean(tf.square(error), name="mse")
```

在范围内定义的每个op的名称现在以“loss /”为前缀：

```
>>> print(error.op.name)
loss/sub
>>> print(mse.op.name)
loss/mse
```

在TensorBoard中，mse和error节点现在出现在Loss命名空间中，默认情况下会出现崩溃（图9-5）。

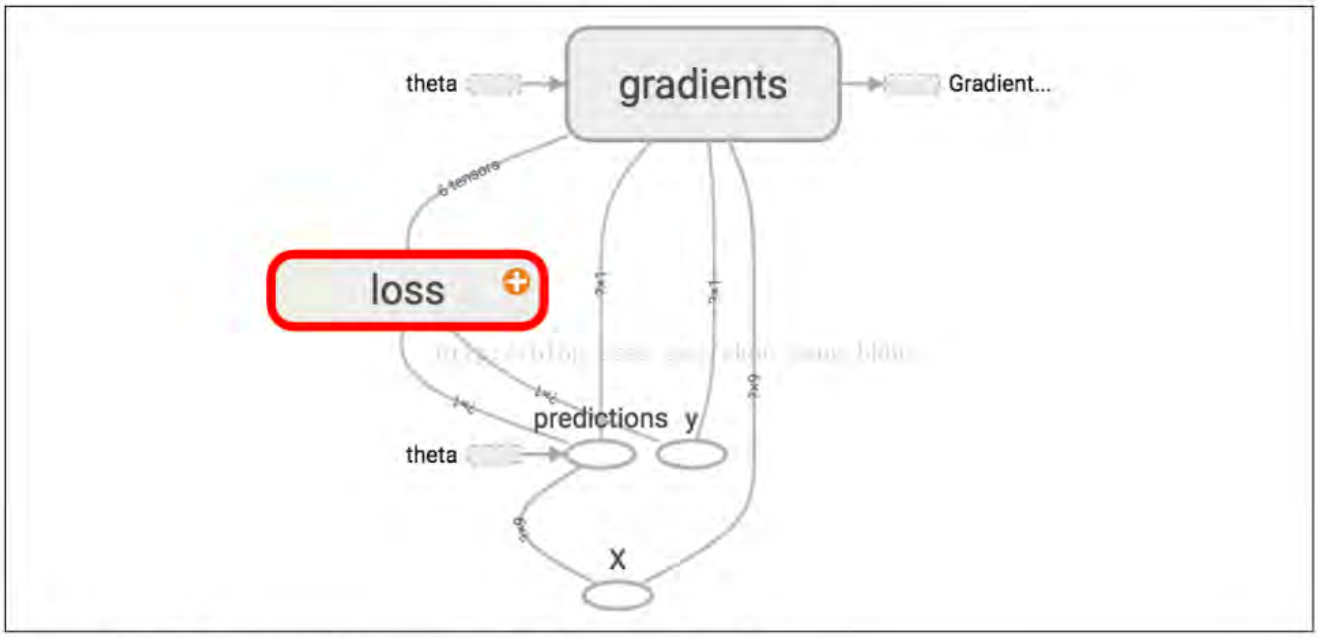
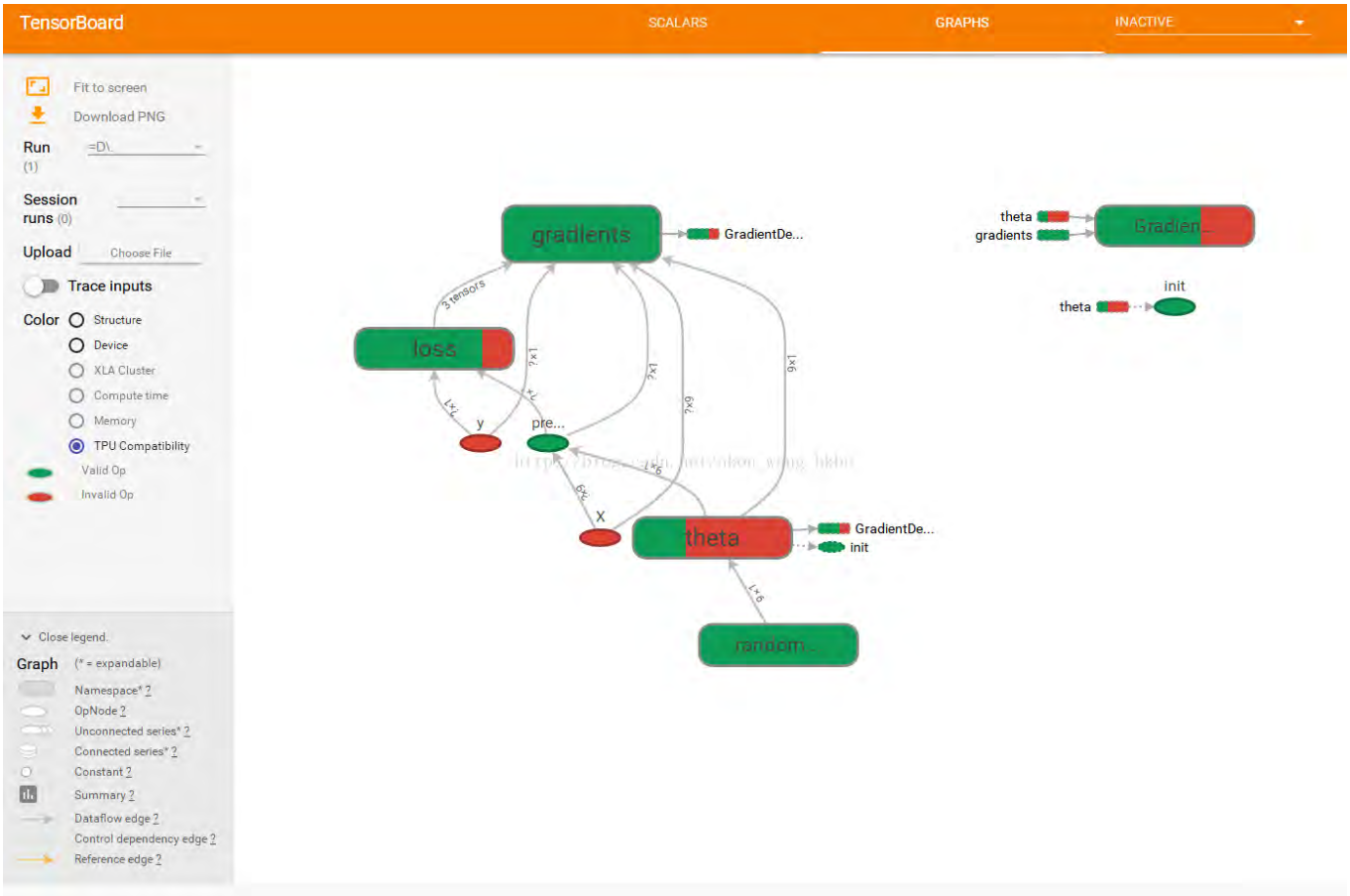


Figure 9-5. A collapsed namescope in TensorBoard

完整代码

```
import numpy as np
from sklearn.datasets import fetch_california_housing
import tensorflow as tf
from sklearn.preprocessing import StandardScaler

housing = fetch_california_housing()
m, n = housing.data.shape
print("数据集:{}行,{}列".format(m,n))
housing_data_plus_bias = np.c_[np.ones((m, 1)), housing.data]
scaler = StandardScaler()
scaled_housing_data = scaler.fit_transform(housing.data)
scaled_housing_data_plus_bias = np.c_[np.ones((m, 1)), scaled_housing_data]

from datetime import datetime

now = datetime.utcnow().strftime("%Y%m%d%H%M%S")
root_logdir = r"D://tf_logs"
logdir = "{}/run-{}/".format(root_logdir, now)

n_epochs = 1000
learning_rate = 0.01

X = tf.placeholder(tf.float32, shape=(None, n + 1), name="X")
y = tf.placeholder(tf.float32, shape=(None, 1), name="y")
theta = tf.Variable(tf.random_uniform([n + 1, 1], -1.0, 1.0, seed=42), name="theta")
```

```
y_pred = tf.matmul(X, theta, name="predictions")


def fetch_batch(epoch, batch_index, batch_size):
    np.random.seed(epoch * n_batches + batch_index)  # not shown in the book
    indices = np.random.randint(m, size=batch_size)  # not shown
    X_batch = scaled_housing_data_plus_bias[indices] # not shown
    y_batch = housing.target.reshape(-1, 1)[indices] # not shown
    return X_batch, y_batch


with tf.name_scope("loss") as scope:
    error = y_pred - y
    mse = tf.reduce_mean(tf.square(error), name="mse")


optimizer = tf.train.GradientDescentOptimizer(learning_rate=learning_rate)
training_op = optimizer.minimize(mse)

init = tf.global_variables_initializer()

mse_summary = tf.summary.scalar('MSE', mse)
file_writer = tf.summary.FileWriter(logdir, tf.get_default_graph())

n_epochs = 10
batch_size = 100
n_batches = int(np.ceil(m / batch_size))

with tf.Session() as sess:
    sess.run(init)

    for epoch in range(n_epochs):
        for batch_index in range(n_batches):
            X_batch, y_batch = fetch_batch(epoch, batch_index, batch_size)
            if batch_index % 10 == 0:
                summary_str = mse_summary.eval(feed_dict={X: X_batch, y: y_batch})
                step = epoch * n_batches + batch_index
                file_writer.add_summary(summary_str, step)
            sess.run(training_op, feed_dict={X: X_batch, y: y_batch})

    best_theta = theta.eval()

file_writer.flush()
file_writer.close()
print("Best theta:")
print(best_theta)
```



模块性

假设您要创建一个添加两个整流线性单元（ReLU）的输出的图形。 ReLU计算输入的线性函数，如果为正，则输出结果，否则为0，如等式9-1所示。

Equation 9-1. Rectified linear unit

$$h_{\mathbf{w},b}(\mathbf{X}) = \max(\mathbf{X} \cdot \mathbf{w} + b, 0)$$

下面的代码做这个工作，但是它是相当重复的：

```
n_features = 3
X = tf.placeholder(tf.float32, shape=(None, n_features), name="X")
w1 = tf.Variable(tf.random_normal((n_features, 1)), name="weights1")
w2 = tf.Variable(tf.random_normal((n_features, 1)), name="weights2")
b1 = tf.Variable(0.0, name="bias1")
b2 = tf.Variable(0.0, name="bias2")
z1 = tf.add(tf.matmul(X, w1), b1, name="z1")
z2 = tf.add(tf.matmul(X, w2), b2, name="z2")
relu1 = tf.maximum(z1, 0., name="relu1")
relu2 = tf.maximum(z1, 0., name="relu2")
output = tf.add(relu1, relu2, name="output")
```

这样的重复代码很难维护，容易出错（实际上，这个代码包含了一个剪贴错误，你发现了吗？） 如果你想添加更多的ReLU，会变得更糟。 幸运的是，TensorFlow可以让您保持DRY（不要重复自己）：只需创建一个功能

来构建ReLU。 以下代码创建五个ReLU并输出其总和（注意，add_n（）创建一个将计算张量列表的和的操作）：

```
def relu(X):
    w_shape = (int(X.get_shape()[1]), 1)
    w = tf.Variable(tf.random_normal(w_shape), name="weights")
    b = tf.Variable(0.0, name="bias")
    z = tf.add(tf.matmul(X, w), b, name="z")
    return tf.maximum(z, 0., name="relu")

n_features = 3
X = tf.placeholder(tf.float32, shape=(None, n_features), name="X")
relus = [relu(X) for i in range(5)]
output = tf.add_n(relus, name="output")
```

请注意，创建节点时，TensorFlow将检查其名称是否已存在，如果它已经存在，则会附加一个下划线，后跟一个索引，以使该名称是唯一的。 因此，第一个ReLU包含名为“权重”，“偏差”，“z”和“relu”的节点（加上其他默认名称的更多节点，如“MatMul”）；第二个ReLU包含名为“weights_1”，“bias_1”等节点的节点; 第三个ReLU包含名为 “weights_2”，“bias_2”的节点，依此类推。 TensorBoard识别这样的系列并将它们折叠在一起以减少混乱（如图9-6所示）

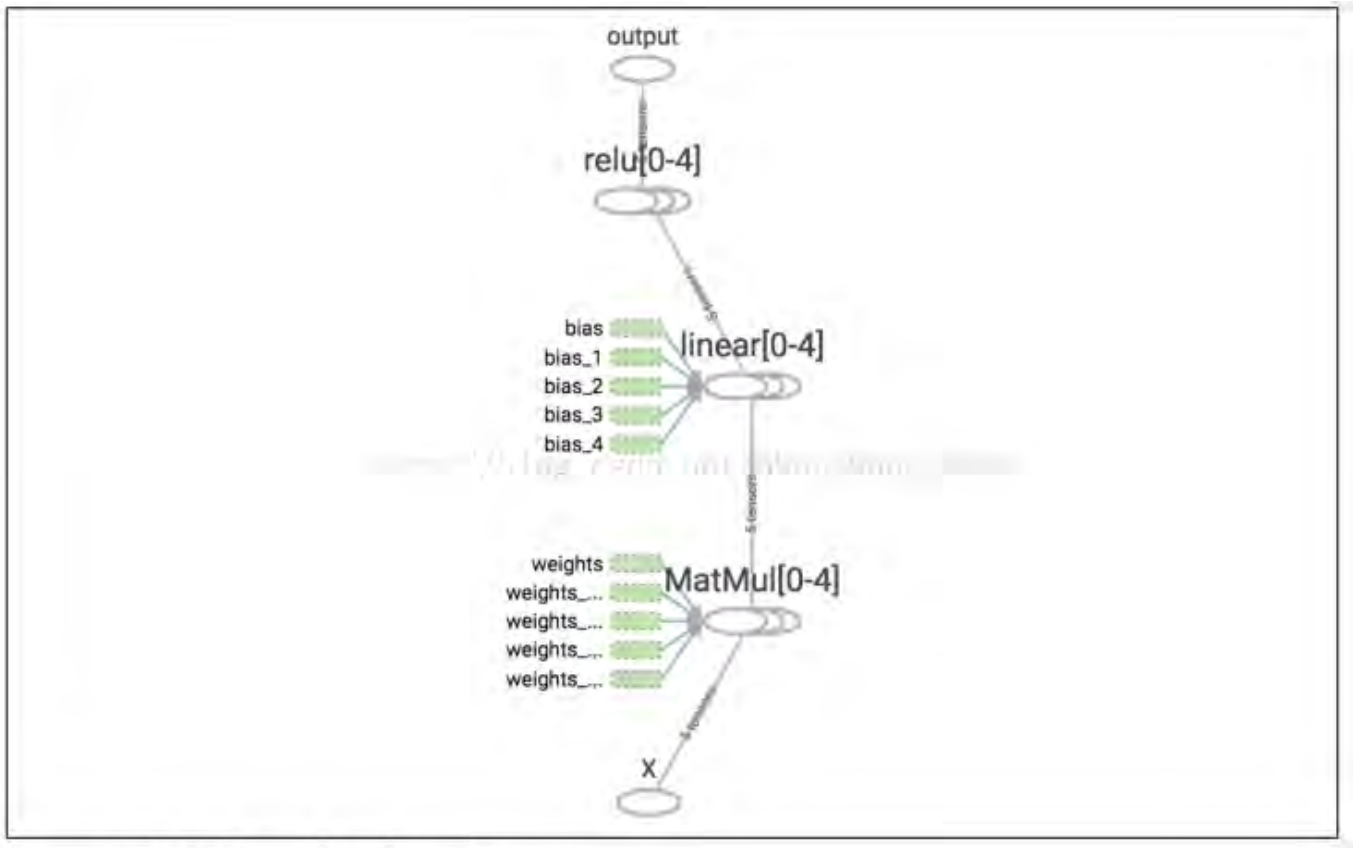


Figure 9-6. Collapsed node series

使用名称范围，您可以使图形更清晰。 简单地将relu（）函数的所有内容移动到名称范围内。 图9-7显示了结果图。 请注意，TensorFlow还通过附加_1，_2等来给出名称范围的唯一名称。

```
def relu(X):
    with tf.name_scope("relu"):
        w_shape = (int(X.get_shape()[1]), 1)                          # not shown in the book
        w = tf.Variable(tf.random_normal(w_shape), name="weights")    # not shown
```

```
        b = tf.Variable(0.0, name="bias")                                # not shown
        z = tf.add(tf.matmul(X, w), b, name="z")                         # not shown
        return tf.maximum(z, 0., name="max")                             # not shown
```

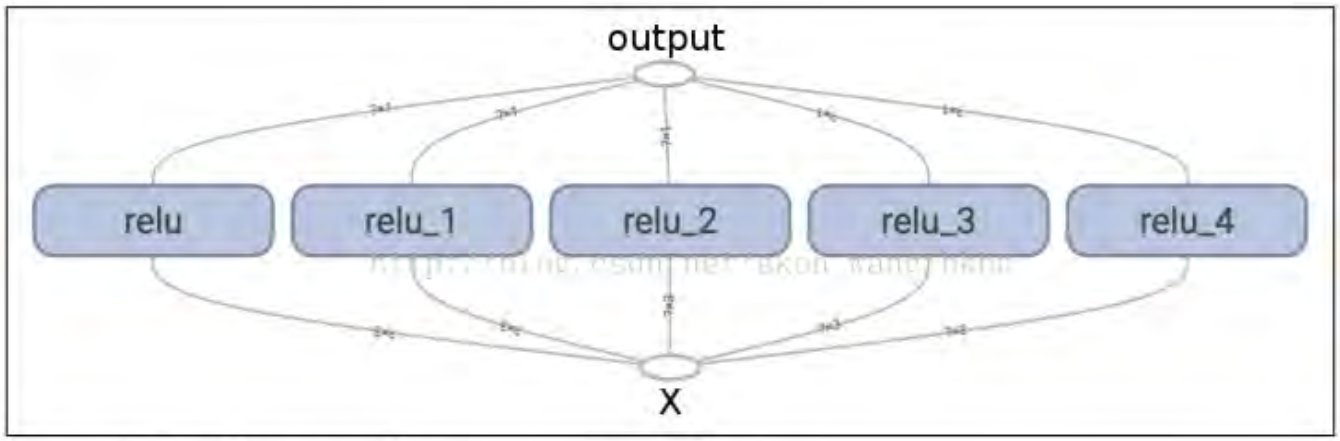


Figure 9-7. A clearer graph using name-scoped units

共享变量

如果要在图形的各个组件之间共享一个变量，一个简单的选项是首先创建它，然后将其作为参数传递给需要它的函数。 例如，假设要使用所有ReLU的共享阈值变量来控制ReLU阈值（当前硬编码为0）。 您可以先创建该变量，然后将其传递给relu（）函数：

```
reset_graph()

def relu(X, threshold):
    with tf.name_scope("relu"):
        w_shape = (int(X.get_shape()[1]), 1)                          # not shown in the book
        w = tf.Variable(tf.random_normal(w_shape), name="weights")  # not shown
        b = tf.Variable(0.0, name="bias")                             # not shown
        z = tf.add(tf.matmul(X, w), b, name="z")                      # not shown
        return tf.maximum(z, threshold, name="max")

threshold = tf.Variable(0.0, name="threshold")
X = tf.placeholder(tf.float32, shape=(None, n_features), name="X")
relus = [relu(X, threshold) for i in range(5)]
output = tf.add_n(relus, name="output")
```

这很好:现在您可以使用阈值变量来控制所有ReLUs的阈值。但是，如果有许多共享参数，比如这一项，那么必须一直将它们作为参数传递，这将是非常痛苦的。许多人创建了一个包含模型中所有变量的Python字典，并将其传递给每个函数。另一些则为每个模块创建一个类(例如:一个使用类变量来处理共享参数的ReLU类。另一种选择是在第一次调用时将共享变量设置为relu()函数的属性，如下所列:

```
def relu(X):
    with tf.name_scope("relu"):
        if not hasattr(relu, "threshold"):
            relu.threshold = tf.Variable(0.0, name="threshold")
        w_shape = int(X.get_shape()[1]), 1                            # not shown in the book
        w = tf.Variable(tf.random_normal(w_shape), name="weights")  # not shown
        b = tf.Variable(0.0, name="bias")                             # not shown
```

```
        z = tf.add(tf.matmul(X, w), b, name="z")                    # not shown
        return tf.maximum(z, relu.threshold, name="max")
```

TensorFlow提供了另一个选项，这可能会导致比以前的解决方案稍微更清洁和更模块化的代码.5这个解决方案首先要明白一点，但是由于它在TensorFlow中使用了很多，值得深入细节。 这个想法是使用get_variable（）函数来创建共享变量，如果它还不存在，或者如果已经存在，则重用它。 所需的行为（创建或重用）由当前variable_scope（）的属性控制。 例如，以下代码将创建一个名为“relu / threshold”的变量（作为标量，因为shape =（），并使用0.0作为初始值）：

```
with tf.variable_scope("relu"):
    threshold = tf.get_variable("threshold", shape=(),
                                initializer=tf.constant_initializer(0.0))
```

请注意，如果变量已经通过较早的get_variable（）调用创建，则此代码将引发异常。 这种行为可以防止错误地重用变量。 如果要重用变量，则需要通过将变量scope的重用属性设置为True来明确说明（在这种情况下，您不必指定形状或初始值）：

```
with tf.variable_scope("relu", reuse=True):
    threshold = tf.get_variable("threshold")
```

该代码将获取现有的“relu / threshold”变量，如果不存在或引发异常（如果没有使用get_variable（）创建）。 或者，您可以通过调用scope的reuse_variables（）方法将重用属性设置为true：

```
with tf.variable_scope("relu") as scope:
    scope.reuse_variables()
    threshold = tf.get_variable("threshold")
```

一旦重新使用设置为True，它将不能在块内设置为False。 而且，如果在其中定义其他变量作用域，它们将自动继承reuse = True。 最后，只有通过get_variable（）创建的变量才可以这样重用.

现在，您拥有所有需要的部分，使relu（）函数访问阈值变量，而不必将其作为参数传递：

```
def relu(X):
    with tf.variable_scope("relu", reuse=True):
        threshold = tf.get_variable("threshold")
        w_shape = int(X.get_shape()[1]), 1                          # not shown
        w = tf.Variable(tf.random_normal(w_shape), name="weights")  # not shown
        b = tf.Variable(0.0, name="bias")                           # not shown
        z = tf.add(tf.matmul(X, w), b, name="z")                    # not shown
        return tf.maximum(z, threshold, name="max")

X = tf.placeholder(tf.float32, shape=(None, n_features), name="X")
with tf.variable_scope("relu"):
    threshold = tf.get_variable("threshold", shape=(),
                                initializer=tf.constant_initializer(0.0))
relus = [relu(X) for relu_index in range(5)]
output = tf.add_n(relus, name="output")
```

该代码首先定义relu（）函数，然后创建relu / threshold变量（作为标量，稍后将被初始化为0.0），并通过调用relu（）函数构建五个ReLU。 relu（）函数重用relu / threshold变量，并创建其他ReLU节点。

使用get_variable（）创建的变量始终以其variable_scope的名称作为前缀命名（例如，“relu / threshold”），但对于所有其他节点（包括使用tf.Variable（）创建的变量），变量范围的行为就像一个新名称的范围。 特别是，如果已经创建了具有相同名称的名称范围，则添加后缀以使该名称是唯一的。 例如，在前面的代码中创建的所有节点（阈值变量除外）的名称前缀为“relu_1 /”到“relu_5 /”，如图9-8所示。

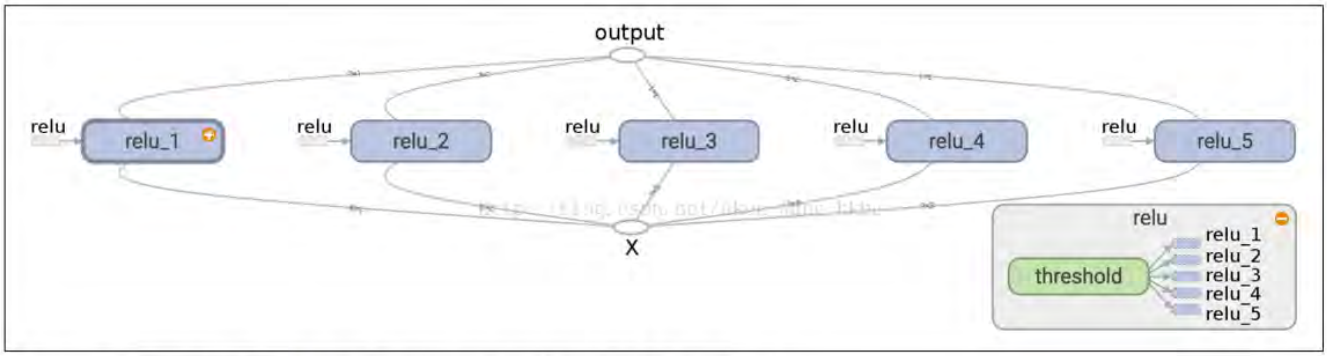


Figure 9-8. Five ReLUs sharing the threshold variable

不幸的是，必须在relu（）函数之外定义阈值变量，其中ReLU代码的其余部分都驻留在其中。 要解决此问题，以下代码在第一次调用时在relu（）函数中创建阈值变量，然后在后续调用中重新使用。 现在，relu（）函数不必担心名称范围或变量共享：它只是调用get_variable（），它将创建或重用阈值变量（它不需要知道是哪种情况）。 其余的代码调用relu（）五次，确保在第一次调用时设置reuse = False，而对于其他调用来说，reuse = True。

```
def relu(X):
    threshold = tf.get_variable("threshold", shape=(),
                                initializer=tf.constant_initializer(0.0))
    w_shape = (int(X.get_shape()[1]), 1)                        # not shown in the book
    w = tf.Variable(tf.random_normal(w_shape), name="weights")  # not shown
    b = tf.Variable(0.0, name="bias")                           # not shown
    z = tf.add(tf.matmul(X, w), b, name="z")                    # not shown
    return tf.maximum(z, threshold, name="max")

X = tf.placeholder(tf.float32, shape=(None, n_features), name="X")
relus = []
for relu_index in range(5):
    with tf.variable_scope("relu", reuse=(relu_index >= 1)) as scope:
        relus.append(relu(X))
output = tf.add_n(relus, name="output")
```

生成的图形与之前略有不同，因为共享变量存在于第一个ReLU中（见图9-9）。

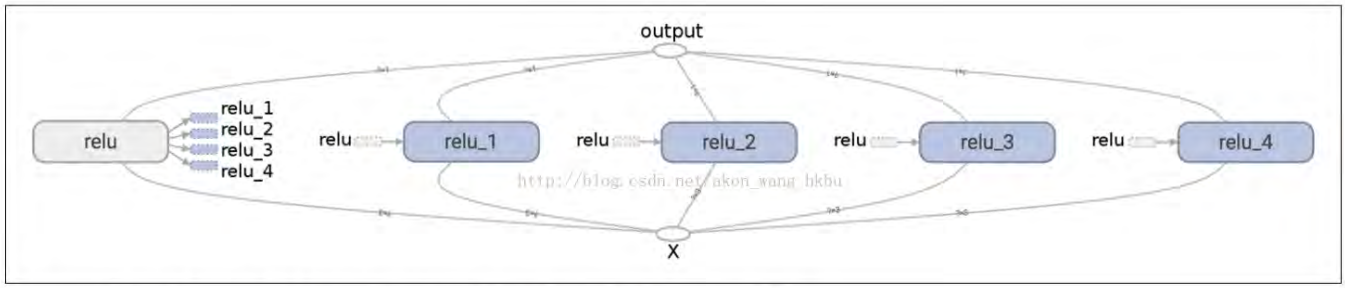


Figure 9-9. Five ReLUs sharing the threshold variable

TensorFlow的这个介绍到此结束。 我们将在以下章节中讨论更多高级课题，特别是与深层神经网络，卷积神经网络和复发神经网络相关的许多操作，以及如何使用多线程，队列，多个GPU以及TensorFlow进行扩展多台服务器。

十、Introduction to Artifcial Neural Networks

本篇文章是个人翻译的,如有商业用途,请通知本人谢谢.

使用普通TensorFlow 训练DNN

如果您想要更好地控制网络架构，您可能更喜欢使用TensorFlow的较低级别的Python API（在第9章中介绍）。 在本节中，我们将使用与之前的API相同的模型，我们将实施Minibatch 梯度下降来在MNIST数据集上进行训练。 第一步是建设阶段，构建TensorFlow图。 第二步是执行阶段，您实际运行计算图谱来训练模型。

Construction Phase

开始吧。 首先我们需要导入tensorflow库。 然后我们必须指定输入和输出的数量，并设置每个层中隐藏的神经元数量：

```
import tensorflow as tf
n_inputs = 28*28 # MNIST
n_hidden1 = 300
n_hidden2 = 100
n_outputs = 10
```

接下来，与第9章一样，您可以使用占位符节点来表示训练数据和目标。 X的形状仅有部分被定义。 我们知道它将是一个2D张量（即一个矩阵），沿着第一个维度的实例和第二个维度的特征，我们知道特征的数量将是28×28（每像素一个特征） 但是我们不知道每个训练批次将包含多少个实例。 所以X的形状是（None，n_inputs）。 同样，我们知道y将是一个1D张量，每个实例有一个入口，但是我们再次不知道在这一点上训练批次的大小，所以形状（None）。

```
X = tf.placeholder(tf.float32, shape=(None, n_inputs), name="X")
y = tf.placeholder(tf.int64, shape=(None), name="y")
```

现在让我们创建一个实际的神经网络。 占位符X将作为输入层; 在执行阶段，它将一次更换一个训练批次（注意训练批中的所有实例将由神经网络同时处理）。 现在您需要创建两个隐藏层和输出层。 两个隐藏的层几乎相同：它们只是它们所连接的输入和它们包含的神经元的数量不同。 输出层也非常相似，但它使用softmax激活功能而不是ReLU激活功能。 所以让我们创建一个neuron_layer() 函数，我们将一次创建一个图层。 它将需要参数来指定输入，神经元数量，激活功能和图层的名称：

```
def neuron_layer(X, n_neurons, name, activation=None):
    with tf.name_scope(name):
        n_inputs = int(X.get_shape()[1])
        stddev = 2 / np.sqrt(n_inputs)
        init = tf.truncated_normal((n_inputs, n_neurons), stddev=stddev)
        W = tf.Variable(init, name="weights")
        b = tf.Variable(tf.zeros([n_neurons]), name="biases")
        z = tf.matmul(X, W) + b
        if activation == "relu":
            return tf.nn.relu(z)
```

```
        else:
            return z
```

我们逐行浏览这个代码：

1.首先，我们使用名称范围来创建每层的名称：它将包含该神经元层的所有计算节点。 这是可选的，但如果节点组织良好，则TensorBoard图形将会更加出色。

2.接下来，我们通过查找输入矩阵的形状并获得第二个维度的大小来获得输入数量（第一个维度用于实例）。

3. 接下来的三行创建一个保存权重矩阵的W变量。 它将是包含每个输入和每个神经元之间的所有连接权重的2D张量;因此，它的形状将是（n_inputs，n_neurons）。它将被随机初始化，使用具有标准差为$2/\sqrt{n_{inputs}}$的截断的正态（高斯）分布(使用截断的正态分布而不是常规正态分布确保不会有任何大的权重，这可能会减慢培训。).使用这个特定的标准差有助于算法的收敛速度更快（我们将在第11章中进一步讨论这一点），这是对神经网络的微小调整之一，对它们的效率产生了巨大的影响）。 重要的是为所有隐藏层随机初始化连接权重，以避免梯度下降算法无法中断的任何对称性。

例如，如果将所有权重设置为0，则所有神经元将输出0，并且给定隐藏层中的所有神经元的误差梯度将相同。 然后，梯度下降步骤将在每个层中以相同的方式更新所有权重，因此它们将保持相等。 换句话说，尽管每层有数百个神经元，你的模型就像每层只有一个神经元一样。

4. 下一行创建一个偏差的b变量，初始化为0（在这种情况下无对称问题），每个神经元有一个偏置参数。

5. 然后我们创建一个子图来计算z = X·W + b。 该向量化实现将有效地计算输入的加权和加上层中每个神经元的偏置，对于批次中的所有实例，仅需一次.

6.最后，如果激活参数设置为“relu”，则代码返回relu（z）（即max（0，z）），否则它只返回z。

好了，现在你有一个很好的函数来创建一个神经元层。 让我们用它来创建深层神经网络！ 第一个隐藏层以X为输入。 第二个将第一个隐藏层的输出作为其输入。 最后，输出层将第二个隐藏层的输出作为其输入。

```
    with tf.name_scope("dnn"):
        hidden1 = neuron_layer(X, n_hidden1, "hidden1", activation="relu")
    hidden2 = neuron_layer(hidden1, n_hidden2, "hidden2", activation="relu")
    logits = neuron_layer(hidden2, n_outputs, "outputs")
```

请注意，为了清楚起见，我们再次使用名称范围。 还要注意，logit是在通过softmax激活函数之前神经网络的输出：为了优化，我们稍后将处理softmax计算。

正如你所期望的，TensorFlow有许多方便的功能来创建标准的神经网络层，所以通常不需要像我们刚才那样定义你自己的neuron_layer（）函数。 例如，TensorFlow的fully_connected（）函数创建一个完全连接的层，其中所有输入都连接到图层中的所有神经元。 它使用正确的初始化策略来负责创建权重和偏置变量，并且默认情况下使用ReLU激活功能（我们可以使用activate_fn参数来更改它）。 正如我们将在第11章中看到的，它还支持正则化和归一化参数。 我们来调整上面的代码来使用fully_connected（）函数，而不是我们的neuron_layer（）函数。 只需导入该功能，并使用以下代码替换dnn构建部分：

```
    from tensorflow.contrib.layers import fully_connected
    with tf.name_scope("dnn"):
        hidden1 = fully_connected(X, n_hidden1, scope="hidden1")
```

```
        hidden2 = fully_connected(hidden1, n_hidden2, scope="hidden2")
        logits = fully_connected(hidden2, n_outputs, scope="outputs",
                                 activation_fn=None)
```

tensorflow.contrib包包含许多有用的功能，但它是一个尚未分级成为主要TensorFlow API一部分的实验代码的地方。 因此，full_connected（）函数（和任何其他contrib代码）可能会在将来更改或移动。

使用dense（）代替neuron_layer（）

注意：本书使用tensorflow.contrib.layers.fully_connected（）而不是tf.layers.dense（）（本章编写时不存在）。

现在最好使用tf.layers.dense（），因为contrib模块中的任何内容可能会更改或删除，恕不另行通知。 dense（）函数与fully_connected（）函数几乎相同，除了一些细微的差别：

几个参数被重命名：scope变为名称，activation_fn变为激活（同样_fn后缀从其他参数（如normalizer_fn）中删除），weights_initializer成为kernel_initializer等。默认激活现在是无，而不是tf.nn.relu。 第11章还介绍了更多的差异。

```
with tf.name_scope("dnn"):
    hidden1 = tf.layers.dense(X, n_hidden1, name="hidden1",
                              activation=tf.nn.relu)
    hidden2 = tf.layers.dense(hidden1, n_hidden2, name="hidden2",
                              activation=tf.nn.relu)
    logits = tf.layers.dense(hidden2, n_outputs, name="outputs")
```

现在我们已经有了神经网络模型，我们需要定义我们用来训练的损失函数。 正如我们在第4章中对Softmax回归所做的那样，我们将使用交叉熵。 正如我们之前讨论的，交叉熵将惩罚估计目标类的概率较低的模型。 TensorFlow提供了几种计算交叉熵的功能。 我们将使用sparse_softmax_cross_entropy_with_logits（）：它根据“logit”计算交叉熵（即，在通过softmax激活函数之前的网络输出），并且期望以0到-1数量的整数形式的标签（在我们的例子中，从0到9）。 这将给我们一个包含每个实例的交叉熵的1D张量。 然后，我们可以使用TensorFlow的reduce_mean（）函数来计算所有实例的平均交叉熵。

```
with tf.name_scope("loss"):
    xentropy = tf.nn.sparse_softmax_cross_entropy_with_logits(labels=y, logits=logits)
    loss = tf.reduce_mean(xentropy, name="loss")
```

该sparse_softmax_cross_entropy_with_logits（）函数等同于应用SOFTMAX激活功能，然后计算交叉熵，但它更高效，它妥善照顾的角落情况下，比如logits等于0,这就是为什么我们没有较早的应用SOFTMAX激活函数。 还有称为softmax_cross_entropy_with_logits（）的另一个函数，该函数在标签one-hot载体的形式（而不是整数0至类的数目减1）。

我们有神经网络模型，我们有成本函数，现在我们需要定义一个GradientDescentOptimizer来调整模型参数以最小化损失函数。 没什么新鲜的; 就像我们在第9章中所做的那样：

```
learning_rate = 0.01

with tf.name_scope("train"):
    optimizer = tf.train.GradientDescentOptimizer(learning_rate)
    training_op = optimizer.minimize(loss)
```

建模阶段的最后一个重要步骤是指定如何评估模型。 我们将简单地将精度用作我们的绩效指标。 首先，对于每个实例，通过检查最高logit是否对应于目标类别来确定神经网络的预测是否正确。 为此，您可以使用in_top_k（）函数。 这返回一个充满布尔值的1D张量，因此我们需要将这些布尔值转换为浮点数，然后计算平均值。 这将给我们网络的整体准确性.

```
with tf.name_scope("eval"):
    correct = tf.nn.in_top_k(logits, y, 1)
    accuracy = tf.reduce_mean(tf.cast(correct, tf.float32))
```

而且，像往常一样，我们需要创建一个初始化所有变量的节点，我们还将创建一个Saver来将我们训练有素的模型参数保存到磁盘中：

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

唷！ 建模阶段结束。 这是不到40行代码，但相当激烈：我们为输入和目标创建占位符，我们创建了一个构建神经元层的函数，我们用它来创建DNN，我们定义了成本函数，我们 创建了一个优化器，最后定义了性能指标。 现在到执行阶段。

Execution Phase

这部分要短得多，更简单。 首先，我们加载MNIST。 我们可以像之前的章节那样使用ScikitLearn，但是TensorFlow提供了自己的帮助器来获取数据，将其缩放（0到1之间），将它洗牌，并提供一个简单的功能来一次加载一个小批量：

```
from tensorflow.examples.tutorials.mnist import input_data

mnist = input_data.read_data_sets("/tmp/data/")
```

现在我们定义我们要运行的迭代数，以及小批量的大小：

```
n_epochs = 10001
batch_size = 50
```

现在我们去训练模型:

```
with tf.Session() as sess:
    init.run()
    for epoch in range(n_epochs):
        for iteration in range(mnist.train.num_examples // batch_size):
            X_batch, y_batch = mnist.train.next_batch(batch_size)
            sess.run(training_op, feed_dict={X: X_batch, y: y_batch})
        acc_train = accuracy.eval(feed_dict={X: X_batch, y: y_batch})
        acc_test = accuracy.eval(feed_dict={X: mnist.test.images, y: mnist.test.labels})
        print(epoch, "Train accuracy:", acc_train, "Test accuracy:", acc_test)

    save_path = saver.save(sess, "./my_model_final.ckpt")
```

该代码打开一个TensorFlow会话，并运行初始化所有变量的init节点。 然后它运行的主要训练循环：在每个时期，通过一些小批次的对应于训练集的大小的代码进行迭代。 每个小批量通过next_batch（）方法获取，然后

代码简单地运行训练操作，为当前的小批量输入数据和目标提供。 接下来，在每个时期结束时，代码评估最后一个小批量和完整训练集上的模型，并打印出结果。 最后，模型参数保存到磁盘。

Using the Neural Network

现在神经网络被训练了，你可以用它进行预测。 为此，您可以重复使用相同的建模阶段，但是更改执行阶段，如下所示：

```
with tf.Session() as sess:
    saver.restore(sess, "./my_model_final.ckpt") # or better, use save_path
    X_new_scaled = mnist.test.images[:20]
    Z = logits.eval(feed_dict={X: X_new_scaled})
    y_pred = np.argmax(Z, axis=1)
```

首先，代码从磁盘加载模型参数。 然后加载一些您想要分类的新图像。 记住应用与训练数据相同的特征缩放（在这种情况下，将其从0缩放到1）。 然后代码评估对数点节点。 如果您想知道所有估计的类概率，则需要将softmax（）函数应用于对数，但如果您只想预测一个类，则可以简单地选择具有最高logit值的类（使用argmax（）函数做的伎俩）。

Fine-Tuning Neural Network Hyperparameters

神经网络的灵活性也是其主要缺点之一：有很多超参数要进行调整。 不仅可以使用任何可想象的网络拓扑（如何神经元互连），而且即使在简单的MLP中，您可以更改层数，每层神经元数，每层使用的激活函数类型，weights 初始化逻辑等等。 你怎么知道什么组合的超参数是最适合你的任务？

当然，您可以使用具有交叉验证的网格搜索来查找正确的超参数，就像您在前几章中所做的那样，但是由于要调整许多超参数，并且由于在大型数据集上训练神经网络需要很多时间， 您只能在合理的时间内探索超参数空间的一小部分。 正如我们在第2章中讨论的那样，使用随机搜索 http://http://www.jmlr.org/papers/volume13/bergstra12a/bergstra12a.pdf 要好得多。另一个选择是使用诸如Oscar之类的工具，它可以实现更复杂的算法，以帮助您快速找到一组好的超参数.

它有助于了解每个超级参数的值是合理的，因此您可以限制搜索空间。 我们从隐藏层数开始。

Number of Hidden Layers

对于许多问题，您只需从单个隐藏层开始，您将获得合理的结果。 实际上已经表明，只有一个隐藏层的MLP可以建模甚至最复杂的功能，只要它具有足够的神经元。 长期以来，这些事实说服了研究人员，没有必要调查任何更深层次的神经网络。 但是他们忽略了这样一个事实：深层网络具有比浅层网络更高的参数效率：他们可以使用比浅网格更少的神经元来建模复杂的函数，使得训练更快。

要了解为什么，假设您被要求使用一些绘图软件绘制一个森林，但是您被禁止使用复制/粘贴。 你必须单独绘制每棵树，每枝分枝，每叶叶。 如果你可以画一个叶，复制/粘贴它来绘制一个分支，然后复制/粘贴该分支来创建一个树，最后复制/粘贴这个树来制作一个林，你将很快完成。 现实世界的数据通常以这样一种分层的方式进行结构化，DNN自动利用这一事实：较低的隐藏层模拟低级结构（例如，各种形状和方向的线段），中间隐藏层将这些低级结构组合到 模型中级结构（例如，正方形，圆形）和最高隐藏层和输出层将这些中间结构组合在一起，以模拟高级结构（如面）。

这种分层架构不仅可以帮助DNN更快地融合到一个很好的解决方案，而且还可以提高其将其推广到新数据集的能力。 例如，如果您已经训练了模型以识别图片中的脸部，并且您现在想要训练一个新的神经网络来识别发型，那么您可以通过重新使用第一个网络的较低层次来启动训练。 而不是随机初始化新神经网络的前几层的权

重和偏置，您可以将其初始化为第一个网络的较低层的权重和偏置的值。这样，网络将不必从大多数图片中低结构中从头学习; 它只需要学习更高层次的结构（例如发型）。

总而言之，对于许多问题，您可以从一个或两个隐藏层开始，它可以正常工作（例如，您可以使用只有一个隐藏层和几百个神经元，在MNIST数据集上容易达到97％以上的准确度使用两个具有相同总神经元数量的隐藏层，在大致相同的训练时间量中精确度为98％）。对于更复杂的问题，您可以逐渐增加隐藏层的数量，直到您开始覆盖训练集。非常复杂的任务，例如大型图像分类或语音识别，通常需要具有数十个层（或甚至数百个但不完全相连的网络）的网络，正如我们将在第13章中看到的那样），并且需要大量的训练数据。但是，您将很少从头开始训练这样的网络：重用预先训练的最先进的网络执行类似任务的部分更为常见。训练将会更快，需要更少的数据（我们将在第11章中进行讨论）

Number of Neurons per Hidden Layer

显然，输入和输出层中神经元的数量由您的任务需要的输入和输出类型决定。例如，MNIST任务需要28×28 = 784个输入神经元和10个输出神经元。对于隐藏的层次来说，通常的做法是将其设置为形成一个漏斗，每个层面上的神经元越来越少，原因在于许多低级别功能可以合并成更少的高级功能。例如，MNIST的典型神经网络可能具有两个隐藏层，第一个具有300个神经元，第二个具有100个。但是，这种做法现在并不常见，您可以为所有隐藏层使用相同的大小 - 例如，所有隐藏的层与150个神经元：这样只用调整一次超参数而不是每层都需要调整(因为如果每层一样,比如150,之后调就每层都调成160)。就像层数一样，您可以尝试逐渐增加神经元的数量，直到网络开始过度拟合。一般来说，通过增加每层的神经元数量，可以增加层数，从而获得更多的消耗。不幸的是，正如你所看到的，找到完美的神经元数量仍然是黑色的艺术.

一个更简单的方法是选择一个具有比实际需要的更多层次和神经元的模型，然后使用early stopping 来防止它过度拟合（以及其他正则化技术，特别是drop out，我们将在第11章中看到）。 这被称为“拉伸裤”的方法：而不是浪费时间寻找完美匹配您的大小的裤子，只需使用大型伸缩裤，缩小到合适的尺寸。

Activation Functions

在大多数情况下，您可以在隐藏层中使用ReLU激活功能（或其中一个变体，我们将在第11章中看到）。 与其他激活功能相比，计算速度要快一些，而梯度下降在局部最高点上并不会被卡住，因为它不会对大的输入值饱和（与逻辑函数或双曲正切函数相反, 他们容易在1饱和)

对于输出层，softmax激活函数通常是分类任务的良好选择（当这些类是互斥的时）。 对于回归任务，您完全可以不使用激活函数。

这就是人造神经网络的这个介绍。 在接下来的章节中，我们将讨论训练非常深的网络的技术，并分发多个服务器和GPU的培训。 然后我们将探讨一些其他流行的神经网络架构：卷积神经网络，复发神经网络和自动编码器。

完整代码

```
from tensorflow.examples.tutorials.mnist import input_data
import tensorflow as tf
from sklearn.metrics import accuracy_score
import numpy as np



if __name__ == '__main__':
    n_inputs = 28 * 28
    n_hidden1 = 300
```

```
n_hidden2 = 100
n_outputs = 10


mnist = input_data.read_data_sets("/tmp/data/")

X_train = mnist.train.images
X_test = mnist.test.images
y_train = mnist.train.labels.astype("int")
y_test = mnist.test.labels.astype("int")


X = tf.placeholder(tf.float32, shape= (None, n_inputs), name='X')
y = tf.placeholder(tf.int64, shape=(None), name = 'y')

with tf.name_scope('dnn'):
    hidden1 = tf.layers.dense(X, n_hidden1, activation=tf.nn.relu
                              ,name= 'hidden1')

    hidden2 = tf.layers.dense(hidden1, n_hidden2, name='hidden2',
                              activation= tf.nn.relu)

    logits = tf.layers.dense(hidden2, n_outputs, name='outputs')

with tf.name_scope('loss'):
    xentropy = tf.nn.sparse_softmax_cross_entropy_with_logits(labels = y,
                                                              logits = logits)
    loss = tf.reduce_mean(xentropy, name='loss')#所有值求平均

learning_rate = 0.01

with tf.name_scope('train'):
    optimizer = tf.train.GradientDescentOptimizer(learning_rate)
    training_op = optimizer.minimize(loss)

with tf.name_scope('eval'):
    correct = tf.nn.in_top_k(logits ,y ,1)#是否与真值一致 返回布尔值
    accuracy = tf.reduce_mean(tf.cast(correct, tf.float32)) #tf.cast将数据转化为0,1序列

init = tf.global_variables_initializer()


n_epochs = 20
batch_size = 50
with tf.Session() as sess:
    init.run()
    for epoch in range(n_epochs):
        for iteration in range(mnist.train.num_examples // batch_size):
            X_batch, y_batch = mnist.train.next_batch(batch_size)
            sess.run(training_op,feed_dict={X:X_batch,
                                            y: y_batch})
        acc_train = accuracy.eval(feed_dict={X:X_batch,
                                            y: y_batch})
        acc_test = accuracy.eval(feed_dict={X: mnist.test.images,
```

```
                                            y: mnist.test.labels})
        print(epoch, "Train accuracy:", acc_train, "Test accuracy:", acc_test)
```

十一、Training Deep Neural Nets

本篇文章是个人翻译的,如有商业用途,请通知本人谢谢.

Vanishing/Exploding Gradients Problems

正如我们在第10章中所讨论的那样，反向传播算法的工作原理是从输出层到输入层，传播错误梯度。 一旦该算法已经计算了网络中每个参数的损失函数的梯度，它就使用这些梯度来用梯度下降步骤来更新每个参数。

不幸的是，梯度往往变得越来越小，随着算法进展到下层。 结果，梯度下降更新使得低层连接权重实际上保持不变，并且训练永远不会收敛到良好的解决方案。 这被称为消失梯度问题。 在某些情况下，可能会发生相反的情况：梯度可能变得越来越大，许多分层得到疯狂的权重更新，算法发散。 这是梯度爆炸的问题，这在回归神经网络中最常见（见第14章）。 更一般地说，深度神经网络遭受不稳定的梯度; 不同的层次可能以不同的速度学习。

虽然这种不幸的行为已经经过了相当长的一段时间的实验观察（这是造成深度神经网络大部分时间都被抛弃的原因之一），但在2010年左右，人们才有了明显的进步。 Xavier Glorot和Yoshua Bengio发表的题为“Understanding the Difficulty of Training Deep Feedforward Neural Networks”的论文发现了一些疑问，包括流行的sigmoid激活函数和当时最受欢迎的权重初始化技术的组合，即随机初始化时使用平均值为0，标准偏差为1的正态分布。简而言之，他们表明，用这个激活函数和这个初始化方案，每层输出的方差远大于其输入的方差。在网络中前进，每层之后的变化持续增加，直到激活函数饱和在顶层。这实际上是因为对数函数的平均值为0.5而不是0（双曲正切函数的平均值为0，表现略好于深层网络中的逻辑函数）

看一下sigmoid激活函数（参见图11-1），可以看到当输入变大（负或正）时，函数饱和在0或1，导数非常接近0.因此，当反向传播开始时， 它几乎没有梯度通过网络传播回来，而且由于反向传播通过顶层向下传递，所以存在的小梯度不断地被稀释，因此下层确实没有任何东西可用。

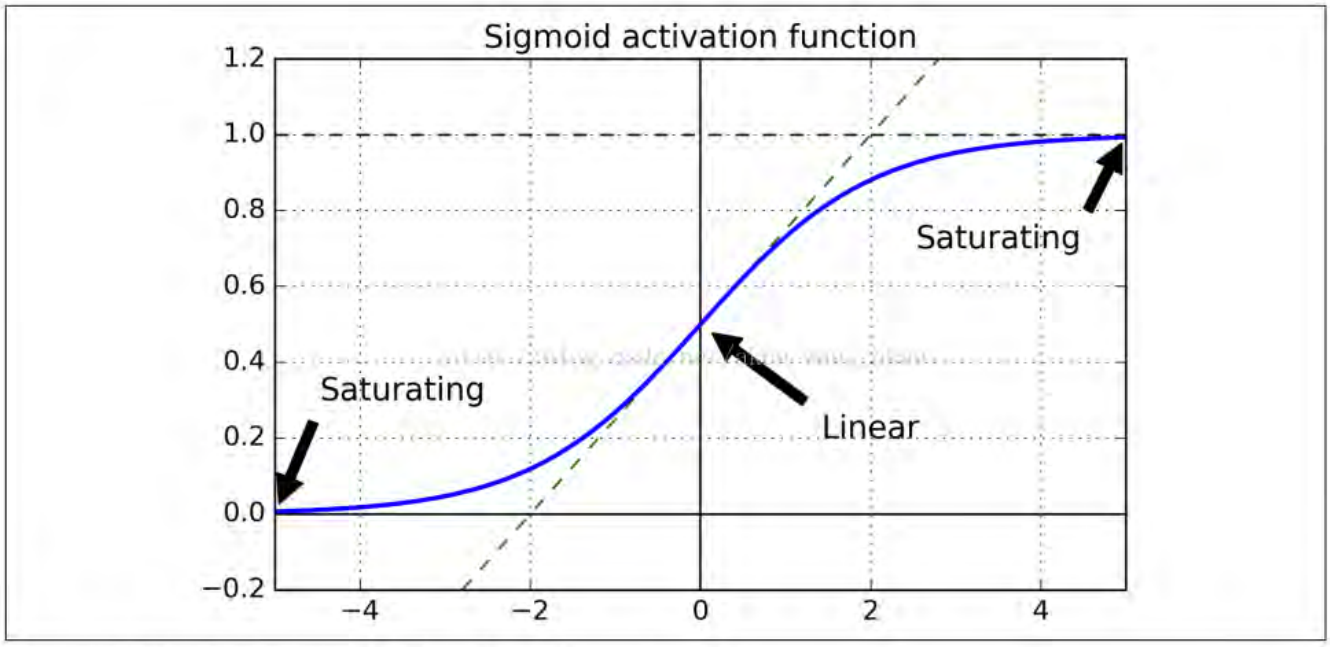


Figure 11-1. Logistic activation function saturation

Glorot和Bengio在他们的论文中提出了一种显着缓解这个问题的方法。 我们需要信号在两个方向上正确地流动：在进行预测时是正向的，在反向传播梯度时是反向的。 我们不希望信号消失，也不希望它爆炸并饱和。为了使信号正确流动，作者认为，我们需要每层输出的方差等于其输入的方差.(这里有一个比喻：如果将麦克风放大器的旋钮设置得太接近于零，人们听不到声音，但是如果将麦克风放大器设置得太接近麦克风，声音就会饱和，人们不会理解你在说什么。 现在想象一下这样一个放大器的链条：它们都需要正确设置，以便在链条的末端响亮而清晰地发出声音。 你的声音必须以每个放大器的振幅相同的幅度出来。)而且我们也需要梯度在相反方向上流过一层之前和之后有相同的方差（如果您对数学细节感兴趣，请查阅论文）。实际上不可能保证两者都是一样的，除非这个层具有相同数量的输入和输出连接，但是他们提出了一个很好的折衷办法，在实践中证明这个折中办法非常好：随机初始化连接权重必须如公式11-1所描述的那样.其中n_inputs和n_outputs是权重正在被初始化的层（也称为扇入和扇出）的输入和输出连接的数量。 这种初始化策略通常被称为Xavier初始化（在作者的名字之后），或者有时是Glorot初始化。

Equation 11-1. Xavier initialization (when using the logistic activation function)

Normal distribution with mean 0 and standard deviation $\sigma = \sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$

[数] 均匀分布

Or a uniform distribution between -r and +r, with $r = \sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$

当输入连接的数量大致等于输出连接的数量时，可以得到更简单的等式

$(\text{e.g., } \sigma = 1/\sqrt{n_{\text{inputs}}} \text{ or } r = \sqrt{3}/\sqrt{n_{\text{inputs}}})$，我们在第10章中使用了这个简化的策略。

使用Xavier初始化策略可以大大加快训练速度，这是导致Deep Learning目前取得成功的技巧之一。 最近的一些论文针对不同的激活函数提供了类似的策略，如表11-1所示。 ReLU激活函数（及其变体，包括简称ELU激活）的初始化策略有时称为He初始化（在其作者的姓氏之后）。

Table 11-1. Initialization parameters for each type of activation function

Activation function	Uniform distribution [–r, r]	Normal distribution
Logistic	$r = \sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$	$\sigma = \sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$
Hyperbolic tangent	$r = 4\sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$	$\sigma = 4\sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$
ReLU (and its variants)	$r = \sqrt{2}\sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$	$\sigma = \sqrt{2}\sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$

默认情况下，fully_connected（）函数（在第10章中介绍）使用Xavier初始化（具有统一的分布）。 你可以通过使用如下所示的variance_scaling_initializer（）函数来将其更改为He初始化：

注意：本书使用tensorflow.contrib.layers.fully_connected（）而不是tf.layers.dense（）（本章编写时不存在）。 现在最好使用tf.layers.dense（），因为contrib模块中的任何内容可能会更改或删除，恕不另行通知。 dense（）函数几乎与fully_connected（）函数完全相同。 与本章有关的主要差异是：

几个参数被重新命名：范围变成名字，activation_fn变成激活（类似地，_fn后缀从诸如normalizer_fn之类的其他参数中移除），weights_initializer变成kernel_initializer等等。默认激活现在是None，而不是tf.nn.relu。 它不支持tensorflow.contrib.framework.arg_scope（）（稍后在第11章中介绍）。 它不支持正规化的参数（稍后在第11章介绍）。

```
he_init = tf.contrib.layers.variance_scaling_initializer()
hidden1 = tf.layers.dense(X, n_hidden1, activation=tf.nn.relu,
                          kernel_initializer=he_init, name="hidden1")
```

他的初始化只考虑了扇入，而不是像Xavier初始化那样扇入和扇出之间的平均值。 这也是variance_scaling_initializer（）函数的默认值，但您可以通过设置参数mode =“FAN_AVG”来更改它。

Nonsaturating Activation Functions

Glorot和Bengio在2010年的论文中的一个见解是，消失/爆炸的梯度问题部分是由于激活函数的选择不好造成的。 在那之前，大多数人都认为，如果大自然选择在生物神经元中使用sigmoid激活函数，它们必定是一个很好的选择。 但事实证明，其他激活函数在深度神经网络中表现得更好，特别是ReLU激活函数，主要是因为它对正值不会饱和（也因为这样所以计算速度很快）。

不幸的是，ReLU激活功能并不完美。 它有一个被称为死亡ReLUs的问题：在训练过程中，一些神经元有效地死亡，意味着它们停止输出0以外的任何东西。在某些情况下，你可能会发现你网络的一半神经元已经死亡，特别是如果你使用大 学习率。 在训练期间，如果神经元的权重得到更新，使得神经元输入的加权和为负，则它将开始输出0.当发生这种情况时，由于ReLU函数的梯度为0时，神经元不可能恢复生命当其输入为负。

为了解决这个问题，你可能需要使用ReLU函数的一个变体，比如leaky ReLU。这个函数定义为LeakyReLUα（z）= max（αz，z）（见图11-2）。超参数α定义了函数“leaks”的程度：它是z <0的函数的斜率，通常设置为0.01。这个小斜坡确保leaky ReLUs永不死亡;他们可能会长期昏迷，但他们有机会最终醒来。最近的一篇论文比较了几种ReLU激活功能的变体，其中一个结论是leaky Relu总是优于严格的ReLU激活函数。事实上，设定α= 0.2（巨大leak）似乎导致比α= 0.01（小leak）更好的性能。他们还评估了随机leakt ReLU（RReLU），其中α在训练期间在给定范围内随机挑选，并在测试期间固定为平均值。它表现相当好，似乎是一个正规化者（减少训练集的过拟合风险）。最后，他们还评估了参数 leaky ReLU（PReLU），其中α被授权在训练期间被学习（而不是超参数，它变成可以像任何其他参数一样被反向传播修改的参数）。据报道这在大型图像数据集上的表现强于ReLU，但是对于较小的数据集，其具有过度拟合训练集的风险。

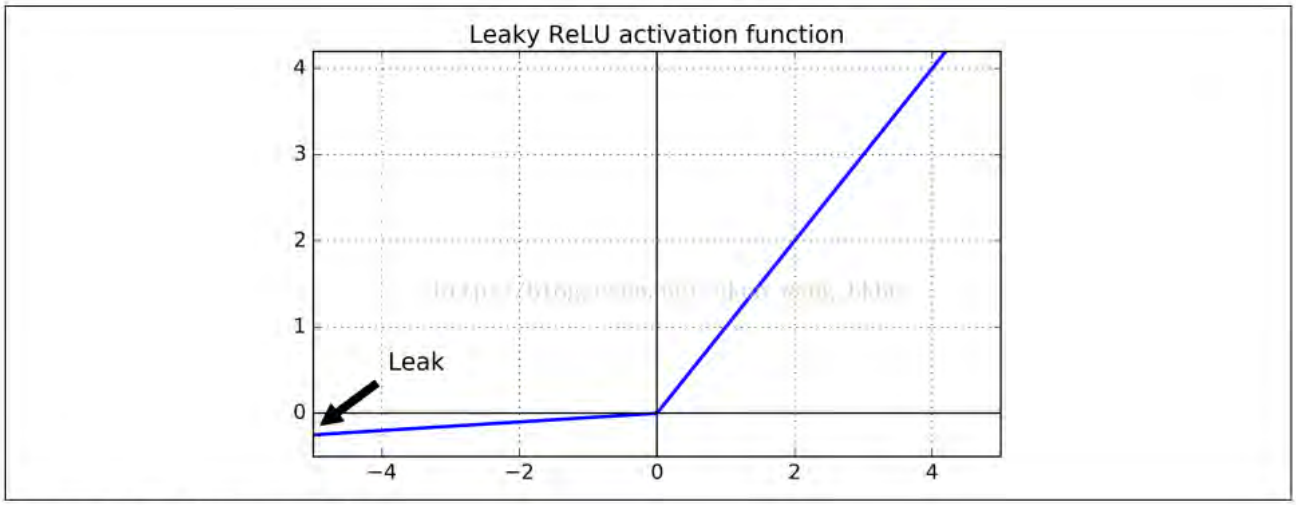


Figure 11-2. Leaky ReLU

最后但并非最不重要的一点是，Djork-ArnéClevert等人在2015年的一篇论文中提出了一种称为指数线性单元exponential linear unit （ELU）的新的激活函数，在他们的实验中表现优于所有的ReLU变体：训练时间减少，神经网络在测试集上表现的更好。 如图11-3所示，公式11-2给出了它的定义。

Equation 11-2. ELU activation function

$$\mathrm{ELU}_{\alpha}(z) = \begin{cases} \alpha(\exp(z) - 1) & \text{if } z < 0 \\ z & if z \geq 0 \end{cases}$$

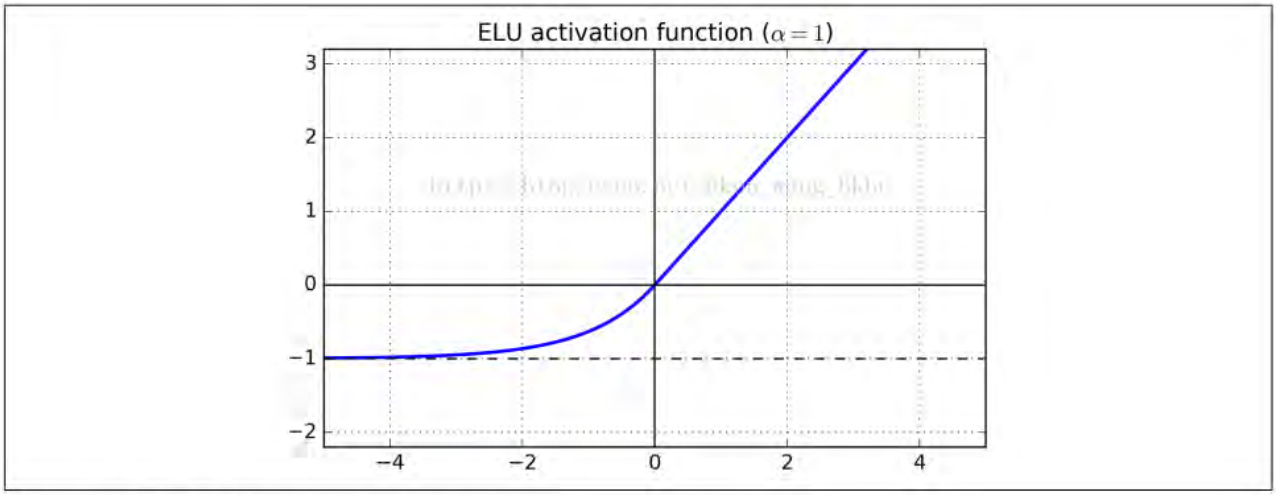


Figure 11-3. ELU activation function

它看起来很像ReLU功能，但有一些区别,主要区别在于：

- 首先它在z <0时取负值，这使得该单元的平均输出接近于0.这有助于减轻消失梯度问题，如前所述。 超参数α定义当z是一个大的负数时，ELU函数接近的值。 它通常设置为1，但是如果你愿意，你可以像调整其他超参数一样调整它。
- 其次，它对z <0有一个非零的梯度，避免了神经元死亡的问题.
- 第三，函数在任何地方都是平滑的，包括z = 0左右，这有助于加速梯度下降，因为它不会像z = 0的左边和右边那样反弹。

ELU激活函数的主要缺点是计算速度慢于ReLU及其变体（由于使用指数函数），但是在训练过程中，这是通过更快的收敛速度来补偿的。 然而，在测试时间，ELU网络将比ReLU网络慢。

那么你应该使用哪个激活函数来处理深层神经网络的隐藏层？ 虽然你的里程会有所不同，一般ELU>leaky ReLU（及其变体）> ReLU> tanh>sigmoid。 如果您关心运行时性能，那么您可能更喜欢ELU相对 leaky ReLU。 如果你不想调整另一个超参数，你可以使用前面提到的默认的α值（leaky ReLU为0.01，ELU为1）。 如果您有充足的时间和计算能力，您可以使用交叉验证来评估其他激活功能，特别是如果您的神经网络过拟合，则为RReLU;如果您拥有庞大的训练组，则为PReLU。
TensorFlow提供了一个可以用来建立神经网络的elu（）函数。 调用fully_connected（）函数时，只需设置activation_fn参数即可：

```
hidden1 = tf.layers.dense(X, n_hidden1, activation=tf.nn.elu, name="hidden1")
```

TensorFlow没有针对leaky ReLU的预定义函数，但是很容易定义：

```
def leaky_relu(z, name=None):
    return tf.maximum(0.01 * z, z, name=name)

hidden1 = tf.layers.dense(X, n_hidden1, activation=leaky_relu, name="hidden1")
```

Batch Normalization

尽管使用HE初始化和ELU（或任何ReLU变体）可以显着减少训练开始阶段的消失/爆炸梯度问题，但不保证在训练期间问题不会回来。

在2015年的一篇论文中，Sergey Ioffe和Christian Szegedy提出了一种称为批量标准化（Batch Normalization，BN）的技术来解决消失/爆炸梯度问题，更普遍的问题是当之前的层次改变,每个层次输入的分布会在训练过程中发生变化（他们称之为内部协变量问题）。
该技术包括在每层的激活函数之前在模型中添加操作，简单地对输入进行零中心和归一化，然后使用每个层的两个新参数（一个用于缩放，另一个用于移位）对结果进行缩放和移位。 换句话说，这个操作可以让模型学习到每层输入值的最佳尺度,均值。
为了对输入进行归零和归一化，算法需要估计输入的均值和标准差。 它通过评估当前小批量输入的均值和标准偏差（因此命名为“批量标准化”）来实现。 整个操作在方程11-3中。

Equation 11-3. Batch Normalization algorithm

$$1.\quad \mu_B = \frac{1}{m_B}\sum_{i=1}^{m_B}\mathbf{x}^{(i)}$$

$$2.\quad \sigma_B^{\,2} = \frac{1}{m_B}\sum_{i=1}^{m_B}\left(\mathbf{x}^{(i)} - \mu_B\right)^2$$

$$3.\quad \widehat{\mathbf{x}}^{(i)} = \frac{\mathbf{x}^{(i)} - \mu_B}{\sqrt{\sigma_B^{\,2} + \epsilon}}$$

$$4.\quad \mathbf{z}^{(i)} = \gamma\widehat{\mathbf{x}}^{(i)} + \beta$$

μB 是整个小批量B的经验均值

σB 是经验性的标准差，也是来评估整个小批量的。

mB 是小批量中的实例数量。

$\widehat{\mathbf{x}}^{(i)}$ 是以为零中心和标准化的输入。

γ 是层的缩放参数。

β是层的移动参数（偏移量）

ε是一个很小的数字，以避免被零除（通常为10 ^ -3）。 这被称为平滑术语(拉布拉斯平滑Laplace Smoothing)。

$\mathbf{z}^{(i)}$ 是BN操作的输出：它是输入的缩放和移位版本。

在测试时，没有小批量计算经验均值和标准差，所以您只需使用整个训练集的均值和标准差。 这些通常在训练期间使用移动平均值进行有效计算。 因此，总的来说，每个批次标准化的层次都学习了四个参数：γ（标度），β（偏移），μ（平均值）和σ（标准偏差）。

作者证明，这项技术大大改善了他们试验的所有深度神经网络。消失梯度问题大大减少了，他们可以使用饱和激活函数，如tanh甚至sigmoid激活函数。网络对权重初始化也不那么敏感。他们能够使用更大的学习速度，显着加快了学习过程。具体地，他们指出，“适用于一个国家的最先进的图像分类模型，批标准化实现了与14倍更少的训练步骤相同的精度，和以显著的优势击败了原始模型。 [...]使用批量归一化的网络集合，我们改进了ImageNet分类上的最佳公布结果：达到4.9％的前5个验证错误（和4.8％的测试错误），超出了人类评估者的准确性。批量标准化也像一个正规化者一样，减少了对其他正则化技术（如本章稍后描述的dropout）.

但是，批量标准化确实增加了模型的一些复杂性（尽管它消除了对输入数据进行标准化的需要，因为如果批的标准化处理，第一个隐藏层将处理这个问题）。 此外，还存在运行时间的损失：由于每层所需的额外计算，神经网络的预测速度较慢。 所以，如果你需要预测闪电般快速，你可能想要检查普通ELU + He初始化执行之前如何执行批量规范化。

您可能会发现，训练起初相当缓慢，而渐变下降正在寻找每层的最佳尺度和偏移量，但一旦找到合理的好值，它就会加速。

Implementing Batch Normalization with TensorFlow

TensorFlow提供了一个batch_normalization（）函数，它简单地对输入进行居中和标准化，但是您必须自己计算平均值和标准偏差（基于训练期间的小批量数据或测试过程中的完整数据集） 作为这个函数的参数，并且还必须处理缩放和偏移量参数的创建（并将它们传递给此函数）。 这是可行的，但不是最方便的方法。 相反，你应该使用batch_norm（）函数，它为你处理所有这些。 您可以直接调用它，或者告诉fully_connected（）函数使用它，如下面的代码所示：

注意：本书使用tensorflow.contrib.layers.batch_norm（）而不是tf.layers.batch_normalization（）（本章写作时不存在）。 现在最好使用tf.layers.batch_normalization（），因为contrib模块中的任何内容都可能会改变或被删除，恕不另行通知。 我们现在不使用batch_norm（）函数作为fully_connected（）函数的正则化参数，而是使用batch_normalization（），并明确地创建一个不同的层。 参数有些不同，特别是：

- decay更名为momentum
- is_training被重命名为training
- updates_collections被删除：批量标准化所需的更新操作被添加到UPDATE_OPS集合中，并且您需要在训练期间明确地运行这些操作（请参阅下面的执行阶段）
- 我们不需要指定scale = True，因为这是默认值。

还要注意，为了在每个隐藏层激活函数之前运行batch norm，我们手动应用RELU激活函数，在批量规范层之后。
注意：由于tf.layers.dense（）函数与本书中使用的tf.contrib.layers.arg_scope（）不兼容，我们现在使用python的functools.partial（）函数。 它可以很容易地创建一个my_dense_layer（）函数，只需调用tf.layers.dense（），并自动设置所需的参数（除非在调用my_dense_layer（）时覆盖它们）。 如您所见，代码保持非常相似。

```
import tensorflow as tf

n_inputs = 28 * 28
n_hidden1 = 300
n_hidden2 = 100
n_outputs = 10

X = tf.placeholder(tf.float32, shape=(None, n_inputs), name="X")

training = tf.placeholder_with_default(False, shape=(), name='training')

hidden1 = tf.layers.dense(X, n_hidden1, name="hidden1")
bn1 = tf.layers.batch_normalization(hidden1, training=training, momentum=0.9)
bn1_act = tf.nn.elu(bn1)

hidden2 = tf.layers.dense(bn1_act, n_hidden2, name="hidden2")
bn2 = tf.layers.batch_normalization(hidden2, training=training, momentum=0.9)
```

```
bn2_act = tf.nn.elu(bn2)

logits_before_bn = tf.layers.dense(bn2_act, n_outputs, name="outputs")
logits = tf.layers.batch_normalization(logits_before_bn, training=training,
                                       momentum=0.9)
```

```
X = tf.placeholder(tf.float32, shape=(None, n_inputs), name="X")
training = tf.placeholder_with_default(False, shape=(), name='training')
```

为了避免一遍又一遍重复相同的参数，我们可以使用Python的partial（）函数：

```
from functools import partial

my_batch_norm_layer = partial(tf.layers.batch_normalization,
                              training=training, momentum=0.9)

hidden1 = tf.layers.dense(X, n_hidden1, name="hidden1")
bn1 = my_batch_norm_layer(hidden1)
bn1_act = tf.nn.elu(bn1)
hidden2 = tf.layers.dense(bn1_act, n_hidden2, name="hidden2")
bn2 = my_batch_norm_layer(hidden2)
bn2_act = tf.nn.elu(bn2)
logits_before_bn = tf.layers.dense(bn2_act, n_outputs, name="outputs")
logits = my_batch_norm_layer(logits_before_bn)
```

完整代码

```
from functools import partial
from tensorflow.examples.tutorials.mnist import input_data
import tensorflow as tf



if __name__ == '__main__':
    n_inputs = 28 * 28
    n_hidden1 = 300
    n_hidden2 = 100
    n_outputs = 10



    mnist = input_data.read_data_sets("/tmp/data/")

    batch_norm_momentum = 0.9
    learning_rate = 0.01

    X = tf.placeholder(tf.float32, shape=(None, n_inputs), name = 'X')
    y = tf.placeholder(tf.int64, shape=None, name = 'y')
    training = tf.placeholder_with_default(False, shape=(), name = 'training')#给Batch norm加

    with tf.name_scope("dnn"):
        he_init = tf.contrib.layers.variance_scaling_initializer()
        #对权重的初始化
```

```
        my_batch_norm_layer = partial(
            tf.layers.batch_normalization,
            training = training,
            momentum = batch_norm_momentum
        )

        my_dense_layer = partial(
            tf.layers.dense,
            kernel_initializer = he_init
        )

        hidden1 = my_dense_layer(X ,n_hidden1 ,name = 'hidden1')
        bn1 = tf.nn.elu(my_batch_norm_layer(hidden1))
        hidden2 = my_dense_layer(bn1, n_hidden2, name = 'hidden2')
        bn2 = tf.nn.elu(my_batch_norm_layer(hidden2))
        logists_before_bn = my_dense_layer(bn2, n_outputs, name = 'outputs')
        logists = my_batch_norm_layer(logists_before_bn)

    with tf.name_scope('loss'):
        xentropy = tf.nn.sparse_softmax_cross_entropy_with_logits(labels = y, logits= logists
        loss = tf.reduce_mean(xentropy, name = 'loss')

    with tf.name_scope('train'):
        optimizer = tf.train.GradientDescentOptimizer(learning_rate)
        training_op = optimizer.minimize(loss)

    with tf.name_scope("eval"):
        correct = tf.nn.in_top_k(logists, y, 1)
        accuracy = tf.reduce_mean(tf.cast(correct, tf.float32))

    init = tf.global_variables_initializer()
    saver = tf.train.Saver()

    n_epoches = 20
    batch_size = 200
#注意：由于我们使用的是tf.layers.batch_normalization（）而不是tf.contrib.layers.batch_norm（）（
#所以我们需要明确运行批量规范化所需的额外更新操作（sess.run（[ training_op，extra_update_ops]，...
    extra_update_ops = tf.get_collection(tf.GraphKeys.UPDATE_OPS)

    with tf.Session() as sess:
        init.run()
        for epoch in range(n_epoches):
            for iteraton in range(mnist.train.num_examples//batch_size):
                X_batch, y_batch = mnist.train.next_batch(batch_size)
                sess.run([training_op,extra_update_ops],
                         feed_dict={training:True, X:X_batch, y:y_batch})
            accuracy_val = accuracy.eval(feed_dict= {X:mnist.test.images,
                                                     y:mnist.test.labels})
            print(epoch, 'Test accuracy:', accuracy_val)
```

```
0 Test accuracy: 0.865
1 Test accuracy: 0.8951
2 Test accuracy: 0.9097
3 Test accuracy: 0.919
4 Test accuracy: 0.9279
5 Test accuracy: 0.9344
6 Test accuracy: 0.9384
7 Test accuracy: 0.9436
8 Test accuracy: 0.9473
9 Test accuracy: 0.9515
10 Test accuracy: 0.9536
11 Test accuracy: 0.9539
12 Test accuracy: 0.9564
13 Test accuracy: 0.9593
14 Test accuracy: 0.9598
15 Test accuracy: 0.9611
16 Test accuracy: 0.9631
17 Test accuracy: 0.964
18 Test accuracy: 0.9649
19 Test accuracy: 0.9652
```

什么！？ 这对MNIST来说不是一个很好的准确性。 当然，如果你训练的时间越长，准确性就越好，但是由于这样一个浅的网络，Batch Norm和ELU不太可能产生非常积极的影响：它们大部分都是为了更深的网络而发光。
请注意，您还可以训练操作取决于更新操作：

```
with tf.name_scope("train"):
    optimizer = tf.train.GradientDescentOptimizer(learning_rate)
    extra_update_ops = tf.get_collection(tf.GraphKeys.UPDATE_OPS)
    with tf.control_dependencies(extra_update_ops):
        training_op = optimizer.minimize(loss)
```

这样，你只需要在训练过程中评估training_op，TensorFlow也会自动运行更新操作：

```
sess.run(training_op, feed_dict={training: True, X: X_batch, y: y_batch})
```

Gradient Clipping (梯度剪裁)

减少爆炸梯度问题的一种常用技术是在反向传播过程中简单地剪切梯度，使它们不超过某个阈值（这对于递归神经网络是非常有用的;参见第14章）。 这就是所谓的渐变剪裁。一般来说，人们更喜欢批量标准化，但了解渐变剪裁以及如何实现它仍然是有用的。

在TensorFlow中，优化器的minimize（）函数负责计算梯度并应用它们，所以您必须首先调用优化器的compute_gradients（）方法，然后使用clip_by_value（）函数创建一个剪辑梯度的操作，最后 创建一个操作来使用优化器的apply_gradients（）方法应用剪切梯度：

```
threshold = 1.0

optimizer = tf.train.GradientDescentOptimizer(learning_rate)
```

```
grads_and_vars = optimizer.compute_gradients(loss)
capped_gvs = [(tf.clip_by_value(grad, -threshold, threshold), var)
              for grad, var in grads_and_vars]
training_op = optimizer.apply_gradients(capped_gvs)
```

像往常一样，您将在每个训练阶段运行这个training_op。 它将计算梯度，将它们夹在-1.0和1.0之间，并应用它们。 threhold是您可以调整的超参数。

Reusing Pretrained Layers (重用预训练层)

从零开始训练一个非常大的DNN通常不是一个好主意，相反，您应该总是尝试找到一个现有的神经网络来完成与您正在尝试解决的任务类似的任务，然后重新使用这个较低层的 网络：这就是所谓的迁移学习。 这不仅会大大加快培训速度，还将需要更少的培训数据。

例如，假设您可以访问经过培训的DNN，将图片分为100个不同的类别，包括动物，植物，车辆和日常物品。您现在想要训练一个DNN来对特定类型的车辆进行分类。 这些任务非常相似，因此您应该尝试重新使用第一个网络的一部分（请参见图11-4）。

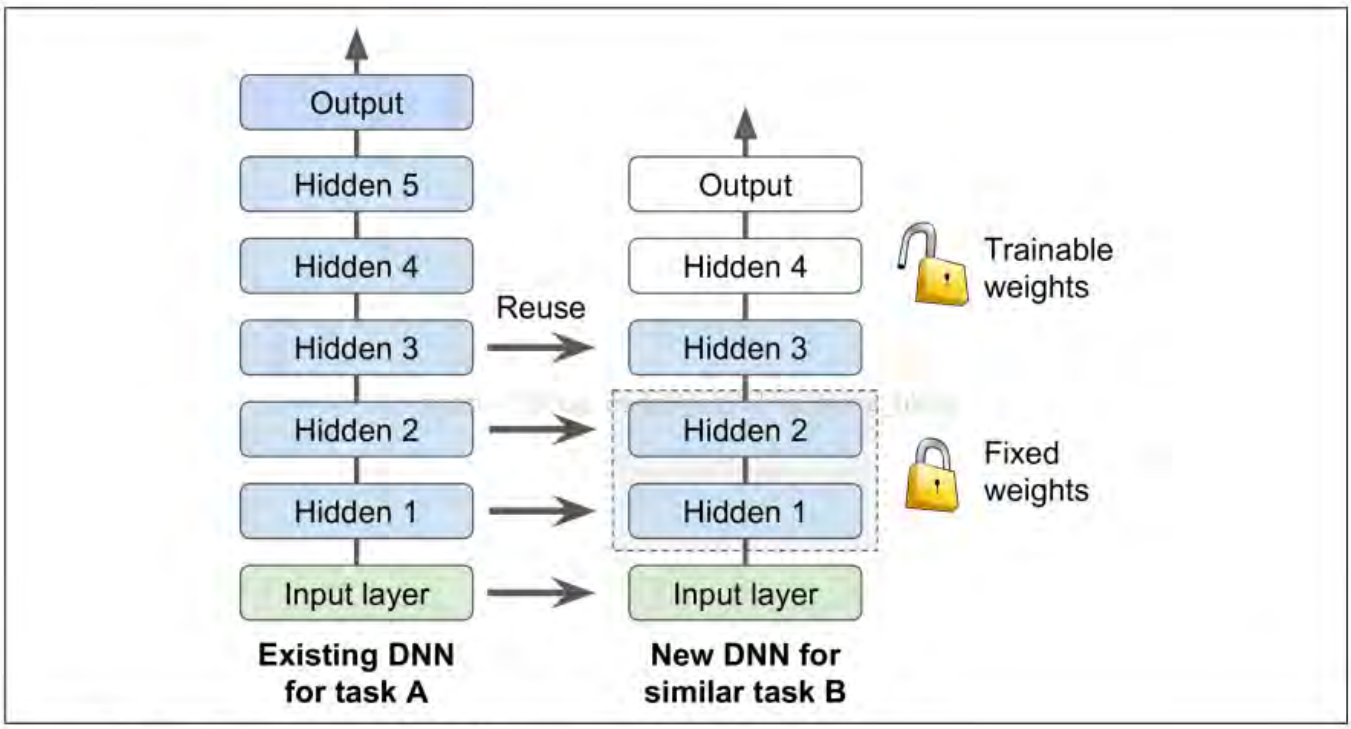


Figure 11-4. Reusing pretrained layers

如果新任务的输入图像与原始任务中使用的输入图像的大小不一致，则必须添加预处理步骤以将其大小调整为原始模型的预期大小。 更一般地说，如果输入具有类似的低级层次的特征，则迁移学习将很好地工作。

Reusing a TensorFlow Model

如果原始模型使用TensorFlow进行训练，则可以简单地将其恢复并在新任务上进行训练：

```
[...] # construct the original model
```

```
with tf.Session() as sess:
    saver.restore(sess, "./my_model_final.ckpt")
    # continue training the model...
```

完整代码:

```
n_inputs = 28 * 28  # MNIST
n_hidden1 = 300
n_hidden2 = 50
n_hidden3 = 50
n_hidden4 = 50
n_outputs = 10

X = tf.placeholder(tf.float32, shape=(None, n_inputs), name="X")
y = tf.placeholder(tf.int64, shape=(None), name="y")

with tf.name_scope("dnn"):
    hidden1 = tf.layers.dense(X, n_hidden1, activation=tf.nn.relu, name="hidden1")
    hidden2 = tf.layers.dense(hidden1, n_hidden2, activation=tf.nn.relu, name="hidden2")
    hidden3 = tf.layers.dense(hidden2, n_hidden3, activation=tf.nn.relu, name="hidden3")
    hidden4 = tf.layers.dense(hidden3, n_hidden4, activation=tf.nn.relu, name="hidden4")
    hidden5 = tf.layers.dense(hidden4, n_hidden5, activation=tf.nn.relu, name="hidden5")
    logits = tf.layers.dense(hidden5, n_outputs, name="outputs")

with tf.name_scope("loss"):
    xentropy = tf.nn.sparse_softmax_cross_entropy_with_logits(labels=y, logits=logits)
    loss = tf.reduce_mean(xentropy, name="loss")

with tf.name_scope("eval"):
    correct = tf.nn.in_top_k(logits, y, 1)
    accuracy = tf.reduce_mean(tf.cast(correct, tf.float32), name="accuracy")

learning_rate = 0.01
threshold = 1.0

optimizer = tf.train.GradientDescentOptimizer(learning_rate)
grads_and_vars = optimizer.compute_gradients(loss)
capped_gvs = [(tf.clip_by_value(grad, -threshold, threshold), var)
              for grad, var in grads_and_vars]
training_op = optimizer.apply_gradients(capped_gvs)

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

```
with tf.Session() as sess:
    saver.restore(sess, "./my_model_final.ckpt")

    for epoch in range(n_epochs):
        for iteration in range(mnist.train.num_examples // batch_size):
            X_batch, y_batch = mnist.train.next_batch(batch_size)
            sess.run(training_op, feed_dict={X: X_batch, y: y_batch})
        accuracy_val = accuracy.eval(feed_dict={X: mnist.test.images,
                                                y: mnist.test.labels})
        print(epoch, "Test accuracy:", accuracy_val)

    save_path = saver.save(sess, "./my_new_model_final.ckpt")
```

但是，一般情况下，您只需要重新使用原始模型的一部分（就像我们将要讨论的那样）。 一个简单的解决方案是将Saver配置为仅恢复原始模型中的一部分变量。 例如，下面的代码只恢复隐藏的层1,2和3：

```
n_inputs = 28 * 28  # MNIST
n_hidden1 = 300 # reused
n_hidden2 = 50  # reused
n_hidden3 = 50  # reused
n_hidden4 = 20  # new!
n_outputs = 10  # new!

X = tf.placeholder(tf.float32, shape=(None, n_inputs), name="X")
y = tf.placeholder(tf.int64, shape=(None), name="y")

with tf.name_scope("dnn"):
    hidden1 = tf.layers.dense(X, n_hidden1, activation=tf.nn.relu, name="hidden1")       # re
    hidden2 = tf.layers.dense(hidden1, n_hidden2, activation=tf.nn.relu, name="hidden2") # re
    hidden3 = tf.layers.dense(hidden2, n_hidden3, activation=tf.nn.relu, name="hidden3") # re
    hidden4 = tf.layers.dense(hidden3, n_hidden4, activation=tf.nn.relu, name="hidden4") # ne
    logits = tf.layers.dense(hidden4, n_outputs, name="outputs")                         # ne

with tf.name_scope("loss"):
    xentropy = tf.nn.sparse_softmax_cross_entropy_with_logits(labels=y, logits=logits)
    loss = tf.reduce_mean(xentropy, name="loss")

with tf.name_scope("eval"):
    correct = tf.nn.in_top_k(logits, y, 1)
    accuracy = tf.reduce_mean(tf.cast(correct, tf.float32), name="accuracy")

with tf.name_scope("train"):
    optimizer = tf.train.GradientDescentOptimizer(learning_rate)
    training_op = optimizer.minimize(loss)
```

```
[...] # build new model with the same definition as before for hidden layers 1-3
```

```
reuse_vars = tf.get_collection(tf.GraphKeys.GLOBAL_VARIABLES,
                               scope="hidden[123]") # regular expression
reuse_vars_dict = dict([(var.op.name, var) for var in reuse_vars])
restore_saver = tf.train.Saver(reuse_vars_dict) # to restore layers 1-3

init = tf.global_variables_initializer()
saver = tf.train.Saver()

with tf.Session() as sess:
    init.run()
    restore_saver.restore(sess, "./my_model_final.ckpt")

    for epoch in range(n_epochs):                                            # not shown in the boo
        for iteration in range(mnist.train.num_examples // batch_size): # not shown
            X_batch, y_batch = mnist.train.next_batch(batch_size)      # not shown
            sess.run(training_op, feed_dict={X: X_batch, y: y_batch})  # not shown
```

```
        accuracy_val = accuracy.eval(feed_dict={X: mnist.test.images,  # not shown
                                                y: mnist.test.labels}) # not shown
        print(epoch, "Test accuracy:", accuracy_val)                   # not shown

    save_path = saver.save(sess, "./my_new_model_final.ckpt")
```

首先我们建立新的模型，确保复制原始模型的隐藏层1到3.我们还创建一个节点来初始化所有变量。 然后我们得到刚刚用“trainable = True”（这是默认值）创建的所有变量的列表，我们只保留那些范围与正则表达式“hidden [123]”相匹配的变量（即，我们得到所有可训练的 隐藏层1到3中的变量）。 接下来，我们创建一个字典，将原始模型中每个变量的名称映射到新模型中的名称（通常需要保持完全相同的名称）。 然后，我们创建一个Saver，它将只恢复这些变量，并且创建另一个Saver来保存整个新模型，而不仅仅是第1层到第3层。然后，我们开始一个会话并初始化模型中的所有变量，然后从 原始模型的层1到3.最后，我们在新任务上训练模型并保存。

任务越相似，您可以重复使用的层越多（从较低层开始）。 对于非常相似的任务，您可以尝试保留所有隐藏的图层，然后替换输出图层。

Reusing Models from Other Frameworks

如果模型是使用其他框架进行训练的，则需要手动加载权重（例如，如果使用Theano训练，则使用Theano代码），然后将它们分配给相应的变量。 这可能是相当乏味的。 例如，下面的代码显示了如何复制使用另一个框架训练的模型的第一个隐藏层的权重和偏置：

Freezing the Lower Layers

第一个DNN的低层可能已经学会了检测图片中的低级特征，这将在两个图像分类任务中有用，因此您可以按照原样重新使用这些图层。 在训练新的DNN时，“冻结”权重通常是一个好主意：如果下层权重是固定的，那么上层权重将更容易训练（因为他们不需要学习一个移动的目标）。 要在训练期间冻结较低层，最简单的解决方案是给优化器列出要训练的变量，不包括来自较低层的变量：

```
train_vars = tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES,
                               scope="hidden[34]|outputs")
training_op = optimizer.minimize(loss, var_list=train_vars)
```

第一行获得隐藏层3和4以及输出层中所有可训练变量的列表。 这留下了隐藏层1和2中的变量。接下来，我们将这个受限制的可列表变量列表提供给optimizer的minimize（）函数。当当！ 现在，图层1和图层2被冻结：在训练过程中不会发生变化（通常称为冻结图层）。

Caching the Frozen Layers (缓存冻层)

由于冻结层不会改变，因此可以为每个训练实例缓存最上面的冻结层的输出。 由于训练贯穿整个数据集很多次，这将给你一个巨大的速度提升，因为每个训练实例只需要经过一次冻结层（而不是每个时期一次）。 例如，你可以先运行整个训练集（假设你有足够的内存）：

```
hidden2_outputs = sess.run(hidden2, feed_dict={X: X_train})
```

然后在训练过程中，不是建立批次的训练实例，而是从隐藏层2建立成批的输出，并将它们提供给训练操作：

最后一行运行先前定义的训练操作（冻结图层1和2），并从第二个隐藏层（以及该批次的目标）为其输出一批输出。 因为我们给TensorFlow隐藏层2的输出，所以它不会去评估它（或者它所依赖的任何节点）。

Tweaking, Dropping, or Replacing the Upper Layers (调整，删除或替换上层)

原始模型的输出层通常应该被替换，因为对于新的任务来说，最有可能没有用处，甚至可能没有适合新任务的输出。

类似地，原始模型的上层隐藏层不太可能像下层一样有用，因为对于新任务来说最有用的高层特征可能与对原始任务最有用的高层特征明显不同。 你想找到正确的层数来重用。

尝试先冻结所有复制的图层，然后训练模型并查看它是如何执行的。 然后尝试解冻一个或两个顶层隐藏层，让反向传播调整它们，看看性能是否提高。 您拥有的训练数据越多，您可以解冻的层数就越多。

如果仍然无法获得良好的性能，并且您的训练数据很少，请尝试删除顶部的隐藏层，并再次冻结所有剩余的隐藏层。 您可以迭代，直到找到正确的层数重复使用。 如果您有足够的训练数据，您可以尝试替换顶部的隐藏层，而不是放下它们，甚至可以添加更多的隐藏层。

Model Zoos

你在哪里可以找到一个类似于你想要解决的任务训练的神经网络？ 首先看看显然是在你自己的模型目录。 这是保存所有模型并组织它们的一个很好的理由，以便您以后可以轻松地检索它们。 另一个选择是在模型动物园中搜索。 许多人为了各种不同的任务而训练机器学习模型，并且善意地向公众发布预训练模型。

TensorFlow在https://github.com/tensor流/ models中有自己的模型动物园。 特别是，它包含了大多数最先进的图像分类网络，如VGG，Inception和ResNet（参见第13章，检查模型/ slim目录），包括代码，预训练模型和工具来下载流行的图像数据集。

另一个流行的模型动物园是Caffe模型动物园。 它还包含许多在各种数据集（例如，ImageNet，Places数据库，CIFAR10等）上训练的计算机视觉模型（例如，LeNet，AlexNet，ZFNet，GoogLeNet，VGGNet，开始）。 Saumitro Dasgupta写了一个转换器，可以在https://github.com/ethereon/ca‰etensorflow.

Unsupervised Pretraining(　无监督的预训练)

假设你想要解决一个复杂的任务，你没有太多的标记的训练数据，但不幸的是，你不能找到一个类似的任务训练模型。 不要失去所有希望！ 首先，你当然应该尝试收集更多的有标签的训练数据，但是如果这太难或太昂贵，你仍然可以进行无监督的训练（见图11-5）。 也就是说，如果你有很多未标记的训练数据，你可以尝试逐层训练图层，从最低层开始，然后上升，使用无监督的特征检测器算法，如限制玻尔兹曼机器（RBMs;见附录E ）或自动编码器（见第15章）。 每个图层都被训练成先前训练过的图层的输出（除了被训练的图层之外的所有图层都被冻结）。 一旦所有图层都以这种方式进行了训练，就可以使用监督式学习（即反向传播）对网络进行微调。

这是一个相当漫长而乏味的过程，但通常运作良好。 实际上，这是Geoffrey Hinton和他的团队在2006年使用的技术，导致了神经网络的复兴和深度学习的成功。 直到2010年，无监督预训练（通常使用RBMs）是深网的标准，只有在消失梯度问题得到缓解之后，纯训练DNN才更为普遍。 然而，当您有一个复杂的任务需要解决时，无监督训练（现在通常使用自动编码器而不是RBM）仍然是一个很好的选择，没有类似的模型可以重复使用，而且标记的训练数据很少，但是大量的未标记的训练数据。(另一个选择是提出一个监督的任务，您可以轻松地收集大量标记的训练数据，然后使用迁移学习，如前所述。 例如，如果要训练一个模型来识别图片中的朋

友，你可以在互联网上下载数百万张脸并训练一个分类器来检测两张脸是否相同，然后使用此分类器将新图片与你朋友的每张照片做比较。）

Pretraining on an Auxiliary Task (在辅助任务上预训练)

最后一种选择是在辅助任务上训练第一个神经网络，您可以轻松获取或生成标记的训练数据，然后重新使用该网络的较低层来完成您的实际任务。 第一个神经网络的下层将学习可能被第二个神经网络重复使用的特征检测器。

例如，如果你想建立一个识别面孔的系统，你可能只有几个人的照片 - 显然不足以训练一个好的分类器。 收集每个人的数百张照片将是不实际的。 但是，您可以在互联网上收集大量随机人员的照片，并训练第一个神经网络来检测两张不同的照片是否属于同一个人。 这样的网络将学习面部优秀的特征检测器，所以重复使用它的较低层将允许你使用很少的训练数据来训练一个好的面部分类器。

收集没有标签的训练样本通常是相当便宜的，但标注它们却相当昂贵。 在这种情况下，一种常见的技术是将所有训练样例标记为“好”，然后通过破坏好的训练样例产生许多新的训练样例，并将这些样例标记为“坏”。然后，您可以训练第一个神经网络 将实例分类为好或不好。 例如，您可以下载数百万个句子，将其标记为“好”，然后在每个句子中随机更改一个单词，并将结果语句标记为“不好”。如果神经网络可以告诉“狗睡觉”是好的 句子，但“他们的狗”是坏的，它可能知道相当多的语言。 重用其较低层可能有助于许多语言处理任务。

另一种方法是训练第一个网络为每个训练实例输出一个分数，并使用一个损失函数确保一个好的实例的分数大于一个坏实例的分数至少一定的边际。 这被称为最大边际学习.

Faster Optimizers

训练一个非常大的深度神经网络可能会非常缓慢。 到目前为止，我们已经看到了四种加速培训的方法（并且达到更好的解决方案）：对连接权重应用良好的初始化策略，使用良好的激活功能，使用批量规范化以及重用预训练网络的部分。 另一个巨大的速度提升来自使用比普通渐变下降优化器更快的优化器。 在本节中，我们将介绍最流行的：动量优化，Nesterov加速梯度，AdaGrad，RMSProp，最后是Adam优化。

剧透：本节的结论是，您几乎总是应该使用Adam_optimization，所以如果您不关心它是如何工作的，只需使用AdamOptimizer替换您的GradientDescentOptimizer，然后跳到下一节！ 只需要这么小的改动，训练通常会快几倍。 但是，Adam优化确实有三个可以调整的超参数（加上学习率）。 默认值通常工作的不错，但如果您需要调整它们，可能会有助于知道他们做什么。 Adam optimization结合了来自其他优化算法的几个想法，所以先看看这些算法是有用的。

Momentum optimization

想象一下，一个保龄球在一个光滑的表面上平缓的斜坡上滚动：它会缓慢地开始，但是它会很快地达到最终的速度（如果有一些摩擦或空气阻力的话）。 这是Boris Polyak在1964年提出的Momentum优化背后的一个非常简单的想法.相比之下，普通的Gradient Descent只需要沿着斜坡进行小的有规律的下降步骤，所以需要更多的时间才能到达底部。

回想一下，梯度下降只是通过直接减去损失函数J（θ）相对于权重（θJ（θ））乘以学习率η的梯度来更新权重θ。 方程是：θ←θ-η∇θJ（θ）。 它不关心早期的梯度是什么。 如果局部梯度很小，则会非常缓慢。

Equation 11-4. Momentum algorithm

1. $\mathbf{m} \leftarrow \beta\mathbf{m} + \eta\nabla_\theta J(\theta)$
2. $\theta \leftarrow \theta - \mathbf{m}$

动量优化很关心以前的梯度：在每次迭代时，它将动量矢量m（乘以学习率η）的局部梯度相加，并且通过简单地减去该动量矢量来更新权重（参见公式11-4）。 换句话说，梯度用作加速度，不用作速度。 为了模拟某种摩擦机制，避免动量过大，该算法引入了一个新的超参数β，简称为动量，它必须设置在0（高摩擦）和1（无摩擦）之间。 典型的动量值是0.9。

您可以很容易地验证，如果梯度保持不变，则终端速度（即，权重更新的最大大小）等于该梯度乘以学习率η乘以例如，如果β= 0.9，则最终速度等于学习速率的梯度乘以10倍，因此动量优化比梯度下降快10倍！ 这使Momentum优化比Gradient Descent快得多。 特别是，我们在第四章中看到，当输入量具有非常不同的尺度时，损失函数看起来像一个细长的碗（见图4-7）。 梯度下降速度很快，但要花很长的时间才能到达底部。 相反，动量优化会越来越快地滚下山谷底部，直到达到底部（最佳）。

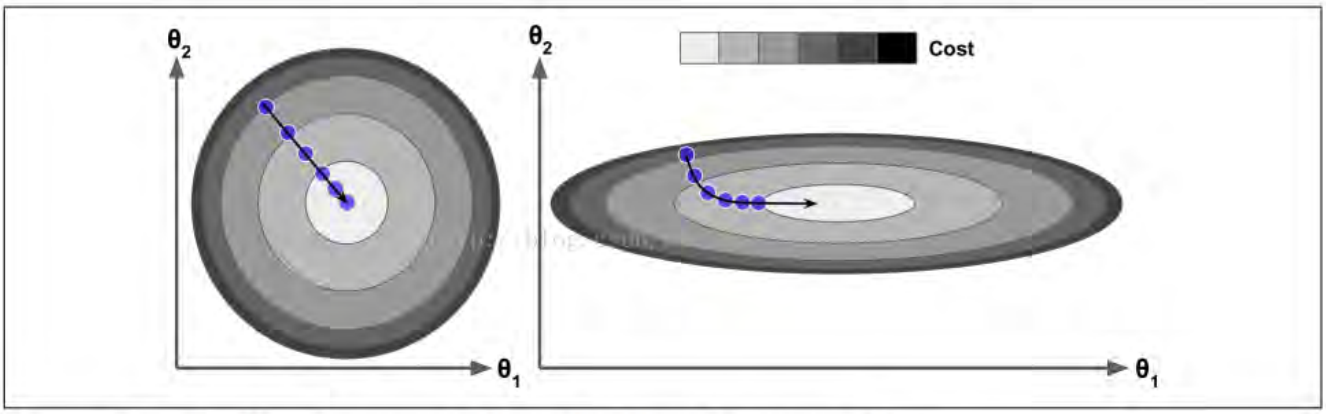


Figure 4-7. Gradient Descent with and without feature scaling

在不使用批处理标准化的深层神经网络中，上层通常最终输入具有不同的尺度，所以使用Momentum优化会有很大的帮助。 它也可以帮助滚过局部的最佳状态。

由于动力的原因，优化器可能会超调一些，然后再回来，再次超调，并在稳定在最小值之前多次振荡。 这就是为什么在系统中有一点摩擦的原因之一：它消除了这些振荡，从而加速了收敛。

在TensorFlow中实现Momentum优化是一件简单的事情：只需用MomentumOptimizer替换GradientDescentOptimizer，然后躺下来赚钱！

```
optimizer = tf.train.MomentumOptimizer(learning_rate=learning_rate,
                                       momentum=0.9)
```

Momentum优化的一个缺点是它增加了另一个超参数来调整。 然而，0.9的动量值通常在实践中运行良好，几乎总是比梯度下降快。

Nesterov Accelerated Gradient

Yurii Nesterov在1983年提出的Momentum优化的一个小变量几乎总是比vanilla Momentum优化更快。Nesterov Momentum优化或Nesterov加速梯度（Nesterov Accelerated Gradient，NAG）的思想是测量损失函数的梯度不是在局部位置，而是在动量方向稍微靠前（见公式11-5）。 与vanilla Momentum优化的唯一区别在于梯度是在θ+βm而不是在θ处测量的。

Equation 11-5. Nesterov Accelerated Gradient algorithm

1. $\mathbf{m} \leftarrow \beta\mathbf{m} + \eta\nabla_\theta J(\theta + \beta\mathbf{m})$
2. $\theta \leftarrow \theta - \mathbf{m}$

这个小小的调整是可行的，因为一般来说，动量矢量将指向正确的方向（即朝向最优方向），所以使用在该方向上测得的梯度稍微更精确，而不是使用 原始位置的梯度，如图11-6所示（其中∇1代表在起点θ处测量的代价函数的梯度，∇2代表位于θ+βm的点处的梯度）。

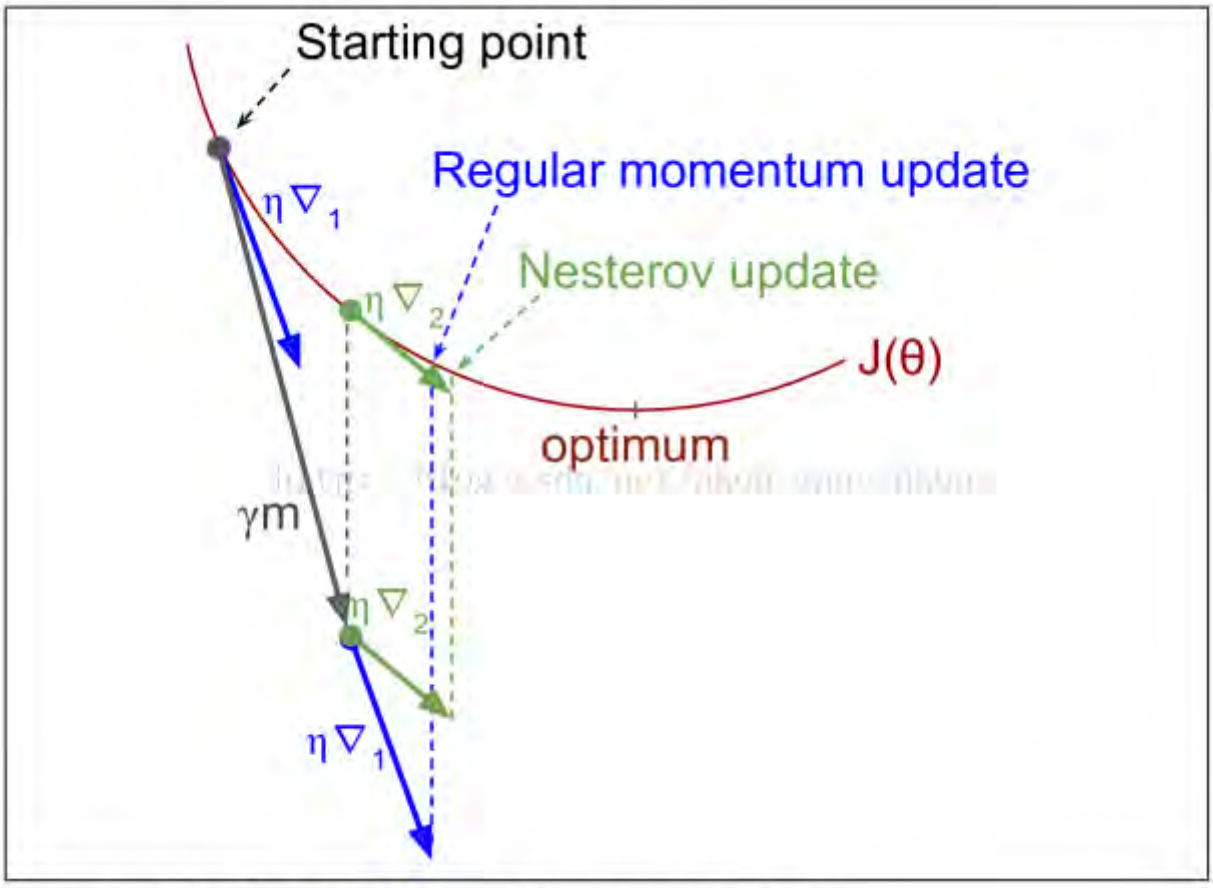


Figure 11-6. Regular versus Nesterov Momentum optimization

正如你所看到的，Nesterov更新稍微靠近最佳值。 过了一段时间，这些小的改进加起来，NAG最终比常规的Momentum优化快得多。 此外，请注意，当动量推动权重横跨山谷时，▿1继续推进越过山谷，而▿2推回山谷的底部。 这有助于减少振荡，从而更快地收敛。

与常规的动量优化相比，NAG几乎总能加速训练。 要使用它，只需在创建MomentumOptimizer时设置use_nesterov = True：

```
optimizer = tf.train.MomentumOptimizer(learning_rate=learning_rate,
                                       momentum=0.9, use_nesterov=True)
```

AdaGrad

再次考虑细长碗的问题：梯度下降从最陡峭的斜坡快速下降，然后缓慢地下到谷底。 如果算法能够早期检测到这个问题并且纠正它的方向来指向全局最优点，那将是非常好的。

AdaGrad算法通过沿着最陡的维度缩小梯度向量来实现这一点（见公式11-6）：

Equation 11-6. AdaGrad algorithm

1. $\mathbf{s} \leftarrow \mathbf{s} + \nabla_\theta J(\theta) \otimes \nabla_\theta J(\theta)$
2. $\theta \leftarrow \theta - \eta \nabla_\theta J(\theta) \oslash \sqrt{\mathbf{s} + \epsilon}$

第一步将梯度的平方累加到矢量s中（⊗符号表示单元乘法）。 这个向量化形式相当于向量s的每个元素si计算si←si +（∂/∂θiJ（θ））^ 2; 换一种说法，每个si关于参数θi累加损失函数的偏导数的平方。 如果损失函数沿着第i维陡峭，则在每次迭代时，si将变得越来越大。

第二步几乎与梯度下降相同，但有一个很大的不同：梯度矢量按比例缩小 $\sqrt{\mathbf{s} + \epsilon}$ （⊘符号表示元素分割，ε是避免被零除的平滑项，通常设置为 10^{-10}）. 这个矢量化的形式相当于计算 $\theta_i \leftarrow \theta_i - \eta \partial / \partial \theta_i J(\theta) / \sqrt{s_i + \epsilon}$ 对于所有参数θi（同时）。

简而言之，这种算法会降低学习速度，但对于陡峭的尺寸，其速度要快于具有温和的斜率的尺寸。 这被称为自适应学习率。 它有助于将更新的结果更直接地指向全局最优（见图11-7）。 另一个好处是它不需要那么多的去调整学习速率超参数η。

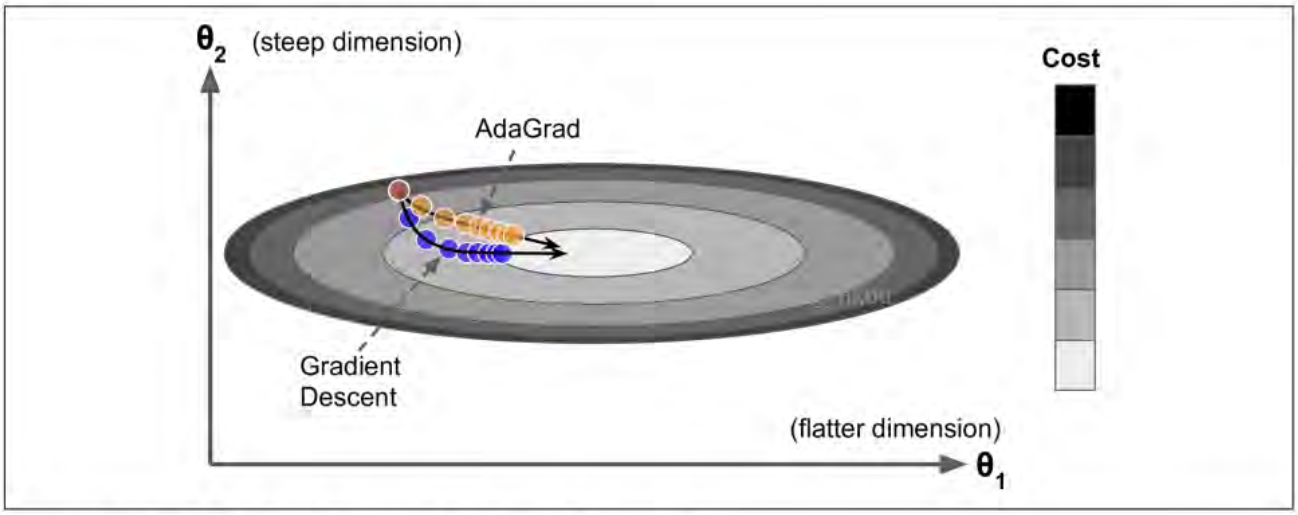


Figure 11-7. AdaGrad versus Gradient Descent

对于简单的二次问题，AdaGrad经常表现良好，但不幸的是，在训练神经网络时，它经常停止得太早。 学习速率被缩减得太多，以至于在达到全局最优之前，算法完全停止。 所以，即使TensorFlow有一个AdagradOptimizer，你也不应该用它来训练深度神经网络（虽然对线性回归这样简单的任务可能是有效的）。

RMSProp

尽管AdaGrad的速度变慢了一点，并且从未收敛到全局最优，但是RMSProp算法通过仅累积最近迭代（而不是从训练开始以来的所有梯度）的梯度来修正这个问题。 它通过在第一步中使用指数衰减来实现（见公式11-

7）。

Equation 11-7. RMSProp algorithm

$$1.\quad \mathbf{s} \leftarrow \beta\mathbf{s} + (1-\beta)\nabla_\theta J(\theta) \otimes \nabla_\theta J(\theta)$$

$$2.\quad \theta \leftarrow \theta - \eta\,\nabla_\theta J(\theta) \oslash \sqrt{\mathbf{s}+\epsilon}$$

他的衰变率β通常设定为0.9。 是的，它又是一个新的超参数，但是这个默认值通常运行良好，所以你可能根本不需要调整它。

正如您所料，TensorFlow拥有一个RMSPropOptimizer类：

```
optimizer = tf.train.RMSPropOptimizer(learning_rate=learning_rate,
                                      momentum=0.9, decay=0.9, epsilon=1e-10)
```

除了非常简单的问题，这个优化器几乎总是比AdaGrad执行得更好。 它通常也比Momentum优化和Nesterov加速梯度表现更好。 事实上，这是许多研究人员首选的优化算法，直到Adam optimization出现。

Adam Optimization

Adam，代表自适应矩估计，结合了动量优化和RMSProp的思想：就像动量优化一样，它追踪过去梯度的指数衰减平均值，就像RMSProp一样，它跟踪过去平方梯度的指数衰减平均值 （见方程式11-8）。

Equation 11-8. Adam algorithm

$$1.\quad \mathbf{m} \leftarrow \beta_1\mathbf{m} + \left(1-\beta_1\right)\nabla_\theta J(\theta)$$

$$2.\quad \mathbf{s} \leftarrow \beta_2\mathbf{s} + \left(1-\beta_2\right)\nabla_\theta J(\theta) \otimes \nabla_\theta J(\theta)$$

$$3.\quad \mathbf{m} \leftarrow \frac{\mathbf{m}}{1-{\beta_1}^T}$$

$$4.\quad \mathbf{s} \leftarrow \frac{\mathbf{s}}{1-{\beta_2}^T}$$

$$5.\quad \theta \leftarrow \theta - \eta\,\mathbf{m} \oslash \sqrt{\mathbf{s}+\epsilon}$$

T代表迭代次数（从1开始）。

如果你只看步骤1,2和5，你会注意到Adam与Momentum优化和RMSProp的相似性。 唯一的区别是第1步计算指数衰减的平均值，而不是指数衰减的和，但除了一个常数因子（衰减平均值只是衰减和的1 - β1倍）之外，它们实际上是等效的。 步骤3和步骤4是一个技术细节：由于m和s初始化为0，所以在训练开始时它们会偏向0，所以这两步将在训练开始时帮助提高m和s。

动量衰减超参数β1通常初始化为0.9，而缩放衰减超参数β2通常初始化为0.999。 如前所述，平滑项ε通常被初始化为一个很小的数，例如 10^{-8}.这些是TensorFlow的AdamOptimizer类的默认值，所以你可以简单地使用：

```
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
```

实际上，由于Adam是一种自适应学习速率算法（如AdaGrad和RMSProp），所以对学习速率超参数η的调整较少。 您经常可以使用默认值η= 0.001，使Adam更容易使用x相对于梯度下降。

迄今为止所讨论的所有优化技术都只依赖于一阶偏导数（雅可比矩阵）。 优化文献包含基于二阶偏导数（Hessians）的惊人算法。 不幸的是，这些算法很难应用于深度神经网络，因为每个输出有n ^ 2个Hessians（其中n是参数的数量），而不是每个输出只有n个Jacobian。 由于DNN通常具有数以万计的参数，二阶优化算法通常甚至不适合内存，甚至在他们这样做时，计算Hessians也是太慢了。

Training Sparse Models(训练稀疏模型)

所有刚刚提出的优化算法都会产生密集的模型，这意味着大多数参数都是非零的。 如果你在运行时需要一个非常快速的模型，或者如果你需要它占用较少的内存，你可能更喜欢用一个稀疏模型来代替。

实现这一点的一个微不足道的方法是像平常一样训练模型，然后摆脱微小的权重（将它们设置为0）。

另一个选择是在训练过程中应用强正则化，因为它会推动优化器尽可能多地消除权重（如第4章关于套索回归的讨论）。

但是，在某些情况下，这些技术可能仍然不足。 最后一个选择是应用双重平均，通常称为遵循正规化领导者（FTRL），一种由尤里·涅斯捷罗夫（Yurii Nesterov）提出的技术。 当与l1正则化一起使用时，这种技术通常导致非常稀疏的模型。 TensorFlow在FTRLOptimizer类中实现称为FTRL-Proximal的FTRL变体。

Learning Rate Scheduling(学习速率调度)

找到一个好的学习速度可能会非常棘手。 如果设置太高，训练实际上可能偏离（如我们在第4章）。 如果设置得太低，训练最终会收敛到最佳状态，但这需要很长时间。 如果将其设置得太高，开始的进度会非常快，但最终会围绕最佳方式跳舞，永远不会安顿下来（除非您使用自适应学习速率优化算法，如AdaGrad，RMSProp或Adam，但是 即使这样可能需要时间来解决）。 如果您的计算预算有限，那么您可能必须在正确收敛之前中断培训，产生次优解决方案（参见图11-8）。

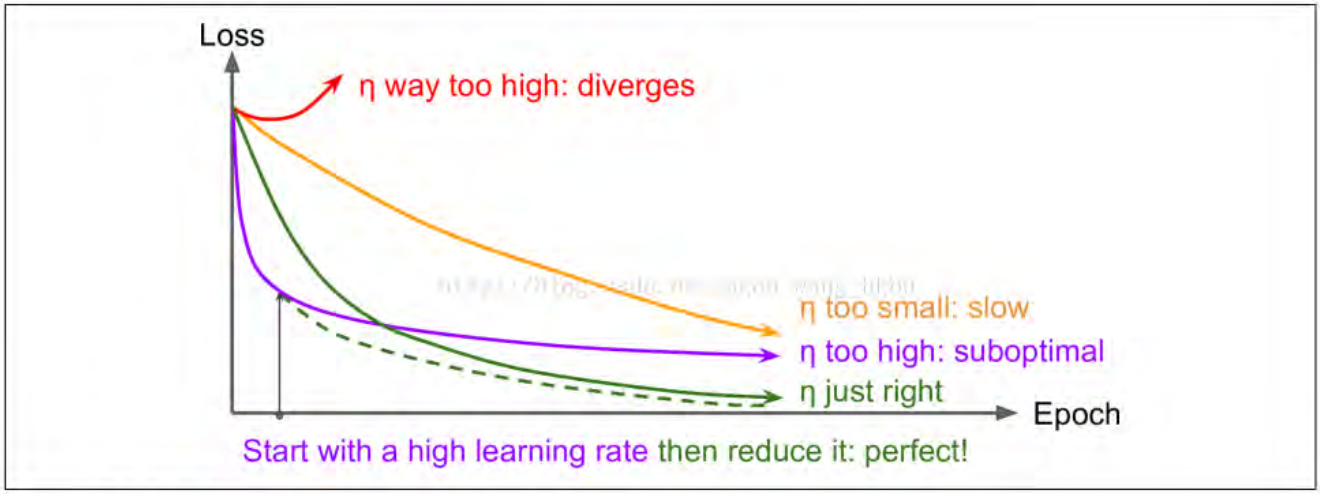


Figure 11-8. Learning curves for various learning rates η

通过使用各种学习速率和比较学习曲线，在几个时期内对您的网络进行多次训练，您也许能够找到相当好的学习速度。 理想的学习速度将会快速学习并收敛到良好的解决方案。

然而，你可以做得比不断的学习速度更好：如果你从一个高的学习速度开始，然后一旦它停止快速的进步就减少它，你可以比最佳的恒定学习速度更快地达到一个好的解决方案。 有许多不同的策略，以减少训练期间的学习率。 这些策略被称为学习时间表（我们在第4章中简要介绍了这个概念），其中最常见的是：

预定的分段恒定学习率:

例如，首先将学习率设置为η0= 0.1，然后在50个时期之后将学习率设置为η1= 0.001。 虽然这个解决方案可以很好地工作，但是通常需要弄清楚正确的学习速度以及何时使用它们。

性能调度:

每N步测量验证错误（就像提前停止一样），当错误停止下降时，将学习速率降低一个因子λ。

指数调度:

将学习率设置为迭代次数t的函数：$\eta(t) = \eta_0\ 10^{-t/r}$。 这很好，但它需要调整η0和r。 学习率将由每r步下降10个因素。

幂调度:

设学习率为$\eta(t) = \eta_0\ (1 + t/r)^{-c}$。 超参数c通常被设置为1.这与指数调度类似，但是学习速率下降要慢得多.

Andrew Senior等2013年的论文。 比较了使用Momentum优化训练深度神经网络进行语音识别时一些最流行的学习计划的性能。 作者得出结论：在这种情况下，性能调度和指数调度都表现良好，但他们更喜欢指数调度，因为它实现起来比较简单，容易调整，收敛速度略快于最佳解决方案。

使用TensorFlow实现学习计划非常简单：

```
    initial_learning_rate = 0.1
    decay_steps = 10000
    decay_rate = 1/10
    global_step = tf.Variable(0, trainable=False, name="global_step")
    learning_rate = tf.train.exponential_decay(initial_learning_rate, global_step,
                                               decay_steps, decay_rate)
    optimizer = tf.train.MomentumOptimizer(learning_rate, momentum=0.9)
    training_op = optimizer.minimize(loss, global_step=global_step)
```

设置超参数值后，我们创建一个不可跟踪的变量global_step（初始化为0）以跟踪当前的训练迭代次数。 然后我们使用TensorFlow的exponential_decay（）函数来定义指数衰减的学习率（η0= 0.1和r = 10,000）。 接下来，我们使用这个衰减的学习率创建一个优化器（在这个例子中是一个MomentumOptimizer）。 最后，我们通过调用优化器的minimize（）方法来创建训练操作; 因为我们将global_step变量传递给它，所以请注意增加它。 就是这样！

由于AdaGrad，RMSProp和Adam优化自动降低了培训期间的学习率，因此不需要添加额外的学习计划。 对于其他优化算法，使用指数衰减或性能调度可显着加速收敛。

完整代码:

```
n_inputs = 28 * 28  # MNIST
n_hidden1 = 300
```

```
n_hidden2 = 50
n_outputs = 10

X = tf.placeholder(tf.float32, shape=(None, n_inputs), name="X")
y = tf.placeholder(tf.int64, shape=(None), name="y")

with tf.name_scope("dnn"):
    hidden1 = tf.layers.dense(X, n_hidden1, activation=tf.nn.relu, name="hidden1")
    hidden2 = tf.layers.dense(hidden1, n_hidden2, activation=tf.nn.relu, name="hidden2")
    logits = tf.layers.dense(hidden2, n_outputs, name="outputs")

with tf.name_scope("loss"):
    xentropy = tf.nn.sparse_softmax_cross_entropy_with_logits(labels=y, logits=logits)
    loss = tf.reduce_mean(xentropy, name="loss")

with tf.name_scope("eval"):
    correct = tf.nn.in_top_k(logits, y, 1)
    accuracy = tf.reduce_mean(tf.cast(correct, tf.float32), name="accuracy")
```

```
with tf.name_scope("train"):       # not shown in the book
    initial_learning_rate = 0.1
    decay_steps = 10000
    decay_rate = 1/10
    global_step = tf.Variable(0, trainable=False, name="global_step")
    learning_rate = tf.train.exponential_decay(initial_learning_rate, global_step,
                                               decay_steps, decay_rate)
    optimizer = tf.train.MomentumOptimizer(learning_rate, momentum=0.9)
    training_op = optimizer.minimize(loss, global_step=global_step)
```

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

```
n_epochs = 5
batch_size = 50

with tf.Session() as sess:
    init.run()
    for epoch in range(n_epochs):
        for iteration in range(mnist.train.num_examples // batch_size):
            X_batch, y_batch = mnist.train.next_batch(batch_size)
            sess.run(training_op, feed_dict={X: X_batch, y: y_batch})
        accuracy_val = accuracy.eval(feed_dict={X: mnist.test.images,
                                                y: mnist.test.labels})
        print(epoch, "Test accuracy:", accuracy_val)

    save_path = saver.save(sess, "./my_model_final.ckpt")
```

Avoiding Overftting Through Regularization

有四个参数，我可以fit一个大象，五个我可以让他摆动他的象鼻。

—John von Neumann,cited by Enrico Fermi in Nature 427

深度神经网络通常具有数以万计的参数，有时甚至是数百万。 有了这么多的参数，网络拥有难以置信的自由度，可以适应各种复杂的数据集。 但是这个很大的灵活性也意味着它很容易过度训练集。有了数以百万计的参数，你可以适应整个动物园。 在本节中，我们将介绍一些最流行的神经网络正则化技术，以及如何用TensorFlow实现它们：早期停止，l1和l2正则化，drop out，最大范数正则化和数据增强。

Early Stopping

为避免过度拟合训练集，一个很好的解决方案就是尽早停止训练（在第4章中介绍）：只要在训练集的性能开始下降时中断训练。

与TensorFlow实现方法之一是评估其对设置定期（例如，每50步）验证模型，并保存一个“winner”的快照，如果它优于以前“winner”的快照。 计算自上次“winner”快照保存以来的步数，并在达到某个限制时（例如2000步）中断训练。 然后恢复最后的“winner”快照。

虽然早期停止在实践中运行良好，但是通过将其与其他正则化技术相结合，您通常可以在网络中获得更高的性能。后恢复最后的“winner”快照。

ℓ1and ℓ2Regularization

就像你在第4章中对简单线性模型所做的那样，你可以使用l1和l2正则化约束一个神经网络的连接权重（但通常不是它的偏置）。

使用TensorFlow做到这一点的一种方法是简单地将适当的正则化术语添加到您的成本函数中。 例如，假设您只有一个权重为weight1的隐藏层和一个权重为weight2的输出层，那么您可以像这样应用l1正则化：

我们可以将正则化函数传递给tf.layers.dense（）函数，该函数将使用它来创建计算正则化损失的操作，并将这些操作添加到正则化损失集合中。 开始和上面一样：

```
n_inputs = 28 * 28  # MNIST
n_hidden1 = 300
n_hidden2 = 50
n_outputs = 10

X = tf.placeholder(tf.float32, shape=(None, n_inputs), name="X")
y = tf.placeholder(tf.int64, shape=(None), name="y")
```

接下来，我们将使用Python partial（）函数来避免一遍又一遍地重复相同的参数。 请注意，我们设置了内核正则化参数(正则化函数有l1_regularizer(),l2_regularizer(), andl1_l2_regularizer()

：

```
scale = 0.001
```

```
my_dense_layer = partial(
    tf.layers.dense, activation=tf.nn.relu,
    kernel_regularizer=tf.contrib.layers.l1_regularizer(scale))

with tf.name_scope("dnn"):
```

```
    hidden1 = my_dense_layer(X, n_hidden1, name="hidden1")
    hidden2 = my_dense_layer(hidden1, n_hidden2, name="hidden2")
    logits = my_dense_layer(hidden2, n_outputs, activation=None,
                            name="outputs")
```

该代码创建了一个具有两个隐藏层和一个输出层的神经网络，并且还在图中创建节点以计算与每个层的权重相对应的l1正则化损失。 TensorFlow会自动将这些节点添加到包含所有正则化损失的特殊集合中。 您只需要将这些正则化损失添加到您的整体损失中，如下所示：

接下来，我们必须将正则化损失加到基本损失上：

```
with tf.name_scope("loss"):                                     # not shown in the book
    xentropy = tf.nn.sparse_softmax_cross_entropy_with_logits(  # not shown
        labels=y, logits=logits)                                # not shown
    base_loss = tf.reduce_mean(xentropy, name="avg_xentropy")   # not shown
    reg_losses = tf.get_collection(tf.GraphKeys.REGULARIZATION_LOSSES)
    loss = tf.add_n([base_loss] + reg_losses, name="loss")
```

其余的和往常一样：

```
with tf.name_scope("eval"):
    correct = tf.nn.in_top_k(logits, y, 1)
    accuracy = tf.reduce_mean(tf.cast(correct, tf.float32), name="accuracy")

learning_rate = 0.01

with tf.name_scope("train"):
    optimizer = tf.train.GradientDescentOptimizer(learning_rate)
    training_op = optimizer.minimize(loss)

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

```
n_epochs = 20
batch_size = 200

with tf.Session() as sess:
    init.run()
    for epoch in range(n_epochs):
        for iteration in range(mnist.train.num_examples // batch_size):
            X_batch, y_batch = mnist.train.next_batch(batch_size)
            sess.run(training_op, feed_dict={X: X_batch, y: y_batch})
        accuracy_val = accuracy.eval(feed_dict={X: mnist.test.images,
                                                y: mnist.test.labels})
        print(epoch, "Test accuracy:", accuracy_val)

    save_path = saver.save(sess, "./my_model_final.ckpt")
```

不要忘记把正常化的损失加在你的整体损失上，否则就会被忽略。

Dropout

深度神经网络最流行的正则化技术可以说是drop out。 它由GE Hinton于2012年提出，并在Nitish Srivastava等人的论文中进一步详细描述，并且已被证明是非常成功的：即使是现有技术的神经网络，通过添加drop out。这听起来可能不是很多，但是当一个模型已经具有95％的准确率时，获得2％的准确度提升意味着将误差率降低近40％（从5％误差降至大约3％）。

这是一个相当简单的算法：在每个训练步骤中，每个神经元（包括输入神经元，但不包括输出神经元）都有一个暂时“退出”的概率p，这意味着在这个训练步骤中它将被完全忽略， 在下一步可能会激活（见图11-9）。 超参数p称为丢失率，通常设为50％。 训练后，神经元不会再下降。 这就是全部（除了我们将要讨论的技术细节）。

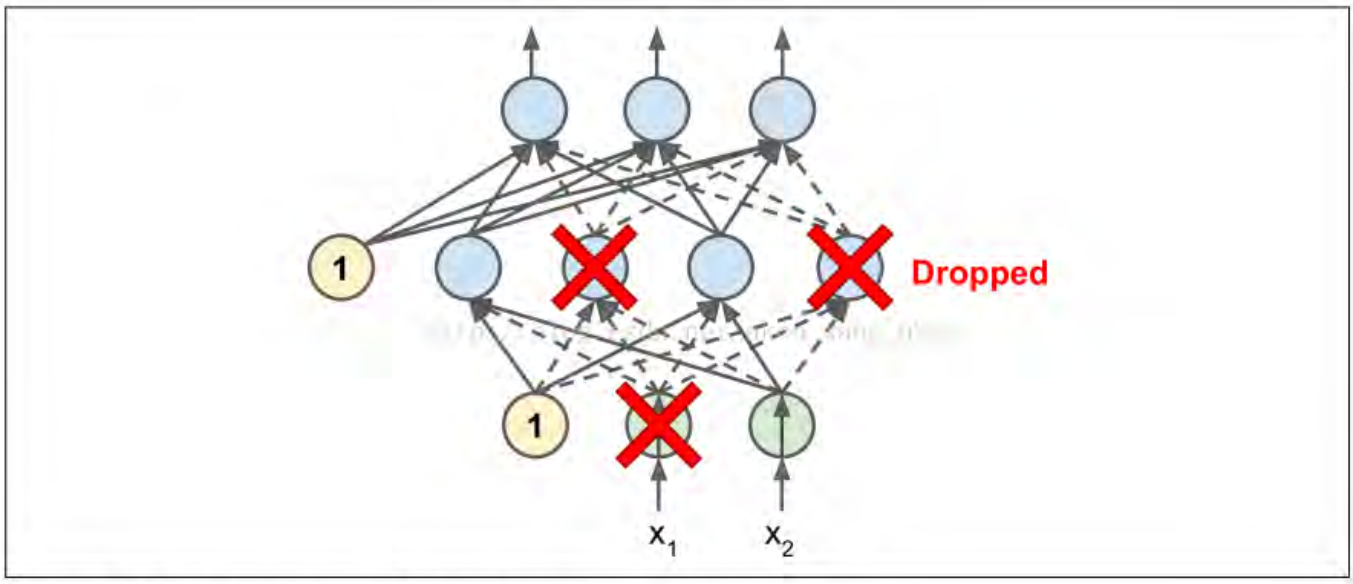


Figure 11-9. Dropout regularization

一开始这个技术是相当粗鲁，这是相当令人惊讶的。如果一个公司的员工每天早上被告知要掷硬币来决定是否上班，公司的表现会不会更好呢？那么，谁知道;也许会！公司显然将被迫适应这样的组织构架;它不能依靠任何一个人填写咖啡机或执行任何其他关键任务，所以这个专业知识将不得不分散在几个人身上。员工必须学会与其他的许多同事合作，而不仅仅是其中的一小部分。该公司将变得更有弹性。如果一个人放弃了，那就没有什么区别了。目前还不清楚这个想法是否真的可以在公司实行，但它确实对于神经网络是可以的。神经元退化训练不能与其相邻的神经元共同适应;他们必须尽可能让自己变得有用。他们也不能过分依赖一些输入神经元;他们必须注意他们的每个输入神经元。他们最终对输入的微小变化会不太敏感。最后，你会得到一个更强大的网络，更好地推广。

了解dropout的另一种方法是认识到每个训练步骤都会产生一个独特的神经网络。 由于每个神经元可以存在或不存在，总共有2 ^ N个可能的网络（其中N是可丢弃神经元的总数）。 这是一个巨大的数字，实际上不可能对同一个神经网络进行两次采样。 一旦你运行了10,000个训练步骤，你基本上已经训练了10,000个不同的神经网络（每个神经网络只有一个训练实例）。 这些神经网络显然不是独立的，因为它们共享许多权重，但是它们都是不同的。 由此产生的神经网络可以看作是所有这些较小的神经网络的平均集合。

有一个小而重要的技术细节。 假设p = 50，在这种情况下，在测试期间，在训练期间神经元将被连接到两倍于（平均）的输入神经元。 为了弥补这个事实，我们需要在训练之后将每个神经元的输入连接权重乘以0.5。 如果我们不这样做，每个神经元的总输入信号大概是网络训练的两倍，而且不太可能表现良好。 更一般地说，我们需要将每个输入连接权重乘以训练后的保持概率（1-p）。 或者，我们可以在训练过程中将每个神经元的输出除以保持概率（这些替代方案并不完全等价，但它们工作得同样好）。

要使用TensorFlow实现压缩，可以简单地将dropout（）函数应用于输入层和每个隐藏层的输出。 在训练过程中，这个功能随机丢弃一些项目（将它们设置为0），并用保留概率来划分剩余项目。 训练结束后，这个功能

什么都不做。 下面的代码将丢失正则化应用于我们的三层神经网络：

注意：本书使用tf.contrib.layers.dropout（）而不是tf.layers.dropout（）（本章写作时不存在）。 现在最好使用tf.layers.dropout（），因为contrib模块中的任何内容都可能会改变或被删除，恕不另行通知。
tf.layers.dropout（）函数几乎与tf.contrib.layers.dropout（）函数相同，只是有一些细微差别。 最重要的是：

- 您必须指定丢失率（率）而不是保持概率（keep_prob），其中rate简单地等于1 - keep_prob
- is_training参数被重命名为training。

```
X = tf.placeholder(tf.float32, shape=(None, n_inputs), name="X")
y = tf.placeholder(tf.int64, shape=(None), name="y")
```

```
training = tf.placeholder_with_default(False, shape=(), name='training')

dropout_rate = 0.5  # == 1 - keep_prob
X_drop = tf.layers.dropout(X, dropout_rate, training=training)

with tf.name_scope("dnn"):
    hidden1 = tf.layers.dense(X_drop, n_hidden1, activation=tf.nn.relu,
                              name="hidden1")
    hidden1_drop = tf.layers.dropout(hidden1, dropout_rate, training=training)
    hidden2 = tf.layers.dense(hidden1_drop, n_hidden2, activation=tf.nn.relu,
                              name="hidden2")
    hidden2_drop = tf.layers.dropout(hidden2, dropout_rate, training=training)
    logits = tf.layers.dense(hidden2_drop, n_outputs, name="outputs")
```

```
with tf.name_scope("loss"):
    xentropy = tf.nn.sparse_softmax_cross_entropy_with_logits(labels=y, logits=logits)
    loss = tf.reduce_mean(xentropy, name="loss")

with tf.name_scope("train"):
    optimizer = tf.train.MomentumOptimizer(learning_rate, momentum=0.9)
    training_op = optimizer.minimize(loss)

with tf.name_scope("eval"):
    correct = tf.nn.in_top_k(logits, y, 1)
    accuracy = tf.reduce_mean(tf.cast(correct, tf.float32))

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

```
n_epochs = 20
batch_size = 50

with tf.Session() as sess:
    init.run()
    for epoch in range(n_epochs):
        for iteration in range(mnist.train.num_examples // batch_size):
            X_batch, y_batch = mnist.train.next_batch(batch_size)
```

```
            sess.run(training_op, feed_dict={training: True, X: X_batch, y: y_batch})
        acc_test = accuracy.eval(feed_dict={X: mnist.test.images, y: mnist.test.labels})
        print(epoch, "Test accuracy:", acc_test)

    save_path = saver.save(sess, "./my_model_final.ckpt")
```

你想在tensorflow.contrib.layers中使用dropout（）函数，而不是tensorflow.nn中的那个。 第一个在不训练的时候关掉（没有操作），这是你想要的，而第二个不是

如果观察到模型过度拟合，则可以增加dropout率（即，减少keep_prob超参数）。 相反，如果模型不适合训练集，则应尝试降低dropout率（即增加keep_prob）。 它也可以帮助增加大层的dropout率，并减少小层的dropout率。

dropout倾向于显着减缓收敛，但通常会导致一个好得多的模型,在适当调整之后。 所以，这通常是值得去付出额外的时间和精力的。

Max-Norm Regularization

另一种在神经网络中非常流行的正则化技术被称为最大范数正则化：对于每个神经元，它约束输入连接的权重w，使得 $\| \mathbf{w} \|_2 \leq r$，其中r是最大范数超参数,$\| \cdot \|_2$是l2范数。

我们通常通过在每个训练步骤之后计算 $\| \mathbf{w} \|_2$来实现这个约束，并且如果需要的话可以剪切W $\left(\mathbf{w} \leftarrow \mathbf{w} \frac{r}{\| \mathbf{w} \|_2} \right)$.

减少r增加了正则化的数量，并有助于减少过度配合。 Maxnorm正则化还可以帮助减轻消失/爆炸梯度问题（如果您不使用批量标准化）。

让我们回到MNIST的简单而简单的神经网络，只有两个隐藏层：

```
n_inputs = 28 * 28
n_hidden1 = 300
n_hidden2 = 50
n_outputs = 10

learning_rate = 0.01
momentum = 0.9

X = tf.placeholder(tf.float32, shape=(None, n_inputs), name="X")
y = tf.placeholder(tf.int64, shape=(None), name="y")

with tf.name_scope("dnn"):
    hidden1 = tf.layers.dense(X, n_hidden1, activation=tf.nn.relu, name="hidden1")
    hidden2 = tf.layers.dense(hidden1, n_hidden2, activation=tf.nn.relu, name="hidden2")
    logits = tf.layers.dense(hidden2, n_outputs, name="outputs")

with tf.name_scope("loss"):
    xentropy = tf.nn.sparse_softmax_cross_entropy_with_logits(labels=y, logits=logits)
    loss = tf.reduce_mean(xentropy, name="loss")
```

```
with tf.name_scope("train"):
    optimizer = tf.train.MomentumOptimizer(learning_rate, momentum)
    training_op = optimizer.minimize(loss)

with tf.name_scope("eval"):
    correct = tf.nn.in_top_k(logits, y, 1)
    accuracy = tf.reduce_mean(tf.cast(correct, tf.float32))
```

接下来，让我们来处理第一个隐藏层的权重，并创建一个操作，使用clip_by_norm（）函数计算剪切后的权重。 然后我们创建一个赋值操作来将权值赋给权值变量：

```
threshold = 1.0
weights = tf.get_default_graph().get_tensor_by_name("hidden1/kernel:0")
clipped_weights = tf.clip_by_norm(weights, clip_norm=threshold, axes=1)
clip_weights = tf.assign(weights, clipped_weights)
```

我们也可以为第二个隐藏层做到这一点：

```
weights2 = tf.get_default_graph().get_tensor_by_name("hidden2/kernel:0")
clipped_weights2 = tf.clip_by_norm(weights2, clip_norm=threshold, axes=1)
clip_weights2 = tf.assign(weights2, clipped_weights2)
```

让我们添加一个初始化器和一个保存器：

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

现在我们可以训练模型。 与往常一样，除了在运行training_op之后，我们运行clip_weights和clip_weights2操作：

```
n_epochs = 20
batch_size = 50
```

```

```

```
with tf.Session() as sess:                                                # not shown in the bo
    init.run()                                                            # not shown
    for epoch in range(n_epochs):                                         # not shown
        for iteration in range(mnist.train.num_examples // batch_size):  # not shown
            X_batch, y_batch = mnist.train.next_batch(batch_size)       # not shown
            sess.run(training_op, feed_dict={X: X_batch, y: y_batch})
            clip_weights.eval()
            clip_weights2.eval()                                          # not shown
        acc_test = accuracy.eval(feed_dict={X: mnist.test.images,       # not shown
                                            y: mnist.test.labels})      # not shown
        print(epoch, "Test accuracy:", acc_test)                          # not shown

    save_path = saver.save(sess, "./my_model_final.ckpt")                # not shown
```

上面的实现很简单，工作正常，但有点麻烦。 更好的方法是定义一个max_norm_regularizer（）函数：

```
def max_norm_regularizer(threshold, axes=1, name="max_norm",
                         collection="max_norm"):
    def max_norm(weights):
        clipped = tf.clip_by_norm(weights, clip_norm=threshold, axes=axes)
        clip_weights = tf.assign(weights, clipped, name=name)
        tf.add_to_collection(collection, clip_weights)
        return None # there is no regularization loss term
    return max_norm
```

然后你可以调用这个函数来得到一个最大规范调节器（与你想要的阈值）。 当你创建一个隐藏层时，你可以将这个正规化器传递给kernel_regularizer参数：

```
n_inputs = 28 * 28
n_hidden1 = 300
n_hidden2 = 50
n_outputs = 10

learning_rate = 0.01
momentum = 0.9

X = tf.placeholder(tf.float32, shape=(None, n_inputs), name="X")
y = tf.placeholder(tf.int64, shape=(None), name="y")
```

```
max_norm_reg = max_norm_regularizer(threshold=1.0)
with tf.name_scope("dnn"):
    hidden1 = tf.layers.dense(X, n_hidden1, activation=tf.nn.relu,
                              kernel_regularizer=max_norm_reg, name="hidden1")
    hidden2 = tf.layers.dense(hidden1, n_hidden2, activation=tf.nn.relu,
                              kernel_regularizer=max_norm_reg, name="hidden2")
    logits = tf.layers.dense(hidden2, n_outputs, name="outputs")
```

```
with tf.name_scope("loss"):
    xentropy = tf.nn.sparse_softmax_cross_entropy_with_logits(labels=y, logits=logits)
    loss = tf.reduce_mean(xentropy, name="loss")

with tf.name_scope("train"):
    optimizer = tf.train.MomentumOptimizer(learning_rate, momentum)
    training_op = optimizer.minimize(loss)

with tf.name_scope("eval"):
    correct = tf.nn.in_top_k(logits, y, 1)
    accuracy = tf.reduce_mean(tf.cast(correct, tf.float32))

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

训练与往常一样，除了每次训练后必须运行重量裁剪操作：

请注意，最大范数正则化不需要在整体损失函数中添加正则化损失项，所以max_norm（）函数返回None。 但是，在每个训练步骤之后，仍需要运行clip_weights操作，因此您需要能够掌握它。 这就是为什么max_norm（）函数将clip_weights节点添加到max-norm剪裁操作的集合中的原因。您需要获取这些裁剪操作并在每个训练步骤后运行它们：

```
n_epochs = 20
batch_size = 50
```

```
clip_all_weights = tf.get_collection("max_norm")

with tf.Session() as sess:
    init.run()
    for epoch in range(n_epochs):
        for iteration in range(mnist.train.num_examples // batch_size):
            X_batch, y_batch = mnist.train.next_batch(batch_size)
            sess.run(training_op, feed_dict={X: X_batch, y: y_batch})
            sess.run(clip_all_weights)
        acc_test = accuracy.eval(feed_dict={X: mnist.test.images,     # not shown in the book
                                            y: mnist.test.labels})    # not shown
        print(epoch, "Test accuracy:", acc_test)                      # not shown

    save_path = saver.save(sess, "./my_model_final.ckpt")             # not shown
```

Data Augmentation (数据扩张)

最后一个正规化技术，数据增强，包括从现有的训练实例中产生新的训练实例，人为地增加了训练集的大小。 这将减少过度拟合，使之成为正规化技术。 诀窍是生成逼真的训练实例; 理想情况下，一个人不应该能够分辨出哪些是生成的，哪些不是生成的。 而且，简单地加白噪声也无济于事。 你应用的修改应该是可以学习的（白噪声不是）。

例如，如果您的模型是为了分类蘑菇图片，您可以稍微移动，旋转和调整训练集中的每个图片的大小，并将结果图片添加到训练集（见图11-10）。 这迫使模型更能容忍图片中蘑菇的位置，方向和大小。 如果您希望模型对光照条件更加宽容，则可以类似地生成具有各种对比度的许多图像。 假设蘑菇是对称的，你也可以水平翻转图片。 通过结合这些转换，可以大大增加训练集的大小。

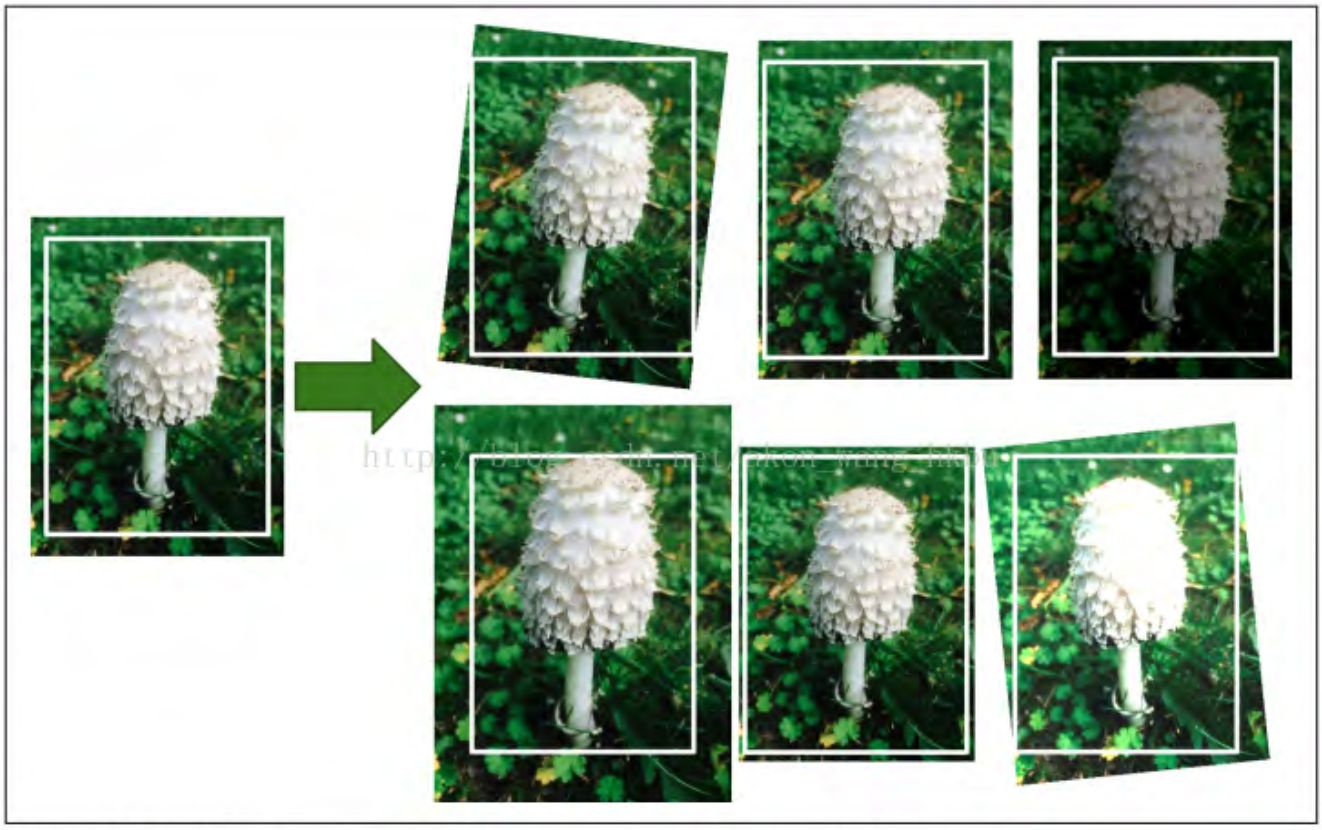


Figure 11-10. Generating new training instances from existing ones

在培训期间通常优先生成训练实例，而不是浪费存储空间和网络带宽。 TensorFlow提供了多种图像处理操作，例如移调（shift），旋转，调整大小，翻转和裁剪，以及调整亮度，对比度，饱和度和色调（请参阅API文档以获取更多详细信息）。 这可以很容易地为图像数据集实现数据增强。

训练非常深的神经网络的另一个强大的技术是添加跳过连接（跳过连接是将层的输入添加到更高层的输出时）。 当我们谈论深度残差网络时，我们将在第13章中探讨这个想法。

Practical Guidelines (实际指导)

在本章中，我们已经涵盖了很多技术，你可能想知道应该使用哪些技术。 表11-2中的配置在大多数情况下都能正常工作。

Table 11-2. Default DNN configuration

Initialization	**He initialization**
Activation function	**ELU**
Normalization	**Batch Normalization**
Regularization	**Dropout**
Optimizer	**Adam**
Learning rate schedule	**None**

当然，如果你能找到解决类似问题的方法，你应该尝试重用预训练的神经网络的一部分。

这个默认配置可能需要调整：

- 如果你找不到一个好的学习速度（收敛速度太慢，所以你增加了训练速度，现在收敛速度很快，但是网络的准确性不是最理想的），那么你可以尝试添加一个学习计划，如指数衰减。
- 如果你的训练集太小，你可以实现数据增强。
- 如果你需要一个稀疏的模型，你可以添加一个正则化到混合（并可以选择在训练后将微小的权重归零）。 如果您需要更稀疏的模型，您可以尝试使用FTRL而不是Adam优化以及l1正则化。
- 如果在运行时需要快速模型，则可能需要删除批处理标准化，并可能用leakyReLU替换ELU激活函数。 有一个稀疏的模型也将有所帮助。

有了这些指导方针，你现在已经准备好训练非常深的网络 - 好吧，如果你非常有耐心的话，那就是！ 如果使用单台机器，则可能需要等待几天甚至几个月才能完成培训。 在下一章中，我们将讨论如何使用分布式TensorFlow在许多服务器和GPU上训练和运行模型。

十三、Convolutional Neural Networks

本篇文章是个人翻译的,如有商业用途,请通知本人谢谢.

Convolutional Layer

CNN最重要的组成部分是卷积层：第一卷积层中的神经元不是连接到输入图像中的每一个像素（就像它们在前面的章节中那样），而是仅仅连接到它们的局部感受野中的像素（参见图13-2）。 进而，第二卷积层中的每个神经元只与位于第一层中的小矩形内的神经元连接。 这种架构允许网络专注于第一隐藏层中的低级特征，然后将其组装成下一隐藏层中的高级特征，等等。 这种层次结构在现实世界的图像中是很常见的，这也是CNN在图像识别方面效果很好的原因之一。

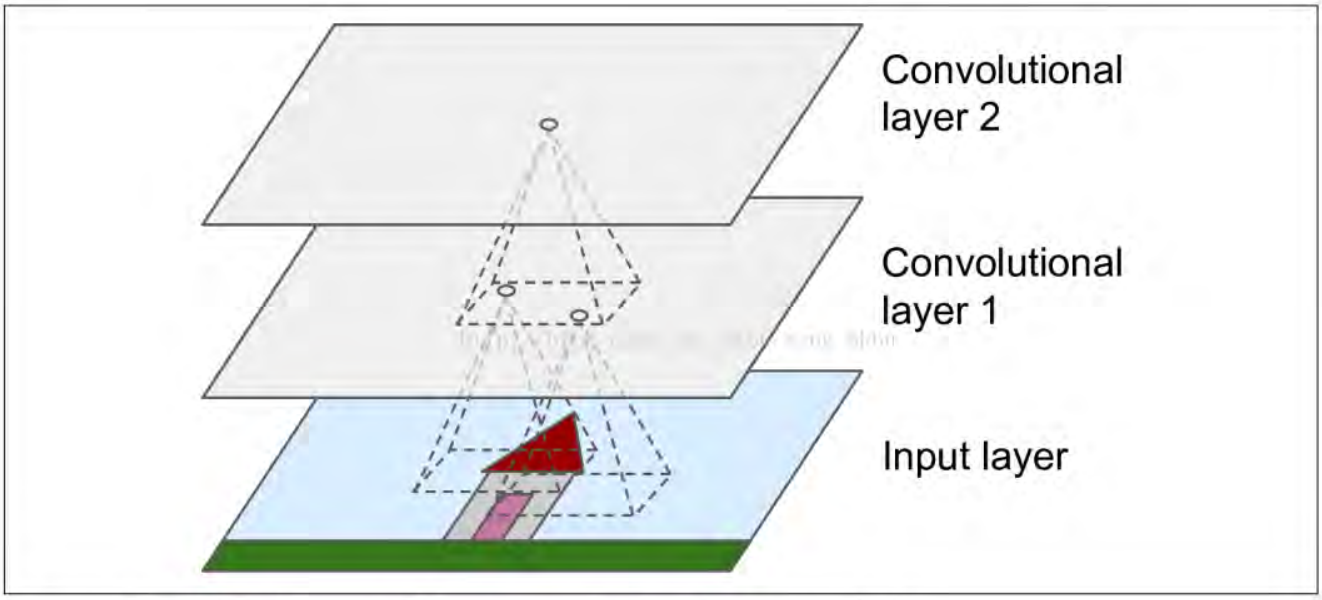


Figure 13-2. CNN layers with rectangular local receptive fields

到目前为止，我们所看到的所有多层神经网络都有由一长串神经元组成的层，在输入到神经网络之前我们必须将输入图像压缩成1D。 现在，每个图层都以2D表示，这使得神经元与其相应的输入进行匹配变得更加容易。

位于给定层的第i行第j列的神经元连接到位于前一层中的神经元的输出的第i行到第$i + f_h - 1$行，第j列到第$j + f_w - 1$列.fh和fw是局部感受野的高度和宽度（见图13-3）。 为了使图层具有与前一图层相同的高度和宽度，通常在输入周围添加零，如图所示。 这被称为零填充.

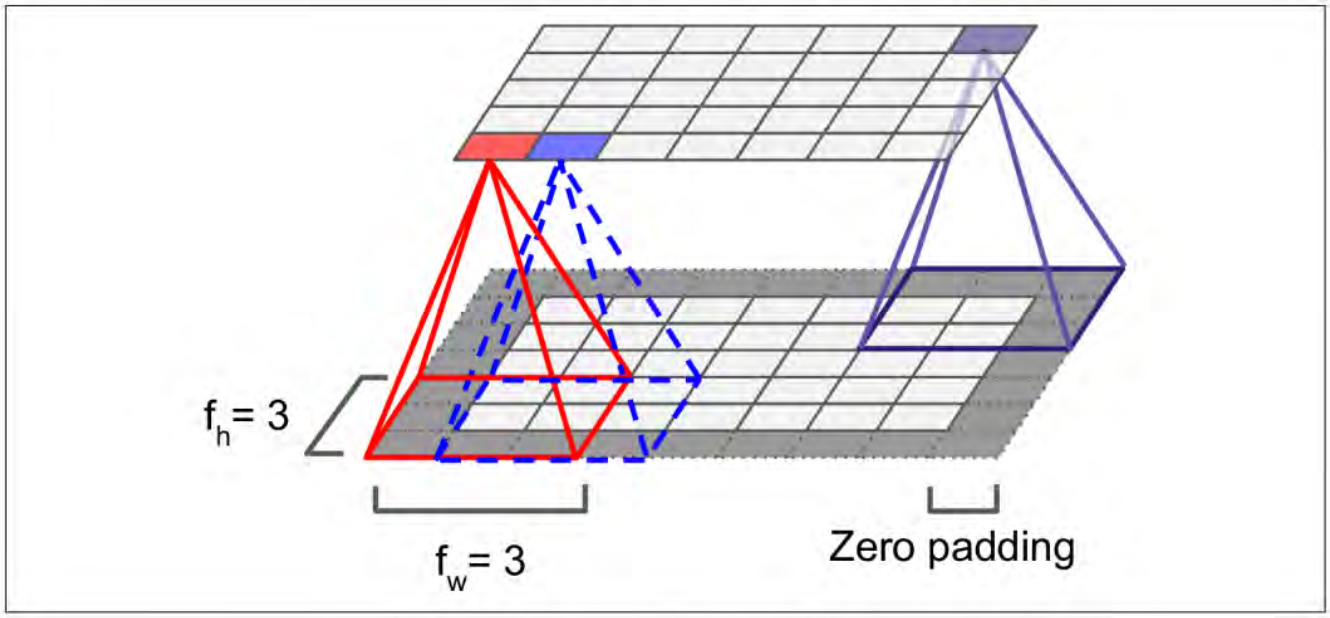


Figure 13-3. Connections between layers and zero padding

如图13-4所示，通过将局部感受野隔开，还可以将较大的输入层连接到更小的层。 两个连续的感受野之间的距离被称为步幅。 在图中，一个5×7的输入层（加零填充）连接到一个3×4层，使用3×3的卷积核和一个步幅为2（在这个例子中，步幅在两个方向是相同的，但是它并不一定总是如此）。 位于上层第i行第j列的神经元与位于前一层中的神经元的输出连接的第$i \times s_h$至$i \times s_h + f_h - 1$行,第$j \times sw + fw - 1$列，Sh和Sw是垂直和水平的步幅。

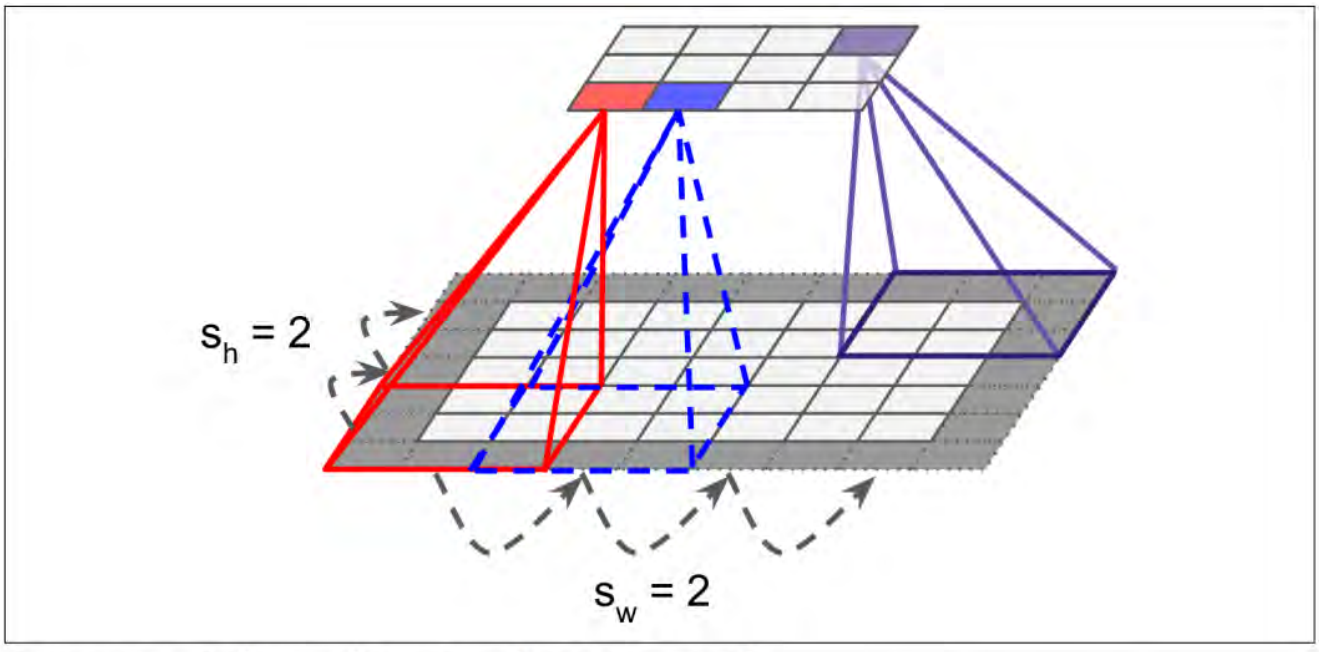


Figure 13-4. Reducing dimensionality using a stride

Filters(卷积核)

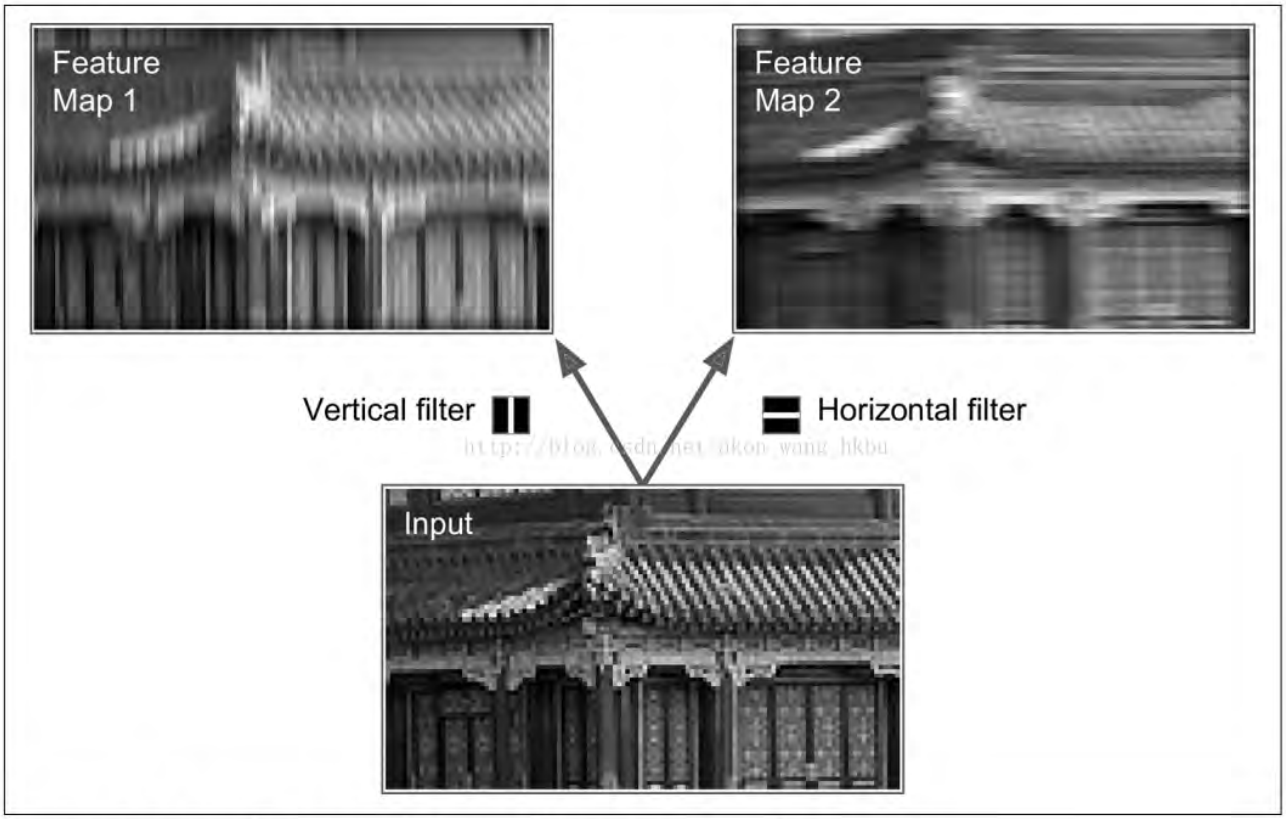


Figure 13-5. Applying two different filters to get two feature maps

神经元的权重可以表示为局部感受野大小的小图像。 例如，图13-5显示了两个可能的权重集，称为filters（或卷积核）。第一个表示为中间有一条垂直的白线的黑色正方形（除了中间一列外，这是一个充满0的7×7矩阵，除了中央垂直线是1）。 使用这些权重的神经元会忽略他们的局部感受野的一切，除了中央垂直线（因为所有的inputs将得到0乘，除位于中央垂直线的）。第二个卷积核是一个黑色的正方形，中间有一条水平的白线。再一次，使用这些权重的神经元将忽略除了中心水平线之外的局部感受野中的一切。

现在，如果一个图层中的所有神经元都使用相同的垂直线滤波器（以及相同的偏置项），并且将网络输入到图13-5（底部图像）中所示的输入图像，则该图层将输出左上图像。 请注意，垂直的白线得到增强，其余的变得模糊。 类似地，如果所有的神经元都使用水平线滤波器，右上角的图像就是你所得到的。 注意到水平的白线得到增强，其余的则被模糊了。因此，使用相同滤波器的一个充满神经元的图层将为您提供一个特征图，该特征图突出显示图像中与滤波器最相似的区域。 在训练过程中，CNN为其任务找到最有用的过滤器，并学习将它们组合成更复杂的模式（例如，交叉是图像中垂直过滤器和水平过滤器都活动的区域）。

Stacking Multiple Feature Maps(叠加的多个特征图)

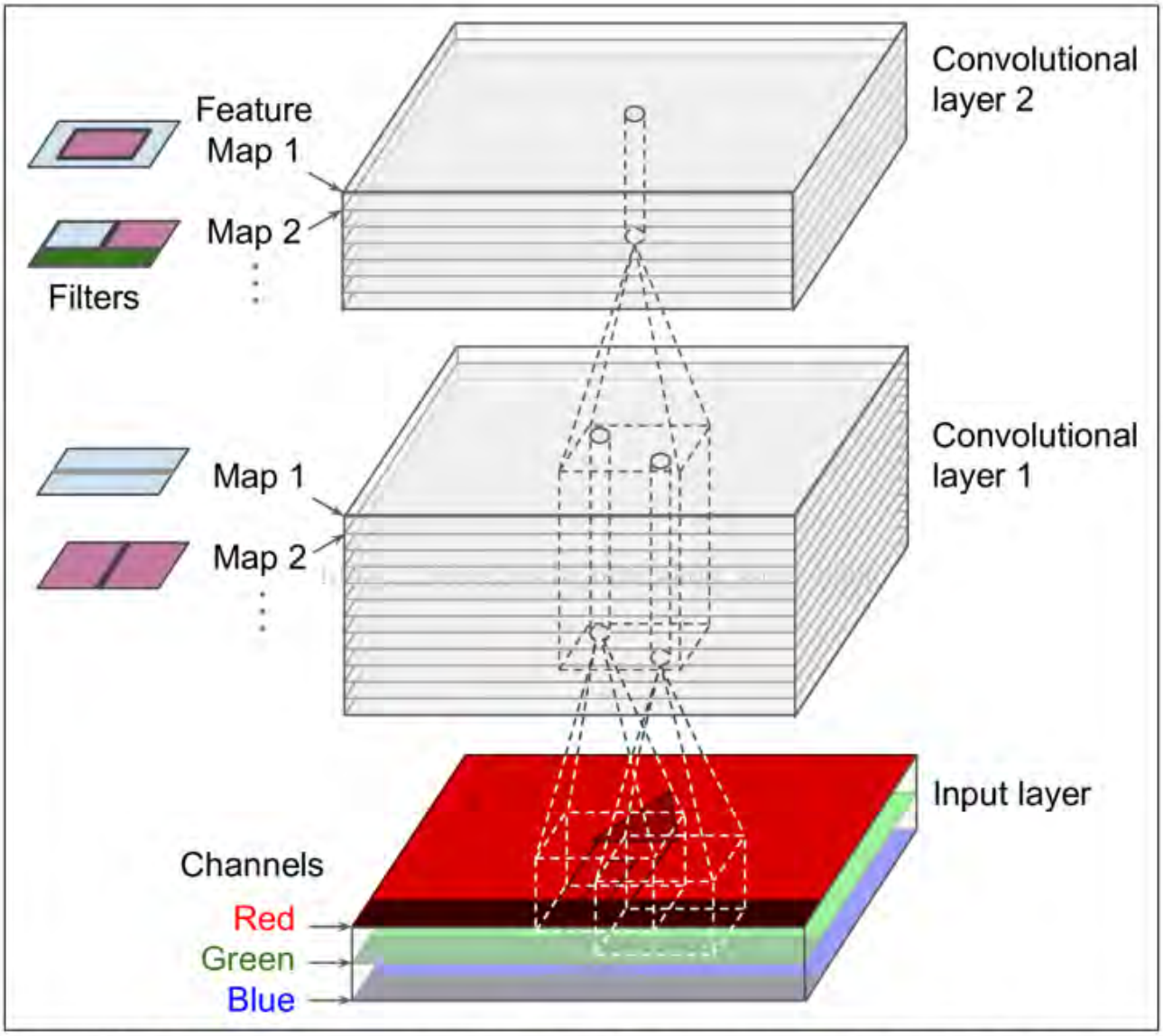


Figure 13-6. Convolution layers with multiple feature maps, and images with three channels

到目前为止，为了简单起见，我们已经将每个卷积层表示为一个薄的二维层，但是实际上它是由几个相同大小的特征图组成的，所以它在3D中被更精确地表示（见图13-6）。 在一个特征映射中，所有神经元共享相同的参数(权值共享)（权重和偏置），但是不同的特征映射可能具有不同的参数。 神经元的感受野与前面描述的相同，但是它延伸到所有先前的层的特征图。 简而言之，卷积层同时对其输入应用多个滤波器，使其能够检测输入中的任何位置的多个特征。

事实上，特征地图中的所有神经元共享相同的参数会显着减少模型中的参数数量，但最重要的是，一旦CNN学会识别一个位置的模式，就可以在任何其他位置识别它。 相比之下，一旦一个常规DNN学会识别一个位置的模式，它只能在该特定位置识别它。

而且，输入图像也由多个子图层组成：每个颜色通道一个。 通常有三种：红色，绿色和蓝色（RGB）。 灰度图像只有一个通道，但是一些图像可能更多 - 例如捕捉额外光频（如红外线）的卫星图像。

具体地，位于给定卷积层L中的特征映射k的i行，j列中的神经元连接到前一层(L-1)位于

$$i \times s_w \text{ to } i \times s_w + f_w - 1$$ 行, $$j \times s_h \text{ to } j \times s_h + f_h - 1$$ 列的神经元的输出.请注意，位于同一行第i列和第j列但位于不同特征映射中的所有神经元都连接到上一层中完全相同神经元的输出。

公式13-1在一个总结前面解释的大的数学公式：它展示了如何计算卷积层中给定神经元的输出。它是计算所有投入的加权总并且加上偏置。

Equation 13-1. Computing the output of a neuron in a convolutional layer

$$z_{i,j,k} = b_k + \sum_{u=1}^{f_h} \sum_{v=1}^{f_w} \sum_{k'=1}^{f_{n'}} x_{i',j',k'} \cdot w_{u,v,k',k} \quad \text{with} \begin{cases} i' = u \cdot s_h + f_h - 1 \\ j' = v \cdot s_w + f_w - 1 \end{cases}$$

- $z_{i,j,k}$ 是卷积层（L层）特征映射k中位于第i行第j列的神经元的输出.
- 如前所述， s_h 和 s_w 是垂直和水平的步幅，f_h和f_w 是感受野的高度和宽度，$f_{n'}$ 是前一层（第l - 1层）的特征图的数量。
- $x_{i',j',k'}$ 是位于层L-1，i'行，j'列，特征图k'（或者如果前一层是输入层的通道k'）的神经元的输出。
- b_k 是特征映射k的偏置项（在L层中）。 您可以将其视为调整特征映射k的整体亮度的旋钮。
- $w_{u,v,k',k}$是层L的特征映射k中的任何神经元与位于行u，列v（相对于神经元的感受野）的输入之间的连接权重，以及特征映射k'。

TensorFlow Implementation (TensorFlow实现)
在张量流中，3D张量的每个输入图像通常被表示为形状 `[height, width, channels]`。一个小批量被表示为四维张量的形状 `[mini-batch size, height, width, channels]`。卷积层的权重表示为形状的四维张量 $[f_h, f_w, f_n, f_{n'}]$。 卷积层的偏差项简单地表示为一维形状的张量 $[f_n]$。

我们来看一个简单的例子。 下面的代码使用Scikit-Learn的load_sample_images（）（加载两个彩色图像，一个华人庙宇，另一个是一朵花）加载两个样本图像。 然后创建两个7×7的卷积核（一个中间是垂直的白线，另一个是水平的白线），并将他们应用到两张图形中,使用TensorFlow的conv2d（）函数构建的卷积图层（使用零填充 且步幅为2)。 最后，绘制其中一个结果特征图（类似于图13-5中的右上图）。

```
from sklearn.datasets import load_sample_image
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf


if __name__ == '__main__':


    # Load sample images
    china = load_sample_image("china.jpg")
```

```
flower = load_sample_image("flower.jpg")
dataset = np.array([china, flower], dtype=np.float32)
batch_size, height, width, channels = dataset.shape

# Create 2 filters
filters = np.zeros(shape=(7, 7, channels, 2), dtype=np.float32)
filters[:, 3, :, 0] = 1  # vertical line
filters[3, :, :, 1] = 1  # horizontal line

# Create a graph with input X plus a convolutional layer applying the 2 filters
X = tf.placeholder(tf.float32, shape=(None, height, width, channels))
convolution = tf.nn.conv2d(X, filters, strides=[1,2,2,1], padding="SAME")

with tf.Session() as sess:
    output = sess.run(convolution, feed_dict={X: dataset})

plt.imshow(output[0, :, :, 1], cmap="gray") # plot 1st image's 2nd feature map
plt.show()
```

大部分代码是不言而喻的，但conv2d（）行值得解释一下：

- X是输入最小批量（4D张量，如前所述）
- 卷积核是应用的一组卷积核（也是一个4D张量，如前所述）。
- 步幅是一个四元素的一维数组，其中两个中心元素是垂直和水平的步幅（sh和sw）。 第一个和最后一个元素现在必须等于1.他们可能有一天会被用来指定批量步长（跳过一些实例）和频道步幅（跳过上一层的特征映射或通道）。
- padding必须是“VALID”或“SAME”：

- 如果设置为“VALID”，卷积层不使用零填充，并且可能会忽略输入图像底部和右侧的某些行和列，具体取决于步幅，如图13-7所示（为简单起见， 这里只显示水平尺寸，当然，垂直尺寸也适用相同的逻辑）
- 如果设置为“SAME”，则卷积层在必要时使用零填充。 在这种情况下，输出神经元的数量等于输入神经元的数量除以该步幅，向上舍入（在这个例子中，ceil（13/5）= 3）。 然后在输入周围尽可能均匀地添加零。

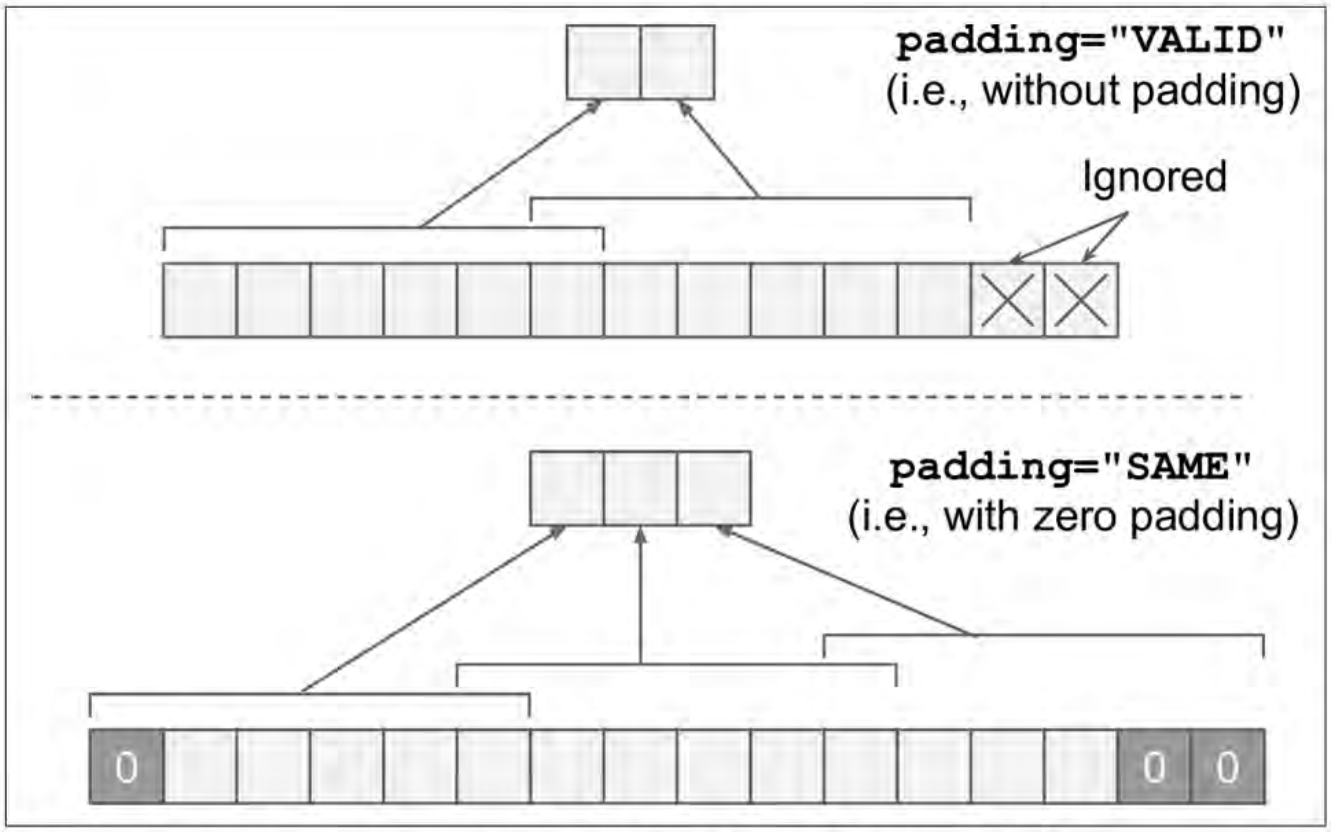


Figure 13-7. Padding options—input width: 13, filter width: 6, stride: 5

不幸的是，卷积图层有很多超参数：你必须选择卷积核的数量，高度和宽度，步幅和填充类型。 与往常一样，您可以使用交叉验证来查找正确的超参数值，但这非常耗时。 稍后我们将讨论常见的CNN体系结构，以便让您了解超参数值在实践中的最佳工作方式。

Memory Requirements (内存需求)
CNN的另一个问题是卷积层需要大量的RAM，特别是在训练期间，因为反向传播的反向传递需要在正向传递期间计算的所有中间值。

例如，考虑具有5×5滤波器的卷积层，输出200个尺寸为150×100的特征图，步长为1，使用SAME padding。 如果输入是150×100 RGB图像（三个通道），则参数的数量是（5×5×3 + 1）×200 = 15,200（+1对应于偏置项），这跟全连接层比较是相当小的.(具有150×100神经元的全连接层，每个连接到所有150×100×3输入，将具有150 ^ 2×100 ^ 2×3 = 675,000,000个参数！)然而，200个特征图中的每一个包含150×100个神经元，并且这些神经元中的每一个都需要计算其5×5×3 = 75个输入的权重和：总共2.25亿次浮点乘法。不像完全连接的层那么糟糕，但仍然是计算密集型的。 而且，如果使用32位浮点数来表示特征映射，则卷积层的输出将占用RAM的200×150×100×32 = 9600万位（大约11.4MB）。 这只是一个例子！ 如果培训批次包含100个实例，则该层将占用超过1 GB的RAM！

在推理过程中（即对新实例进行预测时），一旦下一层计算完毕，一层所占用的RAM就可以被释放，因此只需要两个连续层所需的RAM数量。 但是在训练期间，在正向传递期间计算的所有内容都需要被保留用于反向传递，所以所需的RAM量（至少）是所有层所需的RAM总量。
如果由于内存不足错误导致培训崩溃，则可以尝试减少小批量大小。 或者，您可以尝试使用步幅降低维度，或者删除几个图层。 或者你可以尝试使用16位浮点数而不是32位浮点数。 或者你可以在多个设备上分发CNN。
Pooling Layer （池化层）

一旦你理解了卷积层是如何工作的，池化层很容易掌握。 他们的目标是对输入图像进行二次抽样（即收缩）以减少计算负担，内存使用量和参数数量（从而限制过度拟合的风险）。 减少输入图像的大小也使得神经网络容忍一点点的图像移位（位置不变）。

就像在卷积图层中一样，池化层中的每个神经元都连接到前一层中有限数量的神经元的输出，位于一个小的矩形感受域内。 您必须像以前一样定义其大小，跨度和填充类型。 但是，汇集的神经元没有权重; 它所做的只是使用聚合函数（如最大值或平均值）来聚合输入。 图13-8显示了最大池层，这是最常见的池化类型。 在这个例子中，我们使用一个2×2的内核，步幅为2，没有填充。 请注意，只有每个内核中的最大输入值才会进入下一层。 其他输入被丢弃。

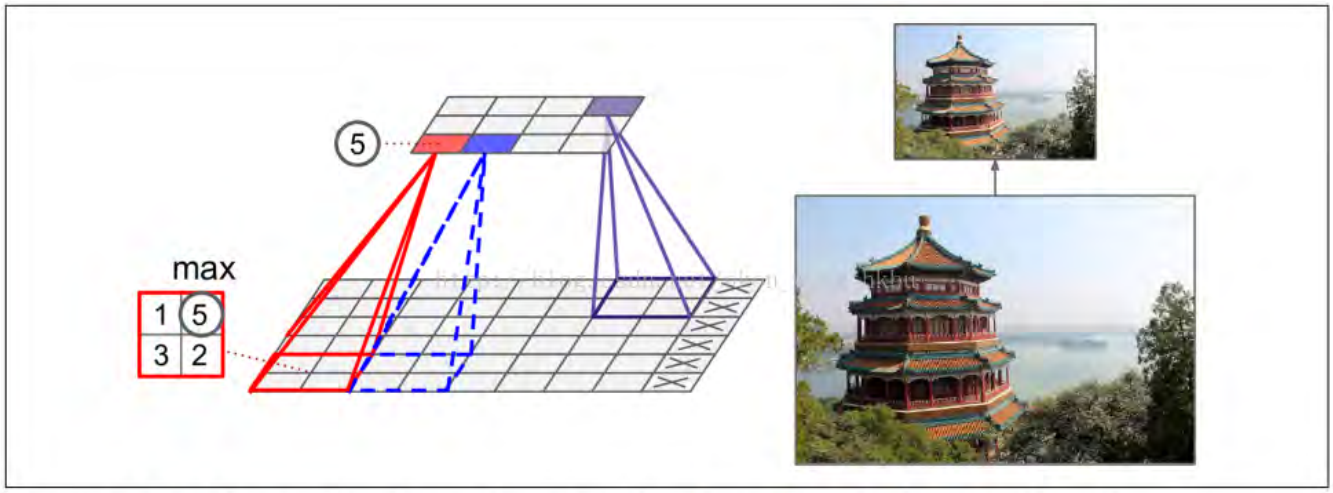


Figure 13-8. Max pooling layer (2 × 2 pooling kernel, stride 2, no padding)

这显然是一个非常具有破坏性的层次：即使只有一个2×2的内核和一个2的步幅，输出在两个方向上都会减小两倍（所以它的面积将减少四倍），一下减少了75%的输入值。

池化层通常独立于每个输入通道工作，因此输出深度与输入深度相同。 接下来可以看到，在这种情况下，图像的空间维度（高度和宽度）保持不变，但是通道数目可以减少。

在TensorFlow中实现一个最大池层是非常容易的。 以下代码使用2×2内核创建最大池化层，跨度为2，没有填充，然后将其应用于数据集中的所有图像：

```
import numpy as np
from sklearn.datasets import load_sample_image
import tensorflow as tf
import matplotlib.pyplot as plt

china = load_sample_image("china.jpg")
flower = load_sample_image("flower.jpg")

dataset = np.array([china, flower], dtype=np.float32)
batch_size, height, width, channels = dataset.shape

# Create 2 filters
filters = np.zeros(shape=(7, 7, channels, 2), dtype=np.float32)
filters[:, 3, :, 0] = 1  # vertical line
filters[3, :, :, 1] = 1  # horizontal line

X = tf.placeholder(tf.float32, shape=(None, height, width, channels))
max_pool = tf.nn.max_pool(X, ksize=[1,2,2,1], strides=[1,2,2,1],padding="VALID")

with tf.Session() as sess:
    output = sess.run(max_pool, feed_dict={X: dataset})

plt.imshow(output[0].astype(np.uint8))  # plot the output for the 1st image
plt.show()
```

ksize参数包含沿输入张量的所有四个维度的内核形状：[批量大小，高度，宽度，通道]。 TensorFlow目前不支持在多个实例上合并，因此ksize的第一个元素必须等于1.此外，它不支持在空间维度（高度和宽度）和深度维度上合并，因此ksize [1] 和ksize [2]都必须等于1，否则ksize [3]必须等于1。

要创建一个平均池层，只需使用avg_pool（）函数而不是max_pool（）。
现在你知道所有的构建模块来创建一个卷积神经网络。 我们来看看如何组装它们。
CNN Architectures
典型的CNN体系结构将一些卷积层（每一个通常跟着一个ReLU层），然后是一个合并层，然后是另外几个卷积层（+ ReLU），然后是另一个池化层，等等。 随着网络的进展，图像变得越来越小，但是由于卷积层的缘故，图像通常也会越来越深（即更多的特征图）（见图13-9）。 在堆栈的顶部，添加由几个完全连接的层（+ ReLU）组成的常规前馈神经网络，并且最终层输出预测（例如，输出估计类别概率的softmax层）。

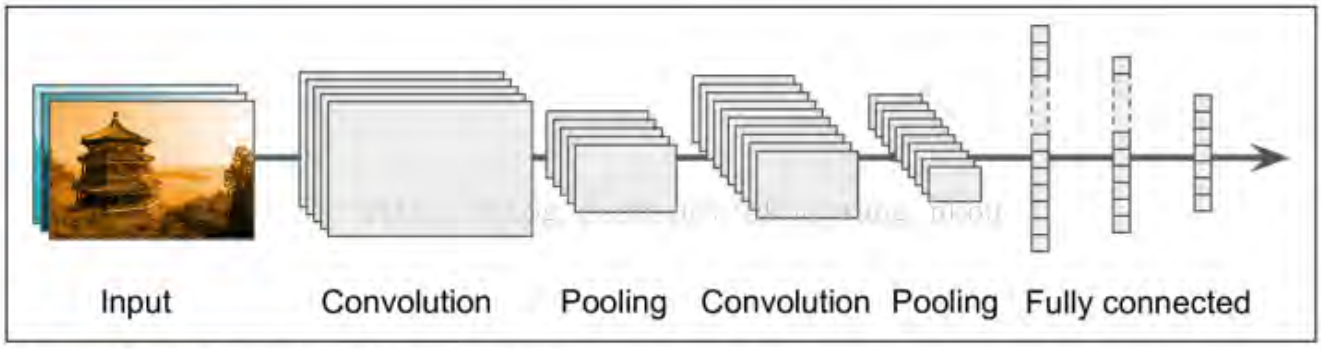


Figure 13-9. Typical CNN architecture

一个常见的错误是使用太大的卷积内核。 通常可以通过将两个3×3内核堆叠在一起来获得与9×9内核相同的效果，计算量更少。
多年来，这种基础架构的变体已经被开发出来，导致了该领域的惊人进步。 这种进步的一个很好的衡量标准是比赛中的错误率，比如ILSVRC ImageNet的挑战。 在这个比赛中，图像分类的五大误差率在五年内从26％下降到仅仅3％以上。 前五位错误率是系统前5位预测未包含正确答案的测试图像的数量。 图像很大（256像素高），有1000个类，其中一些非常微妙（尝试区分120个品种）。 查看获奖作品的演变是了解CNN如何工作的好方法。
我们先来看看经典的LeNet-5架构（1998年），然后是ILSVRC挑战赛的三名获胜者AlexNet（2012），GoogLeNet（2014）和ResNet（2015）
其他视觉任务
在其他视觉任务中，如物体检测和定位以及图像分割，也取得了惊人的进展。 在物体检测和定位中，神经网络通常输出图像中各种物体周围的一系列边界框。 例如，参见Maxine Oquab等人的2015年论文，该论文为每个客体类别输出热图，或者Russell Stewart等人的2015年论文，该论文结合使用CNN来检测人脸，并使用递归神经网络来输出 围绕它们的一系列边界框。 在图像分割中，网络输出图像（通常与输入大小相同），其中每个像素指示相应输入像素所属的对象的类别。 例如，查看Evan Shelhamer等人的2016年论文。
LeNet-5

LeNet-5架构也许是最广为人知的CNN架构。 如前所述，它是由Yann LeCun于1998年创建的，广泛用于手写数字识别（MNIST）。 它由表13-1所示的层组成。

Table 13-1. LeNet-5 architecture

Layer	Type	Maps	Size	Kernel size	Stride	Activation
Out	Fully Connected	–	10	–	–	RBF
F6	Fully Connected	–	84	–	–	tanh
C5	Convolution	120	1 × 1	5 × 5	1	tanh
S4	Avg Pooling	16	5 × 5	2 × 2	2	tanh
C3	Convolution	16	10 × 10	5 × 5	1	tanh
S2	Avg Pooling	6	14 × 14	2 × 2	2	tanh
C1	Convolution	6	28 × 28	5 × 5	1	tanh
In	Input	1	32 × 32	–	–	–

这里有一些额外的细节要注意：

- MNIST图像是28×28像素，但是它们被补零到32×32像素，并且在被喂到网络之前被归一化。 网络的其余部分不使用任何填充，这就是为什么随着图像在网络中的进展，大小不断缩小。
- 平均池化层比平常稍微复杂一些：每个神经元计算输入的平均值，然后将结果乘以一个可学习的系数（每个特征图一个），并添加一个可学习的偏差项（每个特征图一个），然后最后应用激活函数。
- C3地图中的大多数神经元仅在三个或四个S2地图（而不是全部六个S2地图）中连接到神经元。 有关详细信息，请参阅原始纸张中的表1。
- 输出层有点特殊：每个神经元不是计算输入和权向量的点积，而是输出其输入向量和其权向量之间的欧几里德距离的平方。 每个输出测量图像属于特定数字类别的多少。 交叉熵损失函数现在是首选，因为它更多地惩罚不好的预测，产生更大的梯度，从而更快地收敛。

Yann LeCun的网站（“LENET”部分）展示了LeNet-5分类数字的很好的演示。

AlexNet

AlexNet CNN架构赢得了2012年的ImageNet ILSVRC挑战赛：它实现了前5%的17%的错误率，而且后一位错误率有26%！ 它由Alex Krizhevsky（因此而得名），Ilya Sutskever和Geoffrey Hinton开发。 它与LeNet-5非常相似，只是更大更深，它是第一个将卷积层直接堆叠在一起，而不是在每个卷积层顶部堆叠一个池化层。 表13-2介绍了这种架构。

Table 13-2. AlexNet architecture

Layer	Type	Maps	Size	Kernel size	Stride	Padding	Activation
Out	Fully Connected	–	1,000	–	–	–	Softmax
F9	Fully Connected	–	4,096	–	–	–	ReLU
F8	Fully Connected	–	4,096	–	–	–	ReLU
C7	Convolution	256	13 × 13	3 × 3	1	SAME	ReLU
C6	Convolution	384	13 × 13	3 × 3	1	SAME	ReLU
C5	Convolution	384	13 × 13	3 × 3	1	SAME	ReLU
S4	Max Pooling	256	13 × 13	3 × 3	2	VALID	–
C3	Convolution	256	27 × 27	5 × 5	1	SAME	ReLU
S2	Max Pooling	96	27 × 27	3 × 3	2	VALID	–
C1	Convolution	96	55 × 55	11 × 11	4	SAME	ReLU
In	Input	3 (RGB)	224 × 224	–	–	–	–

为了减少过拟合，作者使用了前面章节中讨论的两种正则化技术：首先他们在训练期间将丢失率（dropout率为50％）应用于层F8和F9的输出。其次，他们通过随机移动训练图像的各种偏移，水平翻转和改变照明条件来执行数据增强。

AlexNet还在层C1和C3的ReLU步骤之后立即使用竞争标准化步骤，称为局部响应标准化。 这种归一化形式使得在相同的位置的神经元被最强烈的激活但是在相邻的特征图中抑制神经元（在生物神经元中观察到了这种竞争激活）。 这鼓励不同的特征映射专业化，推它们分开，并迫使他们探索更广泛的功能，最终提升泛化能力。公式13-2显示了如何应用LRN。

Equation 13-2. Local response normalization

$$b_i = a_i\left(k + \alpha \sum_{j=j_{\text{low}}}^{j_{\text{high}}} a_j^2\right)^{-\beta} \quad \text{with} \begin{cases} j_{\text{high}} = \min\left(i + \frac{r}{2}, f_n - 1\right) \\ j_{\text{low}} = \max\left(0, i - \frac{r}{2}\right) \end{cases}$$

- b_i 是位于特征映射i的神经元的标准化输出，在某行u和列v（注意，在这个等式中我们只考虑位于这个行和列的神经元，所以u和v没有显示）。
- a_i 是在ReLU步骤之后，但在归一化之前的那个神经元的激活。
- k，α，β和r是超参数。 k称为偏置，r称为深度半径。
- f_n 是特征图的数量。

例如，如果r = 2且神经元具有强激活，则将抑制位于其上下的特征图中的神经元的激活。

在AlexNet中，超参数设置如下：r = 2，α= 0.00002，β= 0.75，k = 1。这个步骤可以使用TensorFlow的local_response_normalization（）操作来实现。
AlexNet的一个名为ZF Net的变体由Matthew Zeiler和Rob Fergus开发，赢得了2013年ILSVRC的挑战。 它基本上是AlexNet的一些调整的超参数（特征映射的数量，内核大小，步幅等）。
GoogLeNet
GoogLeNet架构是由Christian Szegedy等人开发的。 来自Google Research，通过低于前5名7％的错误率，赢得了ILSVRC 2014的挑战赛。 这个伟大的表现很大程度上来自《盗梦空间》模块，它是比以前CNN更深的网络（见图13-11）。 这是通过称为初始模块的子网络实现的，这使得GoogLeNet比以前的架构更有效地使用参数：实际上，GoogLeNet的参数比AlexNet少了10倍（约600万而不是6000万）。

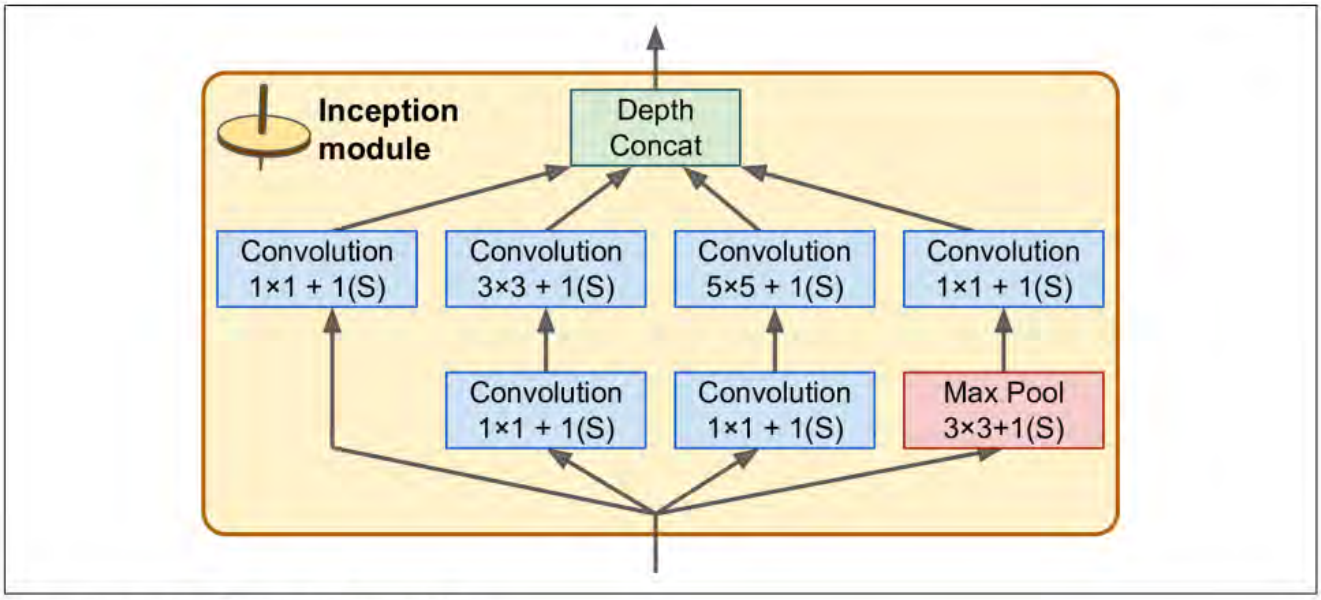


Figure 13-10. Inception module

盗梦空间模块的架构如图13-10所示。
启动模块的架构如图13-10所示。 符号“3×3 + 2（S）”表示该层使用3×3内核，步幅2和相同填充。 输入信号首先被复制并馈送到四个不同的层。 所有卷积层都使用ReLU激活功能。 请注意，第二组卷积层使用不同的内核大小（1×1,3×3和5×5），允许它们以不同的比例捕获图案。 还要注意，每一层都使用1和SAME填充的跨度（即使是最大的池化层），所以它们的输出全都具有与其输入相同的高度和宽度。 这使得有可能连接最终深度连续层中沿着深度维度的所有输出（即，堆叠来自所有四个顶部卷积层的特征映射）。 这个连接层可以在TensorFlow中使用concat（）操作实现，其中axis = 3（axis 3是深度）。
您可能想知道为什么初始模块具有1×1内核的卷积层。 当然这些图层不能捕获任何功能，因为他们一次只能看一个像素？ 实际上，这些层次有两个目的：
首先，它们被配置为输出比输入少得多的特征图，所以它们作为瓶颈层，意味着它们降低了维度。 在3×3和5×5卷积之前，这是特别有用的，因为这些在计算上是非常耗费内存的层。
简而言之，您可以将整个“盗梦空间”模块视为类固醇卷积层，能够输出捕捉各种尺度复杂模式的特征映射。
每个卷积层的卷积核的数量是一个超参数。 不幸的是，这意味着你有六个超参数来调整你添加的每个初始层。

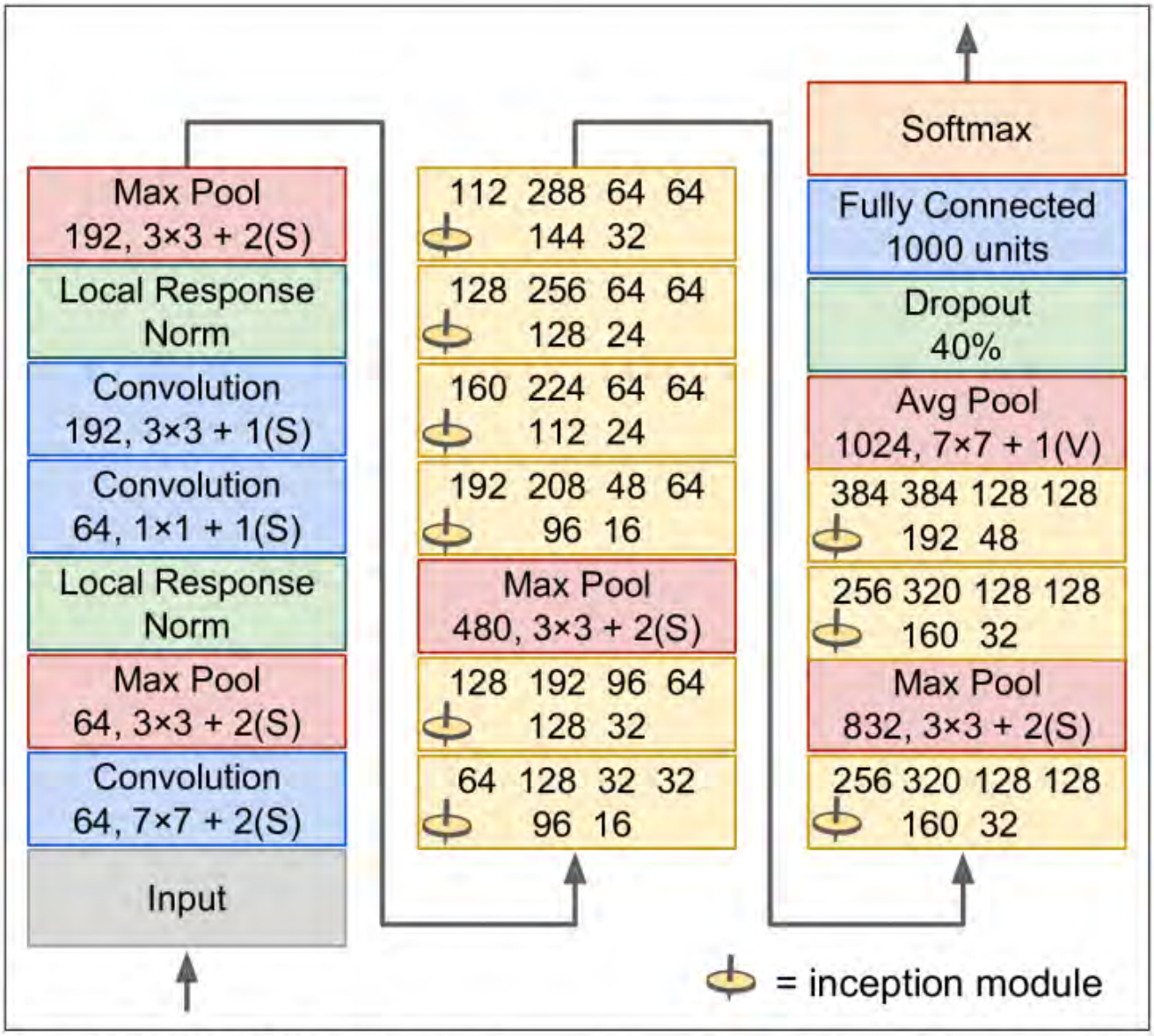


Figure 13-11. GoogLeNet architecture

现在让我们来看看GoogLeNet CNN的架构（见图13-11）。 它非常深，我们不得不将它分成三列，但是GoogLeNet实际上是一个高堆栈，包括九个“盗梦空间”模块（带有旋转顶端的框），每个模块实际上包含三层。 在内核大小之前显示每个卷积层和每个池化层输出的特征映射的数量。 “盗梦空间”模块中的六个数字表示模块中每个卷积层输出的特征映射的数量（与图13-10中的顺序相同）。 请注意，所有的卷积层都使用ReLU激活功能。

让我们来过一遍这个网络：

- 前两层将图像的高度和宽度除以4（使其面积除以16），以减少计算负担。
- 然后，局部响应规范化层确保前面的层学习各种各样的功能（如前所述）
- 接下来是两个卷积层，其中第一个像瓶颈层一样。 正如前面所解释的，你可以把这一对看作是一个单一的更智能的卷积层。
- 再次，局部响应标准化层确保了先前的层捕捉各种各样的模式。
- 接下来，最大池化层将图像高度和宽度减少2，再次加快计算速度。
- 然后是九个“盗梦空间”模块的高堆栈，与几个最大池层交织，以降低维度并加速网络。
- 接下来，平均池化层使用具有VALID填充的特征映射的大小的内核，输出1×1特征映射：这种令人惊讶的策略被称为全局平均共享。 它有效地强制以前的图层产生特征图，这些特征图实际上是每个目标类的置信图（因为其他类型的功能将被平均步骤破坏）。 这样就不必在CNN的顶部有几个完全连接的层（如AlexNet），大大减少了网络中的参数数量，并限制了过度拟合的风险。

- 最后一层是不言自明的：正则化drop out，然后是具有softmax激活函数的完全连接层来输出估计类的概率。

这个图略有简化：原来的GoogLeNet架构还包括两个插在第三和第六个启动模块之上的辅助分类器。 它们都由一个平均池层，一个卷积层，两个完全连接的层和一个softmax激活层组成。 在训练期间，他们的损失（缩小了70％）加在了整体损失上。 我们的目标是解决消失梯度问题，规范网络。 但是，结果显示其效果相对较小。
ResNet

最后但并非最不重要的是，2015年ILSVRC挑战赛的赢家Kaiming He等人开发的剩余网络(或ResNet)，该网络的打败前5错误率低到3.6%，它使用了一个非常深的CNN，由152层组成。 能够训练如此深的网络的关键是使用跳过连接（也称为快捷连接）：喂到一个层的信号也被添加到位于比堆栈高一点的层的输出。 让我们看看为什么这是有用的.

当训练一个神经网络时，目标是使其模拟一个目标函数h（x）。 如果将输入x添加到网络的输出中（即添加跳过连接），那么网络将被迫模拟f（x）= h（x） - x而不是h（x）。 这被称为残留学习（见图13-12）。

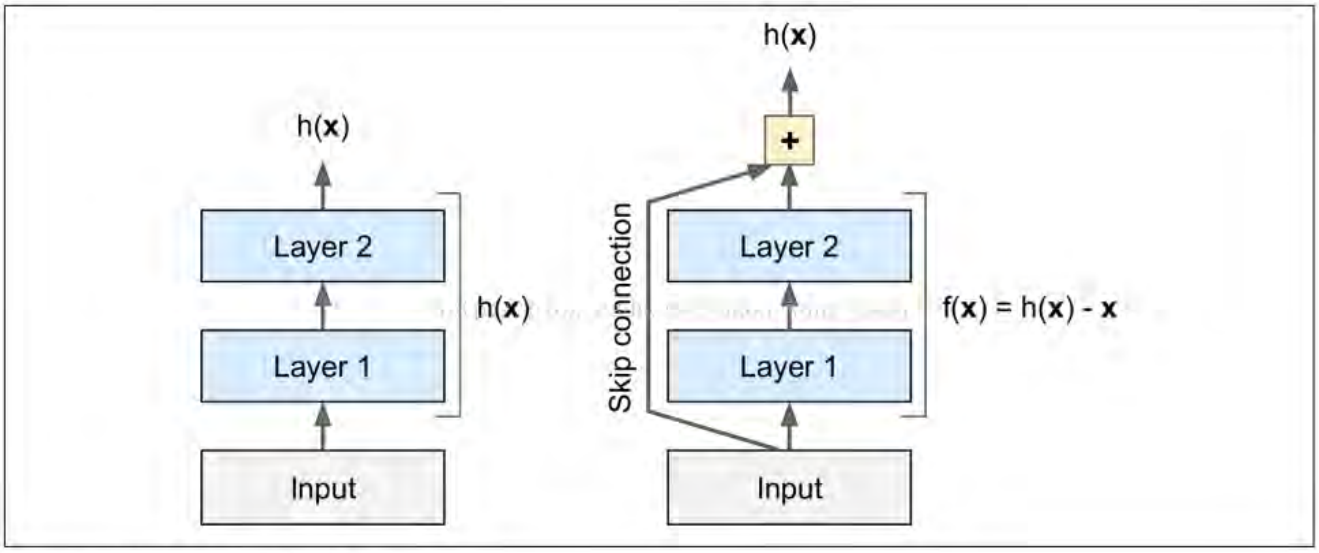


Figure 13-12. Residual learning

当你初始化一个规则的神经网络时，它的权重接近于零，所以网络只输出接近零的值。 如果添加跳过连接，则生成的网络只输出其输入的副本; 换句话说，它最初对身份函数进行建模。 如果目标函数与身份函数非常接近（常常是这种情况），这将大大加快训练速度。
而且，如果添加了许多跳转连接，即使几个层还没有开始学习，网络也可以开始进行（见图13-13）。 由于跳过连接，信号可以很容易地通过整个网络。 深度剩余网络可以看作是一堆剩余单位，其中每个剩余单位是一个有跳过连接的小型神经网络。

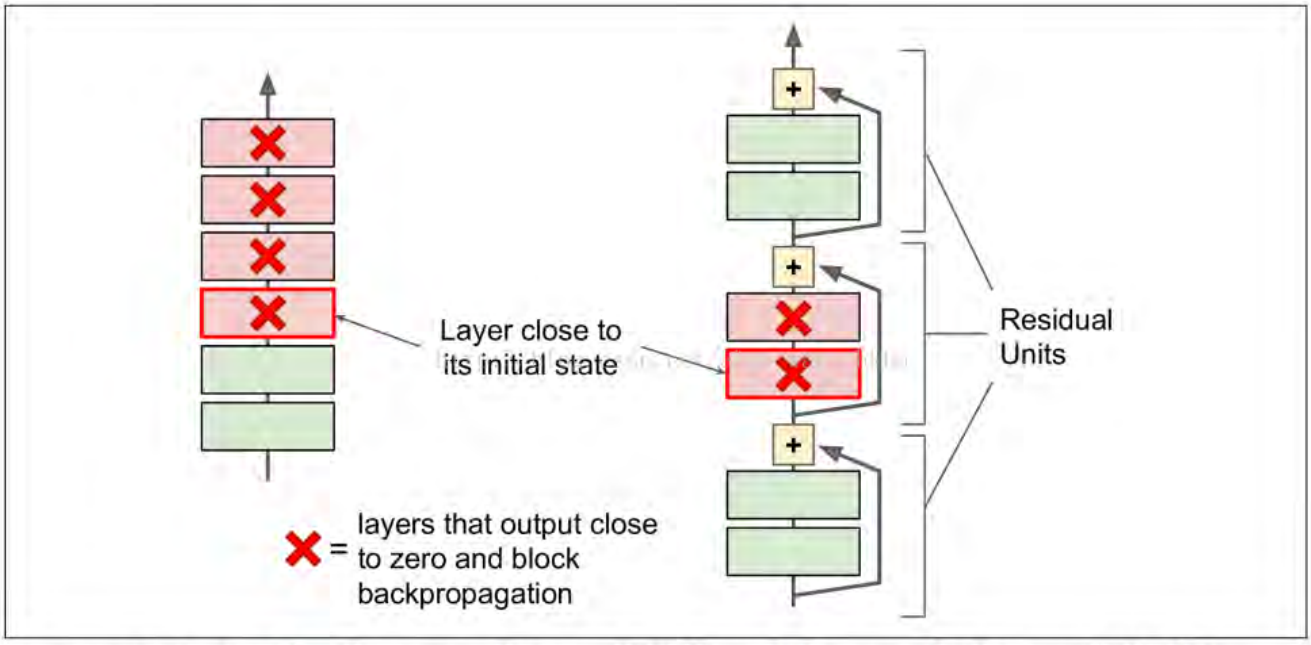


Figure 13-13. Regular deep neural network (left) and deep residual network (right)

现在让我们看看ResNet的架构（见图13-14）。 这实际上是令人惊讶的简单。 它的开始和结束与GoogLeNet完全一样（除了没有dropout层），而在两者之间只是一堆很简单的残余单位。 每个残差单元由两个卷积层组成，使用3×3的内核和保存空间维度（步幅1，SAME填充），批量归一化（BN）和ReLU激活。

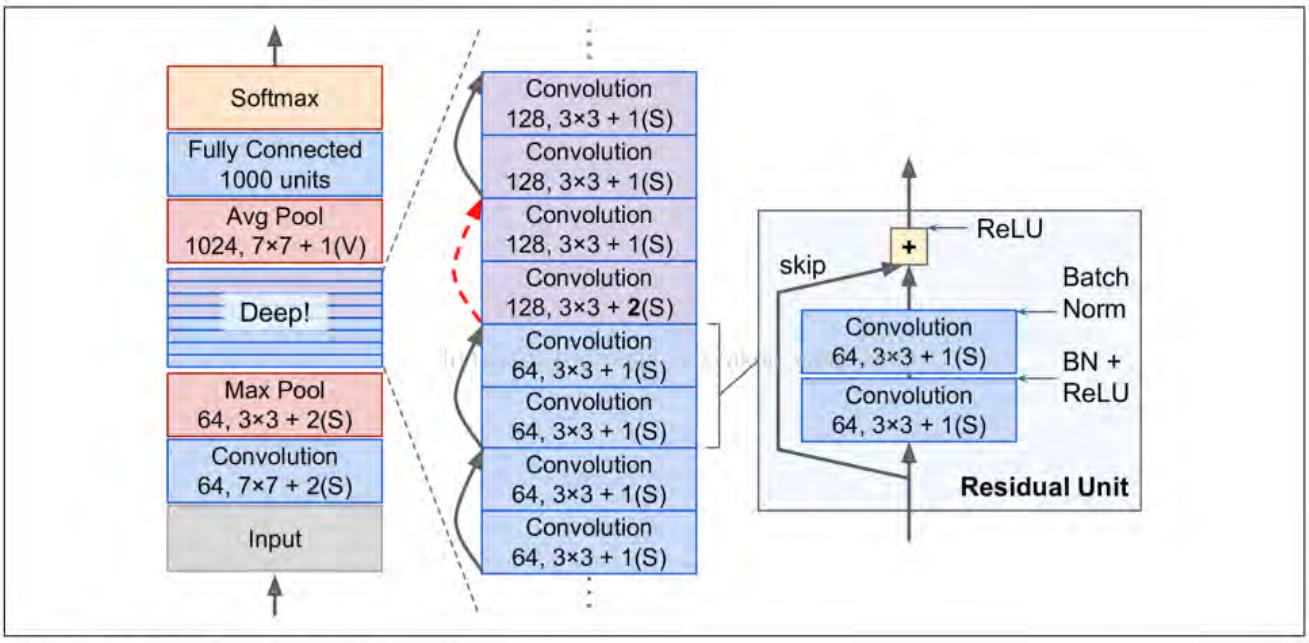


Figure 13-14. ResNet architecture

需要注意的是特征图的数量每隔几个残差单位会加倍，同时它们的高度和宽度减半（使用步幅2卷积层）。 发生这种情况时，输入不能直接添加到剩余单元的输出中，因为它们不具有相同的形状（例如，此问题影响图13-14中的虚线箭头表示的跳过连接）。 为了解决这个问题，输入通过一个1×1卷积层，步长2和右侧数量的输出特征映射（见图13-15）。

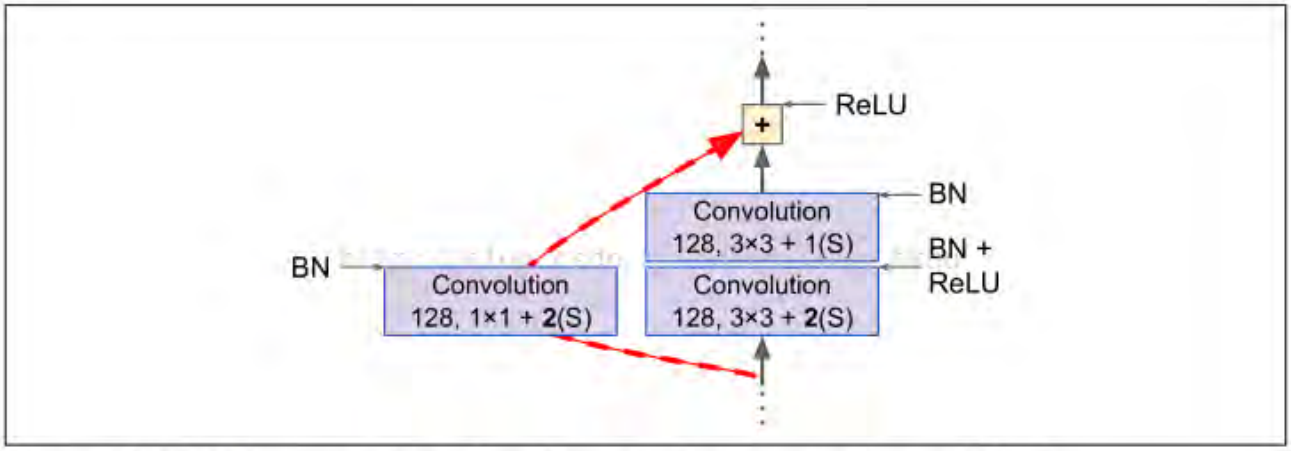


Figure 13-15. Skip connection when changing feature map size and depth

ResNet-34是具有34个层（仅计算卷积层和完全连接层）的ResNet，包含3个剩余单元输出64个特征图，4个剩余单元输出128个特征图，6个剩余单元输出256个特征图，3个剩余单元输出512个特征图
ResNet比ResNet-152更深，使用稍微不同的剩余单位。 他们使用三个卷积层，而不是两个3×3的卷积层（比如说），它们使用三个卷积层：第一个卷积层只有64个特征映射（少4倍），这是一个瓶颈层（已经讨论过），然后是具有64个特征图的3×3层，最后是具有256个特征图（4×64）的另一个1×1卷积层，以恢复原始深度。 ResNet-152包含三个这样的RU，输出256个地图，然后是8个RU，512个地图，高达36 RU，1024个地图，最后是3个RU，共2048个地图。
正如你所看到的，这个领域正在迅速发展，每年都会有各种各样的架构出现。 一个明显的趋势是CNN越来越深入。 他们也越来越轻，需要越来越少的参数。 目前，ResNet架构既是最强大的，也是最简单的，所以它现在应该是你应该使用的，但是每年都要继续关注ILSVRC的挑战。 2016年获奖者是来自中国的Trimps-Soushen团队，他们的出错率惊人的缩减到2.99%。 为了达到这个目标，他们训练了以前模型的组合，并将它们合并为一个整体。 根据任务的不同，降低的错误率可能会或可能不值得额外的复杂性。还有其他一些架构可供您参考，特别是VGGNet13（2014年ILSVRC挑战赛的亚军）和Inception-v414（将GoGoLeNet和ResNet的思想融合在一起，实现了接近3%的5大误差 ImageNet分类率）。
实施我们刚刚讨论的各种CNN架构真的没什么特别的。 我们之前看到如何构建所有的独立构建模块，所以现在您只需要组装它们来创建所需的构架。 我们将在即将开始的练习中构建ResNet-34，您将在Jupyter笔记本中找到完整的工作代码。
TensorFlow Convolution Operations

TensorFlow还提供了一些其他类型的卷积层：

- conv1d（）为1D输入创建一个卷积层。 例如，在自然语言处理中这是有用的，其中句子可以表示为一维单词阵列，并且接受场覆盖一些邻近单词。
- conv3d（）创建一个3D输入的卷积层，如3D PET扫描。
- atrous_conv2d（）创建了一个有趣的卷积层（“àtrous”是法语“with holes”）。 这相当于使用具有通过插入行和列（即，孔）而扩大的滤波器的规则卷积层。 例如，等于[[1,2,3]]的1×3滤波器可以以4的扩张率扩张，导致扩张的滤波器[[1,0,0,0,2,0,0,0,3]]。 这使得卷积层在没有计算价格的情况下具有更大的局部感受野，并且不使用额外的参数。
- conv2d_transpose（）创建了一个转置卷积层，有时称为去卷积层，它对图像进行上采样（这个名称是非常具有误导性的，因为这个层并不执行去卷积，这是一个定义良好的数学运算（卷积的逆））。这是通过在输入之间插入零来实现的，所以你可以把它看作是一个使用分数步长的规则卷积层。例如，在图像分割中，上采样是有用的：在典型的CNN中，特征映射越来越小当通过网络时，所以如果你想输出一个与输入大小相同的图像，你需要一个上采样层。
- depthwise_conv2d（）创建一个深度卷积层，将每个滤波器独立应用于每个单独的输入通道。 因此，如果有fn滤波器和fn'输入通道，那么这将输出fn×fn'特征映射。

- separable_conv2d（）创建一个可分离的卷积层，首先像深度卷积层一样工作，然后将1×1卷积层应用于结果特征映射。 这使得可以将滤波器应用于任意的输入通道组。

十四、Recurrent Neural Networks

本篇文章是个人翻译的,如有商业用途,请通知本人谢谢.

Recurrent Neurons

到目前为止，我们主要关注的是前馈神经网络，其中激活仅从输入层到输出层的一个方向流动（附录E中的几个网络除外）。 循环神经网络看起来非常像一个前馈神经网络，除了它也有连接指向后方。 让我们看一下最简单的RNN，它由一个神经元接收输入，产生一个输出，并将输出发送回自己，如图14-1（左）所示。 在每个时间步t（也称为一个帧），这个循环神经元接收输入x（t）以及它自己的前一时间步长y（t-1）的输出。 我们可以用时间轴来表示这个微小的网络，如图14-1（右）所示。 这被称为随着时间的推移展开网络。

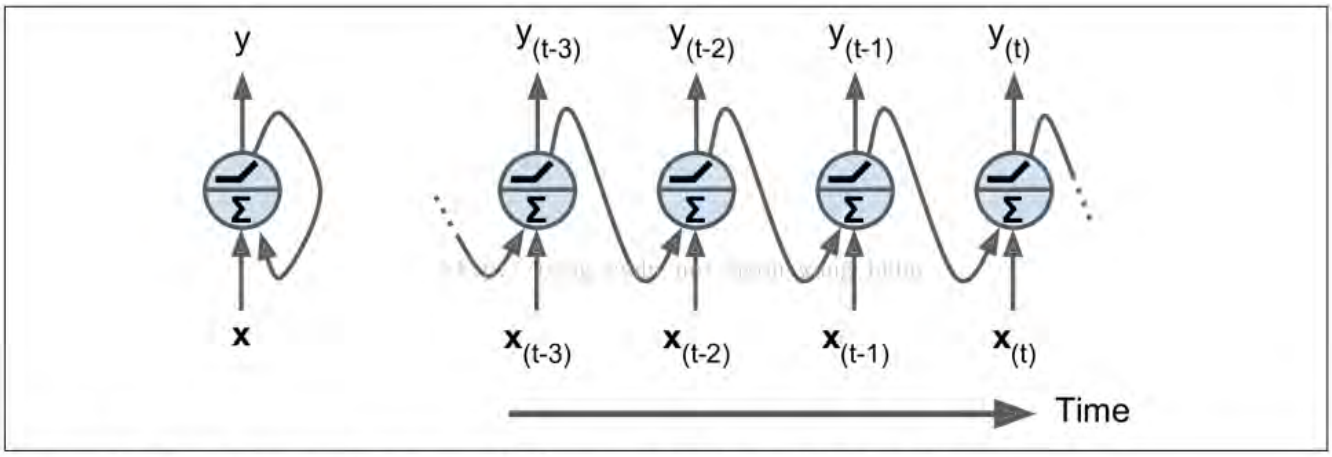


Figure 14-1. A recurrent neuron (left), unrolled through time (right)

您可以轻松创建一个复发神经元层。 在每个时间步t，每个神经元都接收输入向量x（t）和前一个时间步y（t-1）的输出向量，如图14-2所示。 请注意，输入和输出都是矢量（当只有一个神经元时，输出是一个标量）。

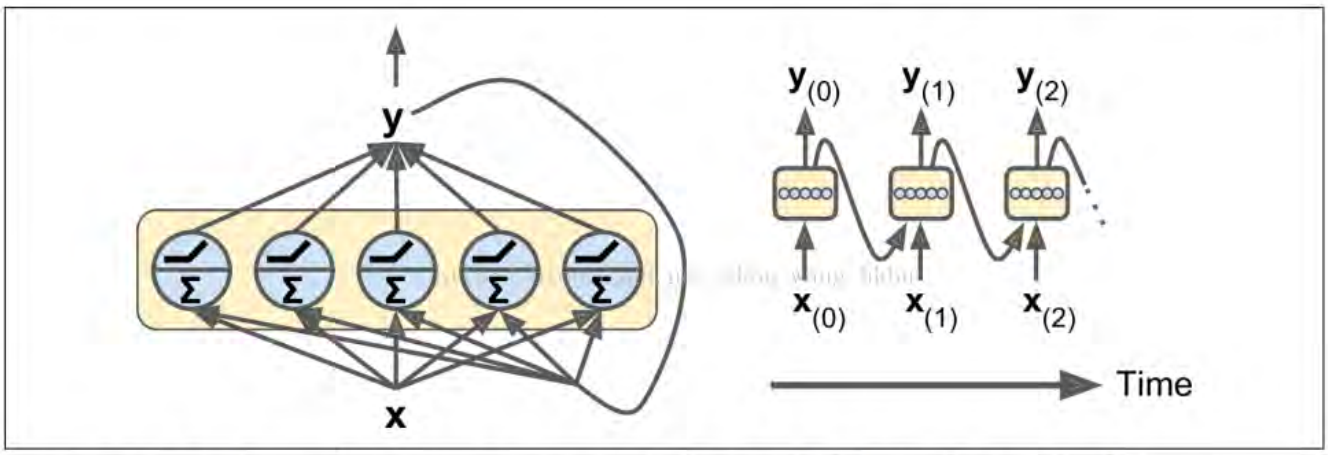


Figure 14-2. A layer of recurrent neurons (left), unrolled through time (right)

每个循环神经元有两组权重：一组用于输入x（t），另一组用于前一时间步长y（t-1）的输出。 我们称这些权重向量为wx和wy。如方程14-1所示（b是偏差项，φ（·）是激活函数，例如ReLU），可以计算单个循环神经元的输出。

Equation 14-1. Output of a single recurrent neuron for a single instance

$$\mathbf{y}_{(t)} = \phi\left(\mathbf{x}_{(t)}^{T} \cdot \mathbf{w}_x + \mathbf{y}_{(t-1)}^{T} \cdot \mathbf{w}_y + b\right)$$

就像前馈神经网络一样，我们可以使用上一个方程的向量化形式，对整个小批量计算整个层的输出（见公式14-2）。

Equation 14-2. Outputs of a layer of recurrent neurons for all instances in a mini-batch

$$\begin{aligned}\mathbf{Y}_{(t)} &= \phi\left(\mathbf{X}_{(t)} \cdot \mathbf{W}_x + \mathbf{Y}_{(t-1)} \cdot \mathbf{W}_y + \mathbf{b}\right) \\ &= \phi\left(\left[\mathbf{X}_{(t)} \quad \mathbf{Y}_{(t-1)}\right] \cdot \mathbf{W} + \mathbf{b}\right) \text{ with } \mathbf{W} = \begin{bmatrix}\mathbf{W}_x \\ \mathbf{W}_y\end{bmatrix}\end{aligned}$$

- $\mathbf{Y}_{(t)}$是一个m×n（神经元）矩阵，包含在最小批次中每个实例在时间步t处的层输出（m是最小批次中的实例数，n（神经元）是神经元）。
- $\mathbf{X}_{(t)}$是包含所有实例的输入的m×n（输入）矩阵（n（输入）是输入特征的数量）。
- $\mathbf{W}_x$是包含当前时间步的输入的连接权重的n（输入）×n（神经元）矩阵。
- $\mathbf{W}_y$是包含当前时间步的输出的连接权重的n（输入）×n（神经元）矩阵。
- 权重矩阵Wx和Wy通常连接成形状（n（输入）+ n（神经元））×n（神经元）的单个权重矩阵W（见等式14-2的第二行）
- b是包含每个神经元的偏置项的大小为n（神经元）的向量。

注意，Y(t)是X(t)和Y(t - 1)的函数，它是X(t - 1)和Y(t - 2)的函数，它是X(t - 2)和Y(t - 3)的函数，等等。 这使得Y（t）是从时间t = 0（即X（0），X（1），...，X（t））开始的所有输入的函数。 在第一个时间步，t = 0，没有以前的输出，所以它们通常被假定为全零。

Memory Cells

由于时间t的循环神经元的输出是由所有先前时间步骤计算来的的函数，你可以说它有一种记忆形式。一个神经网络的一部分，跨越时间步长保留一些状态，称为存储单元（或简称为单元）。单个循环神经元或循环神经元层是非常基本的单元，但本章后面我们将介绍一些更为复杂和强大的单元类型。

一般情况下，时间步t处的单元状态，记为h（t）（“h”代表“隐藏”），是该时间步的某些输入和前一时间步的状态的函数：h（t）= f（h（t-1），x（t））。 其在时间步t处的输出，表示为y（t），也和前一状态和当前输入的函数有关。 在我们已经讨论过的基本单元的情况下，输出等于单元状态，但是在更复杂的单元中并不总是如此，如图14-3所示。

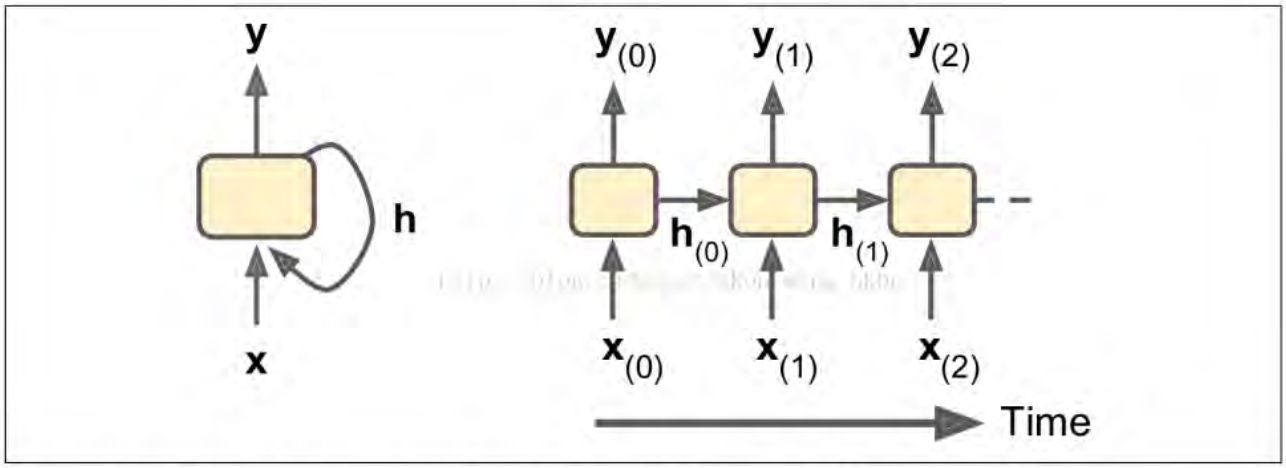


Figure 14-3. A cell's hidden state and its output may be different

Input and Output Sequences (输入和输出序列)

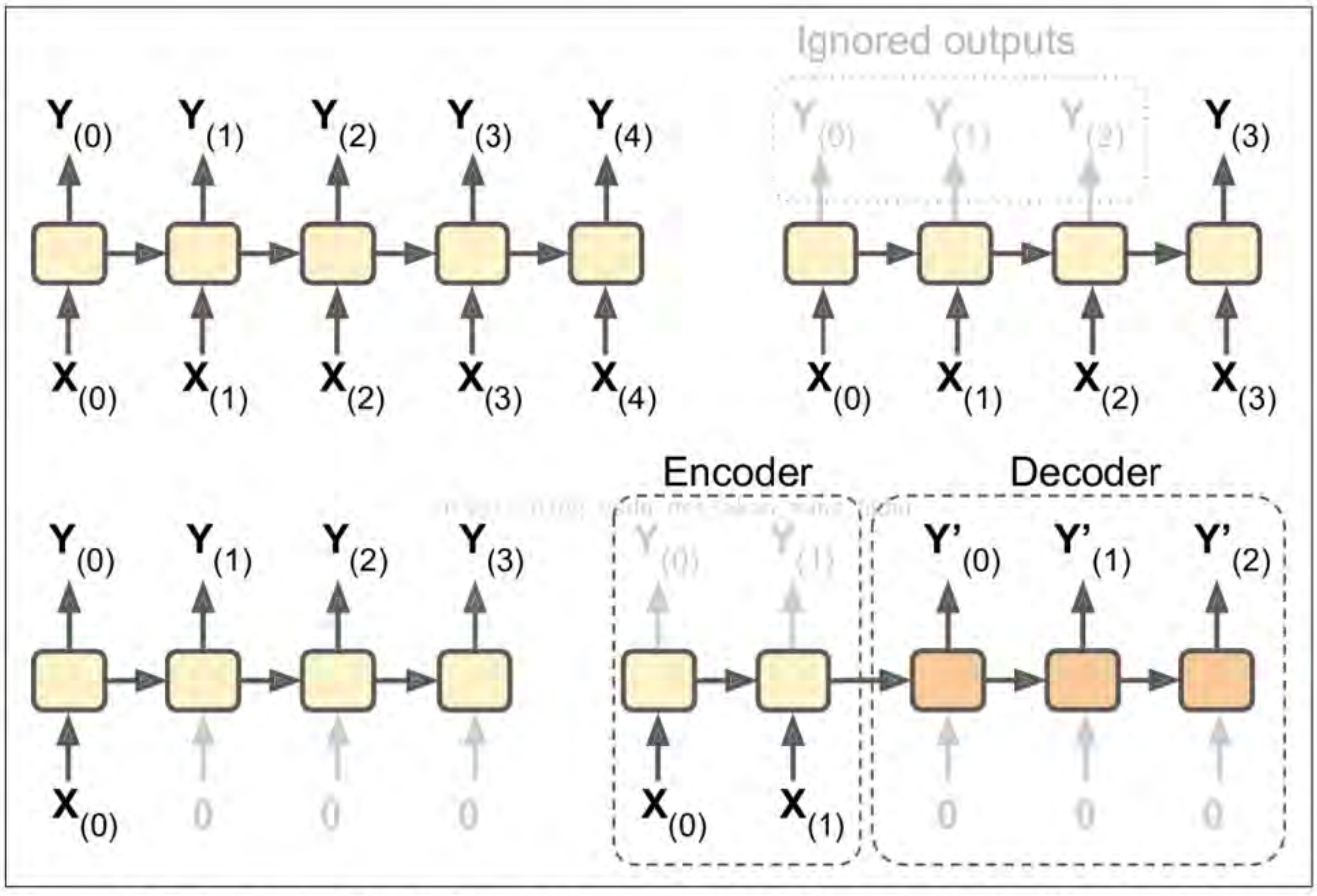


Figure 14-4. Seq to seq (top left), seq to vector (top right), vector to seq (bottom left), delayed seq to seq (bottom right)

RNN可以同时进行一系列输入并产生一系列输出（见图14-4，左上角的网络）。 例如，这种类型的网络对于预测时间序列（如股票价格）非常有用：您在过去的N天内给出价格，并且它必须输出向未来一天移动的价格（即从N - 1天 前到明天）。

或者，您可以向网络输入一系列输入，并忽略除最后一个之外的所有输出（请参阅右上角的网络）。 换句话说，这是一个向量网络的序列。 例如，您可以向网络提供与电影评论相对应的单词序列，并且网络将输出情感评分（例如，从-1 [仇恨]到+1 [爱情]）。

相反，您可以在第一个时间步中为网络提供一个输入（而在其他所有时间步中为零），然后让它输出一个序列（请参阅左下角的网络）。 这是一个矢量到序列的网络。 例如，输入可以是图像，输出可以是该图像的标题。

最后，你可以有一个序列到矢量网络，称为编码器，后面跟着一个称为解码器的矢量到序列网络（参见右下角的网络）。 例如，这可以用于将句子从一种语言翻译成另一种语言。 你会用一种语言给网络喂一个句子，编码器会把这个句子转换成单一的矢量表示，然后解码器将这个矢量解码成另一种语言的句子。 这种称为编码器 - 解码器的两步模型，比用单个序列到序列的RNN（如左上方所示的那个）快速地进行翻译要好得多，因为句子的最后一个单词可以 影响翻译的第一句话，所以你需要等到听完整个句子才能翻译。

Basic RNNs in TensorFlow

首先，我们来实现一个非常简单的RNN模型，而不使用任何TensorFlow的RNN操作，以更好地理解发生了什么。 我们将使用tanh激活函数创建一个由5个复发神经元层组成的RNN（如图14-2所示的RNN）。 我们将假设RNN只运行两个时间步，每个时间步输入大小为3的矢量。 下面的代码构建了这个RNN，展开了两个时间步骤：

```
    n_inputs = 3
    n_neurons = 5
    X0 = tf.placeholder(tf.float32, [None, n_inputs])
    X1 = tf.placeholder(tf.float32, [None, n_inputs])
    Wx = tf.Variable(tf.random_normal(shape=[n_inputs, n_neurons], dtype=tf.float32))
    Wy = tf.Variable(tf.random_normal(shape=[n_neurons, n_neurons], dtype=tf.float32))
    b = tf.Variable(tf.zeros([1, n_neurons], dtype=tf.float32))
    Y0 = tf.tanh(tf.matmul(X0, Wx) + b)
    Y1 = tf.tanh(tf.matmul(Y0, Wy) + tf.matmul(X1, Wx) + b)
    init = tf.global_variables_initializer()
```

这个网络看起来很像一个双层前馈神经网络，有一些改动：首先，两个层共享相同的权重和偏差项，其次，我们在每一层都有输入，并从每个层获得输出。 为了运行模型，我们需要在两个时间步中都有输入，如下所示：

```
    # Mini-batch: instance 0,instance 1,instance 2,instance 3
    X0_batch = np.array([[0, 1, 2], [3, 4, 5], [6, 7, 8], [9, 0, 1]])  # t = 0
    X1_batch = np.array([[9, 8, 7], [0, 0, 0], [6, 5, 4], [3, 2, 1]])  # t = 1
    with tf.Session() as sess:
        init.run()
    Y0_val, Y1_val = sess.run([Y0, Y1], feed_dict={X0: X0_batch, X1: X1_batch})
```

这个小批量包含四个实例，每个实例都有一个由两个输入组成的输入序列。 最后，Y0_val和Y1_val在所有神经元和小批量中的所有实例的两个时间步中包含网络的输出：

```
>>> print(Y0_val) # output at t = 0
[[-0.2964572 0.82874775 -0.34216955 -0.75720584 0.19011548] # instance 0
[-0.12842922 0.99981797 0.84704727 -0.99570125 0.38665548] # instance 1
[ 0.04731077 0.99999976 0.99330056 -0.999933 0.55339795] # instance 2
[ 0.70323634 0.99309105 0.99909431 -0.85363263 0.7472108 ]] # instance 3
>>> print(Y1_val) # output at t = 1
[[ 0.51955646 1. 0.99999022 -0.99984968 -0.24616946] # instance 0
[-0.70553327 -0.11918639 0.48885304 0.08917919 -0.26579669] # instance 1
[-0.32477224 0.99996376 0.99933046 -0.99711186 0.10981458] # instance 2
[-0.43738723 0.91517633 0.97817528 -0.91763324 0.11047263]] # instance 3
```

这并不难，但是当然如果你想能够运行100多个时间步骤的RNN，这个图形将会非常大。 现在让我们看看如何使用TensorFlow的RNN操作创建相同的模型。

完整代码

```
import numpy as np
import tensorflow as tf


if __name__ == '__main__':
    n_inputs = 3
    n_neurons = 5
    X0 = tf.placeholder(tf.float32, [None, n_inputs])
    X1 = tf.placeholder(tf.float32, [None, n_inputs])
    Wx = tf.Variable(tf.random_normal(shape=[n_inputs, n_neurons], dtype=tf.float32))
    Wy = tf.Variable(tf.random_normal(shape=[n_neurons, n_neurons], dtype=tf.float32))
    b = tf.Variable(tf.zeros([1, n_neurons], dtype=tf.float32))
    Y0 = tf.tanh(tf.matmul(X0, Wx) + b)
    Y1 = tf.tanh(tf.matmul(Y0, Wy) + tf.matmul(X1, Wx) + b)
    init = tf.global_variables_initializer()

    # Mini-batch: instance 0,instance 1,instance 2,instance 3
    X0_batch = np.array([[0, 1, 2], [3, 4, 5], [6, 7, 8], [9, 0, 1]])  # t = 0
    X1_batch = np.array([[9, 8, 7], [0, 0, 0], [6, 5, 4], [3, 2, 1]])  # t = 1
    with tf.Session() as sess:
        init.run()
        Y0_val, Y1_val = sess.run([Y0, Y1], feed_dict={X0: X0_batch, X1: X1_batch})

    print(Y0_val,'\n')
    print(Y1_val)
```

Static Unrolling Through Time

static_rnn（）函数通过链接单元格来创建一个展开的RNN网络。 下面的代码创建了与上一个完全相同的模型：

```
    X0 = tf.placeholder(tf.float32, [None, n_inputs])
    X1 = tf.placeholder(tf.float32, [None, n_inputs])

    basic_cell = tf.contrib.rnn.BasicRNNCell(num_units=n_neurons)
    output_seqs, states = tf.contrib.rnn.static_rnn(basic_cell, [X0, X1],
                                                    dtype=tf.float32)
    Y0, Y1 = output_seqs
```

首先，我们像以前一样创建输入占位符。 然后，我们创建一个BasicRNNCell，您可以将其视为一个工厂，创建单元的副本以构建展开的RNN（每个时间步一个）。 然后我们调用static_rnn（），给它的单元工厂和输入张量，并告诉它输入的数据类型（这是用来创建初始状态矩阵，默认情况下是满零）。 static_rnn（）函数为每个输入调用单元工厂的__call __（）函数，创建单元格的两个拷贝（每个单元包含一个5个循环神经元层），并具有共享的权重和偏置项，并且像前面一样。 static_rnn（）函数返回两个对象。 第一个是包含每个

时间步的输出张量的Python列表。 第二个是包含网络最终状态的张量。 当你使用基本的单元格时，最后的状态就等于最后的输出。

如果有50个时间步长，则不得不定义50个输入占位符和50个输出张量。而且，在执行时，您将不得不为50个占位符中的每个占位符输入数据并且还要操纵50个输出。我们来简化一下。下面的代码再次构建相同的RNN，但是这次它需要一个形状为[None，n_steps，n_inputs]的单个输入占位符，其中第一个维度是最小批量大小。然后提取每个时间步的输入序列列表。 X_seqs是形状n_steps张量的Python列表[None，n_inputs]，其中第一个维度再次是最小批量大小。为此，我们首先使用转置（）函数交换前两个维度，以便时间步骤现在是第一维度。然后，我们使 unstack（）函数沿第一维（即每个时间步的一个张量）提取张量的Python列表。接下来的两行和以前一样。最后，我们使用stack（）函数将所有输出张量合并成一个张量，然后我们交换前两个维度得到形状的最终输出张量[None, n_steps，n_neurons]（第一个维度是mini-批量大小）。

```
X = tf.placeholder(tf.float32, [None, n_steps, n_inputs])
X_seqs = tf.unstack(tf.transpose(X, perm=[1, 0, 2]))

basic_cell = tf.contrib.rnn.BasicRNNCell(num_units=n_neurons)
output_seqs, states = tf.contrib.rnn.static_rnn(basic_cell, X_seqs,
                                                dtype=tf.float32)
outputs = tf.transpose(tf.stack(output_seqs), perm=[1, 0, 2])
```

现在我们可以通过给它提供一个包含所有minibatch序列的张量来运行网络：

```
X_batch = np.array([
        # t = 0      t = 1
        [[0, 1, 2], [9, 8, 7]], # instance 1
        [[3, 4, 5], [0, 0, 0]], # instance 2
        [[6, 7, 8], [6, 5, 4]], # instance 3
        [[9, 0, 1], [3, 2, 1]], # instance 4
    ])

with tf.Session() as sess:
    init.run()
    outputs_val = outputs.eval(feed_dict={X: X_batch})
```

我们得到所有实例，所有时间步长和所有神经元的单一outputs_val张量：

```
>>> print(outputs_val)
[[[-0.2964572   0.82874775 -0.34216955 -0.75720584  0.19011548]
  [ 0.51955646  1.          0.99999022 -0.99984968 -0.24616946]]

 [[-0.12842922  0.99981797  0.84704727 -0.99570125  0.38665548]
  [-0.70553327 -0.11918639  0.48885304  0.08917919 -0.26579669]]

 [[ 0.04731077  0.99999976  0.99330056 -0.999933    0.55339795]
  [-0.32477224  0.99996376  0.99933046 -0.99711186  0.10981458]]

 [[ 0.70323634  0.99309105  0.99909431 -0.85363263  0.7472108 ]
  [-0.43738723  0.91517633  0.97817528 -0.91763324  0.11047263]]]
```

但是，这种方法仍然会建立一个每个时间步包含一个单元的图。 如果有50个时间步，这个图看起来会非常难看。 这有点像写一个程序而没有使用循环（例如，Y0 = f（0，X0）; Y1 = f（Y0，X1）; Y2 = f（Y1，X2）; ...; Y50 = f Y49，X50））。 如果使用大图，在反向传播期间（特别是在GPU卡内存有限的情况下），您甚至可能会发生内存不足（OOM）错误，因为它必须在正向传递期间存储所有张量值，因此可以使用它们来计算梯度在反向通过。

幸运的是，有一个更好的解决方案：dynamic_rnn（）函数。

Dynamic Unrolling Through T　ime

dynamic_rnn（）函数使用while_loop（）操作在单元格上运行适当的次数，如果要在反向传播期间将GPU内存交换到CPU内存，可以设置swap_memory = True，以避免内存不足错误。 方便的是，它还可以在每个时间步（形状[None，n_steps，n_inputs]）接受所有输入的单张量，并且在每个时间步（形状[None，n_steps，n_neurons]）上输出所有输出的单张量。 没有必要堆叠，拆散或转置。 以下代码使用dynamic_rnn（）函数创建与之前相同的RNN。 这太好了！

完整代码

```
import numpy as np
import tensorflow as tf
import pandas as pd

if __name__ == '__main__':
    n_steps = 2
    n_inputs = 3
    n_neurons = 5


    X = tf.placeholder(tf.float32, [None, n_steps, n_inputs])

    basic_cell = tf.contrib.rnn.BasicRNNCell(num_units=n_neurons)
    outputs, states = tf.nn.dynamic_rnn(basic_cell, X, dtype=tf.float32)

    init = tf.global_variables_initializer()

    X_batch = np.array([
        [[0, 1, 2], [9, 8, 7]],  # instance 1
        [[3, 4, 5], [0, 0, 0]],  # instance 2
        [[6, 7, 8], [6, 5, 4]],  # instance 3
        [[9, 0, 1], [3, 2, 1]],  # instance 4
    ])

    with tf.Session() as sess:
        init.run()
        outputs_val = outputs.eval(feed_dict={X: X_batch})

    print(outputs_val)
```

在反向传播期间，while_loop（）操作会执行相应的步骤：在正向传递期间存储每次迭代的张量值，以便在反向传递期间使用它们来计算梯度。

Handling Variable Length Input Sequences

到目前为止，我们只使用固定大小的输入序列（全部正好两个步长）。 如果输入序列具有可变长度（例如，像句子）呢？ 在这种情况下，您应该在调用dynamic_rnn（）（或static_rnn（））函数时设置sequence_length参数; 它必须是一维张量，表示每个实例的输入序列的长度。 例如：

```
n_steps = 2
n_inputs = 3
n_neurons = 5

reset_graph()

X = tf.placeholder(tf.float32, [None, n_steps, n_inputs])
basic_cell = tf.contrib.rnn.BasicRNNCell(num_units=n_neurons)
```

```
seq_length = tf.placeholder(tf.int32, [None])
outputs, states = tf.nn.dynamic_rnn(basic_cell, X, dtype=tf.float32,
                                    sequence_length=seq_length)
```

例如，假设第二个输入序列只包含一个输入而不是两个输入。 为了适应输入张量X，必须填充零向量（因为输入张量的第二维是最长序列的大小，即2）

```
X_batch = np.array([
        # step 0     step 1
        [[0, 1, 2], [9, 8, 7]], # instance 1
        [[3, 4, 5], [0, 0, 0]], # instance 2 (padded with zero vectors)
        [[6, 7, 8], [6, 5, 4]], # instance 3
        [[9, 0, 1], [3, 2, 1]], # instance 4
    ])
seq_length_batch = np.array([2, 1, 2, 2])
```

当然，您现在需要为两个占位符X和seq_length提供值：

```
with tf.Session() as sess:
    init.run()
    outputs_val, states_val = sess.run(
        [outputs, states], feed_dict={X: X_batch, seq_length: seq_length_batch})
```

现在，RNN输出序列长度的每个时间步都会输出零矢量（查看第二个时间步的第二个输出）：

```
>>> print(outputs_val)
[[[-0.2964572   0.82874775 -0.34216955 -0.75720584  0.19011548]
  [ 0.51955646  1.          0.99999022 -0.99984968 -0.24616946]]  # final state

 [[-0.12842922  0.99981797  0.84704727 -0.99570125  0.38665548]   # final state
  [ 0.          0.          0.          0.          0.        ]]  # zero vector

 [[ 0.04731077  0.99999976  0.99330056 -0.999933    0.55339795]
  [-0.32477224  0.99996376  0.99933046 -0.99711186  0.10981458]]  # final state

 [[ 0.70323634  0.99309105  0.99909431 -0.85363263  0.7472108 ]
  [-0.43738723  0.91517633  0.97817528 -0.91763324  0.11047263]]] # final state
```

此外，状态张量包含每个单元的最终状态（不包括零向量）：

```
>>> print(states_val)
[[ 0.51955646  1.          0.99999022 -0.99984968 -0.24616946]    # t = 1
 [-0.12842922  0.99981797  0.84704727 -0.99570125  0.38665548]    # t = 0 !!!
 [-0.32477224  0.99996376  0.99933046 -0.99711186  0.10981458]    # t = 1
 [-0.43738723  0.91517633  0.97817528 -0.91763324  0.11047263]]   # t = 1
```

Handling Variable-Length Output Sequences(处理可变长度输出序列)

如果输出序列长度不一样呢？ 如果事先知道每个序列的长度（例如，如果知道长度与输入序列的长度相同），那么可以按照上面所述设置sequence_length参数。 不幸的是，通常这是不可能的：例如，翻译后的句子的长度通常与输入句子的长度不同。 在这种情况下，最常见的解决方案是定义一个称为序列结束标记（EOS标记）的特殊输出。 任何通过EOS的输出应该被忽略（我们将在本章稍后讨论）。

好，现在你知道如何建立一个RNN网络（或者更准确地说是一个随着时间的推移而展开的RNN网络）。 但是你怎么训练呢？

Training RNNs

为了训练一个RNN，诀窍是通过时间展开（就像我们刚刚做的那样），然后简单地使用常规反向传播（见图14-5）。 这个策略被称为反向传播（BPTT）。

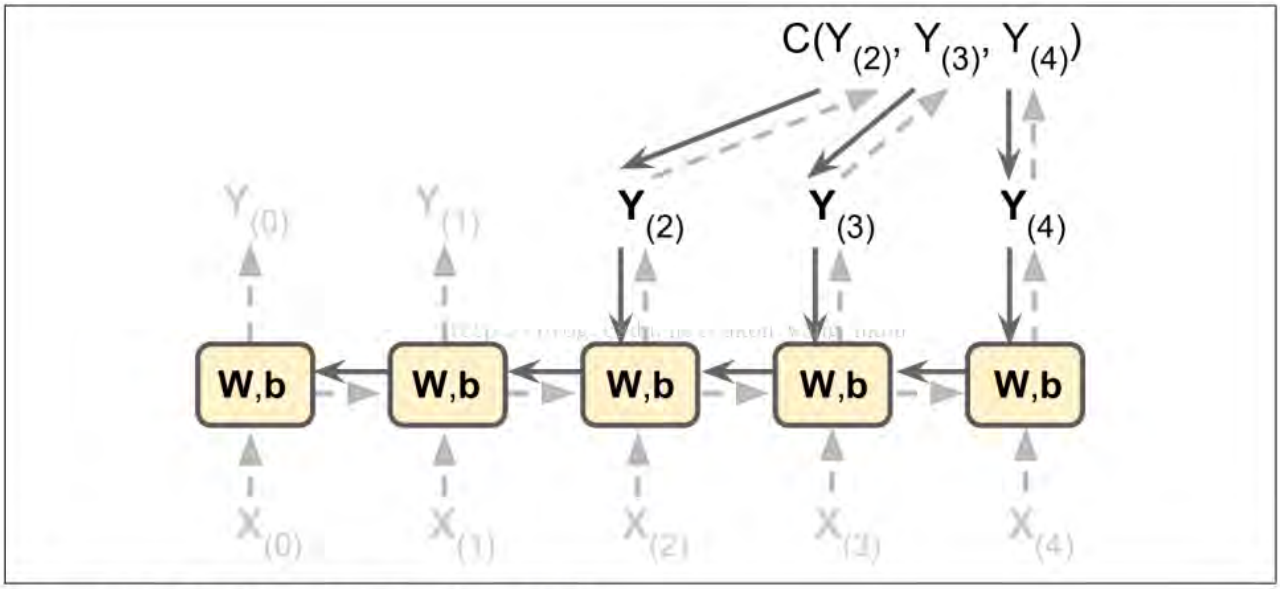


Figure 14-5. Backpropagation through time

就像在正常的反向传播中一样，展开的网络（用虚线箭头表示）有第一个正向传递。 那么使用成本函数评估输出序列 $C\left(\mathbf{Y}_{(t_{\min})}, \mathbf{Y}_{(t_{\min}+1)}, \cdots, \mathbf{Y}_{(t_{\max})}\right)$（其中tmin和tmax是第一个和最后一个输出时间步长，不计算忽略的输出），并且该损失函数的梯度通过展开的网络向后传播（由固体箭头）；最后使用在BPTT期间计算的梯度来更新模型参数。 请注意，梯度在损失函数所使用的所有输出中反向流动，而不仅仅通过最终输出（例如，在图14-5中，损失函数使用网络的最后三个输出Y（2）， Y（3）和Y（4），所以梯度流经这三个输出，但不通过Y（0）和Y（1））。 而且，由于在每个时间步骤使用相同的参数W和b，所以反向传播将做正确的事情并且总结所有时间步骤。

Training a Sequence Classifer (训练序列分类器)

我们训练一个RNN来分类MNIST图像。 卷积神经网络将更适合于图像分类（见第13章），但这是一个你已经熟悉的简单例子。 我们将把每个图像视为28行28像素的序列（因为每个MNIST图像是28×28像素）。 我们将使用150个循环神经元的细胞，再加上一个完整的连接图层，其中包含连接到上一个时间步的输出的10个神经元（每个类一个），然后是一个softmax层（见图14-6）。

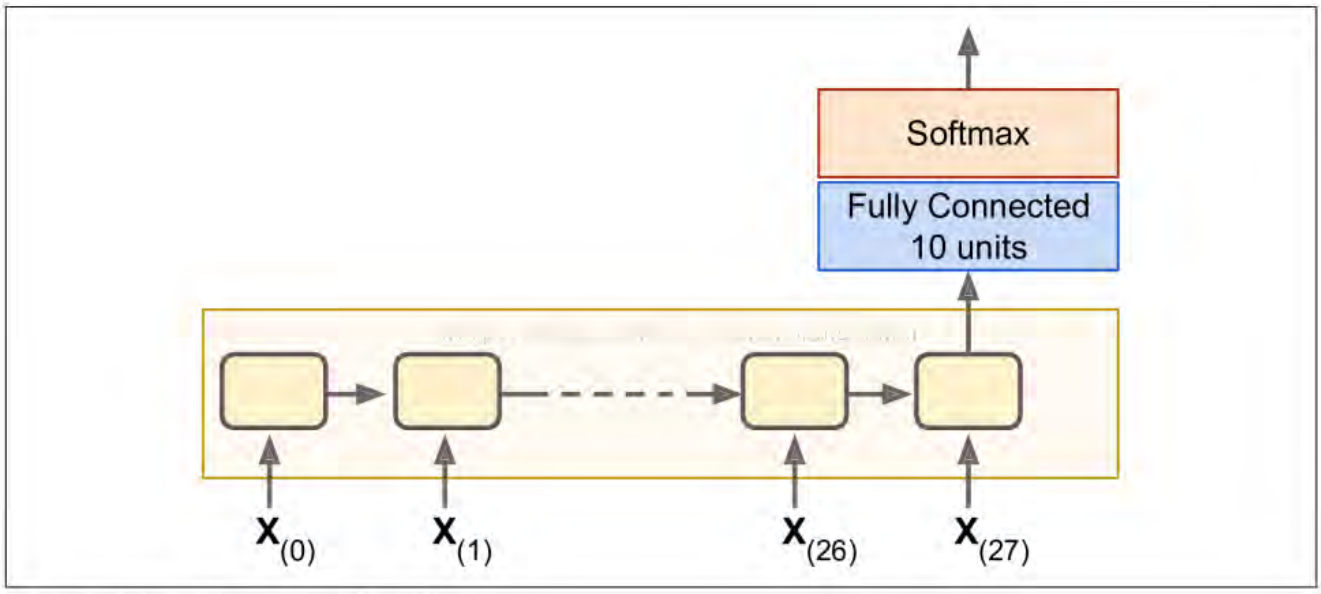


Figure 14-6. Sequence classifier

建模阶段非常简单， 它和我们在第10章中建立的MNIST分类器几乎是一样的，只是展开的RNN替换了隐藏的层。 注意，完全连接的层连接到状态张量，其仅包含RNN的最终状态（即，第28个输出）。 另请注意，y是目标类的占位符。

```
n_steps = 28
n_inputs = 28
n_neurons = 150
n_outputs = 10

learning_rate = 0.001

X = tf.placeholder(tf.float32, [None, n_steps, n_inputs])
y = tf.placeholder(tf.int32, [None])

basic_cell = tf.contrib.rnn.BasicRNNCell(num_units=n_neurons)
outputs, states = tf.nn.dynamic_rnn(basic_cell, X, dtype=tf.float32)

logits = tf.layers.dense(states, n_outputs)
xentropy = tf.nn.sparse_softmax_cross_entropy_with_logits(labels=y,
                                                          logits=logits)
loss = tf.reduce_mean(xentropy)
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
training_op = optimizer.minimize(loss)
correct = tf.nn.in_top_k(logits, y, 1)
accuracy = tf.reduce_mean(tf.cast(correct, tf.float32))

init = tf.global_variables_initializer()
```

现在让我们加载MNIST数据，并按照网络预期的方式将测试数据重塑为[batch_size，n_steps，n_inputs]。 我们将立即关注重塑训练数据。

```
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("/tmp/data/")
X_test = mnist.test.images.reshape((-1, n_steps, n_inputs))
y_test = mnist.test.labels
```

现在我们准备训练RNN了。 执行阶段与第10章中MNIST分类器的执行阶段完全相同，不同之处在于我们在将每个训练的批量喂到网络之前要重新调整。

```
batch_size = 150

with tf.Session() as sess:
    init.run()
    for epoch in range(n_epochs):
        for iteration in range(mnist.train.num_examples // batch_size):
            X_batch, y_batch = mnist.train.next_batch(batch_size)
            X_batch = X_batch.reshape((-1, n_steps, n_inputs))
            sess.run(training_op, feed_dict={X: X_batch, y: y_batch})
        acc_train = accuracy.eval(feed_dict={X: X_batch, y: y_batch})
        acc_test = accuracy.eval(feed_dict={X: X_test, y: y_test})
        print(epoch, "Train accuracy:", acc_train, "Test accuracy:", acc_test)
```

输出应该是这样的：

```
0 Train accuracy: 0.713333 Test accuracy: 0.7299
1 Train accuracy: 0.766667 Test accuracy: 0.7977
...
98 Train accuracy: 0.986667 Test accuracy: 0.9777
99 Train accuracy: 0.986667 Test accuracy: 0.9809
```

我们获得了超过98％的准确性 - 不错！ 另外，通过调整超参数，使用He初始化初始化RNN权重，更长时间训练或添加一些正则化（例如，丢失），您肯定会获得更好的结果。

您可以通过将其构造代码包装在一个变量范围内（例如，使用variable_scope（“rnn”，initializer = variance_scaling_initializer（））来使用He初始化）来为RNN指定初始化器。

Training to Predict Time Series

现在让我们来看看如何处理时间序列，如股价，气温，脑电波模式等等。 在本节中，我们将训练一个RNN来预测生成的时间序列中的下一个值。 每个训练实例是从时间序列中随机选取的20个连续值的序列，目标序列与输入序列相同，除了向后移动一个时间步（参见图14-7）。

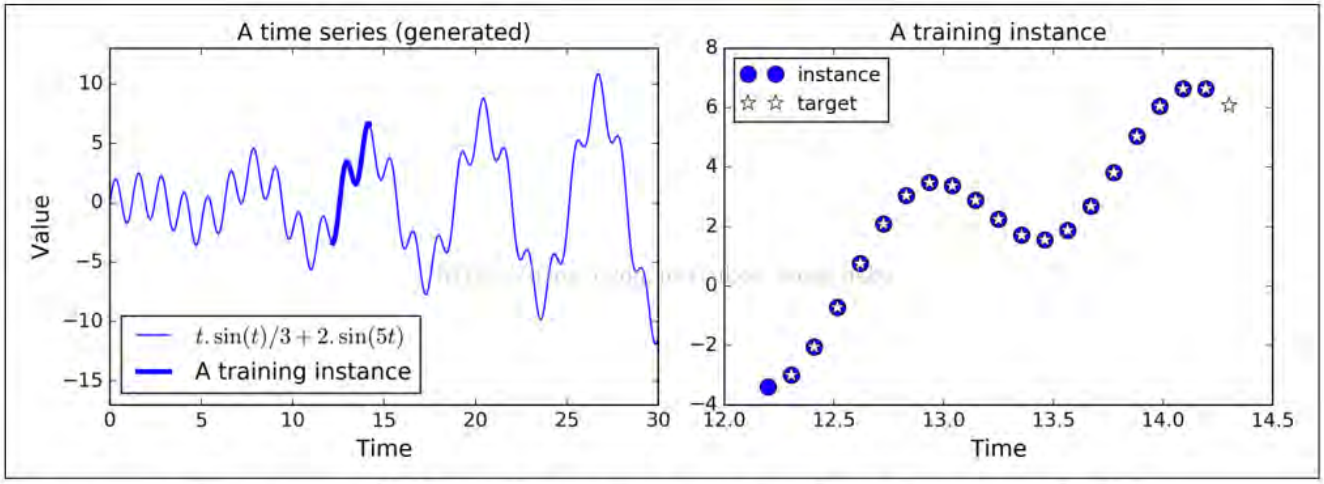


Figure 14-7. Time series (left), and a training instance from that series (right)

首先，我们来创建一个RNN。 它将包含100个经常性的神经元，并且我们将在20个时间步骤中展开它，因为每个训练实例将是20个输入长。 每个输入将仅包含一个特征（在该时间的值）。 目标也是20个输入序列，每个输入包含一个值。 代码与之前几乎相同：

```
n_steps = 20
n_inputs = 1
n_neurons = 100
n_outputs = 1

X = tf.placeholder(tf.float32, [None, n_steps, n_inputs])
y = tf.placeholder(tf.float32, [None, n_steps, n_outputs])
cell = tf.contrib.rnn.BasicRNNCell(num_units=n_neurons, activation=tf.nn.relu)
outputs, states = tf.nn.dynamic_rnn(cell, X, dtype=tf.float32)
```

一般来说，你将不只有一个输入功能。 例如，如果您试图预测股票价格，则您可能在每个时间步骤都会有许多其他输入功能，例如竞争股票的价格，分析师的评级或可能帮助系统进行预测的任何其他功能。

在每个时间步，我们现在有一个大小为100的输出向量。但是我们实际需要的是每个时间步的单个输出值。 最简单的解决方法是将单元格包装在OutputProjectionWrapper中。 单元格包装器就像一个普通的单元格，代理每个方法调用一个底层单元格，但是它也增加了一些功能。 Out putProjectionWrapper在每个输出之上添加一个完全连接的线性神经元层（即没有任何激活函数）（但不影响单元状态）。 所有这些完全连接的层共享相同（可训练）的权重和偏差项。 结果RNN如图14-8所示。

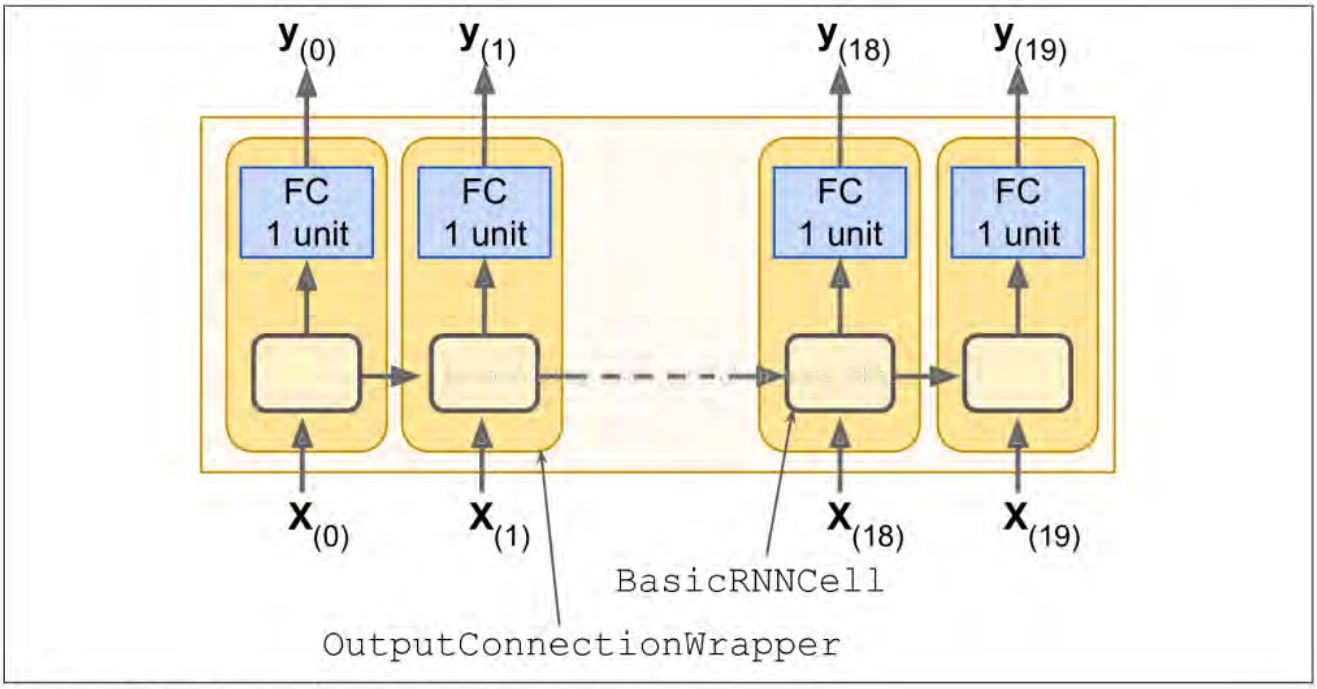


Figure 14-8. RNN cells using output projections

包装细胞是相当容易的。 让我们通过将BasicRNNCell包装到OutputProjectionWrapper中来调整前面的代码：

```
cell = tf.contrib.rnn.OutputProjectionWrapper(
    tf.contrib.rnn.BasicRNNCell(num_units=n_neurons, activation=tf.nn.relu),
    output_size=n_outputs)
```

包装细胞是相当容易的。 让我们通过将BasicRNNCell包装到OutputProjectionWrapper中来调整前面的代码：

cell =tf.contrib.rnn.OutputProjectionWrapper(

tf.contrib.rnn.BasicRNNCell(num_units=n_neurons,activation=tf.nn.relu),

output_size=n_outputs)

到现在为止还挺好。 现在我们需要定义成本函数。 我们将使用均方误差（MSE），就像我们在之前的回归任务中所做的那样。 接下来，我们将像往常一样创建一个Adam优化器，训练操作和变量初始化操作：