仅限非商业用途或学习研究使用

Broadview®
www.broadview.com.cn

深度学习框架 PyTorch入门与实践

陈云◎编著

中国工信出版集团
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

仅限非商业用途或学习研究使用

作者简介

陈 云

Python 程序员、Linux 爱好者和 PyTorch 源码贡献者。主要研究方向包括计算机视觉和机器学习。曾获得“2017 知乎 · 看山杯机器学习挑战赛”一等奖，“2017 天池医疗 AI 大赛”第八名。热衷于推广 PyTorch，并有丰富的使用经验，活跃于 PyTorch 论坛和知乎相关板块。

轻松注册成为博文视点社区用户（www.broadview.com.cn），扫码直达本书页面。

下载资源： 本书所提供的示例代码及资源文件均可在【下载资源】处下载。

提交勘误： 您对书中内容的修改意见可在【提交勘误】处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。

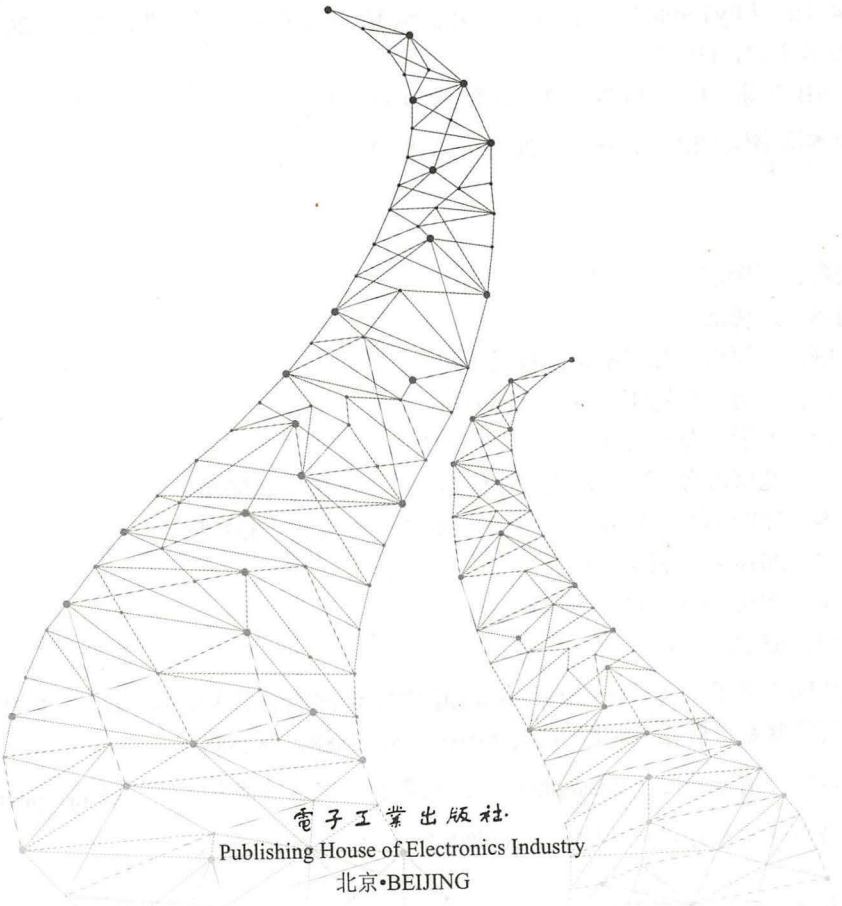
与读者交流： 在页面下方【读者评论】处留下您的疑问或观点，与其他读者一同学习交流。

页面入口：
http://www.broadview.com.cn/33077

欢迎加入本书读者QQ群：397326578

深度学习框架
PyTorch入门与实践

陈云◎编著



電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内容简介

本书从多维数组 Tensor 开始，循序渐进地带领读者了解 PyTorch 各方面的基础知识，并结合基础知识和前沿研究，带领读者从零开始完成几个经典有趣的深度学习小项目，包括 GAN 生成动漫头像、AI 滤镜、AI 写诗等。本书没有简单机械地介绍各个函数接口的使用，而是尝试分门别类、循序渐进地向读者介绍 PyTorch 的知识，希望读者对 PyTorch 有一个完整的认识。

本书内容由浅入深，无论是深度学习的初学者，还是第一次接触 PyTorch 的研究人员，都能在学习本书的过程中快速掌握 PyTorch。即使是有一定 PyTorch 使用经验的用户，也能够从本书中获得对 PyTorch 不一样的理解。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。

版权所有，侵权必究。

图书在版编目（CIP）数据

深度学习框架 PyTorch：入门与实践 / 陈云编著. —北京：电子工业出版社，2018.1

ISBN 978-7-121-33077-3

I. ①深…II. ①陈…III. ①机器学习 – 研究 IV. ①TP181

中国版本图书馆 CIP 数据核字（2017）第 286730 号

策划编辑：郑柳洁

责任编辑：郑柳洁

印　　刷：三河市双峰印刷装订有限公司

装　　订：三河市双峰印刷装订有限公司

出版发行：电子工业出版社

　　北京市海淀区万寿路 173 信箱　　邮编：100036

开　　本：787×980　1/16　　印张：18.75　　字数：353 千字

版　　次：2018 年 1 月第 1 版

印　　次：2018 年 4 月第 3 次印刷

定　　价：65.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：（010）51260888-819　faq@phei.com.cn。

前言

为什么写这本书

2016 年是属于 TensorFlow 的一年，凭借谷歌的大力推广，TensorFlow 占据了各大媒体的头条。2017 年年初，PyTorch 的横空出世吸引了研究人员极大的关注，PyTorch 简洁优雅的设计、统一易用的接口、追风逐电的速度和变化无方的灵活性给人留下深刻的印象。

作为一门 2017 年刚刚发布的深度学习框架，研究人员所能获取的学习资料有限，中文资料更是比较少。笔者长期关注 PyTorch 发展，经常在论坛上帮助 PyTorch 新手解决问题，在平时的科研中利用 PyTorch 进行各个方面的研究，有着丰富的使用经验。看到国内的用户对 PyTorch 十分感兴趣，迫切需要一本能够全面讲解 PyTorch 的书籍，于是本书就这么诞生了。

本书的结构

本书分为两部分：第 2~5 章主要介绍 PyTorch 的基础知识。

- 第 2 章介绍 PyTorch 的安装和配置学习环境。同时以最概要的方式介绍 PyTorch 的主要内容，让读者对 PyTorch 有一个大概的整体印象。
- 第 3 章介绍 PyTorch 中多维数组 Tensor 和动态图 autograd/Variable 的使用，并配以例子，让读者分别使用 Tensor 和 autograd 实现线性回归，比较二者的不同点。本章还对 Tensor 的底层设计，以及 autograd 的原理进行了分析，给读者以更全面具体的讲解。
- 第 4 章介绍 PyTorch 中神经网络模块 nn 的基础用法，同时讲解了神经网络中的“层”、“损失函数”、“优化器”等，最后带领读者用不到 50 行的代码搭建出曾夺得 ImageNet 冠军的 ResNet。
- 第 5 章介绍 PyTorch 中数据加载、GPU 加速和可视化等相关工具。

第 6~10 章主要介绍实战案例。

- 第 6 章是承上启下的一章，目标不是教会读者新函数、新知识，而是结合 Kaggle 中一个经典的比赛，实现一个深度学习中比较简单的图像二分类问题。在实现的过程中，带领读者复习前 5 章的知识，并提出代码规范以合理地组织程序和代码，使程序更可读、可维护。第 6 章还介绍在 PyTorch 中如何进行 debug。
- 第 7 章为读者讲解当前最火爆的生成对抗网络（GAN），带领读者从零开始实现一个动漫头像生成器，能够利用 GAN 生成风格多变的动漫头像。
- 第 8 章为读者讲解风格迁移的相关知识，并带领读者实现风格迁移网络，将自己的照片变成“高大上”的名画。
- 第 9 章为读者讲解一些自然语言处理的基础知识，并讲解 CharRNN 的原理。然后利用其收集几万首唐诗，训练出一个可以自动写诗歌的小程序。这个小程序可以控制生成诗歌的格式和意境，还能生成藏头诗。
- 第 10 章为读者介绍图像描述任务，并以最新的 AI Challenger 比赛的数据为例，带领读者实现一个可以进行简单图像描述的小程序。

第 1 章和第 11 章是本书的首章和末章，第 1 章介绍 PyTorch 的优势，以及和市面上其他几款框架的对比。第 11 章是对本书的总结，以及对 PyTorch 不足之处的思考，同时对读者未来的学习提出建议。

关于代码

本书的所有代码都开源在 GitHub[①]上，其中：

- 第 2~5 章的代码以 Jupyter Notebook 形式提供，读者可以在自己的计算机上交互式地修改运行它。
- 第 6~10 章的代码以单独的程序给出，每个函数的作用与细节在代码中有大量的注释。

本书的代码，在最新版的 PyTorch 0.2 上运行，同时支持 Python 2 和 Python 3，其中：

- 前 5 章的代码同时在 Python 2.7 和 Python 3.5 上验证，并得到最终结果。
- 第 6~10 章的代码，主要在 Python 2.7 上运行并得到最终结果，同时在 Python 3.5 上测试未报错。

[①] https://github.com/chenyuntc/pytorch-book

适读人群

学习本书需要读者具备以下基础知识：

- 了解 Python 的基础语法，掌握基础的 Python 使用方法。
- 有一定深度学习基础，了解反向传播、卷积神经网络等基础知识，但并不要求深入了解。
- 具备梯度、导数等高中数学基础知识。

以下知识不是必需的，但最好了解：

- numpy 的使用。
- 深度学习的基本流程或者其他深度学习框架的使用。

本书不适合哪些读者：

- 没有任何深度学习基础的用户。
- 没有 Python 基础的用户。
- 只能使用 Windows 的用户。

本书约定

在本书中，笔者是本书编著者的自称，作者指的是软件、论文等的作者，读者指阅读本书的你。

本书前 5 章的代码由 Jupyter Notebook 转换而来，其中：

- In 后面跟着的是输入的代码。
- Out 是指程序的运行结果，运行结果取决于 In 的最后一行。
- Print 后面跟着程序的打印输出内容，只有在 In 程序中调用了 `print` 函数/语句才会有 Print 输出。
- Jupyter 会自动输出 Image 对象和 matplotlib 可视化结果，所以书中以“程序输出”命名的图片都来自 Jupyter 的程序输出。这些图片的说明在代码注释中。

如何使用本书

本书第 2 章是 PyTorch 快速入门，第 3~5 章是对这些内容的详细深入介绍。第 6 章是一个简单而完整的深度学习案例。

如果你是经验丰富的研究人员，之前对 PyTorch 十分熟悉，对本书的某些例子比较感兴趣，那么你可以跳过前 5 章，直接阅读第 6 章，了解这些例子的程序设计与文件组织安排，然后阅读相应的例子。

如果你是初学者，想以最快的速度掌握 PyTorch 并将 PyTorch 应用到实际项目中，那么你可以花费 2~3 小时阅读 2.2 节的相关内容。如果你需要深入了解某部分的内容，那么可以阅读相应章节。

如果你是初学者，想完整全面地掌握 PyTorch，那么建议你：

- 先阅读第 1~5 章，了解 PyTorch 的各个基础知识。
- 再阅读第 6 章，了解 PyTorch 实践中的技巧。
- 最后从第 7~10 章挑选出感兴趣的例子，动手实践。

最后，希望读者在阅读本书的时候，尽量结合本书的配套代码阅读、修改、运行之。

致谢

杜玉姣同学在我编写本书的时候，给了我许多建议，并协助审阅了部分章节，在此特向她表示谢意。在编写本书时，本书编辑郑柳洁女士给予了很大的帮助，在此特向她致谢。感谢我的家人一直以来对我的支持，感谢我的导师肖波副教授对我的指导。感谢我的同学、师弟师妹们，他们在使用 PyTorch 中遇到了很多问题，给了我许多反馈意见。

由于笔者水平所限，书中难免有错误和不当之处，欢迎读者批评指正。具体意见可以发表在 GitHub 上的 issue（`https://github.com/chenyuntc/pytorch-book/issues`）中，或者通过邮箱（i@knew.be）联系笔者。

目录

1 PyTorch 简介 **1**

1.1 PyTorch 的诞生 1

1.2 常见的深度学习框架简介 2

1.2.1 Theano 3

1.2.2 TensorFlow 3

1.2.3 Keras 5

1.2.4 Caffe/Caffe2 5

1.2.5 MXNet 6

1.2.6 CNTK 7

1.2.7 其他框架 8

1.3 属于动态图的未来 8

1.4 为什么选择 PyTorch 10

1.5 星火燎原 12

1.6 fast.ai 放弃 Keras+TensorFlow 选择 PyTorch 13

2 快速入门 **16**

2.1 安装与配置 16

2.1.1 安装 PyTorch 16

2.1.2 学习环境配置 20

2.2 PyTorch 入门第一步 30

2.2.1 Tensor 30

2.2.2 Autograd：自动微分 35

2.2.3 神经网络 38

2.2.4 小试牛刀：CIFAR-10 分类 43

3 Tensor 和 autograd **51**

3.1 Tensor 51

3.1.1 基础操作 52

3.1.2 Tensor 和 Numpy 70

3.1.3 内部结构 73

3.1.4 其他有关 Tensor 的话题 76

3.1.5 小试牛刀：线性回归 78

3.2 autograd 81

3.2.1 Variable 82

3.2.2 计算图 86

3.2.3 扩展 autograd 94

3.2.4 小试牛刀：用 Variable 实现线性回归 99

4 神经网络工具箱 nn **103**

4.1 nn.Module 103

4.2 常用的神经网络层 107

4.2.1 图像相关层 107

4.2.2 激活函数 110

4.2.3 循环神经网络层 114

4.2.4 损失函数 116

4.3 优化器 116

4.4 nn.functional 118

4.5 初始化策略 120

4.6 nn.Module 深入分析 122

4.7 nn 和 autograd 的关系 129

4.8 小试牛刀：用 50 行代码搭建 ResNet 130

5 PyTorch 中常用的工具 **135**

5.1 数据处理 135

5.2 计算机视觉工具包：torchvision 147

5.3 可视化工具 149

5.3.1 Tensorboard 150

5.3.2 visdom 152

5.4 使用 GPU 加速：cuda 158

5.5 持久化 161

6 PyTorch 实战指南 **164**

6.1 编程实战：猫和狗二分类 164

6.1.1 比赛介绍 165

6.1.2 文件组织架构 165

6.1.3 关于`__init__.py` 167

6.1.4 数据加载 167

6.1.5 模型定义 170

6.1.6 工具函数 171

6.1.7 配置文件 174

6.1.8 main.py 176

6.1.9 使用 184

6.1.10 争议 185

6.2 PyTorch Debug 指南 187

6.2.1 ipdb 介绍 187

6.2.2 在 PyTorch 中 Debug 191

7 AI 插画师：生成对抗网络 **197**

7.1 GAN 的原理简介 198

7.2 用 GAN 生成动漫头像 202

7.3 实验结果分析 211

8 AI 艺术家：神经网络风格迁移 215

8.1 风格迁移原理介绍 216

8.2 用 PyTorch 实现风格迁移 222

8.3 实验结果分析 233

9 AI 诗人：用 RNN 写诗 237

9.1 自然语言处理的基础知识 237

9.1.1 词向量 238

9.1.2 RNN 240

9.2 CharRNN 243

9.3 用 PyTorch 实现 CharRNN 246

9.4 实验结果分析 257

10 Image Caption：让神经网络看图讲故事 260

10.1 图像描述介绍 261

10.2 数据 262

10.2.1 数据介绍 262

10.2.2 图像数据处理 270

10.2.3 数据加载 272

10.3 模型与训练 275

10.4 实验结果分析 280

11 展望与未来 282

11.1 PyTorch 的局限与发展 282

11.2 使用建议 286

1 PyTorch 简介

1.1 PyTorch 的诞生

2017 年 1 月，Facebook 人工智能研究院（FAIR）团队在 GitHub 上开源了 PyTorch（PyTorch 的 Logo 如图 1-1 所示），并迅速占领 GitHub 热度榜榜首。

作为一个 2017 年才发布，具有先进设计理念的框架，PyTorch 的历史可追溯到 2002 年就诞生于纽约大学的 Torch。Torch 使用了一种不是很大众的语言 Lua 作为接口。Lua 简洁高效，但由于其过于小众，用的人不是很多，以至于很多人听说要掌握 Torch 必须新学一门语言就望而却步（其实 Lua 是一门比 Python 还简单的语言）。

图 1-1　PyTorch 的 Logo，英文 Torch 是火炬的意思，所以 Logo 中有火焰

考虑到 Python 在计算科学领域的领先地位，以及其生态完整性和接口易用性，几乎任何框架都不可避免地要提供 Python 接口。终于，在 2017 年，Torch 的幕后团队推出了 PyTorch。PyTorch 不是简单地封装 Lua Torch 提供 Python 接口，而是对 Tensor 之上的所有模块进行了重构，并新增了最先进的自动求导系统，成为当下最流行的动态图框架。

PyTorch 一经推出就立刻引起了广泛关注，并迅速在研究领域流行起来。图 1-2 所示为 Google 指数，PyTorch 自发布起关注度就在不断上升，截至 2017 年 10 月 18 日，

PyTorch 的热度已然超越了其他三个框架（Caffe、MXNet 和 Theano），并且其热度还在持续上升中。

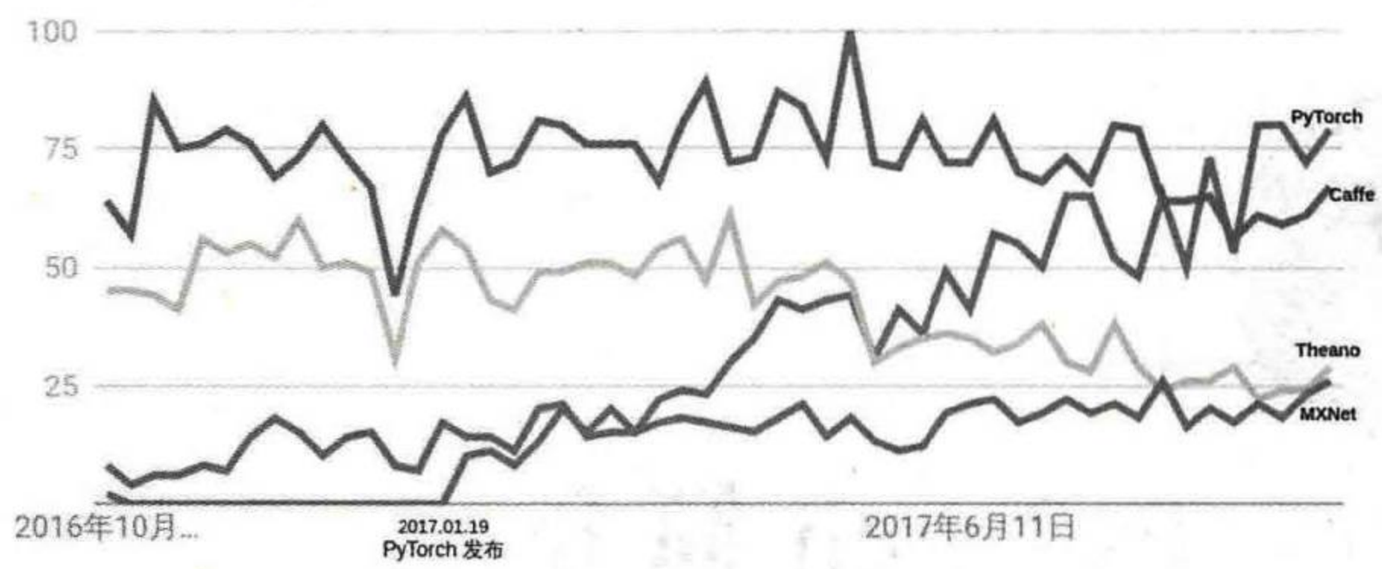


图 1-2　PyTorch 和 Caffe、Theano、MXNet 的 Google 指数对比（类别为科学）

1.2　常见的深度学习框架简介

随着深度学习的发展，深度学习框架如雨后春笋般诞生于高校和公司中。尤其是近两年，Google、Facebook、Microsoft 等巨头都围绕深度学习重点投资了一系列新兴项目，他们也一直在支持一些开源的深度学习框架。

目前研究人员正在使用的深度学习框架不尽相同，有 TensorFlow 、Caffe、Theano、Keras 等，常见的深度学习框架如图 1-3 所示。这些深度学习框架被应用于计算机视觉、语音识别、自然语言处理与生物信息学等领域，并获取了极好的效果。本节主要介绍当前深度学习领域影响力比较大的几个框架，限于笔者个人使用经验和了解程度，对各个框架的评价可能有不准确的地方。

图 1-3　常见的深度学习框架

1.2.1 Theano

Theano 最初诞生于蒙特利尔大学 LISA 实验室，于 2008 年开始开发，是第一个有较大影响力的 Python 深度学习框架。

Theano 是一个 Python 库，可用于定义、优化和计算数学表达式，特别是多维数组（numpy.ndarray）。在解决包含大量数据的问题时，使用 Theano 编程可实现比手写 C 语言更快的速度，而通过 GPU 加速，Theano 甚至可以比基于 CPU 计算的 C 语言快上好几个数量级。Theano 结合了计算机代数系统（Computer Algebra System，CAS）和优化编译器，还可以为多种数学运算生成定制的 C 语言代码。对于包含重复计算的复杂数学表达式的任务而言，计算速度很重要，因此这种 CAS 和优化编译器的组合是很有用的。对需要将每一种不同的数学表达式都计算一遍的情况，Theano 可以最小化编译/解析的计算量，但仍然会给出如自动微分那样的符号特征。

Theano 诞生于研究机构，服务于研究人员，其设计具有较浓厚的学术气息，但在工程设计上有较大的缺陷。一直以来，Theano 因难调试、构建图慢等缺点为人所诟病。为了加速深度学习研究，人们在它的基础之上，开发了 Lasagne、Blocks、PyLearn2 和 Keras 等第三方框架，这些框架以 Theano 为基础，提供了更好的封装接口以方便用户使用。

2017 年 9 月 28 日，在 Theano 1.0 正式版即将发布前夕，LISA 实验室负责人，深度学习三巨头之一的 Yoshua Bengio 宣布 Theano 即将停止开发：“Theano is Dead”。尽管 Theano 即将退出历史舞台，但作为第一个 Python 深度学习框架，它很好地完成了自己的使命，为深度学习研究人员的早期拓荒提供了极大的帮助，同时也为之后深度学习框架的开发奠定了基本设计方向：以计算图为框架的核心，采用 GPU 加速计算。

2017 年 11 月，LISA 实验室在 GitHub 上开启了一个初学者入门项目，旨在帮助实验室新生快速掌握机器学习相关的实践基础，而该项目正是使用 PyTorch 作为教学框架。

点评：由于 Theano 已经停止开发，不建议作为研究工具继续学习。

1.2.2 TensorFlow

2015 年 11 月 10 日，Google 宣布推出全新的机器学习开源工具 TensorFlow。TensorFlow 最初是由 Google 机器智能研究部门的 Google Brain 团队开发，基于 Google 2011 年开发的深度学习基础架构 DistBelief 构建起来的。TensorFlow 主要用于进行机器学习和深度神经网络研究，但它是一个非常基础的系统，因此也可以应用于众多领域。由于

Google 在深度学习领域的巨大影响力和强大的推广能力，TensorFlow 一经推出就获得了极大的关注，并迅速成为如今用户最多的深度学习框架。

TensorFlow 在很大程度上可以看作 Theano 的后继者，不仅因为它们有很大一批共同的开发者，而且它们还拥有相近的设计理念，都是基于计算图实现自动微分系统。TensorFlow 使用数据流图进行数值计算，图中的节点代表数学运算，而图中的边则代表在这些节点之间传递的多维数组（张量）。

TensorFlow 编程接口支持 Python 和 C++。随着 1.0 版本的公布，Java、Go、R 和 Haskell API 的 alpha 版本也被支持。此外，TensorFlow 还可在 Google Cloud 和 AWS 中运行。TensorFlow 还支持 Windows 7、Windows 10 和 Windows Server 2016。由于 TensorFlow 使用 C++ Eigen 库，所以库可在 ARM 架构上编译和优化。这也就意味着用户可以在各种服务器和移动设备上部署自己的训练模型，无须执行单独的模型解码器或者加载 Python 解释器。

作为当前最流行的深度学习框架，TensorFlow 获得了极大的成功，对它的批评也不绝于耳，总结起来主要有以下四点。

- 过于复杂的系统设计，TensorFlow 在 GitHub 代码仓库的总代码量超过 100 万行。这么大的代码仓库，对于项目维护者来说维护成为了一个难以完成的任务，而对读者来说，学习 TensorFlow 底层运行机制更是一个极其痛苦的过程，并且大多数时候这种尝试以放弃告终。
- 频繁变动的接口。TensorFlow 的接口一直处于快速迭代之中，并且没有很好地考虑向后兼容性，这导致现在许多开源代码已经无法在新版的 TensorFlow 上运行，同时也间接导致了许多基于 TensorFlow 的第三方框架出现 BUG。
- 接口设计过于晦涩难懂。在设计 TensorFlow 时，创造了图、会话、命名空间、PlaceHolder 等诸多抽象概念，对普通用户来说难以理解。同一个功能，TensorFlow 提供了多种实现，这些实现良莠不齐，使用中还有细微的区别，很容易将用户带入坑中。
- 文档混乱脱节。TensorFlow 作为一个复杂的系统，文档和教程众多，但缺乏明显的条理和层次，虽然查找很方便，但用户却很难找到一个真正循序渐进的入门教程。

由于直接使用 TensorFlow 的生产力过于低下，包括 Google 官方等众多开发者尝试基于 TensorFlow 构建一个更易用的接口，包括 Keras、Sonnet、TFLearn、TensorLayer、Slim、Fold、PrettyLayer 等数不胜数的第三方框架每隔几个月就会在新闻中出现一次，但是又大多归于沉寂，至今 TensorFlow 仍没有一个统一易用的接口。

凭借 Google 着强大的推广能力，TensorFlow 已经成为当今最炙手可热的深度学习框架，但是由于自身的缺陷，TensorFlow 离最初的设计目标还很遥远。另外，由于 Google

对 TensorFlow 略显严格的把控，目前各大公司都在开发自己的深度学习框架。

点评：不完美但最流行的深度学习框架，社区强大，适合生产环境。

1.2.3 Keras

Keras 是一个高层神经网络 API，由纯 Python 编写而成并使用 TensorFlow、Theano 及 CNTK 作为后端。Keras 为支持快速实验而生，能够把想法迅速转换为结果。Keras 应该是深度学习框架之中最容易上手的一个，它提供了一致而简洁的 API，能够极大地减少一般应用下用户的工作量，避免用户重复造轮子。

严格意义上讲，Keras 并不能称为一个深度学习框架，它更像一个深度学习接口，它构建于第三方框架之上。Keras 的缺点很明显：过度封装导致丧失灵活性。Keras 最初作为 Theano 的高级 API 而诞生，后来增加了 TensorFlow 和 CNTK 作为后端。为了屏蔽后端的差异性，提供一致的用户接口，Keras 做了层层封装，导致用户在新增操作或是获取底层的数据信息时过于困难。同时，过度封装也使得 Keras 的程序过于缓慢，许多 BUG 都隐藏于封装之中，在绝大多数场景下，Keras 是本节介绍的所有框架中最慢的一个。

学习 Keras 十分容易，但是很快就会遇到瓶颈，因为它缺少灵活性。另外，在使用 Keras 的大多数时间里，用户主要是在调用接口，很难真正学习到深度学习的内容。

点评：入门最简单，但是不够灵活，使用受限。

1.2.4 Caffe/Caffe2

Caffe 的全称是 Convolutional Architecture for Fast Feature Embedding，它是一个清晰、高效的深度学习框架，核心语言是 C++，它支持命令行、Python 和 MATLAB 接口，既可以在 CPU 上运行，也可以在 GPU 上运行。

Caffe 的优点是简洁快速，缺点是缺少灵活性。不同于 Keras 因为太多的封装导致灵活性丧失，Caffe 灵活性的缺失主要是因为它的设计。在 Caffe 中最主要的抽象对象是层，每实现一个新的层，必须要利用 C++ 实现它的前向传播和反向传播代码，而如果想要新层运行在 GPU 上，还需要同时利用 CUDA 实现这一层的前向传播和反向传播。这种限制使得不熟悉 C++ 和 CUDA 的用户扩展 Caffe 十分困难。

Caffe 凭借其易用性、简洁明了的源码、出众的性能和快速的原型设计获取了众多用户，曾经占据深度学习领域的半壁江山。但是在深度学习新时代到来之时，Caffe 已经表现出明显的力不从心，诸多问题逐渐显现（包括灵活性缺失、扩展难、依赖众多环

境难以配置、应用局限等）。尽管现在在 GitHub 上还能找到许多基于 Caffe 的项目，但是新的项目已经越来越少。

Caffe 的作者从加州大学伯克利分校毕业后加入了 Google，参与过 TensorFlow 的开发，后来离开 Google 加入 FAIR，担任工程主管，并开发了 Caffe2。Caffe2 是一个兼具表现力、速度和模块性的开源深度学习框架。它沿袭了大量的 Caffe 设计，可解决多年来在 Caffe 的使用和部署中发现的瓶颈问题。Caffe2 的设计追求轻量级，在保有扩展性和高性能的同时，Caffe2 也强调了便携性。Caffe2 从一开始就以性能、扩展、移动端部署作为主要设计目标。Caffe2 的核心 C++ 库能提供速度和便携性，而其 Python 和 C++ API 使用户可以轻松地在 Linux、Windows、iOS、Android，甚至 Raspberry Pi 和 NVIDIA Tegra 上进行原型设计、训练和部署。

Caffe2 继承了 Caffe 的优点，在速度上令人印象深刻。Facebook 人工智能实验室与应用机器学习团队合作，利用 Caffe2 大幅加速机器视觉任务的模型训练过程，仅需 1 小时就训练完 ImageNet 这样超大规模的数据集。然而尽管已经发布半年多，开发一年多，Caffe2 仍然是一个不太成熟的框架，官网至今没提供完整的文档，安装也比较麻烦，编译过程时常出现异常，在 GitHub 上也很少找到相应的代码。

极盛的时候，Caffe 占据了计算机视觉研究领域的半壁江山，虽然如今 Caffe 已经很少用于学术界，但是仍有不少计算机视觉相关的论文使用 Caffe。由于其稳定、出众的性能，不少公司还在使用 Caffe 部署模型。Caffe2 尽管做了许多改进，但是还远没有达到替代 Caffe 的地步。

点评：文档不够完善，但性能优异，几乎全平台支持（Caffe2），适合生产环境。

1.2.5 MXNet

MXNet 是一个深度学习库，支持 C++、Python、R、Scala、Julia、MATLAB 及 JavaScript 等语言；支持命令和符号编程；可以运行在 CPU、GPU、集群、服务器、台式机或者移动设备上。MXNet 是 CXXNet 的下一代，CXXNet 借鉴了 Caffe 的思想，但是在实现上更干净。在 2014 年的 NIPS 上，同为上海交大校友的陈天奇与李沐碰头，讨论到各自在做深度学习 Toolkits 的项目组，发现大家普遍在做很多重复性的工作，例如文件 loading 等。于是他们决定组建 DMLC〔Distributied（Deep）Machine Learning Community〕，号召大家一起合作开发 MXNet，发挥各自的特长，避免重复造轮子。

MXNet 以其超强的分布式支持，明显的内存、显存优化为人所称道。同样的模型，MXNet 往往占用更小的内存和显存，并且在分布式环境下，MXNet 展现出了明显优于其他框架的扩展性能。

由于 MXNet 最初由一群学生开发，缺乏商业应用，极大地限制了 MXNet 的使用。2016 年 11 月，MXNet 被 AWS 正式选择为其云计算的官方深度学习平台。2017 年 1 月，MXNet 项目进入 Apache 基金会，成为 Apache 的孵化器项目。

尽管 MXNet 拥有最多的接口，也获得了不少人的支持，但其始终处于一种不温不火的状态。个人认为这在很大程度上归结于推广不给力及接口文档不够完善。MXNet 长期处于快速迭代的过程，其文档却长时间未更新，导致新手用户难以掌握 MXNet，老用户常常需要查阅源码才能真正理解 MXNet 接口的用法。

为了完善 MXNet 的生态圈，推广 MXNet，MXNet 先后推出了包括 MinPy、Keras 和 Gluon 等诸多接口，但前两个接口目前基本停止了开发，Gluon 模仿 PyTorch 的接口设计，MXNet 的作者李沐更是亲自上阵，在线讲授如何从零开始利用 Gluon 学习深度学习，诚意满满，吸引了许多新用户。

点评：文档略混乱，但分布式性能强大，语言支持最多，适合 AWS 云平台使用。

1.2.6 CNTK

2015 年 8 月，微软公司在 CodePlex 上宣布由微软研究院开发的计算网络工具集 CNTK 将开源。5 个月后，2016 年 1 月 25 日，微软公司在他们的 GitHub 仓库上正式开源了 CNTK。早在 2014 年，在微软公司内部，黄学东博士和他的团队正在对计算机能够理解语音的能力进行改进，但当时使用的工具显然拖慢了他们的进度。于是，一组由志愿者组成的开发团队构想设计了他们自己的解决方案，最终诞生了 CNTK。

根据微软开发者的描述，CNTK 的性能比 Caffe、Theano、TensoFlow 等主流工具都要强。CNTK 支持 CPU 和 GPU 模式，和 TensorFlow/Theano 一样，它把神经网络描述成一个计算图的结构，叶子节点代表输入或者网络参数，其他节点代表计算步骤。CNTK 是一个非常强大的命令行系统，可以创建神经网络预测系统。CNTK 最初是出于在 Microsoft 内部使用的目的而开发的，一开始甚至没有 Python 接口，而是使用了一种几乎没什么人用的语言开发的，而且文档有些晦涩难懂，推广不是很给力，导致现在用户比较少。但就框架本身的质量而言，CNTK 表现得比较均衡，没有明显的短板，并且在语音领域效果比较突出。

点评：社区不够活跃，但是性能突出，擅长语音方面的相关研究。

1.2.7 其他框架

除了上述的几个框架，还有不少的框架，都有一定的影响力和用户。比如百度开源的 PaddlePaddle，CMU 开发的 DyNet，简洁无依赖符合 C++11 标准的 tiny-dnn，使用 Java 开发并且文档极其优秀的 Deeplearning4J，还有英特尔开源的 Nervana，Amazon 开源的 DSSTNE。这些框架各有优缺点，但是大多流行度和关注度不够，或者局限于一定的领域，因此不做过多的介绍。此外，还有许多专门针对移动设备开发的框架，如 CoreML、MDL，这些框架纯粹为部署而诞生，不具有通用性，也不适合作为研究工具，同样不做介绍。

1.3 属于动态图的未来

2016 年，随着 TensorFlow 的如日中天，几乎所有人都觉得深度学习框架之争接近尾声，但 2017 年却迎来了基于动态图的深度学习框架的爆发。

几乎所有的框架都是基于计算图[①]的，而计算图又可以分为静态计算图和动态计算图，静态计算图先定义再运行（define and run），一次定义多次运行，而动态计算图是在运行过程中被定义的，在运行的时候构建（define by run），可以多次构建多次运行。PyTorch 和 TensorFlow 都是基于计算图的深度学习框架，PyTorch 使用的是动态图，而 TensorFlow 使用的是静态图。在 PyTorch 中每一次前向传播（每一次运行代码）都会创建一幅新的计算图，如图 1-4 所示。

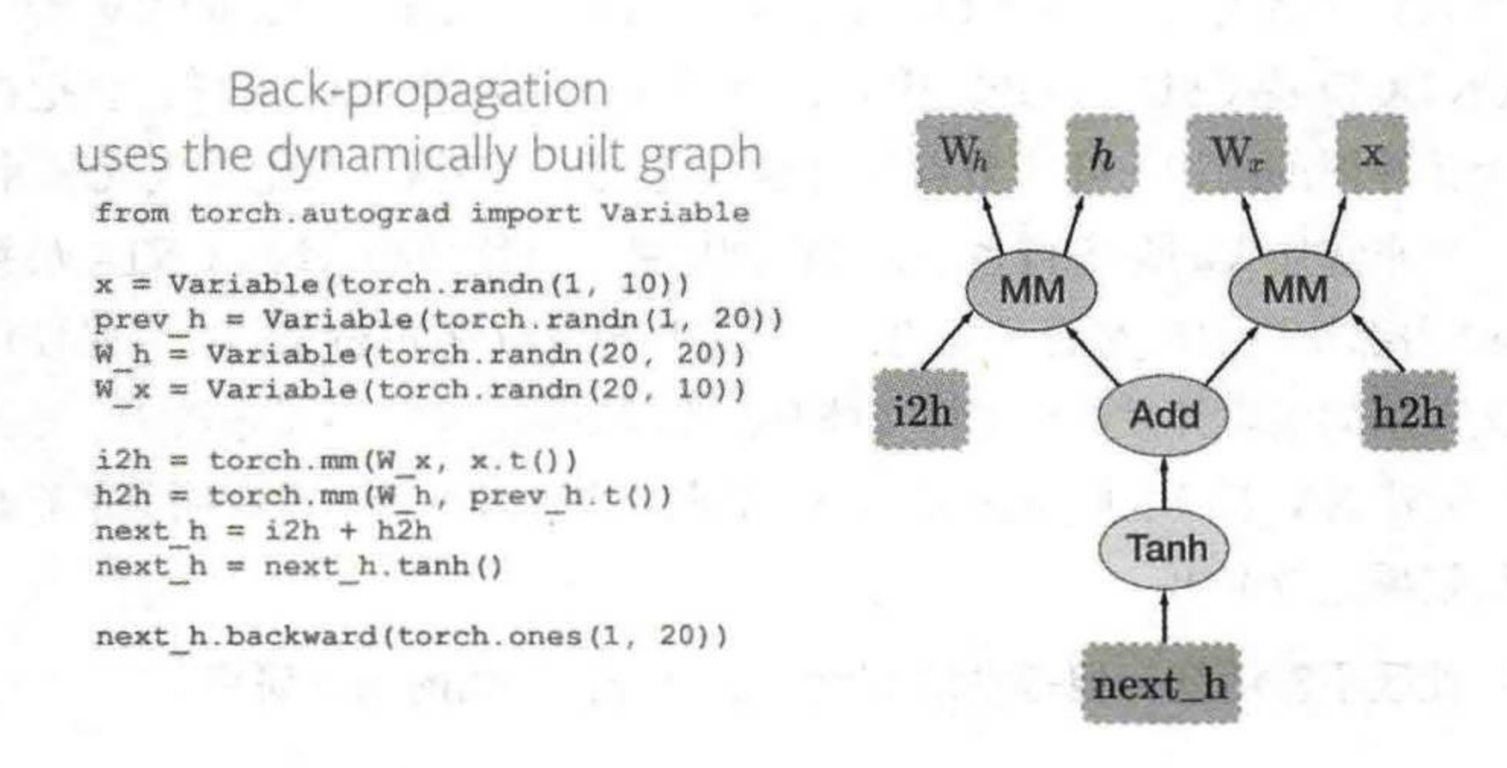


图 1-4 动态计算图，计算图随着代码运行而构建

[①] http://colah.github.io/posts/2015-08-Backprop/

静态图一旦创建就不能修改，而且静态图定义的时候，使用了特殊的语法，就像新学一门语言。这还意味着你无法使用 if、while、for-loop 等常用的 Python 语句。因此静态图框架不得不为这些操作专门设计语法，同时在构建图的时候必须把所有可能出现的情况都包含进去，这也导致了静态图过于庞大，可能占用过高的显存。动态图框架就没有这个问题，它可以使用 Python 的 if、while、for-loop 等条件语句，最终创建的计算图取决于你执行的条件分支。

我们来看看 if 条件语句在 TensorFlow 和 PyTorch 中的两种实现方式，第一个利用 PyTorch 动态图的方式实现。

```
import torch as t
from torch.autograd import Variable

N, D, H = 3, 4, 5

x = Variable(t.randn(N, D))
w1 = Variable(t.randn(D, H))
w2 = Variable(t.randn(D, H))

z = 10
if  z>0:
    y = x.mm(w1)
else:
    y = x.mm(w2)
```

第二个利用 TensorFlow 静态图的方式实现。

```
import tensorflow as tf
import numpy as np

N, D, H = 3, 4, 5
x = tf.placeholder(tf.float32, shape=(N, D))
z = tf.placeholder(tf.float32, shape=None)
w1 = tf.placeholder(tf.float32, shape= (D, H))
w2 = tf.placeholder(tf.float32, shape=(D,H))

def f1():
    return tf.matmul(x,w1)
def f2():
```

```
    return tf.matmul(x,w2)
y = tf.cond(tf.less(z,0), f1, f2)

with tf.Session() as sess:
    values = {
            x: np.random.randn(N,D),
            z: 10,
            w1: np.random.randn(D, H),
            w2: np.random.randn(D, H)
    }
    y_val = sess.run(y, feed_dict=values)
```

可以看出，PyTorch 的实现方式完全和 Python 的语法一致，简洁直观；而 TensorFlow 的实现不仅代码冗长，而且十分不直观。

动态计算图的设计思想正被越来越多人所接受，2017 年 1 月 20 日前后，先后有三款深度学习框架发布：PyTorch、MinPy 和 DyNet，这三个框架都是基于动态图的设计模式。PyTorch 便是其中的佼佼者，至今已成为动态图框架的代表。在 PyTorch 之前，Chainer 就以动态图思想设计框架，并获得用户的一致好评，然而 Chainer 是由日本科学家开发的，开发人员和文档都偏向于日本本土，没有很好地做推广。PyTorch 的发布让许多用户第一次发现原来深度学习框架可以如此灵活、如此容易、还如此快速。

动态图的思想直观明了，更符合人的思考过程。动态图的方式使得我们可以任意修改前向传播，还可以随时查看变量的值。如果说静态图框架好比 C++，每次运行都要编译才行（`session.run`），那么动态图框架就是 Python，动态执行，可以交互式查看修改。动态图的这个特性使得我们可以在 IPython 和 Jupyter Notebook 上随时查看和修改变量，十分灵活。

动态图带来的另外一个优势是调试更容易，在 PyTorch 中，代码报错的地方，往往就是你写错代码的地方，而静态图需要先根据你的代码生成 Graph 对象，然后在`session.run()`时报错，这种报错几乎很难找到对应的代码中真正错误的地方。

1.4 为什么选择 PyTorch

这么多深度学习框架，为什么选择 PyTorch 呢？

因为 PyTorch 是当前难得的简洁优雅且高效快速的框架。在笔者眼里，PyTorch 达到目前深度学习框架的最高水平。当前开源的框架中，没有哪一个框架能够在灵活性、

易用性、速度这三个方面有两个能同时超过 PyTorch。下面是许多研究人员选择 PyTorch 的原因。

- **简洁**：PyTorch 的设计追求最少的封装，尽量避免重复造轮子。不像 TensorFlow 中充斥着 session、graph、operation、name_scope、variable、tensor、layer 等全新的概念，PyTorch 的设计遵循 tensor→variable(autograd)→nn.Module 三个由低到高的抽象层次，分别代表高维数组（张量）、自动求导（变量）和神经网络（层/模块），而且这三个抽象之间联系紧密，可以同时进行修改和操作。

  简洁的设计带来的另外一个好处就是代码易于理解。PyTorch 的源码只有 TensorFlow 的十分之一左右，更少的抽象、更直观的设计使得 PyTorch 的源码十分易于阅读。在笔者眼里，PyTorch 的源码甚至比许多框架的文档更容易理解。

- **速度**：PyTorch 的灵活性不以速度为代价，在许多评测中，PyTorch 的速度表现胜过 TensorFlow 和 Keras 等框架[①,②]。框架的运行速度和程序员的编码水平有极大关系，但同样的算法，使用 PyTorch 实现的那个更有可能快过用其他框架实现的。

- **易用**：PyTorch 是所有的框架中面向对象设计的最优雅的一个。PyTorch 的面向对象的接口设计来源于 Torch，而 Torch 的接口设计以灵活易用而著称，Keras 作者最初就是受 Torch 的启发才开发了 Keras。PyTorch 继承了 Torch 的衣钵，尤其是 API 的设计和模块的接口都与 Torch 高度一致。PyTorch 的设计最符合人们的思维，它让用户尽可能地专注于实现自己的想法，即所思即所得，不需要考虑太多关于框架本身的束缚。

- **活跃的社区**：PyTorch 提供了完整的文档，循序渐进的指南，作者亲自维护的论坛[③]供用户交流和求教问题。Facebook 人工智能研究院对 PyTorch 提供了强力支持，作为当今排名前三的深度学习研究机构，FAIR 的支持足以确保 PyTorch 获得持续的开发更新，不至于像许多由个人开发的框架那样昙花一现。

  在 PyTorch 推出不到一年的时间内，各类深度学习问题都有利用 PyTorch 实现的解决方案在 GitHub 上开源。同时也有许多新发表的论文采用 PyTorch 作为论文实现的工具，PyTorch 正在受到越来越多人的追捧[④]。

如果说 TensorFlow 的设计是“Make It Complicated”，Keras 的设计是“Make It Complicated And Hide It”，那么 PyTorch 的设计真正做到了“Keep it Simple，Stupid”。简洁即是美。

使用 TensorFlow 能找到很多别人的代码，使用 PyTorch 能轻松实现自己的想法。

---

① https://github.com/tensorflow/tensorflow/issues/9322

② https://github.com/tensorflow/tensorflow/issues/7065

③ https://discuss.pytorch.org/

④ https://github.com/ritchieng/the-incredible-pytorch

1.5 星火燎原

尽管 2017 年 TensorFlow 的新闻依旧铺天盖地，但是我们能很明显地感受到 PyTorch 正越来越流行。2017 年的年度深度学习框架属于 PyTorch。

2017 年 1 月 18 日，PyTorch 发布。

2017 年 2 月，最著名的深度学习课程，斯坦福大学的 CS231n 公布了课程大纲，将发布才一个多月的 PyTorch 选为课程教学框架，使用 PyTorch 布置作业，并提供教程。

2017 年 3 月 31 日 ~ 4 月 12 日，奖金高达 100 万美元的 Kaggle 数据科学竞赛（Data Science Bowl 2017）落幕，名为 grt123 的队伍使用刚发布不久的 PyTorch 以较大优势夺冠。

2017 年 4 月 25 日，深度学习年度盛会 ICLR 2017 在法国举行，PyTorch 获得了极大关注。短短三个月，PyTorch 就获得了极大的认可。

2017 年下半年，PyTorch 的新闻越来越多，关注度持续提升。PyTorch 0.2 版本发布，新增分布式训练、高阶导数、自动广播法则等众多新特性。

艾伦人工智能研究院开源了 AllenNLP，基于 PyTorch 轻松构建 NLP 模型，几乎适用于任何 NLP 问题。

Facebook 和微软宣布，推出 Open Neural Network Exchange（ONNX，开放神经网络交换）格式，这是一个用于表示深度学习模型的标准，可使模型在不同框架之间进行转移。ONNX 是迈向开放生态系统的第一步，ONNX 目前支持 PyTorch、Caffe2 和 CNTK，未来会支持更多的框架。除了 Facebook 和微软，AMD、ARM、华为、IBM、英特尔、高通也宣布支持 ONNX。

著名的深度学习教育网站 fast.ai 宣布，他们的下一个课程，将完全基于 PyTorch，抛弃原来的 TensorFlow 和 Keras。

不同于 Google 在各个场合大力宣传 TensorFlow，PyTorch 的流行更多是由于其简洁优雅的设计吸引了用户，几乎每一个 PyTorch 用户都会自发地宣传 PyTorch。TensorFlow 确实流行，但正如 PyTorch slack 中用户制作的一张调侃图（如图 1-5 所示）所说，如果你无法用 TensorFlow 快速实现你的想法，不要因为 TensorFlow 最流行就使用它。

就在 PyTorch 发布不久后，OpenAI 的科学家，Tesla 的 AI 部门主管 Andrej Karpathy 就发了一篇意味深长的 Twitter：

> Matlab is so 2012. Caffe is so 2013. Theano is so 2014. Torch is so 2015. TensorFlow is so 2016. :D

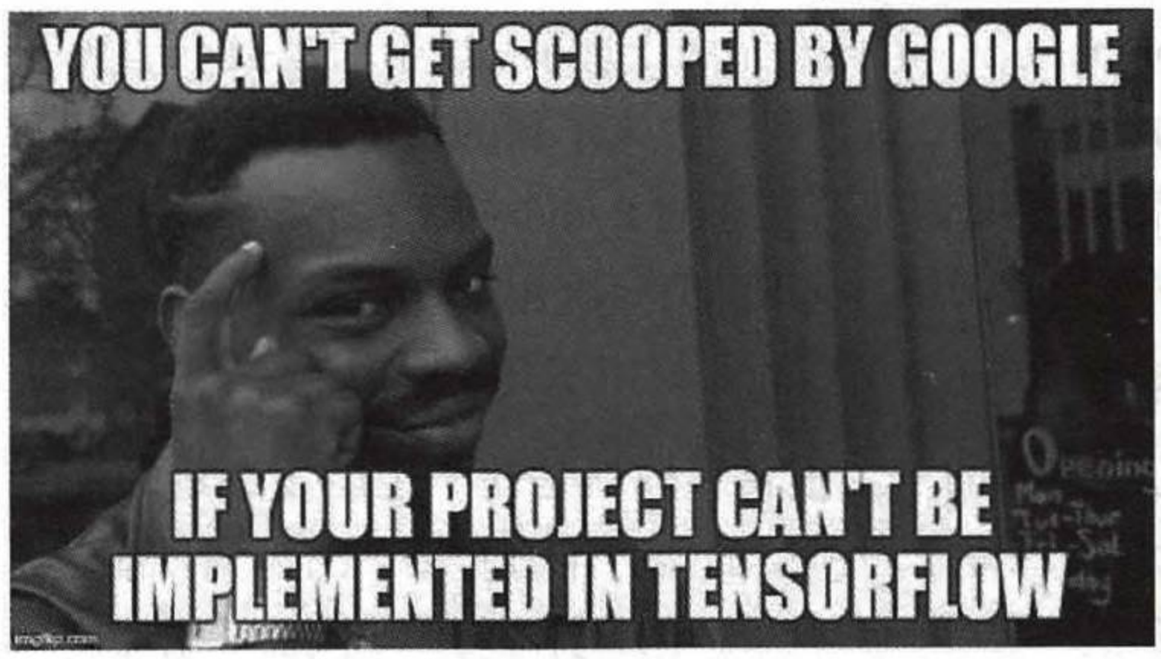


图 1-5　如果不能用 TensorFlow 实现自己的想法，就不用它

2017 年 5 月，Andrej Karpathy 又发了一篇 Twitter，调侃道：

> I've been using PyTorch a few months now. I've never felt better. I have more energy. My skin is clearer. My eye sight has improved.

2017 年马上要过去了，你还在等什么？

1.6　fast.ai 放弃 Keras+TensorFlow 选择 PyTorch

fast.ai CEO Jeremy Howard 在 fast.ai 官网[①]宣布下一个课程将完全基于一个使用 PyTorch 开发的框架，抛弃原来的 TensorFlow 和 Keras 框架。

官网通告部分翻译如下。

> 我们在开发《面向程序员的前沿深度学习》这门课的时候遇到瓶颈，因为原来选的 TensorFlow 和 Keras 框架让我们处处碰壁。例如，现在自然语言处理中最重要的技术，大概是 Attention 模型。可是我们发现，当时在 Keras 上没有 Attention 模型的有效实现，而 TensorFlow 实现缺乏必要文档、不断的变化，而且过于复杂以至于难以理解。于是我们决定利用 Keras 实现 Attention 模型，这花了我们好长时间，调试过程也十分痛苦。
>
> 随后，我们开始尝试实现 dynamic teacher forcing，这是神经网络翻译系统的关键，但无论是在 Keras 里还是在 TensorFlow 里，我们都找不到这个模型的参考实现，而我们自己尝试实现的系统直接就不能用。

---

① http://www.fast.ai/2017/09/08/introducing-pytorch-for-fastai/

这时，PyTorch 的第一个预发布版出现了。这个新框架不是基于静态计算图，而是一个动态的框架，这为我们带来了新的希望。动态框架让我们在开发自己的神经网络时，只需要写普通的 Python 代码，像正常用 Python 一样调试。我们都没有专门学习 PyTorch，第一次用 PyTorch，就用它实现了 Attention 模型和 dynamic teacher forcing，只用了几个小时。

上文提到的那门课的一个重要目标就是让学生能读最近的论文，然后实现它们。自己实现深度学习模型的能力十分重要，因为到目前为止，我们十分关注学术研究，对深度学习的应用反倒很有限。因此，用深度学习解决很多现实世界问题的时候，不仅需要了解基础技术，还要能针对特定的问题和数据实现定制化的深度学习模型。

PyTorch，让学生能充分利用普通 Python 代码的灵活性和能力构建、训练神经网络。这样，他们就能解决更广泛的问题。

PyTorch 的另一个好处是，它能让学生更深入地了解每个算法中发生了什么。用 TensorFlow 那样的静态计算图库，你一旦声明性地表达了你的计算，就把它发送到了 GPU，整个处理过程就是一个黑箱。但是通过动态的方法，你可以完全进入计算的每一层，清楚地看到正在发生的情况。我们认为学习深度学习的最佳途径就是通过编程、实验，动态的方法正是我们的学生所需要的。

令我们惊奇的是，我们还发现很多模型在 PyTorch 上训练比在 TensorFlow 上更快。这和我们所熟知的“静态计算图能带来更多优化，所以应该性能更好”恰恰相反。

在实践中我们看到，有些模型快一点，有些慢一点，每个月都不一样。问题的关键似乎在以下两点。

- PyTorch 提高了开发人员的生产力和调试经验，因此可以带来更快的开发迭代和更好的实现。
- PyTorch 中更小、更集中的开发团队不会对每个功能都进行微优化，而是在整体设计上寻求“大胜”。

笔者认为 fast.ai 的这篇博客很好地对比了 PyTorch 和 Keras+TensorFlow。

- 许多论文方法都没有 TensorFlow 的开源实现，或者实现的质量不如人意，并且 TensorFlow 和 Keras 难以调试。
- PyTorch 容易上手（fast.ai 研究人员第一次用 PyTorch 就实现了 Attention 模型和 dynamic teacher forcing）。

- PyTorch 易于调试，十分灵活、透明；TensorFlow 和 Keras 难以调试，就像一个黑箱。
- PyTorch 比 TensorFlow 快。

2 快速入门

本章主要介绍两个内容，2.1 节介绍如何安装 PyTorch，以及如何配置学习环境；2.2 节将带领读者快速浏览 PyTorch 中主要的内容，给读者一个关于 PyTorch 的大致印象。

2.1 安装与配置

2.1.1 安装 PyTorch

PyTorch 是一款以 Python 语言主导开发的轻量级深度学习框架。在使用 PyTorch 之前，需要安装 Python 环境及其 pip 包管理工具，推荐使用 Virtualenv 配置虚拟 Python 环境。本书中所有代码使用 PyTorch 0.3 版本，同时兼容 Python2 和 Python3，并全部在 Python2 环境中运行得到最终结果，在 Python3 环境测试未报错，但并不保证得到和 Python2 环境一致的结果。另外，本书默认使用 Linux 作为开发环境。

为方便用户安装使用，PyTorch 官方提供了多种安装方法。本节将介绍几种常用的安装方式，读者可以根据自己的需求选用。

使用 pip 安装

目前，使用 pip 安装 PyTorch 二进制包是最简单、最不容易出错，同时也是最适合新手的安装方式。从 PyTorch 官网[①]选择操作系统、包管理器 pip、Python 版本及 CUDA

[①] http://pytorch.org/

版本，会对应不同的安装命令，如图 2-1 所示。

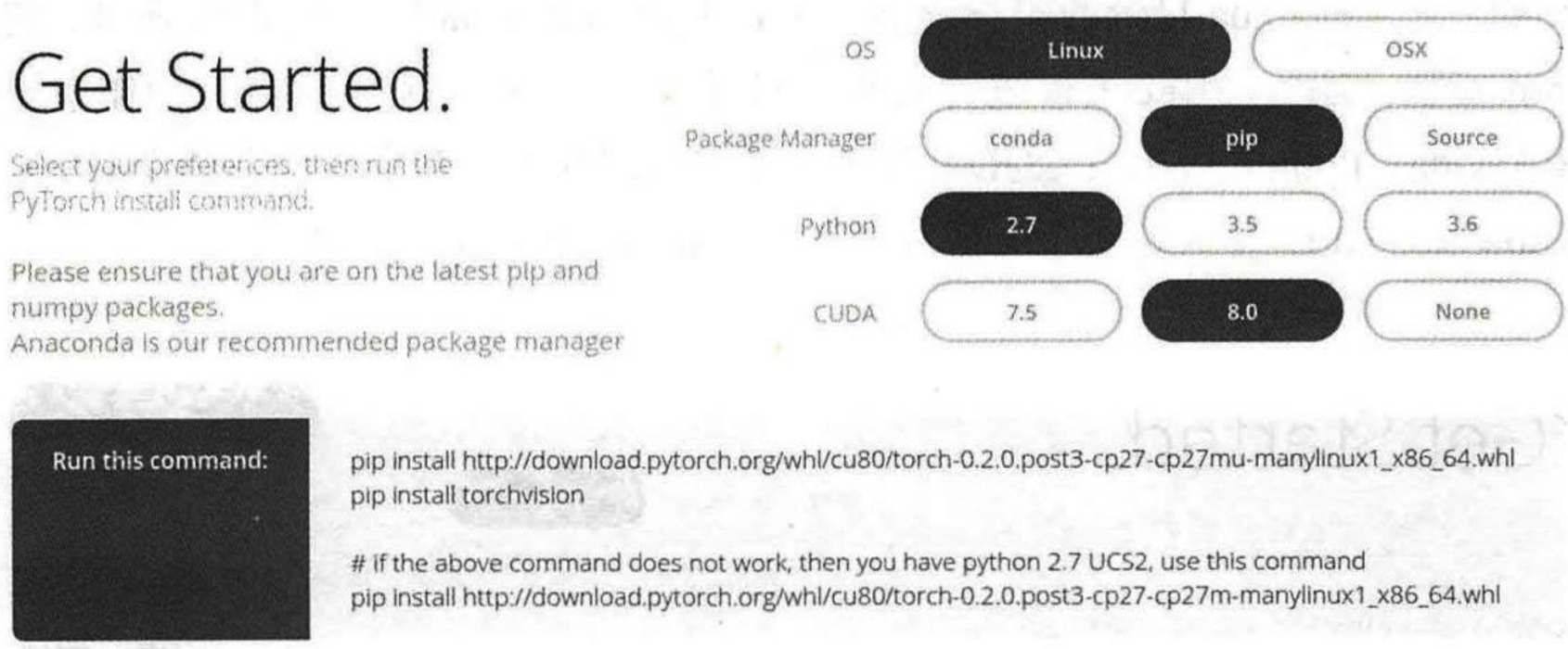


图 2-1　使用 pip 安装 PyTorch

以 Linux 平台、pip 安装、Python2.7 及 CUDA8.0 为例，安装命令如下（根据不同系统配置，可将 pip 改为 pip2 或 pip3）。

```
pip install http://download.pytorch.org/whl/cu80/torch-0.2.0.post3-cp27-
cp27mu-manylinux1_x86_64.whl
pip install torchvision
```

安装好 PyTorch 之后，还需安装`Numpy`，安装命令如下。

```
pip install --upgrade numpy
```

或者使用系统自带的包管理器（apt，yum 等）安装 numpy，然后使用 pip 升级。

```
apt install python-numpy
pip install --upgrade numpy
```

全部安装完成后，打开 Python，运行如下命令。

```
>>> import torch as t
```

没有报错则表示 PyTorch 安装成功。

安装过程中需要注意以下几点。

（1）PyTorch 对应的 Python 包名为`torch`而非`pytorch`。

（2）若需使用 GPU 版本的 PyTorch，需要先配置英伟达显卡驱动，再安装 PyTorch。

使用 conda 安装

conda 是 Anaconda 自带的包管理器。如果使用 Anaconda 作为 Python 环境，则除了使用 pip 安装，还可使用 conda 进行安装。同样，在 PyTorch 官网中选择操作系统、包管理器 conda、Python 版本及 CUDA 版本，对应不同的安装命令。我们以在 OS X 下安装 Python3.6、CPU 版本的 PyTorch 为例介绍，如图 2-2 所示。

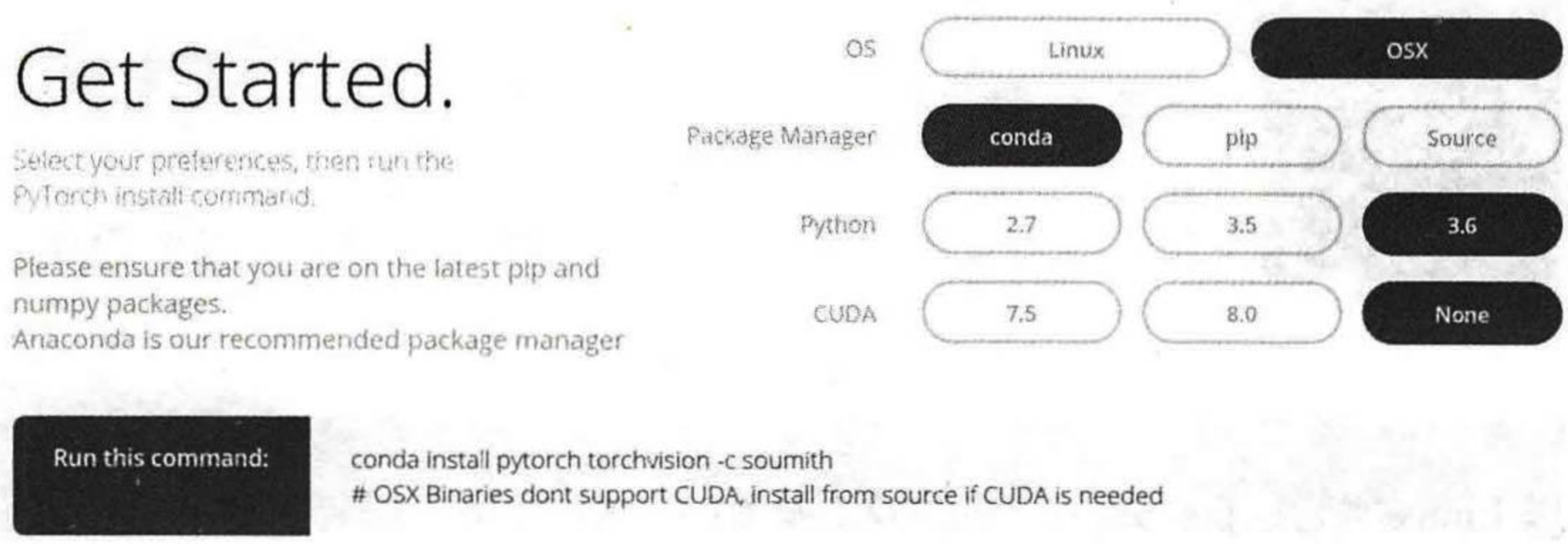


图 2-2　使用 conda 安装 PyTorch

安装命令如下：

```
conda install pytorch torchvision -c soumith
```

conda 的安装速度可能较慢，建议国内用户，尤其是教育网用户把 conda 源设置为清华 tuna。在命令行输入如下命令即可完成修改。

```
conda config --add channels  https://mirrors.tuna.tsinghua.edu.cn/anaconda
/pkgs/free/
conda config --set show_channel_urls yes
```

即使是使用 Anaconda 的用户，也建议使用 pip 安装 PyTorch，一劳永逸，而且不易出错。

从源码编译安装

不建议新手从源码编译安装，因为这种安装方式对环境比较敏感，需要用户具备一定的编译安装知识，以及应对错误的能力。但若想使用官方未发布的最新功能，或某个 BUG 刚修复，官方还未提供二进制安装包，而读者又亟需这个补丁，此时就需要从 GitHub 上下载源码编译安装。

从源码编译安装，推荐使用 Anaconda 环境。如果想使用 GPU 版本，则需安装 CUDA 7.5 及以上和 cuDNN v5 及以上（如果已装有 CUDA，但不想被 PyTorch 使用，只需设置环境变量`NO_CUDA=1`）。

首先，安装可选依赖。

1）Linux

```
export CMAKE_PREFIX_PATH="$(dirname $(which conda))/../"

# 安装基础依赖
conda install numpy pyyaml mkl setuptools cmake gcc cffi

# 为GPU添加LAPACK支持
conda install -c soumith magma-cuda80 # 如果使用CUDA 7.5，则将magma-cuda80
改成magma-cuda75
```

2）OS X

```
export CMAKE_PREFIX_PATH="$(dirname $(which conda))/../"
conda install numpy pyyaml setuptools cmake cffi
```

其次，下载 PyTorch 源码。

```
git clone https://github.com/pytorch/pytorch
cd pytorch
```

最后，完成编译安装。

1）Linux

```
python setup.py install
```

2）OS X

```
MACOSX_DEPLOYMENT_TARGET=10.9 CC=clang CXX=clang++ python setup.py install
```

使用 Docker 部署

Docker 是一个开源的应用容器引擎，让开发者可以打包他们的应用及依赖包到一个可移植的容器中，并发布到任何流行的 Linux 机器上，也可实现虚拟化。PyTorch 官方提供了 Dockerfile，支持 CUDA 和 cuDNN v6。可通过如下命令构建 Docker 镜像。

```
docker build -t pytorch-cudnnv6 .
```

通过如下命令运行：

```
nvidia-docker run --rm -ti --ipc=host pytorch-cudnnv6
```

注意：PyTorch 中数据加载（Dataloader）使用了大量的共享内存，可能超出容器限制，需设置`--shm-size`选项或使用`--ipc=host`选项解决。

Windows 用户安装 PyTorch

PyTorch 官方尚不支持 Windows 平台，推荐 Windows 用户在虚拟机中安装 Linux，或者使用双系统。尽管现在官方还未支持 Windows 系统，但开发者的热情高涨[①]，并已有用户提供了对应的 Anaconda 安装包，只需按照如下命令执行即可安装。

```
conda install -c peterjc123 pytorch=0.2.0
```

但需注意这个版本较不稳定，未获得官方的维护，因此不建议新手使用。关于 Windows 版本 PyTorch 的更多信息，请查阅蒲嘉宸的知乎专栏[②]。

2.1.2 学习环境配置

工欲善其事，必先利其器，在从事科学计算相关工作时，IPython 和 Jupyter 是两个必不可少的工具。推荐使用 IPython 和 Jupyter Notebook 学习本书的示例代码。

IPython

IPython 是一个交互式计算系统，可认为是增强版的 Python Shell，提供强大的 REPL（交互式解析器）功能。对于从事科学计算的用户来说，它提供方便的可交互式学习及调试功能。

安装 IPython 十分简单，对于 Python2 的用户，安装命令如下。

```
pip2 install ipython==5.1
```

IPython 5.x 是最后一个支持 Python2 的 IPython。Python3 的用户可通过如下命令安装最新版 IPython 6.0。

```
pip install ipython
```

安装完成后，在命令行输入`ipython`即可启动 IPython，启动界面如下。

① https://github.com/pytorch/pytorch/issues/494

② https://zhuanlan.zhihu.com/p/26871672

```
Python 2.7.13 (default, Feb 11 2017, 12:22:40)
Type "copyright", "credits" or "license" for more information.

IPython 5.1.0 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

In [1]: import torch as t
```

输入exit命令或者按“Ctrl+D”快捷键可退出 IPython。IPython 有许多强大的功能，其中最常用的功能如下。

自动补全 IPython 最方便的功能之一是自动补全，输入一个函数或者变量的前几个字母，按下Tab键，就能实现自动补全，如图 2-3 所示。

```
In [1]: import torch as t

In [2]: t.Float
                t.FloatStorage
                t.FloatTensor
```

图 2-3 IPython 自动补全

内省 所谓内省，主要是指在 Runtime 时获得一个对象的全部类型信息，这对实际的学习有很大帮助。输入某一个函数或者模块之后，接着输入?可看到它对应的帮助文档，有些帮助文档比较长，可能跨页，这时可按空格键翻页，输入q退出。例如：

```
In [1]: import torch as t

In [2]: t.abs? #查看abs函数的文档
Docstring:
abs(input, out=None) -> Tensor

Computes the element-wise absolute value of the given :attr:`input` a
tensor.

Example::
```

```
    >>> torch.abs(torch.FloatTensor([-1, -2, 3]))
    FloatTensor([1, 2, 3])
Type:       builtin_function_or_method
```

在函数或模块名之后输入两个问号，例如`t.FloatTensor??`即可查看这个对象的源码，但只能查看对应 Python 的源码，无法查看 C/C++ 的源码。

快捷键 IPython 提供了很多快捷键。例如，按上箭头可以重新输入上一条代码；一直按上箭头，可以追溯到之前输入的代码。按“Ctrl+C”快捷键可以清空当前输入或停止运行的程序。常用的快捷键如表 2-1 所示。

表 2-1 IPython 常用快捷键

快捷键	功能
Ctrl+P 或上箭头	搜索之**前**命令历史中以当前输入文本开头的命令
Ctrl+N 或下箭头	搜索之**后**命令历史中以当前输入文本开头的命令
Ctrl+Shift+V	粘贴代码或代码块
Ctrl+A	跳转到行头
Ctrl+E	跳转到行尾

魔术方法 IPython 中还提供了一些特殊的命令，这些命令以%开头，称为魔术命令，例如可通过`%hist`查看在当前 IPython 下的输入历史等，示例如下。

```
In [1]: import torch as t

In [2]: a=t.Tensor(3,4)

In [3]: %timeit a.sum() # 检测某条语句的执行时间
1000000 loops, best of 3: 359 ns per loop
```

```
In [4]: %hist # 查看输入历史
import torch as t
a=t.Tensor(3,4)
%timeit a.sum()
%hist
```

```
In [5]: %paste # 执行粘贴板中的代码，如果只粘贴不执行使用Ctrl+Shift+V快捷
键
def add(x,y,z):
     return x+y+z
## -- End pasted text --

In [6]: %cat a.py # 查看某一个文件的内容，这个文件只有两行代码
b = a + 1
print(b.size())

In [7]: %run -i a.py # 执行文件，-i选项代表在当前命名空间中执行
                     # 此时会使用当前命名空间中的变量，结果也会返回至当前
                     命名空间
(3L,)

In [8]: b
Out[8]:

-4.5902e-22
 4.5577e-41
 1.1404e-11
[torch.FloatTensor of size 3]
```

和普通 Python 对象一样，魔术方法也支持自省，因此也可在命令后面加“?”或“??”来查看对应的帮助文档或源代码，例如通过%run?可查看它的使用说明。其他常用魔术命令如表 2-2 所示。

表 2-2　IPython 的常用魔术命令

命令	说明
%quickref	显示快速参考
%who	显示当前命名空间中的变量
%debug	进入调试模式（按 q 键退出）
%magic	查看所有魔术命令
%env	查看系统环境变量
%xdel	删除变量并删除其在 IPython 上的一切引用

“%xdel”与“del”的不同在于前者会删除其在 IPython 上的一切引用，具体例子如下。

```
In [1]: import torch as t

In [2]: a=t.Tensor(5,5)

In [3]: a

Out[3]:

1.00000e-16 *
  8.4887  0.0000  8.4887  0.0000  0.0000
  0.0000  0.0000  0.0000  0.0000  0.0000
  0.0000  0.0000  0.0000  0.0000  0.0000
  0.0000  0.0000  0.0000  0.0000  0.0000
  0.0000  0.0000  0.0000  0.0000  0.0000
[torch.FloatTensor of size 5x5]

In [4]: del a #并未彻底释放空间，因为这块空间还被Out[3]所引用

In [5]: c=t.Tensor(1000,1000)

In [6]: c
Out[6]:

    0     0     0  ...     0     0     0
    0     0     0  ...     0     0     0
    0     0     0  ...     0     0     0
       ...                   ...
    0     0     0  ...     0     0     0
    0     0     0  ...     0     0     0
    0     0     0  ...     0     0     0
[torch.FloatTensor of size 1000x1000]

In [7]: %xdel c

In [8]: Out [3]
```

```
Out[8]:

1.00000e-16 *
  8.4887  0.0000  8.4887  0.0000  0.0000
  0.0000  0.0000  0.0000  0.0000  0.0000
  0.0000  0.0000  0.0000  0.0000  0.0000
  0.0000  0.0000  0.0000  0.0000  0.0000
  0.0000  0.0000  0.0000  0.0000  0.0000
[torch.FloatTensor of size 5x5]

In [9]: Out[6]
---------------------------------------------------------------------------

KeyError                                  Traceback (most recent call last
)
<ipython-input-9-c0115ca8d9b5> in <module>()
----> 1 Out[6]

KeyError: 6
```

粘贴 IPython 支持多种格式的粘贴，除了`%paste`魔法方法，还可以直接粘贴多行代码、doctest 代码和 IPython 的代码，举例如下（下面的代码都使用“Ctrl+V”快捷键的方式直接粘贴。如果是 Linux 终端，则应该使用“`Ctrl+Shift+V`”快捷键直接粘贴，或者单击鼠标右键，选择“粘贴”选项）。

```
In [1]: In [1]: import torch as t
   ...:
   ...: In [2]: a=t.Tensor(5,5)
   ...:
   ...: In [3]: a
   ...:
Out[1]:

1.00000e-33 *
 -1.0299  0.0000  0.0001  0.0000  0.0000
  0.0000  0.0000  0.0000  0.0000  0.0000
  0.0000  0.0000  0.0000  0.0000  0.0000
```

```
  0.0000  0.0000  0.0000  0.0000  0.0000
  0.0000  0.0000  0.0000  0.0000  0.0000
[torch.FloatTensor of size 5x5]

In [2]: >>> import torch as t
   ...: >>> a=t.Tensor(5,5)
   ...: >>> a
   ...:
Out[2]:

1.00000e-33 *
 -1.0298  0.0000 -1.0298  0.0000  0.0000
  0.0000  0.0000  0.0000  0.0000  0.0000
  0.0000  0.0000  0.0000  0.0000  0.0000
  0.0000  0.0000  0.0000  0.0000  0.0000
  0.0000  0.0000  0.0000  0.0000  0.0000
[torch.FloatTensor of size 5x5]

In [3]: import torch as t
   ...: a = t.Tensor(5,5)
   ...: a
   ...:
Out[3]:

1.00000e-33 *
 -1.0298  0.0000 -1.0298  0.0000  0.0000
  0.0000  0.0000  0.0000  0.0000  0.0000
  0.0000  0.0000  0.0000  0.0000  0.0000
  0.0000  0.0000  0.0000  0.0000  0.0000
  0.0000  0.0000  0.0000  0.0000  0.0000
[torch.FloatTensor of size 5x5]
```

使用 IPython 进行调试 IPython 的调试器 ipdb 增强了 pdb，提供了很多实用功能，例如 Tab 键自动补全、语法高亮等。在 IPython 中进入 pdb 的最快速方式是使用魔术命令%debug，此时用户能够直接跳到报错的代码处，可通过 u、d 实现堆栈中的上下移动，常用的调试命令如表 2-3 所示。

表 2-3 ipdb 常用的调试命令

命令	功能
h(elp)	显示帮助信息，help command 显示这条命令的帮助信息
u(p)	在函数调用栈中向上移动
d(own)	在函数调用栈中向下移动
n(ext)	单步执行，执行下一步
s(tep)	单步进入当前函数调用
a(rgs)	查看当前调用函数的参数
l(ist)	查看当前行的上下文参考代码
b(reak)	在指定位置上设置断点
q(uit)	退出

debug 是一个重要功能，不仅在学习 PyTorch 时需要用到，在平时学习 Python 或使用 IPython 时也会经常使用。更多的 debug 功能，可通过h <命令>查看该命令的使用方法。

如果想在 IPython 之外使用 debug 功能，则需安装 ipdb（pip install ipdb），而后在需要进入调试的地方加上如下代码即可。

```
import ipdb
ipdb.set_trace()
```

当程序运行到这一步时，会自动进入 debug 模式。

Jupyter Notebook

Jupyter Notebook 是一个交互式笔记本，前身是 IPython Notebook，后来从 IPython 中独立出来，现支持运行 40 多种编程语言。对希望编写漂亮的交互式文档和从事科学计算的用户来说是一个不错的选择。

Jupyter Notebook 的使用方法与 IPython 非常类似，推荐使用 Jupyter Notebook 主要有如下三个原因。

- 更美观的界面：相比在终端下使用 IPython，Notebook 提供图形化操作界面，对新手而言更美观简洁。
- 更好的可视化支持：Notebook 与 Web 技术深度融合，支持在 Notebook 中直接可视化，这对需要经常绘图的科学运算实验来说很方便。

- 方便远程访问：在服务器端开启 Notebook 服务后，客户端只需有浏览器且能访问服务器，就可使用服务器上的 Notebook，这对于很多使用 Linux 服务器，但办公电脑使用 Windows 的人来说十分方便，避免了在本地配置环境的复杂流程。

安装 Jupyter 只需一条 pip 命令。

```
pip install jupyter
```

安装完成后，在命令行输入`jupyter notebook`命令即可启动 Jupyter，此时浏览器会自动弹出，并打开 Jupyter 主界面，也可手动打开浏览器，输入`http://127.0.0.1:8888`访问 Jupyter，界面如图 2-4 所示。

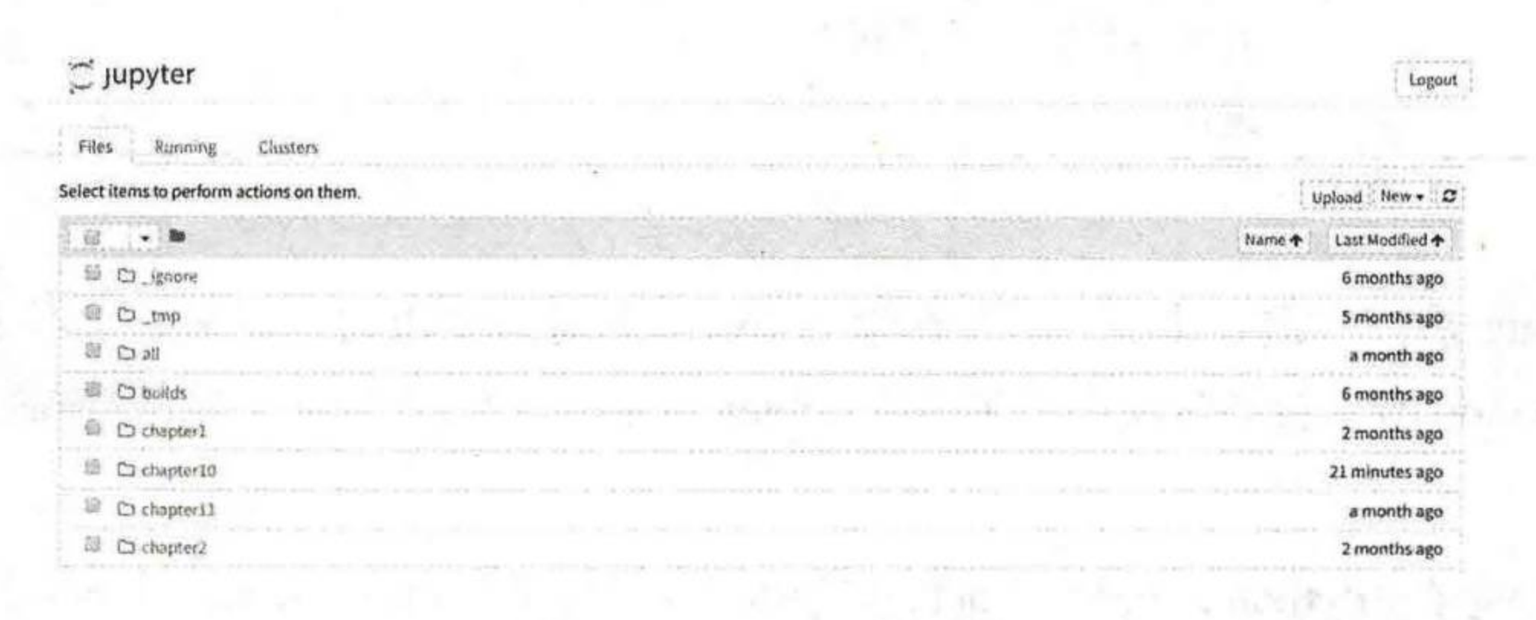


图 2-4　Jupyter 主页面

单击页面右上角的“new”选项，选择相应的 Notebook 类型（Python3/Python2），可新建一个 Notebook，在 In[] 后面的编辑区输入代码，按“Ctrl+Enter”快捷键，即可运行代码，如图 2-5 所示。

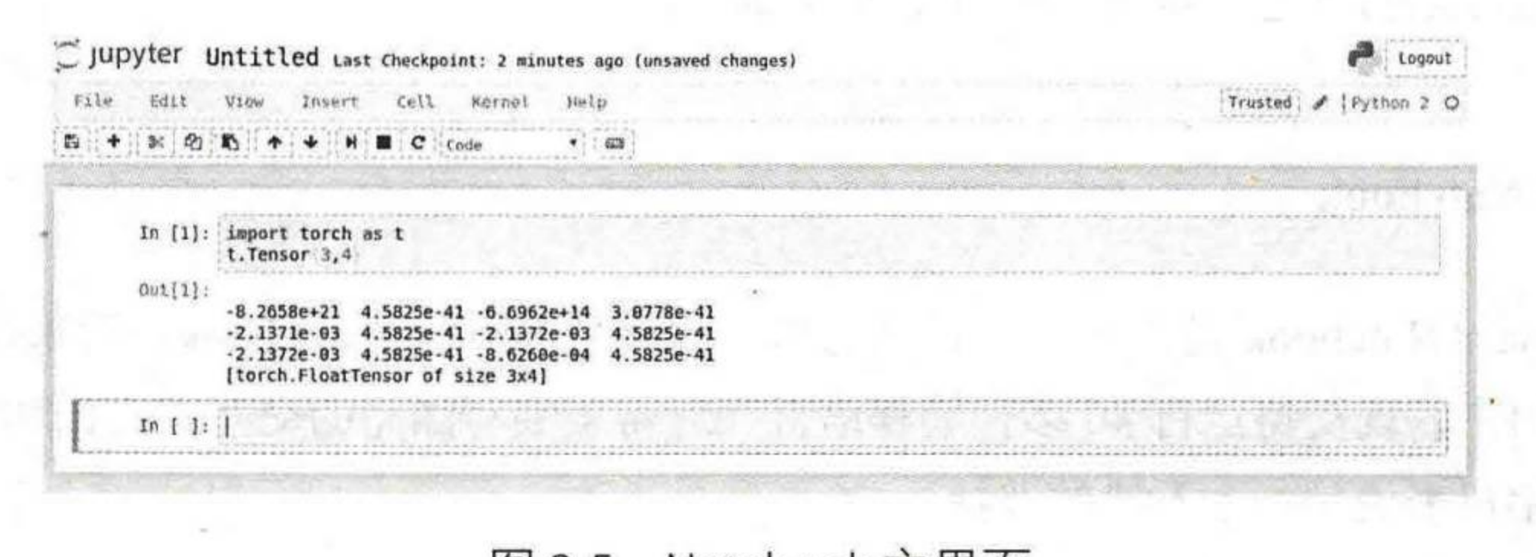


图 2-5　Notebook 主界面

远程访问服务器 Jupyter 的用户需要在服务器中搭建 Jupyter Notebook 服务，然后通过浏览器访问。可以根据需要对 Jupyter 设置访问密码。

首先，打开 IPython，设置密码，获取加密后的密码。

```
In [1]: from notebook.auth import passwd
```

```
In [2]: passwd()# 输入密码
Enter password:
Verify password:
Out[2]: 'sha1:f9c17b4cc163:43b6b4c8c....'
```

sha1:f9c17b... 即为加密后的密码，新建 jupyter_config.py，输入如下配置。

```
# 加密后的密码
c.NotebookApp.password = u'sha1:f9c17b4cc163:43b6b4c8c....'

# ::绑定所有IP地址，包括IPv4/IPv6的地址
# 如果只想绑定某个ip，改成对应的ip即可
c.NotebookApp.ip = '::'

# 绑定的端口号，如果该端口已经被占用，
# 会自动使用下一个端口号10000
c.NotebookApp.port = 9999
```

其次，启动 Jupyter Notebook 并指定配置文件，输入如下命令。

```
jupyter notebook --config=jupyter_config.py
```

最后，客户端打开浏览器，访问 url http://[服务器 ip]:9999 ，输入密码，即可访问 Jupyter。

若客户端浏览器无法打开 Jupyter，有可能是防火墙的缘故，输入如下命令开放对应的端口（若使用 IPv6，把命令 iptables 改成 ip6tables ）。

```
iptables -I INPUT -p tcp --dport 9999 -j ACCEPT
iptables save
```

Jupyter 的使用和 IPython 极为类似，我们介绍的 IPython 使用技巧对 Jupyter 基本都适用。它支持自动补全、内省、魔术方法、debug 等功能，但它的快捷键与 IPython 有较大不同，可通过菜单栏的【Help】→【Keyboard Shortcuts】查看详细的快捷键。

Jupyter 还支持很多功能，如 Markdown 语法、HTML、各种可视化等。更多关于 IPython 和 Jupyter Notebook 的使用技巧，读者可以从网上找到很多学习资源，这里只介绍一些最基础的、本书会用到的内容。

2.2 PyTorch 入门第一步

PyTorch 的简洁设计使得它易于入门，在深入介绍 PyTorch 之前，本节先介绍一些 PyTorch 的基础知识，使读者能够对 PyTorch 有一个大致的了解，并能够用 PyTorch 搭建一个简单的神经网络。部分内容读者可能不太理解，可先不予深究，本书的第 3 章和第 4 章将会对此进行深入讲解。

本节内容参考了 PyTorch 官方教程[①]并做了相应的增删，使得内容更贴合新版本的 PyTorch 接口，同时也更适合新手快速入门。另外，本书需要读者先掌握基础的 numpy 使用，numpy 的基础知识可以参考 CS231n 上关于 numpy 的教程[②]。

2.2.1 Tensor

Tensor 是 PyTorch 中重要的数据结构，可认为是一个高维数组。它可以是一个数（标量）、一维数组（向量）、二维数组（矩阵）或更高维的数组。Tensor 和 numpy 的 ndarrays 类似，但 Tensor 可以使用 GPU 加速。Tensor 的使用和 numpy 及 MATLAB 的接口十分相似，下面通过几个示例了解 Tensor 的基本使用方法。

```
In: from __future__ import print_function
    import torch as t
```

```
In: # 构建5*3矩阵，只是分配了空间，未初始化
    x = t.Tensor(5, 3)
    x
```

```
Out: 1.00000e-18 *
       0.0000  0.0000  1.6687
       0.0000  0.0000  0.0000
       0.0000  0.0000  0.0000
       0.0000  1.6788  0.0000
       0.0000  0.0000  0.0000
     [torch.FloatTensor of size 5x3]
```

```
In: # 使用[0,1]均匀分布随机初始化二维数组
    x = t.rand(5, 3)
    x
```

[①] http://pytorch.org/tutorials/beginner/deep_learning_60min_blitz.html

[②] http://cs231n.github.io/python-numpy-tutorial/

```
Out:  0.3175  0.5136  0.6991
      0.7968  0.8288  0.6663
      0.2217  0.3570  0.1961
      0.0779  0.4476  0.2365
      0.8144  0.1300  0.8572
     [torch.FloatTensor of size 5x3]
```

```
In: print(x.size()) # 查看x的形状
    x.size()[0], x.size(1) # 查看列的个数，两种写法等价
```

```
Print: torch.Size([5, 3])
```

```
Out: (5L, 3)
```

torch.Size 是 tuple 对象的子类，因此它支持 tuple 的所有操作，如 x.size()[0]。

```
In: y = t.rand(5, 3)
    # 加法的第一种写法
    x + y
```

```
Out:  0.5989  1.4450  0.8684
      0.8170  1.1645  1.4846
      0.3208  1.1928  1.1725
      0.7517  0.7156  0.8332
      0.8643  0.2580  1.3766
     [torch.FloatTensor of size 5x3]
```

```
In: # 加法的第二种写法
    t.add(x, y)
```

```
Out:  0.5989  1.4450  0.8684
      0.8170  1.1645  1.4846
      0.3208  1.1928  1.1725
      0.7517  0.7156  0.8332
      0.8643  0.2580  1.3766
     [torch.FloatTensor of size 5x3]
```

```
In: # 加法的第三种写法：指定加法结果的输出目标为result
    result = t.Tensor(5, 3) # 预先分配空间
    t.add(x, y, out=result) # 输入到result
    result
```

```
Out:  0.5989  1.4450  0.8684
      0.8170  1.1645  1.4846
      0.3208  1.1928  1.1725
      0.7517  0.7156  0.8332
      0.8643  0.2580  1.3766
     [torch.FloatTensor of size 5x3]
```

```
In: print('最初y')
    print(y)

    print('第一种加法，y的结果')
    y.add(x) # 普通加法，不改变y的内容
    print(y)

    print('第二种加法，y的结果')
    y.add_(x) # inplace加法，y变了
    print(y)
```

```
Print: 最初y

       0.2815  0.9315  0.1694
       0.0202  0.3357  0.8183
       0.0991  0.8358  0.9764
       0.6738  0.2680  0.5967
       0.0499  0.1280  0.5193
      [torch.FloatTensor of size 5x3]

      第一种加法，y的结果

       0.2815  0.9315  0.1694
       0.0202  0.3357  0.8183
       0.0991  0.8358  0.9764
       0.6738  0.2680  0.5967
       0.0499  0.1280  0.5193
      [torch.FloatTensor of size 5x3]

      第二种加法，y的结果
```

```
 0.5989  1.4450  0.8684
 0.8170  1.1645  1.4846
 0.3208  1.1928  1.1725
 0.7517  0.7156  0.8332
 0.8643  0.2580  1.3766
[torch.FloatTensor of size 5x3]
```

注意，函数名后面带下画线_的函数会修改 Tensor 本身。例如，`x.add_(y)`和`x.t_()`会改变 `x`，但`x.add(y)`和`x.t()`会返回一个新的 Tensor，而`x`不变。

```
In: # Tensor的选取操作与numpy类似
    x[:, 1]
```

```
Out:  0.5136
      0.8288
      0.3570
      0.4476
      0.1300
     [torch.FloatTensor of size 5]
```

Tensor 还支持很多操作，包括数学运算、线性代数、选择、切片等，其接口设计与 numpy 极为相似。更详细的使用方法会在第 3 章系统讲解。

Tensor 和 numpy 的数组间的互操作非常容易且快速。Tensor 不支持的操作，可以先转为 numpy 数组处理，之后再转回 Tensor。

```
In: a = t.ones(5) # 新建一个全是1的Tensor
    a
```

```
Out:  1
      1
      1
      1
      1
     [torch.FloatTensor of size 5]
```

```
In: b = a.numpy() # Tensor -> Numpy
    b
```

```
Out: array([ 1.,  1.,  1.,  1.,  1.], dtype=float32)
```

```
In: import numpy as np
    a = np.ones(5)
    b = t.from_numpy(a) # Numpy->Tensor
    print(a)
    print(b)
```

```
Print: [ 1.  1.  1.  1.  1.]

       1
       1
       1
       1
       1
      [torch.DoubleTensor of size 5]
```

Tensor 和 numpy 对象共享内存，所以它们之间的转换很快，而且几乎不会消耗资源。这也意味着，如果其中一个变了，另外一个也会随之改变。

```
In: b.add_(1) # 以_结尾的函数会修改自身
    print(a)
    print(b) # Tensor和Numpy共享内存
```

```
Print: [ 2.  2.  2.  2.  2.]

       2
       2
       2
       2
       2
      [torch.DoubleTensor of size 5]
```

Tensor 可通过.cuda 方法转为 GPU 的 Tensor，从而享受 GPU 带来的加速运算。

```
In: # 在不支持CUDA的机器下，下一步不会运行
    if t.cuda.is_available():
        x = x.cuda()
        y = y.cuda()
        x + y
```

在此处可能会发现 GPU 运算的速度并未提升太多，这是因为 x 和 y 太小且运算也较简单，而且将数据从内存转移到显存还需要花费额外的开销。GPU 的优势需在大规模数据和复杂运算下才能体现出来。

2.2.2 Autograd：自动微分

深度学习的算法本质上是通过反向传播求导数，PyTorch 的`Autograd`模块实现了此功能。在 Tensor 上的所有操作，Autograd 都能为它们自动提供微分，避免手动计算导数的复杂过程。

`autograd.Variable`是 Autograd 中的核心类，它简单封装了 Tensor，并支持几乎所有 Tensor 的操作。Tensor 在被封装为 Variable 之后，可以调用它的`.backward`实现反向传播，自动计算所有梯度。Variable 的数据结构如图 2-6 所示。

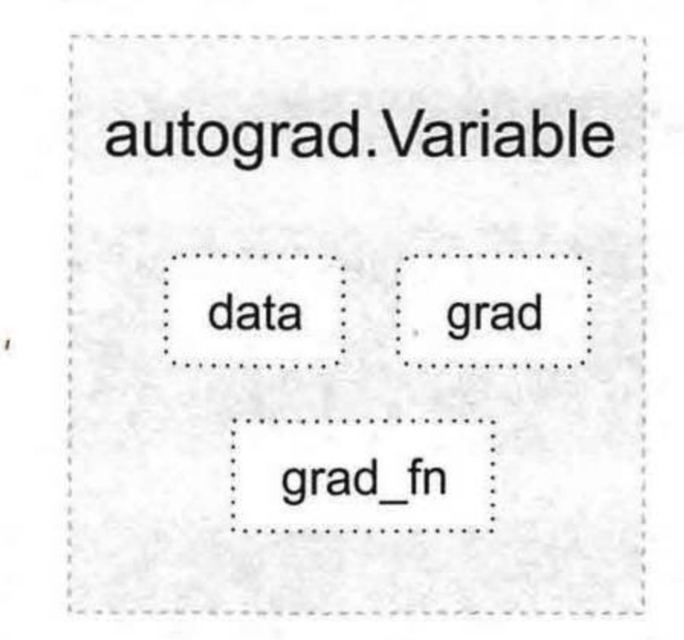


图 2-6 Variable 的数据结构

Variable 主要包含三个属性。

- `data`：保存 Variable 所包含的 Tensor。
- `grad`：保存`data`对应的梯度，`grad`也是个 Variable，而不是 Tensor，它和`data`的形状一样。
- `grad_fn`：指向一个`Function`对象，这个`Function`用来反向传播计算输入的梯度，具体细节会在第 3 章讲解。

```
In: from torch.autograd import Variable
```

```
In: # 使用Tensor新建一个Variable
    x = Variable(t.ones(2, 2), requires_grad = True)
    x
```

```
Out: Variable containing:
      1  1
      1  1
     [torch.FloatTensor of size 2x2]
```

```
In: y = x.sum()
    y
```

```
Out: Variable containing:
      4
     [torch.FloatTensor of size 1]
```

```
In: y.grad_fn
```

```
Out: <torch.autograd.function.SumBackward at 0x7fdd7be58240>
```

```
In: y.backward() # 反向传播,计算梯度
```

```
In: # y = x.sum() = (x[0][0] + x[0][1] + x[1][0] + x[1][1])
    # 每个值的梯度都为1
    x.grad
```

```
Out: Variable containing:
      1  1
      1  1
     [torch.FloatTensor of size 2x2]
```

注意：grad在反向传播过程中是累加的（accumulated），这意味着每次运行反向传播，梯度都会累加之前的梯度，所以反向传播之前需把梯度清零。

```
In: y.backward()
    x.grad
```

```
Out: Variable containing:
      2  2
      2  2
     [torch.FloatTensor of size 2x2]
```

```
In: y.backward()
    x.grad
```

```
Out: Variable containing:
      3  3
      3  3
     [torch.FloatTensor of size 2x2]
```

```
In: # 以下画线结束的函数是inplace操作
    x.grad.data.zero_()
```

```
Out:  0  0
      0  0
     [torch.FloatTensor of size 2x2]
```

```
In: y.backward()
    x.grad
```

```
Out: Variable containing:
      1  1
      1  1
     [torch.FloatTensor of size 2x2]
```

Variable 和 Tensor 具有近乎一致的接口，在实际使用中可以无缝切换。

```
In: x = Variable(t.ones(4,5))
    y = t.cos(x)
    x_tensor_cos = t.cos(x.data)
    print(y)
    x_tensor_cos
```

```
Print: Variable containing:
        0.5403  0.5403  0.5403  0.5403  0.5403
        0.5403  0.5403  0.5403  0.5403  0.5403
        0.5403  0.5403  0.5403  0.5403  0.5403
        0.5403  0.5403  0.5403  0.5403  0.5403
       [torch.FloatTensor of size 4x5]
```

```
Out:  0.5403  0.5403  0.5403  0.5403  0.5403
      0.5403  0.5403  0.5403  0.5403  0.5403
      0.5403  0.5403  0.5403  0.5403  0.5403
      0.5403  0.5403  0.5403  0.5403  0.5403
     [torch.FloatTensor of size 4x5]
```

2.2.3 神经网络

Autograd 实现了反向传播功能，但是直接用来写深度学习的代码在很多情况下还是稍显复杂，torch.nn 是专门为神经网络设计的模块化接口。nn 构建于 Autograd 之上，可用来定义和运行神经网络。nn.Module 是 nn 中最重要的类，可以把它看作一个网络的封装，包含网络各层定义及 forward 方法，调用 forward(input) 方法，可返回前向传播的结果。我们以最早的卷积神经网络 LeNet 为例，来看看如何用`nn.Module`实现。LeNet 的网络结构如图 2-7 所示。

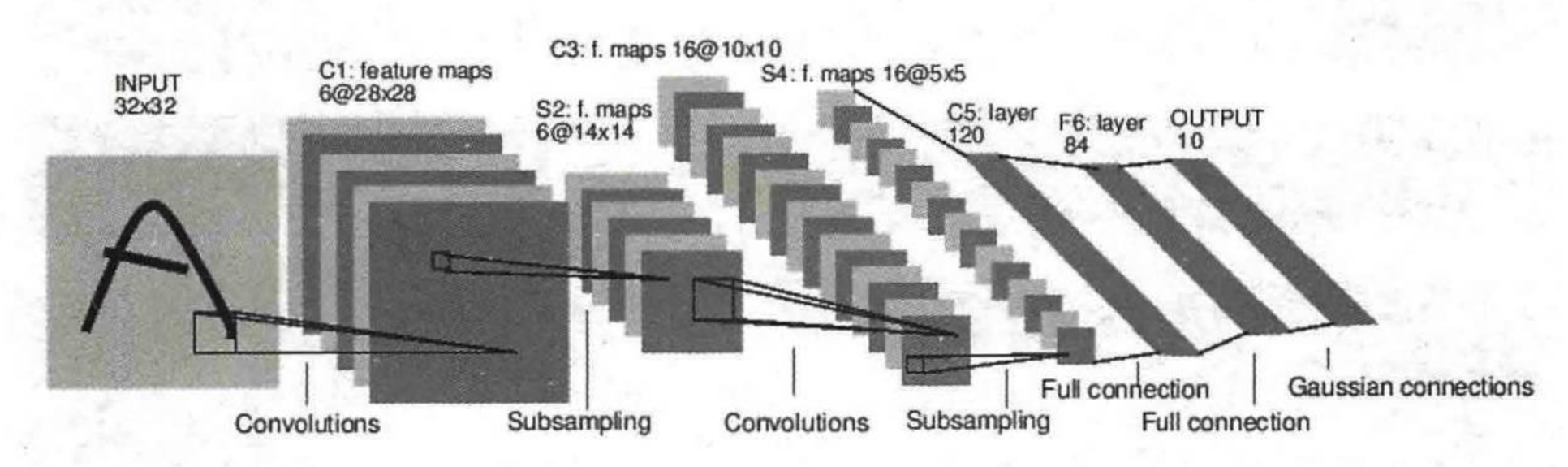


图 2-7　LeNet 网络结构

这是一个基础的前向传播（feed-forward）网络：接收输入，经过层层传递运算，得到输出。

定义网络

定义网络时，需要继承`nn.Module`，并实现它的 forward 方法，把网络中具有可学习参数的层放在构造函数`__init__`中。如果某一层（如 ReLU）不具有可学习的参数，则既可以放在构造函数中，也可以不放。

```
In: import torch.nn as nn
    import torch.nn.functional as F

    class Net(nn.Module):
        def __init__(self):
            # nn.Module子类的函数必须在构造函数中执行父类的构造函数
            # 下式等价于nn.Module.__init__(self)
            super(Net, self).__init__()
```

```
        # 卷积层'1'表示输入图片为单通道，'6'表示输出通道数
        # '5'表示卷积核为5*5
        self.conv1 = nn.Conv2d(1, 6, 5)
        # 卷积层
        self.conv2 = nn.Conv2d(6, 16, 5)
        # 仿射层/全连接层，y = Wx + b
        self.fc1   = nn.Linear(16*5*5, 120)
        self.fc2   = nn.Linear(120, 84)
        self.fc3   = nn.Linear(84, 10)

    def forward(self, x):
        # 卷积 -> 激活 -> 池化
        x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
        x = F.max_pool2d(F.relu(self.conv2(x)), 2)
        # reshape，'-1' 表示自适应
        x = x.view(x.size()[0], -1)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

net = Net()
print(net)
```

```
Print: Net (
        (conv1): Conv2d(1, 6, kernel_size=(5, 5), stride=(1, 1))
        (conv2): Conv2d(6, 16, kernel_size=(5, 5), stride=(1, 1))
        (fc1): Linear (400 -> 120)
        (fc2): Linear (120 -> 84)
        (fc3): Linear (84 -> 10)
    )
```

只要在 nn.Module 的子类中定义了 forward 函数，backward 函数就会被自动实现（利用`Autograd`）。在`forward` 函数中可使用任何 Variable 支持的函数，还可以使用 if、for 循环、print、log 等 Python 语法，写法和标准的 Python 写法一致。

网络的可学习参数通过`net.parameters()`返回，`net.named_parameters`可同时返回可学习的参数及名称。

```
In: params = list(net.parameters())
    print(len(params))
```

```
Print: 10
```

```
In: for name,parameters in net.named_parameters():
        print(name,':',parameters.size())
```

```
Print: conv1.weight : torch.Size([6, 1, 5, 5])
       conv1.bias : torch.Size([6])
       conv2.weight : torch.Size([16, 6, 5, 5])
       conv2.bias : torch.Size([16])
       fc1.weight : torch.Size([120, 400])
       fc1.bias : torch.Size([120])
       fc2.weight : torch.Size([84, 120])
       fc2.bias : torch.Size([84])
       fc3.weight : torch.Size([10, 84])
       fc3.bias : torch.Size([10])
```

forward 函数的输入和输出都是 Variable，只有 Variable 才具有自动求导功能，Tensor 是没有的，所以在输入时，需要把 Tensor 封装成 Variable。

```
In: input = Variable(t.randn(1, 1, 32, 32))
    out = net(input)
    out.size()
```

```
Out: torch.Size([1, 10])
```

```
In: net.zero_grad() #所有参数的梯度清零
    out.backward(Variable(t.ones(1,10))) #反向传播
```

需要注意的是，torch.nn 只支持 mini-batches，不支持一次只输入一个样本，即一次必须是一个 batch。如果只想输入一个样本，则用 `input.unsqueeze(0)`将 batch_size 设为 1 。例如，`nn.Conv2d` 输入必须是 4 维的，形如 nSamples × nChannels × Height × Width。可将 nSample 设为 1，即 1 × nChannels × Height × Width。

损失函数

nn 实现了神经网络中大多数的损失函数，例如 nn.MSELoss 用来计算均方误差，nn.CrossEntropyLoss 用来计算交叉熵损失。

```
In: output = net(input)
    target = Variable(t.arange(0,10))
    criterion = nn.MSELoss()
    loss = criterion(output, target)
    loss
```

```
Out: Variable containing:
      28.6064
     [torch.FloatTensor of size 1]
```

如果对 loss 进行反向传播溯源（使用grad_fn属性），可看到它的计算图如下：

```
input -> conv2d -> relu -> maxpool2d -> conv2d -> relu -> maxpool2d
      -> view -> linear -> relu -> linear -> relu -> linear
      -> MSELoss
      -> loss
```

当调用loss.backward()时，该图会动态生成并自动微分，也会自动计算图中参数（Parameter）的导数。

```
In: # 运行.backward，观察调用之前和调用之后的grad
    net.zero_grad() # 把net中所有可学习参数的梯度清零
    print('反向传播之前conv1.bias的梯度')
    print(net.conv1.bias.grad)
    loss.backward()
    print('反向传播之后conv1.bias的梯度')
    print(net.conv1.bias.grad)
```

```
Print: 反向传播之前conv1.bias的梯度
       Variable containing:
        0
        0
        0
        0
        0
        0
       [torch.FloatTensor of size 6]

       反向传播之后conv1.bias的梯度
       Variable containing:
```

```
1.00000e-02 *
 -0.4356
 -2.2457
 -2.8671
 -7.6732
 -2.6371
  5.4218
[torch.FloatTensor of size 6]
```

优化器

在反向传播计算完所有参数的梯度后，还需要使用优化方法更新网络的权重和参数。例如，随机梯度下降法（SGD）的更新策略如下：

```
weight = weight - learning_rate * gradient
```

手动实现如下：

```
learning_rate = 0.01
for f in net.parameters():
    f.data.sub_(f.grad.data * learning_rate)# inplace 减法
```

torch.optim中实现了深度学习中绝大多数的优化方法，例如 RMSProp、Adam、SGD 等，更便于使用，因此通常并不需要手动写上述代码。

```
In: import torch.optim as optim
    #新建一个优化器，指定要调整的参数和学习率
    optimizer = optim.SGD(net.parameters(), lr = 0.01)

    #在训练过程中
    #先梯度清零(与net.zero_grad()效果一样)
    optimizer.zero_grad()

    #计算损失
    output = net(input)
    loss = criterion(output, target)

    #反向传播
    loss.backward()
```

```
#更新参数
optimizer.step()
```

数据加载与预处理

在深度学习中数据加载及预处理是非常复杂烦琐的，但 PyTorch 提供了一些可极大简化和加快数据处理流程的工具。同时，对于常用的数据集，PyTorch 也提供了封装好的接口供用户快速调用，这些数据集主要保存在 torchvision 中。

`torchvision`实现了常用的图像数据加载功能，例如 Imagenet、CIFAR10、MNIST 等，以及常用的数据转换操作，这极大地方便了数据加载。

2.2.4 小试牛刀：CIFAR-10 分类

下面我们来尝试实现对 CIFAR-10 数据集的分类，步骤如下：

（1）使用 torchvision 加载并预处理 CIFAR-10 数据集。

（2）定义网络。

（3）定义损失函数和优化器。

（4）训练网络并更新网络参数。

（5）测试网络。

CIFAR-10 数据加载及预处理

CIFAR-10[①]是一个常用的彩色图片数据集，它有 10 个类别 airplane、automobile、bird、cat、deer、dog、frog、horse、ship 和 truck。每张图片都是 $3 \times 32 \times 32$，也即 3 通道彩色图片，分辨率为 32×32。

```
In: import torchvision as tv
    import torchvision.transforms as transforms
    from torchvision.transforms import ToPILImage
    show = ToPILImage() # 可以把Tensor转成Image，方便可视化
```

① http://www.cs.toronto.edu/~kriz/cifar.html

```
In: # 第一次运行程序torchvision会自动下载CIFAR-10数据集，
    # 大约100MB，需花费一定的时间，
    # 如果已经下载有CIFAR-10，可通过root参数指定

    # 定义对数据的预处理
    transform = transforms.Compose([
            transforms.ToTensor(), # 转为Tensor
            transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)), # 归一
            化
                                ])

    # 训练集
    trainset = tv.datasets.CIFAR10(
                        root='/home/cy/data/',
                        train=True,
                        download=True,
                        transform=transform)

    trainloader = t.utils.data.DataLoader(
                        trainset,
                        batch_size=4,
                        shuffle=True,
                        num_workers=2)

    # 测试集
    testset = tv.datasets.CIFAR10(
                        '/home/cy/data/',
                        train=False,
                        download=True,
                        transform=transform)

    testloader = t.utils.data.DataLoader(
                        testset,
                        batch_size=4,
                        shuffle=False,
                        num_workers=2)
```

```
    classes = ('plane', 'car', 'bird', 'cat',
               'deer', 'dog', 'frog', 'horse', 'ship', 'truck')
```

```
Print: Files already downloaded and verified
       Files already downloaded and verified
```

Dataset 对象是一个数据集，可以按下标访问，返回形如（data，label）的数据。

```
In: (data, label) = trainset[100]
    print(classes[label])

    # (data + 1) / 2是为了还原被归一化的数据，程序输出的图片如图2-8所示
    show((data + 1) / 2).resize((100, 100))
```

```
Print: ship
```

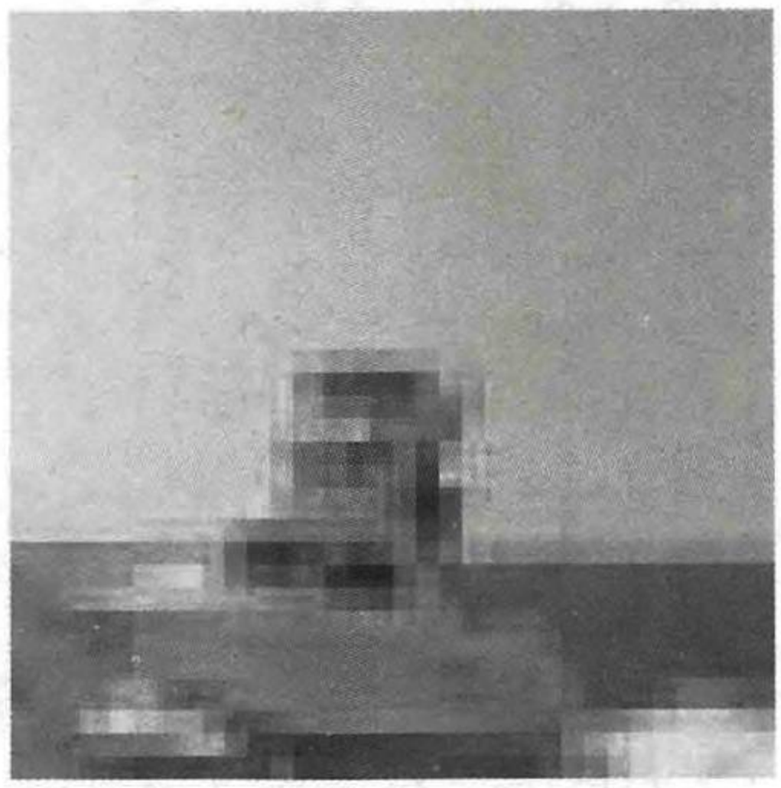


图 2-8　程序输出：CIFAR10 的示例图片

Dataloader 是一个可迭代的对象，它将 dataset 返回的每一条数据样本拼接成一个 batch，并提供多线程加速优化和数据打乱等操作。当程序对 dataset 的所有数据遍历完一遍之后，对 Dataloader 也完成了一次迭代。

```
In: dataiter = iter(trainloader)
    images, labels = dataiter.next() # 返回4张图片及标签，如图2-9所示
    print(' '.join('%11s'%classes[labels[j]] for j in range(4)))
    show(tv.utils.make_grid((images+1)/2)).resize((400,100))
```

```
Print:       truck        car        dog       horse
```

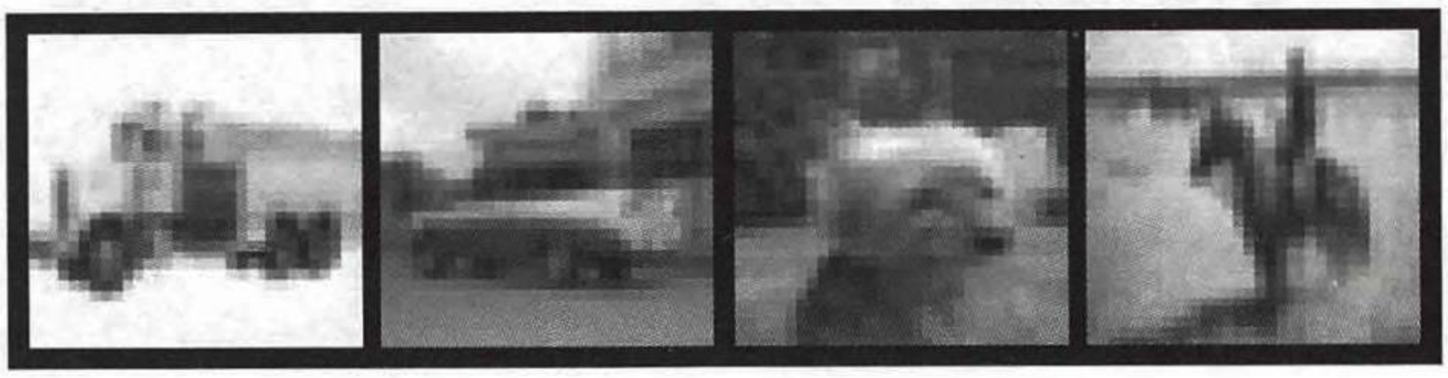


图 2-9　程序输出：测试集的图片

定义网络

复制上面的 LeNet 网络，修改 self.conv1 中第一个参数为 3 通道，因为 CIFAR-10 是 3 通道彩图。

```
In: import torch.nn as nn
    import torch.nn.functional as F

    class Net(nn.Module):
        def __init__(self):
            super(Net, self).__init__()
            self.conv1 = nn.Conv2d(3, 6, 5)
            self.conv2 = nn.Conv2d(6, 16, 5)
            self.fc1   = nn.Linear(16*5*5, 120)
            self.fc2   = nn.Linear(120, 84)
            self.fc3   = nn.Linear(84, 10)

        def forward(self, x):
            x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
            x = F.max_pool2d(F.relu(self.conv2(x)), 2)
            x = x.view(x.size()[0], -1)
            x = F.relu(self.fc1(x))
            x = F.relu(self.fc2(x))
            x = self.fc3(x)
            return x


    net = Net()
    print(net)
```

```
Print: Net (
```

```
    (conv1): Conv2d(3, 6, kernel_size=(5, 5), stride=(1, 1))
    (conv2): Conv2d(6, 16, kernel_size=(5, 5), stride=(1, 1))
    (fc1): Linear (400 -> 120)
    (fc2): Linear (120 -> 84)
    (fc3): Linear (84 -> 10)
  )
```

定义损失函数和优化器（loss 和 optimizer）

```
In: from torch import optim
    criterion = nn.CrossEntropyLoss() # 交叉熵损失函数
    optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)
```

训练网络

所有网络的训练流程都是类似的，不断地执行如下流程。

- 输入数据。
- 前向传播 + 反向传播。
- 更新参数。

```
In:for epoch in range(2):

      running_loss = 0.0
      for i, data in enumerate(trainloader, 0):

          # 输入数据
          inputs, labels = data
          inputs, labels = Variable(inputs), Variable(labels)

          # 梯度清零
          optimizer.zero_grad()

          # forward + backward
          outputs = net(inputs)
          loss = criterion(outputs, labels)
          loss.backward()
```

```
        # 更新参数
        optimizer.step()

        # 打印log信息
        running_loss += loss.data[0]
        if i % 2000 == 1999: # 每2000个batch打印一次训练状态
            print('[%d, %5d] loss: %.3f' \
                  % (epoch+1, i+1, running_loss / 2000))
            running_loss = 0.0
    print('Finished Training')
```

```
Print: [1,  2000] loss: 2.236
       [1,  4000] loss: 1.862
       [1,  6000] loss: 1.662
       [1,  8000] loss: 1.599
       [1, 10000] loss: 1.530
       [1, 12000] loss: 1.484
       [2,  2000] loss: 1.436
       [2,  4000] loss: 1.413
       [2,  6000] loss: 1.383
       [2,  8000] loss: 1.374
       [2, 10000] loss: 1.326
       [2, 12000] loss: 1.304
       Finished Training
```

此处仅训练了 2 个 epoch（遍历完一遍数据集称为一个 epoch），我们来看看网络有没有效果。将测试图片输入网络，计算它的 label，然后与实际的 label 进行比较。

```
In: dataiter = iter(testloader)
    images, labels = dataiter.next() # 一个batch返回4张图片，如图2-10所示
    print('实际的label: ', ' '.join(\
                '%08s'%classes[labels[j]] for j in range(4)))
    show(tv.utils.make_grid(images / 2 - 0.5)).resize((400,100))
```

```
Print: 实际的label:       cat     ship     ship    plane
```

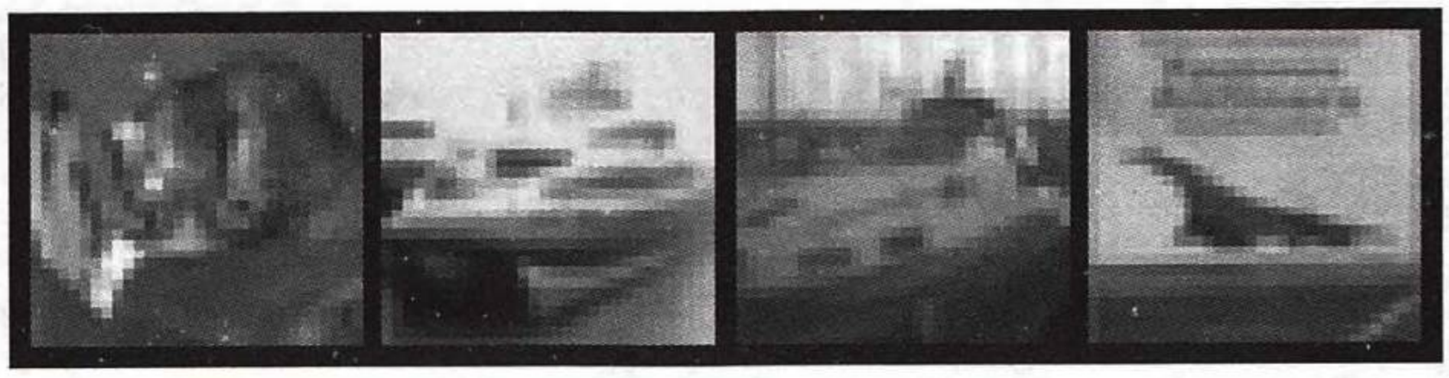


图 2-10 程序输出的图片

接着计算网络预测的 label：

```
In: # 计算图片在每个类别上的分数
    outputs = net(Variable(images))
    # 得分最高的那个类
    _, predicted = t.max(outputs.data, 1)

    print('预测结果: ', ' '.join('%5s'\
                % classes[predicted[j]] for j in range(4)))
```

```
Print: 预测结果:    cat  ship  ship  ship
```

我们已经可以看出效果，准确率为 75%，但这只是一部分图片，我们再来看看在整个测试集上的效果。

```
In: correct = 0 # 预测正确的图片数
    total = 0 # 总共的图片数
    for data in testloader:
        images, labels = data
        outputs = net(Variable(images))
        _, predicted = t.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum()

    print('10000张测试集中的准确率为: %d %%' % (100 * correct / total))
```

```
Print: 10000张测试集中的准确率为: 52%
```

训练的准确率远比随机猜测（准确率为 10%）好，证明网络确实学到了东西。

在 GPU 上训练

就像之前把 Tensor 从 CPU 转到 GPU 一样，模型也可以类似地从 CPU 转到 GPU。

```
In: if t.cuda.is_available():
        net.cuda()
        images = images.cuda()
        labels = labels.cuda()
        output = net(Variable(images))
        loss= criterion(output,Variable(labels))
```

如果发现在 GPU 上训练的速度并没比在 CPU 上提速很多，实际是因为网络比较小，GPU 没有完全发挥自己的真正实力。

对 PyTorch 的基础介绍至此结束。总结一下，本节主要介绍以下内容。

（1）Tensor：类似 numpy 数组的数据结构，与 numpy 接口类似，可方便地互相转换。

（2）autograd/Variable：Variable 封装了 Tensor，并提供自动求导功能。

（3）nn：专门为神经网络设计的接口，提供了很多有用的功能（神经网络层、损失函数、优化器等）。

（4）神经网络训练：以 CIFAR-10 分类为例演示了神经网络的训练流程，包括数据加载、网络搭建、训练及测试。

通过本章的学习，读者能够配置 PyTorch+Jupyter+IPython 的学习环境。另外，通过 2.2 节关于 PyTorch 的概要介绍，相信读者可以体会出 PyTorch 接口简单、使用灵活等特点。如果有哪些内容读者没有理解，不用着急，这些内容会在后续章节深入和详细地讲解。

3 Tensor 和 autograd

几乎所有的深度学习框架背后的设计核心都是张量和计算图，PyTorch 也不例外，本章我们将学习 PyTorch 中的张量系统（Tensor）和自动微分系统（autograd）。

3.1 Tensor

Tensor，又名张量，读者可能对这个名词似曾相识，因为它不仅在 PyTorch 中出现过，也是 Theano、TensorFlow、Torch 和 MXNet 中重要的数据结构。关于张量的本质不乏深度剖析的文章，但从工程角度讲，可简单地认为它就是一个数组，且支持高效的科学计算。它可以是一个数（标量）、一维数组（向量）、二维数组（矩阵）或更高维的数组（高阶数据）。Tensor 和 numpy 的 ndarrays 类似，但 PyTorch 的 tensor 支持 GPU 加速。

本节将系统讲解 tensor 的使用，力求面面俱到，但不会涉及每个函数。对于更多函数及其用法，读者可通过在 IPython/Notebook 中使用`<function>?`查看帮助文档，或查阅 PyTorch 的官方文档[1]。

```
In: # Let's begin
    from __future__ import print_function
    import torch  as t
```

[1] http://docs.pytorch.org

3.1.1 基础操作

学习过 numpy 的读者会对本节内容非常熟悉，因为 tensor 的接口设计得与 numpy 类似，以方便用户使用。若不熟悉 numpy 也没关系，本节内容并不要求读者先掌握 numpy。

从接口的角度讲，对 tensor 的操作可分为两类：

（1）`torch.function`，如`torch.save`等。

（2）`tensor.function`，如`tensor.view`等。

为方便使用，对 tensor 的大部分操作同时支持这两类接口，在本书中不做具体区分，如`torch.sum(a, b)`与`a.sum(b)`功能等价。

从存储的角度讲，对 tensor 的操作又可分为两类：

（1）不会修改自身的数据，如 `a.add(b)`，加法的结果会返回一个新的 tensor。

（2）会修改自身的数据，如 `a.add_(b)`，加法的结果仍存储在 a 中，a 被修改了。

函数名以_结尾的都是 inplace 方式，即会修改调用者自己的数据，在实际应用中需加以区分。

创建 Tensor

在 PyTorch 中新建 tensor 的方法有很多，具体如表 3-1 所示。

表 3-1　常见的新建 tensor 的方法

函数	功能
Tensor(*sizes)	基础构造函数
ones(*sizes)	全 1Tensor
zeros(*sizes)	全 0Tensor
eye(*sizes)	对角线为 1，其他为 0
arange(s,e,step)	从 s 到 e，步长为 step
linspace(s,e,steps)	从 s 到 e，均匀切分成 steps 份
rand/randn(*sizes)	均匀/标准分布
normal(mean,std)/uniform(from,to)	正态分布/均匀分布
randperm(m)	随机排列

其中使用`Tensor`函数新建 tensor 是最复杂多变的方式，它既可以接收一个 list，并根据 list 的数据新建 tensor，也能根据指定的形状新建 tensor，还能传入其他的 tensor，

下面举几个例子。

```
In: # 指定tensor的形状
    a = t.Tensor(2, 3)
    a # 数值取决于内存空间的状态
```

```
Out: 1.00000e-04 *
       2.8929  0.0000  0.0000
       0.0000  0.0000  0.0000
     [torch.FloatTensor of size 2x3]
```

```
In: # 用list的数据创建tensor
    b = t.Tensor([[1,2,3],[4,5,6]])
    b
```

```
Out:  1  2  3
      4  5  6
     [torch.FloatTensor of size 2x3]
```

```
In: b.tolist() # 把tensor转为list
```

```
Out: [[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]]
```

`tensor.size()`返回`torch.Size`对象，它是 tuple 的子类，但其使用方式与 tuple 略有区别。

```
In: b_size = b.size()
    b_size
```

```
Out: torch.Size([2, 3])
```

```
In: b.numel() # b中元素总个数，2*3，等价于b.nelement()
```

```
Out: 6
```

```
In: # 创建一个和b形状一样的tensor
    c = t.Tensor(b_size)
    # 创建一个元素为2和3的tensor
    d = t.Tensor((2, 3))
    c, d
```

```
Out: (
      1.00000e-04 *
        2.8929  0.0000  0.0000
        0.0000  0.0000  0.0000
      [torch.FloatTensor of size 2x3],
       2
       3
      [torch.FloatTensor of size 2])
```

除了tensor.size()，还可以利用tensor.shape直接查看 tensor 的形状，tensor.shape等价于tensor.size()。

```
In: c.shape
```

```
Out: torch.Size([2, 3])
```

```
In: c.shape??
```

需要注意的是，t.Tensor(*sizes)创建 tensor 时，系统不会马上分配空间，只会计算剩余的内存是否足够使用，使用到 tensor 时才会分配，而其他操作都是在创建完 tensor 后马上进行空间分配。其他常用的创建 tensor 方法举例如下。

```
In: t.ones(2, 3)
```

```
Out:  1  1  1
      1  1  1
     [torch.FloatTensor of size 2x3]
```

```
In: t.zeros(2, 3)
```

```
Out:  0  0  0
      0  0  0
     [torch.FloatTensor of size 2x3]
```

```
In: t.arange(1, 6, 2)
```

```
Out:  1
      3
      5
     [torch.FloatTensor of size 3]
```

```
In: t.linspace(1, 10, 3)
```

```
Out:   1.0000
       5.5000
      10.0000
     [torch.FloatTensor of size 3]
```

```
In: t.randn(2, 3)
```

```
Out:  1.1974 -0.6145 -1.1441
      0.1930  1.5890  2.1035
     [torch.FloatTensor of size 2x3]
```

```
In: t.randperm(5) # 长度为5的随机排列
```

```
Out:  1
      2
      4
      0
      3
     [torch.LongTensor of size 5]
```

```
In: t.eye(2, 3) # 对角线为1，不要求行列数一致
```

```
Out:  1  0  0
      0  1  0
     [torch.FloatTensor of size 2x3]
```

常用 Tensor 操作

通过`tensor.view`方法可以调整 tensor 的形状，但必须保证调整前后元素总数一致。`view`不会修改自身的数据，返回的新 tensor 与源 tensor 共享内存，即更改其中一个，另外一个也会跟着改变。在实际应用中可能经常需要添加或减少某一维度，这时`squeeze`和`unsqueeze`两个函数就派上了用场。

```
In: a = t.arange(0, 6)
    a.view(2, 3)
```

```
Out:  0  1  2
      3  4  5
     [torch.FloatTensor of size 2x3]
```

```
In: b = a.view(-1, 3) # 当某一维为-1的时候，会自动计算它的大小
    b
```

```
Out:  0  1  2
      3  4  5
     [torch.FloatTensor of size 2x3]
```

```
In: b.unsqueeze(1) # 注意形状，在第1维（下标从0开始）上增加“1”
```

```
Out: (0 ,.,.) =
       0  1  2

     (1 ,.,.) =
       3  4  5
     [torch.FloatTensor of size 2x1x3]
```

```
In: b.unsqueeze(-2) # -2表示倒数第二个维度
```

```
Out: (0 ,.,.) =
       0  1  2

     (1 ,.,.) =
       3  4  5
     [torch.FloatTensor of size 2x1x3]
```

```
In: c = b.view(1, 1, 1, 2, 3)
    c.squeeze(0) # 压缩第0维的“1”
```

```
Out: (0 ,0 ,.,.) =
       0  1  2
       3  4  5
     [torch.FloatTensor of size 1x1x2x3]
```

```
In: c.squeeze() # 把所有维度为“1”的压缩
```

```
Out:  0  1  2
      3  4  5
     [torch.FloatTensor of size 2x3]
```

```
In: a[1] = 100
    b # a和b共享内存，修改了a，b也变了
```

```
Out:     0  100    2
         3    4    5
     [torch.FloatTensor of size 2x3]
```

resize是另一种可用来调整size的方法，但与view不同，它可以修改 tensor 的尺寸。如果新尺寸超过了原尺寸，会自动分配新的内存空间，而如果新尺寸小于原尺寸，则之前的数据依旧会被保存，我们来看一个例子。

```
In: b.resize_(1, 3)
    b
```

```
Out:     0  100    2
     [torch.FloatTensor of size 1x3]
```

```
In: b.resize_(3, 3) # 旧的数据依旧保存着，多出的数据会分配新空间
    b
```

```
Out:     0.0000  100.0000    2.0000
         3.0000    4.0000    5.0000
         0.0000    0.0000    0.0000
     [torch.FloatTensor of size 3x3]
```

索引操作

Tensor 支持与 numpy.ndarray 类似的索引操作，语法上也类似，下面通过一些例子，讲解常用的索引操作。如无特殊说明，索引出来的结果与原 tensor 共享内存，即修改一个，另一个会跟着修改。

```
In: a = t.randn(3, 4)
    a
```

```
Out: -1.1651 -1.6413  1.4957 -0.9682
     -0.9403 -0.8954  1.5083 -0.4903
     -1.6095  0.6315 -0.4384 -1.2031
     [torch.FloatTensor of size 3x4]
```

```
In: a[0] # 第0行(下标从0开始)
```

```
Out: -1.1651
     -1.6413
      1.4957
     -0.9682
     [torch.FloatTensor of size 4]
```

```
In: a[:, 0] # 第0列
```

```
Out: -1.1651
     -0.9403
     -1.6095
     [torch.FloatTensor of size 3]
```

```
In: a[0][2] # 第0行第2个元素，等价于a[0, 2]
```

```
Out: 1.4957412481307983
```

```
In: a[0, -1] # 第0行最后一个元素
```

```
Out: -0.9682474732398987
```

```
In: a[:2] # 前两行
```

```
Out: -1.1651 -1.6413  1.4957 -0.9682
     -0.9403 -0.8954  1.5083 -0.4903
     [torch.FloatTensor of size 2x4]
```

```
In: a[:2, 0:2] # 前两行，第0,1列
```

```
Out: -1.1651 -1.6413
     -0.9403 -0.8954
     [torch.FloatTensor of size 2x2]
```

```
In: print(a[0:1, :2]) # 第0行，前两列
    print(a[0, :2]) # 注意两者的区别：形状不同
```

```
Print: -1.1651 -1.6413
       [torch.FloatTensor of size 1x2]


       -1.1651
       -1.6413
       [torch.FloatTensor of size 2]
```

```
In: a > 1 # 返回一个ByteTensor
```

```
Out:  0  0  1  0
      0  0  1  0
      0  0  0  0
     [torch.ByteTensor of size 3x4]
```

```
In: a[a>1] # 等价于a.masked_select(a>1)
    # 选择结果与原tensor不共享内存空间
```

```
Out:  1.4957
      1.5083
     [torch.FloatTensor of size 2]
```

```
In: a[t.LongTensor([0,1])] # 第0行和第1行
```

```
Out: -1.1651 -1.6413  1.4957 -0.9682
     -0.9403 -0.8954  1.5083 -0.4903
     [torch.FloatTensor of size 2x4]
```

其他常用的选择函数如表 3-2 所示。

表 3-2　常用的选择函数

函数	功能
index_select(input, dim, index)	在指定维度 dim 上选取，例如选取某些行、某些列
masked_select(input, mask)	例子如上，a[a>0]，使用 ByteTensor 进行选取
non_zero(input)	非 0 元素的下标
gather(input, dim, index)	根据 index，在 dim 维度上选取数据，输出的 size 与 index 一样

gather是一个比较复杂的操作，对一个二维 tensor，输出的每个元素如下：

```
out[i][j] = input[index[i][j]][j]  # dim=0
out[i][j] = input[i][index[i][j]]  # dim=1
```

三维 tensor 的gather操作同理，下面举几个例子。

```
In: a = t.arange(0, 16).view(4, 4)
    a
```

```
Out:   0   1   2   3
       4   5   6   7
       8   9  10  11
      12  13  14  15
     [torch.FloatTensor of size 4x4]
```

```
In: # 选取对角线的元素
    index = t.LongTensor([[0,1,2,3]])
    a.gather(0, index)
```

```
Out:   0   5  10  15
     [torch.FloatTensor of size 1x4]
```

```
In: # 选取反对角线上的元素
    index = t.LongTensor([[3,2,1,0]]).t()
    a.gather(1, index)
```

```
Out:   3
       6
       9
      12
     [torch.FloatTensor of size 4x1]
```

```
In: # 选取反对角线上的元素，注意与上面的不同
    index = t.LongTensor([[3,2,1,0]])
    a.gather(0, index)
```

```
Out:  12   9   6   3
     [torch.FloatTensor of size 1x4]
```

```
In: # 选取两个对角线上的元素
    index = t.LongTensor([[0,1,2,3],[3,2,1,0]]).t()
    b = a.gather(1, index)
    b
```

```
Out:   0   3
       5   6
      10   9
      15  12
    [torch.FloatTensor of size 4x2]
```

与gather相对应的逆操作是scatter_，gather把数据从 input 中按 index 取出，而scatter_是把取出的数据再放回去。注意scatter_函数是 inplace 操作。

```
out = input.gather(dim, index)
-->近似逆操作
out = Tensor()
out.scatter_(dim, index)
```

```
In: # 把两个对角线元素放回到指定位置
    c = t.zeros(4,4)
    c.scatter_(1, index, b)
```

```
Out:   0   0   0   3
       0   5   6   0
       0   9  10   0
      12   0   0  15
    [torch.FloatTensor of size 4x4]
```

高级索引

PyTorch 0.2 版中完善了索引操作，目前已经支持绝大多数 numpy 风格的高级索引[①]。高级索引可以看成是普通索引操作的扩展，但是高级索引操作的结果一般不和原始的 Tensor 共享内存。

```
In: x = t.arange(0,27).view(3,3,3)
    x
```

[①] https://docs.scipy.org/doc/numpy/reference/arrays.indexing.html#advanced-indexing

```
Out: (0 ,.,.) =
        0   1   2
        3   4   5
        6   7   8

     (1 ,.,.) =
        9  10  11
       12  13  14
       15  16  17

     (2 ,.,.) =
       18  19  20
       21  22  23
       24  25  26
     [torch.FloatTensor of size 3x3x3]
```

```
In: x[[1, 2], [1, 2], [2, 0]] # x[1,1,2]和x[2,2,0]
```

```
Out:  14
      24
     [torch.FloatTensor of size 2]
```

```
In: x[[2, 1, 0], [0], [1]] # x[2,,0,1],x[1,0,1],x[0,0,1]
```

```
Out:  19
      10
       1
     [torch.FloatTensor of size 3]
```

```
In: x[[0, 2], ...] # x[0] 和 x[2]
```

```
Out: (0 ,.,.) =
        0   1   2
        3   4   5
        6   7   8

     (1 ,.,.) =
       18  19  20
       21  22  23
```

```
    24  25  26
[torch.FloatTensor of size 2x3x3]
```

Tensor 类型

Tensor 有不同的数据类型，如表 3-3 所示，每种类型分别对应有 CPU 和 GPU 版本（HalfTensor 除外）。默认的 tensor 是 FloatTensor，可通过`t.set_default_tensor_type`修改默认 tensor 类型（如果默认类型为 GPU tensor，则所有操作都将在 GPU 上进行）。Tensor 的类型对分析内存占用很有帮助。例如，一个 size 为 (1000, 1000, 1000) 的 FloatTensor，它有`1000x1000x1000=10^9`个元素，每个元素占 32bit/8 = 4Byte 内存，所以共占大约 4GB 内存/显存。HalfTensor 是专门为 GPU 版本设计的，同样的元素个数，显存占用只有 FloatTensor 的一半，所以可以极大地缓解 GPU 显存不足的问题，但由于 HalfTensor 所能表示的数值大小和精度有限[①]，所以可能出现溢出等问题。

表 3-3　tensor 数据类型

数据类型	CPU tensor	GPU tensor
32bit 浮点	torch.FloatTensor	torch.cuda.FloatTensor
64bit 浮点	torch.DoubleTensor	torch.cuda.DoubleTensor
16bit 半精度浮点	N/A	torch.cuda.HalfTensor
8bit 无符号整形 (0~255)	torch.ByteTensor	torch.cuda.ByteTensor
8bit 有符号整形 (-128~127)	torch.CharTensor	torch.cuda.CharTensor
16bit 有符号整形	torch.ShortTensor	torch.cuda.ShortTensor
32bit 有符号整形	torch.IntTensor	torch.cuda.IntTensor
64bit 有符号整形	torch.LongTensor	torch.cuda.LongTensor

各数据类型之间可以互相转换，`type(new_type)`是通用的做法，同时还有`float`、`long`、`half`等快捷方法。CPU tensor 与 GPU tensor 之间的互相转换通过`tensor.cuda`和`tensor.cpu`的方法实现。Tensor 还有一个`new`方法，用法与`t.Tensor`一样，会调用该 tensor 对应类型的构造函数，生成与当前 tensor 类型一致的 tensor。

```
In: # 设置默认tensor，注意参数是字符串
    t.set_default_tensor_type('torch.IntTensor')
```

```
In: a = t.Tensor(2,3)
    a # 现在a是IntTensor
```

[①] https://stackoverflow.com/questions/872544/what-range-of-numbers-can-be-represented-in-a-16-32-and-64-bit-ieee-754-syste

```
Out:  9.6624e+08  3.2757e+04  6.5066e+07
      0.0000e+00  3.2000e+01  0.0000e+00
     [torch.IntTensor of size 2x3]
```

```
In: # 把a转成FloatTensor，等价于b=a.type(t.FloatTensor)
    b = a.float()
    b
```

```
Out:  9.6624e+08  3.2757e+04  6.5066e+07
      0.0000e+00  3.2000e+01  0.0000e+00
     [torch.FloatTensor of size 2x3]
```

```
In: c = a.type_as(b)
    c
```

```
Out:  9.6624e+08  3.2757e+04  6.5066e+07
      0.0000e+00  3.2000e+01  0.0000e+00
     [torch.FloatTensor of size 2x3]
```

```
In: d = a.new(2,3) # 等价于torch.IntTensor(3,4)
    d
```

```
Out:  9.6624e+08  3.2757e+04  6.5072e+07
      0.0000e+00  0.0000e+00  7.0000e+00
     [torch.IntTensor of size 2x3]
```

```
In: # 查看函数new的源码
    a.new??
```

```
In: # 恢复之前的默认设置
    t.set_default_tensor_type('torch.FloatTensor')
```

逐元素操作

这部分操作会对 tensor 的每一个元素（point-wise，又名 element-wise）进行操作，此类操作的输入与输出形状一致。常用的操作如表 3-4 所示。

表 3-4 常见的逐元素操作

函数	功能
abs/sqrt/div/exp/fmod/log/pow..	绝对值/平方根/除法/指数/求余/求幂
cos/sin/asin/atan2/cosh	三角函数
ceil/round/floor/trunc	上取整/四舍五入/下取整/只保留整数部分
clamp(input, min, max)	超过 min 和 max 部分截断
sigmod/tanh…	激活函数

对于很多操作，例如 div、mul、pow、fmod 等，PyTorch 都实现了运算符重载，所以可以直接使用运算符。例如，`a ** 2` 等价于`torch.pow(a,2)`，`a * 2`等价于`torch.mul(a,2)`。

其中`clamp(x, min, max)`的输出满足以下公式：

$$y_i = \begin{cases} \min, & \text{if } x_i < \min \\ x_i, & \text{if } \min \leqslant x_i \leqslant \max \\ \max, & \text{if } x_i > \max \end{cases}$$

`clamp`常用在某些需要比较大小的地方，如取一个 tensor 的每个元素与另一个数的较大值。

```
In: a = t.arange(0, 6).view(2, 3)
    t.cos(a)
```

```
Out:  1.0000  0.5403 -0.4161
     -0.9900 -0.6536  0.2837
     [torch.FloatTensor of size 2x3]
```

```
In: a % 3 # 等价于t.fmod(a, 3)
```

```
Out:  0  1  2
      0  1  2
     [torch.FloatTensor of size 2x3]
```

```
In: a ** 2 # 等价于t.pow(a, 2)
```

```
Out:   0   1   4
       9  16  25
     [torch.FloatTensor of size 2x3]
```

```
In: # a中每一个元素都与3相比，取较大的一个
    print(a)
    t.clamp(a, min=3)
```

```
Print:  0  1  2
        3  4  5
       [torch.FloatTensor of size 2x3]
```

```
Out:  3  3  3
      3  4  5
     [torch.FloatTensor of size 2x3]
```

归并操作

此类操作会使输出形状小于输入形状，并可以沿着某一维度进行指定操作。如加法`sum`，既可以计算整个 tensor 的和，也可以计算 tensor 中每一行或每一列的和。常用的归并操作如表 3-5 所示。

表 3-5　常用的归并操作

函数	功能
mean/sum/median/mode	均值/和/中位数/众数
norm/dist	范数/距离
std/var	标准差/方差
cumsum/cumprod	累加/累乘

以上大多数函数都有一个参数`dim`，用来指定这些操作是在哪个维度上执行的。关于 dim（对应于 Numpy 中的 axis）的解释众说纷纭，这里提供一个简单的记忆方式。

假设输入的形状是 (m, n, k)：

- 如果指定 dim=0，输出的形状就是 (1, n, k) 或者 (n, k)；
- 如果指定 dim=1，输出的形状就是 (m, 1, k) 或者 (m, k)；
- 如果指定 dim=2，输出的形状就是 (m, n, 1) 或者 (m, n)。

size 中是否有“1”，取决于参数`keepdim`，`keepdim=True`会保留维度`1`。从 PyTorch 0.2.0 版本起，`keepdim`默认为 False。注意，以上只是经验总结，并非所有函数都符合这种形状变化方式，如`cumsum`。

```
In: b = t.ones(2, 3)
    b.sum(dim = 0, keepdim=True)
```

```
Out:  2  2  2
     [torch.FloatTensor of size 1x3]
```

```
In: # keepdim=False, 不保留维度"1", 注意形状
    b.sum(dim=0, keepdim=False)
```

```
Out:  2
      2
      2
     [torch.FloatTensor of size 3]
```

```
In: b.sum(dim=1)
```

```
Out:  3
      3
     [torch.FloatTensor of size 2]
```

```
In: a = t.arange(0, 6).view(2, 3)
    print(a)
    a.cumsum(dim=1) # 沿着行累加
```

```
Print:  0  1  2
        3  4  5
       [torch.FloatTensor of size 2x3]
```

```
Out:   0   1   3
       3   7  12
      [torch.FloatTensor of size 2x3]
```

比较

比较函数中有一些是逐元素比较，操作类似于逐元素操作，还有一些则类似于归并操作。常用的比较函数如表 3-6 所示。

表 3-6　常用的比较函数

函数	功能
gt/lt/ge/le/eq/ne	大于/小于/大于等于/小于等于/等于/不等
topk	最大的 k 个数
sort	排序
max/min	比较两个 tensor 的最大值和最小值

表中第一行的比较操作已经实现了运算符重载，因此可以使用`a>=b`、`a>b`、`a!=b`和`a==b`，其返回结果是一个`ByteTensor`，可用来选取元素。max/min 这两个操作比较特殊，以 max 为例，它有以下三种使用情况。

- t.max(tensor)：返回 tensor 中最大的一个数。
- t.max(tensor,dim)：指定维上最大的数，返回 tensor 和下标。
- t.max(tensor1, tensor2)：比较两个 tensor 相比较大的元素。

比较一个 tensor 和一个数，可以使用 clamp 函数。下面举例说明。

```
In: a = t.linspace(0, 15, 6).view(2, 3)
    a
```

```
Out:   0   3   6
       9  12  15
    [torch.FloatTensor of size 2x3]
```

```
In: b = t.linspace(15, 0, 6).view(2, 3)
    b
```

```
Out:  15  12   9
       6   3   0
    [torch.FloatTensor of size 2x3]
```

```
In: a>b
```

```
Out:  0  0  0
      1  1  1
    [torch.ByteTensor of size 2x3]
```

```
In: a[a>b] # a中大于b的元素
```

```
Out:   9
      12
      15
    [torch.FloatTensor of size 3]
```

```
In: t.max(a)
```

```
Out: 15.0
```

```
In: t.max(b, dim=1)
    # 第一个返回值的15和6分别表示第0行和第1行最大的元素
    # 第二个返回值的0和0表示上述最大的数是该行第0个元素
```

```
Out: (
      15
       6
    [torch.FloatTensor of size 2],
      0
      0
    [torch.LongTensor of size 2])
```

```
In: t.max(a,b)
```

```
Out:  15  12   9
       9  12  15
    [torch.FloatTensor of size 2x3]
```

```
In: # 比较a和10较大的元素
    t.clamp(a, min=10)
```

```
Out:  10  10  10
      10  12  15
    [torch.FloatTensor of size 2x3]
```

线性代数

PyTorch 的线性函数主要封装了 Blas 和 Lapack，其用法和接口都与之类似。常用的线性代数函数如表 3-7 所示。

表 3-7　常用的线性代数函数

函数	功能
trace	对角线元素之和（矩阵的迹）
diag	对角线元素
triu/tril	矩阵的上三角/下三角，可指定偏移量
mm/bmm	矩阵乘法，batch 的矩阵乘法
addmm/addbmm/addmv	矩阵运算
t	转置
dot/cross	内积/外积
inverse	求逆矩阵
svd	奇异值分解

具体使用说明请参见官方文档[①]，需要注意的是，矩阵的转置会导致存储空间不连续，需调用它的.contiguous方法将其转为连续。

```
In: b = a.t()
    b.is_contiguous()
```

```
Out: False
```

```
In: b.contiguous()
```

```
Out:   0   9
       3  12
       6  15
    [torch.FloatTensor of size 3x2]
```

3.1.2　Tensor 和 Numpy

Tensor 和 Numpy 数组之间具有很高的相似性，彼此之间的互操作也非常简单高效。需要注意的是，Numpy 和 Tensor 共享内存。由于 Numpy 历史悠久，支持丰富的操作，所以当遇到 Tensor 不支持的操作时，可先转成 Numpy 数组，处理后再转回 tensor，其转换开销很小。

[①] http://pytorch.org/docs/torch.html#blas-and-lapack-operations

- 当输入数组的某个维度的长度为 1 时，计算时沿此维度复制扩充成一样的形状。

PyTorch 当前已经支持了自动广播法则，但笔者还是建议读者通过以下两个函数的组合手动实现广播法则，这样更直观，更不易出错。

- `unsqueeze`或者`view`：为数据某一维的形状补 1，实现法则 1。
- `expand`或者`expand_as`，重复数组，实现法则 3；该操作不会复制数组，所以不会占用额外的空间。

注意：repeat 实现与 expand 相类似的功能，但是 repeat 会把相同数据复制多份，因此会占用额外的空间。

```
In: a = t.ones(3, 2)
    b = t.zeros(2, 3,1)
```

```
In: # 自动广播法则
    # 第一步：a是二维，b是三维，所以先在较小的a前面补1，
    # 即：a.unsqueeze(0)，a的形状变成（1，3，2），b的形状是（2，3，1），
    # 第二步：a和b在第一维和第三维的形状不一样，其中一个为1，
    # 可以利用广播法则扩展，两个形状都变成了（2，3，2）
    a+b
```

```
Out: (0 ,.,.) =
       1  1
       1  1
       1  1

     (1 ,.,.) =
       1  1
       1  1
       1  1
     [torch.FloatTensor of size 2x3x2]
```

```
In: # 手动广播法则
    # 或者a.view(1,3,2).expand(2,3,2)+b.expand(2,3,2)
    a.unsqueeze(0).expand(2, 3, 2) + b.expand(2,3,2)
```

```
Out: (0 ,.,.) =
       1  1
       1  1
       1  1
```

```
In: import numpy as np
    a = np.ones([2, 3], dtype=np.float32)
    a
```

```
Out: array([[ 1.,  1.,  1.],
            [ 1.,  1.,  1.]])
```

```
In: b = t.from_numpy(a)
    b
```

```
Out:  1  1  1
      1  1  1
     [torch.DoubleTensor of size 2x3]
```

```
In: b = t.Tensor(a) # 也可以直接将numpy对象传入Tensor，这种情况下若Numpy类型不是Float32会新建
    b
```

```
Out:  1  1  1
      1  1  1
     [torch.FloatTensor of size 2x3]
```

```
In: a[0, 1]=100
    b
```

```
Out:  1  1  1
      1  1  1
     [torch.FloatTensor of size 2x3]
```

```
In: c = b.numpy() # a, b, c三个对象共享内存
    c
```

```
Out: array([[ 1.,  1.,  1.],
            [ 1.,  1.,  1.]], dtype=float32)
```

广播法则（Broadcast）是科学运算中经常使用的一个技巧，它在快速执行向量化的同时不会占用额外的内存/显存。Numpy 的广播法则定义如下：

- 让所有输入数组都向其中 shape 最长的数组看齐，shape 中不足的部分通过在前面加 1 补齐。
- 两个数组要么在某一个维度的长度一致，要么其中一个为 1，否则不能计算。

```
(1 ,.,.) =
  1  1
  1  1
  1  1
[torch.FloatTensor of size 2x3x2]
```

```
In: # expand不会占用额外空间，只会在需要时才扩充，可极大地节省内存
    e = a.unsqueeze(0).expand(10000000000000, 3,2)
```

3.1.3 内部结构

tensor 的数据结构如图 3-1 所示。tensor 分为头信息区（Tensor）和存储区（Storage），信息区主要保存着 tensor 的形状（size）、步长（stride）、数据类型（type）等信息，而真正的数据则保存成连续数组。由于数据动辄成千上万，因此信息区元素占用内存较少，主要内存占用取决于 tensor 中元素的数目，即存储区的大小。

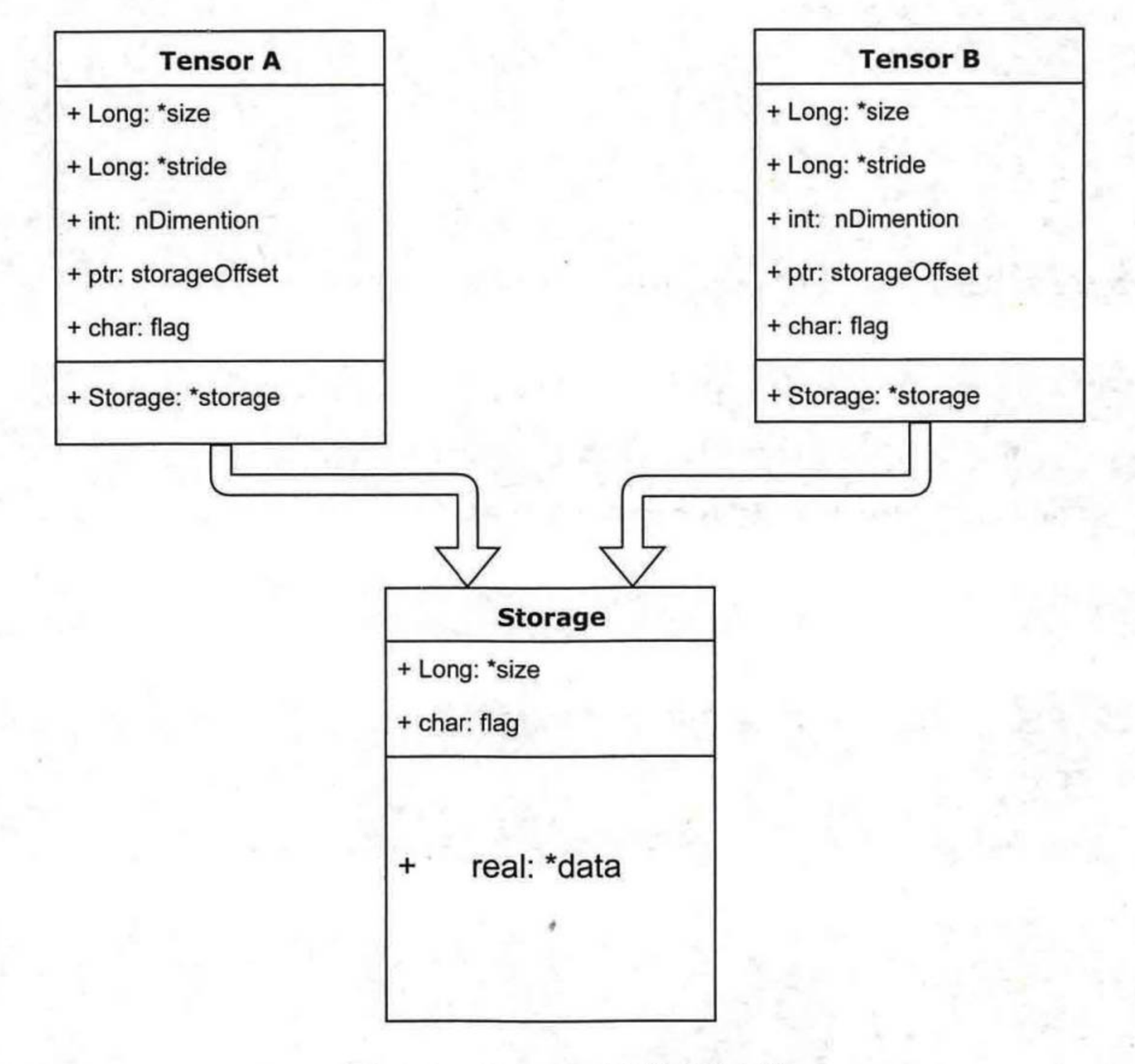


图 3-1 Tensor 的数据结构

一般来说，一个 tensor 有着与之相对应的 storage，storage 是在 data 之上封装的接口，便于使用。不同 tensor 的头信息一般不同，但却可能使用相同的 storage。下面我们来看两个例子。

```
In: a = t.arange(0, 6)
    a.storage()
```

```
Out:  0.0
      1.0
      2.0
      3.0
      4.0
      5.0
     [torch.FloatStorage of size 6]
```

```
In: b = a.view(2, 3)
    b.storage()
```

```
Out:  0.0
      1.0
      2.0
      3.0
      4.0
      5.0
     [torch.FloatStorage of size 6]
```

```
In: # 一个对象的id值可以看作它在内存中的地址
    # a和b storage的内存地址一样，即是同一个storage
    id(b.storage()) == id(a.storage())
```

```
Out: True
```

```
In: # a改变，b也随之改变，因为它们共享storage
    a[1] = 100
    b
```

```
Out:     0  100    2
         3    4    5
     [torch.FloatTensor of size 2x3]
```

```
In: c = a[2:]
    c.storage()
```

```
Out:  0.0
      100.0
      2.0
      3.0
      4.0
      5.0
    [torch.FloatStorage of size 6]
```

```
In: c.data_ptr(), a.data_ptr() # data_ptr返回tensor首元素的内存地址
    # 可以看出相差8，这是因为2x4=8相差两个元素，每个元素占4个字节(float)
```

```
Out: (63322504, 63322496)
```

```
In: c[0] = -100 # c[0]的内存地址对应a[2]的内存地址
    a
```

```
Out:    0
      100
     -100
        3
        4
        5
    [torch.FloatTensor of size 6]
```

```
In: d = t.Tensor(c.storage())
    d[0] = 6666
    b
```

```
Out:  6666   100  -100
         3     4     5
    [torch.FloatTensor of size 2x3]
```

```
In: # 下面 4 个tensor共享storage
    id(a.storage()) == id(b.storage()) == id(c.storage()) == id(d.storage
    ())
```

```
Out: True
```

```
In: a.storage_offset(), c.storage_offset(), d.storage_offset()
```

```
Out: (0, 2, 0)
```

```
In: e = b[::2, ::2] # 隔2行/列取一个元素
    id(e.storage()) == id(a.storage())
```

```
Out: True
```

```
In: b.stride(), e.stride()
```

```
Out: ((3, 1), (6, 2))
```

```
In: e.is_contiguous()
```

```
Out: False
```

可见绝大多数操作并不修改 tensor 的数据，只是修改了 tensor 的头信息。这种做法更节省内存，同时提升了处理速度。此外，有些操作会导致 tensor 不连续，这时需调用`tensor.contiguous`方法将它们变成连续的数据，该方法复制数据到新的内存，不再与原来的数据共享 storage。另外读者可以思考一下，之前说过的高级索引一般不共享 storage，而普通索引共享 storage，这是为什么呢？（提示：普通索引可以通过修改 tensor 的 offset、stride 和 size 实现，不修改 storage 的数据，高级索引则不行）。

3.1.4 其他有关 Tensor 的话题

这部分的内容不好专门划分为一节，但笔者认为值得读者注意，故将其放在本节。

持久化

Tensor 的保存和加载十分简单，使用 t.save 和 t.load 即可完成相应的功能。在 save/load 时可指定使用的`pickle`模块，在 load 时还可将 GPU tensor 映射到 CPU 或其他 GPU 上。

```
In: if t.cuda.is_available():
        a = a.cuda(1) # 把a转为GPU1上的tensor,
        t.save(a,'a.pth')
```

```
# 加载为b，存储于GPU1上(因为保存时tensor就在GPU1上)
b = t.load('a.pth')
# 加载为c，存储于CPU
c = t.load('a.pth', map_location=lambda storage, loc: storage)
# 加载为d，存储于GPU0上
d = t.load('a.pth', map_location={'cuda:1':'cuda:0'})
```

向量化

向量化计算是一种特殊的并行计算方式，一般程序在同一时间只执行一个操作的方式，它可在同一时间执行多个操作，通常是对不同的数据执行同样的一个或一批指令，或者说把指令应用于一个数组/向量上。向量化可极大地提高科学运算的效率，Python 本身是一门高级语言，使用很方便，但许多操作很低效，尤其是`for`循环。在科学计算程序中应当极力避免使用 Python 原生的`for`循环，尽量使用向量化的数值计算。

```
In: def for_loop_add(x, y):
        result = []
        for i,j in zip(x, y):
            result.append(i + j)
        return t.Tensor(result)
```

```
In: x = t.zeros(100)
    y = t.ones(100)
    %timeit -n 10 for_loop_add(x, y)
    %timeit -n 10 x + y
```

```
Print: 10 loops, best of 3: 89.3 µs per loop
       10 loops, best of 3: 8.14 µs per loop
```

可见二者有超过10倍的速度差距，因此在实际使用中应尽量调用内建函数（builtin-function），这些函数底层由 C/C++ 实现，能通过执行底层优化实现高效计算。因此在平时写代码时，就应养成向量化的思维习惯。

此外还有以下几点需要注意：

- 大多数`t.function`都有一个参数`out`，这时产生的结果将保存在 out 指定的 tensor 之中。
- `t.set_num_threads`可以设置 PyTorch 进行 CPU 多线程并行计算时所占用的线程数，用来限制 PyTorch 所占用的 CPU 数目。

- t.set_printoptions 可以用来设置打印 tensor 时的数值精度和格式。下面举例说明。

```
In: a = t.arange(0, 20000000)
    print(a[-1], a[-2]) # 32bit的IntTensor精度有限导致溢出
    b = t.LongTensor()
    t.arange(0, 20000000, out=b) # 64bit的LongTensor不会溢出
    b[-1],b[-2]
```

```
Print: 16777216.0 16777216.0
```

```
Out: (199999, 199998)
```

```
In: a = t.randn(2,3)
    a
```

```
Out:  0.4584 -0.7445  0.5889
      0.8662 -0.3860  1.2659
     [torch.FloatTensor of size 2x3]
```

```
In: t.set_printoptions(precision=10)
    a
```

```
Out: 0.4583995938 -0.7444863915 0.5889241695
     0.8662075400 -0.3859837353 1.2658821344
     [torch.FloatTensor of size 2x3]
```

3.1.5 小试牛刀：线性回归

线性回归是机器学习的入门知识，应用十分广泛。线性回归利用数理统计中的回归分析来确定两种或两种以上变量间相互依赖的定量关系，其表达形式为 $\boldsymbol{y} = \boldsymbol{wx} + \boldsymbol{b} + \boldsymbol{e}$，误差 $\boldsymbol{e}$ 服从均值为 0 的正态分布。线性回归的损失函数是：

$$\text{loss} = \sum_{i}^{N} \frac{1}{2}(\boldsymbol{y_i} - (\boldsymbol{wx_i} + \boldsymbol{b}))^2$$

利用随机梯度下降法更新参数 $\boldsymbol{w}$ 和 $\boldsymbol{b}$ 来最小化损失函数，最终学得 $\boldsymbol{w}$ 和 $\boldsymbol{b}$ 的数值。

```
In: import torch as t
    %matplotlib inline
    from matplotlib import pyplot as plt
    from IPython import display
```

```
In: # 设置随机数种子，保证在不同计算机上运行时下面的输出一致
    t.manual_seed(1000)


    def get_fake_data(batch_size=8):
        ''' 产生随机数据：y=x*2+3，加上了一些噪声'''
        x = t.rand(batch_size, 1) * 20
        y = x * 2 + (1 + t.randn(batch_size, 1))*3
        return x, y
```

```
In: # 来看看产生的x-y分布，输出如图3-2所示
    x, y = get_fake_data()
    plt.scatter(x.squeeze().numpy(), y.squeeze().numpy())
```

```
Out: <matplotlib.collections.PathCollection at 0x7ff502ccd668>
```

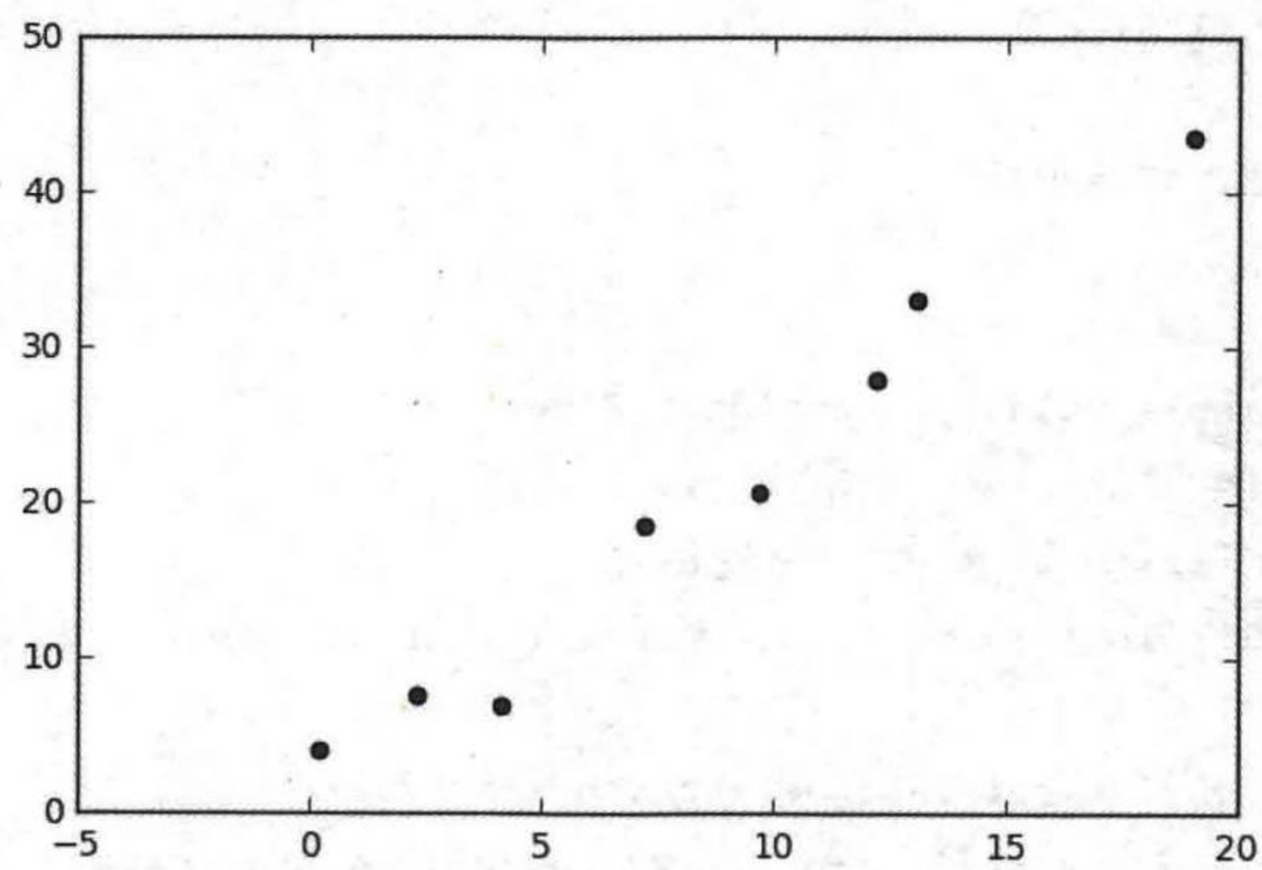


图 3-2　程序输出：x-y 的分布

```
In: # 随机初始化参数
    w = t.rand(1, 1)
    b = t.zeros(1, 1)
```

```
lr =0.001 # 学习率

for ii in range(20000):
    x, y = get_fake_data()

    # forward: 计算loss
    y_pred = x.mm(w) + b.expand_as(y)
    loss = 0.5 * (y_pred - y) ** 2 # 均方误差
    loss = loss.sum()

    # backward: 手动计算梯度
    dloss = 1
    dy_pred = dloss * (y_pred - y)

    dw = x.t().mm(dy_pred)
    db = dy_pred.sum()

    # 更新参数
    w.sub_(lr * dw)
    b.sub_(lr * db)

    if ii%1000 ==0:

        # 画图
        display.clear_output(wait=True)
        x = t.arange(0, 20).view(-1, 1)
        y = x.mm(w) + b.expand_as(x)
        plt.plot(x.numpy(), y.numpy()) # predicted

        x2, y2 = get_fake_data(batch_size=20)
        plt.scatter(x2.numpy(), y2.numpy()) # true data

        plt.xlim(0, 20)
        plt.ylim(0, 41)
        plt.show() # 程序输出如图3-3所示
        plt.pause(0.5)
```

```
print(w.squeeze()[0], b.squeeze()[0])
```

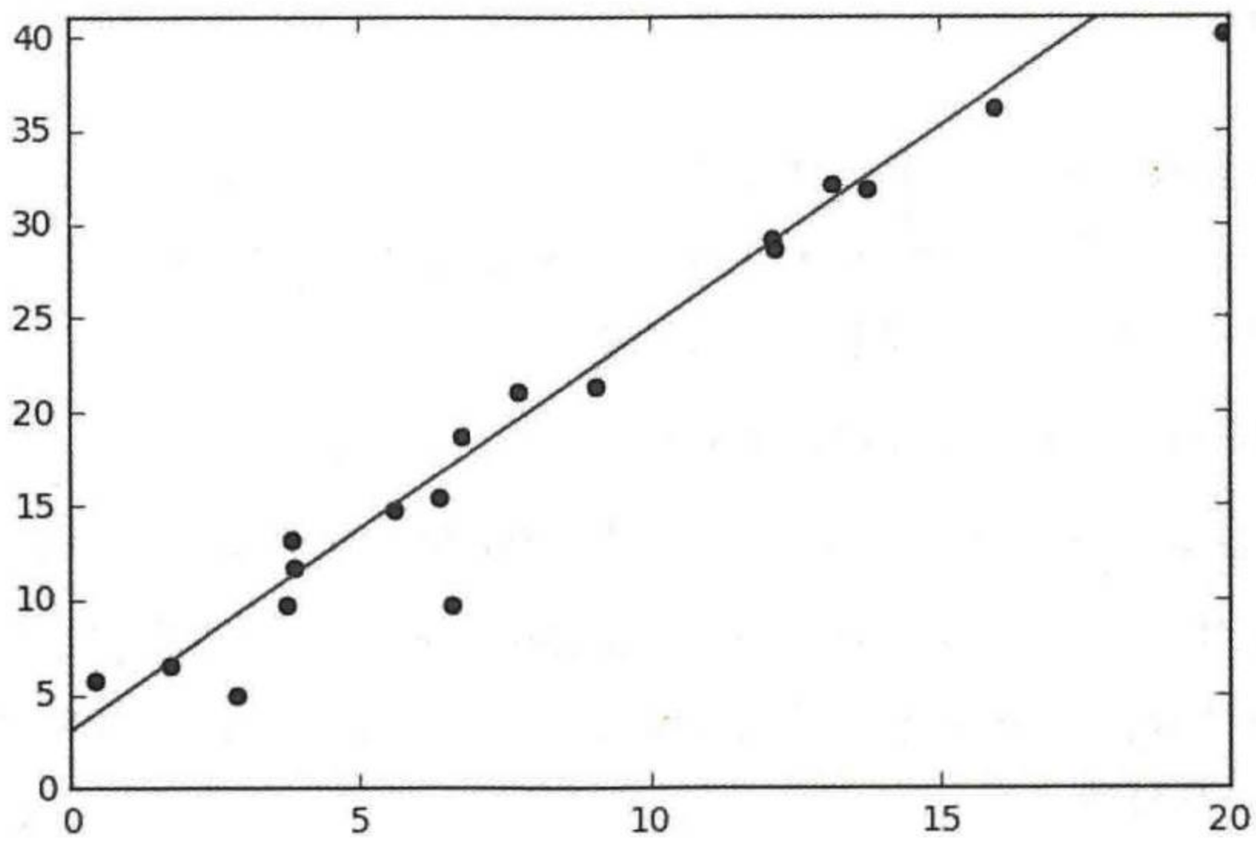


图 3-3　程序输出

```
Print: 2.0185186862945557 3.03572154045105
```

可见程序已经基本学出 w=2、b=3，并且图中直线和数据已经实现较好的拟合。

上面提到了 Tensor 的许多操作，不要求全部掌握，日后遇到时可以再查阅这部分内容或者查找对应文档，在此有个基本印象即可。

3.2　autograd

用 Tensor 训练网络很方便，但从 3.1 节最后的线性回归例子来看，反向传播过程需要手动实现。这对线性回归这种较简单的模型来说还比较容易，但实际使用中经常出现非常复杂的网络结构，此时如果手动实现反向传播，不仅费时费力，而且容易出错，难以检查。torch.autograd 就是为方便用户使用，专门开发的一套自动求导引擎，它能够根据输入和前向传播过程自动构建计算图，并执行反向传播。

计算图（Computation Graph）是现代深度学习框架（如 PyTorch 和 TensorFlow 等）的核心，它为自动求导算法——反向传播（Back Propogation）提供了理论支持，了解计算图在实际写程序过程中会有极大的帮助。本节会涉及一些基础的计算图知识，但

并不要求读者事先对此有深入了解。关于计算图的基础知识推荐阅读 Christopher Olah 的文章[①]。

3.2.1 Variable

PyTorch 在 autograd 模块中实现了计算图的相关功能，autograd 中的核心数据结构是 Variable。Variable 封装了 tensor，并记录对 tensor 的操作记录用来构建计算图。Variable 的数据结构如图 3-4 所示，主要包含三个属性。

- `data`：保存 variable 所包含的 tensor。
- `grad`：保存`data`对应的梯度，`grad`也是 variable，而非 tensor，它与`data`形状一致。
- `grad_fn`：指向一个`Function`，记录 variable 的操作历史，即它是什么操作的输出，用来构建计算图。如果某一个变量是由用户创建的，则它为叶子节点，对应的 grad_fn 等于 None。

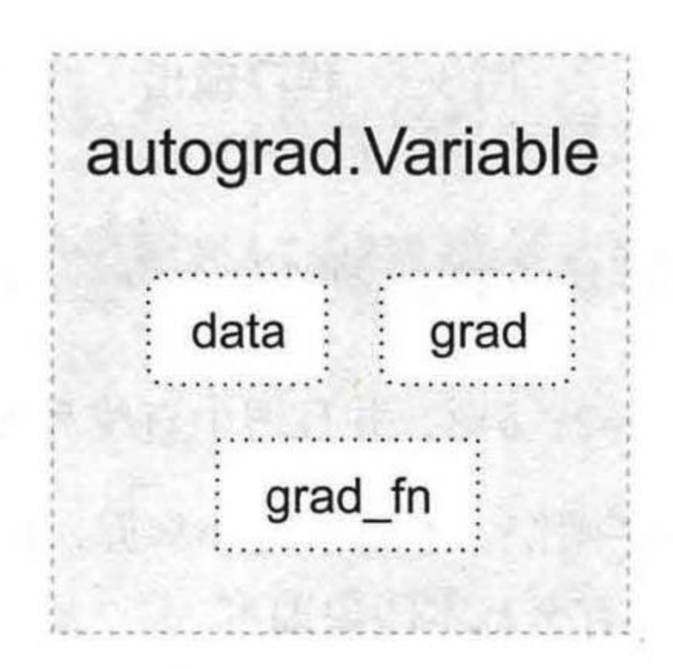


图 3-4 Variable 数据结构

Variable 的构造函数需要传入 tensor，同时有两个可选参数。

- `requires_grad (bool)`：是否需要对该 variable 进行求导。
- `volatile (bool)`：意为“挥发”，设置为 True，构建在该 variable 之上的图都不会求导，专为推理阶段设计。

Variable 支持大部分 tensor 支持的函数，但其不支持部分`inplace`函数，因为这些函数会修改 tensor 自身，而在反向传播中，variable 需要缓存原来的 tensor 来计算梯度。如果想要计算各个 Variable 的梯度，只需调用根节点 variable 的`backward`方法，autograd 会自动沿着计算图反向传播，计算每一个叶子节点的梯度。

[①] http://colah.github.io/posts/2015-08-Backprop/

`variable.backward(grad_variables=None, retain_graph=None, create_graph=None)` 主要有如下参数。

- grad_variables：形状与 variable 一致，对于`y.backward()`，grad_variables 相当于链式法则 $\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y} \times \frac{\partial y}{\partial x}$ 中的 $\frac{\partial z}{\partial y}$。grad_variables 也可以是 tensor 或序列。
- retain_graph：反向传播需要缓存一些中间结果，反向传播之后，这些缓存就被清空，可通过指定这个参数不清空缓存，用来多次反向传播。
- create_graph：对反向传播过程再次构建计算图，可通过`backward of backward`实现求高阶导数。

上述描述可能比较抽象，如果没有看懂也不用着急，笔者会在本节后半部分详细介绍，下面先看几个例子。

```
In: from __future__ import print_function
    import torch as t
    from torch.autograd import Variable as V
```

```
In: # 从tensor中创建variable，指定需要求导
    a = V(t.ones(3,4), requires_grad = True)
    a
```

```
Out: Variable containing:
      1  1  1  1
      1  1  1  1
      1  1  1  1
     [torch.FloatTensor of size 3x4]
```

```
In: b = V(t.zeros(3,4))
    b
```

```
Out: Variable containing:
      0  0  0  0
      0  0  0  0
      0  0  0  0
     [torch.FloatTensor of size 3x4]
```

```
In: # 函数的使用与tensor一致
    # 也可写成c = a + b
    c = a.add(b)
    c
```

```
Out: Variable containing:
      1  1  1  1
      1  1  1  1
      1  1  1  1
     [torch.FloatTensor of size 3x4]
```

```
In: d = c.sum()
    d.backward() # 反向传播
```

```
In: # 注意二者的区别
    # 前者在取data后变为tensor，从tensor计算sum得到float
    # 后者计算sum后仍然是Variable
    c.data.sum(), c.sum()
```

```
Out: (12.0, Variable containing:
      12
     [torch.FloatTensor of size 1])
```

```
In: a.grad
```

```
Out: Variable containing:
      1  1  1  1
      1  1  1  1
      1  1  1  1
     [torch.FloatTensor of size 3x4]
```

```
In: # 此处虽然没有指定c需要求导，但c依赖于a，而a需要求导，
    # 因此c的requires_grad属性会自动设为True
    a.requires_grad, b.requires_grad, c.requires_grad
```

```
Out: (True, False, True)
```

```
In: # 由用户创建的variable属于叶子节点，对应的grad_fn是None
    a.is_leaf, b.is_leaf, c.is_leaf
```

```
Out: (True, True, False)
```

```
In: # c.grad是None，c不是叶子节点，它的梯度是用来计算a的梯度
    # 虽然c.requires_grad = True,但其梯度计算完之后即被释放
    c.grad is None
```

```
Out: True
```

接着我们来看看 autograd 计算的导数和我们手动推导的导数的区别。

$$y = x^2 e^x$$

它的导函数是：

$$\frac{\partial y}{\partial x} = 2xe^x + x^2 e^x$$

```
In: def f(x):
        '''计算y'''
        y = x**2 * t.exp(x)
        return y

    def gradf(x):
        '''手动求导函数'''
        dx = 2*x*t.exp(x) + x**2*t.exp(x)
        return dx
```

```
In: x = V(t.randn(3,4), requires_grad = True)
    y = f(x)
    y
```

```
Out: Variable containing:
      0.5256  0.2784  1.1811  0.5001
      0.0017  0.1093  0.1900  0.0627
      0.1615  0.1219  0.5182  0.1972
     [torch.FloatTensor of size 3x4]
```

```
In: y.backward(t.ones(y.size())) # grad_variables形状与y一致
    x.grad
```

```
Out: Variable containing:
     -0.1015  1.5845  4.3401 -0.1718
     -0.0783  0.8723 -0.4612  0.6228
     -0.4577  0.9336 -0.1253 -0.4610
     [torch.FloatTensor of size 3x4]
```

```
In: # autograd的计算结果与利用公式手动计算的结果一致
    gradf(x)
```

```
Out: Variable containing:
     -0.1015  1.5845  4.3401 -0.1718
     -0.0783  0.8723 -0.4612  0.6228
     -0.4577  0.9336 -0.1253 -0.4610
     [torch.FloatTensor of size 3x4]
```

3.2.2 计算图

PyTorch 中autograd的底层采用了计算图，计算图是一种特殊的有向无环图（DAG），用于记录算子与变量之间的关系。一般用矩形表示算子，椭圆形表示变量。如表达式 $\boldsymbol{z}=\boldsymbol{wx}+\boldsymbol{b}$ 可分解为 $\boldsymbol{y}=\boldsymbol{wx}$ 和 $\boldsymbol{z}=\boldsymbol{y}+\boldsymbol{b}$，其计算图如图 3-5 所示，图中的MUL和ADD都是算子，$\boldsymbol{w}$、$\boldsymbol{x}$、$\boldsymbol{b}$ 为变量。

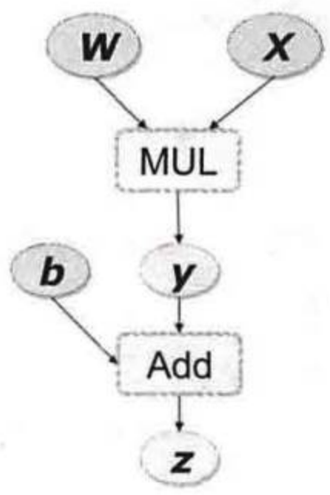


图 3-5　计算图

如上有向无环图中，$\boldsymbol{X}$ 和 $\boldsymbol{b}$ 是叶子节点（leaf node），这些节点通常由用户自己创建，不依赖于其他变量。$\boldsymbol{z}$ 称为根节点，是计算图的最终目标。利用链式法则很容易求得各个叶子节点的梯度。

$$\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{b}}=1, \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{y}}=1$$

$$\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{w}}=\boldsymbol{x}, \frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}=\boldsymbol{w}$$

$$\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{x}}=\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{y}}\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}=1*\boldsymbol{w}$$

$$\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{w}}=\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{y}}\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{w}}=1*\boldsymbol{x}$$

而有了计算图，上述链式求导即可利用计算图的反向传播自动完成，其过程如图 3-6 所示。

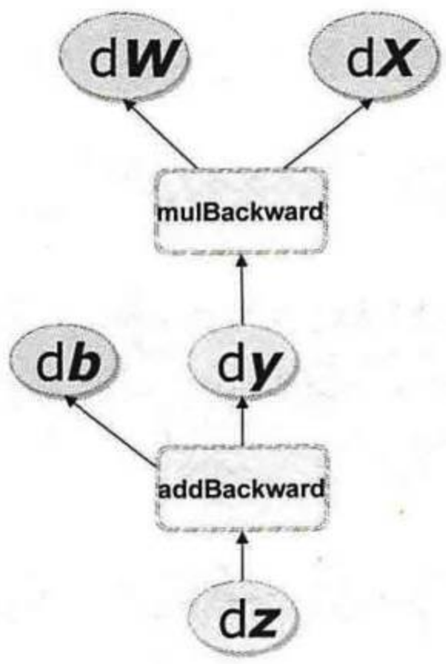


图 3-6 计算图的反向传播

在 PyTorch 实现中，autograd 会随着用户的操作，记录生成当前 variable 的所有操作，并由此建立一个有向无环图。用户每进行一个操作，相应的计算图就会发生改变。更底层的实现中，图中记录了操作`Function`，每一个变量在图中的位置可通过其`grad_fn`属性在图中的位置推测得到。在反向传播过程中，autograd 沿着这个图从当前变量（根节点 z）溯源，可以利用链式求导法则计算所有叶子节点的梯度。每一个前向传播操作的函数都有与之对应的反向传播函数用来计算输入的各个 variable 的梯度，这些函数的函数名通常以`Backward`结尾。下面结合代码学习 autograd 的实现细节。

```
In: x = V(t.ones(1))
    b = V(t.rand(1), requires_grad = True)
    w = V(t.rand(1), requires_grad = True)
    y = w * x # 等价于y=w.mul(x)
    z = y + b # 等价于z=y.add(b)
```

```
In: x.requires_grad, b.requires_grad, w.requires_grad
```

```
Out: (False, True, True)
```

```
In: # 虽然未指定y.requires_grad为True，但由于y依赖于需要求导的w
    # 故而y.requires_grad为True
    y.requires_grad
```

```
Out: True
```

```
In: x.is_leaf, w.is_leaf, b.is_leaf
```

```
Out: (True, True, True)
```

```
In: y.is_leaf, z.is_leaf
```

```
Out: (False, False)
```

```
In: # grad_fn可以查看这个variable的反向传播函数，
    # z是add函数的输出，所以它的反向传播函数是AddBackward
    z.grad_fn
```

```
Out: <torch.autograd.function.AddBackward at 0x7f6ca301b050>
```

```
In: # next_functions保存grad_fn的输入，grad_fn的输入是一个tuple
    # 第一个是y，它是乘法(mul)的输出，所以对应的反向传播函数y.grad_fn是
    MulBackward
    # 第二个是b，它是叶子节点，由用户创建，grad_fn为None，但是有
    z.grad_fn.next_functions
```

```
Out: ((<torch.autograd.function.MulBackward at 0x7f6ca3032ed8>, 0),
      (<AccumulateGrad at 0x7f6ca3007510>, 0))
```

```
In: # variable的grad_fn对应着图中的function
    z.grad_fn.next_functions[0][0] == y.grad_fn
```

```
Out: True
```

```
In: # 第一个是w，叶子节点，需要求导，梯度是累加的
    # 第二个是x，叶子节点，不需要求导，所以为None
    y.grad_fn.next_functions
```

```
Out: ((<AccumulateGrad at 0x7f6ca30076d0>, 0), (None, 0))
```

```
In: # 叶子节点的grad_fn是None
    w.grad_fn,x.grad_fn
```

```
Out: (None, None)
```

计算 $\boldsymbol{w}$ 的梯度时需要用到 $\boldsymbol{x}$ 的数值 ($\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{w}} = \boldsymbol{x}$)，这些数值在前向过程中会保存成 buffer，在计算完梯度之后会自动清空。为了能够多次反向传播需要指定 `retain_graph` 来保留这些 buffer。

```
In: y.grad_fn.saved_variables
```

```
Out: (Variable containing:
      0.2473
     [torch.FloatTensor of size 1], Variable containing:
      1
     [torch.FloatTensor of size 1])
```

```
In: # 使用retain_graph保存buffer
    z.backward(retain_graph=True)
    w.grad
```

```
Out: Variable containing:
      1
     [torch.FloatTensor of size 1]
```

```
In: # 多次反向传播，梯度累加，这也就是w中AccumulateGrad标识的含义
    z.backward()
    w.grad
```

```
Out: Variable containing:
      2
     [torch.FloatTensor of size 1]
```

```
In: # 会报错，因为此时保存的buffer已经被清空了
    # y.grad_fn.saved_variables
```

PyTorch 使用的是动态图，它的计算图在每次前向传播时都是从头开始构建的，所以它能够使用 Python 控制语句（如 for、if 等），根据需求创建计算图。这一点在自然语言处理领域中很有用，它意味着你不需要事先构建所有可能用到的图的路径，图在运行时才构建。

```
In: def abs(x):
        if x.data[0]>0: return x
        else: return -x
    x = V(t.ones(1),requires_grad=True)
    y = abs(x)
    y.backward()
    x.grad
```

```
Out: Variable containing:
      1
     [torch.FloatTensor of size 1]
```

```
In: x = V(-1*t.ones(1),requires_grad=True)
    y = abs(x)
    y.backward()
    print(x.grad)
```

```
Print: Variable containing:
       -1
       [torch.FloatTensor of size 1]
```

```
In: def f(x):
        result = 1
        for ii in x:
            if ii.data[0]>0: result=ii*result
        return result
    x = V(t.arange(-2,4),requires_grad=True)
    y = f(x) # y = x[3]*x[4]*x[5]
    y.backward()
    x.grad
```

```
Out: Variable containing:
       0
       0
       0
       6
       3
       2
     [torch.FloatTensor of size 6]
```

变量的`requires_grad`属性默认为 False，如果某一个节点 requires_grad 被设置为 True，那么所有依赖它的节点`requires_grad`都是 True。这其实很好理解，对于 $\boldsymbol{x} \rightarrow \boldsymbol{y} \rightarrow \boldsymbol{z}$，x.requires_grad = True。当需要计算 $\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{x}}$ 时，根据链式法则，$\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{x}} = \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{y}} \frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}$，自然也需要求 $\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{y}}$，所以 y.requires_grad 会被自动标为 True。

`volatile=True`是另外一个很重要的标识，它能够将所有依赖于它的节点全部设为`volatile=True`，其优先级比`requires_grad=True`高。`volatile=True`的节点不会求导，即使`requires_grad=True`，也无法进行反向传播。对于不需要反向传播的情景（如 inference，即测试推理时），该参数可实现一定程度的速度提升，并节省约一半显存，因为其不需要分配空间保存梯度。

```
In: x = V(t.ones(1))
    w = V(t.rand(1), requires_grad=True)
    y = x * w
    # y依赖于w, 而w.requires_grad = True
    x.requires_grad, w.requires_grad, y.requires_grad
```

```
Out: (False, True, True)
```

```
In: x = V(t.ones(1), volatile=True)
    w = V(t.rand(1), requires_grad = True)
    y = x * w
    # y依赖于w和x, 但x.volatile = True, w.requires_grad = True
    x.requires_grad, w.requires_grad, y.requires_grad
```

```
Out: (False, True, False)
```

```
In: x.volatile, w.volatile, y.volatile
```

```
Out: (True, False, True)
```

在反向传播过程中非叶子节点的导数计算完之后即被清空。若想查看这些变量的梯度，有以下两种方法：

- 使用 autograd.grad 函数
- 使用 hook

`autograd.grad`和`hook`方法都是很强大的工具，更详细的用法参考官方 api 文档，这里只举例说明其基础的使用方法。推荐使用`hook`方法，但是在实际使用中应尽量避免修改 grad 的值。

```
In: x = V(t.ones(3), requires_grad=True)
    w = V(t.rand(3), requires_grad=True)
    y = x * w
    # y依赖于w, 而w.requires_grad = True
    z = y.sum()
    x.requires_grad, w.requires_grad, y.requires_grad
```

```
Out: (True, True, True)
```

```
In: # 非叶子节点grad计算完之后自动清空, y.grad是None
    z.backward()
    (x.grad, w.grad, y.grad)
```

```
Out: (Variable containing:
      0.9814
      0.6895
      0.6189
     [torch.FloatTensor of size 3], Variable containing:
      1
      1
      1
     [torch.FloatTensor of size 3], None)
```

```
In: # 第一种方法：使用grad获取中间变量的梯度
    x = V(t.ones(3), requires_grad=True)
    w = V(t.rand(3), requires_grad=True)
    y = x * w
    z = y.sum()
    # z对y的梯度，隐式调用backward()
    t.autograd.grad(z, y)
```

```
Out: (Variable containing:
      1
      1
      1
     [torch.FloatTensor of size 3],)
```

```
In: # 第二种方法：使用hook
    # hook是一个函数，输入是梯度，不应该有返回值
    def variable_hook(grad):
        print('y的梯度：  \r\n',grad)

    x = V(t.ones(3), requires_grad=True)
    w = V(t.rand(3), requires_grad=True)
    y = x * w
    # 注册hook
    hook_handle = y.register_hook(variable_hook)
    z = y.sum()
    z.backward()

    # 除非你每次都要用hook，否则用完之后记得移除hook
    hook_handle.remove()
```

```
Print: y的梯度:
        Variable containing:
        1
        1
        1
       [torch.FloatTensor of size 3]
```

最后再来看看 variable 中 grad 属性和 backward 函数 `grad_variables` 参数的含义。

- variable $\boldsymbol{x}$ 的梯度是目标函数 $f(x)$ 对 $\boldsymbol{x}$ 的梯度，$\frac{\partial f(\boldsymbol{x})}{\partial \boldsymbol{x}} = (\frac{\partial f(\boldsymbol{x})}{\partial x_0}, \frac{\partial f(\boldsymbol{x})}{\partial x_1}, ..., \frac{\partial f(\boldsymbol{x})}{\partial x_N})$，形状和 $\boldsymbol{x}$ 一致。
- y.backward(grad_variables) 中的 grad_variables 相当于链式求导法则中的 $\frac{\partial z}{\partial \boldsymbol{x}} = \frac{\partial z}{\partial \boldsymbol{y}} \frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}$ 中的 $\frac{\partial z}{\partial \boldsymbol{y}}$。$z$ 是目标函数，一般是一个标量，故而 $\frac{\partial z}{\partial \boldsymbol{y}}$ 的形状与 $\boldsymbol{y}$ 的形状一致。`z.backward()` 等价于 y.backward(grad_y)。`z.backward()` 省略了 grad_variables 参数，是因为 z 是一个标量，而 $\frac{\partial z}{\partial z} = 1$。

```
In: x = V(t.arange(0,3), requires_grad=True)
    y = x**2 + x*2
    z = y.sum()
    z.backward() # 从z开始反向传播
    x.grad
```

```
Out: Variable containing:
      2
      4
      6
     [torch.FloatTensor of size 3]
```

```
In: x = V(t.arange(0,3), requires_grad=True)
    y = x**2 + x*2
    z = y.sum()
    y_grad_variables = V(t.Tensor([1,1,1])) # dz/dy
    y.backward(y_grad_variables) #从y开始反向传播
    x.grad
```

```
Out: Variable containing:
      2
      4
      6
     [torch.FloatTensor of size 3]
```

值得注意的是，只有对 variable 的操作才能使用 autograd，如果对 variable 的 data 直接进行操作，将无法使用反向传播。除了参数初始化，一般我们不会直接修改 variable.data 的值。

在 PyTorch 中计算图的特点可总结如下。

- autograd 根据用户对 variable 的操作构建计算图。对 variable 的操作抽象为`Function`。
- 由用户创建的节点称为叶子节点，叶子节点的`grad_fn`为 None。叶子节点中需要求导的 variable，具有`AccumulateGrad`标识，因其梯度是累加的。
- variable 默认是不需要求导的，即`requires_grad`属性默认为 False。如果某一个节点 requires_grad 被设置为 True，那么所有依赖它的节点`requires_grad`都为 True。
- variable 的`volatile`属性默认为 False，如果某一个 variable 的`volatile`属性被设为 True，那么所有依赖它的节点`volatile`属性都为 True。`volatile`属性为 True 的节点不会求导，`volatile`的优先级比`requires_grad`高。
- 多次反向传播时，梯度是累加的。反向传播的中间缓存会被清空，为进行多次反向传播需指定`retain_graph`=True 来保存这些缓存。
- 非叶子节点的梯度计算完之后即被清空，可以使用`autograd.grad`或`hook`技术获取非叶子节点梯度的值。
- variable 的 grad 与 data 形状一致，应避免直接修改 variable.data，因为对 data 的直接操作无法利用 autograd 进行反向传播。
- 反向传播函数`backward`的参数`grad_variables`可以看成链式求导的中间结果，如果是标量，可以省略，默认为 1。
- PyTorch 采用动态图设计，可以很方便地查看中间层的输出，动态地设计计算图结构。

3.2.3 扩展 autograd

目前，绝大多数函数都可以使用`autograd`实现反向求导，但如果需要自己写一个复杂的函数，不支持自动反向求导怎么办？答案是写一个`Function`，实现它的前向传播和反向传播代码，`Function`对应于计算图中的矩形，它接收参数，计算并返回结果。下面给出一个例子。

```
class Mul(Function):

    @staticmethod
```

```
    def forward(ctx, w, x, b, x_requires_grad = True):
        ctx.x_requires_grad = x_requires_grad
        ctx.save_for_backward(w,x)
        output = w * x + b
        return output

    @staticmethod
    def backward(ctx, grad_output):
        w,x = ctx.saved_variables
        grad_w = grad_output * x
        if ctx.x_requires_grad:
            grad_x = grad_output * w
        else:
            grad_x = None
        grad_b = grad_output * 1
        return grad_w, grad_x, grad_b, None
```

对以上代码的分析如下。

- 自定义的 Function 需要继承 autograd.Function，没有构造函数`__init__`，forward 和 backward 函数都是静态方法。
- forward 函数的输入和输出都是 tensor，backward 函数的输入和输出都是 variable。
- backward 函数的输出和 forward 函数的输入一一对应，backward 函数的输入和 forward 函数的输出一一对应。
- backward 函数的 grad_output 参数即 t.autograd.backward 中的`grad_variables`。
- 如果某一个输入不需要求导，直接返回 None，例如 forward 中的输入参数 x_requires_grad 显然无法对它求导，直接返回 None 即可。
- 反向传播可能需要利用前向传播的某些中间结果，在前向传播过程中，需要保存中间结果，否则前向传播结束后这些对象即被释放。

使用 Function.apply(variable) 即可调用实现的 Function。

```
In: from torch.autograd import Function
    class MultiplyAdd(Function):

        @staticmethod
        def forward(ctx, w, x, b):
            print('type in forward',type(x))
```

```
        ctx.save_for_backward(w,x)
        output = w * x + b
        return output

    @staticmethod
    def backward(ctx, grad_output):
        w,x = ctx.saved_variables
        print('type in backward',type(x))
        grad_w = grad_output * x
        grad_x = grad_output * w
        grad_b = grad_output * 1
        return grad_w, grad_x, grad_b
```

```
In: x = V(t.ones(1))
    w = V(t.rand(1), requires_grad = True)
    b = V(t.rand(1), requires_grad = True)
    print('开始前向传播')
    z=MultiplyAdd.apply(w, x, b)
    print('开始反向传播')
    z.backward()

    # x不需要求导，中间过程还是会计算它的导数，但随后被清空
    x.grad, w.grad, b.grad
```

```
Print: 开始前向传播
       type in forward <class 'torch.FloatTensor'>
       开始反向传播
       type in backward <class 'torch.autograd.variable.Variable'>
```

```
Out: (None, Variable containing:
       1
      [torch.FloatTensor of size 1], Variable containing:
       1
      [torch.FloatTensor of size 1])
```

```
In: x = V(t.ones(1))
    w = V(t.rand(1), requires_grad = True)
    b = V(t.rand(1), requires_grad = True)
```

```
print('开始前向传播')
z=MultiplyAdd.apply(w,x,b)
print('开始反向传播')

# 调用MultiplyAdd.backward
# 输出grad_w, grad_x, grad_b
z.grad_fn.apply(V(t.ones(1)))
```

```
Print: 开始前向传播
       type in forward <class 'torch.FloatTensor'>
       开始反向传播
       type in backward <class 'torch.autograd.variable.Variable'>
```

```
Out: (Variable containing:
       1
      [torch.FloatTensor of size 1], Variable containing:
       0.6984
      [torch.FloatTensor of size 1], Variable containing:
       1
      [torch.FloatTensor of size 1])
```

forward 函数的输入是 tensor，而 backward 函数的输入是 variable，这是为了实现高阶求导。backward 函数的输入值和返回值是 variable，但在实际使用时 autograd.Function 会将输入 variable 提取为 tensor，并将计算结果的 tensor 封装成 variable 返回。在 backward 函数中要对 variable 进行操作，是为了能够计算梯度的梯度。下面举例说明，有关 torch.autograd.grad 的更详细使用方法请参照文档。

```
In: x = V(t.Tensor([5]), requires_grad=True)
    y = x ** 2
    grad_x = t.autograd.grad(y, x, create_graph=True)
    grad_x # dy/dx = 2 * x
```

```
Out: (Variable containing:
       10
      [torch.FloatTensor of size 1],)
```

```
In: grad_grad_x = t.autograd.grad(grad_x[0],x)
    grad_grad_x # 二阶导数 d(2x)/dx = 2
```

```
Out: (Variable containing:
      2
     [torch.FloatTensor of size 1],)
```

这种设计在 PyTorch 0.2 中引入，虽然能让autograd具有高阶求导功能，但其也限制了 Tensor 的使用，因为 autograd 中反向传播的函数只能利用当前已经有的 Variable 操作。为了更好的灵活性，也为了兼容旧版本的代码，PyTorch 还提供了另外一种扩展 autograd 的方法。利用装饰器 `@once_differentiable`，能够在 backward 函数中自动将输入的 variable 提取成 tensor，把计算结果的 tensor 自动封装成 variable。有了这个特性，我们就能够很方便地使用 numpy/scipy 中的函数，操作不再局限于 variable 所支持的操作。这种做法正如名字中所暗示的那样只能求导一次，它打断了反向传播图，不再支持高阶求导。

上面所描述的都是新式 Function，还有个 legacy Function，可以带有 `__init__`方法，`forward`和`backwad`函数也不需要声明为 `@staticmethod`，但随着版本更迭，会越来越少遇到此类 Function，在此不做更多介绍。

在实现了自己的 Function 之后，还可以使用`gradcheck`函数检测实现是否正确。`gradcheck` 通过数值逼近计算梯度，可能具有一定的误差，通过控制`eps`的大小可以控制容忍的误差。新版 autograd 的内容可以参考 GitHub 上开发者的讨论[①]。

下面举例说明如何利用 Function 实现 Sigmoid Function。

```
In: class Sigmoid(Function):

        @staticmethod
        def forward(ctx, x):
            output = 1 / (1 + t.exp(-x))
            ctx.save_for_backward(output)
            return output

        @staticmethod
        def backward(ctx, grad_output):
            output,  = ctx.saved_variables
            grad_x = output * (1 - output) * grad_output
            return grad_x
```

[①] https://github.com/pytorch/pytorch/pull/1016

```
In: # 采用数值逼近的方式检验计算梯度的公式对不对
    test_input = V(t.randn(3,4), requires_grad=True)
    t.autograd.gradcheck(Sigmoid.apply, (test_input,), eps=1e-3)
```

```
Out: True
```

```
In: def f_sigmoid(x):
        y = Sigmoid.apply(x)
        y.backward(t.ones(x.size()))

    def f_naive(x):
        y =  1/(1 + t.exp(-x))
        y.backward(t.ones(x.size()))

    def f_th(x):
        y = t.sigmoid(x)
        y.backward(t.ones(x.size()))

    x=V(t.randn(100, 100), requires_grad=True)
    %timeit -n 100 f_sigmoid(x)
    %timeit -n 100 f_naive(x)
    %timeit -n 100 f_th(x)
```

```
Print: 100 loops, best of 3: 328 µs per loop
       100 loops, best of 3: 539 µs per loop
       100 loops, best of 3: 269 µs per loop
```

显然f_sigmoid要比单纯利用autograd加减和乘方操作实现的函数快不少，因为 f_sigmoid 的 backward 优化了反向传播的过程。另外，可以看出系统实现的 builtin 接口（t.sigmoid）更快。

3.2.4 小试牛刀：用 Variable 实现线性回归

在 3.2.3 节中讲解了利用 tensor 实现线性回归，本节将讲解如何利用 autograd/Variable 实现线性回归，读者可以从中体会 autograd 的便捷之处。

```
In: import torch as t
    from torch.autograd import Variable as V
```

```
%matplotlib inline
from matplotlib import pyplot as plt
from IPython import display
```

```
In: # 为了在不同的计算机上运行时下面的输出一致，设置随机数种子
    t.manual_seed(1000)


    def get_fake_data(batch_size=8):
        ''' 产生随机数据：y = x*2 + 3，加上了一些噪声'''
        x = t.rand(batch_size,1) * 20
        y = x * 2 + (1 + t.randn(batch_size, 1))*3
        return x, y
```

```
In: # 来看看产生的x-y分布是什么样的，输出如图3-7所示
    x, y = get_fake_data()
    plt.scatter(x.squeeze().numpy(), y.squeeze().numpy())
```

```
Out: <matplotlib.collections.PathCollection at 0x7f6d0e425890>
```

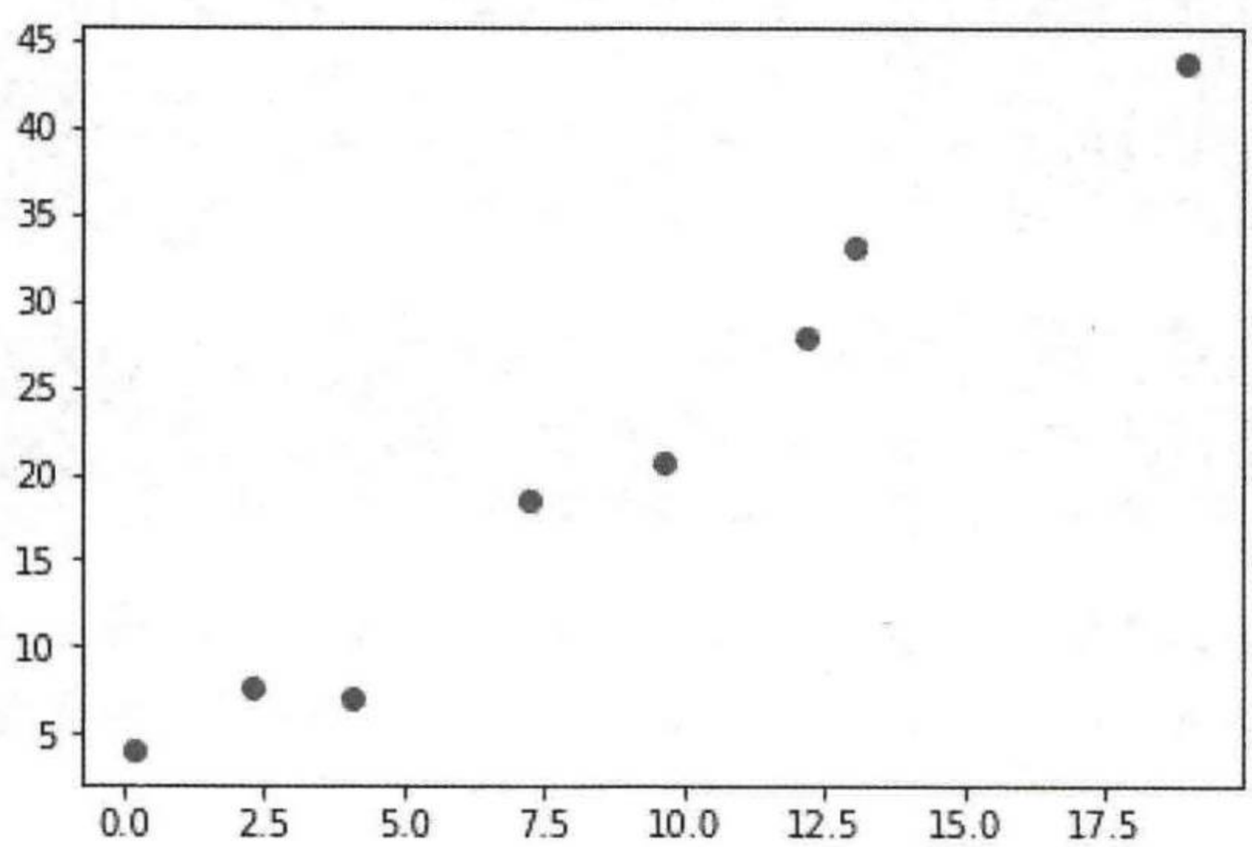


图 3-7　程序输出：x-y 分布

```
In: # 随机初始化参数
    w = V(t.rand(1,1), requires_grad=True)
    b = V(t.zeros(1,1), requires_grad=True)


    lr =0.001 # 学习率
```

```
for ii in range(8000):
    x, y = get_fake_data()
    x, y = V(x), V(y)

    # forward: 计算loss
    y_pred = x.mm(w) + b.expand_as(y)
    loss = 0.5 * (y_pred - y) ** 2
    loss = loss.sum()

    # backward: 自动计算梯度
    loss.backward()

    # 更新参数
    w.data.sub_(lr * w.grad.data)
    b.data.sub_(lr * b.grad.data)

    # 梯度清零
    w.grad.data.zero_()
    b.grad.data.zero_()

    if ii%1000 ==0:
        # 画图
        display.clear_output(wait=True)
        x = t.arange(0, 20).view(-1, 1)
        y = x.mm(w.data) + b.data.expand_as(x)
        plt.plot(x.numpy(), y.numpy()) # predicted

        x2, y2 = get_fake_data(batch_size=20)
        plt.scatter(x2.numpy(), y2.numpy()) # true data

        plt.xlim(0,20)
        plt.ylim(0,41)
        plt.show() # 程序输出如图3-8所示
        plt.pause(0.5)

print(w.data.squeeze()[0], b.data.squeeze()[0])
```

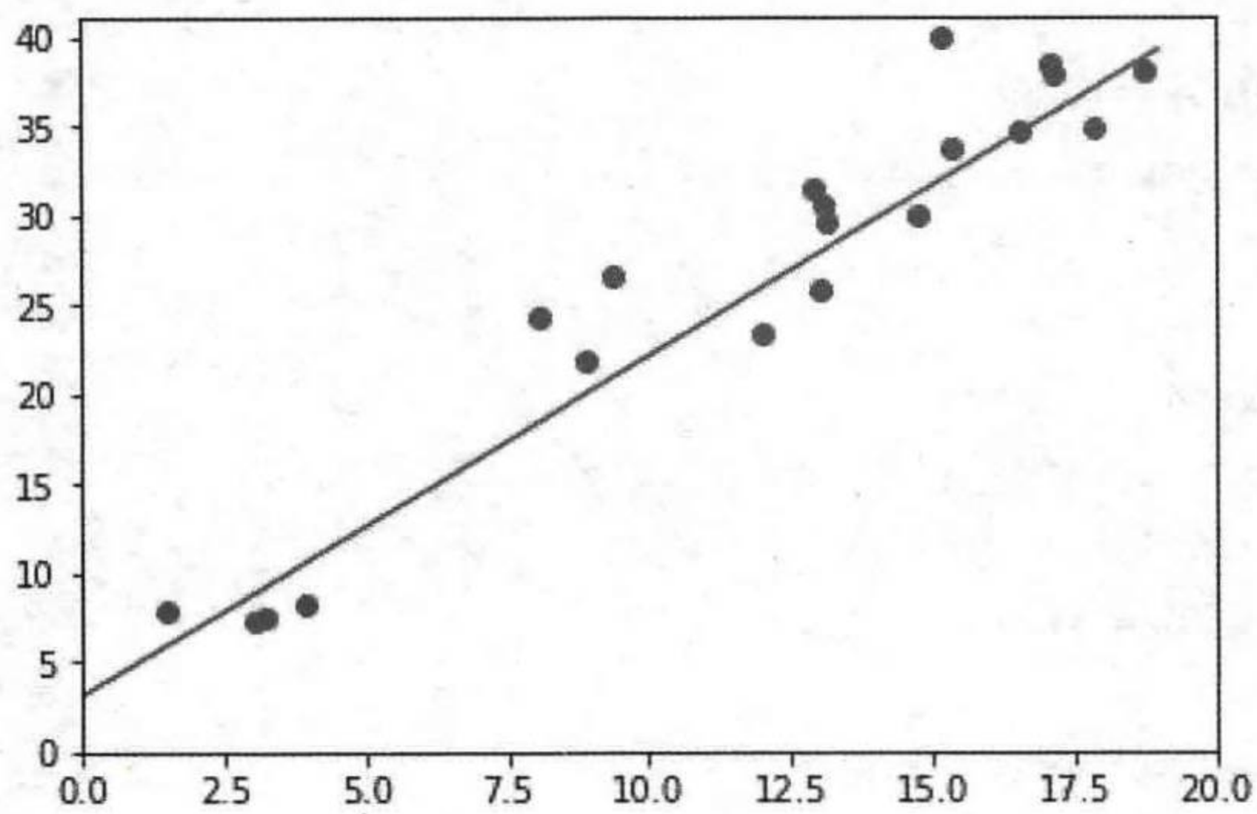


图 3-8　程序输出：x-y 分布与拟合直线

```
Print: 1.93683743477 3.01666927338
```

用 autograd 实现的线性回归最大的不同点就在于利用 autograd 不需要手动计算梯度，可以自动微分。这一点不单是在深度学习中，在许多机器学习的问题中都很有用。另外，需要注意的是在每次反向传播之前要记得先把梯度清零。

本章主要介绍了 PyTorch 中两个基础的数据结构：Tensor 和 autograd 中的 Variable。Tensor 是一个类似 numpy 数组的数据结构，能高效执行数据计算，有着和 numpy 类似的接口，并提供简单易用的 GPU 加速。Variable 封装了 Tensor 并提供自动求导技术，具有和 Tensor 几乎一样的接口。`autograd`是 PyTorch 的自动微分引擎，采用动态计算图技术，能够快速高效地计算导数。

除了讲解 Tensor 和 autograd 的基础用法，本章还介绍了 Tensor 和 autograd 的底层原理和设计思想。部分内容可能比较复杂，即使读者难以理解也不影响使用，但是了解这些运作原理，有助于更好地掌握 PyTorch。

4 神经网络工具箱 nn

autograd实现了自动微分系统，然而对于深度学习来说过于底层，本章将介绍的nn模块，是构建于autograd之上的神经网络模块。除了nn之外，我们还会介绍神经网络中常用的工具，比如优化器optim、初始化init等。

4.1 nn.Module

第 3 章中提到，使用 autograd 可实现深度学习模型，但其抽象程度较低，如果用其来实现深度学习模型，则需要编写的代码量极大。在这种情况下，torch.nn 应运而生，其是专门为深度学习设计的模块。torch.nn 的核心数据结构是`Module`，它是一个抽象的概念，既可以表示神经网络中的某个层（layer），也可以表示一个包含很多层的神经网络。在实际使用中，最常见的做法是继承`nn.Module`，撰写自己的网络/层。下面先来看看如何用 nn.Module 实现自己的全连接层。全连接层，又名仿射层，输出 $\boldsymbol{y}$ 和输入 $\boldsymbol{x}$ 满足 $\boldsymbol{y}=\boldsymbol{W}\boldsymbol{x}+\boldsymbol{b}$，$\boldsymbol{W}$ 和 $\boldsymbol{b}$ 是可学习的参数。

```
In: import torch as t
    from torch import nn
    from torch.autograd import Variable as V
```

```
In: class Linear(nn.Module): # 继承nn.Module
        def __init__(self, in_features, out_features):
            super(Linear, self).__init__() #等价于nn.Module.__init__(self)
            self.w = nn.Parameter(t.randn(in_features, out_features))
```

```
        self.b = nn.Parameter(t.randn(out_features))

    def forward(self, x):
        x = x.mm(self.w)
        return x + self.b.expand_as(x)
```

```
In: layer = Linear(4,3)
    input = V(t.randn(2,4))
    output = layer(input)
    output
```

```
Out: Variable containing:
     -1.4047  0.5203  1.3081
     -2.3126  5.1158  1.1852
     [torch.FloatTensor of size 2x3]
```

```
In: for name, parameter in layer.named_parameters():
        print(name, parameter) # w and b
```

```
Print: w Parameter containing:
       -0.4691  0.3211 -0.8036
       -0.5587  1.5329  0.1607
        1.6624 -0.5606 -1.1634
       -0.5175  1.8075  0.4781
       [torch.FloatTensor of size 4x3]

       b Parameter containing:
        0.3283
       -0.0053
        0.6120
       [torch.FloatTensor of size 3]
```

可见，全连接层的实现非常简单，其代码量不超过 10 行，但需注意以下几点。

- 自定义层`Linear`必须继承`nn.Module`，并且在其构造函数中需调用`nn.Module`的构造函数，即`super(Linear, self).__init__()`或`nn.Module.__init__(self)`。
- 在构造函数`__init__`中必须自己定义可学习的参数，并封装成`Parameter`，如在本例中我们把`w`和`b`封装成`Parameter`。`Parameter`是一种特殊的`Variable`，但其默认需要求导（requires_grad = True），感兴趣的读者可以通过`nn.Parameter??`查看`Parameter`类的源代码。

- forward函数实现前向传播过程，其输入可以是一个或多个 variable，对 x 的任何操作也必须是 variable 支持的操作。
- 无须写反向传播函数，因其前向传播都是对 variable 进行操作，nn.Module 能够利用 autograd 自动实现反向传播，这一点比 Function 简单许多。
- 使用时，直观上可将 layer 看成数学概念中的函数，调用 layer(input) 即可得到 input 对应的结果。它等价于layers.__call__(input)，在__call__函数中，主要调用的是 layer.forward(x)，另外还对钩子做了一些处理。所以在实际使用中应尽量使用layer(x)而不是使用layer.forward(x)，关于钩子技术的具体内容将在下文讲解。
- Module中的可学习参数可以通过named_parameters()或者parameters()返回迭代器，前者会给每个 parameter 附上名字，使其更具有辨识度。

可见利用 Module 实现的全连接层，比利用Function实现的更简单，因其不再需要写反向传播函数。

Module 能够自动检测到自己的parameter，并将其作为学习参数。除了parameter，Module 还包含子Module，主 Module 能够递归查找子Module中的parameter。下面再来看看稍微复杂一点的网络：多层感知机。

多层感知机的网络结构如图 4-1 所示，它由两个全连接层组成，采用 sigmoid 函数作为激活函数（图中没有画出）。

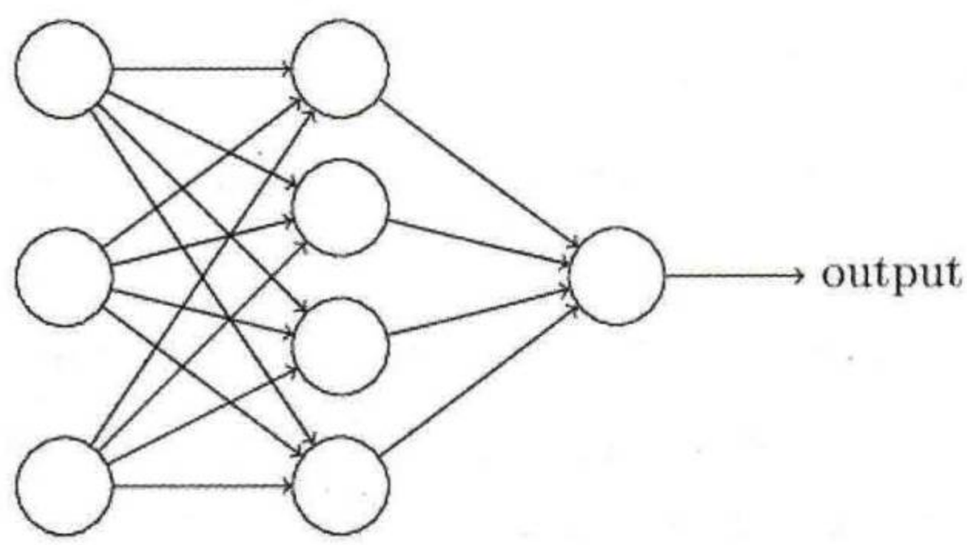


图 4-1　多层感知机

```
In: class Perceptron(nn.Module):
        def __init__(self, in_features, hidden_features, out_features):
            nn.Module.__init__(self)
            self.layer1 = Linear(in_features, hidden_features) # 此处的
            Linear是前面自定义的全连接层
            self.layer2 = Linear(hidden_features, out_features)
        def forward(self,x):
```

```
        x = self.layer1(x)
        x = t.sigmoid(x)
        return self.layer2(x)
```

```
In: perceptron = Perceptron(3,4,1)
    for name, param in perceptron.named_parameters():
        print(name, param.size())
```

```
Print: layer1.w torch.Size([3, 4])
       layer1.b torch.Size([4])
       layer2.w torch.Size([4, 1])
       layer2.b torch.Size([1])
```

可见，即使是稍复杂的多层感知机，其实现依旧很简单。这里需要注意以下两个知识点。

- 构造函数`__init__`中，可利用前面自定义的 Linear 层（module）作为当前 module 对象的一个子 module，它的可学习参数，也会成为当前 module 的可学习参数。
- 在前向传播函数中，我们有意识地将输出变量都命名成`x`，是为了能让 Python 回收一些中间层的输出，从而节省内存。但并不是所有的中间结果都会被回收，有些 variable 虽然名字被覆盖，但其在反向传播时仍需要用到，此时 Python 的内存回收模块将通过检查引用计数，不会回收这一部分内存。

module 中 parameter 的全局命名规范如下。

- Parameter 直接命名。例如`self.param_name = nn.Parameter(t.randn(3, 4))`，命名为`param_name`。
- 子 module 中的 parameter，会在其名字之前加上当前 module 的名字。例如`self.sub_module = SubModel()`，SubModel 中有个 parameter 的名字也叫做 param_name，那么二者拼接而成的 parameter name 就是`sub_module.param_name`。

为方便用户使用，PyTorch 实现了神经网络中绝大多数的 layer，这些 layer 都继承于 nn.Module，封装了可学习参数`parameter`，并实现了 forward 函数，且专门针对 GPU 运算进行了 CuDNN 优化，其速度和性能都十分优异。本书不准备对 nn.Module 中的所有层进行详细介绍，具体内容读者可参照官方文档[①]或在 IPython/Jupyter 中使用`nn.layer?`查看。阅读文档时应主要关注以下几点。

- 构造函数的参数，如 nn.Linear(in_features, out_features, bias)，需关注这三个参数的作用。

[①] http://pytorch.org/docs/nn.html

- 属性、可学习参数和子 module。如 nn.Linear 中有`weight`和`bias`两个可学习参数，不包含子 module。
- 输入输出的形状，如 nn.linear 的输入形状是 (N, input_features)，输出为 (N，output_features)，N 是 batch_size。

这些自定义 layer 对输入形状都有假设：输入的不是单个数据，而是一个 batch。若想输入一个数据，必须调用`unsqueeze(0)`函数将数据伪装成 batch_size=1 的 batch。

下面将从应用层面出发，对一些常用的 layer 做简单介绍，更详细的用法请查看官方文档。

4.2 常用的神经网络层

4.2.1 图像相关层

图像相关层主要包括卷积层（Conv）、池化层（Pool）等，这些层在实际使用中可分为一维（1D）、二维（2D）和三维（3D），池化方式又分为平均池化（AvgPool）、最大值池化（MaxPool）、自适应池化（AdaptiveAvgPool）等。卷积层除了常用的前向卷积外，还有逆卷积（TransposeConv）。下面举例说明。

```
In: from PIL import Image
    from torchvision.transforms import ToTensor, ToPILImage
    to_tensor = ToTensor() # img -> tensor
    to_pil = ToPILImage()
    lena = Image.open('imgs/lena.png')
    lena # 程序输出如图4-2所示
```

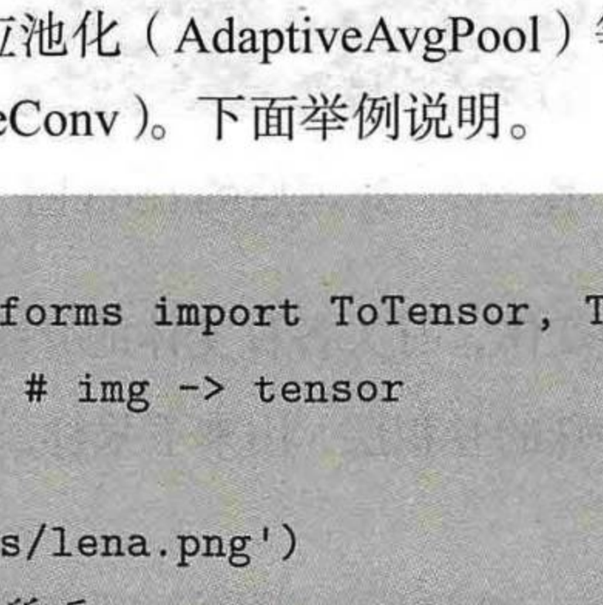


图 4-2 程序输出：Lena 图

```
In: # 输入是一个batch, batch_size=1
    input = to_tensor(lena).unsqueeze(0)

    # 锐化卷积核
    kernel = t.ones(3, 3)/-9.
    kernel[1][1] = 1
    conv = nn.Conv2d(1, 1, (3, 3), 1, bias=False)
    conv.weight.data = kernel.view(1, 1, 3, 3)

    out = conv(V(input))
    to_pil(out.data.squeeze(0))  # 程序输出如图4-3所示
```

图 4-3　处理后的 Lena 图

图像的卷积操作还有各种变体，有关各种变体的介绍可以参照此处的介绍[①]。

池化层可以看作是一种特殊的卷积层，用来下采样。但池化层没有可学习参数，其 weight 是固定的。

```
In: pool = nn.AvgPool2d(2,2)
    list(pool.parameters())
```

```
Out: []
```

```
In: out = pool(V(input))
    to_pil(out.data.squeeze(0))  # 程序输出如图4-4所示
```

[①] https://github.com/vdumoulin/conv_arithmetic/blob/master/README.md

图 4-4 池化处理后的 Lena 图

除了卷积层和池化层，深度学习中还将常用到以下几个层。

- Linear：全连接层。
- BatchNorm：批规范化层，分为 1D、2D 和 3D。除了标准的 BatchNorm 之外，还有在风格迁移中常用到的 InstanceNorm 层。
- Dropout：dropout 层，用来防止过拟合，同样分为 1D、2D 和 3D。

下面通过例子讲解它们的使用方法。

```
In: # 输入 batch_size=2, 维度3
    input = V(t.randn(2, 3))
    linear = nn.Linear(3, 4)
    h = linear(input)
    h
```

```
Out: Variable containing:
      0.5981  0.5827 -1.1883  1.1446
     -0.5606  0.5075 -0.6172  0.3501
     [torch.FloatTensor of size 2x4]
```

```
In: # 4 channel, 初始化标准差为4, 均值为0
    bn = nn.BatchNorm1d(4)
    bn.weight.data = t.ones(4) * 4
    bn.bias.data = t.zeros(4)

    bn_out = bn(h)
    # 注意输出的均值和方差
    # 方差是标准差的平方, 计算无偏方差分母会减1
    # 使用unbiased=False, 分母不减1
    bn_out.mean(0), bn_out.var(0, unbiased=False)
```

```
Out: (Variable containing:
      1.00000e-06 *
        0.0000
        3.0994
        0.0000
        0.0000
      [torch.FloatTensor of size 4], Variable containing:
       15.9995
       15.8876
       15.9980
       15.9990
      [torch.FloatTensor of size 4])
```

```
In: # 每个元素以0.5的概率舍弃
    dropout = nn.Dropout(0.5)
    o = dropout(bn_out)
    o # 有一半左右的数变为0
```

```
Out: Variable containing:
       7.9999  0.0000 -7.9995  7.9997
      -7.9999 -0.0000  7.9995 -0.0000
      [torch.FloatTensor of size 2x4]
```

以上很多例子中都对 module 的属性直接操作，其大多数是可学习参数，一般会随着学习的进行而不断改变。实际使用中除非需要使用特殊的初始化，否则应尽量不要直接修改这些参数。

4.2.2 激活函数

PyTorch 实现了常见的激活函数，其具体的接口信息可参见官方文档[①]，这些激活函数可作为独立的 layer 使用。这里将介绍最常用的激活函数 ReLU，其数学表达式为：

$$\mathrm{ReLU}(x) = \max(0, x)$$

```
In: relu = nn.ReLU(inplace=True)
```

[①] http://pytorch.org/docs/nn.html#non-linear-activations

```
input = V(t.randn(2, 3))
print(input)
output = relu(input)
print(output) # 小于0的都被截断为0
# 等价于input.clamp(min=0)
```

```
Print: Variable containing:
       -0.0438  0.6640  0.9179
        1.9569  0.9913  1.7955
       [torch.FloatTensor of size 2x3]

       Variable containing:
        0.0000  0.6640  0.9179
        1.9569  0.9913  1.7955
       [torch.FloatTensor of size 2x3]
```

ReLU 函数有个 inplace 参数，如果设为 True，它会把输出直接覆盖到输入中，这样可以节省内存/显存。之所以可以覆盖是因为在计算 ReLU 的反向传播时，只需根据输出就能够推算出反向传播的梯度。但是只有少数的 autograd 操作支持 inplace 操作（如 variable.sigmoid_()），除非你明确地知道自己在做什么，否则一般不要使用 inplace 操作。在以上例子中，都是将每一层的输出直接作为下一层的输入，这种网络称为前馈传播网络（Feedforward Neural Network）。对于此类网络，如果每次都写复杂的 forward 函数会有些麻烦，在此就有两种简化方式，ModuleList 和 Sequential。其中 Sequential 是一个特殊的 Module，它包含几个子 module，前向传播时会将输入一层接一层地传递下去。ModuleList 也是一个特殊的 Module，可以包含几个子 module，可以像用 list 一样使用它，但不能直接把输入传给 ModuleList。下面我们举例说明。

```
In: # Sequential的三种写法
    net1 = nn.Sequential()
    net1.add_module('conv', nn.Conv2d(3, 3, 3))
    net1.add_module('batchnorm', nn.BatchNorm2d(3))
    net1.add_module('activation_layer', nn.ReLU())

    net2 = nn.Sequential(
            nn.Conv2d(3, 3, 3),
            nn.BatchNorm2d(3),
            nn.ReLU()
            )
```

```
from collections import OrderedDict
net3= nn.Sequential(OrderedDict([
          ('conv1', nn.Conv2d(3, 3, 3)),
          ('bn1', nn.BatchNorm2d(3)),
          ('relu1', nn.ReLU())
        ]))
print('net1:', net1)
print('net2:', net2)
print('net3:', net3)
```

```
Print: net1: Sequential (
         (conv): Conv2d(3, 3, kernel_size=(3, 3), stride=(1, 1))
         (batchnorm): BatchNorm2d(3, eps=1e-05, momentum=0.1, affine=True)
         (activation_layer): ReLU ()
       )
       net2: Sequential (
         (0): Conv2d(3, 3, kernel_size=(3, 3), stride=(1, 1))
         (1): BatchNorm2d(3, eps=1e-05, momentum=0.1, affine=True)
         (2): ReLU ()
       )
       net3: Sequential (
         (conv1): Conv2d(3, 3, kernel_size=(3, 3), stride=(1, 1))
         (bn1): BatchNorm2d(3, eps=1e-05, momentum=0.1, affine=True)
         (relu1): ReLU ()
       )
```

```
In: # 可根据名字或序号取出子module
    net1.conv, net2[0], net3.conv1
```

```
Out: (Conv2d(3, 3, kernel_size=(3, 3), stride=(1, 1)),
      Conv2d(3, 3, kernel_size=(3, 3), stride=(1, 1)),
      Conv2d(3, 3, kernel_size=(3, 3), stride=(1, 1)))
```

```
In: input = V(t.rand(1, 3, 4, 4))
    output = net1(input)
    output = net2(input)
    output = net3(input)
    output = net3.relu1(net1.batchnorm(net1.conv(input)))
```

```
In: modellist = nn.ModuleList([nn.Linear(3,4), nn.ReLU(), nn.Linear(4,2)])
    input = V(t.randn(1, 3))
    for model in modellist:
        input = model(input)
    # 下面会报错,因为modellist没有实现forward方法
    # output = modelist(input)
```

看到这里，读者可能会问，为何不直接使用 Python 中自带的 list，而非要多此一举呢？这是因为`ModuleList`是`Module`的子类，当在`Module`中使用它时，就能自动识别为子 module。

下面我们举例说明。

```
In: class MyModule(nn.Module):
        def __init__(self):
            super(MyModule, self).__init__()
            self.list = [nn.Linear(3, 4), nn.ReLU()]
            self.module_list = nn.ModuleList([nn.Conv2d(3, 3, 3), nn.ReLU
            ()])
        def forward(self):
            pass
    model = MyModule()
    model
```

```
Out: MyModule (
        (module_list): ModuleList (
          (0): Conv2d(3, 3, kernel_size=(3, 3), stride=(1, 1))
          (1): ReLU ()
        )
      )
```

```
In: for name, param in model.named_parameters():
        print(name, param.size())
```

```
Print: module_list.0.weight torch.Size([3, 3, 3, 3])
       module_list.0.bias torch.Size([3])
```

可见，list 中的子 module 并不能被主 module 识别，而 ModuleList 中的子 module 能够被主 module 识别。这意味着如果用 list 保存子 module，将无法调整其参数，因其未加入到主 module 的参数中。

除 ModuleList 之外还有 ParameterList，它是一个可以包含多个 parameter 的类 list 对象。在实际应用中，使用方式与 ModuleList 类似。在构造函数`__init__`中用到 list、tuple、dict 等对象时，一定要思考是否应该用 ModuleList 或 ParameterList 代替。

4.2.3 循环神经网络层

近些年，随着深度学习和自然语言处理的结合加深，循环神经网络（RNN）的使用也越来越多，关于 RNN 的基础知识，推荐阅读 colah 的文章[①]入门。PyTorch 中实现了如今最常用的三种 RNN：RNN（vanilla RNN）、LSTM 和 GRU。此外还有对应的三种 RNNCell。

RNN 和 RNNCell 层的区别在于前者能够处理整个序列，而后者一次只处理序列中一个时间点的数据，前者封装更完备更易于使用，后者更具灵活性。RNN 层可以通过组合调用 RNNCell 来实现。

```
In: t.manual_seed(1000)
    # 输入: batch_size=3, 序列长度都为2, 序列中每个元素占4维
    input = V(t.randn(2, 3, 4))
    # lstm输入向量4维, 3个隐藏元, 1层
    lstm = nn.LSTM(4, 3, 1)
    # 初始状态: 1层, batch_size=3, 3个隐藏元
    h0 = V(t.randn(1, 3, 3))
    c0 = V(t.randn(1, 3, 3))
    out, hn = lstm(input, (h0, c0))
    out
```

```
Out: Variable containing:
     (0 ,.,.) =
       0.2430 -0.1219 -0.0283
      -0.2495 -0.1840  0.0766
      -0.7958 -0.0006 -0.0898


     (1 ,.,.) =
       0.1831  0.0436 -0.0776
      -0.1026 -0.1055  0.1306
      -0.2092 -0.0362  0.0734
     [torch.FloatTensor of size 2x3x3]
```

[①] http://colah.github.io/posts/2015-08-Understanding-LSTMs/

```
In: t.manual_seed(1000)
    input = V(t.randn(2, 3, 4))
    # 一个LSTMCell对应的层数只能是一层
    lstm = nn.LSTMCell(4, 3)
    hx = V(t.randn(3, 3))
    cx = V(t.randn(3, 3))
    out = []
    for i_ in input:
        hx, cx=lstm(i_, (hx, cx))
        out.append(hx)
    t.stack(out)
```

```
Out: Variable containing:
     (0 ,.,.) =
       0.2430 -0.1219 -0.0283
      -0.2495 -0.1840  0.0766
      -0.7958 -0.0006 -0.0898

     (1 ,.,.) =
       0.1831  0.0436 -0.0776
      -0.1026 -0.1055  0.1306
      -0.2092 -0.0362  0.0734
     [torch.FloatTensor of size 2x3x3]
```

词向量在自然语言中应用十分广泛，PyTorch 同样提供了 Embedding 层。

```
In: # 有4个词，每个词用5维的向量表示
    embedding = nn.Embedding(4, 5)
    # 可以用预训练好的词向量初始化embedding
    embedding.weight.data = t.arange(0,20).view(4,5)
```

```
In: input = V(t.arange(3, 0, -1)).long()
    output = embedding(input)
    output
```

```
Out: Variable containing:
      15  16  17  18  19
      10  11  12  13  14
       5   6   7   8   9
     [torch.FloatTensor of size 3x5]
```

4.2.4 损失函数

在深度学习中要用到各种各样的损失函数（Loss Function），这些损失函数可看作是一种特殊的 layer，PyTorch 也将这些损失函数实现为`nn.Module`的子类。然而在实际使用中通常将这些损失函数专门提取出来，作为独立的一部分。详细的 loss 使用请参照官方文档[①]，这里以分类中最常用的交叉熵损失 CrossEntropyloss 为例讲解。

```
In: # batch_size=3，计算对应每个类别的分数（只有两个类别）
    score = V(t.randn(3, 2))
    # 三个样本分别属于1，0，1类，label必须是LongTensor
    label = V(t.Tensor([1, 0, 1])).long()

    # loss与普通的layer无差异
    criterion = nn.CrossEntropyLoss()
    loss = criterion(score, label)
    loss
```

```
Out: Variable containing:
      1.0721
     [torch.FloatTensor of size 1]
```

4.3 优化器

PyTorch 将深度学习中常用的优化方法全部封装在`torch.optim`中，其设计十分灵活，能够很方便地扩展成自定义的优化方法。

所有的优化方法都是继承基类`optim.Optimizer`，并实现了自己的优化步骤。下面就以最基本的优化方法——随机梯度下降法（SGD）举例说明。这里需要重点掌握：

- 优化方法的基本使用方法。
- 如何对模型的不同部分设置不同的学习率。
- 如何调整学习率。

```
In: # 首先定义一个LeNet网络
    class Net(nn.Module):
        def __init__(self):
```

[①] http://pytorch.org/docs/nn.html#loss-functions

```
        super(Net, self).__init__()
        self.features = nn.Sequential(
                    nn.Conv2d(3, 6, 5),
                    nn.ReLU(),
                    nn.MaxPool2d(2,2),
                    nn.Conv2d(6, 16, 5),
                    nn.ReLU(),
                    nn.MaxPool2d(2,2)
        )
        self.classifier = nn.Sequential(
            nn.Linear(16 * 5 * 5, 120),
            nn.ReLU(),
            nn.Linear(120, 84),
            nn.ReLU(),
            nn.Linear(84, 10)
        )

    def forward(self, x):
        x = self.features(x)
        x = x.view(-1, 16 * 5 * 5)
        x = self.classifier(x)
        return x

net = Net()
```

```
In: from torch import  optim
    optimizer = optim.SGD(params=net.parameters(), lr=1)
    optimizer.zero_grad() # 梯度清零，等价于net.zero_grad()

    input = V(t.randn(1, 3, 32, 32))
    output = net(input)
    output.backward(output) # fake backward

    optimizer.step() # 执行优化
```

```
In: # 为不同子网络设置不同的学习率，在finetune中经常用到
    # 如果对某个参数不指定学习率，就使用默认学习率
    optimizer =optim.SGD([
```

```
                {'params': net.features.parameters()}, # 学习率为1e-5
                {'params': net.classifier.parameters(), 'lr': 1e-2}
            ], lr=1e-5)
```

```
In: # 只为两个全连接层设置较大的学习率，其余层的学习率较小
    special_layers = nn.ModuleList([net.classifier[0], net.classifier[3]])
    special_layers_params = list(map(id, special_layers.parameters()))
    base_params = filter(lambda p: id(p) not in special_layers_params,
                         net.parameters())

    optimizer = torch.optim.SGD([
            {'params': base_params},
            {'params': special_layers.parameters(), 'lr': 0.01}
        ], lr=0.001 )
```

调整学习率主要有两种做法。一种是修改 optimizer.param_groups 中对应的学习率，另一种是新建优化器（更简单也是更推荐的做法），由于 optimizer 十分轻量级，构建开销很小，故可以构建新的 optimizer。但是新建优化器会重新初始化动量等状态信息，这对使用动量的优化器来说（如带 momentum 的 sgd），可能会造成损失函数在收敛过程中出现震荡。

```
In: # 调整学习率，新建一个optimizer
    old_lr = 0.1
    optimizer =optim.SGD([
                {'params': net.features.parameters()},
                {'params': net.classifier.parameters(), 'lr': old_lr
                *0.1}
            ], lr=1e-5)
```

4.4 nn.functional

nn 中还有一个很常用的模块：`nn.functional`。nn 中的大多数 layer 在`functional`中都有一个与之相对应的函数。`nn.functional`中的函数和`nn.Module`的主要区别在于，用 nn.Module 实现的 layers 是一个特殊的类，都是由`class Layer(nn.Module)`定义，会自动提取可学习的参数；而`nn.functional`中的函数更像是纯函数，由`def function(input)`定义。下面举例说明 functional 的使用，并对比二者的不同之处。

```
In: input = V(t.randn(2, 3))
    model = nn.Linear(3, 4)
    output1 = model(input)
    output2 = nn.functional.linear(input, model.weight, model.bias)
    output1 == output2
```

```
Out: Variable containing:
      1  1  1  1
      1  1  1  1
     [torch.ByteTensor of size 2x4]
```

```
In: b = nn.functional.relu(input)
    b2 = nn.ReLU()(input)
    b == b2
```

```
Out: Variable containing:
      1  1  1
      1  1  1
     [torch.ByteTensor of size 2x3]
```

此时读者可能会问，应该什么时候使用 nn.Module，什么时候使用 nn.functional 呢？答案很简单，如果模型有可学习的参数，最好用 nn.Module，否则既可以使用 nn.functional 也可以使用 nn.Module，二者在性能上没有太大差异，具体的使用方式取决于个人喜好。由于激活函数（ReLU、sigmoid、tanh）、池化（MaxPool）等层没有可学习参数，可以使用对应的 functional 函数代替，而卷积、全连接等具有可学习参数的网络建议使用 nn.Module。下面举例说明如何在模型中搭配使用 nn.Module 和 nn.functional。另外，虽然 dropout 操作也没有可学习参数，但建议还是使用`nn.Dropout`而不是`nn.functional.dropout`，因为 dropout 在训练和测试两个阶段的行为有所差别，使用`nn.Module`对象能够通过`model.eval`操作加以区分。

```
In: from torch.nn import functional as F
    class Net(nn.Module):
        def __init__(self):
            super(Net, self).__init__()
            self.conv1 = nn.Conv2d(3, 6, 5)
            self.conv2 = nn.Conv2d(6, 16, 5)
            self.fc1 = nn.Linear(16 * 5 * 5, 120)
            self.fc2 = nn.Linear(120, 84)
```

```
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = F.pool(F.relu(self.conv1(x)), 2)
        x = F.pool(F.relu(self.conv2(x)), 2)
        x = x.view(-1, 16 * 5 * 5)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

不具备可学习参数的层（激活层、池化层等），将它们用函数代替，这样可以不用放置在构造函数`__init__`中。有可学习参数的模块，也可以用 functional 代替，只不过实现起来较烦琐，需要手动定义参数 parameter，如前面实现自定义的全连接层，就可将 weight 和 bias 两个参数单独拿出来，在构造函数中初始化为 parameter。

```
In: class MyLinear(nn.Module):
        def __init__(self):
            super(MyLinear, self).__init__()
            self.weight = nn.Parameter(t.randn(3, 4))
            self.bias = nn.Parameter(t.zeros(3))
        def forward(self):
            return F.linear(input, weight, bias)
```

关于 nn.functional 的设计初衷，以及它和 nn.Module 更多的比较说明，可参看论坛的讨论和作者写的说明[①]。

4.5 初始化策略

在深度学习中参数的初始化十分重要，良好的初始化能让模型更快收敛，并达到更高水平，而糟糕的初始化可能使模型迅速崩溃。PyTorch 中 nn.Module 的模块参数都采取了较合理的初始化策略，因此一般不用我们考虑。当然，我们也可以用自定义初始化代替系统的默认初始化。当我们使用 Parameter 时，自定义初始化尤为重要，因为 t.Tensor() 返回的是内存中的随机数，很可能会有极大值，这在实际训练网络中会造成溢出或者梯度消失。PyTorch 中的`nn.init`模块专门为初始化设计，实现了常用的初始化策略。如果某种初始化策略`nn.init`不提供，用户也可以自己直接初始化。

[①] https://discuss.pytorch.org/search?q=nn.functional

```
In: # 利用nn.init初始化
    from torch.nn import init
    linear = nn.Linear(3, 4)

    t.manual_seed(1)
    # 等价于 linear.weight.data.normal_(0, std)
    init.xavier_normal(linear.weight)
```

```
Out: Parameter containing:
     -1.5885  0.9124 -0.2301
     -1.2198  0.2799  0.0002
     -0.6435  1.8860  0.2370
      0.3126  0.4493  0.2945
     [torch.FloatTensor of size 4x3]
```

```
In: # 直接初始化
    import math
    t.manual_seed(1)

    # xavier初始化的计算公式
    std = math.sqrt(2)/math.sqrt(7.)
    linear.weight.data.normal_(0,std)
```

```
Out: -1.5885  0.9124 -0.2301
     -1.2198  0.2799  0.0002
     -0.6435  1.8860  0.2370
      0.3126  0.4493  0.2945
     [torch.FloatTensor of size 4x3]
```

```
In: # 对模型的所有参数进行初始化
    for name, params in net.named_parameters():
        if name.find('linear') != -1:
            # init linear
            params[0] # weight
            params[1] # bias
        elif name.find('conv') != -1:
            pass
        elif name.find('norm') != -1:
            pass
```

4.6 nn.Module 深入分析

如果想要更深入地理解 nn.Module，研究其原理是很有必要的。首先来看看 nn.Module 基类的构造函数的源代码：

```
def __init__(self):
    self._parameters = OrderedDict()
    self._modules = OrderedDict()
    self._buffers = OrderedDict()
    self._backward_hooks = OrderedDict()
    self._forward_hooks = OrderedDict()
    self.training = True
```

其中每个属性的解释如下。

- `_parameters`：字典。保存用户直接设置的 parameter，`self.param1 = nn.Parameter(t.randn(3, 3))`会被检测到，在字典中加入一个 key 为`param`，value 为对应 parameter 的 item，而 self.submodule = nn.Linear(3, 4) 中的 parameter 则不会存于此。
- `_modules`：子 module。通过`self.submodel = nn.Linear(3, 4)`指定的子 module 会保存于此。
- `_buffers`：缓存。如 batchnorm 使用 momentum 机制，每次前向传播需用到上一次前向传播的结果。
- `_backward_hooks` 与 `_forward_hooks`：钩子技术，用来提取中间变量，类似 variable 的 hook。
- `training`：BatchNorm 与 Dropout 层在训练阶段和测试阶段中采取的策略不同，通过判断 training 值决定前向传播策略。

上述几个属性中，`_parameters`、`_modules`和`_buffers`这三个字典中的键值，都可以通过`self.key`方式获得，效果等价于`self._parameters['key']`。

下面举例说明。

```
In: class Net(nn.Module):
        def __init__(self):
            super(Net, self).__init__()
            # 等价于self.register_parameter('param1' ,nn.Parameter(t.randn
            (3, 3)))
            self.param1 = nn.Parameter(t.rand(3, 3))
```

```
        self.submodel1 = nn.Linear(3, 4)
    def forward(self, input):
        x = self.param1@input
        x = self.submodel11(x)
        return x
net = Net()
net
```

```
Out: Net (
        (submodel1): Linear (3 -> 4)
    )
```

```
In: net._modules
```

```
Out: OrderedDict([('submodel1', Linear (3 -> 4))])
```

```
In: net._parameters
```

```
Out: OrderedDict([('param1', Parameter containing:
                    0.1863  0.3879  0.3456
                    0.6697  0.3968  0.9355
                    0.5388  0.8463  0.4192
                   [torch.FloatTensor of size 3x3])])
```

```
In: net.param1 # 等价于net._parameters['param1']
```

```
Out: Parameter containing:
     0.1863  0.3879  0.3456
     0.6697  0.3968  0.9355
     0.5388  0.8463  0.4192
    [torch.FloatTensor of size 3x3]
```

```
In: for name, param in net.named_parameters():
        print(name, param.size())
```

```
Print: param1 torch.Size([3, 3])
       submodel1.weight torch.Size([4, 3])
       submodel1.bias torch.Size([4])
```

```
In: for name, submodel in net. named_modules():
        print(name, submodel)
```

```
Print:  Net (
          (submodel1): Linear (3 -> 4)
        )
       submodel1 Linear (3 -> 4)
```

```
In: bn = nn.BatchNorm1d(2)
    input = V(t.rand(3, 2), requires_grad=True)
    output = bn(input)
    bn._buffers
```

```
Out: OrderedDict([('running_mean',
                    1.00000e-02 *
                      7.6571
                      6.9414
                    [torch.FloatTensor of size 2]), ('running_var',
                     0.9002
                     0.9116
                    [torch.FloatTensor of size 2])])
```

nn.Module 在实际使用中可能层层嵌套，一个 module 包含若干个子 module，每一个子 module 又包含了更多的子 module。为方便用户访问各个子 module，nn.Module 实现了很多方法，如函数`children`可以查看直接子 module，函数`modules`可以查看所有的子 module（包括当前 module）。与之相对应的还有函数`named_childen`和`named_modules`，其能够在返回 module 列表的同时返回它们的名字。

```
In: input = V(t.arange(0, 12).view(3, 4))
    model = nn.Dropout()
    # 在训练阶段，会有一半左右的数被随机置为0
    model(input)
```

```
Out: Variable containing:
       0   0   0   0
       0   0   0  14
       0  18  20  22
     [torch.FloatTensor of size 3x4]
```

```
In: model.training  = False
    # 在测试阶段，dropout什么都不做
    model(input)
```

```
Out: Variable containing:
       0   1   2   3
       4   5   6   7
       8   9  10  11
     [torch.FloatTensor of size 3x4]
```

对 batchnorm、dropout、instancenorm 等在训练和测试阶段行为差距较大的层，如果在测试时不将其 training 值设为 False，则可能会有很大影响，这在实际使用中要千万注意。虽然可通过直接设置`training`属性将子 module 设为 train 和 eval 模式，但这种方式较烦琐，因为如果一个模型具有多个 dropout 层，就需要为每个 dropout 层指定 training 属性。笔者推荐的做法是调用`model.train()`函数，它会将当前 module 及其子 module 中的所有 training 属性都设为 True。相应地，`model.eval()`函数会把 training 属性都设为 False。

```
In: print(net.training, net.submodel1.training)
    net.eval()
    net.training, net.submodel1.training
```

```
Print: True True
```

```
Out: (False, False)
```

```
In: list(net.named_modules())
```

```
Out: [('', Net (
         (submodel1): Linear (3 -> 4)
       )), ('submodel1', Linear (3 -> 4))]
```

`register_forward_hook` 和 `register_backward_hook` 函数的功能类似于 variable 的 `register_hook`，可在 module 前向传播或反向传播时注册钩子。每次前向传播执行结束后会执行钩子函数（hook）。前向传播的钩子函数具有如下形式：`hook(module, input, output) -> None`，而反向传播则具有如下形式：`hook(module, grad_input, grad_output) -> Tensor or None`。钩子函数不应修改输入和输出，并且在使用后应及时删除，以避免每次都运行钩子增加运行负载。钩子函数主要用在获取某些中间结果的情景，如中间某一层的输出或某一层的梯度。这些结果本应写在 forward 函数中，但

如果在 forward 函数中加上这些处理，可能会使处理逻辑比较复杂，这时使用钩子技术就更合适。下面考虑一种场景：有一个预训练的模型，需要提取模型的某一层（不是最后一层）的输出作为特征进行分类，希望不修改其原有的模型定义文件，这时就可以利用钩子函数。下面给出实现的伪代码。

```
model = VGG()
features = t.Tensor()
def hook(module, input, output):
    '''把这层的输出复制到features中'''
    features.copy_(output.data)

handle = model.layer8.register_forward_hook(hook)
_ = model(input)
# 用完hook后删除
handle.remove()
```

nn.Module对象在构造函数中的行为看起来有些怪异，想要真正掌握其原理，就需要看两个魔法方法 __getattr__ 和 __setattr__。在 Python 中有两个常用的 builtin 方法：getattr和setattr。getattr(obj, 'attr1')等价于obj.attr，如果getattr函数无法找到所需属性，Python 会调用obj.__getattr__('attr1')方法，即getattr函数无法找到的交给 __getattr__ 函数处理；如果这个对象没有实现 __getattr__ 方法，程序就会抛出异常 AttributeError。setattr(obj, 'name', value)等价于obj.name=value，如果 obj 对象实现了 __setattr__ 方法，setattr 会直接调用obj.__setattr__('name', value)，否则调用 builtin 方法。总结如下：

- result = obj.name 会调用 builtin 函数getattr(obj, 'name')，如果该属性找不到，会调用obj.__getattr__('name')。
- obj.name = value 会调用 builtin 函数setattr(obj, 'name', value)，如果 obj 对象实现了 __setattr__ 方法，setattr会直接调用obj.__setattr__('name', value)。

nn.Module 实现了自定义的 __setattr__ 函数，当执行module.name=value时，会在 __setattr__ 中判断 value 是否为Parameter或nn.Module对象，如果是则将这些对象加到 _parameters 和 _modules 两个字典中；如果是其他类型的对象，如Variable、list、dict等，则调用默认的操作，将这个值保存在 __dict__ 中。

```
In: module = nn.Module()
    module.param = nn.Parameter(t.ones(2, 2))
    module._parameters
```

```
Out: OrderedDict([('param', Parameter containing:
                    1  1
                    1  1
                   [torch.FloatTensor of size 2x2])])
```

```
In: submodule1 = nn.Linear(2, 2)
    submodule2 = nn.Linear(2, 2)
    module_list =  [submodule1, submodule2]
    # 对于list对象，调用builtin函数，保存在__dict__中
    module.submodules = module_list
    print('_modules: ', module._modules)
    print("__dict__['submodules']:",module.__dict__.get('submodules'))
```

```
Print: _modules:  OrderedDict()
       __dict__['submodules']: [Linear (2 -> 2), Linear (2 -> 2)]
```

```
In: module_list = nn.ModuleList(module_list)
    module.submodules = module_list
    print('ModuleList is instance of nn.Module: ', isinstance(module_list,
     nn.Module))
    print('_modules: ', module._modules)
    print("__dict__['submodules']:", module.__dict__.get('submodules'))
```

```
Print: ModuleList is instance of nn.Module:  True
       _modules:  OrderedDict([('submodules', ModuleList (
         (0): Linear (2 -> 2)
         (1): Linear (2 -> 2)
       ))])
       __dict__['submodules']: None
```

因 _modules 和 _parameters 中的 item 未保存在 __dict__ 中，所以默认的 getattr 方法无法获取它，因而nn.Module实现了自定义的 __getattr__ 方法。如果默认的getattr无法处理，就调用自定义的 __getattr__ 方法，尝试从 _modules 、_parameters 和 _buffers 这三个字典中获取。

```
In: getattr(module, 'training') # 等价于module.training
    # error
    # module.__getattr__('training')
```

```
Out: True
```

```
In: module.attr1 = 2
    getattr(module, 'attr1')
    # 报错
    # module.__getattr__('attr1')
```

```
Out: 2
```

```
In: # 即module.param，会调用module.__getattr__('param')
    getattr(module, 'param')
```

```
Out: Parameter containing:
      1  1
      1  1
     [torch.FloatTensor of size 2x2]
```

在 PyTorch 中保存模型十分简单，所有的 Module 对象都具有 state_dict() 函数，返回当前 Module 所有的状态数据。将这些状态数据保存后，下次使用模型时即可利用`model.load_state_dict()`函数将状态加载进来。优化器（optimizer）也有类似的机制，不过一般并不需要保存优化器的运行状态。

```
In: # 保存模型
    t.save(net.state_dict(), 'net.pth')

    # 加载已保存的模型
    net2 = Net()
    net2.load_state_dict(t.load('net.pth'))
```

还有另外一种保存模型的方法，但因其严重依赖模型定义方式及文件路径结构等，所以很容易出问题，因而不建议使用。

```
In: t.save(net, 'net_all.pth')
    net2 = t.load('net_all.pth')
    net2
```

```
Print: /home/cy/git/pytorch/pytorch/torch/serialization.py:147:
UserWarning: Couldn't retrieve source code for container of type Net. It
won't be checked for correctness upon loading.
        "type " + obj.__name__ + ". It won't be checked "
```

```
Out: Net (
      (submodel1): Linear (3 -> 4)
    )
```

将 Module 放在 GPU 上运行也十分简单，只需以下两步。

- model = model.cuda()：将模型的所有参数转存到 GPU。
- input.cuda()：将输入数据放置到 GPU 上。

至于如何在多个 GPU 上并行计算，PyTorch 也提供了两个函数，可实现简单高效的并行 GPU 计算。

- nn.parallel.data_parallel(module, inputs, device_ids=None, output_device=None, dim=0, module_kwargs=None)
- class torch.nn.DataParallel(module, device_ids=None, output_device=None, dim=0)

可见二者的参数十分相似，通过`device_ids`参数可以指定在哪些 GPU 上进行优化，output_device 指定输出到哪个 GPU 上。唯一的不同在于前者直接利用多 GPU 并行计算得出结果，后者则返回一个新的 module，能够自动在多 GPU 上进行并行加速。

```
# method 1
new_net = nn.DataParallel(net, device_ids=[0, 1])
output = new_net(input)

# method 2
output = nn.parallel.data_parallel(net, input, device_ids=[0, 1])
```

DataParallel 并行的方式，是将输入一个 batch 的数据均分成多份，分别送到对应的 GPU 进行计算，然后将各个 GPU 得到的梯度相加。与 Module 相关的所有数据也会以浅复制的方式复制多份。

4.7 nn 和 autograd 的关系

nn.Module 利用的是 autograd 技术，其主要工作是实现前向传播。在 forward 函数中，nn.Module 对输入的 Variable 进行的各种操作，本质上都用到了 autograd 技术。这里需要对比 autograd.Function 和 nn.Module 之间的区别。

- autograd.Function 利用 Tensor 对 autograd 技术的扩展，为 autograd 实现了新的运算 op，不仅要实现前向传播还要手动实现反向传播。

- nn.Module 利用了 autograd 技术，对 nn 的功能进行扩展，实现了深度学习中更多的层。只需实现前向传播功能，autograd 即会自动实现反向传播。
- nn.functional 是一些 autograd 操作的集合，是经过封装的函数。

作为两种扩充 PyTorch 接口的方法，我们在实际使用中应该如何选择呢？如果某一个操作在 autograd 中尚未支持，那么需要利用 Function 手动实现对应的前向传播和反向传播。如果某些时候利用 autograd 接口比较复杂，则可以利用 Function 将多个操作聚合，实现优化，正如第 3 章实现的`Sigmoid`一样，比直接利用 autograd 低级别的操作要快。如果只是想在深度学习中增加某一层，使用 nn.Module 进行封装则更简单高效。

4.8 小试牛刀：用 50 行代码搭建 ResNet

Kaiming He 的深度残差网络（ResNet）[①]在深度学习的发展中起到了很重要的作用，ResNet 不仅一举拿下了 2015 年多个计算机视觉比赛项目的冠军，更重要的是这一结构解决了训练极深网络时的梯度消失问题。

这里选取 ResNet34 讲解 ResNet 的网络结构。ResNet34 的网络结构如图 4-5 所示，除了最开始的卷积池化和最后的池化全连接之外，网络中有很多结构相似的单元，这些重复单元的共同点就是有个跨层直连的 shortcut。ResNet 中将一个跨层直连的单元称为 Residual block，其结构如图 4-6 所示，左边部分是普通的卷积网络结构，右边是直连，如果输入和输出的通道数不一致，或其步长不为 1，就需要有一个专门的单元将二者转成一致的，使其可以相加。

另外，可以发现 Residual block 的大小也是有规律的，在最开始的 pool 之后有连续的几个一模一样的 Residual block 单元，这些单元的通道数一样，在这里我们将这几个拥有多个 Residual block 单元的结构称之为 layer，注意要和之前讲的 layer 区分开，这里的 layer 是几个层的集合。

考虑到 Residual block 和 layer 出现了多次，我们可以把它们实现为一个子 Module 或函数。这里我们将 Residual block 实现为一个子 module，而将 layer 实现为一个函数。下面是实现代码，规律总结如下：

- 对模型中的重复部分，实现为子 module 或用函数生成相应的 module。
- nn.Module 和 nn.Functional 结合使用。
- 尽量使用`nn.Seqential`。

[①] He K, Zhang X, Ren S, et al. Deep residual learning for image recognition[C]//Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. 2016: 770-778.

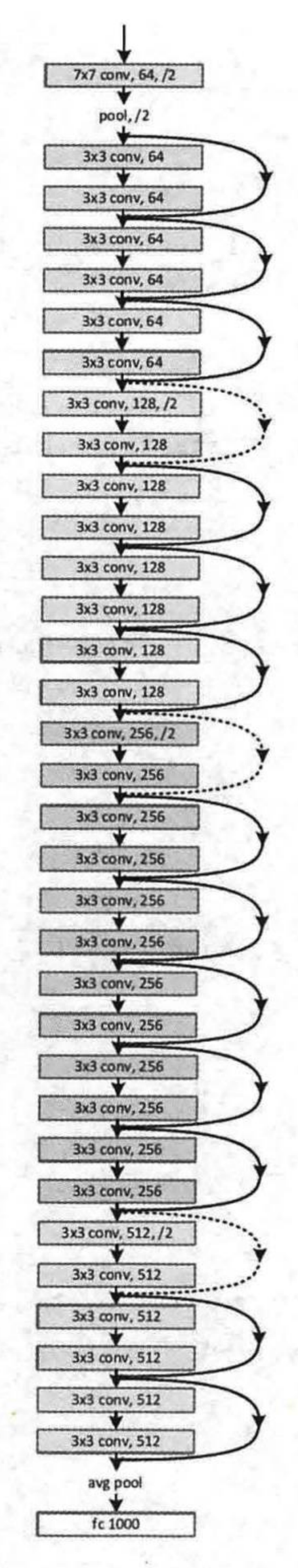


图 4-5　ResNet34 的网络结构

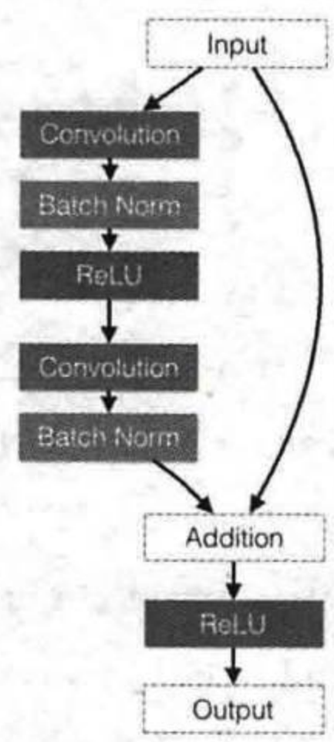


图 4-6　Residual block 结构图

```
In: from torch import  nn
    import torch as t
    from torch.nn import  functional as F
```

```
In: class ResidualBlock(nn.Module):
        '''
        实现子module: Residual Block
        '''
        def __init__(self, inchannel, outchannel, stride=1, shortcut=None)
        :
            super(ResidualBlock, self).__init__()
            self.left = nn.Sequential(
                    nn.Conv2d(inchannel,outchannel,3,stride, 1,bias=False)
                    ,
                    nn.BatchNorm2d(outchannel),
                    nn.ReLU(inplace=True),
                    nn.Conv2d(outchannel,outchannel,3,1,1,bias=False),
                    nn.BatchNorm2d(outchannel) )
        self.right = shortcut

    def forward(self, x):
        out = self.left(x)
        residual = x if self.right is None else self.right(x)
        out += residual
        return F.relu(out)

class ResNet(nn.Module):
    '''
    实现主module: ResNet34
    ResNet34包含多个layer，每个layer又包含多个residual block
    用子module实现residual block，用_make_layer函数实现layer
    '''
    def __init__(self, num_classes=1000):
        super(ResNet, self).__init__()
        # 前几层图像转换
        self.pre = nn.Sequential(
                nn.Conv2d(3, 64, 7, 2, 3, bias=False),
                nn.BatchNorm2d(64),
                nn.ReLU(inplace=True),
                nn.MaxPool2d(3, 2, 1))
```

```
        # 重复的layer，分别有3，4，6，3个residual block
        self.layer1 = self._make_layer( 64, 128, 3)
        self.layer2 = self._make_layer( 128, 256, 4, stride=2)
        self.layer3 = self._make_layer( 256, 512, 6, stride=2)
        self.layer4 = self._make_layer( 512, 512, 3, stride=2)

        # 分类用的全连接
        self.fc = nn.Linear(512, num_classes)

    def _make_layer(self,  inchannel, outchannel, block_num, stride=1)
    :
        '''
        构建layer,包含多个residual block
        '''
        shortcut = nn.Sequential(
                nn.Conv2d(inchannel,outchannel,1,stride, bias=False),
                nn.BatchNorm2d(outchannel))

        layers = []
        layers.append(ResidualBlock(inchannel, outchannel, stride,
        shortcut))

        for i in range(1, block_num):
            layers.append(ResidualBlock(outchannel, outchannel))
        return nn.Sequential(*layers)

    def forward(self, x):
        x = self.pre(x)

        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        x = self.layer4(x)

        x = F.avg_pool2d(x, 7)
        x = x.view(x.size(0), -1)
        return self.fc(x)
```

```
In: model = ResNet()
    input  = t.autograd.Variable(t.randn(1, 3, 224, 224))
    o = model(input)
```

感兴趣的读者可以尝试实现 Google 的 Inception 网络结构或 ResNet 的其他变体，看看如何能够简洁明了地实现它，实现代码尽量控制在 80 行以内（本例去掉空行和注释总共不超过 50 行）。另外，与 PyTorch 配套的图像工具包`torchvision`已经实现了深度学习中大多数经典的模型，其中就包括 ResNet34，读者可以通过下面两行代码使用：

```
from torchvision import models
model = models.resnet34()
```

本例中 ResNet34 的实现参考了 torchvision 中的实现并做了简化，读者可以阅读相应的源码，比较这里的实现和 torchvision 中的不同。

通过本章的学习，读者可以掌握 PyTorch 中神经网络工具箱中大部分类和函数的用法。关于这部分的更多内容，读者可以参阅官方文档，文档中有更多详细的说明。

5 PyTorch 中常用的工具

在训练神经网络的过程中需要用到很多工具，其中最重要的三部分是数据、可视化和 GPU 加速。本章主要介绍 PyTorch 在这几方面常用的工具，合理使用这些工具能极大地提高编码效率。

5.1 数据处理

在解决深度学习问题的过程中，往往需要花费大量的精力去处理数据，包括图像、文本、语音或其他二进制数据等。数据的处理对训练神经网络来说十分重要，良好的数据处理不仅会加速模型训练，也会提高模型效果。考虑到这一点，PyTorch 提供了几个高效便捷的工具，以便使用者进行数据处理或增强等操作，同时可通过并行化加速数据加载。

数据加载

在 PyTorch 中，数据加载可通过自定义的数据集对象实现。数据集对象被抽象为`Dataset`类，实现自定义的数据集需要继承 Dataset，并实现两个 Python 魔法方法。

- `__getitem__`：返回一条数据或一个样本。`obj[index]`等价于`obj.__getitem__(index)`。
- `__len__`：返回样本的数量。`len(obj)`等价于`obj.__len__()`。

这里我们以 Kaggle 经典挑战赛“Dogs vs. Cat”的数据为例，详细讲解如何处理数据。“Dogs vs. Cats”是一个分类问题，判断一张图片是狗还是猫，其所有图片都存放在一个文件夹下，根据文件名的前缀判断是狗还是猫。

```
In: %env LS_COLORS = None
    !tree --charset ascii  data/dogcat/
```

```
Print: env: LS_COLORS=None
       data/dogcat/
       |-- cat.12484.jpg
       |-- cat.12485.jpg
       |-- cat.12486.jpg
       |-- cat.12487.jpg
       |-- dog.12496.jpg
       |-- dog.12497.jpg
       |-- dog.12498.jpg
       `-- dog.12499.jpg

       0 directories, 8 files
```

```
In: import torch as t
    from torch.utils import data
```

```
In: import os
    from PIL import  Image
    import numpy as np

    class DogCat(data.Dataset):
        def __init__(self, root):
            imgs = os.listdir(root)
            # 所有图片的绝对路径
            # 这里不实际加载图片，只是指定路径
            # 当调用__getitem__时才会真正读图片
            self.imgs = [os.path.join(root, img) for img in imgs]

        def __getitem__(self, index):
            img_path = self.imgs[index]
            # dog->1,  cat->0
            label = 1 if 'dog' in img_path.split('/')[-1] else 0
```

```
        pil_img = Image.open(img_path)
        array = np.asarray(pil_img)
        data = t.from_numpy(array)
        return data, label

    def __len__(self):
        return len(self.imgs)
```

```
In: dataset = DogCat('./data/dogcat/')
    img, label = dataset[0] # 相当于调用dataset.__getitem__(0)
    for img, label in dataset:
        print(img.size(), img.float().mean(), label)
```

```
Print: (torch.Size([236, 289, 3]), 130.30038805153168, 0)
       (torch.Size([374, 499, 3]), 115.51768778198108, 0)
       (torch.Size([377, 499, 3]), 151.7174171508357, 1)
       (torch.Size([375, 499, 3]), 116.81384992206635, 1)
       (torch.Size([375, 499, 3]), 150.50795635715878, 1)
       (torch.Size([400, 300, 3]), 128.154975, 1)
       (torch.Size([499, 379, 3]), 171.80847887507645, 0)
       (torch.Size([500, 497, 3]), 106.4915063715627, 0)
```

通过上面的代码，我们学习了如何自定义自己的数据集，并可以依次获取。但这里返回的数据不适合实际使用，因其具有如下两方面问题：

- 返回样本的形状不一，每张图片的大小不一样，这对于需要取 batch 训练的神经网络来说很不友好。
- 返回样本的数值较大，未归一化至 [-1, 1]。

针对上述问题，PyTorch 提供了 torchvision[①]。它是一个视觉工具包，提供了很多视觉图像处理的工具，其中`transforms`模块提供了对 PIL `Image`对象和`Tensor`对象的常用操作。

对 PIL Image 的常见操作如下。

- `Resize`：调整图片尺寸。
- `CenterCrop`、`RandomCrop`、`RandomSizedCrop`：裁剪图片。
- `Pad`：填充。

[①] https://github.com/pytorch/vision/

- `ToTensor`：将 PIL Image 对象转成 Tensor，会自动将 [0, 255] 归一化至 [0, 1]。

对 Tensor 的常见操作如下。

- Normalize：标准化，即减均值，除以标准差。
- ToPILImage：将 Tensor 转为 PIL Image 对象。

如果要对图片进行多个操作，可通过`Compose`将这些操作拼接起来，类似于 `nn.Sequential`。注意，这些操作定义后是以对象的形式存在，真正使用时需调用它的`__call__`方法，类似于`nn.Module`。例如，要将图片调整为 224 × 224，首先应构建操作`trans = Scale((224, 224))`，然后调用`trans(img)`。下面我们就用 transforms 的这些操作来优化上面实现的 dataset。

```
In: import os
    from PIL import  Image
    import numpy as np
    from torchvision import transforms as T

    transform = T.Compose([
        T.Resize(224), # 缩放图片(Image)，保持长宽比不变，最短边为224像素
        T.CenterCrop(224), # 从图片中间切出224*224的图片
        T.ToTensor(), # 将图片(Image)转成Tensor，归一化至[0, 1]
        T.Normalize(mean=[.5, .5, .5], std=[.5, .5, .5]) # 标准化至[-1, 1]
])

    class DogCat(data.Dataset):
        def __init__(self, root, transforms=None):
            imgs = os.listdir(root)
            self.imgs = [os.path.join(root, img) for img in imgs]
            self.transforms=transforms

        def __getitem__(self, index):
            img_path = self.imgs[index]
            label = 0 if 'dog' in img_path.split('/')[-1] else 1
            data = Image.open(img_path)
            if self.transforms:
                data = self.transforms(data)
            return data, label
```

```
    def __len__(self):
        return len(self.imgs)

dataset = DogCat('./data/dogcat/', transforms=transform)
img, label = dataset[0]
for img, label in dataset:
    print(img.size(), label)
```

```
Print: (torch.Size([3, 224, 224]), 1)
       (torch.Size([3, 224, 224]), 1)
       (torch.Size([3, 224, 224]), 0)
       (torch.Size([3, 224, 224]), 0)
       (torch.Size([3, 224, 224]), 0)
       (torch.Size([3, 224, 224]), 0)
       (torch.Size([3, 224, 224]), 1)
       (torch.Size([3, 224, 224]), 1)
```

除了上述操作之外，transforms 还可以通过`Lambda`封装自定义的转换策略。例如，想对 PIL Image 进行随机旋转，则可写成`trans=T.Lambda(lambda img: img.rotate(random()*360))`。

torchvision 已经预先实现了常用的 Dataset，包括前面使用过的 CIFAR-10，以及 ImageNet、COCO、MNIST、LSUN 等数据集，可通过调用`torchvision.datasets`下相应对象来调用相关数据集，具体使用方法请参看官方文档[①]。本节介绍一个读者会经常使用到的 Dataset——`ImageFolder`，它的实现和上述`DogCat`很相似。`ImageFolder`假设所有的文件按文件夹保存，每个文件夹下存储同一个类别的图片，文件夹名为类名，其构造函数如下：

```
ImageFolder(root, transform=None, target_transform=None, loader=
default_loader)
```

它主要有以下四个参数。

- `root`：在 root 指定的路径下寻找图片。
- `transform`：对 PIL Image 进行转换操作，transform 的输入是使用 loader 读取图片的返回对象。
- `target_transform`：对 label 的转换。

[①] http://pytorch.org/docs/master/torchvision/datasets.html

- `loader`：指定加载图片的函数，默认操作是读取为 PIL Image 对象。

label 是按照文件夹名顺序排序后存成字典的，即 {类名: 类序号 (从 0 开始)}，一般来说最好直接将文件夹命名为从 0 开始的数字，这样会和 ImageFolder 实际的 label 一致，如果不是这种命名规范，建议通过`self.class_to_idx`属性了解 label 和文件夹名的映射关系。

```
In: !tree --charset ASCII  data/dogcat_2/
```

```
Print: data/dogcat_2/
       |-- cat
       |   |-- cat.12484.jpg
       |   |-- cat.12485.jpg
       |   |-- cat.12486.jpg
       |   `-- cat.12487.jpg
       `-- dog
           |-- dog.12496.jpg
           |-- dog.12497.jpg
           |-- dog.12498.jpg
           `-- dog.12499.jpg

       2 directories, 8 files
```

```
In: from torchvision.datasets import ImageFolder
    dataset = ImageFolder('data/dogcat_2/')
```

```
In: # cat文件夹的图片对应label 0, dog对应1
    dataset.class_to_idx
```

```
Out: {'cat': 0, 'dog': 1}
```

```
In: # 所有图片的路径和对应的label
    dataset.imgs
```

```
Out: [('data/dogcat_2/cat/cat.12484.jpg', 0),
      ('data/dogcat_2/cat/cat.12485.jpg', 0),
      ('data/dogcat_2/cat/cat.12486.jpg', 0),
      ('data/dogcat_2/cat/cat.12487.jpg', 0),
      ('data/dogcat_2/dog/dog.12496.jpg', 1),
      ('data/dogcat_2/dog/dog.12497.jpg', 1),
```

```
    ('data/dogcat_2/dog/dog.12498.jpg', 1),
    ('data/dogcat_2/dog/dog.12499.jpg', 1)]
```

```
In: # 没有任何的transform，所以返回的还是PIL Image对象
    dataset[0][1] # 第一维是第几张图，第二维为1返回label
    dataset[0][0] # 为0返回图片数据，返回的Image对象如图5-1所示
```

图 5-1　程序输出的图片 1

```
In: # 加上transform
    normalize = T.Normalize(mean=[0.4, 0.4, 0.4], std=[0.2, 0.2, 0.2])
    transform  = T.Compose([
            T.RandomSizedCrop(224),
            T.RandomHorizontalFlip(),
            T.ToTensor(),
            normalize,
    ])
```

```
In: dataset = ImageFolder('data/dogcat_2/', transform=transform)
```

```
In: # 深度学习中图片数据一般保存成CxHxW，即通道数x图片高x图片宽
    dataset[0][0].size()
```

```
Out: torch.Size([3, 224, 224])
```

```
In: to_img = T.ToPILImage()
    # 0.2和0.4是标准差和均值的近似
    to_img(dataset[0][0]*0.2+0.4) # 程序输出的图片如图5-2所示
```

图 5-2　程序输出的图片 2

Dataset 只负责数据的抽象，一次调用 __getitem__ 只返回一个样本。前面提到过，在训练神经网络时，是对一个 batch 的数据进行操作，同时还需要对数据进行 shuffle 和并行加速等。对此，PyTorch 提供了 DataLoader 帮助我们实现这些功能。

DataLoader 的函数定义如下。

```
DataLoader(dataset, batch_size=1, shuffle=False, sampler=None, num_workers
=0, collate_fn=default_collate, pin_memory=False, drop_last=False)
```

- dataset：加载的数据集（Dataset 对象）。
- batch_size：batch size（批大小）。
- shuffle:：是否将数据打乱。
- sampler：样本抽样，后续会详细介绍。
- num_workers：使用多进程加载的进程数，0 代表不使用多进程。
- collate_fn：如何将多个样本数据拼接成一个 batch，一般使用默认的拼接方式即可。
- pin_memory：是否将数据保存在 pin memory 区，pin memory 中的数据转到 GPU 会快一些。
- drop_last：dataset 中的数据个数可能不是 batch_size 的整数倍，drop_last 为 True 会将多出来不足一个 batch 的数据丢弃。

```
In: from torch.utils.data import DataLoader
```

```
In: dataloader = DataLoader(dataset, batch_size=3, shuffle=True,
num_workers=0, drop_last=False)
```

```
In: dataiter = iter(dataloader)
    imgs, labels = next(dataiter)
    imgs.size() # batch_size, channel, height, weight
```

```
Out: torch.Size([3, 3, 224, 224])
```

dataloader 是一个可迭代的对象，我们可以像使用迭代器一样使用它，例如：

```
for batch_datas, batch_labels in dataloader:
    train()
```

或

```
dataiter = iter(dataloader)
batch_datas, batch_labesl = next(dataiter)
```

在数据处理中，有时会出现某个样本无法读取等问题，例如某张图片损坏。这时在 `__getitem__` 函数中将出现异常，此时最好的解决方案即是将出错的样本剔除。如果遇到这种情况实在无法处理，则可以返回 None 对象，然后在 `Dataloader` 中实现自定义的 `collate_fn`，将空对象过滤掉。但要注意，在这种情况下 dataloader 返回的一个 batch 的样本数目会少于 batch_size。

```
In: class NewDogCat(DogCat): # 继承前面实现的DogCat数据集
        def __getitem__(self, index):
            try:
                # 调用父类的获取函数，即 DogCat.__getitem__(self, index)
                return super(NewDogCat,self).__getitem__(index)
            except:
                return None, None

    from torch.utils.data.dataloader import default_collate # 导入默认的拼
    接方式
    def my_collate_fn(batch):
        '''
        batch中每个元素形如(data, label)
        '''
        # 过滤为None的数据
        batch = list(filter(lambda x:x[0] is not None, batch))
        return default_collate(batch) # 用默认方式拼接过滤后的batch数据
```

```
In: dataset = NewDogCat('data/dogcat_wrong/', transforms=transform)
```

```
In: dataset[5]
```

```
Out: (None, None)
```

```
In: dataloader = DataLoader(dataset, 2, collate_fn=my_collate_fn,
num_workers=1)
    for batch_datas, batch_labels in dataloader:
        print(batch_datas.size(),batch_labels.size())
```

```
Print: (torch.Size([2, 3, 224, 224]), torch.Size([2]))
       (torch.Size([2, 3, 224, 224]), torch.Size([2]))
       (torch.Size([1, 3, 224, 224]), torch.Size([1]))
       (torch.Size([2, 3, 224, 224]), torch.Size([2]))
       (torch.Size([1, 3, 224, 224]), torch.Size([1]))
```

我们来看上述 batch_size 的大小。其中第 3 个 batch 的 batch_size 为 1，这是因为有一张图片损坏，导致其无法正常返回。而最后一个 batch 的 batch_size 也为 1，这是因为共有 9 张（包括损坏的文件）图片，无法整除 2（batch_size），因此最后一个 batch 的数据会少于 batch_szie，可通过指定`drop_last=True`丢弃最后一个样本数目不足 batch_size 的 batch。

对样本损坏或数据集加载异常等情况，还可以通过其他方式解决。例如遇到异常情况，就随机取一张图片代替：

```
class NewDogCat(DogCat):
    def __getitem__(self, index):
        try:
            return super(NewDogCat, self).__getitem__(index)
        except:
            new_index = random.randint(0, len(self)-1)
            return self[new_index]
```

相比较丢弃异常图片而言，这种做法会更好一些，因为它能保证每个 batch 样本的数目仍是 batch_size。但在大多数情况下，最好的方式还是对数据进行彻底清洗。

DataLoader 里并没有太多的魔法方法，它封装了 Python 的标准库`multiprocessing`，使其能够实现多进程加速。在 Dataset 和 DataLoader 的使用方面有以下建议。

（1）高负载的操作放在`__getitem__`中，如加载图片等。

（2）dataset 中应尽量只包含只读对象，避免修改任何可变对象。

第一点是因为多进程会并行地调用 `__getitem__` 函数，将负载高的放在 `__getitem__` 函数中能够实现并行加速。第二点是因为 dataloader 使用多进程加载，如果在`Dataset`中使用了可变对象，可能会有意想不到的冲突。在多线程/多进程中，修改一个可变对象需要加锁，但是 dataloader 的设计使得其很难加锁（在实际使用中也应尽量避免锁的存在），因此最好避免在 dataset 中修改可变对象。下面就是一个不好的例子，在多进程处理中`self.num`可能与预期不符，这种问题不会报错，因此难以发现。如果一定要修改可变对象，建议使用 Python 标准库`Queue`中的相关数据结构。

```
class BadDataset(Dataset):
    def __init__(self):
        self.datas = range(100)
        self.num = 0 # 取数据的次数
    def __getitem__(self, index):
        self.num += 1
        return self.datas[index]
```

使用 Python `multiprocessing`库的另一个问题是，在使用多进程时，如果主程序异常终止（比如用“Ctrl+C”快捷键强行退出），相应的数据加载进程可能无法正常退出。这时你可能会发现程序已经退出了，但 GPU 显存和内存依旧被占用着，通过`top`、`ps aux`依旧能够看到已经退出的程序，这时就需要手动强行终止进程。建议使用如下命令：

```
ps x | grep <cmdline> | awk '{print $1}' | xargs kill
```

- `ps x`：获取当前用户的所有进程。
- `grep <cmdline>`：找到已经停止的 PyTorch 程序的进程，例如你是通过 python train.py 启动的，那就需要写`grep 'python train.py'`。
- `awk '{print $1}'`：获取进程的 pid。
- `xargs kill`：终止进程，根据需要可能要写成`xargs kill -9`强制终止进程。

在执行这句命令之前，建议先打印确认进程。

```
ps x | grep <cmdline>
```

PyTorch 中还单独提供了一个`sampler`模块，用来对数据进行采样。常用的有随机采样器`RandomSampler`，当 dataloader 的`shuffle`参数为 True 时，系统会自动调用这个采样器，实现打乱数据。默认的采样器是`SequentialSampler`，它会按顺序一个一个进行

采样。这里介绍另外一个很有用的采样方法：`WeightedRandomSampler`，它会根据每个样本的权重选取数据，在样本比例不均衡的问题中，可用它进行重采样。

构建 `WeightedRandomSampler` 时需提供两个参数：每个样本的权重`weights`、共选取的样本总数`num_samples`，以及一个可选参数`replacement`。权重越大的样本被选中的概率越大，待选取的样本数目一般小于全部的样本数目。`replacement`用于指定是否可以重复选取某一个样本，默认为 True，即允许在一个 epoch 中重复采样某一个数据。如果设为 False，则当某一类样本被全部选取完，但其样本数目仍未达到 num_samples 时，sampler 将不会再从该类中选择数据，此时可能导致`weights`参数失效。下面举例说明。

```
In: dataset = DogCat('data/dogcat/', transforms=transform)

    # 狗的图片被取出的概率是猫的概率的两倍
    # 两类图片被取出的概率与weights的绝对大小无关，只和比值有关
    weights = [2 if label == 1 else 1 for data, label in dataset]
    weights
```

```
Out: [2, 2, 1, 1, 1, 1, 2, 2]
```

```
In: from torch.utils.data.sampler import  WeightedRandomSampler
    sampler = WeightedRandomSampler(weights,\
                                    num_samples=9,\
                                    replacement=True)
    dataloader = DataLoader(dataset,
                            batch_size=3,
                            sampler=sampler)
    for datas, labels in dataloader:
        print(labels.tolist())
```

```
Print: [1, 1, 1]
       [0, 1, 1]
       [1, 0, 1]
```

可见猫狗样本比例约为 1:2，另外一共只有 8 个样本，却返回了 9 个，说明有样本被重复返回的，这就是 replacement 参数的作用，下面我们将 replacement 设为 False。

```
In: sampler = WeightedRandomSampler(weights, 8, replacement=False)
    dataloader = DataLoader(dataset, batch_size=4, sampler=sampler)
    for datas, labels in dataloader:
        print(labels.tolist())
```

```
Print: [1, 1, 1, 1]
       [0, 0, 0, 0]
```

在这种情况下，num_samples 等于 dataset 的样本总数，为了不重复选取，sampler 会将每个样本都返回，这样就失去了 weight 参数的意义。

从上面的例子可见 sampler 在样本采样中的作用：如果指定了 sampler，shuffle 将不再生效，并且 sampler.num_samples 会覆盖 dataset 的实际大小，即一个 epoch 返回的图片总数取决于`sampler.num_samples`。

5.2 计算机视觉工具包：torchvision

计算机视觉是深度学习中最重要的一类应用，为了方便研究者使用，PyTorch 团队专门开发了一个视觉工具包`torchvision`，这个包独立于 PyTorch，需通过`pip install torchvision`安装。在之前的例子中我们已经使用过它的部分功能，这里再做一个系统性的介绍。torchvision 主要包含以下三部分。

- models：提供深度学习中各种经典网络的网络结构及预训练好的模型，包括 `AlexNet`、VGG 系列、ResNet 系列、Inception 系列等。
- datasets：提供常用的数据集加载，设计上都是继承`torch.utils.data.Dataset`，主要包括`MNIST`、`CIFAR10/100`、`ImageNet`、`COCO`等。
- transforms：提供常用的数据预处理操作，主要包括对 Tensor 及 PIL Image 对象的操作。

```
In: from torchvision import models
    from torch import nn
    # 加载预训练好的模型，如果不存在会下载
    # 预训练好的模型保存在 ~/.torch/models/下面
    resnet34 = models.resnet34(pretrained=True, num_classes=1000)

    # 修改最后的全连接层为10分类问题（默认是ImageNet上的1000分类）
    resnet34.fc=nn.Linear(512, 10)
```

```
In: from torchvision import datasets
    # 指定数据集路径为data，如果数据集不存在则进行下载
    # 通过train=False获取测试集
    dataset = datasets.MNIST('data/', download=True, train=False,
    transform=transform)
```

Transforms 中涵盖了大部分对 Tensor 和 PIL Image 的常用处理，这些已在上文提到，本节就不再详细介绍。需要注意的是转换分为两步，第一步：构建转换操作，例如`transf = transforms.Normalize(mean=x, std=y)`；第二步：执行转换操作，例如`output = transf(input)`。另外，还可将多个处理操作用 Compose 拼接起来，构成一个处理转换流程。

```
In: from torchvision import transforms
    to_pil = transforms.ToPILImage()
    to_pil(t.randn(3, 64, 64))# 程序输出如图5-3所示
```

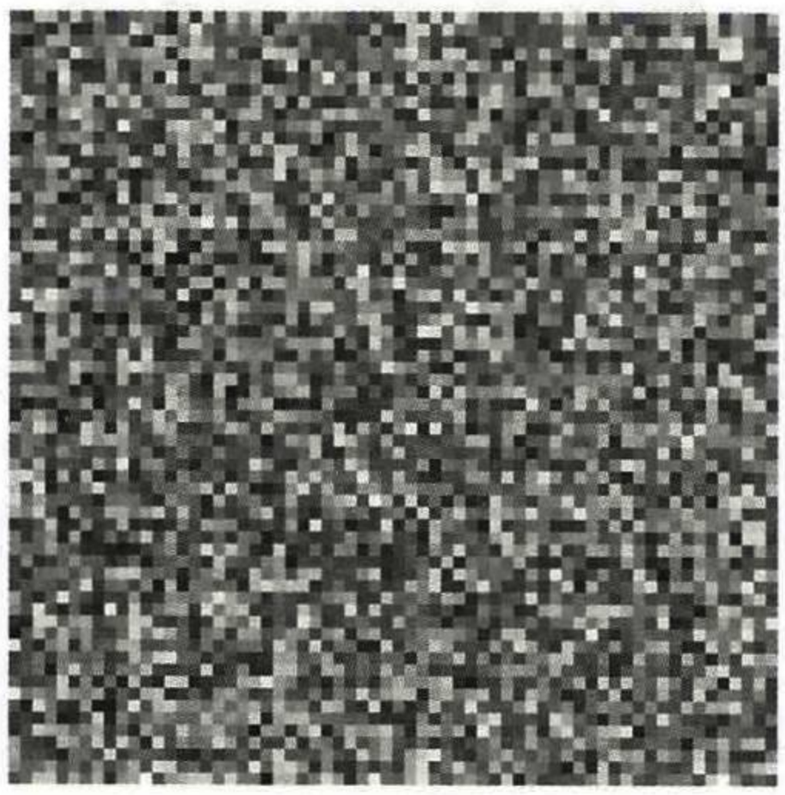


图 5-3　程序输出：随机噪声

torchvision 还提供了两个常用的函数。一个是`make_grid`，它能将多张图片拼接在一个网格中；另一个是`save_img`，它能将 Tensor 保存成图片。

```
In: len(dataset)
```

```
Out: 10000
```

```
In: dataloader = DataLoader(dataset, shuffle=True, batch_size=16)
    from torchvision.utils import make_grid, save_image
    dataiter = iter(dataloader)
    img = make_grid(next(dataiter)[0], 4) # 拼成4*4网格图片，且会转成 3 通
    道，如图5-4所示
    to_img(img)
```

```
In: save_image(img, 'a.png')
    Image.open('a.png')# 读取保存的图片，如图5-5所示
```

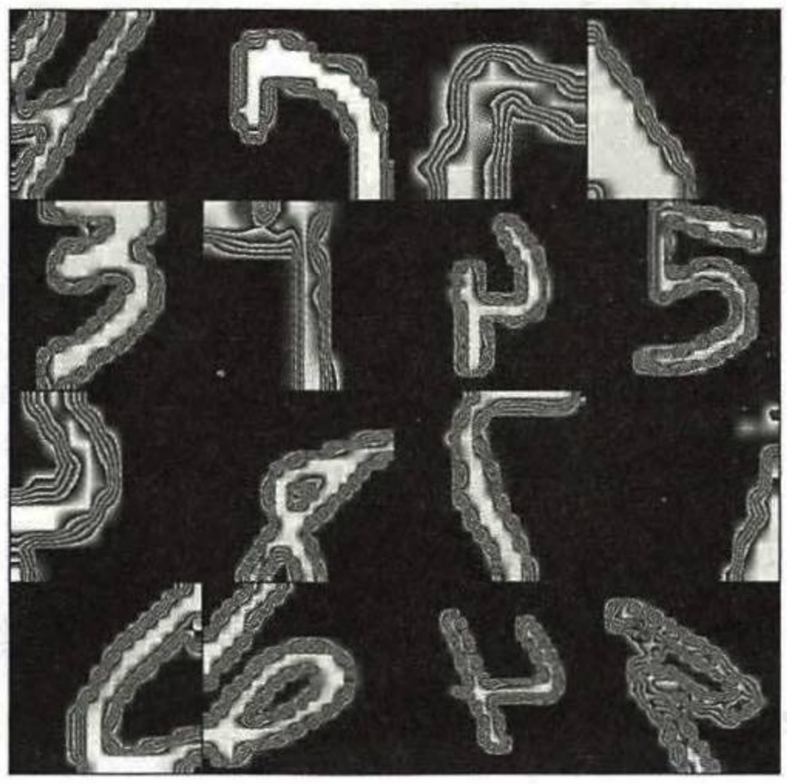


图 5-4 程序输出：经过数据增强处理的 MNIST 数据

图 5-5 程序输出：将图 5-4 保存成 png 文件

5.3 可视化工具

在训练神经网络时，我们希望能更直观地了解训练情况，包括损失曲线、输入图片、输出图片、卷积核的参数分布等信息。这些信息能帮助我们更好地监督网络的训练过程，并为参数优化提供方向和依据。最简单的办法就是打印输出，但其只能打印数值信息，不够直观，同时无法查看分布、图片、声音等。本节我们将介绍两个深度学习中常用的可视化工具：Tensorboard 和 visdom。

5.3.1 Tensorboard

最初，Tensorboard 是作为 TensorFlow 的可视化工具迅速流行开来的。作为和 TensorFlow 深度集成的工具，Tensorboard 能够展现 TensorFlow 网络计算图，绘制图像生成的定量指标图及附加数据，界面如图 5-6 所示。同时 Tensorboard 也是一个相对独立的工具，只要用户保存的数据遵循相应的格式，Tensorboard 就能读取这些数据并进行可视化。这里我们将主要介绍如何在 PyTorch 中使用 tensorboard_logger[①]进行训练损失的可视化。Tensorboard_logger 是 TeamHG-Memex 开发的一款轻量级工具，它将 Tensorboard 的功能抽取出来，使非 TensorFlow 用户也能使用它进行可视化，但其支持的功能有限。

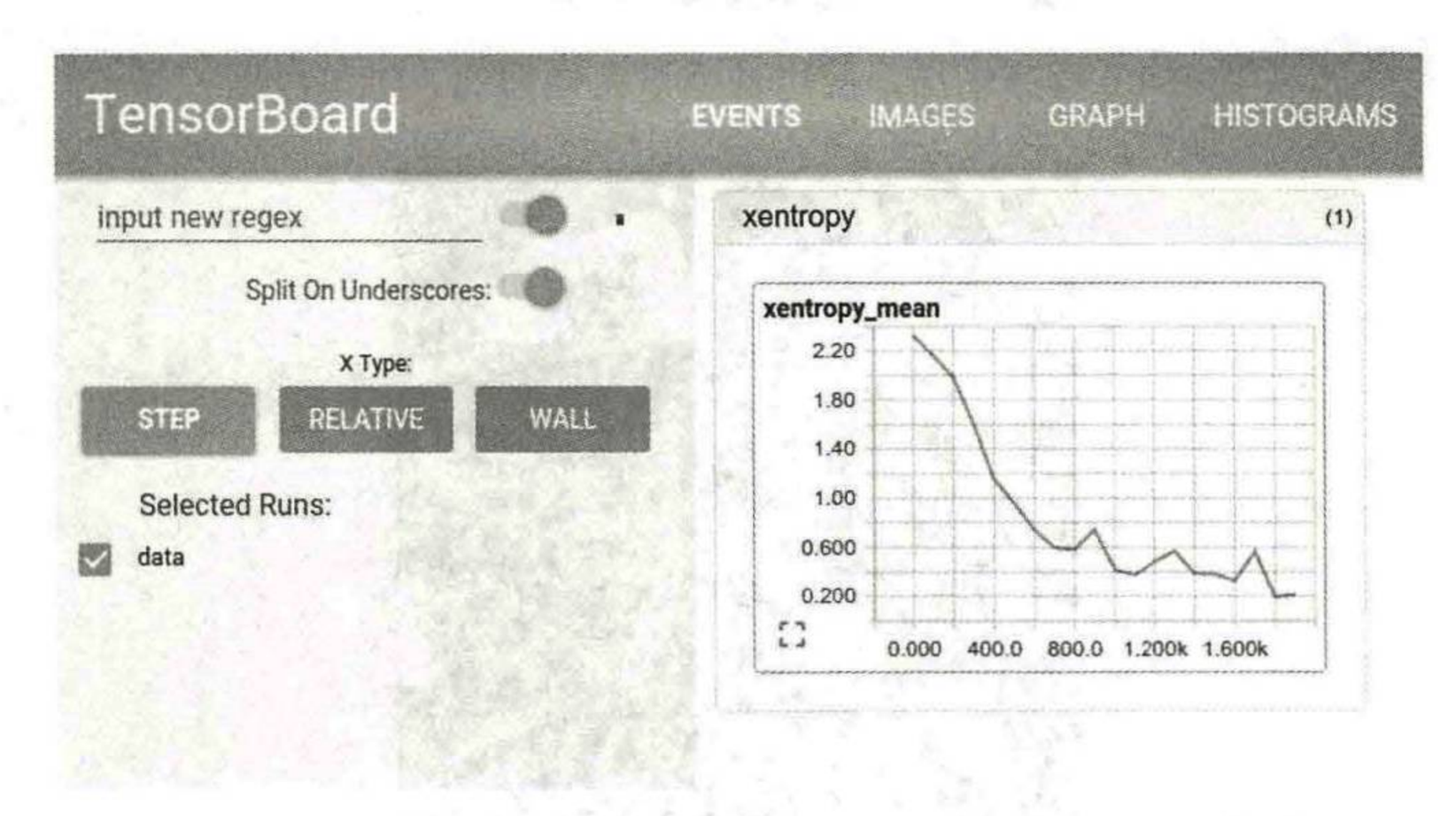


图 5-6 Tensorboard 界面

tensorboard_logger 的安装主要分为以下两步。

- 安装 TensorFlow：如果计算机中已经安装完 TensorFlow 可以跳过这一步，如果计算机中尚未安装，建议安装 CPU-Only 的版本，具体安装教程参见 TensorFlow 官网[②]，或使用 pip 直接安装，教育网用户可通过清华的开源镜像提高速度[③]。
- 安装 tensorboard_logger：可通过`pip install tensorboard_logger`命令直接安装。

tensorboard_logger 的使用非常简单。首先用如下命令启动 Tensorboard：

```
tensorboard --logdir <your/running/dir> --port <your_bind_port>
```

下面举例说明 tensorboard_logger 的使用。

---

① https://github.com/TeamHG-Memex/tensorboard_logger

② https://www.tensorflow.org/install/

③ https://mirrors.tuna.tsinghua.edu.cn/help/tensorflow/

```
In: from tensorboard_logger import Logger
```

```
In: # 构建logger对象，logdir用来指定log文件的保存路径
    # flush_secs用来指定刷新同步间隔
    logger = Logger(logdir='experimient_cnn', flush_secs=2)
```

```
In: for ii in range(100):
        logger.log_value('loss', 10-ii**0.5, step=ii)
        logger.log_value('accuracy', ii**0.5/10)
```

打开浏览器输入`http://localhost:6006`（其中 6006 应改成你的 Tensorboard 所绑定的端口），即可看到如图 5-7 所示的结果。

图 5-7　Tensorboard 可视化结果

左侧的 Horizontal Axis 下有如下三个选项。

- Step：根据步长来记录，log_value 是指如果有步长，则将其作为 x 轴坐标描点画线。
- Relative：用前后相对顺序描点画线，可认为 logger 自己维护了一个`step`属性，每调用一次 log_value 就自动加 1 。
- Wall：按时间排序描点画线。

左侧的 Smoothing 条可以左右拖动，用来调节平滑的幅度。单击页面右上角的刷新按钮可立即刷新结果，默认是每 30s 自动刷新数据。tensorboard_logger 的使用十分简单，但它只能统计简单的数值信息，不支持其他功能。

除了 tensorboard_logger，还有专门针对 PyTorch 开发的 TensorboardX[①]，它封装了

[①] https://github.com/lanpa/tensorboard-pytorch

更多的 Tensorboard 接口，支持记录标量、图片、直方图、声音、文本、计算图和 embedding 等信息，几乎包括和 TensorFlow 的 Tensorboard 完全一样的功能，使用接口甚至比 TensorFlow 的 Tensorboard 接口还要简单。感兴趣的读者可以自行了解，本节将重点介绍另一个可视化工具 visdom。

5.3.2 visdom

visdom[①]是 Facebook 专门为 PyTorch 开发的一款可视化工具，开源于 2017 年 3 月。visdom 十分轻量级，却支持非常丰富的功能，能胜任大多数的科学运算可视化任务，其可视化界面如图 5-8 所示。

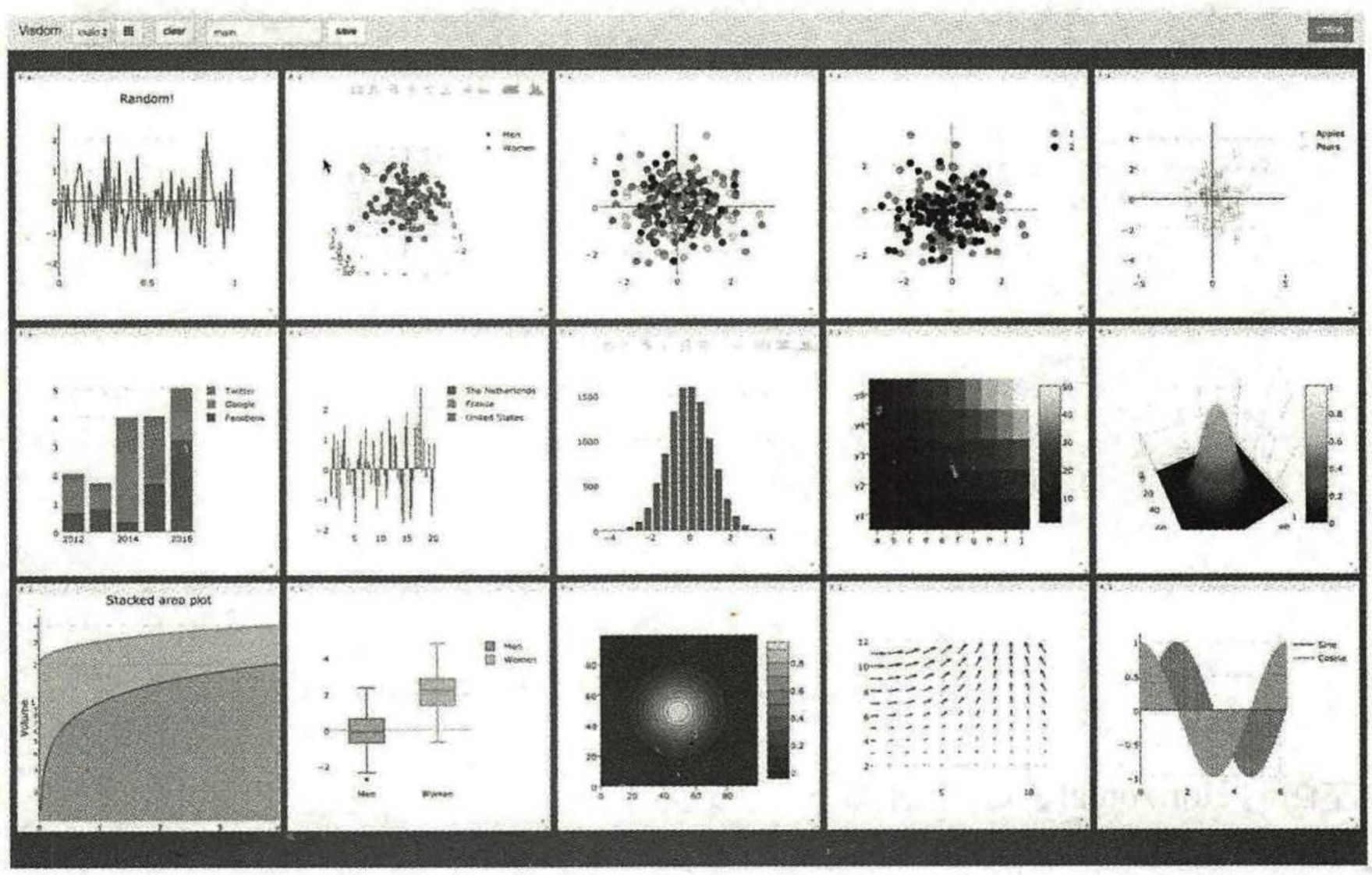


图 5-8 visdom 界面

visdom 可以创造、组织和共享多种数据的可视化，包括数值、图像、文本，甚至是视频，支持 PyTorch、Torch 及 Numpy。用户可通过编程组织可视化空间或通过用户接口为数据打造仪表板，检查实验结果和调试代码。

visdom 中有以下两个重要概念。

- env：环境。不同环境的可视化结果相互隔离，互不影响，在使用时如果不指定 env，默认使用`main`。不同用户、不同程序一般使用不同的 env。

[①] https://github.com/facebookresearch/visdom

- pane：窗格。窗格可用于可视化图像、数值或打印文本等，其可以拖动、缩放、保存和关闭。一个程序可使用同一个 env 中的不同 pane，每个 pane 可视化或记录某一信息。

如图 5-9 所示，当前 env 共有两个 pane，一个用于打印 log，另一个用于记录损失函数的变化。单击“clear”按钮可以清空当前 env 的所有 pane，单击“save”按钮可将当前 env 保存成 json 文件，保存路径位于`~/.visdom/`目录下。修改 env 的名字后单击 fork，可将当前 env 另存为新文件。

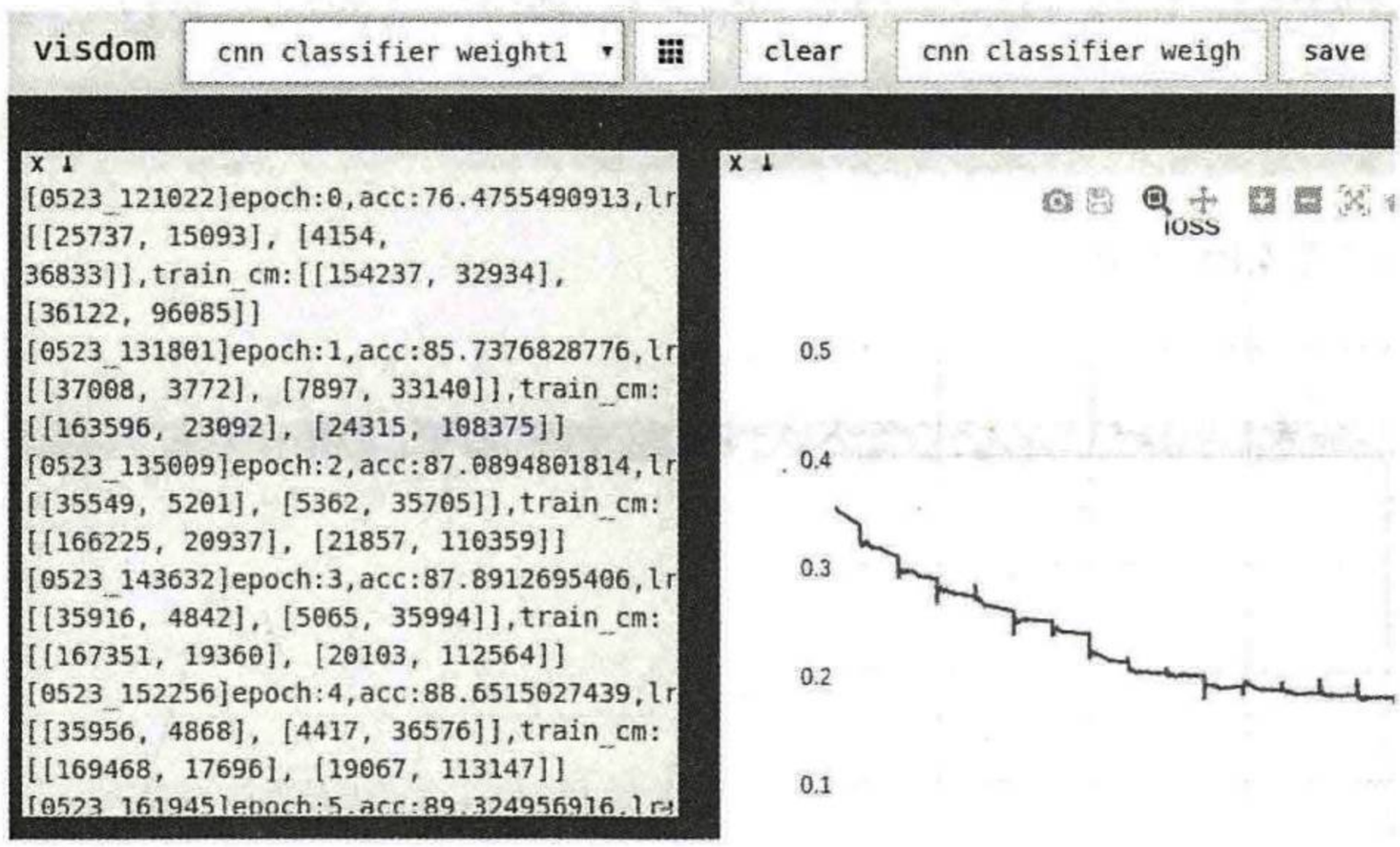


图 5-9 visdom_env

通过命令`pip install visdom`即可完成 visdom 的安装。安装完成后，需通过`python -m visdom.server`命令启动 visdom 服务，或通过`nohup python -m visdom.server &`命令将服务放至后台运行。visdom 服务是一个 Web Server 服务，默认绑定 8097 端口，客户端与服务器间通过 tornado 进行非阻塞交互。

在使用 visdom 时有两点需要注意的地方。

- 需手动指定保存 env，可在 Web 界面单击“save”按钮或在程序中调用 save 方法，否则 visdom 服务重启后，env 等信息会丢失。
- 客户端与服务器之间的交互采用 tornado 异步框架，可视化操作不会阻塞当前程序，网络异常也不会导致程序退出。

visdom 以 Plotly 为基础，支持丰富的可视化操作，下面举例说明一些最常用的操作。

```
In: %%sh
    # 启动visdom服务器
    # nohup python -m visdom.server &
```

```
In: import visdom

    # 新建一个连接客户端
    # 指定env = u'test1', 默认端口为8097, host是'localhost'
    vis = visdom.Visdom(env=u'test1')

    x = t.arange(1, 30, 0.01)
    y = t.sin(x)
    vis.line(X=x, Y=y, win='sinx', opts={'title': 'y=sin(x)'})
```

```
Out: u'sinx'
```

输出的结果如图 5-10 所示。

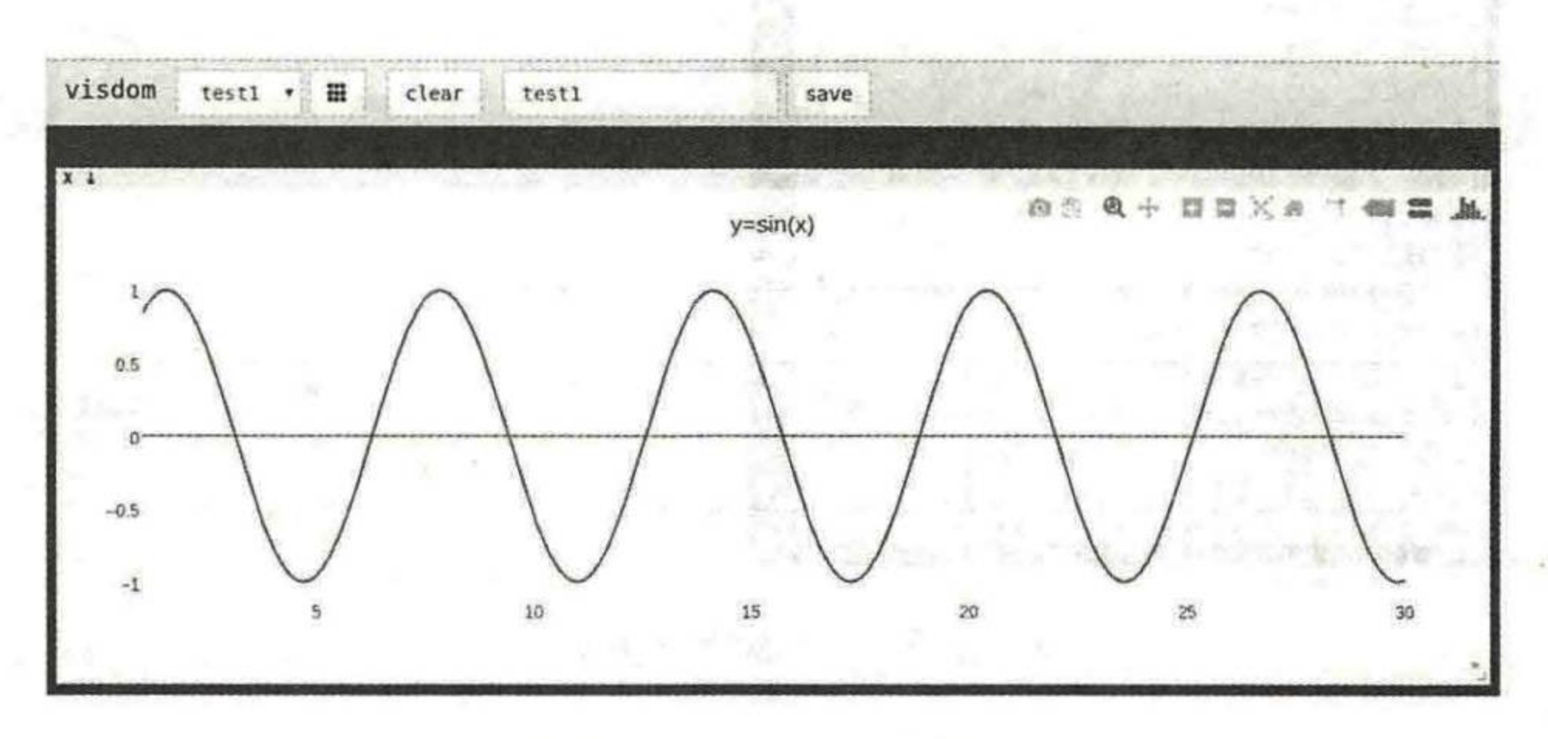


图 5-10　visdom 的输出

下面我们逐一分析这几行代码。

- `vis = visdom.Visdom(env=u'test1')`，用于构建一个客户端，客户端除指定 env 外，还可以指定 host、port 等参数。
- vis 作为一个客户端对象，可以使用如下常见的画图函数。
  - line：类似 MATLAB 中的`plot`操作，用于记录某些标量的变化，例如损失、准确率等。
  - image：可视化图片，可以是输入的图片，也可以是 GAN 生成的图片，还可以是卷积核的信息。
  - text：用于记录日志等文字信息，支持 HTML 格式。
  - histgram：可视化分布，主要是查看数据、参数的分布。
  - scatter：绘制散点图。

- bar：绘制柱状图。
- pie：绘制饼状图。
- 更多操作可参考 visdom 的 GitHub 主页。

本节主要介绍深度学习中常见的 line、image 和 text 的操作。

visdom 同时支持 PyTorch 的 tensor 和 numpy 的 ndarray 两种数据结构，但不支持 Python 的 int、float 等类型，因此每次传入时都需要先将数据转成 ndarray 或 tensor。上述操作的参数一般不同，但有两个参数是绝大多数操作都具备的。

- win：用于指定 pane 的名字，如果不指定，visdom 将自动分配一个新的 pane。如果两次操作指定的 win 名字一样，新的操作将覆盖当前 pane 的内容，因此建议每次操作都指定 win。
- opts：用来可视化配置，接收一个字典，常见的 option 包括`title`、`xlabel`、`ylabel`、`width`等，主要用于设置 pane 的显示格式。

之前提到过，每次操作都会覆盖之前的数值，但我们在训练网络的过程中往往需要不断更新数值，如损失值等，这时就需要指定参数`update='append'`来避免覆盖之前的数值。除了使用 update 参数，还可以使用`vis.updateTrace`方法更新图，`updateTrace`不仅能在指定 pane 上新增一个和已有数据相互独立的 Trace，还能像`update='append'`那样在同一条 trace 上追加数据。

```
In: # append 追加数据
    for ii in range(0, 10):
        # y = x
        x = t.Tensor([ii])
        y = x
        vis.line(X=x, Y=y, win='polynomial', update='append' if ii>0 else
        None)

    # updateTrace 新增一条线
    x = t.arange(0, 9, 0.1)
    y = (x ** 2) / 9
    vis.updateTrace(X=x, Y=y, win='polynomial', name='this is a new Trace'
    )
```

```
Out: u'polynomial'
```

打开浏览器，输入`http://localhost:8097`，可以看到如图 5-11 所示的结果。

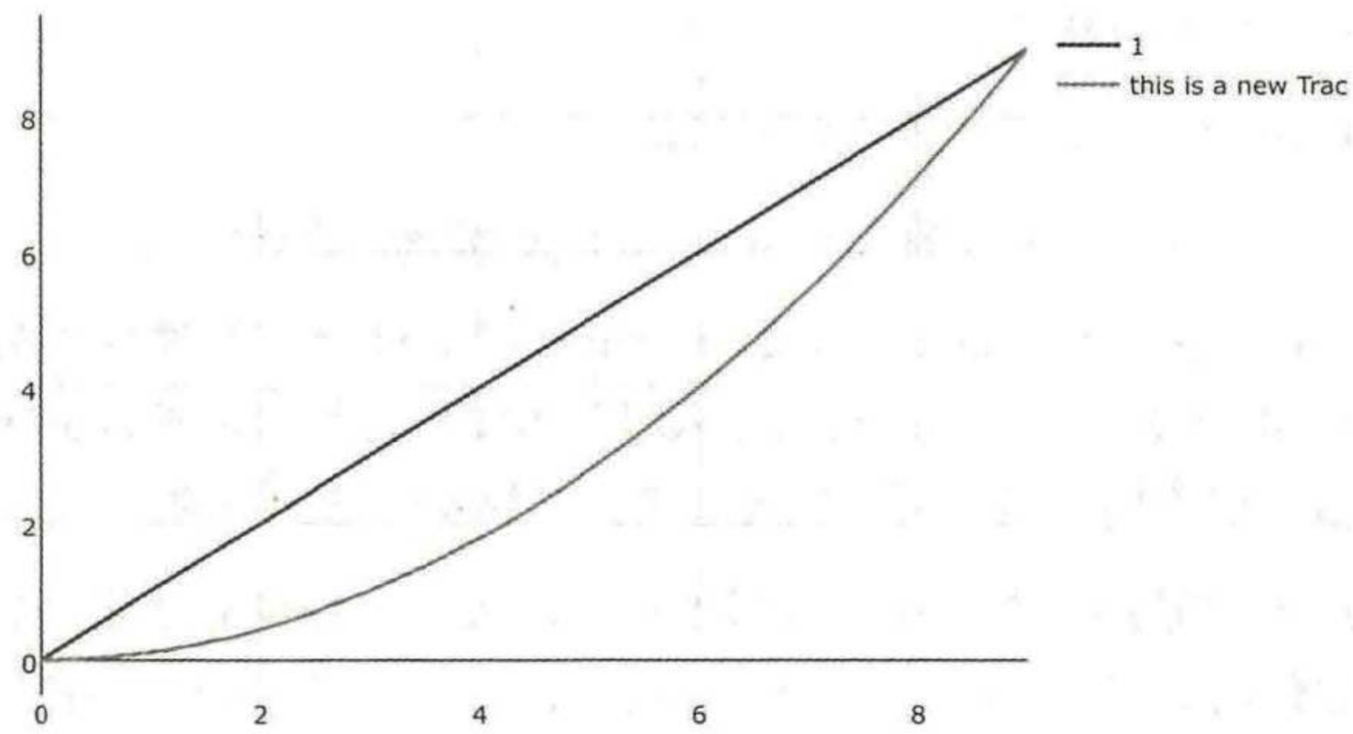


图 5-11　append 和 updateTrace 可视化效果

image 的画图功能可分为如下两类。

- `image`接收一个二维或三维向量，$H \times W$ 或 $3 \times H \times W$，前者是黑白图像，后者是彩色图像。
- `images`接收一个四维向量 $N \times C \times H \times W$，$C$ 可以是 1 或 3，分别代表黑白和彩色图像。可实现类似 torchvision 中 make_grid 的功能，将多张图片拼接在一起。`images`也可以接收一个二维或三维的向量，此时它所实现的功能与 image 一致。

```
In: # 可视化一张随机的黑白图片
    vis.image(t.randn(64, 64).numpy())

    # 可视化一张随机的彩色图片
    vis.image(t.randn(3, 64, 64).numpy(), win='random2')

    # 可视化36张随机的彩色图片，每一行6张
    vis.images(t.randn(36, 3, 64, 64).numpy(), nrow=6, win='random3', opts
    ={'title':'random_imgs'})
```

```
Out: u'random3'
```

其中 images 的可视化输出如图 5-12 所示。

`vis.text`用于可视化文本，图 5-13 是 visdom 的 text 的可视化输出，它支持所有的 html 标签，同时也遵循着 html 的语法标准。例如，换行需使用`<br>`标签，`\r\n`无法实现换行。下面举例说明。

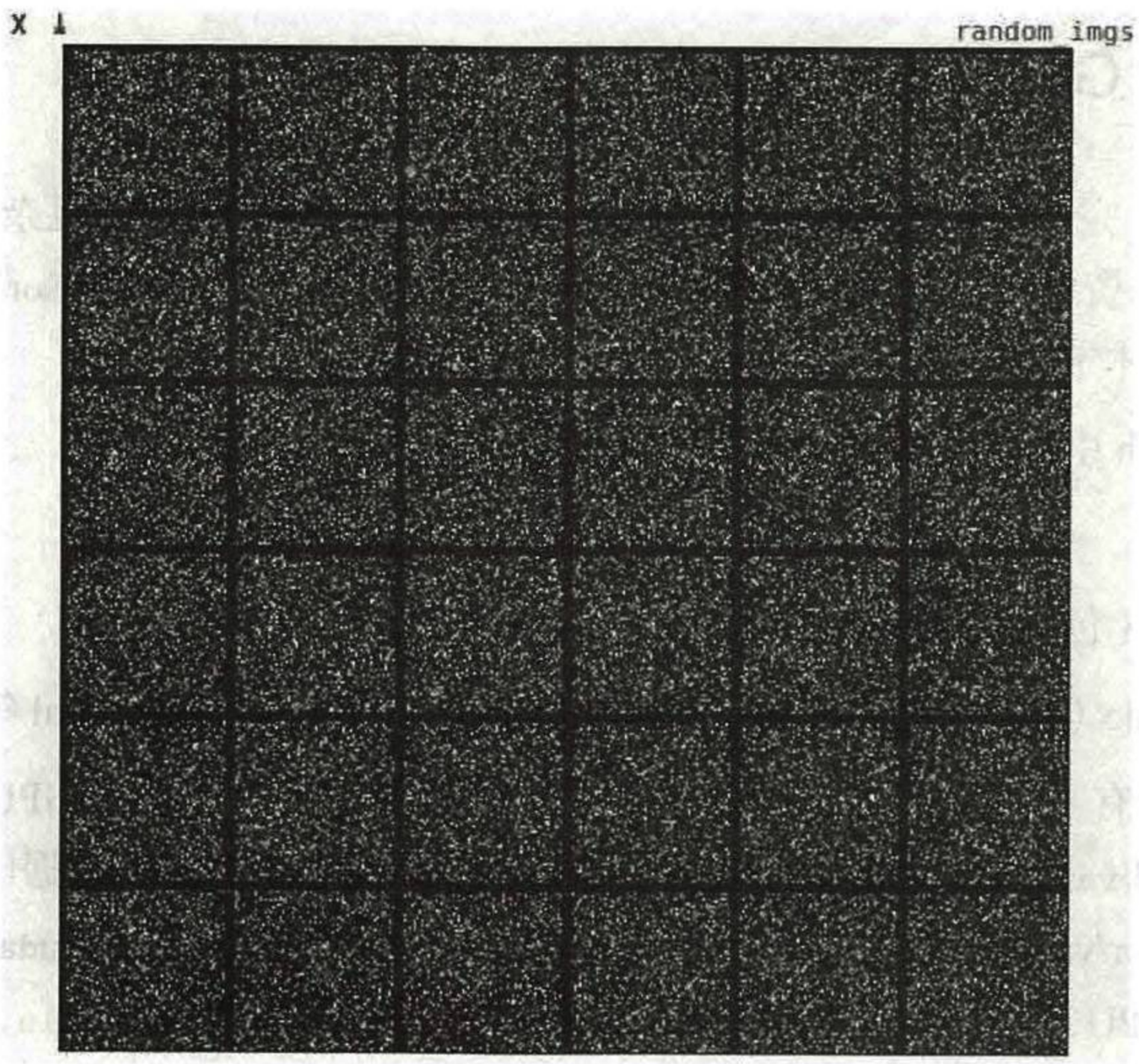


图 5-12　images 可视化输出

图 5-13　text 的可视化输出

```
In: vis.text(u'''<h1>Hello visdom</h1><br>visdom是Facebook专门为<b>PyTorch
</b>开发的一个可视化工具，
            在内部使用了很久，于2017年3月开源。

        Visdom十分轻量级，却有十分强大的功能，支持几乎所有的科学运算
        可视化任务''',
        win='visdom',
        opts={'title': u'visdom简介' }
       )
```

```
Out: u'visdom'
```

5.4 使用 GPU 加速：cuda

与对 GPU 完全透明的 Theano 相比，在 PyTorch 中使用 GPU 会复杂一些，但这也意味着对 GPU 资源更加灵活高效的控制。这部分内容在前面介绍 Tensor、Module 时大多都提到过，本节将对其做一个总结，并介绍相关应用。

在 PyTorch 中以下数据结构分为 CPU 和 GPU 两个版本。

- Tensor
- Variable（包括 Parameter）
- nn.Module（包括常用的 layer、loss function，以及容器 Sequential 等）

它们都带有一个`.cuda`方法，调用此方法即可将其转为对应的 GPU 对象。注意，`tensor.cuda`和`variable.cuda`都会返回一个新对象，这个新对象的数据已转移至 GPU，而之前的 tensor/variable 的数据还在原来的设备上（CPU）。`module.cuda`会将所有的数据都迁移至 GPU，并返回自己。所以`module = module.cuda()`和`module.cuda()`的效果相同。

Variable 和 nn.Module 在 GPU 与 CPU 之间的转换，本质上还是利用了 Tensor 在 GPU 和 CPU 之间的转换。`Variable.cuda`操作实际上是将`variable.data`转移至指定的 GPU。而`nn.Module`的 cuda 方法是将 nn.Module 下的所有 parameter（包括子 module 的 parameter）都转移至 GPU，而 Parameter 本质上也是 Variable。

下面将举例说明，运行这部分代码需要读者有两块 GPU 设备。

注意：为什么将数据转移至 GPU 的方法叫做`.cuda`而不是`.gpu`呢？这是因为 GPU 的编程接口采用 CUDA，而目前并不是所有的 GPU 都支持 CUDA，只有部分 NVIDIA 的 GPU 才支持。PyTorch 未来可能会支持 AMD 的 GPU，而 AMD GPU 的编程接口采用 OpenCL，因此 PyTorch 还预留着`.cl`方法，用于以后支持 AMD 等的 GPU。

```
In: tensor = t.Tensor(3, 4)
    # 返回一个新的tensor，保存在第1块GPU上，但原来的tensor并没有改变
    tensor.cuda(0)
    tensor.is_cuda # False
```

```
In: # 不指定所使用的GPU设备，将默认使用第1块GPU
    tensor = tensor.cuda()
    tensor.is_cuda # True
```

```
In: variable = t.autograd.Variable(tensor)
    variable.cuda()
    variable.is_cuda # False
```

```
In: module = nn.Linear(3, 4)
    module.cuda(device_id = 1)
    module.weight.is_cuda # True
```

```
In: class VeryBigModule(nn.Module):
        def __init__(self):
            super(VeryBigModule, self).__init__()
            self.GiantParameter1 = t.nn.Parameter(t.randn(100000, 20000)).
            cuda(0)
            self.GiantParameter2 = t.nn.Parameter(t.randn(20000, 100000)).
            cuda(1)

        def forward(self, x):
            x = self.GiantParameter1.mm(x.cuda(0))
            x = self.GiantParameter2.mm(x.cuda(1))
            return x
```

上面最后一部分中，两个 Parameter 所占用的内存空间都非常大，大概是 8GB，如果将这两个 Parameter 同时放在一块 GPU 上几乎会将显存占满，无法再进行任何其他运算。此时可通过这种方式将不同的计算分布到不同的 GPU 中。

关于使用 GPU 的一些建议：

- GPU 运算很快，但运算量小时，并不能体现出它的优势，因此一些简单的操作可直接利用 CPU 完成。
- 数据在 CPU 和 GPU 之间的传递会比较耗时，应当尽量避免。
- 在进行低精度的计算时，可以考虑`HalfTensor`，相比`FloatTensor`能节省一半的显存，但需千万注意数值溢出的情况。

注意：大部分的损失函数也都属于`nn.Module`，但在使用 GPU 时，很多时候我们都忘记使用它的`.cuda`方法，在大多数情况下不会报错，因为损失函数本身没有可学习的参数（learnable parameters）。但在某些情况下会出现问题，为了保险起见同时也为了代码更规范，应记得调用`criterion.cuda`。下面我们举例说明。

```
In: # 交叉熵损失函数，带权重
    criterion = t.nn.CrossEntropyLoss(weight=t.Tensor([1, 3]))
```

```
input = t.autograd.Variable(t.randn(4, 2)).cuda()
target = t.autograd.Variable(t.Tensor([1, 0, 0, 1])).long().cuda()

# 下面这行会报错，因为weight未被转移至GPU
# loss = criterion(input, target)

# 这行则不会报错
criterion.cuda()
loss = criterion(input, target)

criterion._buffers
```

除了调用对象的.cuda方法外，还可以使用torch.cuda.device指定默认使用哪一块GPU，或使用torch.set_default_tensor_type使程序默认使用GPU，不需要手动调用cuda。

```
In: # 如果未指定使用哪块GPU，默认使用GPU 0
    x = t.cuda.FloatTensor(2, 3)
    # x.get_device() == 0
    y = t.FloatTensor(2, 3).cuda()
    # y.get_device() == 0

    # 指定默认使用GPU 1
    with t.cuda.device(1):
        # 在GPU 1上构建tensor
        a = t.cuda.FloatTensor(2, 3)

        # 将tensor转移至GPU 1
        b = t.FloatTensor(2, 3).cuda()
        print(a.get_device() == b.get_device() == 1 )

        c = a + b
        print(c.get_device() == 1)

        z = x + y
        print(z.get_device() == 0)

        # 手动指定使用GPU 0
```

```
d = t.randn(2, 3).cuda(0)
print(d.get_device() == 2)
```

```
In: t.set_default_tensor_type('torch.cuda.FloatTensor') # 指定默认tensor的
类型为GPU上的FloatTensor
    a = t.ones(2, 3)
    a.is_cuda
```

如果服务器具有多个 GPU，`tensor.cuda()`方法会将 tensor 保存到第一块 GPU 上，等价于`tensor.cuda(0)`。此时如果想使用第二块 GPU，需手动指定`tensor.cuda(1)`，而这需要修改大量代码很烦琐。这里有两种替代方法：

- 一种方法是先调用`t.cuda.set_device(1)`指定使用第二块 GPU，后续的`.cuda()`都无须更改，切换 GPU 只需修改这一行代码。
- 另一种方法是设置环境变量 `CUDA_VISIBLE_DEVICES`，例如当 `export CUDA_VISIBLE_DEVICE=1` 时（下标是从 0 开始，1 代表第二块 GPU），只使用第二块物理 GPU，但在程序中这块 GPU 会被看成是第一块逻辑 GPU，因此此时调用`tensor.cuda()`会将 Tensor 转移至第二块物理 GPU。`CUDA_VISIBLE_DEVICES` 还可以指定多个 GPU，如`export CUDA_VISIBLE_DEVICES =0,2,3`，那么第一、三、四块物理 GPU 会被映射成第一、二、三块逻辑 GPU，`tensor.cuda(1)`会将 Tensor 转移到第三块物理 GPU 上。

设置 `CUDA_VISIBLE_DEVICES` 有两种方法，一种是在命令行中 `CUDA_VISIBLE_DEVICES =0,1 python main.py`，一种是在程序中`import os;os.environ["CUDA_VISIBLE_DEVICES"] = "2"`。

从 PyTorch 0.2 版本中，PyTorch 新增分布式 GPU 支持。注意分布式和并行的区别：分布式是指有多个 GPU 在多台服务器上，而并行一般指的是一台服务器上的多个 GPU。分布式涉及了服务器之间的通信，因此比较复杂，PyTorch 封装了相应的接口，可以用几句简单的代码实现分布式训练。分布式对普通用户来说比较遥远，因为搭建一个分布式集群的代价很大，使用也比较复杂。相比之下，一机多卡更现实。对于分布式训练，这里不做太多的介绍，感兴趣的读者可参考文档[①]。

5.5 持久化

在 PyTorch 中，以下对象可以持久化到硬盘，并能通过相应的方法加载到内存中。

[①] http://pytorch.org/docs/0.2.0/distributed.html

- Tensor
- Variable
- nn.Module
- Optimizer

本质上，上述这些信息最终都是保存成 Tensor。Tensor 的保存和加载十分简单，使用 t.save 和 t.load 即可完成相应的功能。在 save/load 时可指定使用的 pickle 模块，在 load 时还可将 GPU tensor 映射到 CPU 或其他 GPU 上。

我们可以通过`t.save(obj, file_name)`等方法保存任意可序列化的对象，然后通过`obj = t.load(file_name)`方法加载保存的数据。对 Module 和 Optimizer 对象，这里建议保存对应的`state_dict`，而不是直接保存整个 Module/Optimizer 对象。Optimizer 对象保存的是参数及动量信息，通过加载之前的动量信息，能够有效地减少模型震荡，下面举例说明。

```
In: a = t.Tensor(3, 4)
    if t.cuda.is_available():
            a = a.cuda(1) # 把a转为GPU1上的tensor,
            t.save(a,'a.pth')

            # 加载为b, 存储于GPU1上(因为保存时tensor就在GPU1上)
            b = t.load('a.pth')

            # 加载为c, 存储于CPU
            c = t.load('a.pth', map_location=lambda storage, loc: storage)

            # 加载为d, 存储于GPU0上
            d = t.load('a.pth', map_location={'cuda:1':'cuda:0'})
```

```
In: t.set_default_tensor_type('torch.FloatTensor')
    from torchvision.models import AlexNet
    model = AlexNet()
    # module的state_dict是一个字典
    model.state_dict().keys()
```

```
In: # Module对象的保存与加载
    t.save(model.state_dict(), 'alexnet.pth')
    model.load_state_dict(t.load('alexnet.pth'))
```

```
In: optimizer = t.optim.Adam(model.parameters(), lr=0.1)
```

```
In: t.save(optimizer.state_dict(), 'optimizer.pth')
    optimizer.load_state_dict(t.load('optimizer.pth'))
```

```
In: all_data = dict(
        optimizer = optimizer.state_dict(),
        model = model.state_dict(),
        info = u'模型和优化器的所有参数'
    )
    t.save(all_data, 'all.pth')
```

```
In: all_data = t.load('all.pth')
    all_data.keys()
```

本章介绍了一些工具模块，这些工具有些位于 PyTorch 中，有些独立于 PyTorch 的第三方模块。这些模块主要涉及数据加载、可视化和 GPU 加速相关的内容，合理地使用这些模块能极大地提升我们的编程效率。

6 PyTorch 实战指南

通过前面几章的学习，我们已经掌握了 PyTorch 中大部分的基础知识，本章将结合之前讲的内容，带领读者从头实现一个完整的深度学习项目。本章的重点不在于如何使用 PyTorch 的接口，而在于合理地设计程序的结构，使得程序更具可读性、更易用。

6.1 编程实战：猫和狗二分类

在学习某个深度学习框架时，掌握其基本知识和接口固然重要，但如何合理地组织代码，使代码具有良好的可读性和可扩展性也必很关键。本章将不再深入讲解过多知识性的东西，更多是传授一些经验，这些内容可能有些争议，因其受笔者个人喜好和 coding 风格影响较大，读者可以将这部分当成是一种参考或提议，而不是作为必须遵循的准则。归根到底，都是希望读者能以一种更合理的方式组织自己的程序。

在做深度学习实验或项目时，为了得到最优的模型结果，中间往往需要很多次尝试和修改。合理的文件组织结构，以及一些小技巧可以极大地提高代码的易读易用性。根据笔者的个人经验，在从事大多数深度学习研究时，程序都需要实现以下几个功能。

- 模型定义
- 数据处理和加载
- 训练模型（Train&Validate）
- 训练过程的可视化
- 测试（Test/Inference）

另外，程序还应该满足以下几个要求：模型需具有高度可配置性，便于修改参数、修改模型和反复实验；代码应具有良好的组织结构，使人一目了然；代码应具有良好的说明，使其他人能够理解。

在之前的章节中，我们已经讲解了 PyTorch 中的绝大部分内容。本章我们将应用这些内容，并结合实际例子讲解如何用 PyTorch 完成 Kaggle 上的经典比赛：Dogs vs. Cats[①]。本章所有示例程序均在本书的配套代码 `chapter6/best_practice` 中。

6.1.1 比赛介绍

Dogs vs. Cats 是一个传统的二分类问题，其训练集包含 25000 张图片，部分图片如图 6-1 所示，这些图片均放置在同一文件夹下，命名格式为`<category>.<num>.jpg`，例如`cat.10000.jpg`和`dog.100.jpg`，测试集包含 12500 张图片，命名为`<num>.jpg`，例如`1000.jpg`。参赛者需根据训练集的图片训练模型，并在测试集上进行预测，输出它是狗的概率。最后提交的 csv 文件如下，第一列是图片的`<num>`，第二列是图片为狗的概率。

```
id,label
10001,0.889
10002,0.01
...
```

图 6-1 猫和狗的数据

6.1.2 文件组织架构

前面提到过程序的主要功能，其中最重要的三个功能如下。

① https://www.kaggle.com/c/dogs-vs-cats-redux-kernels-edition

- 模型定义
- 数据加载
- 训练和测试

首先来看程序文件的组织结构：

```
checkpoints/
data/
    __init__.py
    dataset.py
    get_data.sh
models/
    __init__.py
    AlexNet.py
    BasicModule.py
    ResNet34.py
utils/
    __init__.py
    visualize.py
config.py
main.py
requirements.txt
README.md
```

其中各个文件的主要内容和作用如下。

- `checkpoints/`：用于保存训练好的模型，可使程序在异常退出后仍能重新载入模型，恢复训练。
- `data/`：数据相关操作，包括数据预处理、dataset 实现等。
- `models/`：模型定义，可以有多个模型，例如上面的 AlexNet 和 ResNet34，一个模型对应一个文件。
- `utils/`：可能用到的工具函数，本次实验中主要封装了可视化工具。
- `config.py`：配置文件，所有可配置的变量都集中在此，并提供默认值。
- `main.py`：主文件，训练和测试程序的入口，可通过不同的命令来指定不同的操作和参数。
- `requirements.txt`：程序依赖的第三方库。
- `README.md`：提供程序的必要说明。

6.1.3 关于__init__.py

可以看到，几乎每个文件夹下都有__init__.py，一个目录如果包含了__init__.py 文件，那么它就变成了一个包（package）。__init__.py可以为空，也可以定义包的属性和方法，但其必须存在，其他程序才能从这个目录中导入相应的模块或函数。例如在data/文件夹下有__init__.py，则在main.py 中就可以from data.dataset import DogCat。如果在__init__.py中写入from .dataset import DogCat，则在 main.py 中就可以直接写为：from data import DogCat，或者import data; dataset = data.DogCat，比写为from data.dataset import DogCat更便捷。

6.1.4 数据加载

数据的相关处理主要保存在data/dataset.py中。关于数据加载的相关操作，在第5章中我们已经提到过，其基本原理就是使用Dataset封装数据集，再使用Dataloader实现数据并行加载。Kaggle 提供的数据包括训练集和测试集，而我们在实际使用中，还需专门从训练集中取出一部分作为验证集。对于这三类数据集，其相应操作也不太一样，而如果专门写三个Dataset，则稍显复杂和冗余，因此这里通过加一些判断来区分。我们希望对训练集做一些数据增强处理，如随机裁剪、随机翻转、加噪声等，而验证集和测试集则不需要。下面看dataset.py的代码：

```
import os
from PIL import Image
from torch.utils import data
import numpy as np
from torchvision import transforms as T

class DogCat(data.Dataset):

    def __init__(self, root, transforms=None, train=True, test=False):
        '''
        目标：获取所有图片地址，并根据训练、验证、测试划分数据
        '''
        self.test = test
        imgs = [os.path.join(root, img) for img in os.listdir(root)]

        # test1: data/test1/8973.jpg
```

```
        # train: data/train/cat.10004.jpg
        if self.test:
            imgs = sorted(imgs, key=lambda x: int(x.split('.')[-2].split('
            /')[-1]))
        else:
            imgs = sorted(imgs, key=lambda x: int(x.split('.')[-2]))

        imgs_num = len(imgs)

        # 划分训练、验证集，验证:训练 = 3:7
        if self.test:
            self.imgs = imgs
        elif train:
            self.imgs = imgs[:int(0.7*imgs_num)]
        else :
            self.imgs = imgs[int(0.7*imgs_num):]

        if transforms is None:

            # 数据转换操作，测试验证和训练的数据转换有所区别

            normalize = T.Normalize(mean = [0.485, 0.456, 0.406],
                                    std = [0.229, 0.224, 0.225])

            # 测试集和验证集
            if self.test or not train:
                self.transforms = T.Compose([
                    T.Scale(224),
                    T.CenterCrop(224),
                    T.ToTensor(),
                    normalize
                ])
            # 训练集
            else :
                self.transforms = T.Compose([
                    T.Scale(256),
                    T.RandomSizedCrop(224),
```

```
                T.RandomHorizontalFlip(),
                T.ToTensor(),
                normalize
            ])


    def __getitem__(self, index):
        '''
        返回一张图片的数据
        如果是测试集，没有图片id，如1000.jpg返回1000
        '''
        img_path = self.imgs[index]
        if self.test:
            label = int(self.imgs[index].split('.')[-2].split('/')[-1])
        else:
            label = 1 if 'dog' in img_path.split('/')[-1] else 0
        data = Image.open(img_path)
        data = self.transforms(data)
        return data, label


    def __len__(self):
        '''
        返回数据集中所有图片的个数
        '''
        return len(self.imgs)
```

有关数据集使用的注意事项在第 5 章中已经提到，将文件读取等费时操作放在 `__getitem__` 函数中，利用多进程加速。一次性将所有图片都读进内存，不仅费时也会占用较大内存，而且不易进行数据增强等操作。我们将训练集中的 30% 作为验证集，可以用来检查模型的训练效果，避免过拟合。在使用时，我们可以通过 dataloader 加载数据。

```
train_dataset = DogCat(opt.train_data_root, train=True)
trainloader = DataLoader(train_dataset,
                         batch_size = opt.batch_size,
                         shuffle = True,
                         num_workers = opt.num_workers)

for ii, (data, label) in enumerate(trainloader):
```

```
train()
```

6.1.5 模型定义

模型的定义主要保存在models/目录下，其中BasicModule是对nn.Module的简易封装，提供快速加载和保存模型的接口。

```
class BasicModule(t.nn.Module):
    '''
    封装了nn.Module，主要提供save和load两个方法
    '''

    def __init__(self):
        super(BasicModule,self).__init__()
        self.model_name = str(type(self)) # 模型的默认名字

    def load(self, path):
        '''
        可加载指定路径的模型
        '''
        self.load_state_dict(t.load(path))

    def save(self, name=None):
        '''
        保存模型，默认使用“模型名字+时间”作为文件名，
        如AlexNet_0710_23:57:29.pth
        '''
        if name is None:
            prefix = 'checkpoints/' + self.model_name + '_'
            name = time.strftime(prefix + '%m%d_%H:%M:%S.pth')
        t.save(self.state_dict(), name)
        return name
```

在实际使用中，直接调用model.save()及model.load(opt.load_path)即可。

其他自定义模型一般继承BasicModule，然后实现自己的模型。其中AlexNet.py实现了 AlexNet，ResNet.py实现了 ResNet34。在models/__init__py中，代码如下：

```
from .AlexNet import AlexNet
from .ResNet34 import ResNet34
```

这样在主函数中就可以写成：

```
from models import AlexNet
或
import models
model = models.AlexNet()
或
import models
model = getattr('models', 'AlexNet')()
```

其中最后一种写法最关键，这意味着我们可以通过字符串直接指定使用的模型，而不必使用判断语句，也不必在每次新增加模型后都修改代码。新增模型后只需要在`models/__init__.py`中加上`from .new_module import new_module`即可。

其他关于模型定义的注意事项，在第 5 章中已详细讲解，本节就不再赘述，总结起来就是：

- 尽量使用`nn.Sequential`。
- 将经常使用的结构封装成子 module。
- 将重复且有规律性的结构用函数生成。

6.1.6 工具函数

在项目中，我们可能会用到一些 helper 方法，这些方法可以统一放在`utils/`文件夹下，需要使用时再引入。本例主要封装了可视化工具 visdom 的一些操作，其代码如下。本次实验中只会用到`plot`方法，用来统计损失信息。

```
#coding:utf8
import visdom
import time
import numpy as np

class Visualizer(object):
    '''
    封装了visdom的基本操作，但仍然可以通过`self.vis.function`
    或者`self.function`调用原生的visdom接口
```

```
    例如
    self.text('hello visdom')
    self.histogram(t.randn(1000))
    self.line(t.arange(0, 10),t.arange(1, 11))
    '''

    def __init__(self, env='default', **kwargs):
        self.vis = visdom.Visdom(env=env, **kwargs)

        # 保存（'loss',23） 即loss的第23个点
        self.index = {}
        self.log_text = ''
    def reinit(self, env='default', **kwargs):
        '''
        修改visdom的配置
        '''
        self.vis = visdom.Visdom(env=env, **kwargs)
        return self

    def plot_many(self, d):
        '''
        一次plot多个
        @params d: dict (name, value) i.e. ('loss', 0.11)
        '''
        for k, v in d.items():
            self.plot(k, v)

    def img_many(self, d):
        for k, v in d.items():
            self.img(k, v)

    def plot(self, name, y, **kwargs):
        '''
        self.plot('loss', 1.00)
        '''
        x = self.index.get(name, 0)
        self.vis.line(Y=np.array([y]), X=np.array([x]),
```

```
                    win=(name),
                    opts=dict(title=name),
                    update=None if x == 0 else 'append',
                    **kwargs
                    )
        self.index[name] = x + 1

    def img(self, name, img_, **kwargs):
        '''
        self.img('input_img', t.Tensor(64, 64))
        self.img('input_imgs', t.Tensor(3, 64, 64))
        self.img('input_imgs', t.Tensor(100, 1, 64, 64))
        self.img('input_imgs', t.Tensor(100, 3, 64, 64), nrows=10)

        !!! don't ~~self.img('input_imgs', t.Tensor(100, 64, 64), nrows
        =10)~~ !!!
        '''
        self.vis.images(img_.cpu().numpy(),
                        win=(name),
                        opts=dict(title=name),
                        **kwargs
                        )

    def log(self, info, win='log_text'):
        '''
        self.log({'loss':1, 'lr':0.0001})
        '''

        self.log_text += ('[{time}] {info} <br>'.format(
                            time=time.strftime('%m%d_%H%M%S'),\
                            info=info))
        self.vis.text(self.log_text, win)

    def __getattr__(self, name):
        '''
        自定义的plot,image,log,plot_many等除外
        self.function 等价于self.vis.function
```

```
        '''
        return getattr(self.vis, name)
```

6.1.7 配置文件

在模型定义、数据处理和训练等过程中有很多变量，这些变量应提供默认值，并统一放置在配置文件中，这样在后期调试、修改代码或迁移程序时会比较方便，在这里我们将所有可配置项放在config.py中。

```
class DefaultConfig(object):
    env = 'default' # visdom 环境
    model = 'AlexNet' # 使用的模型，名字必须与models/__init__.py中的名字一致

    train_data_root = './data/train/' # 训练集存放路径
    test_data_root = './data/test1' # 测试集存放路径
    load_model_path = 'checkpoints/model.pth' # 加载预训练模型的路径，为None代表不加载

    batch_size = 128 # batch size
    use_gpu = True # use GPU or not
    num_workers = 4 # how many workers for loading data
    print_freq = 20 # print info every N batch

    debug_file = '/tmp/debug' # if os.path.exists(debug_file): enter ipdb
    result_file = 'result.csv'

    max_epoch = 10
    lr = 0.1 # initial learning rate
    lr_decay = 0.95 # when val_loss increase, lr = lr*lr_decay
    weight_decay = 1e-4 # 损失函数
```

可配置的参数主要包括：

- 数据集参数（文件路径、batch_size 等）。
- 训练参数（学习率、训练 epoch 等）。
- 模型参数。

在程序中可以这样使用配置参数：

```
import models
from config import DefaultConfig

opt = DefaultConfig()
lr = opt.lr
model = getattr(models, opt.model)
dataset = DogCat(opt.train_data_root)
```

这些都只是默认参数，在这里还提供了更新函数，根据字典更新配置参数。

```
def parse(self, kwargs):
        '''
        根据字典kwargs更新config参数
        '''
        # 更新配置参数
        for k, v in kwargs.items():
            if not hasattr(self, k):
                warnings.warn("Warning: opt has not attribut %s" %k)
            setattr(self, k, v)

        # 打印配置信息
        print('user config:')
        for k, v in self.__class__.__dict__.items():
            if not k.startswith('__'):
                print(k, getattr(self, k))
```

我们在实际使用时不需要每次都修改`config.py`，只需要通过命令行传入所需参数，覆盖默认配置即可。

例如：

```
opt = DefaultConfig()
new_config = {'lr':0.1,'use_gpu':False}
opt.parse(new_config)
opt.lr == 0.1
```

6.1.8 main.py

在讲解主程序main.py之前，我们先来了解 2017 年 3 月谷歌开源的一个命令行工具fire[1]，通过pip install fire即可安装。下面介绍fire的基础用法，假设example.py文件内容如下：

```
import fire

def add(x, y):
  return x + y

def mul(**kwargs):
    a = kwargs['a']
    b = kwargs['b']
    return a * b

if __name__ == '__main__':
  fire.Fire()
```

那么我们可以使用：

```
python example.py add 1 2 # 执行add(1, 2)
python example.py mul --a=1 --b=2 # 执行mul(a=1, b=2), kwargs={'a':1, 'b
':2}
python example.py add --x=1 --y==2 # 执行add(x=1, y=2)
```

可见，只要在程序中运行fire.Fire()，即可使用命令行参数 python file <function> [args,] {--kwargs,}。fire 还支持更多的高级功能，具体请参考官方指南[2]。

在主程序main.py中主要包含四个函数，其中三个需要命令行执行，main.py的代码组织结构如下：

```
def train(**kwargs):
    '''
    训练
    '''
    pass
```

[1] https://github.com/google/python-fire

[2] https://github.com/google/python-fire/blob/master/doc/guide.md

```
def val(model, dataloader):
    '''
    计算模型在验证集上的准确率等信息，用以辅助训练
    '''
    pass

def test(**kwargs):
    '''
    测试（inference）
    '''
    pass

def help():
    '''
    打印帮助的信息
    '''
    print('help')

if __name__=='__main__':
    import fire
    fire.Fire()
```

根据 fire 的使用方法，可以通过`python main.py <function> --args=xx`的方式执行训练或者测试。

训练

训练的主要步骤如下：

- 定义网络
- 定义数据
- 定义损失函数和优化器
- 计算重要指标
- 开始训练
  - 训练网络
  - 可视化各种指标

– 计算在验证集上的指标

训练函数的代码如下：

```
def train(**kwargs):

    # 根据命令行参数更新配置
    opt.parse(kwargs)
    vis = Visualizer(opt.env)

    # step1: 模型
    model = getattr(models, opt.model)()
    if opt.load_model_path:
        model.load(opt.load_model_path)
    if opt.use_gpu: model.cuda()

    # step2: 数据
    train_data = DogCat(opt.train_data_root,train=True)
    val_data = DogCat(opt.train_data_root,train=False)
    train_dataloader = DataLoader(train_data,opt.batch_size,
                        shuffle=True,
                        num_workers=opt.num_workers)
    val_dataloader = DataLoader(val_data,opt.batch_size,
                        shuffle=False,
                        num_workers=opt.num_workers)

    # step3: 目标函数和优化器
    criterion = t.nn.CrossEntropyLoss()
    lr = opt.lr
    optimizer = t.optim.Adam(model.parameters(),
                             lr = lr,
                             weight_decay = opt.weight_decay)

    # step4: 统计指标：平滑处理之后的损失，还有混淆矩阵
    loss_meter = meter.AverageValueMeter()
    confusion_matrix = meter.ConfusionMeter(2)
    previous_loss = 1e100

    # 训练
```

```
for epoch in range(opt.max_epoch):

    loss_meter.reset()
    confusion_matrix.reset()

    for ii,(data,label) in enumerate(train_dataloader):

        # 训练模型参数
        input = Variable(data)
        target = Variable(label)
        if opt.use_gpu:
            input = input.cuda()
            target = target.cuda()
        optimizer.zero_grad()
        score = model(input)
        loss = criterion(score,target)
        loss.backward()
        optimizer.step()

        # 更新统计指标及可视化
        loss_meter.add(loss.data[0])
        confusion_matrix.add(score.data, target.data)

        if ii%opt.print_freq==opt.print_freq-1:
            vis.plot('loss', loss_meter.value()[0])

            # 如果需要的话，进入debug模式
            if os.path.exists(opt.debug_file):
                import ipdb;
                ipdb.set_trace()

    model.save()

    # 计算验证集上的指标及可视化
    val_cm,val_accuracy = val(model,val_dataloader)
    vis.plot('val_accuracy',val_accuracy)
    vis.log("epoch:{epoch},lr:{lr},loss:{loss},train_cm:{train_cm},
```

```
        val_cm:{val_cm}"
        .format(
                    epoch = epoch,
                    loss = loss_meter.value()[0],
                    val_cm = str(val_cm.value()),
                    train_cm=str(confusion_matrix.value()),
                    lr=lr))

        # 如果损失不再下降，则降低学习率
        if loss_meter.value()[0] > previous_loss:
            lr = lr * opt.lr_decay
            for param_group in optimizer.param_groups:
                param_group['lr'] = lr

        previous_loss = loss_meter.value()[0]
```

这里用到了 PyTorchNet[①]里的一个工具：meter。meter 提供了一些轻量级的工具，用于帮助用户快速统计训练过程中的一些指标。`AverageValueMeter` 能够计算所有数的平均值和标准差，可以用来统计一个 epoch 中损失的平均值。`confusionmeter` 用来统计分类问题中的分类情况，是一个比准确率更详细的统计指标。以表 6-1 所示为例，共有 50 张狗的图片，其中有 35 张被正确分类成了狗，还有 15 张被误判成猫；共有 100 张猫的图片，其中有 91 张被正确判为了猫，剩下 9 张被误判成狗。相较准确率等统计信息，混淆矩阵更能体现分类的结果，尤其是在样本比例不均衡的情况下。

表 6-1　混淆矩阵

样本	判为狗	判为猫
实际是狗	35	15
实际是猫	9	91

PyTorchNet 从 TorchNet[②]迁移而来，提供了很多有用的工具，但其目前的开发和文档都还不是很完善，本书不做过多讲解。

① https://github.com/pytorch/tnt

② https://github.com/torchnet/torchnet

验证

验证相对来说比较简单，但要注意需将模型置于验证模式（model.eval()），验证完成后还需要将其置回为训练模式（model.train()），这两句代码会影响BatchNorm和Dropout等层的运行模式。验证模型准确率的代码如下。

```
def val(model,dataloader):
    '''
    计算模型在验证集上的准确率等信息
    '''

    # 把模型设为验证模式
    model.eval()

    confusion_matrix = meter.ConfusionMeter(2)
    for ii, data in enumerate(dataloader):
        input, label = data
        val_input = Variable(input, volatile=True)
        val_label = Variable(label.long(), volatile=True)
        if opt.use_gpu:
            val_input = val_input.cuda()
            val_label = val_label.cuda()
        score = model(val_input)
        confusion_matrix.add(score.data.squeeze(), label.long())

    # 把模型恢复为训练模式
    model.train()

    cm_value = confusion_matrix.value()
    accuracy = 100. * (cm_value[0][0] + cm_value[1][1]) /\
                (cm_value.sum())
    return confusion_matrix, accuracy
```

测试

测试时，需要计算每个样本属于狗的概率，并将结果保存成 csv 文件。测试的代码与验证比较相似，但需要自己加载模型和数据。

```
def test(**kwargs):
    opt.parse(kwargs)

    # 模型
    model = getattr(models, opt.model)().eval()
    if opt.load_model_path:
        model.load(opt.load_model_path)
    if opt.use_gpu: model.cuda()

    # 数据
    train_data = DogCat(opt.test_data_root,test=True)
    test_dataloader = DataLoader(train_data,
                                 batch_size=opt.batch_size,
                                 shuffle=False,
                                 num_workers=opt.num_workers)

    results = []
    for ii,(data,path) in enumerate(test_dataloader):
        input = t.autograd.Variable(data,volatile = True)
        if opt.use_gpu: input = input.cuda()
        score = model(input)
        probability = t.nn.functional.softmax\
            (score)[:,1].data.tolist()
        batch_results = [(path_,probability_)
            for path_,probability_ in zip(path,probability) ]
        results += batch_results
    write_csv(results,opt.result_file)
    return results
```

帮助函数

为了方便他人使用，程序中还应提供一个帮助函数，用于说明函数是如何使用的。程序的命令行接口中有众多参数，如果手动用字符串表示不仅复杂，后期修改 config 文件时还需要修改对应的帮助信息，十分不便。这里使用了 Python 标准库中的 inspect 方法，可以自动获取 config 的源代码。help 的代码如下：

```
def help():
```

```
'''
打印帮助的信息: python file.py help
'''

print('''
usage : python {0} <function> [--args=value,]
<function> := train | test | help
example:
        python {0} train --env='env0701' --lr=0.01
        python {0} test --dataset='path/to/dataset/root/'
        python {0} help
avaiable args:'''.format(__file__))

from inspect import getsource
source = (getsource(opt.__class__))
print(source)
```

当用户执行python main.py help时，会打印如下帮助信息：

```
    usage : python main.py <function> [--args=value,]
    <function> := train | test | help
    example:
            python main.py train --env='env0701' --lr=0.01
            python main.py test --dataset='path/to/dataset/'
            python main.py help
    avaiable args:
class DefaultConfig(object):
    env = 'default' # visdom 环境
    model = 'AlexNet' # 使用的模型

    train_data_root = './data/train/' # 训练集存放路径
    test_data_root = './data/test1' # 测试集存放路径
    load_model_path = 'checkpoints/model.pth' # 加载预训练的模型

    batch_size = 128 # batch size
    use_gpu = True # user GPU or not
    num_workers = 4 # how many workers for loading data
    print_freq = 20 # print info every N batch
```

```
debug_file = '/tmp/debug'
result_file = 'result.csv' # 结果文件

max_epoch = 10
lr = 0.1 # initial learning rate
lr_decay = 0.95 # when val_loss increase, lr = lr*lr_decay
weight_decay = 1e-4 # 损失函数
```

6.1.9 使用

正如help函数的打印信息所述，可以通过命令行参数指定变量名。下面是三个使用例子，fire 会将包含“-”的命令行参数自动转成下画线“_”，也会将非数字的值转成字符串，所以--train-data-root=data/train和-- train_data_root ='data/train'是等价的。

```
# 训练模型
python main.py train
        --train-data-root=data/train/
        --load-model-path='checkpoints/resnet34_16:53:00.pth'
        --lr=0.005
        --batch-size=32
        --model='ResNet34'
        --max-epoch = 20

# 测试模型
python main.py test
        --test-data-root=data/test1
        --load-model-path='checkpoints/resnet34_00:23:05.pth'
        --batch-size=128
        --model='ResNet34'
        --num-workers=12

# 打印帮助信息
python main.py help
```

6.1.10 争议

以上的程序设计规范带有笔者强烈的个人喜好，并不能作为一个标准，而是作为一个提议和一种参考。上述设计在很多地方还有待商榷，例如训练过程中是否应该封装成一个trainer对象，或者直接封装到BaiscModule的train方法之中；对命令行参数的处理也有不少值得讨论之处。因此，不要将本章中的观点作为一个必须遵守的规范，而应该看作是一个参考。

本章中的设计可能会引起不少争议，其中比较值得商榷的部分主要有以下两个方面。

- 命令行参数的设置。目前大多数程序都是使用 Python 标准库中的argparse处理命令行参数的，也有些使用轻量级的click。这种处理对命令行的支持更完备，但根据笔者的经验，这种做法不够直观，并且代码量相对较多。例如argparse，每次增加一个命令行参数，都必须写如下代码：

```
parser.add_argument('-save-interval', type=int, default=500, help='how
many steps to wait before saving [default:500]')
```

在读者眼中，这种实现方式远不如一个专门的config.py来得直观和易用。尤其是对于使用 Jupyter notebook 或 IPython 等交互式调试的用户来说，argparse较难使用。

- 模型训练。有不少人喜欢将模型的训练过程集成于模型的定义之中，代码结构如下所示：

```
class MyModel(nn.Module):

    def __init__(self,opt):
        self.dataloader = Dataloader(opt)
        self.optimizer  = optim.Adam(self.parameters(),lr=0.001)
        self.lr = opt.lr
        self.model = make_model()

    def forward(self,input):
        pass

    def train_(self):
        # 训练模型
        for epoch in range(opt.max_epoch)
          for ii,data in enumerate(self.dataloader):
```

```
            train_epoch()

        model.save()

    def train_epoch(self):
        pass
```

或是专门设计一个Trainer对象，大概结构如下：

```
'''
code simplified from:
https://github.com/pytorch/pytorch/blob/master/torch/utils/trainer/
trainer.py
'''
import heapq
from torch.autograd import Variable

class Trainer(object):

    def __init__(self, model=None, criterion=None, optimizer=None,
    dataset=None):
        self.model = model
        self.criterion = criterion
        self.optimizer = optimizer
        self.dataset = dataset
        self.iterations = 0

    def run(self, epochs=1):
        for i in range(1, epochs + 1):
            self.train()

    def train(self):
        for i, data in enumerate(self.dataset, self.iterations + 1):
            batch_input, batch_target = data
            self.call_plugins('batch', i, batch_input, batch_target)
            input_var = Variable(batch_input)
            target_var = Variable(batch_target)
```

```
        plugin_data = [None, None]

        def closure():
            batch_output = self.model(input_var)
            loss = self.criterion(batch_output, target_var)
            loss.backward()
            if plugin_data[0] is None:
                plugin_data[0] = batch_output.data
                plugin_data[1] = loss.data
            return loss

        self.optimizer.zero_grad()
        self.optimizer.step(closure)

    self.iterations += i
```

还有一些人喜欢模仿 Keras 和 Scikit-learn 的设计，设计一个`fit`接口。对读者来说，这些处理方式很难说哪个更好或更差，找到最适合自己的方法才是最好的。

6.2 PyTorch Debug 指南

6.2.1 ipdb 介绍

很多初学者用 print 或 log 调试程序，这在小规模的程序下很方便。但是更好的调试方法是一边运行一边检查里面的变量和方法。Pdb 是一个交互式的调试工具，集成于 Python 标准库之中，由于其强大的功能，被广泛应用于 Python 环境中。Pdb 能让你根据需求跳转到任意的 Python 代码断点、查看任意变量、单步执行代码，甚至还能修改变量的值，而不必重启程序。ipdb 是一个增强版的 pdb，可通过`pip install ipdb`安装。ipdb 提供了调试模式下的代码自动补全，还具有更好的语法高亮和代码溯源，以及更好的内省功能，更关键的是，它与 pdb 接口完全兼容。

在本书第 2 章曾粗略地提到过 ipdb 的基本使用，本节将继续介绍如何结合 PyTorch 和 ipdb 进行调试。首先看一个例子，要使用 ipdb，只需在想要进行调试的地方插入`ipdb.set_trace()`，当代码运行到此处时，就会自动进入交互式调试模式。

假设有如下程序：

```
try:
    import ipdb
except:
    import pdb as ipdb

def sum(x):
    r = 0
    for ii in x:
        r += ii
    return r

def mul(x):
    r = 1
    for ii in x:
        r *= ii
    return r

ipdb.set_trace()
x = [1, 2, 3, 4, 5]
r = sum(x)
r = mul(x)
```

当程序运行至`ipdb.set_trace()`，会自动进入 debug 模式，在该模式中，我们可使用调试命令，如`next`或缩写`n`单步执行，也可查看 Python 变量，或是运行 Python 代码。如果 Python 变量名和调试命令冲突，需在变量名前加`!`，这样 ipdb 会执行对应的 Python 命令，而不是调试命令。下面举例说明 ipdb 的调试，这里重点讲解 ipdb 的两大功能。

- 查看：在函数调用堆栈中自由跳动，并查看函数的局部变量。
- 修改：修改程序中的变量，并能以此影响程序的运行结果。

```
> /tmp/mem2/debug.py(16)<module>()
     15 ipdb.set_trace()
---> 16 x = [1,2,3,4,5]
     17 r = sum(x)

ipdb> l 1,18 # list 1,18 的缩写，查看第1行到第18行的代码
             # 光标所指的这一行尚未运行
      1 import ipdb
      2
```

```
  3 def sum(x):
  4     r = 0
  5     for ii in x:
  6         r += ii
  7     return r
  8
  9 def mul(x):
 10     r = 1
 11     for ii in x:
 12         r *= ii
 13     return r
 14
 15 ipdb.set_trace()
---> 16 x = [1,2,3,4,5]
 17 r = sum(x)
 18 r = mul(x)

ipdb> n # next的缩写，执行下一步
> /tmp/mem2/debug.py(17)<module>()
     16 x = [1,2,3,4,5]
---> 17 r = sum(x)
     18 r = mul(x)

ipdb> s # step的缩写，进入sum函数内部
--Call--
> /tmp/mem2/debug.py(3)sum()
      2
----> 3 def sum(x):
      4     r = 0

ipdb> n # next单步执行
> /tmp/mem2/debug.py(4)sum()
      3 def sum(x):
----> 4     r = 0
      5     for ii in x:

ipdb> n # 单步执行
```

```
> /tmp/mem2/debug.py(5)sum()
      4     r = 0
----> 5     for ii in x:
      6         r += ii

ipdb> n # 单步执行
> /tmp/mem2/debug.py(6)sum()
      5     for ii in x:
----> 6         r += ii
      7     return r

ipdb> u # up的缩写，跳回上一层的调用
> /tmp/mem2/debug.py(17)<module>()
     16 x = [1,2,3,4,5]
---> 17 r = sum(x)
     18 r = mul(x)

ipdb> d # down的缩写，跳到调用的下一层
> /tmp/mem2/debug.py(6)sum()
      5     for ii in x:
----> 6         r += ii
      7     return r

ipdb> !r # 查看变量r的值,该变量名与调试命令`r(eturn)`冲突
0
ipdb> return # 继续运行直到函数返回
--Return--
15
> /tmp/mem2/debug.py(7)sum()
      6         r += ii
----> 7     return r
      8

ipdb> n # 下一步
> /tmp/mem2/debug.py(18)<module>()
     17 r = sum(x)
---> 18 r = mul(x)
```

```
     19

ipdb> x # 查看变量x
[1, 2, 3, 4, 5]
ipdb> x[0] = 10000 # 修改变量x
ipdb> b 10 # break的缩写，在第10行设置断点
Breakpoint 1 at /tmp/mem2/debug.py:10

ipdb> c # continue的缩写，继续运行，直到遇到断点
> /tmp/mem2/debug.py(10)mul()
      9 def mul(x):
1--> 10     r = 1
     11     for ii in x:

ipdb> return # 可见计算的是修改之后的x的乘积
--Return--
1200000
> /tmp/mem2/debug.py(13)mul()
     12         r *= ii
---> 13     return r
     14

ipdb> q # 退出debug
```

关于 ipdb 的使用还有一些技巧：

- `<tab>`键能够自动补齐，补齐用法与 IPython 中的类似。
- `j(ump) <lineno>` 能够跳过中间某些行代码的执行。
- 可以直接在 ipdb 中修改变量的值。
- `h(elp)`能够查看调试命令的用法，比如`h h`可以查看`h(elp)`命令的用法，`h jump`能够查看`j(ump)`命令的用法。

6.2.2 在 PyTorch 中 Debug

PyTorch 作为一个动态图框架，与 ipdb 结合使用能为调试过程带来便捷。对 TensorFlow 等静态图框架来说，使用 Python 接口定义计算图，然后使用C++代码执行底层

运算，在定义图的时候不进行任何计算，而在计算的时候又无法使用 pdb 进行调试，因为 pdb 调试只能调试 Python 代码，故调试一直是此类静态图框架的一个痛点。与 TensorFlow 不同，PyTorch 可以在执行计算的同时定义计算图，这些计算定义过程是使用 Python 完成的。虽然底层的计算也是用 C/C++ 完成的，但是我们能够查看 Python 定义部分的变量值，这就已经足够了。下面我们将举例说明：

- 如何在 PyTorch 中查看神经网络各个层的输出。
- 如何在 PyTorch 中分析各个参数的梯度。
- 如何动态修改 PyTorch 的训练流程。

首先，运行 6.2.1 节所给的示例程序：

```
python main.py train
            --train-data-root=data/train/
            --lr=0.005
            --batch-size=8
            --model='AlexNet'
            --load-model-path=None
            --debug-file='/tmp/debug'
```

程序运行一段时间后，可通过`touch /tmp/debug`创建 debug 标识文件，当程序检测到这个文件的存在时，会自动进入 debug 模式。

```
> /home/x/data/cy/best_practice/main.py(81)train()
     80
---> 81  for ii,(data,label) in enumerate(train_dataloader):
     82

ipdb> l 90 # 查看第90行附近的代码
     85             target = Variable(label)
     86             if opt.use_gpu:
     87                 input = input.cuda()
     88                 target = target.cuda()
     89
     90             optimizer.zero_grad()
     91             score = model(input)
     92             loss = criterion(score,target)
     93             loss.backward()
     94             optimizer.step()
```

```
     95

ipdb> break 91 # 在第91行设置断点，当程序运行到此处进入debug模式
Breakpoint 1 at /home/x/data/cy/best_practice/main.py:91
ipdb># 打印所有参数及其梯度的标准差
     for (name,p) in model.named_parameters():\
             print(name,p.data.std(),p.grad.data.std())
('features.0.weight', 0.036, 0.0)
('features.0.bias', 0.029, 0.0)
('features.3.weight', 0.020, 0.0)
('features.3.bias', 0.0192, 0.0)
('features.6.weight', 0.0185, 0.0)
('features.6.bias', 0.041, 0.0)
('features.8.weight', 0.020, 0.0)
('features.8.bias', 0.031, 0.0)
('features.10.weight', 0.015, 0.0)
('features.10.bias', 0.021, 0.0)
('classifier.1.weight', 0.0138, 0.0)
('classifier.1.bias', 0.050, 0.001)
('classifier.4.weight', 0.023, 1.03e-05)
('classifier.4.bias', 0.024, 0.0496)
('classifier.6.weight', 0.021, 0.0051)
('classifier.6.bias', 0.014, 0.3294)
ipdb> opt.lr # 查看变量（学习率）
0.005
ipdb> opt.lr = 0.001 # 修改变量值（学习率）
ipdb> for p in optimizer.param_groups:\ # 修改学习率
             p['lr']=opt.lr
ipdb> model.save() # 保存模型
'checkpoints/alexnet_0713_19:24:40.pth'
ipdb> c # 继续运行，直到第91行暂停
> /home/x/data/cy/best_practice/main.py(91)train()
     90                 optimizer.zero_grad()
2--> 91                 score = model(input)
     92                 loss = criterion(score,target)

ipdb> s # 进入model(input)内部，即model.__call__(input)
```

```
--Call--
> torch/nn/modules/module.py(205)__call__()
    204
--> 205     def __call__(self, *input, **kwargs):
    206         result = self.forward(*input, **kwargs)

ipdb> n # 下一步
> torch/nn/modules/module.py(206)__call__()
    205     def __call__(self, *input, **kwargs):
--> 206         result = self.forward(*input, **kwargs)
    207         for hook in self._forward_hooks.values():

ipdb> s # 进入forward函数内部
--Call--
> /home/x/data/cy/best_practice/models/AlexNet.py(41)forward()
     40
---> 41     def forward(self, x):
     42         x = self.features(x)

ipdb> n # 下一步
> /home/x/data/cy/best_practice/models/AlexNet.py(42)forward()
     41     def forward(self, x):
---> 42         x = self.features(x)
     43         x = x.view(x.size(0), 256 * 6 * 6)

ipdb> n # 下一步
> /home/x/data/cy/best_practice/models/AlexNet.py(43)forward()
     42         x = self.features(x)
---> 43         x = x.view(x.size(0), 256 * 6 * 6)
     44         x = self.classifier(x)

ipdb> x.data.mean(),x.data.std() # 查看features层的输出
# 读者还可以接着step into self.features
# 查看每一个卷积层、激活层、Pooling层的输出
(0.0012, 0.123)
ipdb> u  # 跳回上一层
> torch/nn/modules/module.py(206)__call__()
```

```
    204
    205     def __call__(self, *input, **kwargs):
--> 206         result = self.forward(*input, **kwargs)
    207         for hook in self._forward_hooks.values():
    208             hook_result = hook(self, input, result)

ipdb> u # 跳回上一层
> /home/x/data/cy/best_practice/main.py(91)train()
     89
     90             optimizer.zero_grad()
---> 91             score = model(input)
     92             loss = criterion(score,target)
     93             loss.backward()
ipdb> clear # 清除所有断点
Clear all breaks? y
ipdb> c # 继续运行，记得先删除`/tmp/debug`，否则很快又会进入调和模式
```

当我们想要进入 debug 模式，修改程序中某些参数值或者想分析程序时，就可以通过`touch /tmp/debug`命令创建 debug 标识文件，此时程序会进入调试模式，调试完成之后删除这个文件并在 ipdb 调试接口输入`c`继续运行程序。如果想退出程序，也可以使用这种方法，先创建`/tmp/debug`文件使程序进入调试模式，然后输入`quit`在退出 debug 的同时退出程序。这种退出程序的方法，与使用`Ctrl+C`的方法相比更安全，因为这能保证数据加载的多进程（multiprocessing）程序也能正确的退出，并释放内存、显存等资源。

PyTorch 和 ipdb 结合能完成很多其他框架所不能完成或很难实现的功能。根据笔者日常使用的总结，主要有以下几个部分。

（1）通过 debug 暂停程序。当程序进入 debug 模式之后，将不再执行 GPU 和 CPU 运算，但是内存和显存及相应的堆栈空间不会释放。

（2）通过 debug 分析程序，查看每个层的输出，查看网络的参数情况。通过`u(p)`、`d(own)`、`s(tep)`等命令，能够进入指定的代码，通过`n(ext)`可以单步执行，从而看到每一层的运算结果，便于分析网络的数值分布等信息。

（3）作为动态图框架，PyTorch 拥有 Python 动态语言解释执行的优点，我们能够在运行程序时，通过 ipdb 修改某些变量的值或属性，这些修改能够立即生效。例如可以在训练开始不久根据损失函数调整学习率，不必重启程序。

（4）如果在 IPython 中通过`%run`魔法方法运行程序，那么在程序异常退出时，可以使用`%debug`命令，直接进入 debug 模式，通过`u(p)`和`d(own)`调到报错的地方，查看对应

的变量。找出原因后修改相应的代码即可。有时我们的模型训练了好几个小时，却在将要保存模型之前，因为一个小小的拼写错误异常退出。此时，如果修改错误再重新运行程序又要花费好几个小时，太浪费时间。因此最好的方法就是利用`%debug`进入调试模式，在调试模式中直接运行`model.save()`保存模型。在 IPython 中，`%pdb`魔术方法能够使得程序出现问题后，不用手动输入`%debug`而自动进入 debug 模式，建议使用。

PyTorch 调用 CuDNN 报错时，报错信息诸如 `CUDNN_STATUS_BAD_PARAM`，从这些报错内容很难得到有用的帮助信息，最好先利用 CPU 运行代码，此时一般会得到相对友好的报错信息，例如在 ipdb 中执行`model.cpu()(input.cpu())`，PyTorch 底层的 TH 库会给出相对比较详细的信息。

常见的错误主要有以下几种：

- 类型不匹配问题。例如`CrossEntropyLoss`的输入 target 应该是一个`LongTensor`，而很多人输入`FloatTensor`。
- 部分数据忘记从 CPU 转移到 GPU。例如，当`model`存放于 GPU 时，输入`input`也需要转移到 GPU 才能输入到 model 中。还有可能就是把多个 module 存放于一个 list 对象，而在执行`model.cuda()`时，这个 list 中的对象是不会被转移到 CUDA 上的，正确的用法是用`ModuleList`代替。
- Tensor 形状不匹配。此类问题一般是输入数据形状不对，或是网络结构设计有问题，一般通过`u(p)`跳到指定代码，查看输入和模型参数的形状即可得知。

此外，可能还会经常遇到程序正常运行、没有报错，但是模型无法收敛的问题。例如对于二分类问题，交叉熵损失一直徘徊在 0.69 附近（$\ln 2$），或者是数值出现溢出等问题，此时可以进入 debug 模式，用单步执行看看每一层输出的均值和方差，观察从哪一层的输出开始出现数值异常。还要查看每个参数梯度的均值和方差，看看是否出现梯度消失或者梯度爆炸等问题。一般来说，通过在激活函数之前增加`BatchNorm`层、合理的参数初始化、使用`Adam`优化器、学习率设为 0.001，基本就能确保模型在一定程度收敛。

本章带领读者从头完成了一个 Kaggle 上的经典竞赛，重点讲解了如何合理地组织安排程序，同时介绍了一些在 PyTorch 中调试的技巧。

7 AI 插画师：生成对抗网络

生成对抗网络（Generative Adversarial Net，GAN）是近年来深度学习中一个十分热门的方向，卷积网络之父、深度学习元老级人物 LeCun Yan 就曾说过“GAN is the most interesting idea in the last 10 years in machine learning”。尤其是近两年，GAN 的论文呈现井喷的趋势，GitHub[①]上有人收集了各种各样的 GAN 变种、应用、研究论文等，其中有名称的多达数百篇。作者还统计了 GAN 论文发表数目随时间变化的趋势，如图 7-1 所示，足见 GAN 的火爆程度。本节将简要介绍 GAN 的基本原理，并带领读者实现一个简单的生成对抗网络，用以生成动漫人物的头像。

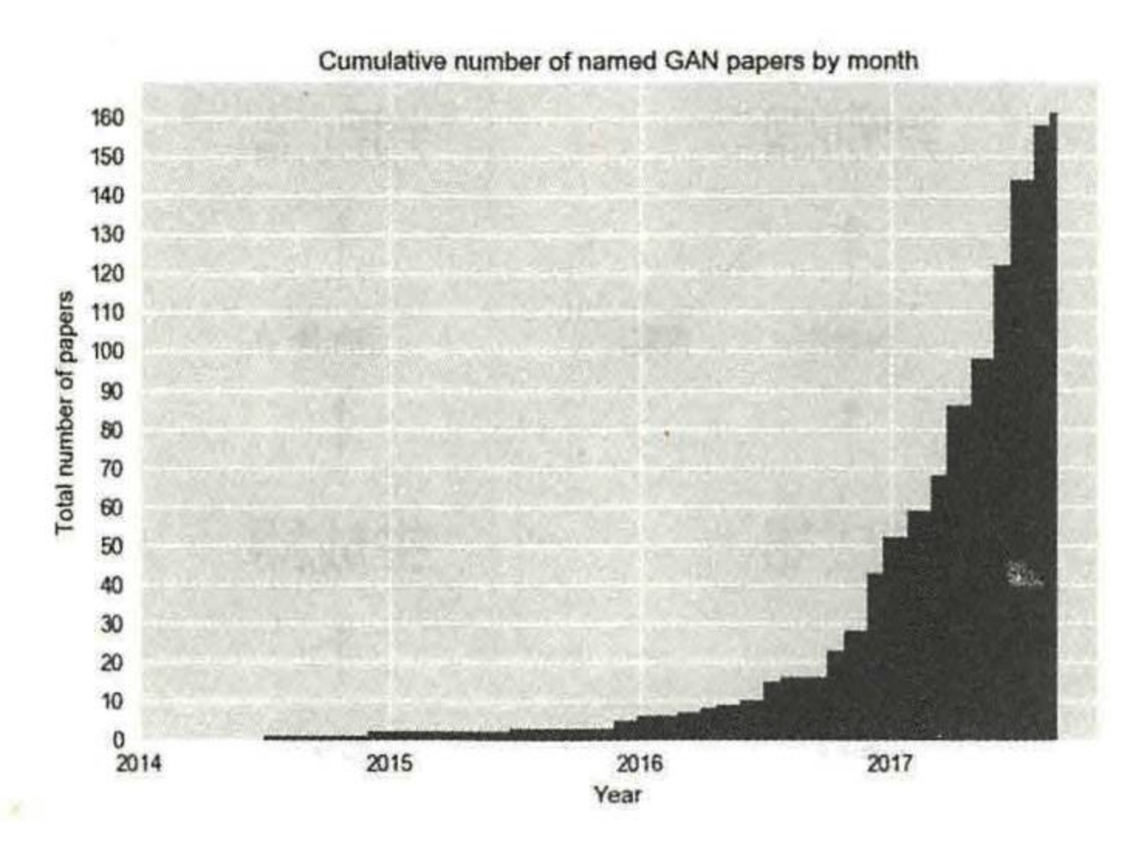


图 7-1　GAN 的论文数目逐月累加图

① https://github.com/hindupuravinash/the-gan-zoo

7.1 GAN 的原理简介

GAN 的开山之作是被称为“GAN 之父”的 Ian Goodfellow 发表于 2014 年的经典论文 *Generative Adversarial Networks*[①]，在这篇论文中他提出了生成对抗网络，并设计了第一个 GAN 实验——手写数字生成。

GAN 的产生来自于一个灵机一动的想法：

> “What I cannot create, I do not understand.”（那些我所不能创造的，我也没有真正地理解它。）
>
> —Richard Feynman

类似地，如果深度学习不能创造图片，那么它也没有真正地理解图片。当时深度学习已经开始在各类计算机视觉领域中攻城略地，在几乎所有任务中都取得了突破。但是人们一直对神经网络的黑盒模型表示质疑，于是越来越多的人从可视化的角度探索卷积网络所学习的特征和特征间的组合，而 GAN 则从生成学习角度展示了神经网络的强大能力。GAN 解决了非监督学习中的著名问题：**给定一批样本，训练一个系统能够生成类似的新样本。**

生成对抗网络的网络结构如图 7-2 所示，主要包含以下两个子网络。

- 生成器（generator）：输入一个随机噪声，生成一张图片。
- 判别器（discriminator）：判断输入的图片是真图片还是假图片。

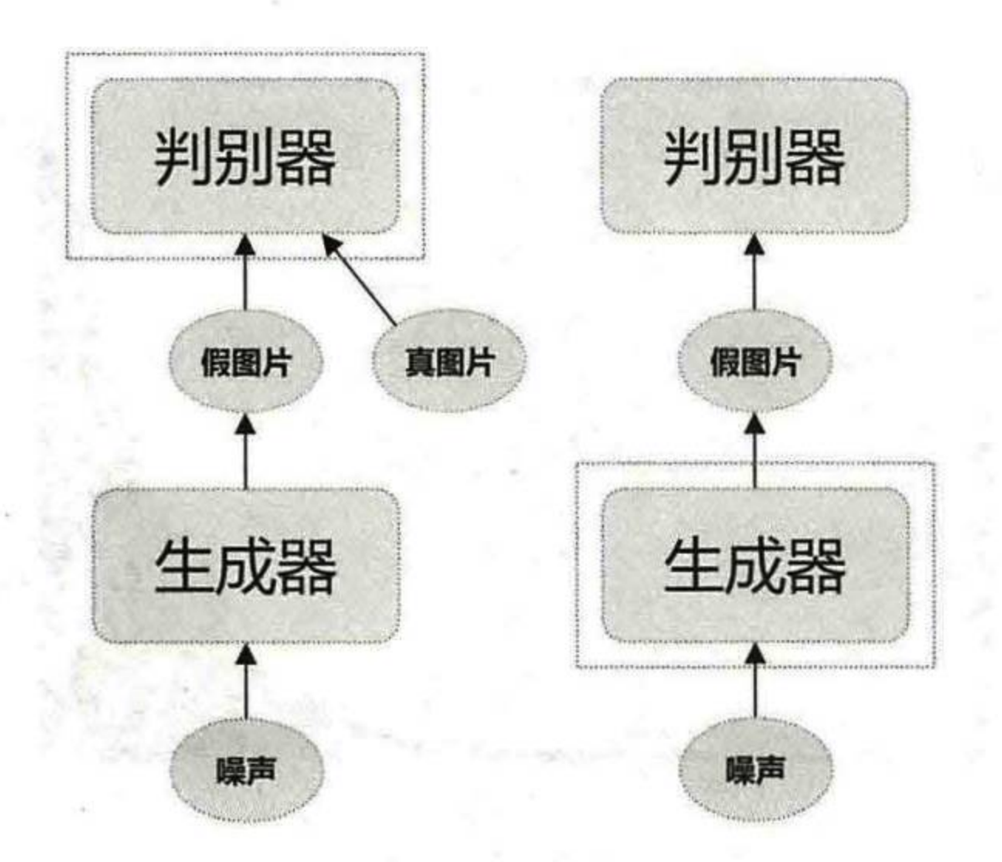


图 7-2　生成对抗网络结构图

[①] Goodfellow, Ian, et al. “Generative adversarial nets.” Advances in Neural Information Processing Systems. 2014.

训练判别器时，需要利用生成器生成的假图片和来自真实世界的真图片；训练生成器时，只用噪声生成假图片。判别器用来评估生成的假图片的质量，促使生成器相应地调整参数。

生成器的目标是尽可能地生成以假乱真的图片，让判别器以为这是真的图片；判别器的目标是将生成器生成的图片和真实世界的图片区分开。可以看出这二者的目标相反，在训练过程中互相对抗，这也是它被称为生成对抗网络的原因。

上面的描述可能有点抽象，让我们用收藏齐白石作品（齐白石作品如图 7-3 所示）的书画收藏家和假画贩子的例子来说明。假画贩子相当于是生成器，他们希望能够模仿大师真迹伪造出以假乱真的假画，骗过收藏家，从而卖出高价；书画收藏家则希望将赝品和真迹区分开，让真迹流传于世，销毁赝品。这里假画贩子和收藏家所交易的画，主要是齐白石画的虾。齐白石画虾可以说是画坛一绝，历来为世人所追捧。

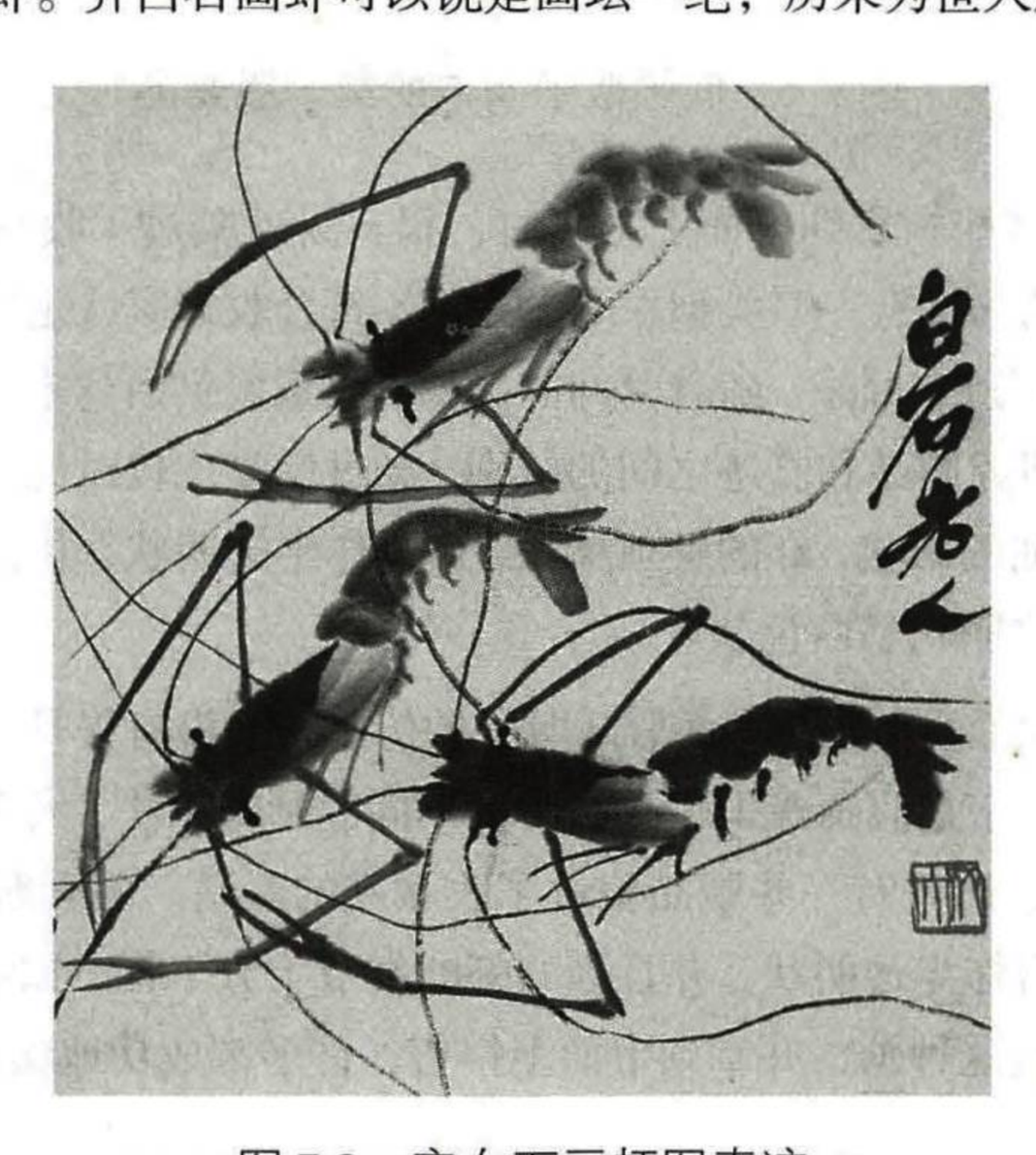


图 7-3　齐白石画虾图真迹

在这个例子中，一开始假画贩子和书画收藏家都是新手，他们对真迹和赝品的概念都很模糊。假画贩子仿造出来的假画几乎都是随机涂鸦，而书画收藏家的鉴定能力很差，有不少赝品被他当成真迹，也有许多真迹被当成赝品。

首先，书画收藏家收集了一大堆市面上的赝品和齐白石大师的真迹，仔细研究对比，初步学习了画中虾的结构，明白画中的生物形状弯曲，并且有一对类似钳子的“螯足”，对于不符合这个条件的假画全部过滤掉。当收藏家用这个标准到市场上进行鉴定时，假画基本无法骗过收藏家，假画贩子损失惨重。但是假画贩子自己仿造的赝品中，还是有一些蒙骗过关，这些蒙骗过关的赝品中都有弯曲的形状，并且有一对类似钳子

的“螯足”。于是假画贩子开始修改仿造的手法，在仿造的作品中加入弯曲的形状和一对类似钳子的“螯足”。除了这些特点，其他地方例如颜色、线条都是随机画的。假画贩子制造出的第一版赝品如图 7-4 所示。

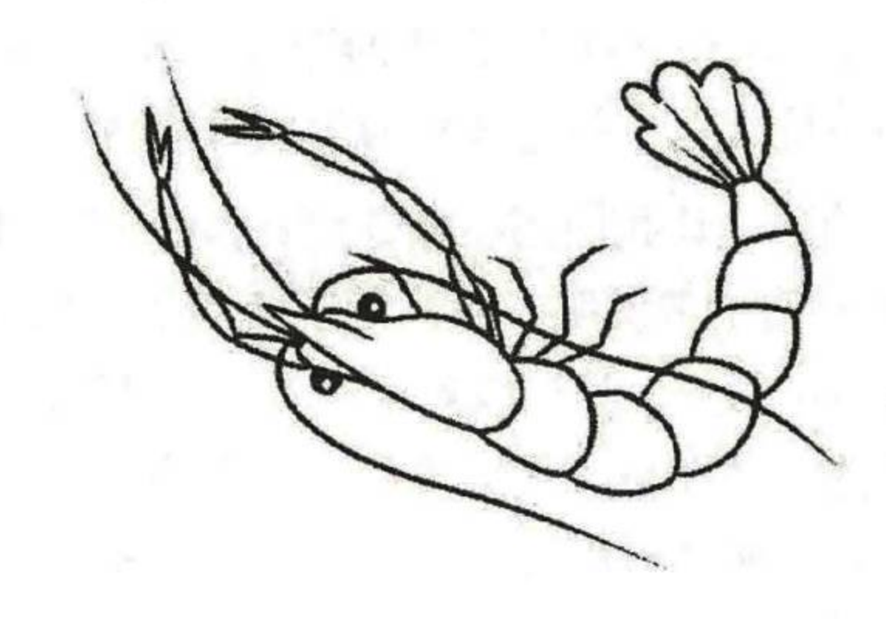


图 7-4　假画贩子制造的第一版赝品

当假画贩子把这些画拿到市面上去卖时，很容易就骗过了收藏家，因为画中有一只弯曲的生物，生物前面有一对类似钳子的东西，符合收藏家认定的真迹的标准，所以收藏家就把它当成真迹买回来。随着时间的推移，收藏家买回越来越多的假画，损失惨重，于是他又闭门研究赝品和真迹之间的区别，经过反复比较对比，他发现齐白石画虾的真迹中除了有弯曲的形状，虾的触须蔓长，通身作半透明状，并且画的虾的细节十分丰富，虾的每一节之间均呈白色状。

收藏家学成之后，重新出山，而假画贩子的仿造技法没有提升，所制造出来的赝品被收藏家轻松识破。于是假画贩子也开始尝试不同的画虾手法，大多都是徒劳无功，不过在众多尝试之中，还是有一些赝品骗过了收藏家的眼睛。假画贩子发现这些仿制的赝品触须蔓长，通身作半透明状，并且画的虾的细节十分丰富，如图 7-5 所示。于是假画贩子开始大量仿造这种画，并拿到市面上销售，许多都成功地骗过了收藏家。

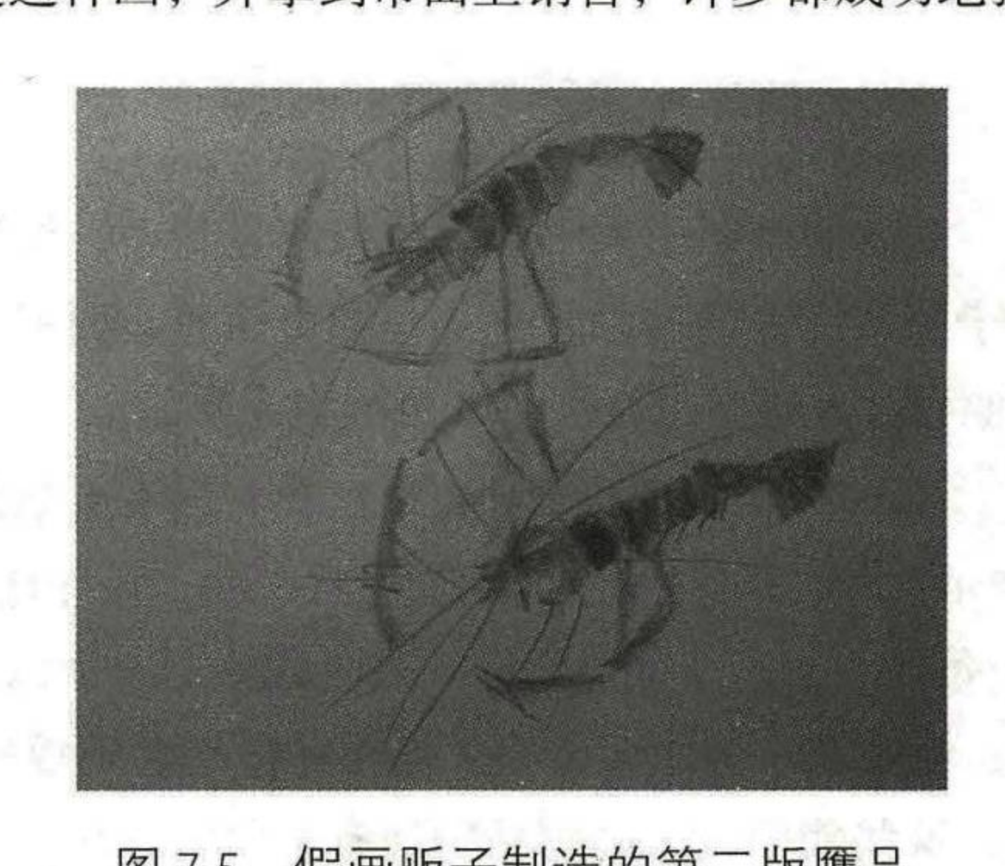


图 7-5　假画贩子制造的第二版赝品

收藏家再度损失惨重，被迫关门研究齐白石的真迹和赝品之间的区别，学习齐白石真迹的特点，提升自己的鉴定能力。就这样，通过收藏家和假画贩子之间的博弈，收藏家从零开始慢慢提升了自己对真迹和赝品的鉴别能力，而假画贩子也不断地提高自己仿造齐白石真迹的水平。收藏家利用假画贩子提供的赝品，作为和真迹的对比，对齐白石画虾真迹有了更好的鉴赏能力；而假画贩子也不断尝试，提升仿造水平，提升仿造假画的质量，即使最后制造出来的仍属于赝品，但是和真迹相比也很接近了。收藏家和假画贩子二者之间互相博弈对抗，同时又不断促使着对方学习进步，达到共同提升的目的。

在这个例子中，假画贩子相当于一个生成器，收藏家相当于一个判别器。一开始生成器和判别器的水平都很差，因为二者都是随机初始化的。训练过程分为两步交替进行，第一步是训练判别器（只修改判别器的参数，固定生成器），目标是把真迹和赝品区分开；第二步是训练生成器（只修改生成器的参数，固定判别器），为的是生成的假画能够被判别器判别为真迹（被收藏家认为是真迹）。这两步交替进行，进而分类器和判别器都达到了一个很高的水平。训练到最后，生成器生成的虾的图片（如图 7-6 所示）和齐白石的真迹几乎没有差别。

图 7-6　生成器生成的虾

下面我们来思考网络结构的设计。判别器的目标是判断输入的图片是真迹还是赝品，所以可以看成是一个二分类网络，参考第 6 章中 Dog vs. Cat 的实验，我们可以设计一个简单的卷积网络。生成器的目标是从噪声中生成一张彩色图片，这里我们采用广泛使用的 DCGAN（Deep Convolutional Generative Adversarial Networks）结构，即采用全卷积网络，其结构如图 7-7 所示。网络的输入是一个 100 维的噪声，输出是一个 $3 \times 64 \times 64$ 的图片。这里的输入可以看成是一个 $100 \times 1 \times 1$ 的图片，通过上卷积慢慢增大为 4×4 、8×8、16×16、32×32 和 64×64。上卷积，或称转置卷积，是一种特殊的卷积操作，类似于卷积操作的逆运算。当卷积的 stride 为 2 时，输出相比输入会下采样到一半的尺寸；而当上卷积的 stride 为 2 时，输出会上采样到输入的两倍尺寸。

这种上采样的做法可以理解为图片的信息保存于 100 个向量之中，神经网络根据这 100 个向量描述的信息，前几步的上采样先勾勒出轮廓、色调等基础信息，后几步上采样慢慢完善细节。网络越深，细节越详细。

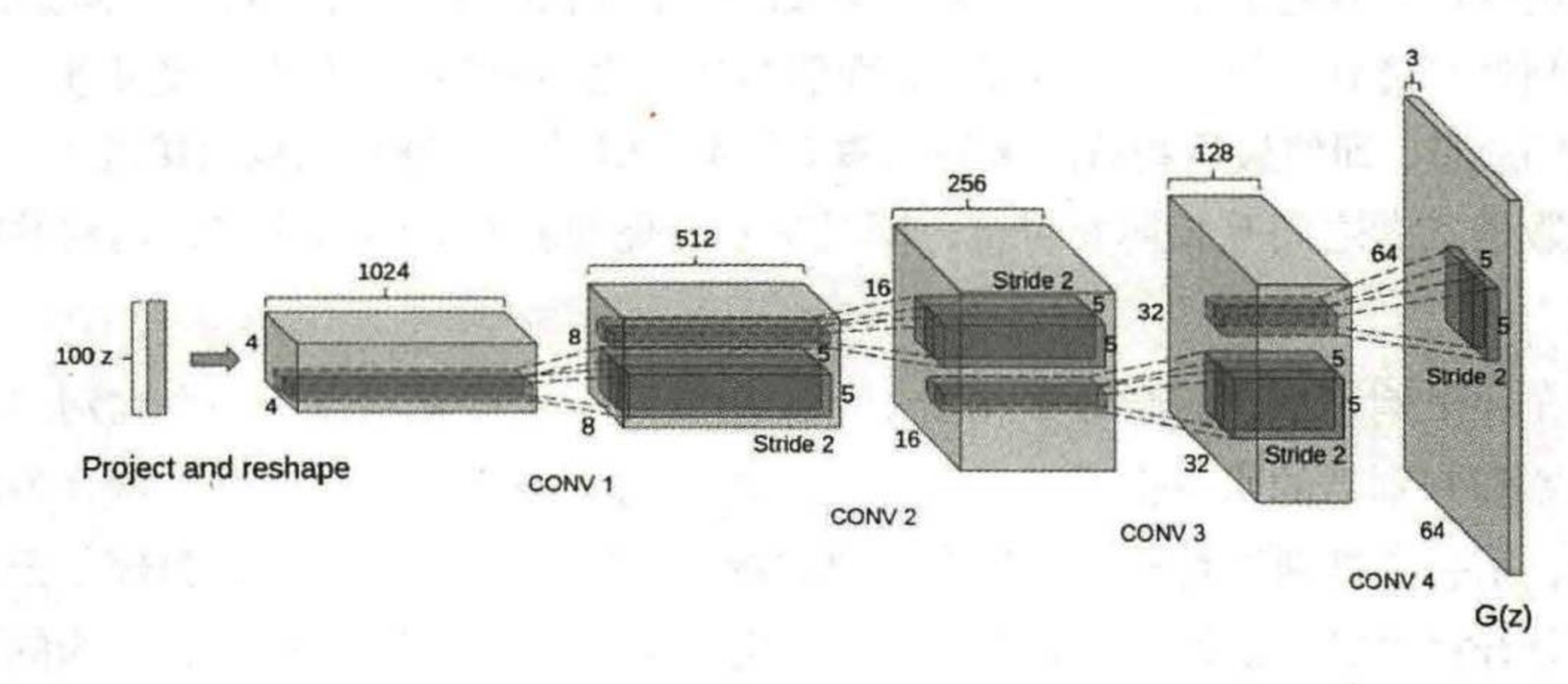


图 7-7　DCGAN 中生成器网络结构图

在 DCGAN 中，判别器的结构和生成器对称：生成器中采用上采样的卷积，判别器中就采用下采样的卷积，生成器是根据噪声输出一张 $64 \times 64 \times 3$ 的图片，而判别器则是根据输入的 $64 \times 64 \times 3$ 的图片输出图片属于正负样本的分数（概率）。

7.2　用 GAN 生成动漫头像

本节将用 GAN 实现一个生成动漫人物头像的例子。在日本的技术博客网站上[①]有个博主（估计是一位二次元的爱好者），利用 DCGAN 从 20 万张动漫头像中学习，最终能够利用程序自动生成动漫头像，生成的图片效果如图 7-8 所示。源程序是利用 Chainer 框架实现的，本节我们尝试利用 PyTorch 实现。

原始的图片是从网站中爬取的，并利用 OpenCV 从中截取头像，处理起来比较麻烦。这里我们使用知乎用户何之源爬取并经过处理的 5 万张图片。可以从本书配套程序的 README.MD 的百度网盘链接下载所有的图片压缩包，并解压缩到指定的文件夹中。需要注意的是，这里图片的分辨率是 $3 \times 96 \times 96$，而不是论文中的 $3 \times 64 \times 64$，因此需要相应地调整网络结构，使生成图像的尺寸为 96。

我们首先来看本实验的代码结构。

```
checkpoints/ # 无代码，用来保存模型
imgs/ # 无代码，用来保存生成的图片
```

[①] http://qiita.com/mattya/items/e5bfe5e04b9d2f0bbd47

```
data/ # 无代码，用来保存训练所需的图片
main.py # 训练和生成
model.py # 模型定义
visualize.py # 可视化工具visdom的封装
requiments.txt # 程序中用到的第三方库
README.MD # 说明
```

图 7-8　DCGAN 生成的动漫头像

接着来看model.py中是如何定义生成器的。

```
class NetG(nn.Module):
    '''
    生成器定义
    '''
    def __init__(self, opt):
        super(NetG, self).__init__()
        ngf = opt.ngf # 生成器feature map数
```

```
        self.main = nn.Sequential(
            # 输入是nz维度的噪声，可认为其是一个nz*1*1的feature map
            nn.ConvTranspose2d( opt.nz , ngf * 8, 4, 1, 0, bias=False),
            nn.BatchNorm2d(ngf * 8),
            nn.ReLU(True),
            # 上一步的输出形状：(ngf*8) x 4 x 4

            nn.ConvTranspose2d(ngf * 8, ngf * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf * 4),
            nn.ReLU(True),
            # 上一步的输出形状： (ngf*4) x 8 x 8

            nn.ConvTranspose2d(ngf * 4, ngf * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf * 2),
            nn.ReLU(True),
            # 上一步的输出形状： (ngf*2) x 16 x 16

            nn.ConvTranspose2d(ngf * 2, ngf, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf),
            nn.ReLU(True),
            # 上一步的输出形状：(ngf) x 32 x 32

            nn.ConvTranspose2d(ngf, 3, 5, 3, 1, bias=False),
            nn.Tanh()
            # 输出形状：3 x 96 x 96
        )

    def forward(self, input):
        return self.main(input)
```

可以看出生成器的搭建相对比较简单，直接使用`nn.Sequential`将上卷积、激活、池化等操作拼接起来即可，这里需要注意上卷积`ConvTransposed2d`的使用。当 kernel size 为 4、stride 为 2、padding 为 1 时，根据公式 $H_{\text{out}} = (H_{\text{in}} - 1) * \text{stride} - 2 * \text{padding} + \text{kernel_size}$，输出尺寸刚好变成输入的两倍。最后一层采用 kernel size 为 5、stride 为 3、padding 为 1，是为了将 32×32 上采样到 96×96，这是本例中图片的尺寸，与论文中 64×64 的尺寸不一样。最后一层用`Tanh`将输出图片的像素归一化至`-1~1`，如果希望归一化至`0~1`，

则需使用Sigmoid。

接着我们来看判别器的网络结构。

```
class NetD(nn.Module):
    '''
    判别器定义
    '''
    def __init__(self, opt):
        super(NetD, self).__init__()
        ndf = opt.ndf
        self.main = nn.Sequential(
            # 输入 3 x 96 x 96
            nn.Conv2d(3, ndf, 5, 3, 1, bias=False),
            nn.LeakyReLU(0.2, inplace=True),
            # 输出 (ndf) x 32 x 32

            nn.Conv2d(ndf, ndf * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 2),
            nn.LeakyReLU(0.2, inplace=True),
            # 输出 (ndf*2) x 16 x 16

            nn.Conv2d(ndf * 2, ndf * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 4),
            nn.LeakyReLU(0.2, inplace=True),
            # 输出 (ndf*4) x 8 x 8

            nn.Conv2d(ndf * 4, ndf * 8, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 8),
            nn.LeakyReLU(0.2, inplace=True),
            # 输出 (ndf*8) x 4 x 4

            nn.Conv2d(ndf * 8, 1, 4, 1, 0, bias=False),
            nn.Sigmoid() # 输出一个数(概率)
        )

    def forward(self, input):
        return self.main(input).view(-1)
```

可以看出判别器和生成器的网络结构几乎是对称的，从卷积核大小到 padding、stride 等设置，几乎一模一样。例如生成器的最后一个卷积层的尺度是（5，3，1），判别器的第一个卷积层的尺度也是（5，3，1）。另外，这里需要注意的是生成器的激活函数用的是ReLU，而判别器使用的是LeakyReLU，二者并无本质区别，这里的选择更多是经验总结。每一个样本经过判别器后，输出一个 0~1 的数，表示这个样本是真图片的概率。

在开始写训练函数前，先来看看模型的配置参数。

```
class Config(object):

    data_path = 'data/' # 数据集存放路径
    num_workers = 4 # 多进程加载数据所用的进程数
    image_size = 96 # 图片尺寸
    batch_size = 256
    max_epoch =  200
    lr1 = 2e-4 # 生成器的学习率
    lr2 = 2e-4 # 判别器的学习率
    beta1 = 0.5 # Adam优化器的beta1参数
    use_gpu = True # 是否使用GPU
    nz = 100 # 噪声维度
    ngf = 64 # 生成器feature map数
    ndf = 64 # 判别器feature map数

    save_path = 'imgs/' # 生成图片保存路径

    vis = True # 是否使用visdom可视化
    env = 'GAN' # visdom的env
    plot_every = 20 # 每间隔20batch，visdom画图一次

    debug_file = '/tmp/debuggan' # 存在该文件则进入debug模式
    d_every = 1 # 每1个batch训练一次判别器
    g_every = 5 # 每5个batch训练一次生成器
    decay_every = 10 # 每10个epoch保存一次模型
    netd_path = 'checkpoints/netd_211.pth' # 预训练模型
    netg_path = 'checkpoints/netg_211.pth'

    # 测试时用的参数
```

```
    gen_img = 'result.png'
    # 从512张生成的图片中保存最好的64张
    gen_num = 64
    gen_search_num = 512
    gen_mean = 0 # 噪声的均值
    gen_std = 1 # 噪声的方差
opt = Config()
```

这些只是模型的默认参数，还可以利用 Fire 等工具通过命令行传入，覆盖默认值。另外，我们也可以直接使用`opt.attr`，还可以利用 IDE/IPython 提供的自动补全功能，十分方便。这里的超参数设置大多是照搬 DCGAN 论文的默认值，作者经过大量实验，发现这些参数能够更快地训练出一个不错的模型。

当我们下载完数据之后，需要将所有图片放在一个文件夹，然后将该文件夹移动至 data 目录下（请确保`data`下没有其他的文件夹）。这种处理方式是为了能够直接使用`torchvision`自带的`ImageFolder`读取图片，而不必自己写`Dataset`。数据读取与加载的代码如下：

```
# 数据处理，输出规整为-1~1
transforms = tv.transforms.Compose([
                tv.transforms.Scale(opt.image_size),
                tv.transforms.CenterCrop(opt.image_size),
                tv.transforms.ToTensor(),
                tv.transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
                                    ])
dataset = tv.datasets.ImageFolder(opt.data_path, transform=transforms)
dataloader = t.utils.data.DataLoader(dataset,
                                     batch_size = opt.batch_size,
                                     shuffle = True,
                                     num_workers = opt.num_workers,
                                     drop_last = True
                                     )
```

可见，用`ImageFolder`配合`DataLoader`加载图片十分方便。

在进行训练之前，我们还需要定义几个变量：模型、优化器、噪声等。

```
# 定义网络
map_location = lambda storage, loc: storage
if opt.netd_path:
```

```
    netd.load_state_dict(t.load(opt.netd_path, map_location = map_location
    ))
if opt.netg_path:
    netg.load_state_dict(t.load(opt.netg_path, map_location = map_location
    ))

# 定义优化器和损失
optimizer_g = t.optim.Adam(netg.parameters(), opt.lr1, betas=(opt.beta1,
0.999))
optimizer_d = t.optim.Adam(netd.parameters(), opt.lr2, betas=(opt.beta1,
0.999))
criterion = t.nn.BCELoss()

# 真图片label为1，假图片label为0，noises为生成网络的输入噪声
true_labels = Variable(t.ones(opt.batch_size))
fake_labels = Variable(t.zeros(opt.batch_size))
fix_noises = Variable(t.randn(opt.batch_size, opt.nz, 1, 1))
noises = Variable(t.randn(opt.batch_size, opt.nz, 1, 1))

# 如果使用GPU训练，将数据移至GPU上
if opt.use_gpu:
    netd.cuda()
    netg.cuda()
    criterion.cuda()
    true_labels, fake_labels = true_labels.cuda(), fake_labels.cuda()
    fix_noises, noises = fix_noises.cuda(), noises.cuda()
```

在加载预训练模型时，最好指定`map_location`。因为如果程序之前在 GPU 上运行，那么模型就会被存成`torch.cuda.Tensor`，这样加载时会默认将数据加载至显存。如果运行该程序的计算机中没有 GPU，加载就会报错，故通过指定`map_location`将 Tensor 默认加载入内存中，待有需要时再移至显存中。

下面开始训练网络，训练步骤如下。

（1）训练判别器。

- 固定生成器
- 对于真图片，判别器的输出概率值尽可能接近 1
- 对于生成器生成的假图片，判别器尽可能输出 0

（2）训练生成器。

- 固定判别器
- 生成器生成图片，尽可能让判别器输出 1

（3）返回第一步，循环交替训练。

```
for ii, (img, _) in tqdm.tqdm(enumerate(dataloader)):
    real_img = Variable(img)
    if opt.use_gpu:
        real_img = real_img.cuda()

    # 训练判别器
    if (ii + 1) % opt.d_every == 0:
        optimizer_d.zero_grad()
        ## 尽可能地把真图片判别为1
        output = netd(real_img)
        error_d_real = criterion(output, true_labels)
        error_d_real.backward()

        ## 尽可能地把假图片判别为0
        noises.data.copy_(t.randn(opt.batch_size, opt.nz, 1, 1))
        fake_img = netg(noises).detach() # 根据噪声生成假图
        fake_output = netd(fake_img)
        error_d_fake = criterion(fake_output, fake_labels)
        error_d_fake.backward()
        optimizer_d.step()

    # 训练生成器
    if (ii + 1) % opt.g_every == 0:
        optimizer_g.zero_grad()
        noises.data.copy_(t.randn(opt.batch_size, opt.nz, 1, 1))
        fake_img = netg(noises)
        fake_output = netd(fake_img)
        ## 尽可能让判别器把假图片也判别为1
        error_g = criterion(fake_output, true_labels)
        error_g.backward()
        optimizer_g.step()
```

这里需要注意以下几点。

- 训练生成器时，无须调整判别器的参数；训练判别器时，无须调整生成器的参数。
- 在训练判别器时，需要对生成器生成的图片用detach操作进行计算图截断，避免反向传播将梯度传到生成器中。因为在训练判别器时我们不需要训练生成器，也就不需要生成器的梯度。
- 在训练分类器时，需要反向传播两次，一次是希望把真图片判为 1，一次是希望把假图片判为 0。也可以将这两者的数据放到一个 batch 中，进行一次前向传播和一次反向传播即可。但是人们发现，在一个 batch 中只包含真图片或只包含假图片的做法最好。
- 对于假图片，在训练判别器时，我们希望它输出为 0；而在训练生成器时，我们希望它输出为 1。因此可以看到一对看似矛盾的代码：error_d_fake = criterion(fake_output, fake_labels)和error_g = criterion(fake_output, true_labels)。其实这也很好理解，判别器希望能够把假图片判别为fake_label，而生成器则希望能把它判别为true_label，判别器和生成器互相对抗提升。

接下来就是一些可视化的代码。每次可视化使用的噪声都是固定的fix_noises，因为这样便于我们比较对于相同的输入，生成器生成的图片是如何一步步提升的。另外，由于我们对输入的图片进行了归一化处理（-1~1），在可视化时则需要将它还原成原来的 scale（0~1）。

```
fix_fake_imgs = netg(fix_noises)
vis.images(fix_fake_imgs.data.cpu().numpy()[:64] * 0.5 + 0.5, win='fixfake
')
```

除此之外，还提供了一个函数，能加载预训练好的模型，并利用噪声随机生成图片。

```
# 定义噪声和网络
netg, netd = NetG(opt).eval(), NetD(opt).eval()
noises = t.randn(opt.gen_search_num, opt.nz, 1, 1).normal_(opt.gen_mean,
opt.gen_std)
noises = Variable(noises, volatile=True)

# 加载预训练的模型
netd.load_state_dict(t.load(opt.netd_path))
netg.load_state_dict(t.load(opt.netg_path))

# 是否使用GPU
if opt.use_gpu:
    netd.cuda()
```

```
    netg.cuda()
    noises = noises.cuda()

# 生成图片，并计算图片在判别器的分数
fake_img = netg(noises)
scores = netd(fake_img).data

# 挑选最好的某几张
indexs = scores.topk(opt.gen_num)[1]
result = []
for ii in indexs:
    result.append(fake_img.data[ii])

# 保存图片
tv.utils.save_image(t.stack(result), opt.gen_img, normalize=True, range
=(-1,1))
```

完整的代码请参考本书的附带样例代码chapter7/AnimeGAN。参照README.MD中的指南配置环境，并准备好数据，而后用如下命令即可开始训练：

```
python main.py train --gpu=True # 使用GPU
                --vis=True # 使用Visdom
                --batch-size=256 # batch size
                --max-epoch=200 # 训练200个epoch
```

如果使用 visdom 的话，此时打开http://[your ip]:8097就能看到生成的图像。

训练完成后，我们可以利用生成网络随机生成动漫头像，输入命令如下：

```
python main.py generate
            --gen-img='result1.5w.png'
            --gen-search-num=15000
```

7.3 实验结果分析

实验结果如图 7-9 所示，分别是训练 1 个、10 个、20 个、30 个、40 个、200 个 epoch 之后神经网络生成的动漫头像。需要注意的是，每次生成器输入的噪声都是一样的，所以我们可以对比在相同的输入下，生成图片的质量是如何慢慢改善的。

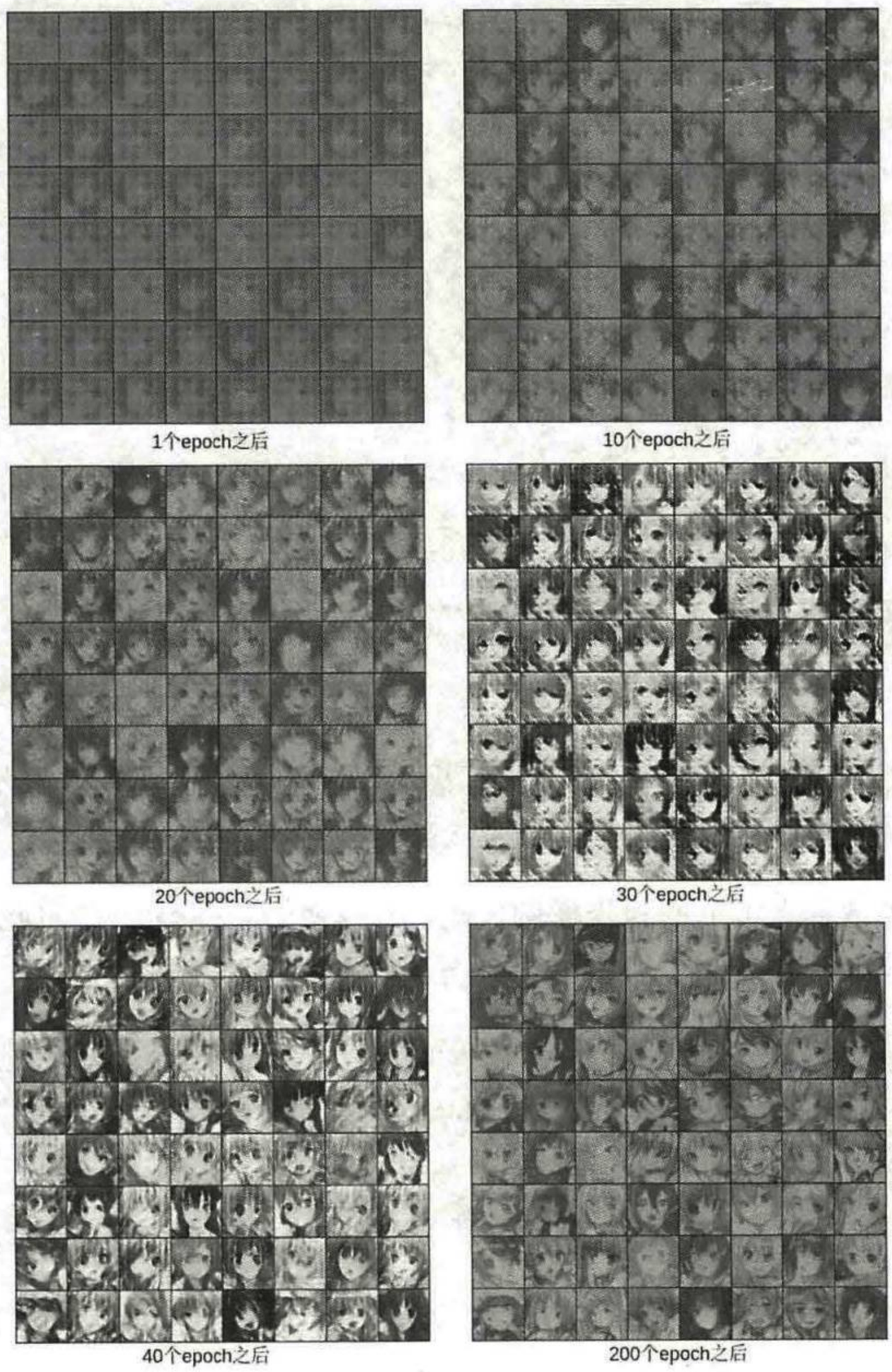


图 7-9　GAN 生成的动漫头像

刚开始生成的图像比较模糊（1 个 epoch），但是可以看出图像已经有面部轮廓。

继续训练 10 个 epoch 之后，生成的图多了很多细节信息，包括头发、颜色等，但是总体还是很模糊。

训练 20 个 epoch 之后，细节继续完善，包括头发的纹理、眼睛的细节等，但还是有不少涂抹的痕迹。

训练到第 40 个 epoch 时，已经能看出明显的面部轮廓和细节，但还是有涂抹现象，并且有些细节不够合理，例如眼睛一大一小，面部的轮廓扭曲严重。

当训练到 200 个 epoch 之后，图片的细节已经十分完善，线条更流畅，轮廓更清晰，虽然还有一些不合理之处，但是已经有不少图片能够以假乱真了。

类似的生成动漫头像的项目还有“用 DRGAN 生成高清的动漫头像”，效果如图 7-10 所示。但遗憾的是，由于论文中使用的数据涉及版权问题，未能公开。这篇论文的主要改进包括使用了更高质量的图片数据和更深、更复杂的模型。

图 7-10　用 DRGAN 生成的动漫头像

本章讲解的样例程序还可以应用到不同的生成图片场景中，只要将训练图片改成其他类型的图片即可，例如 LSUN 客房图片集、MNIST 手写数据集或 CIFAR10 数据集等。事实上，上述模型还有很大的改进空间。在这里，我们使用的全卷积网络只有四层，模型比较浅，而在 ResNet 的论文发表之后，也有不少研究者尝试在 GAN 的网络结构中引入 Residual Block 结构，并取得了不错的视觉效果。感兴趣的读者可以尝试将示例代码中的单层卷积修改为 Residual Block，相信可以取得不错的效果。

近年来，GAN 的一个重大突破在于理论研究。论文 *Towards Principled Methods for Training Generative Adversarial Networks*[①]从理论的角度分析了 GAN 为何难以训练，作

[①] Arjovsky M, Bottou L. Towards principled methods for training generative adversarial networks[J]. arXiv preprint arXiv:1701.04862, 2017.

者随后在另一篇论文 *Wasserstein GAN*[①]中针对性地提出了一个更好的解决方案。但是 *Wasserstein GAN* 这篇论文在部分技术细节上的实现过于随意，所以随后又有人有针对性地提出 *Improved Training of Wasserstein GANs*[②]，更好地训练 WGAN。后面两篇论文分别用 PyTorch 和 TensorFlow 实现，代码可以从 GitHub 上搜索到。笔者当初也尝试用 100 行左右的代码实现了 Wasserstein GAN，感兴趣的读者可以去了解[③]。

随着 GAN 研究的逐渐成熟，人们也尝试把 GAN 用于工业实际问题之中，而在众多相关论文中，最令人印象深刻的就是 *Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks*[④]，论文中提出了一种新的 GAN 结构称为 CycleGAN。CycleGAN 利用 GAN 实现风格迁移、黑白图像彩色化，以及马和斑马相互转化等，效果十分出众。论文的作者用 PyTorch 实现了所有代码，并开源在 GitHub[⑤]上，感兴趣的读者可以自行查阅。

本章主要介绍 GAN 的基本原理，并带领读者利用 GAN 生成动漫头像。GAN 有许多变种，GitHub 上有许多利用 PyTorch 实现的各种 GAN，感兴趣的读者可以自行查阅。

---

① Arjovsky M, Chintala S, Bottou L. Wasserstein gan[J]. arXiv preprint arXiv:1701.07875, 2017.

② Gulrajani I, Ahmed F, Arjovsky M, et al. Improved training of wasserstein gans[J]. arXiv preprint arXiv:1704.00028, 2017.

③ https://github.com/chenyuntc/pytorch-GAN/blob/master/WGAN.ipynb

④ Zhu J Y, Park T, Isola P, et al. Unpaired image-to-image translation using cycle-consistent adversarial networks[J]. arXiv preprint arXiv:1703.10593, 2017.

⑤ https://github.com/junyanz/pytorch-CycleGAN-and-pix2pix

8

AI 艺术家：神经网络风格迁移

本章我们将介绍一个酷炫的深度学习应用——风格迁移（Style Transfer）。近年来，由深度学习引领的人工智能技术浪潮越来越广泛地应用到社会各个领域。这其中，手机应用 Prisma，尝试为用户的照片生成名画效果，一经推出就吸引了海量用户，登顶 App Store 下载排行榜。这神奇背后的核心技术就是基于深度学习的图像风格迁移。

风格迁移又称风格转换，直观点的类比就是给输入的图像加个滤镜，但是又不同于传统滤镜。风格迁移基于人工智能，每个风格都是由真正的艺术家作品训练、创作而成。只需要给定原始图片，并选择艺术家的风格图片，就能把原始图片转化成具有相应艺术家风格的图片。如图 8-1 所示，给定一张风格图片（左上角，手绘糖果图）和一张内容图片（右上角，斯坦福校园图），神经网络能够生成手绘风格的斯坦福校园图（下图）。

本章我们将一起学习风格迁移的原理，并用 PyTorch 从头实现一个风格迁移的神经网络，来看看人工智能与艺术的交叉碰撞会产生什么样的有趣结果。

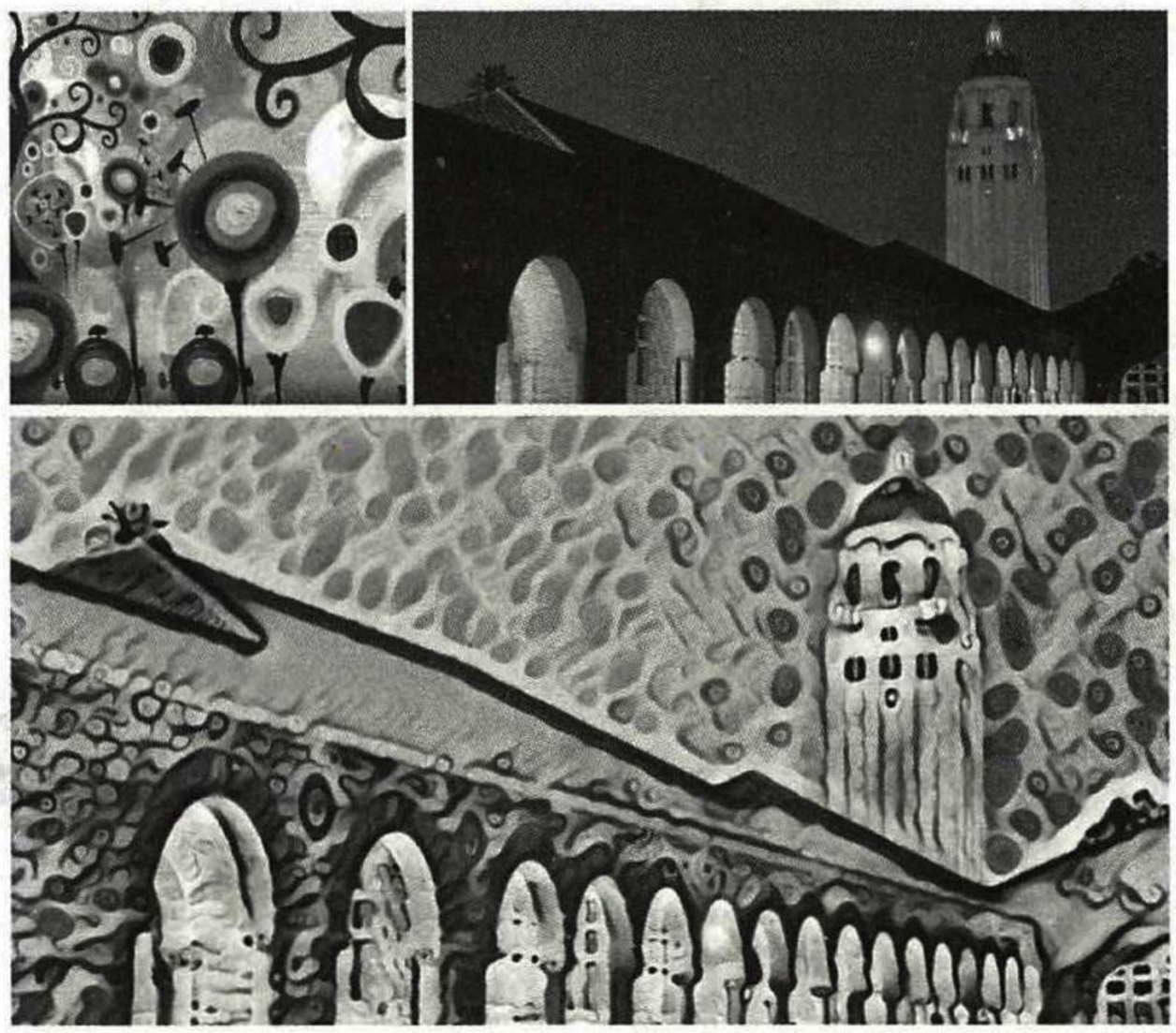


图 8-1　斯坦福校园风格迁移

8.1　风格迁移原理介绍

风格迁移中有两类图片，一类是风格图片，通常是一些艺术家的作品，比较经典的有梵高的《星月夜》《向日葵》，毕加索的《A muse》，莫奈的《印象・日出》，日本浮世绘的《神奈川冲浪里》等，这些图片往往具有比较明显的艺术家风格，包括色彩、线条、轮廓等；另一类是内容图片，这些图片通常来自现实世界中，例如用户个人摄影。利用风格迁移能够将内容图片转成具有艺术家风格的图片。

2015 年，来自德国图宾根大学（University of Tübingen）Bethge 实验室的三位研究员莱昂・盖提斯（Leon Gatys）、亚历山大・埃克（Alexander Ecker）和马蒂亚斯・贝特格（Matthias Bethge）研发了一种算法，模拟人类视觉的处理方式，通过训练多层卷积神经网络（CNN），让计算机识别并学会梵高的“风格”，然后将任何一张普通的照片变成梵高的《星空》。2015 年，他们的发现被整理成两篇论文：*A Neural Algorithm of Artistic Style*[①]和 *Texture Synthesis Using Convolutional Neural Networks*[②]，引起了学术界和工业界的极大兴趣。

Gatys 等人提出的方法被称为 Neural Style，然而他们的做法在实现上过于复杂，每

① Gatys L A, Ecker A S, Bethge M. A neural algorithm of artistic style[J]. arXiv preprint arXiv:1508.06576, 2015.

② Gatys L, Ecker A S, Bethge M. Texture synthesis using convolutional neural networks[C]//Advances in Neural Information Processing Systems. 2015: 262-270.

次进行风格迁移都需要几十分钟甚至几个小时的训练。斯坦福博士生 Justin Johnson 于 2016 年在 ECCV 上发表论文 *Perceptual Losses for Real-Time Style Transfer and Super-Resolution*[①]，提出了一种快速实现风格迁移的算法，这种方法通常被称为 Fast Neural Style。当用 Fast Neural Style 训练好某一个风格的模型之后，通常只需要 GPU 运行几秒，就能生成对应的风格迁移结果。本章中介绍的主要是基于 Justin Johnson 的 Fast Neural Style 方法，即快速风格迁移。

Fast Neural Style 和 Neural Style 主要有以下两点区别。

（1）Fast Neural Style 针对每一个风格图片训练一个模型（在 GPU 上运行大概 4 个小时），而后可以反复使用，进行快速风格迁移（几秒到 20 秒）。Neural Style 不需要专门训练模型，只需要从噪声中不断地调整图像的像素值，直到最后得到结果，速度较慢，需要十几分钟到几十分钟不等。

（2）普遍认为 Neural Style 生成的图片效果会比 Fast Neural Style 的效果好。

关于 Neural Style 的实现，可以参考 PyTorch 官方的 Tutorial 中的教程[②]，实现也比较简单。本节主要介绍 Fast Neural Style 的实现。

要产生效果逼真的风格迁移的图片有两个要求。一是要生成的图片在内容、细节上尽可能地与输入的内容图片相似；二是要生成的图片在风格上尽可能地与风格图片相似。相应地，我们定义两个损失 content loss 和 style loss，分别用来衡量上述两个指标。

图像的内容和风格含义广泛，并且没有严格统一的数学定义，具有很大程度上的主观性，因此很难表示。content loss 比较常用的做法是采用逐像素计算差值，又称 pixel-wise loss，追求生成的图片和原始图片逐像素的差值尽可能小。这种做法有诸多不合理的地方，Justin 在论文中提出了一种更好的计算 content loss 的方法：perceptual loss。不同于 pixel-wise loss 计算像素层面的差异，perceptual loss 计算的是图像在更高层语义层次上的差异，论文中使用预训练好的神经网络的高层输入作为图片的知觉特征，进而计算二者的差异值作为 perceptual loss。

深度学习之所以被称为“深度”，就在于它采用了深层的网络结构，网络的不同层学到的是图像不同层面的特征信息。深度学习网络的输入是像素信息，也可以认为是点，研究表明，几乎所有神经网络的第一层学习到的都是关于线条和颜色的信息，直观理解就是像素组成色彩，点组成线，这与人眼的感知特征十分相像。再往上，神经网络开始关注一些复杂的特征，例如拐角或者某些特殊的形状，这些特征可以看成是低层次的特征组合。随着深度的加深，神经网络关注的信息逐渐抽象，例如有些卷积核关注

---

① Johnson J, Alahi A, Fei-Fei L. Perceptual losses for real-time style transfer and super-resolution[C]//European Conference on Computer Vision. Springer International Publishing, 2016: 694-711.

② http://pytorch.org/tutorials/advanced/neural_style_tutorial.html

的是这张图中有个鼻子，或者是图中有张人脸，以及对象之间的空间关系，例如鼻子在人脸的中间等。

在进行风格迁移时，我们并不要求生成图片的像素和原始图片中的每一个像素都一样，我们追求的是生成图片和原图片具有相同的特征：例如原图中有只猫，我们希望风格迁移之后的图片依旧有猫。图片中“有猫”这个概念不就是我们分类问题最后一层的输出吗？但最后一层的特征对我们来说抽象程度太高，因为我们不仅希望图中有只猫，还希望保存这只猫的部分细节信息，例如它的形状、动作等信息，这些信息相对来说没有那么高层次。因此我们使用中间某些层的特征作为目标，希望原图像和风格迁移的结果在这些层输出的特征尽可能相似，即将图片在深度模型的中间某些层的输出作为图像的知觉特征。

我们一般使用 Gram 矩阵来表示图像的风格特征。对于每一张图片，卷积层的输出形状为 $C \times H \times W$ ，C 是卷积核的通道数，一般称为有 C 个卷积核，每个卷积核学习图像的不同特征。每一个卷积核输出的 $H \times W$ 代表这张图像的一个 feature map，可以认为是一张特殊的图像——原始彩色图像可以看作 RGB 三个 feature map 拼接组合成的特殊 feature maps。通过计算每个 feature map 之间的相似性，我们可以得到图像的风格特征。对于一个 $C \times H \times W$ 的 feature maps $\boldsymbol{F}$，Gram Matrix 的形状为 $C \times C$，其第 i、j 个元素 $\boldsymbol{G}_{i,j}$ 的计算方式定义如下：

$$\boldsymbol{G}_{i,j} = \sum_k \boldsymbol{F}_{ik}\boldsymbol{F}_{jk}$$

其中 $\boldsymbol{F}_{ik}$ 代表第 i 个 feature map 的第 k 个像素点。关于 Gram Matrix，以下三点值得注意：

- Gram Matrix 的计算采用了累加的形式，抛弃了空间信息。一张图片的像素随机打乱之后计算得到的 Gram Matrix 和原图的 Gram Matrix 一样。所以可以认为 Gram Matrix 抛弃了元素之间的空间信息。
- Gram Matrix 的结果与 feature maps $\boldsymbol{F}$ 的尺度无关，只与通道数有关。无论 H 、W 的大小如何，最后 Gram Matrix 的形状都是 $C \times C$。
- 对于一个 $C \times H \times W$ 的 feature maps，可以通过调整形状和矩阵乘法快速计算它的 Gram Matrix，即先将 $\boldsymbol{F}$ 调整为 $C \times (HW)$ 的二维矩阵，然后再计算 $\boldsymbol{F} \cdot \boldsymbol{F}^{\mathrm{T}}$ ，结果就是 Gram Matrix。

图 8-2 展现了 Gram Matrix 的特点：注重风格纹理等特征，忽略空间信息。图中第一行是输入的原图片，经过神经网络计算出不同层的 Gram Matrix，然后尝试从这些层的 Gram Matrix 恢复出原图，换一种角度来说，我们可以认为每一列的图像的 Gram Matrix

值都很接近。尤其是第四行和第一行的对比，我们可以明显地看出，无论恢复的图像清晰度如何，图像的空间信息在计算 Gram Matrix 时都被舍弃，但是纹理、色彩等风格信息被保存下来。

图 8-2　从 Gram Matrix 中恢复原图

实践证明利用 Gram Matrix 表征图像的风格特征在风格迁移、纹理合成等任务中的表现十分出众。

总结如下。

- 神经网络的高层输出可以作为图像的知觉特征描述。
- 神经网络的高层输出的 Gram Matrix 可以作为图像的风格特征描述。
- 风格迁移的目标是使生成图片和原图片的知觉特征尽可能相似，并且和风格图片的风格特征尽可能地相似。

在最初的 Neural Style 论文中，随机初始化目标图片为噪声，然后利用梯度下降法调整图片，使目标图片和风格图片的风格特征（即 Gram Matrix）尽可能地相似，和原图片的知觉特征也尽可能地相似。这种做法生成的图片效果很好，但其十分耗时！每次都需要从一个噪声开始调整图片，直到得到最终的目标图片，在 GPU 上完成一次风格迁移需要十几分钟甚至数小时。

2016 年 Justin Johnson 提出了一种快速风格迁移算法，这种算法被称为 **Fast Neural Style** 或 **Fast Style Transfer**。与 Neural Style 相比，Fast Neural Style 专门设计了一个网络用来进行风格迁移，输入原图片，网络将自动生成目标图片。这个网络需要针对每一种风格图片训练一个相对应的风格网络，但是一旦训练完成，便只需要 20 秒甚至更短的时间就能完成一次风格迁移。

Fast Neural Style 的网络结构如图 8-3 所示，x 是输入图像，在风格迁移任务中 $y_c = x$，y_s 是风格图片，Image Transform Net f_W 是我们设计的风格迁移网络，针对输入的图像 x，能够返回一张新的图像 $\hat{y}$。$\hat{y}$ 在图像内容上与 y_c 相似，但在风格上与 y_s 相似。损失网络（Loss Network）不用训练，只是用来计算知觉特征和风格特征在论文中，损失网络。采用在 ImageNet 上预训练好的 VGG-16。

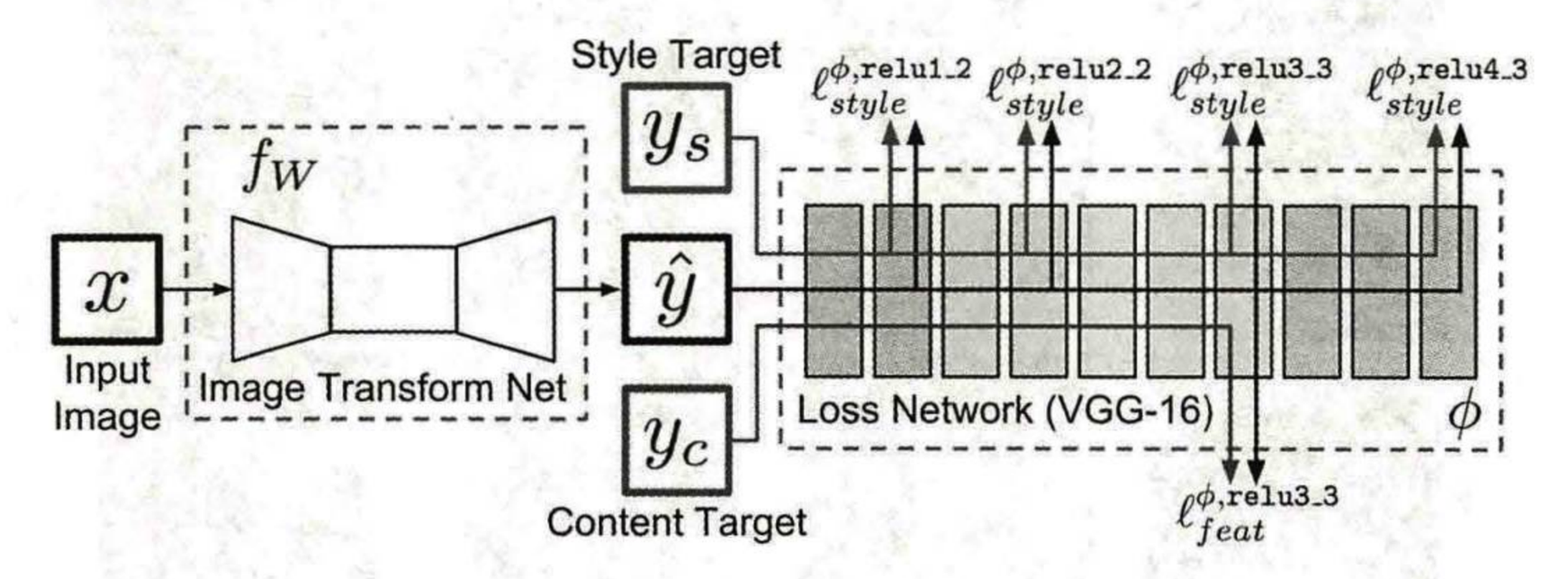


图 8-3　Fast Neural Style 的结构概览

VGG-16 的网络结构如图 8-4 所示，网络从左到右有 5 个卷积块，两个卷积块之间通过 MaxPooling 层区分。每个卷积块有 2~3 个卷积层，每一个卷积层后面都跟着一个 ReLU 激活层。上面的`relu2_2`表示第 2 个卷积块的第 2 个卷积层的激活层（ReLU）输出。

Fast Neural Style 的训练步骤如下。

（1）输入一张图片 x 到 f_W 中得到结果 $\hat{y}$。

（2）将 $\hat{y}$ 和 y_c（其实就是 x）输入到 loss network（VGG-16）中，计算它在`relu3_3`的输出，并计算它们之间的均方误差作为 content loss。

（3）将 $\hat{y}$ 和 y_s（风格图片）输入到 loss network 中，计算它在`relu1_2`、`relu2_2`、`relu3_3`和`relu4_3`的输出，再计算它们的 Gram Matrix 的均方误差作为 style loss。

（4）两个损失相加，并反向传播。更新 f_W 的参数，固定 loss network 不动。

（5）跳回第一步，继续训练 f_W。

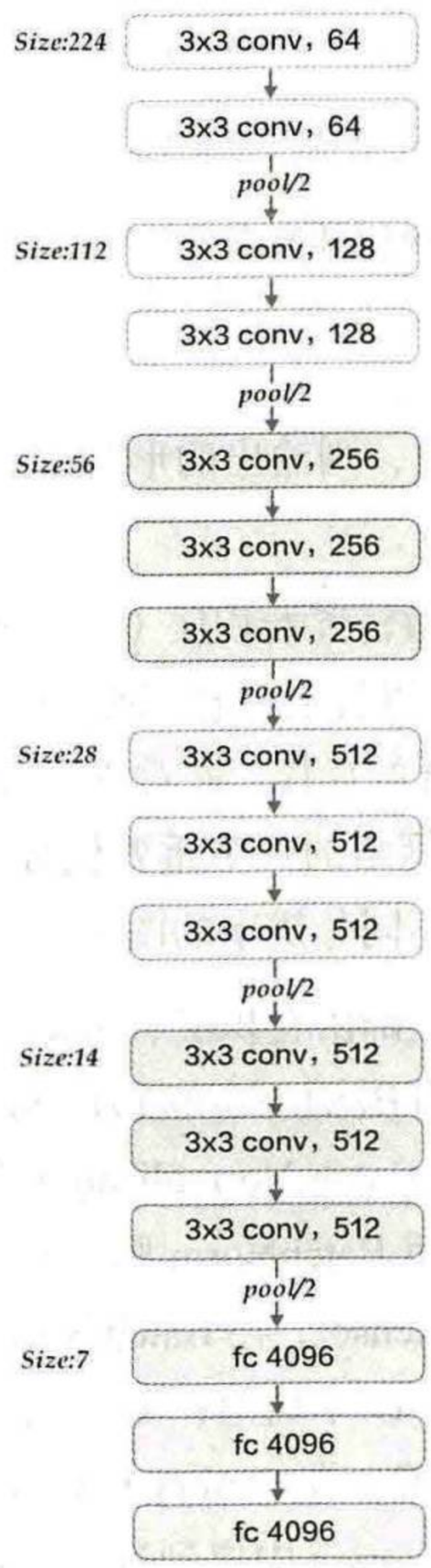


图 8-4　VGG-16 的网络结构

在讲解如何用 PyTorch 实现风格迁移之前，我们先来了解全卷积网络的结构。我们知道风格迁移网络的输入是图片，输出也是图片，对这种网络我们一般实现为一个全部都是卷积层而没有全连接层的网络结构。对于卷积层，当输入 feature maps（或者图片）的尺寸为 $C_{in} \times H_{in} \times W_{in}$，卷积核有 C_{out} 个，卷积核尺寸为 K，padding 大小为 P，步长为 S 时，输出 feature maps 的形状为 $C_{out} \times H_{out} \times W_{out}$，其中

$$H_{out} = \text{floor}(H_{in} + 2 * P - K)/S + 1$$
$$W_{out} = \text{floor}(W_{in} + 2 * P - K)/S + 1$$

举例来说，如果我们输入图片的尺寸是 $3 \times 256 \times 256$，第一层卷积的卷积核大小为 3，padding 为 1，步长为 2，通道数为 128，那么输出的 feature map 形状，按照上述

公式计算的结果就是：

$$H_{out} = \text{floor}(256 + 2 * 1 - 3)/2 + 1 = 128$$
$$W_{out} = \text{floor}(256 + 2 * 1 - 3)/2 + 1 = 128$$

所以最后的输出是 $C_{out} \times H_{out} \times W_{out} = 128 \times 128 \times 128$，即尺度缩小一倍，通道数增加。如果把步长由 2 改成 1，则输出的形状就是 $128 \times 256 \times 256$，即尺度不变，只是通道数增加。

除卷积层之外，还有一种叫做转置卷积层（Transposed Convolution），也有人称之为反卷积（DeConvolution），它可以看成是卷积操作的逆运算。对于卷积操作，当步长大于 1 时，执行的是类似于下采样的操作，而对于转置卷积，当步长大于 1 时，执行的是类似于上采样的操作。全卷积网络的一个重要优势在于对输入的尺寸没有要求，这样在进行风格迁移时就能够接受不同分辨率的图片。

论文中提到的风格迁移结构全部由卷积层、Batch Normalization 和激活层组成，不包含全连接层，在这里我们不使用 Batch Normalization，取而代之的是 Instance Normalization。Instance Normalization 和 Batch Normalization 的唯一区别就在于 InstanceNorm 只对每一个样本求均值和方差，而 BatchNorm 则会对一个 batch 中所有的样本求均值。例如对于一个 $B \times C \times H \times W$ 的 tensor，在 Batch Normalization 中计算均值时，就会计算 $B \times H \times W$ 个数的均值，共有 C 个均值；而 Instance Normalization 会计算 $H \times W$ 个数的均值，即共有 $B \times C$ 个均值。在 Dmitry Ulyanov 的论文 *Instance Normalization: The Missing Ingredient for Fast Stylization* 中提到过，用 InstanceNorm 代替 BatchNorm 能够显著地提升风格迁移的效果。

8.2 用 PyTorch 实现风格迁移

我们先来看看本次实验的文件组织：

```
checkpoints/
data/
    coco -> /mnt/3/coco/train2014
main.py
PackedVGG.py
style.jpg
transformer_net.py
utils.py
```

上述各个文件的主要内容和作用如下。

- `checkpoints/`：用来保存模型。
- `data/`：用来保存数据，可以把数据直接保存于`coco`文件夹下，也可以通过`ln -s /mnt/3/coco/train2014 data/coco`软链接的方式将数据链接到 data/文件夹下，还可以通过命令行参数另外指定数据路径。
- `main.py`：主函数，包括训练和测试。
- `PackedVGG.py`：预训练好的 VGG-16，为了提取中间层的输出，所以做了一些修改简化。
- `transformer_net.py`：风格迁移网络。输入一张图片，输出一张图片。
- `utils.py`：工具集合，主要是可视化工具 visdom 的封装和计算 Gram Matrix 等。

首先来看看如何使用预训练的 VGG，这部分代码保存在`PackedVGG.py`中。在 torchvision 的仓库中有预训练好的 VGG-16 网络，使用十分方便，但在风格迁移网络中，我们需要获得中间层的输出，因此需要修改网络的前向传播过程，将相应层的输出保存下来。同时有很多层不再需要，可删除以节省内存占用。实现的代码如下。

```
#coding:utf8
import torch
import torch.nn as nn
from torchvision.models import vgg16
from collections import namedtuple

class Vgg16(torch.nn.Module):
    def __init__(self):
        super(Vgg16, self).__init__()
        features = list(vgg16(pretrained = True).features)[:23]
        self.features = nn.ModuleList(features).eval()

    def forward(self, x):
        results = []
        # features的第3, 8, 15, 22层分别是: relu1_2, relu2_2, relu3_3,
        relu4_3
        for ii, model in enumerate(self.features):
            x = model(x)
            if ii in {3, 8, 15, 22}:
                results.append(x)
```

```
    vgg_outputs = namedtuple("VggOutputs", ['relu1_2', 'relu2_2', '
    relu3_3', 'relu4_3'])
    return vgg_outputs(*results)
```

在 torchvision 中，VGG 的实现由两个`nn.Sequential`对象组成，第一个是`features`，包含卷积、激活和 MaxPool 等层，用来提取图片特征，另一个是`classifier`，包含全连接等，用来分类。可以通过 vgg.features 直接获得对应的`nn.Sequential`对象。这样在前向传播时，当计算完指定层的输出后，就将结果保存于一个 list 中，然后再使用`namedtuple`进行名称绑定，这样可以通过`output.relu1_2`访问第一个元素，更为方便和直观。当然也可以利用`layer.register_forward_hook`的方式获取相应层的输出，但是在本例中相对比较麻烦。

接下来要实现的是风格迁移网络，其代码在`transformer_net.py`中，实现时参考了 PyTorch 的官方示例[①]，其网络结构如图 8-5 所示。图中 (b) 是网络的总体结构，左边 (d) 是一个残差单元的结构图，右边 (c) 和 (d) 分别是下采样和上采样单元的结构图。网络结构总结起来有以下几个特点。

- 先下采样，然后上采样，这种做法使计算量变小，详细说明请参考论文。
- 使用残差结构使网络变深。
- 边缘补齐的方式不再是传统的补 0，而是采用一种被称为 Reflection Pad 的补齐策略：上下左右反射边缘的像素进行补齐。
- 上采样不再使用传统的`ConvTransposed2d`，而是先用`Upsample`，然后用`Conv2d`，这种做法能避免 Checkerboard Artifacts 现象。
- Batch Normalization 全部改成 Instance Normalizaiton。
- 网络中没有全连接层，线性操作是卷积，因此对输入和输出图片的尺寸没有要求，这里我们输入和输出图片的尺寸都是 $3 \times 256 \times 256$（其他尺寸的一样可以）。

在第 6 章中我们提到过，对于常出现的网络结构，可以实现为`nn.Module`对象，作为一个特殊的层。在本例中，Conv、UpConv 和 Residual Block 都可以如此实现。

例如 Conv，可以实现为如下：

```
class ConvLayer(nn.Module):
    '''
    add ReflectionPad for Conv
    默认的卷积的padding操作是补0，这里使用边界反射填充
```

[①] https://github.com/pytorch/examples/tree/master/fast_neural_style

```
    '''
    def __init__(self, in_channels, out_channels, kernel_size, stride):
        super(ConvLayer, self).__init__()
        reflection_padding = int(np.floor(kernel_size / 2))
        self.reflection_pad = nn.ReflectionPad2d(reflection_padding)
        self.conv2d = nn.Conv2d(in_channels, out_channels, kernel_size,
        stride)

    def forward(self, x):
        out = self.reflection_pad(x)
        out = self.conv2d(out)
        return out
```

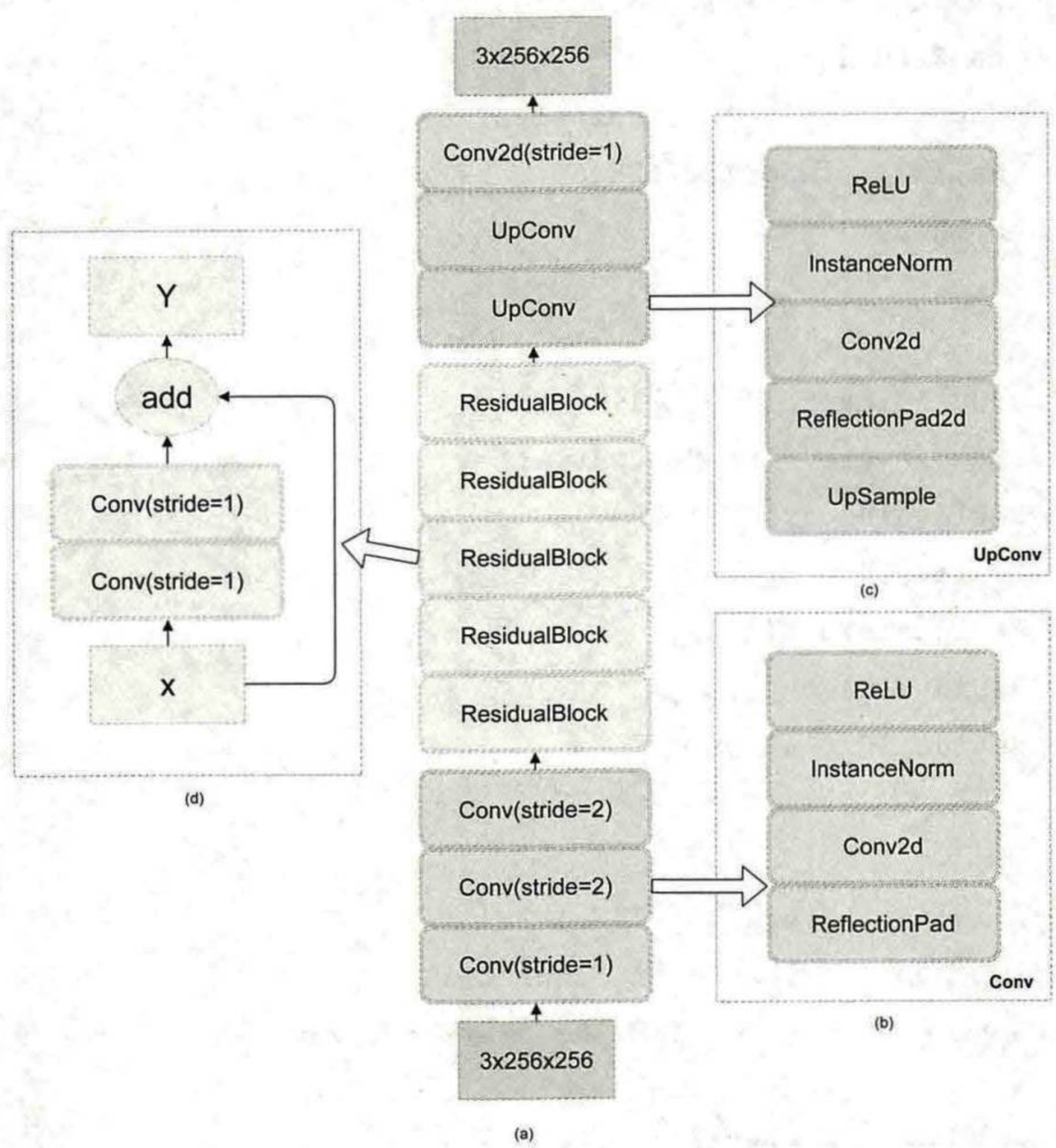


图 8-5　风格迁移的网络结构

UpConv 和 Residual Block 的实现也是类似的，这里不再细说，具体内容请看本章配套代码。

主模型主要包含三部分：下采样的卷积层、深度残差层和上采样的卷积层。实现时充分利用了 nn.Sequential，避免在 forward 中重复写代码，其代码如下。

```
class TransformerNet(nn.Module):
    def __init__(self):
        super(TransformerNet, self).__init__()

        # 下卷积层
        self.initial_layers = nn.Sequential(
            ConvLayer(3, 32, kernel_size=9, stride=1),
            nn.InstanceNorm2d(32,affine=True),
            nn.ReLU(True),
            ConvLayer(32, 64, kernel_size=3, stride=2),
            nn.InstanceNorm2d(64,affine=True),
            nn.ReLU(True),
            ConvLayer(64, 128, kernel_size=3, stride=2),
            nn.InstanceNorm2d(128,affine=True),
            nn.ReLU(True),
        )

        # Residual layers(残差层)
        self.res_layers = nn.Sequential(
            ResidualBlock(128),
            ResidualBlock(128),
            ResidualBlock(128),
            ResidualBlock(128),
            ResidualBlock(128)
        )

        # Upsampling Layers(上卷积层)
        self.upsample_layers = nn.Sequential(
            UpsampleConvLayer(128, 64, kernel_size=3, stride=1, upsample
            =2),
            nn.InstanceNorm2d(64,affine=True),
            nn.ReLU(True),
            UpsampleConvLayer(64, 32, kernel_size=3, stride=1, upsample=2)
            ,
            nn.InstanceNorm2d(32,affine=True),
```

```
            nn.ReLU(True),
            ConvLayer(32, 3, kernel_size=9, stride=1)
        )


    def forward(self, x):
        x = self.initial_layers(x)
        x = self.res_layers(x)
        x = self.upsample_layers(x)
        return x
```

搭建完网络之后，我们还要实现一些工具函数，这部分的代码保存于util.py中。代码如下：

```
IMAGENET_MEAN = [0.485, 0.456, 0.406]
IMAGENET_STD = [0.229, 0.224,  0.225]

def gram_matrix(y):
    '''
    输入形状 b, c, h, w
    输出形状 b, c, c
    '''
    (b, ch, h, w) = y.size()
    features = y.view(b, ch, w * h)
    features_t = features.transpose(1, 2)
    gram = features.bmm(features_t) / (ch * h * w)
    return gram

def get_style_data(path):
    '''
    加载风格图片
    输入: path, 文件路径
    返回: 形状 1 * c * h * w, 分布大约在-2~2, 是一个tensor
    '''
    style_transform = tv.transforms.Compose([
        tv.transforms.ToTensor(),
        tv.transforms.Normalize(mean = IMAGENET_MEAN, std = IMAGENET_STD),
    ])
```

```
    style_image = tv.datasets.folder.default_loader(path)
    style_tensor = style_transform(style_image)
    return style_tensor.unsqueeze(0)

def normalize_batch(batch):
    '''
    输入: b, ch, h, w  0~255, 是一个Variable
    输出: b, ch, h, w  大约-2~2, 是一个Variable
    '''
    mean = batch.data.new(IMAGENET_MEAN).view(1, -1, 1, 1)
    std = batch.data.new(IMAGENET_STD).view(1, -1, 1, 1)
    mean = t.autograd.Variable(mean.expand_as(batch.data))
    std = t.autograd.Variable(std.expand_as(batch.data))
    return (batch / 255.0 - mean) / std
```

可以看出`util.py`中主要包含以下三个功能。

- 计算 Gram Matrix：利用矩阵转置的乘法即可实现，但是这里我们要对 batch 中的每一个样本计算 Gram Matrix，因此用的是`tensor.bmm(tensor2)`，而不是`tensor.mm(tesnor2)`。
- 获取风格图片：根据文件名读取图片，并将它转化成 tensor。这里图片的均值和标准差不是 0.5 和 0.5，而是使用他人专门计算的 ImageNet 上所有图片的均值和标准差，更符合真实世界图片的分布。人们发现按照这个数值处理的图片比简单地使用 0.5 作为均值和标准差效果好。
- 将分布在 0~255 的图片进行标准化，和上述对风格图片的处理类似，需要注意，这里是针对 Variable 对象的处理。

除此之外，还有对 visdom 操作的封装，这里不再展示。

当我们将上述网络定义和工具函数写完之后，就可以开始训练网络了。首先来看看在`config.py`中都有哪些可以配置的参数：

```
class Config(object):
    image_size = 256 # 图片大小
    batch_size = 8
    data_root = 'data/' # 数据集存放路径: data/coco/a.jpg
    num_workers = 4 # 多线程加载数据
    use_gpu = True # 使用GPU
```

```
style_path= 'style.jpg' # 风格图片存放路径
lr = 1e-3 # 学习率

env = 'neural-style' # visdom env
plot_every=10 # 每10个batch可视化一次

epoches = 2 # 训练epoch

content_weight = 1e5 # content_loss的权重
style_weight = 1e10 # style_loss的权重

model_path = None # 预训练模型的路径
debug_file = '/tmp/debugnn' # touch debug_file进入调试模式

content_path = 'input.png' # 需要进行风格迁移的图片
result_path = 'output.png' # 风格迁移结果的保存路径
```

论文中训练用的图片是 MS COCO 数据集，读者可从官网[①]下载。其训练集中包含 8 万张图片，共 13GB。笔者认为 COCO 的数据比 ImageNet 的数据更复杂，更像是日常生活的照片。读者如果有 ImageNet 的图片，也一样可以使用。获取数据之后，将数据解压至`data/coco/`文件夹中，或放到任意位置，然后在训练时指定对应路径。

我们可以在`main.py`中直接利用`ImageFolder`和`DataLoader`加载数据：

```
# 数据预处理，输出图像取值0~255
transfroms = tv.transforms.Compose([
    tv.transforms.Scale(opt.image_size),
    tv.transforms.CenterCrop(opt.image_size),
    tv.transforms.ToTensor(),
    tv.transforms.Lambda(lambda x: x * 255)
    ])
dataset = tv.datasets.ImageFolder(opt.data_root, transfroms)
dataloader = data.DataLoader(dataset, opt.batch_size)
```

正训练前，还需要定义一些额外的信息，包括网络定义、优化器的定义等，还要获取风格图片的风格表征（Gram Matrix）及用来平滑损失的 AverageValueMeter。

① http://images.cocodataset.org/zips/train2014.zip

```
# 转换网络
transformer = TransformerNet()
if opt.model_path:
    transformer.load_state_dict(t.load(opt.model_path, map_location=lambda
     _s, _: _s))

# 损失网络Vgg16
vgg = Vgg16().eval()

# 优化器
optimizer = t.optim.Adam(transformer.parameters(), opt.lr)

# 获取风格图片的数据
style = utils.get_style_data(opt.style_path)
vis.img('style', (style[0] * 0.225 + 0.45).clamp(min=0, max=1))

if opt.use_gpu:
    transformer.cuda()
    style = style.cuda()
    vgg.cuda()

# 风格图片的gram矩阵
style_v = Variable(style, volatile=True)
features_style = vgg(style_v)
gram_style = [Variable(utils.gram_matrix(y.data)) for y in features_style]

# 损失统计标准
style_meter = tnt.meter.AverageValueMeter()
content_meter = tnt.meter.AverageValueMeter()
```

训练步骤在 8.1 节已经讲过，按照训练步骤，很容易写出如下训练代码：

```
for ii, (x, _) in tqdm.tqdm(enumerate(dataloader)):

    # 训练
    optimizer.zero_grad()
    if opt.use_gpu:
        x = x.cuda()
```

```
x = Variable(x)
y = transformer(x)
y = utils.normalize_batch(y)
x = utils.normalize_batch(x)
features_y = vgg(y)
features_x = vgg(x)

# 计算content loss（只用到了relu2_2）
content_loss = opt.content_weight * F.mse_loss(features_y.relu2_2,
features_x.relu2_2)

# style loss同时用到了4层输出
style_loss = 0.
for ft_y, gm_s in zip(features_y, gram_style):
    gram_y = utils.gram_matrix(ft_y)
    style_loss += F.mse_loss(gram_y, gm_s.expand_as(gram_y))
style_loss *= opt.style_weight

# 反向传播、更新梯度，这里只更新transformer的参数，不更新VGG16的
total_loss = content_loss + style_loss
total_loss.backward()
optimizer.step()
```

完整的代码请查看本书的配套源码。这个程序中容易让人混淆的是图片的尺度，有时是0~1，有时是-2~2，还有时是0~255，统一说明如下。

- 图片每个像素的取值范围为0~255。
- 调用 torchvison 的`transforms.ToTensor()`操作，像素会被转换到0~1。
- 这时如果进行标准化（减均值除以标准差），均值和标准差均为 0.5，那么标准化之后图片的分布就是-1~1，但在本次实验中使用的均值和标准差不是 0.5，而是 [0.485, 0.456, 0.406] 和 [0.229, 0.224, 0.225]，这是在 ImageNet 100 万张图片上计算得到的图片的均值和标准差，可以估算得知这时图片的分布范围大概在 $\frac{(0-0.4845)}{0.229} \approx -2.1$ 和 $\frac{(1-0.406)}{0.225} \approx 2.7$ 之间。尽管这时它的分布在-2.1~2.7，但是它的均值接近 0，标准差接近 1，采用 ImageNet 图片的均值和标准差作为标准化参数的目的是图像各个像素的分布接近标准分布。
- VGG-16 网络的输入图像数值大小为使用 ImageNet 均值和标准差进行标准化之后的图片数据，即-2.1~2.7。

- TransformerNet 网络的输入图片的像素值是0~255，输出的像素值也希望是0~255，但是由于输出没有做特殊处理，所以可能出现小于 0 和大于 255 的像素。
- 使用 visdom images 进行可视化和使用torchvision.utils.save_image保存图片时，希望 tensor 的数值位于0~1。

当我们掌握了上述内容后，就不难理解为什么在代码中时不时地出现各种尺度变换（乘以标准差加上均值）和截断操作。尺度变换是为了从一个尺度变成另一个尺度，截断是为了确保数值在一定范围之内（0~1 或者 0~255）。

除了训练模型，我们还希望能加载预训练好的模型对指定的图片进行风格迁移的操作，这部分的代码实现如下。

```
def stylize(**kwargs):
    opt = Config()

    for k_,v_ in kwargs.items():
        setattr(opt, k_, v_)

    # 图片处理
    content_image = tv.datasets.folder.default_loader(opt.content_path)
    content_transform = tv.transforms.Compose([
        tv.transforms.ToTensor(),
        tv.transforms.Lambda(lambda x: x.mul(255))
    ])
    content_image = content_transform(content_image)
    content_image = content_image.unsqueeze(0)
    content_image = Variable(content_image, volatile=True)

    # 模型
    style_model = TransformerNet().eval()
    style_model.load_state_dict(t.load(opt.model_path, map_location=lambda
     _s, _: _s))

    if opt.use_gpu:
        content_image = content_image.cuda()
        style_model.cuda()

    # 风格迁移与保存
    output = style_model(content_image)
```

```
    output_data = output.cpu().data[0]
    tv.utils.save_image(((output_data / 255)).clamp(min=0, max=1), opt.
    result_path)
```

这样，我们就可以通过命令行的方式训练，或者加载预训练好的模型进行风格迁移。

```
# 训练。使用GPU，数据存放于data/下的一个文件夹中
python main.py train \
        --use-gpu \
        --data-root=data \
        --batch-size=2

# 风格迁移。不使用GPU
python main.py stylize \
        --model-path='transformer.pth' \
        --content-path='amber.jpg' \
        --result-path='output.png' \
        --use-gpu=False
```

8.3 实验结果分析

在本例中我们只训练了一个风格的模型，风格图片如图 8-6 所示，风格迁移的结果如图 8-7 所示，上面一行是原始图片（来自无版权图片网站 https://pixabay.com），下面一行是风格迁移的结果。随书附带代码中带有这个预训练的模型，读者可以用其他图片查看风格迁移的效果，图片分辨率越高，风格迁移的效果越好。另外，读者也可以通过指定`--style-path=my_style.png`训练不同风格的迁移网络。

图 8-6　风格迁移使用的风格图片

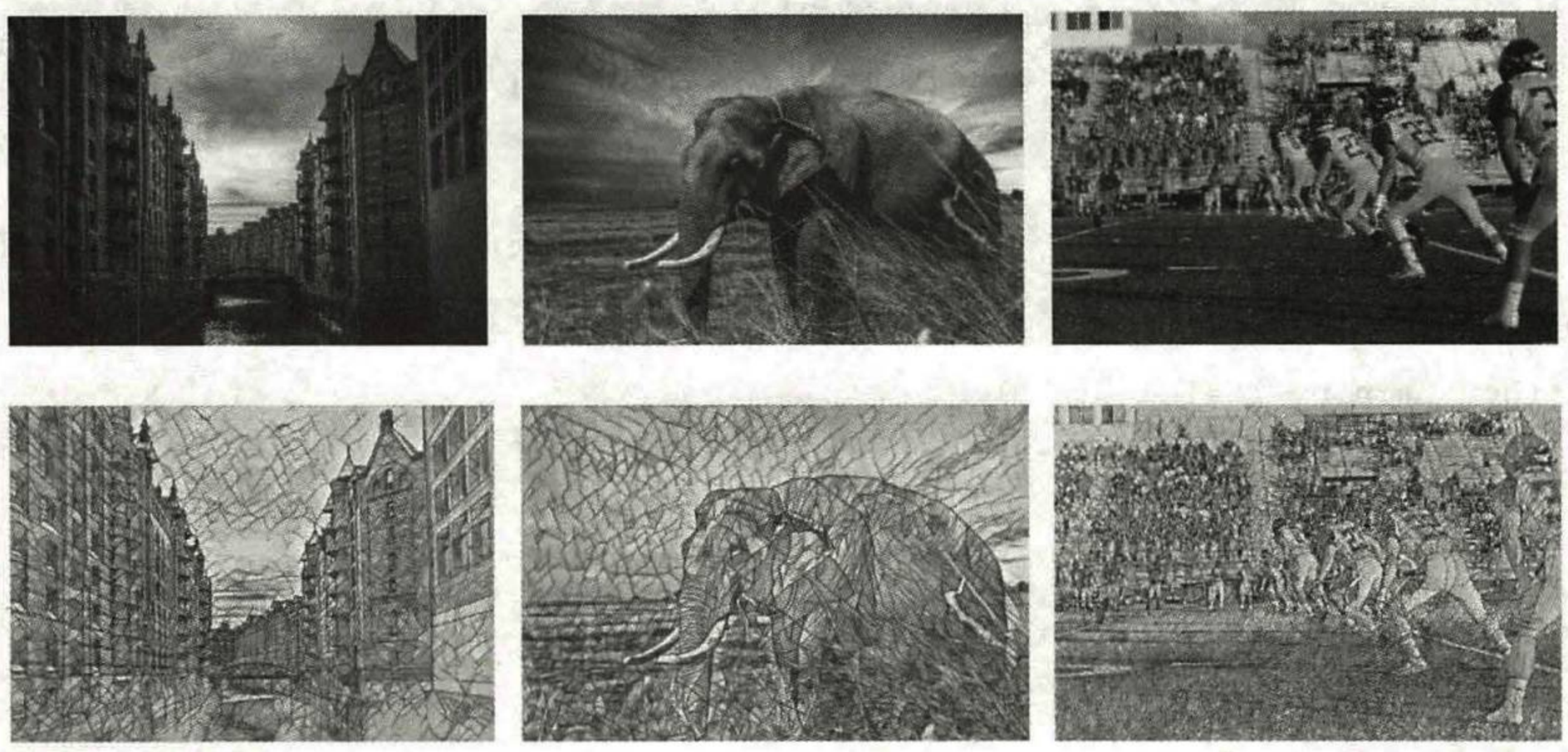


图 8-7　风格迁移结果示意图

除了风格迁移，类似的应用还有 Google DeepDream，可以输入一张图片生成神经网络眼中的这张图片的样子，网络越深，生成的图片中包含的奇幻东西越多，效果如图 8-8 所示。

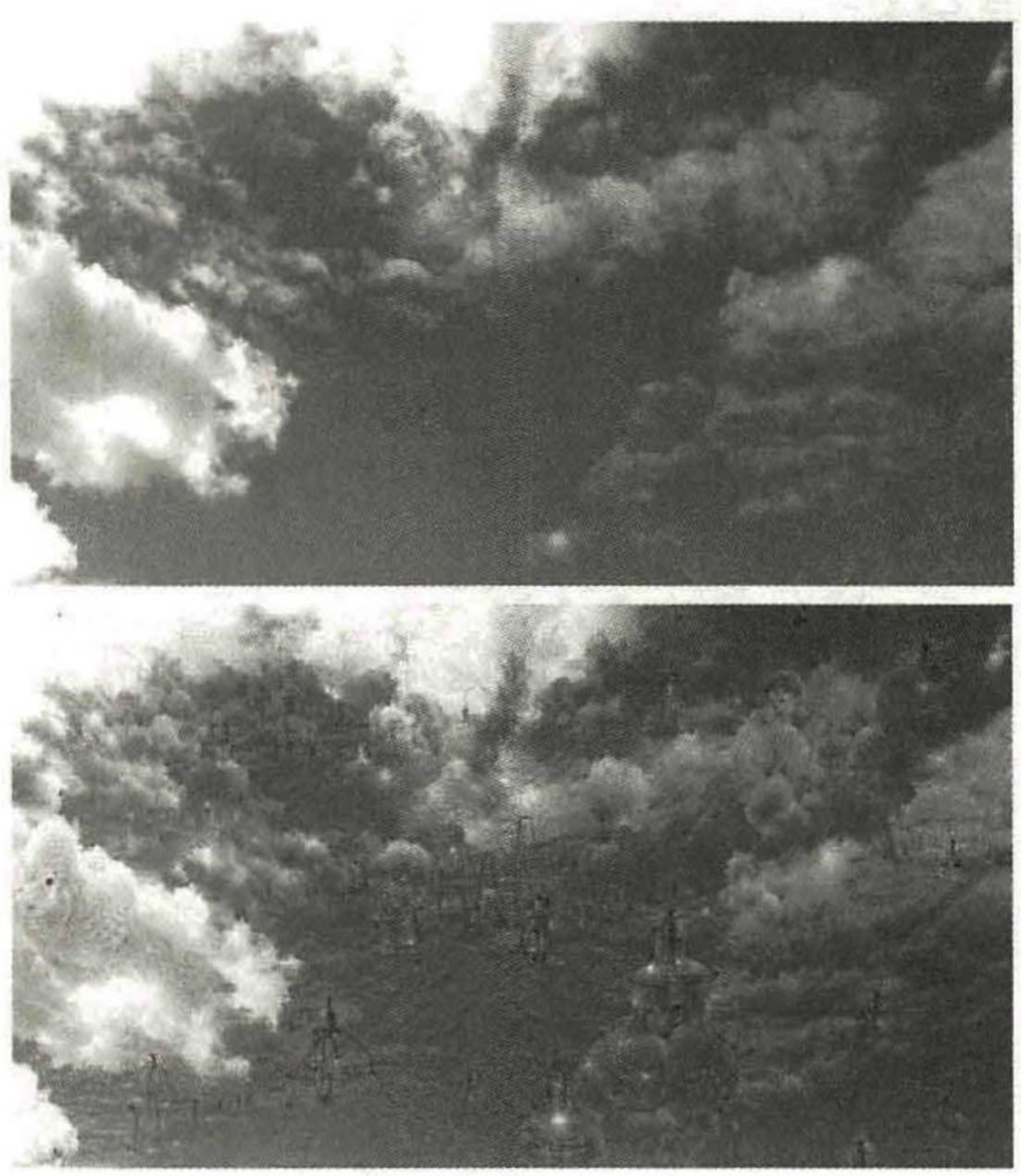


图 8-8　DeepDream 效果图

2017 年有两个比较吸引人的风格迁移项目，一个是来自 Adobe 的图片风格深度迁移（Deep Photo Style Transfer）[①]，这是由康奈尔大学的中国留学生和 Adobe 公司的工程师共同合作的一个新项目，通过深度学习的图片处理方法，直接提取了参考图片的风格，并转换为相对应的滤镜，其效果如图 8-9 所示，最左边是原图，中间是目标风格图片，最右边是将中间的风格迁移到左边的结果。

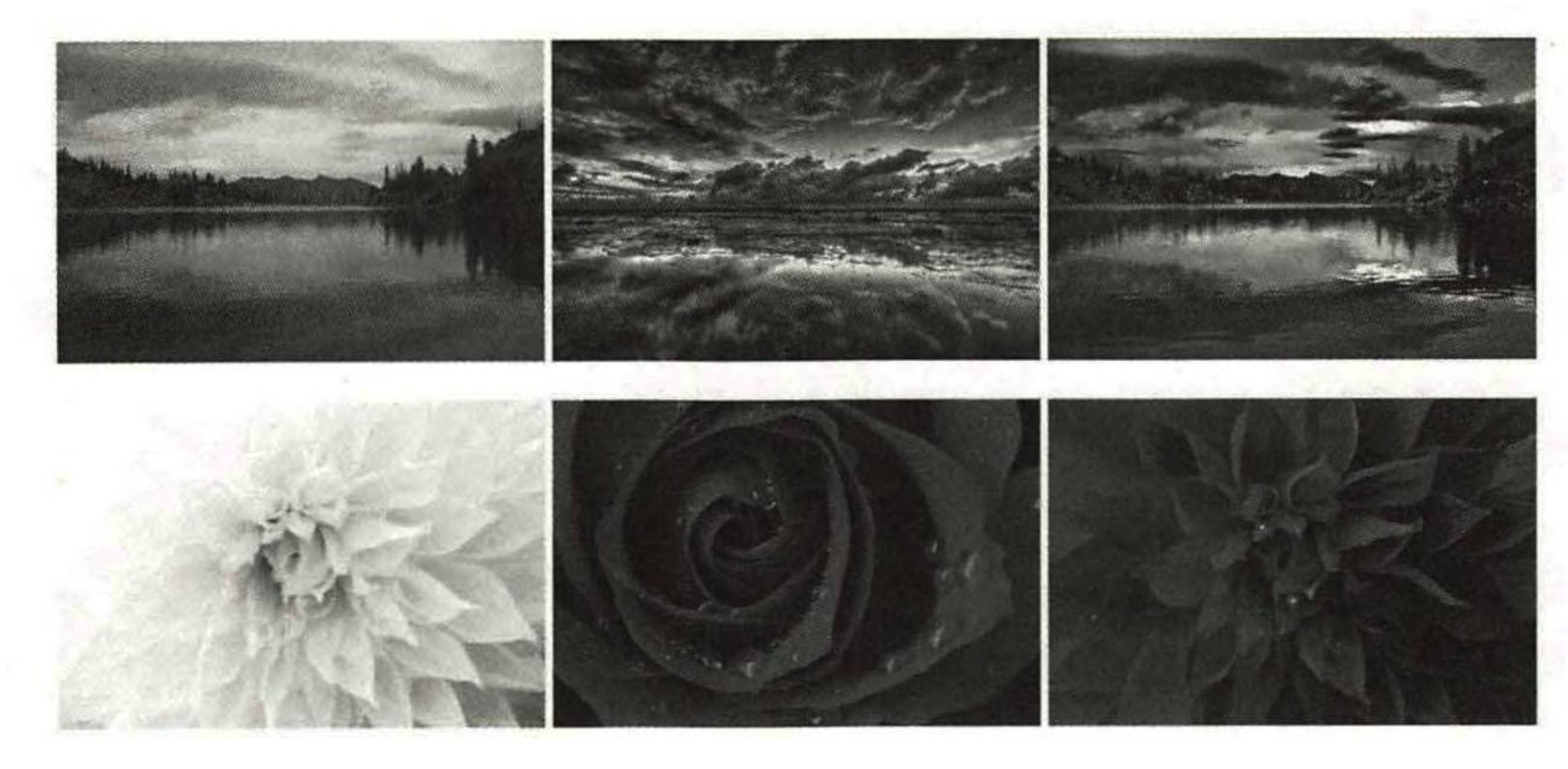


图 8-9　图片风格深度迁移效果图

另外一个项目是来自 UC Berkeley 的 CycleGAN[②]，它能够胜任任何的图像转换和图像翻译任务，在风格迁移上的效果（如图 8-10 所示）尤其令人瞩目。CycleGAN 的网络结构和 Fast Neural Style 的 transformer 类似，但它采用了 GAN 的训练方式，能够实现风格的双向转换，更加通用。

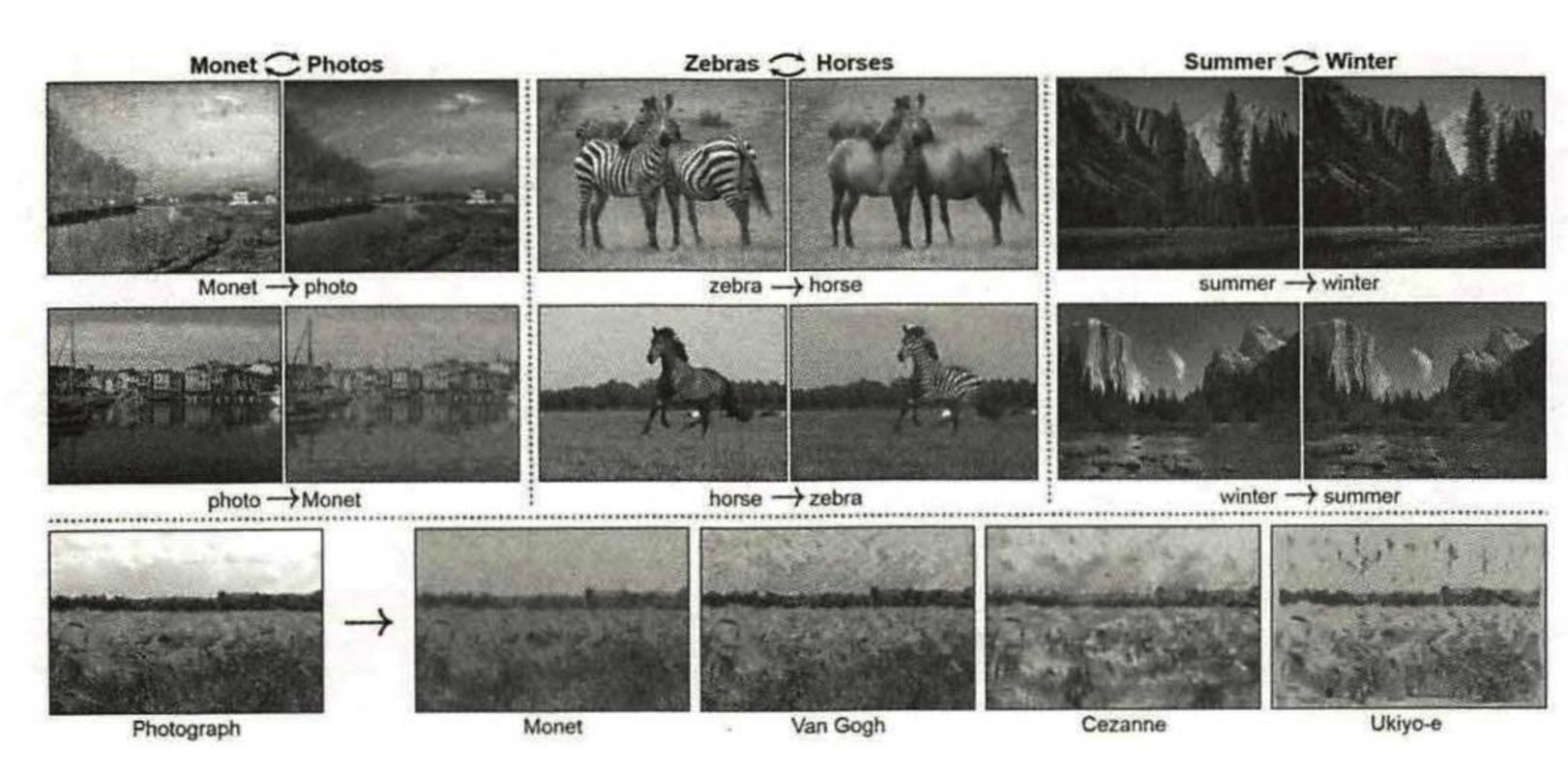


图 8-10　CycleGAN 效果图

[①] https://github.com/luanfujun/deep-photo-styletransfer

[②] https://junyanz.github.io/CycleGAN/

在本章，我们学会了如何实现一个深度学习中很酷的应用：风格迁移。不仅讲解了它的原理、风格损失和内容损失的实现，还用 PyTorch 实现了相应的代码。

9 AI 诗人：用 RNN 写诗

我们先来看一首诗。

深宫有奇物，璞玉冠何有。
度岁忽如何，遐龄复何欲。
学来玉阶上，仰望金闺籍。
习协万壑间，高高万象逼。

这是一首藏头诗，每句诗的第一句组合起来就是“深度学习”。想必你也猜到了，这首诗就是使用深度学习写的！本章我们将学习一些自然语言处理的基本概念，并尝试自己动手，用 RNN 实现自动写诗。

9.1 自然语言处理的基础知识

自然语言处理（Natural Language Processing，NLP）是人工智能和语言学领域的分支学科。自然语言处理是一个很宽泛的学科，涉及机器翻译、句法分析、信息检索等诸多研究方向。由于篇幅的限制，本章重点讲解自然语言处理中的两个基本概念词向量（Word Vector）和循环神经网络（Recurrent Neural Network，RNN）。

9.1.1 词向量

自然语言处理主要研究语言信息，语言（词、句子、篇章等）属于人类认知过程中产生的高层认知抽象实体，而语音和图像属于较底层的原始输入信号。语音、图像数据表达不需要特殊的编码，并且有天生的顺序性和关联性，近似的数字会被认为是近似的特征。正如图像是由像素组成，语言是由词或字组成，可以把语言转化为词或字表示的集合。

然而，不同于像素的大小天生具有色彩信息，词的数值大小很难表征词的含义。最初，人们为了方便，采用 One-Hot 编码格式。以一个只有 10 个不同词的语料库为例（这里只是举个例子，一般中文语料库的字平均在 8000~50000，而词则在几十万左右），我们可以用一个 10 维的向量来表示每个词，该向量在词下标位置的值为 1，而其他全部为 0。示例如下：

```
第1个词：[1,0,0,0,0,0,0,0,0,0]
第2个词：[0,1,0,0,0,0,0,0,0,0]
第3个词：[0,0,1,0,0,0,0,0,0,0]
......
第10个词：[0,0,0,0,0,0,0,0,0,1]
```

这种词的表示方法十分简单，也很容易实现，解决了分类器难以处理属性（Categorical）数据的问题。它的缺点也很明显：冗余太多、无法体现词与词之间的关系。可以看到，这 10 个词的表示，彼此之间都是相互正交的，即任意两个词之间都不相关，并且任何两个词之间的距离也都是一样的。同时，随着词数的增加，One-Hot 向量的维度也会急剧增长，如果有 3000 个不同的词，那么每个 One-Hot 词向量都是 3000 维，而且只有一个位置为 1，其余位置都是 0。虽然 One-Hot 编码格式在传统任务上表现出色，但是由于词的维度太高，应用在深度学习上时，常常出现维度灾难，所以在深度学习中一般采用词向量的表示形式。

词向量（Word Vector），也被称为词嵌入（Word Embedding），并没有严格统一的定义。从概念上讲，它是指把一个维数为所有词的数量的高维空间（几万个字，几十万个词）嵌入一个维数低得多的连续向量空间（通常是 128 或 256 维）中，每个单词或词组被映射为实数域上的向量。

词向量有专门的训练方法，这里不会细讲，感兴趣的读者可以学习斯坦福的 CS224 系列课程（包括 CS224D 和 CS224N）。在本章的学习中，读者只需要知道词向量最重要的特征是相似词的词向量距离相近。每个词的词向量维度都是固定的，每一维都是连续的数。举个例子，如果我们用二维的词向量表示十个词：足球、比赛、教练、队

伍，裤子、长裤、上衣和编织、折叠、拉，那么可视化出来的结果如图 9-1 所示。可以看出，同类的词（足球相关的词、衣服相关的词，以及动词）彼此聚集，相互之间的距离比较近。

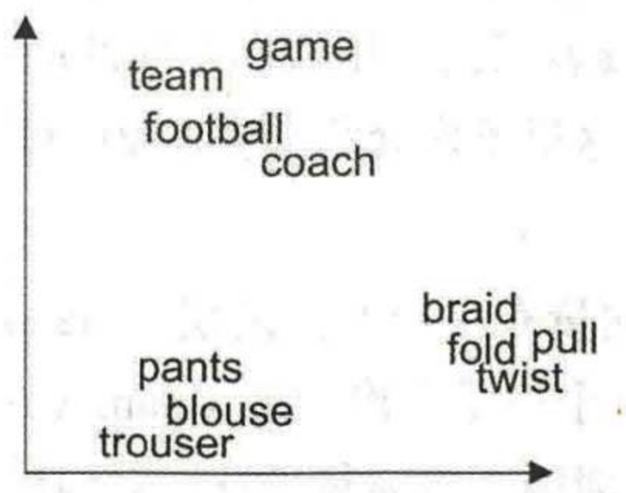


图 9-1　十个词的词向量用二维坐标表示

可见，用词向量表示词，不仅所用维度会变少（由十维变成二维），其中也会包含更合理的语义信息。除了相邻词距离更近之外，词向量还有不少有趣的特征，如图 9-2 所示，虚线的两端分别是男性词和女性词，例如叔叔和阿姨、兄弟和姐妹、男人和女人、先生和女士。可以看出，虚线的方向和长度都差不多，因此可以认为 vector(国王) – vector(女王) ≈ vector(男人) – vector(女人)，换一种写法就是 vector(国王) – vector(男人) ≈ vector(女王) – vector(女人)，即国王可以看成男性君主，女王可以看成女性君主，国王减去男性，只剩下君主的特征；女王减去女性，也只剩下君主的特征，所以这二者近似。

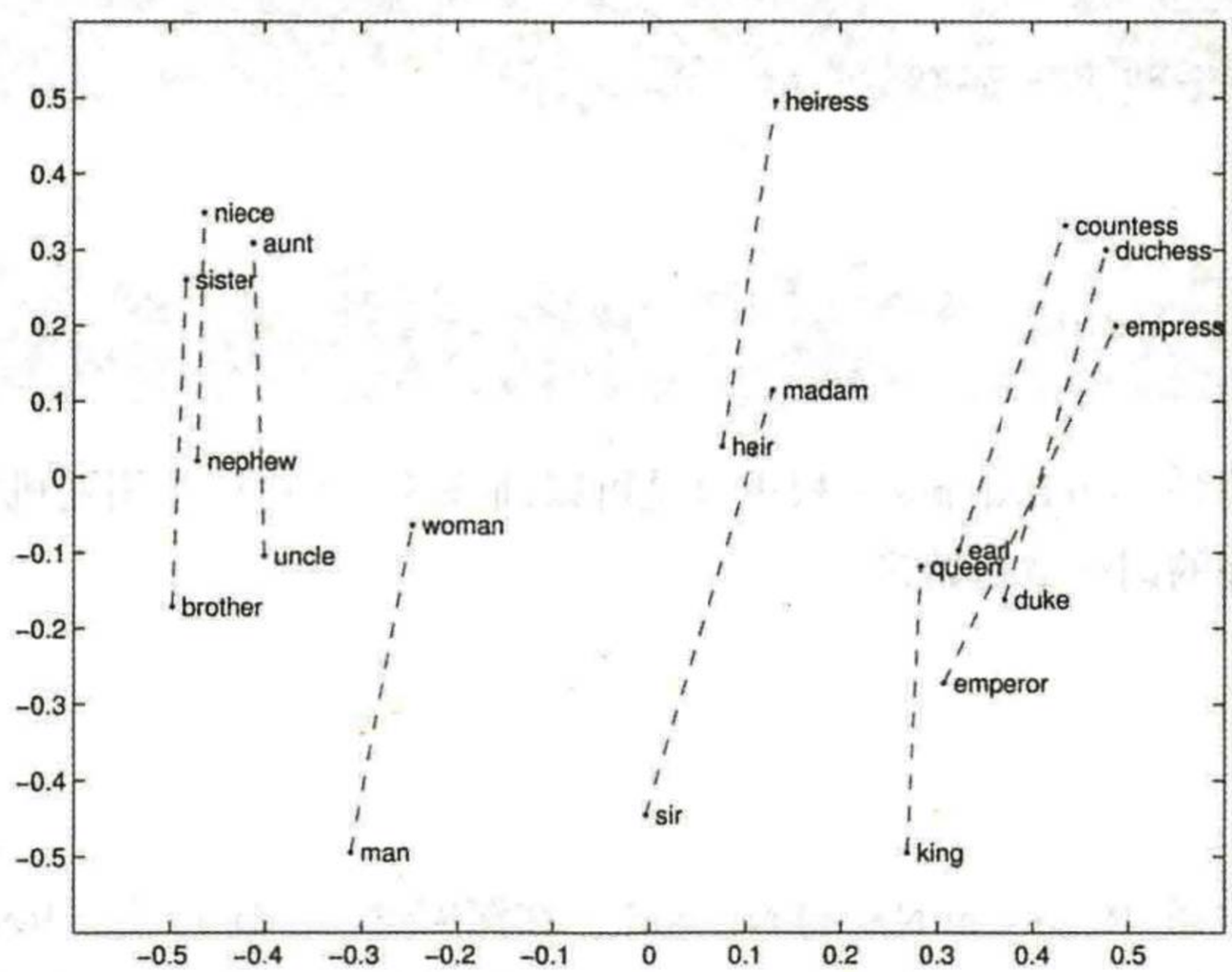


图 9-2　男性词和女性词对应的词向量

英文一般是用一个向量表示一个单词，也有使用一个向量表示一个字母的情况。中文同样也有一个词或者一个字的词向量表示，与英文采用空格来区分词不同，中文的词与词之间没有间隔，因此如果采用基于词的词向量表示，需要先进行中文分词。

这里只对词向量做一个概要性的介绍，让读者对词向量有一个直观的认知。读者只需要掌握词向量技术用向量表征词，相似词之间的向量距离近。至于如何训练词向量，如何评估词向量等内容，这里不做介绍，感兴趣的读者可以参看斯坦福大学的相关课程。

在 PyTorch 中，针对词向量有一个专门的层`nn.Embedding`，用来实现词与词向量的映射。`nn.Embedding`具有一个权重，形状是（num_words, embedding_dim），例如对上述例子中的 10 个词，每个词用 2 维向量表征，对应的权重就是一个 10×2 的矩阵。Embedding 的输入形状是 $N \times W$，N 是 batch size，W 是序列的长度，输出的形状是 $N \times W \times$ embedding_dim。输入必须是`LongTensor`，`FloatTensor` 须通过`tensor.long()`方法转成`LongTensor`。举例如下：

```
#coding:utf8
import torch as t
from torch import nn
embedding = t.nn.Embedding(10, 2) # 十个词，每个词用二维词向量表示
input = t.arange(0, 6).view(3, 2).long() # 三个句子，每句有两个词
input = t.autograd.Variable(input)
output = embedding(input)
print(output.size())
print(embedding.weight.size())
```

输出是：

```
(3L, 2L, 2L)
(10L, 2L)
```

需要注意的是，Embedding 的权重也是可以训练的，既可以采用随机初始化，也可以采用预训练好的词向量初始化。

9.1.2 RNN

RNN 的全称是 Recurrent Neural Network，在深度学习中还有一个 Recursive Neural Network 也被称为 RNN，这里应注意区分，除非特殊说明，我们所遇到的绝大多数 RNN 都是指前者。在用深度学习解决 NLP 问题时，RNN 几乎是必不可少的工具。假设我们现

在已经有每个词的词向量表示，那么我们将如何获得这些词所组成的句子的含义呢？我们无法单纯地分析一个词，因为每一个词都依赖于前一个词，单纯地看某一个词无法获得句子的信息。RNN 则可以很好地解决这个问题，通过每次利用之前词的状态（hidden state）和当前词相结合计算新的状态。

RNN 的网络结构如图 9-3 所示。

- $x_1, x_2, x_3, ..., x_T$：输入词的序列（共有 T 个词），每个词都是一个向量，通常用词向量表示。
- $h_0, h_1, h_2, h_3, ..., h_T$：隐藏元（共 $T+1$ 个），每个隐藏元都由之前的词计算得到，所以可以认为包含之前所有词的信息。h_0 代表初始信息，一般采用全 0 的向量进行初始化。
- f_W：转换函数，根据当前输入 x_t 和前一个隐藏元的状态 (h_{t-1})，计算新的隐藏元状态 h_t。可以认为 h_{t-1} 包含前 $t-1$ 个词的信息，即 $x_1, x_2, ..., x_{t-1}$，由 f_W 利用 h_{t-1} 和 x_t 计算得到的 h_t，可以认为是包含前 t 个词的信息。需要注意的是，每一次计算 h_t 都用同一个 f_W。f_W 一般是一个矩阵乘法运算。

RNN 最后会输出所有隐藏元的信息，一般只使用最后一个隐藏元的信息，可以认为它包含了整个句子的信息。

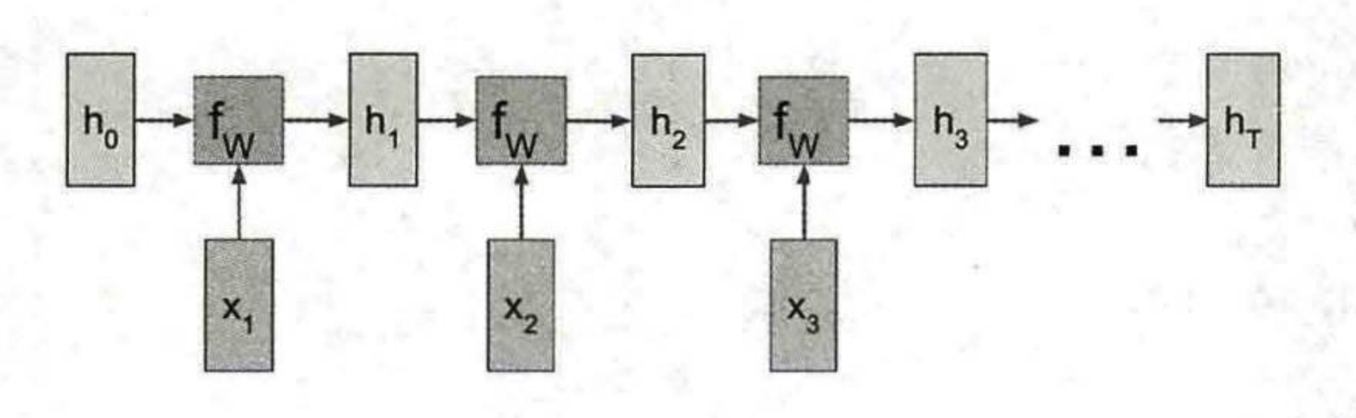


图 9-3　RNN 结构图

图 9-3 所示的 RNN 结构通常被称为 Vanilla RNN，Vanilla RNN 易于实现，并且简单直观，但却具有严重的梯度消失和梯度爆炸问题，难以训练。目前在深度学习中普遍使用的是一种被称为 LSTM 的 RNN 结构。LSTM（Long Short Term Memory Networks，长短期记忆网络）的结构如图 9-4 所示，它的结构与 Vanilla RNN 类似，也是通过不断利用之前的状态和当前的输入来计算新的状态。但其 f_W 函数更复杂，除了隐藏元状态（hidden state h），还有 cell state c。每个 LSTM 单元的输出有两个，一个是下面的 h_t（h_t 同时被创建分支引到上面去），一个是上面的 c_t。c_t 的存在能很好地抑制梯度消失和梯度爆炸等问题。关于 RNN 和 LSTM 的介绍，可以参考 colah 的博客[①]。

[①] http://colah.github.io/posts/2015-08-Understanding-LSTMs/

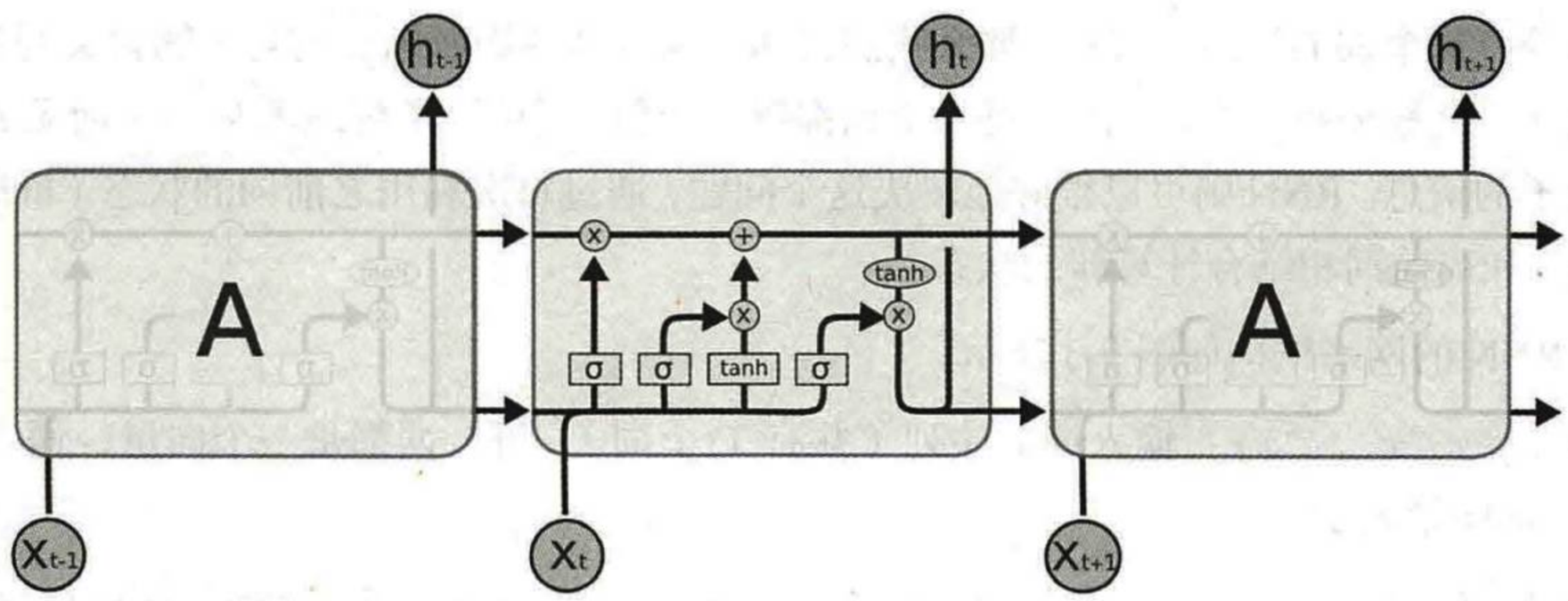


图 9-4　LSTM 结构图

LSTM 很好地解决了训练 RNN 过程中的各种问题，在几乎各类问题中都展现出远好于 Vanilla RNN 的表现。在 PyTorch 中使用 LSTM 的例子如下。

```
#coding:utf8
import torch as t
from torch import nn
from torch.autograd import Variable

# 输入词用10维词向量表示
# 隐藏元用20维向量表示
# 两层的lstm
rnn = nn.LSTM(10, 20, 2)

# 输入每句话有5个词
# 每个词由10维的词向量表示
# 总共有3句话(batch size)
input = Variable(t.randn(5, 3, 10))

# 个隐藏元(hidden state和cell state)的初始值
# 形状(num_layers, batch_size, hidden_size)
h0 = Variable(t.zeros(2, 3, 20))
c0 = Variable(t.zeros(2, 3, 20))

# output是最后一层所有隐藏元的值
# hn和cn是所有层(这里有两层)的最后一个隐藏元的值
output, (hn,cn) = rnn(input, (h0, c0))
```

```
print(output.size())
print(hn.size())
print(cn.size())
```

输出：

```
(5L, 3L, 20L)
(2L, 3L, 20L)
(2L, 3L, 20L)
```

注意：output 的形状与 LSTM 的层数无关，只和序列长度有关，而 hn 和 cn 则相反。

除了 LSTM，PyTorch 中还有 LSTMCell。LSTM 是对一个 LSTM 层的抽象，可以看成是由多个 LSTMCell 组成。而使用 LSTMCell 则可以进行更精细化的操作。LSTM 还有一种变体称为 GRU（Gated Recurrent Unit），相较于 LSTM，GRU 的速度更快，效果也接近。在某些对速度要求十分严格的场景可以使用 GRU 作为 LSTM 的替代品。

9.2 CharRNN

CharRNN 的作者 Andrej karpathy 现任特斯拉 AI 主管，也曾是优秀的深度学习课程 CS231n 的主讲人。关于 Char RNN，Andrej Karpathy 有一篇论文[①]发表于 ICLR2016，同时还有一篇相当精彩的博客[②] 介绍了不可思议的 Char RNN。

CharRNN 从海量文本中学习英文字母（注意，是字母，不是英语单词）的组合，并能够自动生成相对应的文本。例如作者用莎士比亚的剧集训练 Char RNN，最后得到一个能够模仿莎士比亚写剧的程序，生成的莎剧剧本如下：

> PANDARUS: Alas, I think he shall be come approached and the day When little srain would be attain'd into being never fed, And who is but a chain and subjects of his death, I should not sleep.
>
> Second Senator: They are away this miseries, produced upon my soul, Breaking and strongly should be buried, when I perish The earth and thoughts of many states.
>
> DUKE VINCENTIO: Well, your wit is in the care of side and that.
>
> Second Lord: They would be ruled after this chamber, and my fair nues begun out of the fact, to be conveyed, Whose noble souls I'll have the heart of the wars.

---

① Karpathy A, Johnson J, Fei-Fei L. Visualizing and understanding recurrent networks[J]. arXiv preprint arXiv:1506.02078, 2015.

② http://karpathy.github.io/2015/05/21/rnn-effectiveness/

Clown: Come, sir, I will make did behold your worship.

VIOLA: I'll drink it.

作者还做了许多十分有趣的实验，例如模仿 Linux 的源代码写程序，模仿开源的教科书的 LaTeX 源码写书等。

CharRNN 的原理其实十分简单，它分为训练和生成两部分。训练的时候如图 9-5 所示。

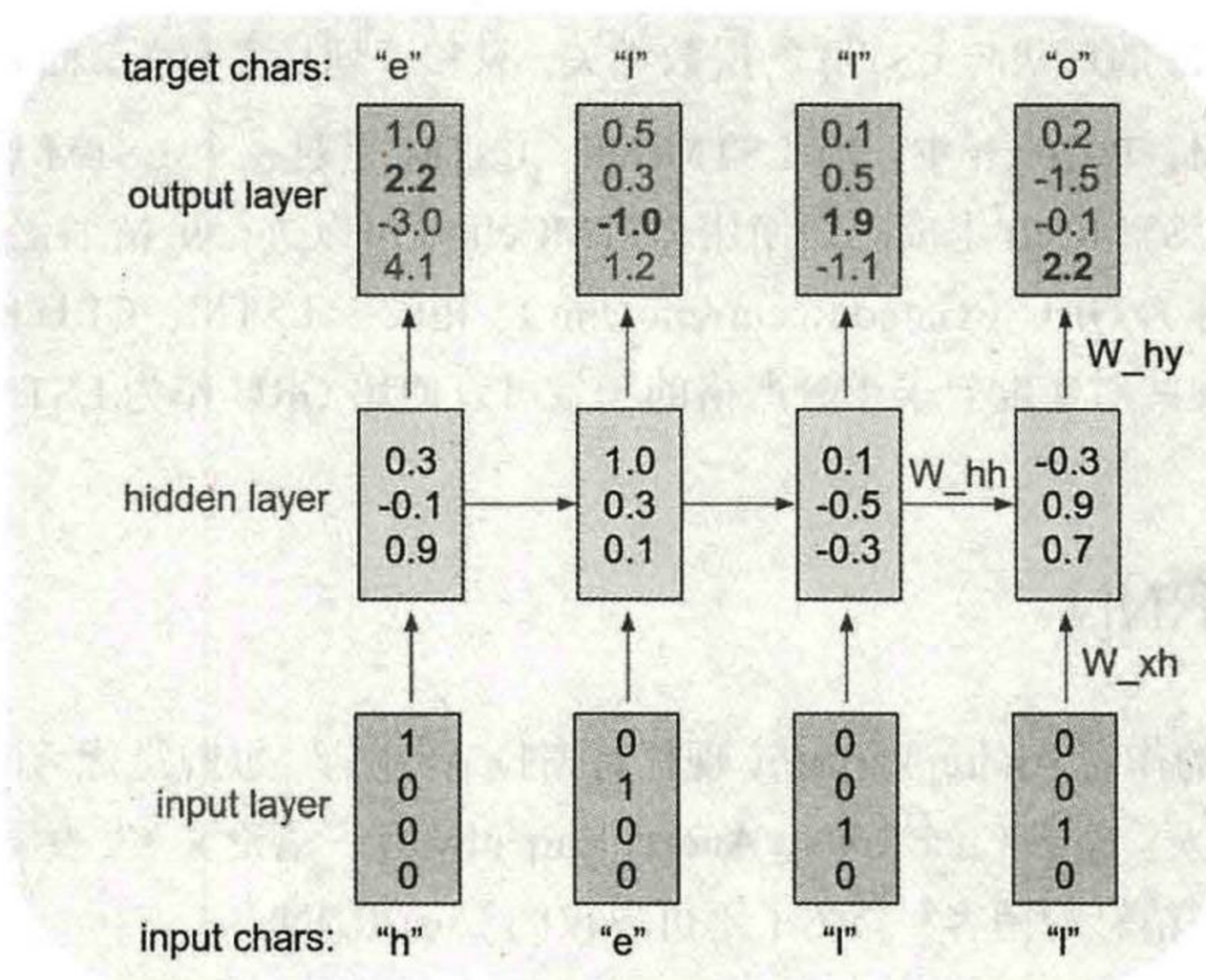


图 9-5　CharRNN 结构图

例如，莎士比亚剧本中有`hello world`这句话，可以把它转化成分类任务。RNN 的输入是`hello worl`，对于 RNN 的每一个隐藏元的输出，都接一个全连接层用来预测下一个字，即：

第一个隐藏元，输入`h`，包含`h`的信息，预测输出`e`；第二个隐藏元，输入`e`，包含`he`的信息，预测输出`l`；第三个隐藏元，输入`l`，包含`hel`的信息，预测输出`l`；第四个隐藏元，输入`l`，包含`hell`的信息，预测输出`o`；……

如上所述，CharRNN 可以看成一个分类问题：根据当前字符，预测下一个字符。对英文字符来说，文本中用到的总共只有不超过 128 个字符（假设就是 128 个字符），所以预测问题可以改成 128 分类问题：将每一个隐藏元的输出，输入到一个全连接层，计算输出属于 128 个字符的概率，计算交叉熵损失即可。

总结成一句话：CharRNN 通过利用当前字的隐藏元状态预测下一个字，把生成问题变成了分类问题。

训练完之后，我们就可以利用网络进行文本生成来写诗和剧本。生成的步骤如图 9-6 所示。

- 首先输入一个起始的字符（一般用<START>标识），计算输出属于每个字符的概率。
- 选择概率最大的一个字符作为输出。
- 将上一步的输出作为输入，继续输入到网络中，计算输出属于每个字符的概率。
- ……

最后将所有字拼接组合在一起，就得到最后的生成结果。

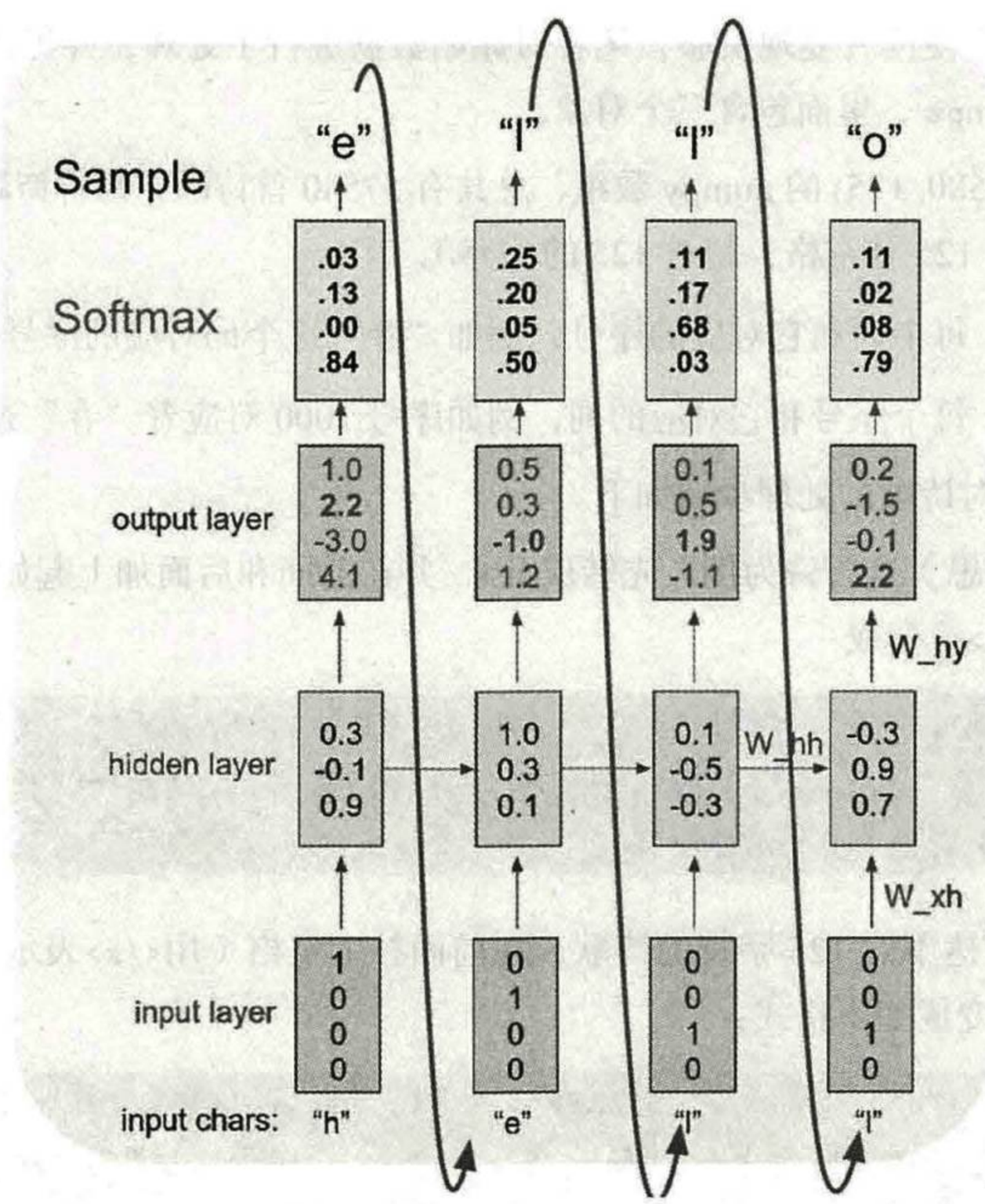


图 9-6　CharRNN 的生成步骤

CharRNN 还有一些不够严谨之处，例如它使用 One-Hot 的形式表示词，而不是使用词向量；使用 RNN 而不是 LSTM。在本次实验中，我们将对这些进行改进，并利用常用的中文语料库进行训练。

9.3 用 PyTorch 实现 CharRNN

本次实验采用的数据是来自 GitHub 上中文诗词爱好者收集的 5 万多首唐诗原文[①]。原始文件是 Json 文件和 Sqlite 数据库的存储格式。笔者在此基础上做了两个修改：

- 繁体中文改简体中文：原始数据是繁体中文的，虽然诗歌更有韵味，但是对于习惯了简体中文的读者来说可能还是有点别扭。
- 把所有的数据进行截断和补齐成一样的长度：由于不同诗歌的长度不一样，不易拼接成一个 batch，因此需要将它们处理成一样的长度。

最后为了方便读者复现实验，笔者对原始数据进行了处理，并提供了一个 numpy 的压缩包`tang.npz`，里面包含三个对象。

- data：(57580, 125) 的 numpy 数组，总共有 57580 首诗歌，每首诗歌长度为 125 字符（不足 125 补空格，超过 125 的丢弃）。
- word2ix：每个词和它对应的序号，例如“春”这个词对应的序号是 1000。
- ix2word：每个序号和它对应的词，例如序号 1000 对应着“春”这个词。

其中 data 对诗歌的处理步骤如下。

- 以《静夜思》这首诗为例，先转成 list，并在前面和后面加上起始符`<START>`和终止符`<END>`。变成：

```
['<START>', '床', '前', '明', '月', '光', ', ', '疑', '是', '地', '上
', '霜', '。', '举', '头', '望', '明', '月', ', ', '低', '头', '思',
'故', '乡', '。','<END>']
```

- 对于长度达不到 125 字符的诗歌，在前面补上空格（用`</s>`表示），直到长度达到 125，变成如下格式：

```
['</s>','</s>'........'<START>', '床', '前', '明', '月', '光', ', ',
'疑', '是', '地', '上', '霜', '。', '举', '头', '望', '明', '月', ',
', '低', '头', '思', '故', '乡', '。','<END>']
```

- 对于长度超过 125 字符的诗歌，把结尾的词截断，变成如下格式：

```
['<START>','春','江',........, '水', '流', '春', '去', '欲', '尽', ',
', '江', '潭', '落', '月', '复', '西', '斜', '。', '斜', '月', '沉',
'沉', '藏', '海', '雾', ', ', '碣', '石']
```

[①] https://github.com/chinese-poetry/chinese-poetry

- 将每个字都转成对应的序号，例如“春”转成 1000，变成如下格式，每个 list 的长度都是 125：

```
[12, 1000, 959, ....., 127, 285, 1000, 695, 50, 622, 545, 299, 3,
906, 155, 236, 828, 61, 635, 87, 262, 704, 957, 23, 68, 912, 200,
539, 819, 494, 398, 296, 94, 905, 871, 34, 818, 766, 58, 881, 469,
22, 385, 696,]
```

- 将序号 list 转成 numpy 数组。

将 numpy 的数据还原成诗歌的例子如下：

```
#coding:utf8
import numpy as np

# 加载数据
datas = np.load('tang.npz')
data = datas['data']
ix2word = datas['ix2word'].item()

# 查看第一首诗歌
poem = data[0]
# 词序号转成对应的汉字
poem_txt = [ix2word[ii]  for ii in poem]

print(''.join(poem_txt))
```

输出的结果如下所示：

```
</s></s></s>
```

```
model.py
README.md
requirements.txt
tang.npz
utils.py
```

其中几个比较重要的文件如下。

- `main.py`：包含程序配置、训练和生成。
- `model.py`：模型定义。
- `utils.py`：可视化工具 visdom 的封装。
- `tang.npz`：将 5 万多首唐诗预处理成 numpy 数据。
- `data.py`：对原始的唐诗文本进行预处理，如果直接使用`tang.npz`，则不需要对 json 的数据进行处理。

程序中主要的配置选项和命令行参数如下：

```
class Config(object):
    data_path = 'data/' # 诗歌的文本文件存放路径
    pickle_path= 'tang.npz' # 预处理好的二进制文件
    author = None # 只学习某位作者的诗歌
    constrain = None # 长度限制
    category = 'poet.tang' # 类别，唐诗还是宋诗歌(poet.song)
    lr = 1e-3
    use_gpu = True
    epoch = 20
    batch_size = 128
    maxlen = 125 # 超过这个长度之后的字被丢弃，小于这个长度的在前面补空格
    plot_every = 20 # 每20个batch可视化一次
    use_env = True # 是否使用visodm
    env='poetry' # visdom env
    max_gen_len = 200 # 生成诗歌最长长度
    debug_file='/tmp/debugp'
    model_path=None # 预训练模型路径

    # 生成诗歌相关配置
    prefix_words = '细雨鱼儿出,微风燕子斜。' # 不是诗歌的组成部分，用来控
    制生成诗歌的意境
```

```
start_words='闲云潭影日悠悠' # 诗歌开始
acrostic = False # 是否是藏头诗
model_prefix = 'checkpoints/tang' # 模型保存路径
```

在data.py中主要有以下三个函数。

- _parseRawData：解析原始的 json 数据，提取成 list。
- pad_sequences：将不同长度的数据截断或补齐成一样的长度。
- get_data：给主程序调用的接口。如果二进制文件存在，则直接读取二进制的 numpy 文件，否则读取文本文件进行处理，并将处理结果保存成二进制文件。

二进制文件tang.npz已在本书附带代码中提供，读者可以不必下载原始的 json 文件，直接加载处理好的二进制文件即可。

data.py中的get_data代码如下：

```
def get_data(opt):
    '''
    @param opt: 配置选项，Config对象
    @return word2ix: dict，每个字对应的序号，形如u'月'->100
    @return ix2word: dict，每个序号对应的字，形如'100'->u'月'
    @return data: numpy数组，每一行是一首诗对应的字的下标
    '''
    if os.path.exists(opt.pickle_path):
        data = np.load(opt.pickle_path)
        data,word2ix,ix2word  = data['data'],data['word2ix'].item(),data['
        ix2word'].item()
        return data,word2ix,ix2word

    # 如果没有处理好的二进制文件，则处理原始的json文件
    # ......
    # ......
```

这样在main.py的训练函数train中就可以这么使用数据：

```
# 获取数据
data,word2ix,ix2word = get_data(opt)
data = t.from_numpy(data)
dataloader = t.utils.data.DataLoader(data,
                batch_size=opt.batch_size,
                shuffle=True,
```

```
                num_workers=i)
```

注意，我们这里没有将 data 实现为一个Dataset对象，但是它还是可以利用DataLoader进行多线程加载。这是因为 data 作为一个 Tensor 对象，自身已经实现了__getitem__和__len__方法。data.__getitem__[0]等价于data[0]，len(data)返回data.size(0)，这种运行方式被称为鸭子类型（Duck Typing），是一种动态类型的风格。在这种风格中，一个对象有效的语义，不是由继承自特定的类或实现特定的接口决定，而是由当前方法和属性的集合决定。这个概念的名字来源于 James Whitcomb Riley 提出的鸭子测试，“鸭子测试”可以这样表述：“当看到一只鸟走起来像鸭子、游起来像鸭子、叫起来也像鸭子，那么这只鸟就可以被称为鸭子”。同理，当一个对象可以像Dataset对象一样提供__getitem__和__len__方法时，它就可以被称为Dataset。

另外需要注意的是，这种直接把所有的数据全部加载到内存的做法，在某些情况下会比较占内存，但是速度会有很大的提升，因为它避免了频繁的硬盘读写，减少了 I/O 等待。在实验中如果数据量足够小，可以酌情选择把数据全部预处理成二进制的文件并全部加载到内存中。

模型构建的代码保存在model.py中：

```
#coding:utf8
import torch
import torch.nn as nn
from torch.autograd import Variable
import torch.nn.functional as F

class PoetryModel(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_dim):
        super(PoetryModel, self).__init__()
        self.hidden_dim = hidden_dim
        self.embeddings = nn.Embedding(vocab_size, embedding_dim)
        self.lstm = nn.LSTM(embedding_dim, self.hidden_dim,num_layers=2,
        batch_  =True)
        self.linear1 = nn.Linear(self.hidden_dim, vocab_size)

    def forward(self, input,hidden=None):
        seq_len,batch_size = input.size()
        if hidden is None:
            # 2是因为有两层的LSTM
```

```
        h_0 = input.data.new(2, batch_size, self.hidden_dim).fill_(0).
        float()
        c_0 = input.data.new(2, batch_size, self.hidden_dim).fill_(0).
        float()
        h_0,c_0 = Variable(h_0),Variable(c_0)
    else:
        h_0,c_0 = hidden
    # size: (seq_len,batch_size,embeding_dim)
    embeds = self.embeddings(input)
    # output size: (seq_len,batch_size,hidden_dim)
    output, hidden = self.lstm(embeds, (h_0,c_0))

    # size: (seq_len*batch_size,vocab_size)
    output = self.linear1(output.view(seq_len*batch_size, -1))
    return output,hidden
```

总体而言，输入的字词序号经过nn.Embedding得到相应词的词向量表示，然后利用两层的 LSTM 提取词的所有隐藏元的信息，再利用隐藏元的信息进行分类，判断输出属于每一个词的概率。这里使用LSTM而不是LSTMCell是为了简化代码。当输入的序列长度为 1 时，LSTM实现的功能与LSTMCell一样。需要注意的是，这里输入（input）的数据形状是 (seq_len,batch_size)，如果输入的尺寸是 (batch_size,seq_len)，需要在输入 LSTM 之前进行转置操作 (variable.transpose)。

训练相关的代码保存于main.py中，总体而言比较简单，训练过程和第 6 章提到的猫和狗二分类问题比较相似，都是分类问题。

```
def train(**kwargs):

    for k,v in kwargs.items():
        setattr(opt,k,v)

    vis = Visualizer(env=opt.env)

    # 获取数据
    data,word2ix,ix2word = get_data(opt)
    data = t.from_numpy(data)
    dataloader = t.utils.data.DataLoader(data,
                    batch_size=opt.batch_size,
                    shuffle=True,
```

```
                    num_workers=1)

    # 模型定义
    model = PoetryModel(len(word2ix), 128, 256)
    optimizer = t.optim.Adam(model.parameters(), lr=opt.lr)
    criterion = nn.CrossEntropyLoss()

    if opt.model_path:
        model.load_state_dict(t.load(opt.model_path))

    if opt.use_gpu:
        model.cuda()
        criterion.cuda()
    loss_meter = meter.AverageValueMeter()

    for epoch in range(opt.epoch):
        loss_meter.reset()
        for ii,data_ in tqdm.tqdm(enumerate(dataloader)):
            # 训练
            data_ = data_.long().transpose(1,0).contiguous()
            if opt.use_gpu: data_ = data_.cuda()
            optimizer.zero_grad()
            ## 输入和目标错开
            input_,target = Variable(data_[:-1,:]),Variable(data_[1:,:])
            output,_  = model(input_)
            loss = criterion(output,target.view(-1))
            loss.backward()
            optimizer.step()

            loss_meter.add(loss.data[0])

            # 可视化
            if (1+ii)%opt.plot_every==0:

                if os.path.exists(opt.debug_file):
                    ipdb.set_trace()
```

```
            vis.plot('loss',loss_meter.value()[0])
            # 诗歌原文
            poetrys=[ [ix2word[_word] for _word in data_[:,_iii]]
                                for _iii in range(data_.size(1))][:16]
            vis.text('</br>'.join(['''.join(poetry)
                                    for poetry in poetrys]),win=u'
                                    origin_poem')

            gen_poetries = []
            # 分别以这几个字作为诗歌的第一个字，生成8首诗
            for word in list(u'春江花月夜凉如水'):
                gen_poetry =  ''.join(generate(model,word,ix2word,
                word2ix))
                gen_poetries.append(gen_poetry)
            vis.text('</br>'.join(['''.join(poetry)
                                    for poetry in gen_poetries]),win=u'
                                    gen_poem')

    t.save(model.state_dict(),'%s_%s.pth' %(opt.model_prefix,epoch))
```

这里需要注意的是数据，以“床前明月光”这句诗为例，输入是“床前明月”，预测的目标是“前明月光”：

- 输入“床”的时候，网络预测的下一个字的目标是“前”。
- 输入“前”的时候，网络预测的下一个字的目标是“明”。
- 输入“明”的时候，网络预测的下一个字的目标是“月”。
- 输入“月”的时候，网络预测的下一个字的目标是“光”。
- .....

这种错位的方式，通过`data_[:-1,:]`和`data_[1:,:]`实现。前者包含从第 0 个词直到最后一个词（不包括），后者是第一个词到结尾（包括最后一个词）。由于是分类问题，因此我们使用交叉熵损失作为评估函数。

接着我们来看看如何用训练好的模型写诗，第一种是给定诗歌的开头几个字接着写诗歌。实现如下：

```
def generate(model,start_words,ix2word,word2ix,prefix_words=None):
    '''
    给定几个词，根据这几个词接着生成一首完整的诗歌
```

```
    比如start_words为床前明月光，生成结果：
    床前明月光，朗朗秋风清。
    昨夜雨后人，一身一朝迎。
    何必在天末，安得佐戎庭。
    岂伊不可越，所以为我情。
    '''
    results = list(start_words) # 初始值，床前明月光
    start_word_len = len(start_words)
    # 手动设置第一个词为<START>
    input = Variable(t.Tensor([word2ix['<START>']]).view(1,1).long())
    if opt.use_gpu:input=input.cuda()
    hidden = None

    # 用以控制生成诗歌的意境和长短（五言还是七言还是四言）
    if prefix_words:
        for word in prefix_words:
            output,hidden = model(input,hidden)
            input = Variable(input.data.new([word2ix[word]])).view(1,1)

    for i in range(opt.max_gen_len):
        output,hidden = model(input,hidden)

        if i<start_word_len:
            # “床前明月光”五个字依次作为输入，计算隐藏元
            w = start_words[i]
            input = Variable(input.data.new([word2ix[w]])).view(1,1)
        else:
            # 用预测的词作为新的输入，计算隐藏元和预测新的输出
            top_index  = output.data[0].topk(1)[1][0]
            w = ix2word[top_index]
            results.append(w)
            input = Variable(input.data.new([top_index])).view(1,1)
        if w=='<EOP>':
            # 遇到结束符就结束
            break
    return results
```

这种生成方式是根据给定部分文字，然后接着完成诗歌余下的部分，生成步骤如下：

- 首先利用给定的文字“床前明月光”，计算隐藏元，并预测下一个词（预测的结果是“,”）。
- 将上一步计算的隐藏元和输出（“,”）作为新的输入，继续预测新的输出和计算隐藏元。
- 将上一步计算的隐藏元和输出作为新的输入，继续预测新的输出和计算隐藏元。
- ……

这里还有一个选项是prefix_word，可以用来控制生成的诗歌的意境和长短。比如以“床前明月光”作为start_words输入，在不指定 prefix_words 时，生成的古诗如下：

床前明月光，朗朗秋风清。昨夜雨后人，一身一朝迎。何必在天末，安得佐戎庭。岂伊不可越，所以为我情。

在指定 prefix_words 为“狂沙将军战燕然，大漠孤烟黄河骑。”的情况下，生成的古诗如下（明显带有边塞气息，而且由五言古诗变成了七言古诗）。

床前明月光照耀，城下射蛟沙漠漠。父子号犬不可亲，剑门弟子何纷纷。胡笳一声下马来，关城缭绕天河去。战士忠州十二纪，后贤美人不敢攀。

还可以生成藏头诗，实现方式如下：

```
def gen_acrostic(model,start_words,ix2word,word2ix, prefix_words = None):
    '''
    生成藏头诗
    start_words : u'深度学习'
    生成:
    深木通中岳，青苔半日脂。
    度山分地险，逆浪到南巴。
    学道兵犹毒，当时燕不移。
    习根通古岸，开镜出清蠃。
    '''

    results = []
    start_word_len = len(start_words)
    input = Variable(t.Tensor([word2ix['<START>']]).view(1,1).long())
    if opt.use_gpu:input=input.cuda()
    hidden = None

    index=0 # 用来指示已经生成了多少句藏头诗
```

```
    pre_word='<START>' # 上一个词

    if prefix_words:
        for word in prefix_words:
            output,hidden = model(input,hidden)
            input = Variable(input.data.new([word2ix[word]])).view(1,1)

    for i in range(opt.max_gen_len):
        output,hidden = model(input,hidden)
        top_index  = output.data[0].topk(1)[1][0]
        w = ix2word[top_index]

        if (pre_word  in {u'。',u'！','<START>'} ):
            # 如果遇到句号、感叹号等，把藏头的词作为下一个句的输入

            if index==start_word_len:
                # 如果生成的诗歌已经包含全部藏头的词，则结束
                break
            else:
                # 把藏头的词作为输入，预测下一个词
                w = start_words[index]
                index+=1
                input = Variable(input.data.new([word2ix[w]])).view(1,1)
        else:
            # 把上一次预测的词作为输入，继续预测下一个词
            input = Variable(input.data.new([word2ix[w]])).view(1,1)
        results.append(w)
        pre_word = w
    return results
```

生成藏头诗歌的步骤如下：

（1）输入藏头的字，开始预测下一个字。

（2）上一步预测的字作为输入，继续预测下一个字。

（3）重复第二步，直到输出的字是“。”或者“！”，说明一句诗结束了，可以继续输入下一句藏头的字，跳到第一步。

（4）重复上述步骤直到所有藏头的字都输入完毕。

上述两种生成诗歌的方法还需要提供命令行接口，实现方式如下：

```
def gen(**kwargs):
    '''
    提供命令行接口，用以生成相应的诗
    '''

    for k,v in kwargs.items():
        setattr(opt,k,v)

    # 加载数据和模型
    data,word2ix,ix2word = get_data(opt)
    model = PoetryModel(len(word2ix), 128, 256);
    map_location = lambda s,l:s
    state_dict = t.load(opt.model_path,map_location=map_location)
    model.load_state_dict(state_dict)

    if opt.use_gpu:
        model.cuda()

    # 编码问题，半角改成全角，古诗中都是全角符号
    start_words = opt.start_words.decode('utf8').replace(',',u', ')\
                                    .replace('.',u'。')\
                                    .replace('?',u'？')
    prefix_words = opt.prefix_words.decode('utf8') if opt.prefix_words
    else None

    # 判断是藏头诗还是普通诗歌
    gen_poetry = gen_acrostic if opt.acrostic else generate

    result = gen_poetry(model,start_words,ix2word,word2ix,prefix_words)
    print(''.join(result))
```

9.4 实验结果分析

训练的命令如下：

```
python main.py train --plot-every=150
                     --batch-size=128
                     --pickle-path='tang.npz'
                     --lr=1e-3
                     --env='poetry3'
                     --epoch=50
```

生成藏头诗的命令如下：

```
python main.py gen  --model-path='model.pth'
                    --pickle-path='tang.npz'
                    --start-words='深度学习' # 藏头词
                    --prefix-words='江流天地外，山色有无中。'
                    --acrostic=True # 藏头诗
深居不可见，浩荡心亦同。度年一何远，宛转三千雄。学立万里外，诸夫四十功。习
习非吾仕，所贵在其功。
```

生成其他诗歌的命令如下：

```
python2 main.py gen  --model-path='model.pth'
                     --pickle-path='tang.npz'
                     --start-words='江流天地外，' # 诗歌的开头
                     --prefix-words='郡邑浮前浦，波澜动远空。'
江流天地外，风日水边东。稍稍愁蝴蝶，心摧芒范蓬。云飞随海远，心似汉阳培。按
俗朝廷上，分军朔雁通。封疆朝照地，赐劒豫章中。畴昔分曹籍，高名翰墨场。翰林
推国器，儒冠见忠贞。阜宙非无事，姦邪亦此中。渥仪非贵盛，儒实不由锋。几度沦
亡阻，千年垒数重。宁知天地外，长恐海西东。邦测期戎逼，箫韶故国通。蜃楼瞻凤
篆，云辂接旌幢。别有三山里，来随万里同。烟霞临海路，山色落云中。渥泽三千
里，青山万古通。何言陪宴侣，复使
```

生成的很多诗歌都是高质量的，有些甚至已经学会了简单的对偶和押韵。例如：

> 落帆迷旧里，望月到西州。浩荡江南岸，高情江海鸥。风帆随雁吹，江月照旌楼。泛泛扬州客，停舟泛水鸥。

很有意思的是，如果生成的诗歌长度足够长，会发现生成的诗歌意境会慢慢改变，以至于和最开始的毫无关系，例如：

> 大漠孤烟照高阁，夹城飞鞚连天阙。青丝不语不知音，一曲繁华空绕山。昔年曾作江南客，今日相逢不相识。今年花落花满园，妾心不似君不同。回头

舞马邯郸陌，回头笑语歌声闹。夫君欲问不相见，今日相看不相见。君不见
君心断断肠，莫言此地情何必？桃花脉脉不堪惜，君恩不惜春光色。

一开始是边塞诗，然后变成羁旅怀人，最后变成了闺怨诗。

意境、格式和韵脚等信息都保存在隐藏元之中，随着输入的不断变化，隐藏元保存的信息也在不断变化，有些信息即使经过了很长的时间依旧可以保存下来（比如诗歌的长短，五言还是七言），而有些信息随着输入变化也发生较大的改变。在本程序中，我们使用`prefix_words`就是为了网络能够利用给定的输入初始化隐藏元的状态。事实上，隐藏元中的每一个数都控制着生成诗歌的某一部分属性，感兴趣的读者可以尝试调整隐藏元的数值，观察生成的诗歌有什么变化。

总体上，程序生成的诗歌效果还不错，字词之间的组合也比较有意境，但是诗歌缺乏一个一以贯之的主题，读者很难从一首诗歌中得到一个主旨。这是因为随着诗歌长度的增长，即使是 LSTM 也不可避免地忘记几十个字之前的输入。另外一个比较突出的问题就是，生成的诗歌中经常出现重复的词，这在传统的诗歌创作中应该是极力避免的现象，而在程序生成的诗歌中却常常出现。

本章介绍了自然语言处理中的一些基础概念，并带领读者实现了一个能够生成古诗的小程序。程序从唐诗中学习，并模仿古人写出了不少优美的诗句。

10

Image Caption：让神经网络看图讲故事

Image Caption，通常被翻译为图像描述，也有人称之为图像标注，本章统一译为图像描述。图像描述直观地解释就是从给定的图像生成一段描述文字。图 10-1 所示就是几个图像描述的例子，上面是图像，下面是神经网络生成的相应的描述。图像描述是深度学习中十分有趣的一个研究方向，也是计算机视觉的一个关键目标。对于图像描述的任务，神经网络不仅要了解图中有哪些对象、对象之间的关系，还要使用自然的语言来描述这些对象的关系，因此图像描述比其他深度学习任务更有趣，也更有挑战性。

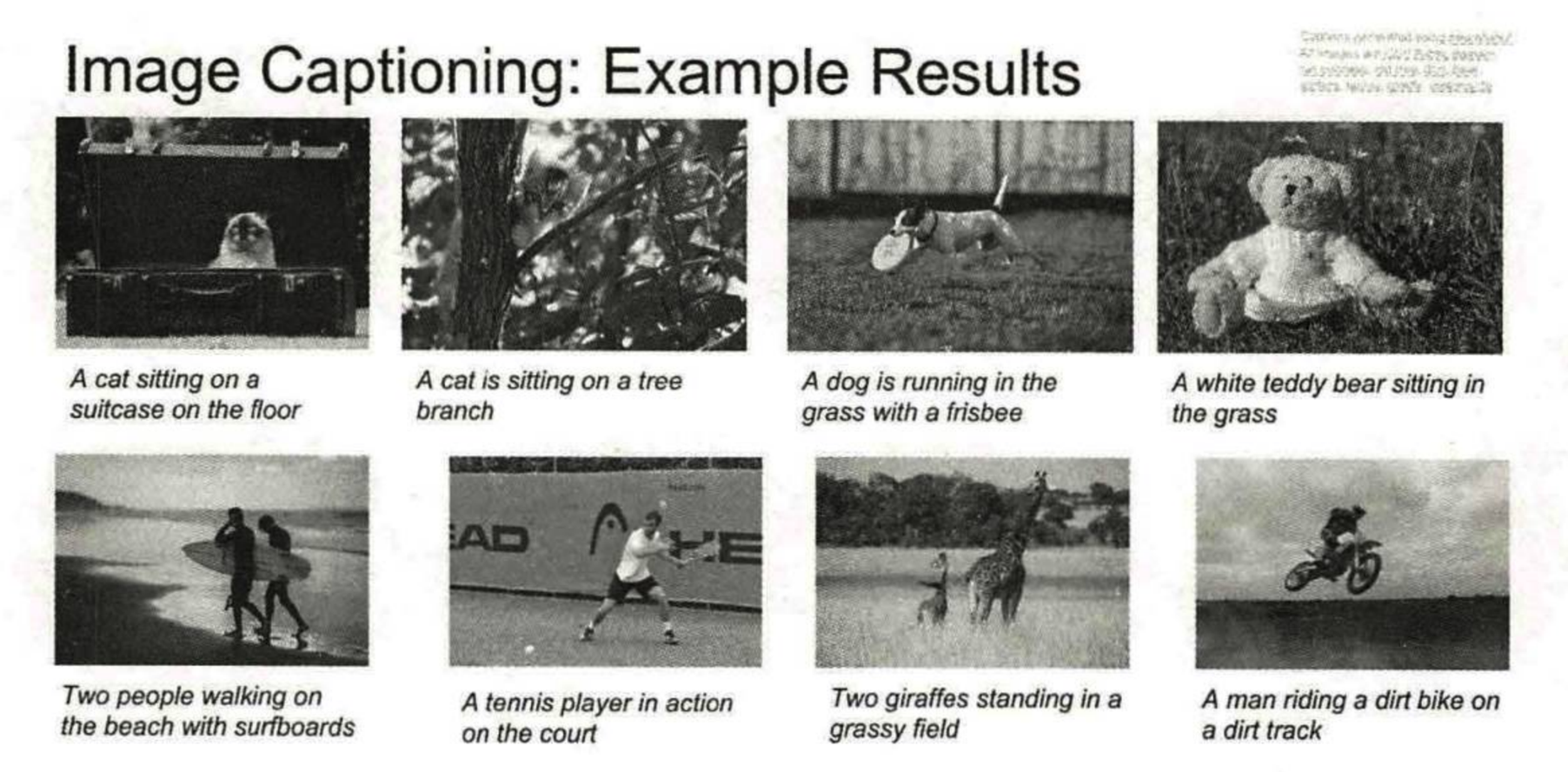


图 10-1　图像描述的例子

图像描述用到的数据集通常是 MS COCO，COCO 数据集使用的是英文语料库，而在本章中，我们将使用 2017 年 9 月 ~12 月举办的 AI Challenger 比赛中的“图像中文描述”子任务的数据，带读者实现一个能够看图描述的神经网络。关于图像中文描述比赛的更多信息，读者可访问 AI Challenger 官网[①]。

10.1 图像描述介绍

对人来说，Image Caption 是简单而自然的一件事，但对机器来说，这项任务充满了挑战性。原因在于机器不仅要能检测出图像中的物体，还要理解物体之间的相互关系，最后还要用合理的语言表达出来。图像描述吸引了许多研究人员的关注，除了它的趣味性外，还因为它具有广阔的应用前景，例如它可以帮助盲人“看”到真实世界发生的事情。

利用深度学习完成图像描述的工作可以追溯到 2014 年百度研究院发表的 *Explain Images with Multimodal Recurrent Neural Networks*[②]论文，作者创造性地将深度卷积神经网络和深度循环神经网络结合，用于解决图像标注与图像和语句检索等问题。现在关于图像描述更为人所熟知的是另外一篇论文：*Show and tell: A neural image caption generator*[③]。这篇论文提出的 Caption 模型如图 10-2 所示，image 是原始图片，左边是 GoogLeNet，实际使用中可以用任意的深度学习网络结构代替（例如 VGG、ResNet 等），S_0 ,S_1 ,S_2 ,..., S_N 是人工对图片进行描述的语句，例如“A dog is playing with a ball”，那么 $S_0 \sim S_6$ 就是这 7 个单词。$W_e S_n$ 就是这几个词对应的词向量。

论文中训练的方法如下。

- 图片经过神经网络提取到图片高层次的语义信息 f
- 将 f 输入到 LSTM 中，并希望 LSTM 的输出是 S_0
- 将 S_0 输入到 LSTM 中，并希望 LSTM 的输出是 S_1
- 将 S_1 输入到 LSTM 中，并希望 LSTM 的输出是 S_2
- 将 S_2 输入到 LSTM 中，并希望 LSTM 的输出是 S_3
- .....
- 将 S_{N-1} 输入到 LSTM 中，并希望 LSTM 的输出是 S_N

---

① https://challenger.ai/competition/caption

② Mao J, Xu W, Yang Y, et al. Explain images with multimodal recurrent neural networks[J]. arXiv preprint arXiv:1410.1090, 2014.

③ Vinyals O, Toshev A, Bengio S, et al. Show and tell: A neural image caption generator[C]//Proceedings of the IEEE conference on computer vision and pattern recognition. 2015: 3156-3164.

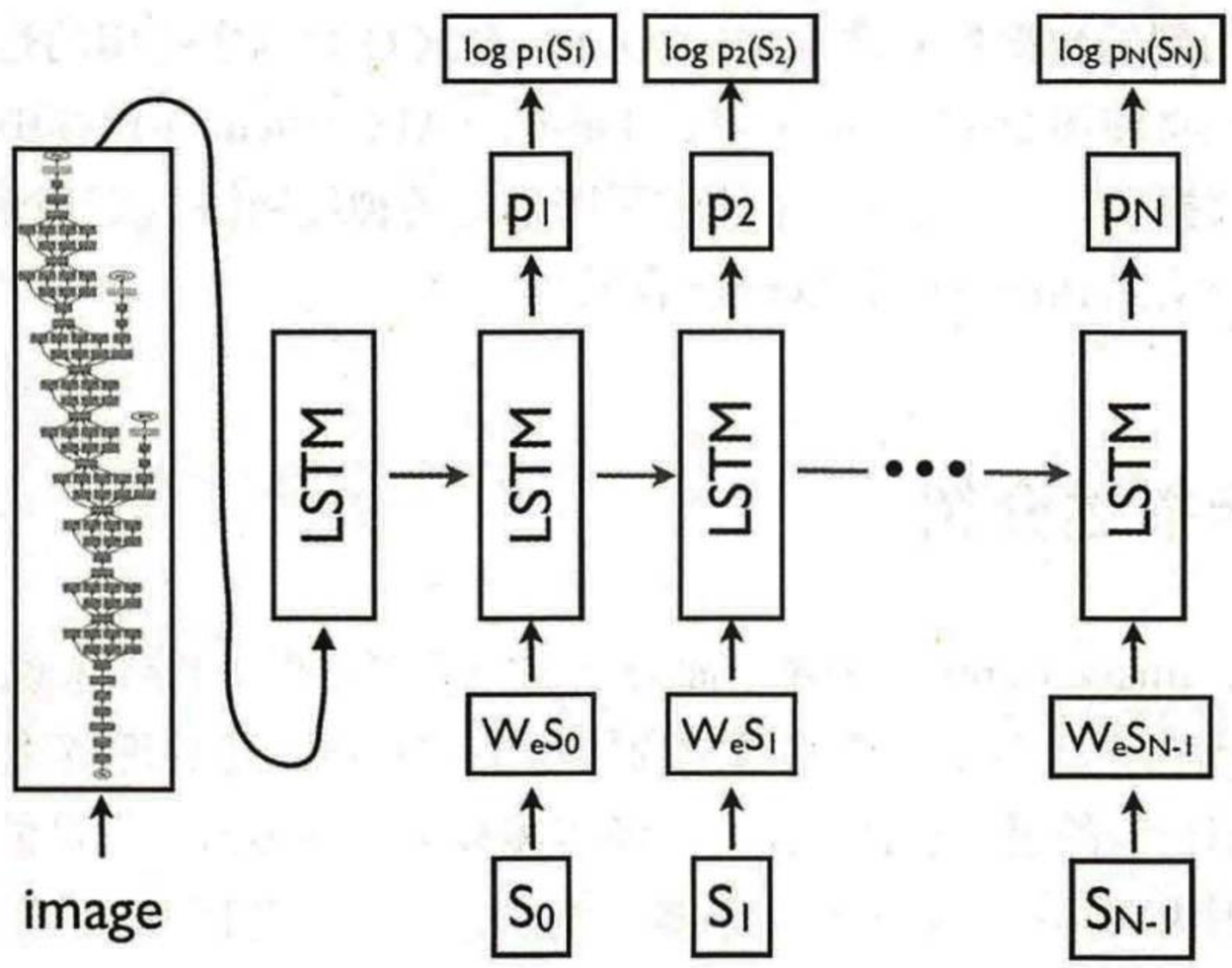


图 10-2　Show and Tell 的图像描述模型

可以看出这个做法和第 9 章所讲的利用 CharRNN 写唐诗的原理很相似。如果读者还没有阅读第 9 章的内容，笔者强烈建议先阅读它，了解关于词向量、RNN 和 CharRNN 的相关知识。图像描述的实现方法可以认为和生成唐诗一样，只不过这里的第一个词用图像的信息来表示。通过神经网络高层的输出，我们可以获得关于图像的高层语义信息。在论文中，作者使用了预训练好的 GoogLeNet 获取图片在全连接分类层之前的输出，作为图像语义。训练的目标就是输出的词尽量和预期的词相符，所以图像描述问题最终也变成了一个分类问题，利用 LSTM 不断预测下一个最有可能出现的词。

10.2　数据

10.2.1　数据介绍

AI Challenger 图像中文描述比赛的数据分为两部分，第一部分是图片，总共 20 万张，第二部分是一个 `caption_train_annotations_20170902 .json`文件，它以 json 的格式保存每张图片的描述，每个样本的格式如下，总共有 20 万条这样的样本。

- `url`：图片的下载地址（没用，因为已经提供了下载好的图片）。
- `image_id`：图片的文件名。
- `caption`：图片对应的五句中文描述。

```
{"url": "http://m4.biz.itc.cn/pic/new/n/71/65/Img8296571_n.jpg",
 "image_id": "8f00f3d0f1008e085ab660e70dffced16a8259f6.jpg",
 "caption": ["\u4e24\u4e2a\u8863\u7740\u4f11\u95f2\u7684\u4eba\u5728\u5e73
 \u6574\u7684\u9053\u8def\u4e0a\u4ea4\u8c08", "\u4e00\u4e2a\u7a7f\u7740\
 u7ea2\u8272\u4e0a\u8863\u7684\u7537\u4eba\u548c\u4e00\u4e2a\u7a7f\u7740\
 u7070\u8272\u88e4\u5b50\u7684\u7537\u4eba\u7ad9\u5728\u5ba4\u5916\u7684\
 u9053\u8def\u4e0a\u4ea4\u8c08", "\u5ba4\u5916\u7684\u516c\u56ed\u91cc\
 u6709\u4e24\u4e2a\u7a7f\u7740\u957f\u88e4\u7684\u7537\u4eba\u5728\u4ea4\
 u6d41", "\u8857\u9053\u4e0a\u6709\u4e00\u4e2a\u7a7f\u7740\u6df1\u8272\
 u5916\u5957\u7684\u7537\u4eba\u548c\u4e00\u4e2a\u7a7f\u7740\u7ea2\u8272\
 u5916\u5957\u7684\u7537\u4eba\u5728\u4ea4\u8c08", "\u9053\u8def\u4e0a\
 u6709\u4e00\u4e2a\u8eab\u7a7f\u7ea2\u8272\u4e0a\u8863\u7684\u7537\u4eba\
 u5728\u548c\u4e00\u4e2a\u62ac\u7740\u5de6\u624b\u7684\u4eba\u8bb2\u8bdd
 "]},
```


示例图片如图 10-3 所示。这张图片有对应的五句描述。

- 一群人站在舞台上，穿红裙子的女人正在话筒前讲话。
- 一个穿红裙子和一个穿青衣服的女性站在一群人前，穿红裙子的女人正在讲话。
- 一群女人和一群穿西装的男人站在舞台上。
- 一群人前面一个手里拿着奖杯的女人在讲话。
- 一个穿白色裙子的女人旁边有一个穿红裙子的女人正在讲话。

图 10-3　艾美奖颁奖典礼（图片来自艾美奖官网）

我们可以看出以上描述有几个特点。

- 每句描述的长短不一。

- 描述不涉及太多额外知识，尽可能客观（例如不知道这张图片中的人是《大小谎言》剧组）。
- 尽可能点名图片中的人物，以及人物之间的关系等。

数据处理主要涉及对图片的预处理和对描述的预处理。对图片的预处理相对比较简单，将图片送入 ResNet，获得指定层的输出并保存即可。

对文字的预处理相对比较麻烦，分为以下几步：

- 中文分词。
- 将词用序号表示（word2ix），并过滤低频词。
- 将所有描述补齐到等长（pad_sequence）。
- 利用 `pack_padded_sequence` 进行计算加速。

第一步是对中文数据进行分词。不同于英文通过空格区分单词，中文的词与词之间没有明显的分界线。研究发现，如果不对中文语料进行分词操作，直接对基于字的数据进行处理，效果会大打折扣。目前有不少的中文分词软件，其中最优秀的就是结巴分词[①]，结巴分词功能强大、使用简单而且效果出众，读者通过`pip install jieba`安装之后即可使用，一个简单的使用例子如下：

```
import jieba
seg_list = jieba.cut("我正在学习深度学习知识", cut_all=False)
print(u"分词结果: " + "/ ".join(seg_list))  # 精确模式
```

输出：

```
分词结果: 我/ 正在/ 学习/ 深度/ 学习/ 知识
```

结巴分词利用自建的词典进行分词，读者还可以指定自己自定义的词典，以便包含结巴词库里没有的词。虽然结巴有新词识别能力，但是自行添加新词可以保证更高的正确率。例如“深度学习”就因为不在词库中被分成了两个词，其实应该属于一个词。

第二步过滤低频词比较简单，就是统计每个词出现的次数，然后删除一些频次太低的词。

第三步也比较简单，我们在第 9 章讲过了如何将不同长度的诗歌 pad 成一样的长度。

第四步需要用到 PyTorch 中比较特殊的函数 `pack_padded_sequence`。这个函数专门对经过pad操作后的序列进行pack操作。经过 pad 操作后的序列中有许多空白的填充值，在计算 RNN 隐藏元时，这些空白的填充值也会进行计算，这种计算没有必要，浪费了

[①] https://github.com/fxsjy/jieba

太多的计算资源，而且计算太多的空白值可能会影响隐藏元的取值。PackedSequence 为解决这个问题专门设立了一个解决方案，它能够知道输入的数据中哪些是 pad 的值，不会计算 pad 的数据的输出，从而节省计算资源。使用方法如下：

- 对不同长度的句子，按长度（从长到短）进行排序并记录每个句子的长短。
- 对不同的句子，统一 pad 成一样的长度。
- 将上一步得到的 variable 和样本的长度输入 pack_padded_sequence，会输出一个 PackedSequence 对象，这个对象可以输入到任何 RNN 类型的 module（包括 RNN、LSTM 和 GRU），还能送入部分损失函数中（例如交叉熵损失函数）。
- PackedSequence 对象可以通过 pad_packed_sequence 方法取出 variable 和 length。这个操作可以看成是 pack_padded_sequence 的逆操作，但是一般不需要取出来，而是直接通过全连接层计算损失。

一个简单的使用例子如下：

```
In [1]: import torch as t
   ...: from torch.nn.utils.rnn import pack_padded_sequence,
  pad_packed_sequence
   ...: from torch import nn


In [2]: # dummy data  4句不同长短的句子
   ...: sen1 = [1,1,1]
   ...: sen2 = [2,2,2,2]
   ...: sen3 = [3,3,3,3,3,3,3,3,3]
   ...: sen4 = [4,4,4,4,4]
   ...: sentences= [sen1,sen2,sen3,sen4]
   ...: sentences
   ...:
Out[2]: [[1, 1, 1], [2, 2, 2, 2], [3, 3, 3, 3, 3, 3, 3, 3, 3], [4, 4, 4,
4, 4]]


In [3]: # 按长短排序
   ...: sentences = sorted(sentences,key=lambda x:len(x),reverse=True)
   ...: sentences
   ...:
Out[3]: [[3, 3, 3, 3, 3, 3, 3, 3, 3], [4, 4, 4, 4, 4], [2, 2, 2, 2], [1,
1, 1]]
```

```
In [4]: # 长于5个词截断到5个词
   ...: lengths = [ 5 if len(sen)>5 else len(sen) for sen in sentences]
   ...: lengths
   ...:
Out[4]: [5, 5, 4, 3]


In [5]: # pad数据，太长的就截断，太短的补0
   ...: def pad_sen(sen,length=5,padded_num=0):
   ...:     origin_len = len(sen)
   ...:     padded_sen = sen[:length]
   ...:     padded_sen = padded_sen + [padded_num for _ in range(
   origin_len,length) ]
   ...:     return padded_sen
   ...: pad_sentences = [pad_sen(sen) for sen in sentences]
   ...: pad_sentences
   ...:
Out[5]: [[3, 3, 3, 3, 3], [4, 4, 4, 4, 4], [2, 2, 2, 2, 0], [1, 1, 1, 0,
0]]


In [6]: # 4*5 batch_size = 4, 词=5
   ...: pad_tensor  = t.Tensor(pad_sentences).long()
   ...: # 5*4, batch_size = 4, 词=5
   ...: pad_tensor = pad_tensor.t()
   ...: pad_variable = t.autograd.Variable(pad_tensor)
   ...: pad_variable # 一列是一句话
   ...:
Out[6]:
Variable containing:
 3  4  2  1
 3  4  2  1
 3  4  2  1
 3  4  2  0
 3  4  0  0
[torch.LongTensor of size 5x4]


In [7]: # 总共5个词，每个词用2维向量表示
```

```
   ...: embedding = nn.Embedding(5,2)
   ...: # 5*4*2
   ...: pad_embeddings = embedding(pad_variable)
   ...: pad_embeddings
   ...:
Out[7]:
Variable containing:
(0 ,.,.) =
  0.2370  1.4615
 -0.7361 -1.1946
  0.5281 -0.9638
 -1.7804 -0.9428
......
(4 ,.,.) =
  0.2370  1.4615
 -0.7361 -1.1946
  0.4237 -0.6647
  0.4237 -0.6647
[torch.FloatTensor of size 5x4x2]

In [8]: # pack数据
   ...: packed_variable = pack_padded_sequence(pad_embeddings,lengths)
   ...: packed_variable # 输出也是PackedSequence
   ...:
Out[8]:
PackedSequence(data=Variable containing:
 0.2370  1.4615
-0.7361 -1.1946
....
 0.2370  1.4615
-0.7361 -1.1946
[torch.FloatTensor of size 17x2]
, batch_sizes=[4, 4, 4, 3, 2])

In [9]: # 输入2维(词向量长度)，隐藏元长度为3
   ...: rnn = t.nn.LSTM(2,3)
   ...: output,hn = rnn(packed_variable)
```

```
   ...: output = pad_packed_sequence(output)
   ...: output
   ...:
Out[9]:
(Variable containing:
 (0 ,.,.) =
   0.1807 -0.1893  0.1730
   0.1199 -0.1013  0.1577
   0.1766 -0.0798  0.1885
   0.0404 -0.1340  0.0459
....
 (4 ,.,.) =
   0.2645 -0.3434  0.2604
   0.2051 -0.2379  0.3338
   0.0000  0.0000  0.0000
   0.0000  0.0000  0.0000
 [torch.FloatTensor of size 5x4x3], [5, 5, 4, 3])
```

为了方便读者阅读，笔者对 caption_train_annotations_20170902 .json进行了一些基础处理，保存成 pickle 格式的 dict 对象。

```
In [1]: import torch as t
   ...: data = t.load('caption.pth')
   ...: list(data.keys())
   ...:
Out[1]: ['word2ix', 'end', 'id2ix', 'ix2id', 'ix2word', 'padding', '
caption', 'readme']

In [2]: print(data['readme'])
    word: 词
    ix:index
    id:图片名
    caption: 分词之后的描述，通过ix2word可以获得原始中文词
```

字典中各个键值对的含义表征如下。

- word2ix：长度为 10003 的字典，词对应的序号，例如“衣服”→ 10。
- ix2word：长度为 10003 的字典，序号对应的词，例如 10 →“衣服”。

- id2ix：长度为 200000 的字典，图片文件名对应的序号，例如“8f00f259f6.jpg”→0。
- ix2id：长度为 200000 的字典，序号对应的图片文件名，例如 0 →“8f00fa8259f6.jpg”
- end：结束标识符</EOS>。
- padding：pad 标识符</PAD>。
- caption：长度为 200000 的列表，每一项是个长度为 5 的列表，保存图片都应为五句描述。描述的数据经过分词，并将词映射到序号。可以通过word2ix查看词和序号的对应关系。

一个使用例子如下：

```
# coding:utf8
import torch as t

data = t.load('caption.pth')

ix2word = data['ix2word']
ix2id = data['ix2id']
caption = data['caption']

img_ix = 100 # 第100张图片
# 图片对应的描述
img_caption = caption[img_ix]

# 图片文件名
# u'0d9ca297dac4bcb577b09ef9d479cbc6a043db89.jpg'
img_id = ix2id[img_ix]

# 图片的第一句描述
# 形如[17, 10, 4, 24, 73, 3, 11, 5, 125]
# 通过ix2word获得对应的词
sen = img_caption[0]
sen = [ix2word[_] for _ in sen]
print(''.join(sen)) # 输出球场上有一个左手抬起的女人在打网球
```

详细的处理逻辑可以查看scripts/data_preprocess.py。

10.2.2 图像数据处理

我们需要利用神经网络提取图像的特征，具体来说就是利用 ResNet 提取图片在倒数第二层（池化层的输出，全连接层的输入）的 2048 维度的向量。为了提取 ResNet 这一层的输出，我们有两种解决方案，第一种是复制并修改 `torchvision` 中的 `ResNet` 源码，让他在导数第二层就输出返回；第二种更简单，直接把最后一层删除并替换成一个恒等映射。

在 torchvision 中，ResNet 的 forward 函数源码如下，我们的目标是获得 `x=self.avgpool(x)` 的输出。

```
def forward(self, x):
    x = self.conv1(x)
    x = self.bn1(x)
    x = self.relu(x)
    x = self.maxpool(x)

    x = self.layer1(x)
    x = self.layer2(x)
    x = self.layer3(x)
    x = self.layer4(x)

    x = self.avgpool(x)
    x = x.view(x.size(0), -1)
    x = self.fc(x)

    return x
```

第一种做法是修改 forward 函数。

```
from torchvision.models import resnet50
def new_forward (self,x):
  x = self.conv1(x)
  .....
  x = self.avgpool(x)
  x = x.view(x.size(0), -1)
  # 注释或删除下面这行
  # x = self.fc(x)
  return x
```

```
model = resnet50(pretrained=True)

# 用新的forward函数覆盖旧的forward函数
model.forward = lambda x:new_forward(model,x)
# 或者 model.__class__.forward = new_forward
```

第二种做法是删除 model 的全连接层，将它改成一个恒等映射。

```
model = resnet50(pretrained=True)
del model.fc
model.fc = lambda x:x
```

第二行删除了全连接层，但是由于 model 没有了`fc`属性，在 forward 函数中会报错，所以这里用一个恒等映射代替`fc`。

修改完 ResNet 的结构之后，就可以提取 20 万张图片的 feature，代码如下：

```
# 数据，获取20万张图片
opt.batch_size=256
dataloader = get_dataloader(opt)
results = t.Tensor(len(dataloader.dataset),2048).fill_(0)
batch_size = opt.batch_size

# 模型
resnet50 = tv.models.resnet50(pretrained=True)
del resnet50.fc
resnet50.fc = lambda x:x
resnet50.cuda()

# 前向传播，获取feature
for ii,(imgs, indexs)  in tqdm.tqdm(enumerate(dataloader)):
    # 确保图片的序号没有对应错
    assert indexs[0]==batch_size*ii
    imgs = imgs.cuda()
    imgs = Variable(imgs,volatile=True)
    features = resnet50(imgs)
    results[ii*batch_size:(ii+1)*batch_size]= features.data.cpu()

# shape: 200000*2048 20万张图片，每张图片2048维
t.save(results,'results.pth')
```

这里需要注意的是 dataloader 不要 shuffle，要按顺序，这样才能和`ix2id`中的序号和图片文件名一一对应，即特征矩阵`results[ix]`中保存图片的文件名为`ix2id[ix]`。

10.2.3 数据加载

数据加载的第一步需要将数据封装成 dataset，这部分的代码实现如下（具体信息都在注释中说明，需要注意的是这里利用`dataset.train()`函数进行训练集和验证集之间的切换）。

```
class CaptionDataset(data.Dataset):

    def __init__(self,opt,transforms=None):
        '''
        Attributes:
            _data (dict): 预处理之后的数据，包括所有图片的文件名和人工描述
            all_imgs (tensor): 形状是（200000，2048），利用resnet50提取的
            图片特征
            caption(list): 长度为20万的list，包括每张图片的描述
            ix2id(dict): 指定序号的图片对应的文件名
            start_(int): 起始序号。训练集的起始序号是0，验证集的起始序号是
            190000，即
                前190000张图片是训练集，剩下的10000张图片是验证集
            len_(init): 数据集大小。如果是训练集，长度就是190000，验证集长
            度为10000
            traininig(bool): 是训练集(True)还是验证集(False)
        '''
        self.opt = opt
        data = t.load(opt.caption_data_path)
        word2ix = data['word2ix']
        self.captions = data['caption']
        self.padding = word2ix.get(data.get('padding'))
        self.end = word2ix.get(data.get('end'))
        self._data = data
        self.ix2id = data['ix2id']
        self.all_imgs = t.load(opt.img_feature_path)
        self.train(True)
```

```
    def __getitem__(self,index):
        '''
        返回:
        - img: 图像feature 2048的向量
        - caption: 描述。形如LongTensor([1,3,5,2]),长度取决于描述长度
        - index: 下标。图像的序号，可以通过ix2id[index]获取对应图片的文件
        名
        '''
        index = index+self._start
        img = self.all_imgs[index]

        caption = self.captions[index]
        # 5句描述随机选一句
        rdn_index = np.random.choice(len(caption),1)[0]
        caption = caption[rdn_index]
        return img,t.LongTensor(caption),index

    def __len__(self):
        return self.len_

    def train(self,training=True):
        '''
        在训练集和验证集之间切换，training为True，getitem返回训练集的数
        据，
        否则返回验证集的数据
        前19万张为训练集，最后1万张为验证集
        '''
        self.training = training
        if self.training:
            self._start = 0
            self.len_ = len(self._data)-10000
        else:
            self._start = len(self.ix2id)-10000
            self.len_ = 10000
        return self
```

在 __getitem__ 中，dataset 返回一个样本的数据。在 dataloader 中，会将每个样本的数据拼接成一个 batch，可是由于描述的长短不一，无法拼接成一个 batch，这里需要自

（1）图片经过 ResNet 提取成 2048 维度的向量，然后利用全连接层转成 256 维的向量。可以认为从图像的语义空间转成了词向量的语义空间。

（2）描述经过 Embedding 层，每个词都变成了 256 维的向量。

（3）将第一步和第二步得到的词向量拼接在一起，送入 LSTM 中，计算每个词的输出。

（4）利用每个词的输出进行分类，预测下一个词（分类）。

```
class CaptionModel(nn.Module):
    def __init__(self,opt,word2ix,ix2word):
        super(CaptionModel, self).__init__()
        self.ix2word = ix2word
        self.word2ix = word2ix
        self.opt = opt
        self.fc = nn.Linear(2048,opt.rnn_hidden)

        self.rnn = nn.LSTM(opt.embedding_dim,opt.rnn_hidden,num_layers=opt
        .num_layers)
        self.classifier = nn.Linear(opt.rnn_hidden,len(word2ix))
        self.embedding = nn.Embedding(len(word2ix),opt.embedding_dim)
        if opt.share_embedding_weights:
            # rnn_hidden=embedding_dim的时候才可以
            self.embedding.weight

    def forward(self,img_feats,captions,lengths):
        embeddings = self.embedding(captions)
        # img_feats是2048维的向量,通过全连接层转为256维的向量,和词向量一样
        img_feats = self.fc(img_feats).unsqueeze(0)
        # 将img_feats看成第一个词的词向量,和其他词向量拼接在一起
        embeddings = t.cat([img_feats,embeddings],0)
        # PackedSequence
        packed_embeddings = pack_padded_sequence(embeddings,lengths)
        outputs,state = self.rnn(packed_embeddings)
        # lstm的输出作为特征用来分类预测下一个词的序号
        # 因为输入是PackedSequence,所以输出的output也是PackedSequence
        # PackedSequence的第一个元素是Variable,即outputs[0]
        # 第二个元素是batch_sizes,即batch中每个样本的长度
        pred = self.classifier(outputs[0])
```

```
    return pred,state
```

这里比较复杂的地方在于 PackedSequence 的使用，由于 LSTM 的输入是 PackedSequence，所以输出也是 PackedSequence，PackedSequence 是一个特殊的 tuple，既可以通过 `packedsequence.data`获得对应的 variable，也可以通过`packedsequence[0]`获得。如果想要获得对应的 tensor，则需要`packedsequence.data.data`，第一个 data 得到的是 variable，第二个 data 得到的才是 tensor。因为总共大约有 10000 词，所以最终变成了一个 10000 分类问题，采用交叉熵损失作为目标函数。

Image Caption 问题被我们转变成了一个分类问题，训练部分的代码如下。

```
def train(**kwargs):

    opt = Config()
    for k,v in kwargs.items():
        setattr(opt,k,v)

    # 数据
    vis = Visualizer(env = opt.env)
    dataloader = get_dataloader(opt)
    _data = dataloader.dataset._data
    word2ix,ix2word = _data['word2ix'],_data['ix2word']

    # 模型
    model = CaptionModel(opt,word2ix,ix2word)
    if opt.model_ckpt:
        model.load(opt.model_ckpt)
    optimizer = model.get_optimizer(opt.lr1,opt.lr2)
    criterion = t.nn.CrossEntropyLoss()
    if opt.use_gpu:
        model.cuda()
        criterion.cuda()

    # 统计
    loss_meter = meter.AverageValueMeter()

    for epoch in range(opt.epoch):
        loss_meter.reset()
```

```
    for ii,(imgs, (captions, lengths),indexes)  in tqdm.tqdm(enumerate
    (dataloader)):
        # 训练
        optimizer.zero_grad()
        input_captions = captions[:-1]
        if opt.use_gpu:
            imgs = imgs.cuda()
            captions = captions.cuda()
        imgs = Variable(imgs)
        captions = Variable(captions)
        input_captions = captions[:-1]
        target_captions = pack_padded_sequence(captions,lengths)[0]
        score,_ = model(imgs,input_captions,lengths)
        loss = criterion(score,target_captions)
        loss.backward()
        optimizer.step()
        loss_meter.add(loss.data[0])
```

训练的步骤和第 9 章中的 CharRNN 比较相似，都是利用前 n-1 个词作为输入，后 n-1 个词作为预测目标，把问题变成一个分类任务。要注意的还是 PackedSequence 的使用，除了 LSTM 等 RNN 层支持 PackedSequence 外，交叉熵损失函数也支持 PackedSequence，只需要给定对应的 lengths 即可。

完成训练后需要利用图片生成描述。我们可以采取第 9 章 CharRNN 的做法：

- 提取图像特征。通过全连接层得到 256 维的向量 v_0，输入到 LSTM 得到输出。
- 利用输出预测下一个词，得到最有可能的一个词（类别）w_1。
- 将上一步得到的词 w_1 经过 embedding 层得到词向量 v_1，将 v_1 输入到 LSTM 得到输出。
- 利用输出预测下一个词，得到最有可能的一个词 w_2。
- 将上一步得到的词 w_2 经过 embedding 层得到词向量 v_2，将 v_2 输入到 LSTM 得到输出。
- ……
- 重复上述步骤，直到遇到结束标识符，退出。

但这种类似贪心算法的搜索很容易陷入局部最优。对应的改进算法是 beam search，beam search 算法是一个动态规划算法，它每次搜索的时候，不是只记下最可能的一个

词，而是记住最可能的 k 个词，然后继续搜索下一个词，找到 k^2 个序列，保存概率最大的 k，就这样不断搜索直到最后得到最优结果。这种做法可以在一定程度上避免陷入局部最优。这部分代码使用 TensorFlow 中的开源代码，代码细节在此不再讲解，请读者自行阅读相应代码（`utils/beam_search.py`）。

测试代码如下：

```
def generate(**kwargs):
    opt = Config()
    for k,v in kwargs.items():
        setattr(opt,k,v)

    # 数据预处理
    data = t.load(opt.caption_data_path)
    word2ix,ix2word = data['word2ix'],data['ix2word']

    IMAGENET_MEAN =  [0.485, 0.456, 0.406]
    IMAGENET_STD =  [0.229, 0.224, 0.225]
    normalize =  tv.transforms.Normalize(mean=IMAGENET_MEAN,std=
    IMAGENET_STD)
    transforms = tv.transforms.Compose([
                tv.transforms.Scale(opt.scale_size),
                tv.transforms.CenterCrop(opt.img_size),
                tv.transforms.ToTensor(),
                normalize
        ])
    img = Image.open(opt.test_img)
    img = transforms(img).unsqueeze(0)

    # 用resnet50提取图片特征
    resnet50 = tv.models.resnet50(True).eval()
    del resnet50.fc
    resnet50.fc = lambda x:x
    if opt.use_gpu:
        resnet50.cuda()
        img = img.cuda()
    img_feats = resnet50(Variable(img,volatile=True))
```

```
# Caption模型
model = CaptionModel(opt,word2ix,ix2word)
model = model.load(opt.model_ckpt).eval()
if opt.use_gpu:
     model.cuda()

results = model.generate(img_feats.data[0])
print('\r\n'.join(results))
```

通过--caption-data-path=input.png指定输入的图片，通过--model_ckpt=model.pth指定预训练好的模型。

10.4 实验结果分析

训练命令：

```
python main.py train  --batch-size=256 \
                      --plot-every=500 \
                      --epoch=1000  \
                      --lr1=0.001
```

测试命令：

```
python main.py generate \
        --model-ckpt='gpuck/caption_0915_0602' \
        --test-img='bad4.jpeg'\
```

模型的效果图如图 10-5 所示，我们可以看出以下几个特点。

- 模型能够提取出图片的基本元素，例如人物性别、动作、场景等。
- 不论图中是否有人物，描述中都会出现人称（女人、男人）例如第二行左边第一副图，只有一匹白马，Caption 已经识别出了白色、马、绿油油、草地等特征，但是因为训练集中所有图片都有人，所以生成的描述“绿油油的草地上有一个穿着白色上衣的女人在骑马”。
- 对过于复杂的关系，神经网络难以描述，例如第一行最右边的一张图片，图中人物较多，模型不能很好地描述他们之间的关系。

总体而言，生成的描述质量虽然距离人类的水平还有不小的距离，但是从学术角度看效果还不错。对于和训练集类似的数据集，生成描述质量较高，但是对于和训练集

图 10-5　模型对图片生成的描述结果

数据差异较大的图片（例如没有人物的图片、风景图、动漫图等）描述质量比较差，而且偏差较多。

本次实验还有许多有待改进的地方。

- 在本次实验中，利用 ResNet50 提取模型特征，只训练 Generator 部分，事实上可以同时训练 ResNet50 和 Generator 两部分，但是这样的训练速度会慢许多。
- 可以采用预训练好的词向量初始化 Embedding，这样能够极大地提高收敛速度。

Image Caption 看似是一个复杂的任务，实际上远比我们想象的简单。本章重点讲解的是对数据的处理，讲解了文本数据常用的处理方式，希望能让读者在后续处理类似问题时有一些启迪和参考。

11 展望与未来

11.1 PyTorch 的局限与发展

PyTorch 采用了动态图为用户带来了极大的灵活性，但有得必有失，动态图也不是十全十美的。许多针对动态图的批评者认为动态图不能像静态计算图一样进行性能优化。这一点在理论上是成立的，但是在实际使用中并非如此。事实上，因为 PyTorch 极少的封装使 PyTorch 能够直接调用底层的代码，尽可能地减少调用开销，即使不进行什么优化也能实现优异的性能。在实际使用中，PyTorch 的速度往往比 TensorFlow 快。在笔者眼中，PyTorch 的缺陷主要有以下三方面。

第一，分布式并行支持有限。PyTorch 支持程序在多个 GPU 并行计算，也支持多个 GPU 多台机器的分布式运行，它可以把一个 batch 的数据拆分成多个子集，然后在多个 GPU 上并行加速，但有时这种并行还不够。假设有如下一个场景：你有一个很大的模型，大到一个 GPU 没法放下模型（哪怕 batch size 为 1），你想让模型的一部分运行在 GPU1 上，另一部分模型运行在 GPU2 上，然后将这两部分模型的运行结果聚合在一起。在这种情况下，你可以实现：

```
class MyModel(nn.Module):
    def __init__(self):
        self.large_submodule1 = ...
        self.large_submodule2 = ...
        self.large_submodule1.cuda(0)
        self.large_submodule1.cuda(1)
```

```
        self.model3 = ....

    def forward(self, x):
        y = self.large_submodule1(x)
        z = self.large_submodule2(x.cuda(1))
        result = self.model3(y,z.cuda(0))
        return result
```

你可能希望能够同时运行下面两句代码，因为它们是独立的：

```
y = self.large_submodule1(x)
z = self.large_submodule2(x.cuda(1))
```

然而这在 PyTorch 中无法实现。PyTorch 中通过 Python 语句直接执行 PyTorch 代码，而在 Python 中必须先执行第一句才能继续执行第二句。在这种场景下，两个 GPU，始终有一个处于空闲状态。目前针对这种需求 PyTorch 还没有太好的解决方法，一个也许可行的方案是利用多线程异步执行。不过幸运的是 PyTorch 中自带的 DataParallel 足以应对绝大多数的并行需求，而我们在实际做实验时一个模型大到需要两个 GPU 才能跑得动的情况，也很少遇到。

第二，难以部署到生产环境。生产环境的可移植需求如图 11-1 所示，包括移动设备、嵌入式设备、云端设备等。PyTorch 以其强大的灵活性著称，这种特性极大地吸引了广大的科研人员。然而 PyTorch 过于灵活，使得它难以在包括云端、移动端、嵌入式等环境中部署。尽管有人成功地将其部署到手机，但 PyTorch 的设计从一开始就不是针对部署而生的，将其部署到生产环境会花费一定的时间，而且效果还不一定能符合需求。虽然现在深度学习以研究应用居多，但是随着研究的不断深入，理论的不断完善，如何将深度学习应用部署到生产环境中越来越重要。

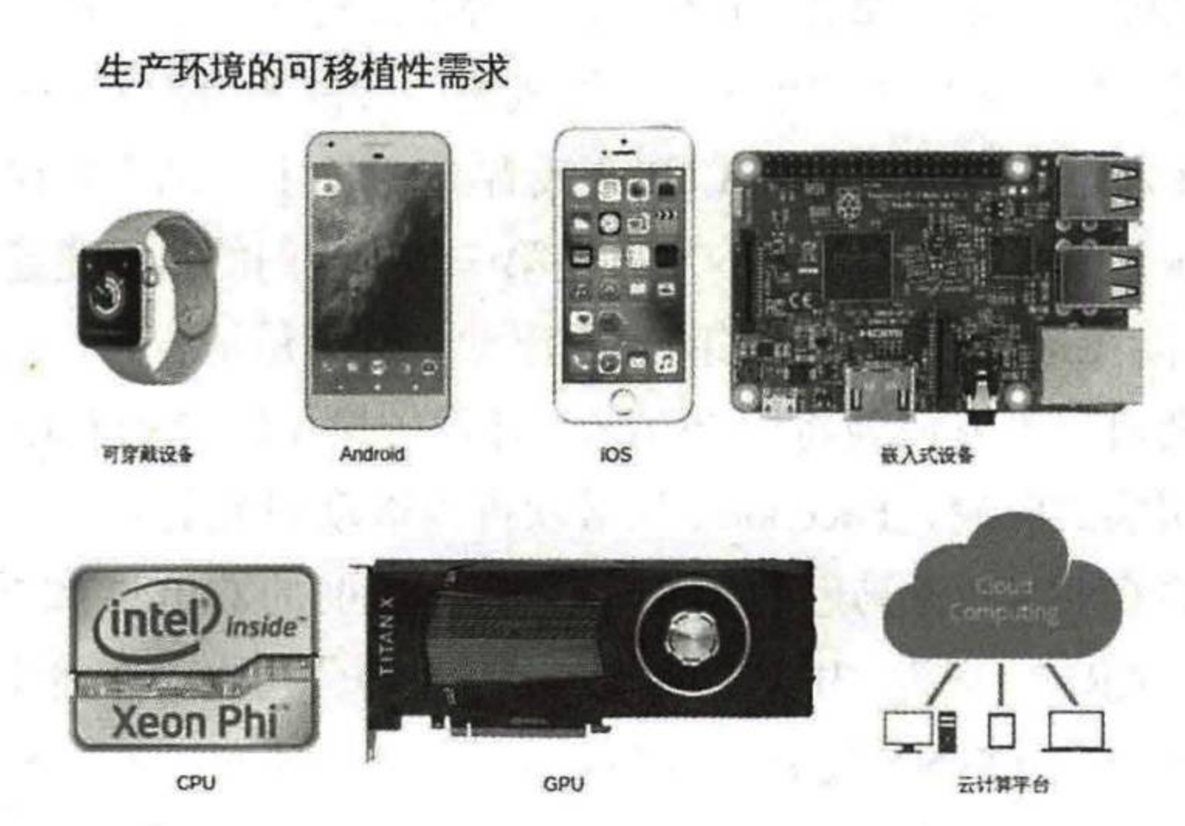


图 11-1　生产部署设备

虽然 PyTorch 在生产部署方面有缺陷，但它却有两个很好的帮手：Caffe2 和 ONNX。

和 PyTorch 一样，Caffe2 框架也是由 FAIR 开发的，它于 2017 年 4 月 19 日发布，可以通过一台机器上的多个 GPU 或具有一个及多个 GPU 的多台机器进行分布式训练；也可以在 iOS 系统、Android 系统和树莓派（Raspberry Pi）上训练和部署模型。Caffe2 框架已经应用在 Facebook 平台上，NVIDIA、Qualcomm（高通）、Intel（英特尔）、Amazon（亚马逊）和 Microsoft（微软）等公司的云平台都已支持 Caffe2。Caffe2 已用于移动和大规模部署。

PyTorch 适合进行研究、实验和尝试不同的神经网络，而 Caffe2 更偏向于工业应用，并且重点关注在移动端上的表现。Caffe2 的作者贾扬清表示 Caffe2 和 PyTorch 团队紧密合作，他们把 Caffe2 视作一种生产力的选择，而把 PyTorch 视作研究型的选择。在构建 AI 模块时，他们也持有一种“非框架”的理念，例如 Gloo、NNPACK 和 FAISS 等可以被用于任何深度学习框架，他们正在计划让 Caffe2、Torch 和 PyTorch 共享后端，这几个框架已经共享 Gloo 用于分布式训练，THCTensor、THNN 和其他 C/C++ 库也将共享。

在 Facebook 内部，研究和产品应用之间有着很明显的区隔，这家公司一直有两个机器学习团队：FAIR 和 AML（应用机器学习），FAIR 专注于前沿性研究，而 AML 则关注如何将人工智能产品化。深度学习框架的选择是造成这种区别的关键。FAIR 习惯于使用 PyTorch，这种深度学习框架可以不考虑资源限制，在研究中取得漂亮的结果。然而在现实世界中，大部分人都受限于智能手机和计算机的计算能力。当 AML 希望开发可以部署、可以规模化的产品时，通常会选择 Caffe2。

为了更好地将研究的成果部署到生产环境中，打通不同框架之间沟通的桥梁，Facebook 联合微软设计了一种开源兼容格式，任何支持该格式的框架都可以共享模型。2017 年 9 月 7 日，Facebook 和微软在各自的博客中发布了 ONNX（Open Neural Network Exchange，开放神经网络转换）。如图 11-2 所示，ONNX 构建起了不同深度学习框架间通信的桥梁，它能把一种框架训练的模型，转换成另一种框架所需的格式。机器学习开发者可以将 PyTorch 训练的模型转换到 Caffe2 或者 CNTK 上，减少从研究到产品化所耗费的时间。Facebook 在博客中说，ONNX 只是第一步，他们的目标是建立一个开放的生态，让 AI 开发者能在最先进的工具之间轻松流动，选择最适合自己的框架。Facebook 和微软的合作帮助研究者方便地将用 PyTorch 开发的模型转换为 Caffe2 模型。通过降低两种框架之间切换的障碍，Facebook 和微软可以推动研究的普及，加速整个商业化进程。目前微软的 CNTK、亚马逊的 MXNet、Facebook 的 Caffe2 和 PyTorch 都宣布支持 ONNX，AMD、ARM、华为、IBM、英特尔、高通等公司也宣布将支持 ONNX，但距离“开放生态”的愿景，ONNX 还有一段路要走，因为并不是所有公司都在用 PyTorch、Caffe2 和 CNTK。也正因为机器学习框架这么多，功能大同小异，才让 ONNX 这样的

工具有用武之地。

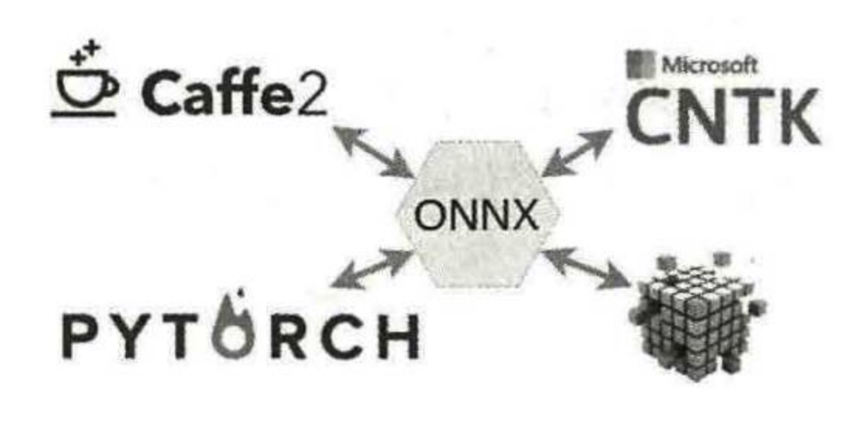


图 11-2 ONNX 构建起了不同深度学习框架间通信的桥梁

笔者更认同不同的框架做不同的事，“一个框架通吃”的思想不仅可能会影响性能，更会无形中加大框架本身的复杂度，即使是最优秀的工程师，维护或者掌握一个复杂的框架，还是会捉衿见肘。如今号称支持全平台的框架，在生产使用中真正实现全平台支持还是相当困难的，需要增加许多额外的工作。研究和生产本就是两个相对独立的环节，指望研究的代码直接用于生产环境有点不切实际。

笔者建议研究的时候，使用最灵活、最好用的框架，而在部署的时候，使用相对应的移动端嵌入式框架或者云端分布式的框架。毕竟学习 TensorFlow 的难度不亚于学习两个简单易用的框架，而且即使是已经掌握了 TensorFlow 的用户，当想要在移动端部署深度学习应用的时候，也需要折腾好长时间，而且效果还不如专门针对移动端开发的第三方框架。

第三，某些场景下性能缺失。PyTorch 大量的代码使用 Python 开发，导致在小负载的情况下性能比较低下。如果 Python 调用的底层是 C/C++ 函数，调用开销可能是微秒级别的时间，而如果一个 Python 函数调用的代码本身就是 Python 写的，那么调用开销可能需要几毫秒。PyTorch 之所以速度快，原因在于它较少的封装，常用的 Python 接口基本上都是直接调用 C/C++/CUDA/CuDNN 接口，调用开销较小。但是有些操作却是由 Python 处理逻辑，如果多次反复调用，这些毫秒级别的开销累加起来还是可能极大地降低运行速度。针对这个问题，除了把更多常用的操作迁移到 C/C++ 之外，PyTorch 还决定在下一个版本中加入 JIT。

JIT（Just-in-time compilation，即时编译），是动态编译的一种形式，是一种提高程序运行效率的方法。通常，程序有两种运行方式：静态编译与动态解释。静态编译的程序在执行前全部被翻译为机器码，而解释执行的则是一句一句边运行边翻译。C/C++ 等属于静态编译，而 Python 属于动态解释。即时编译器则混合了这二者，一句一句编译源代码，但是会将翻译过的代码缓存起来以降低性能损耗。相对于静态编译代码，即时编译的代码可以处理延迟绑定并增强安全性。通过 JIT 可以明显改善解释器的性能。

JIT 能为 PyTorch 带来可编译的特性，能在某些场景下增加 PyTorch 的性能表现，但这也在无形之中增加了复杂度。对于 PyTorch 的这些改进，笔者本人并不是完全赞成，诚然这些特性会使得 PyTorch 的性能表现得更优异、更通用，但有时这些修改带来的性能提升比较有限，却增加了系统复杂度。在笔者眼中，PyTorch 不应当追求小优化，而应专注于整体的“大胜”。对此，开发者们也在热烈的讨论。

11.2 使用建议

结束本书的学习，意味着读者可以完成绝大多数基于 PyTorch 的深度学习任务，但并不意味着学习的结束。除了第 6 章提起的指南之外，以下几个内容对读者在未来的实践也会很有帮助。

- GitHub[①]（如图 11-3 所示）：GitHub 作为当今最大的开源仓库，里面有数不胜数的开源项目。PyTorch 现在在 GitHub 上十分流行，绝大多数研究方向都有不错的开源实现。当读者做实验的时候，不妨先参考 GitHub 上其他人的实现。这样做可以有效地避免思维误区，同时从优秀的代码中学习经验，提升自己的代码质量。不建议拿别人已经写好的代码直接运行，应该参考着别人的代码从头写，尤其是在学习的过程中，只有自己写的代码才能算真正掌握了。

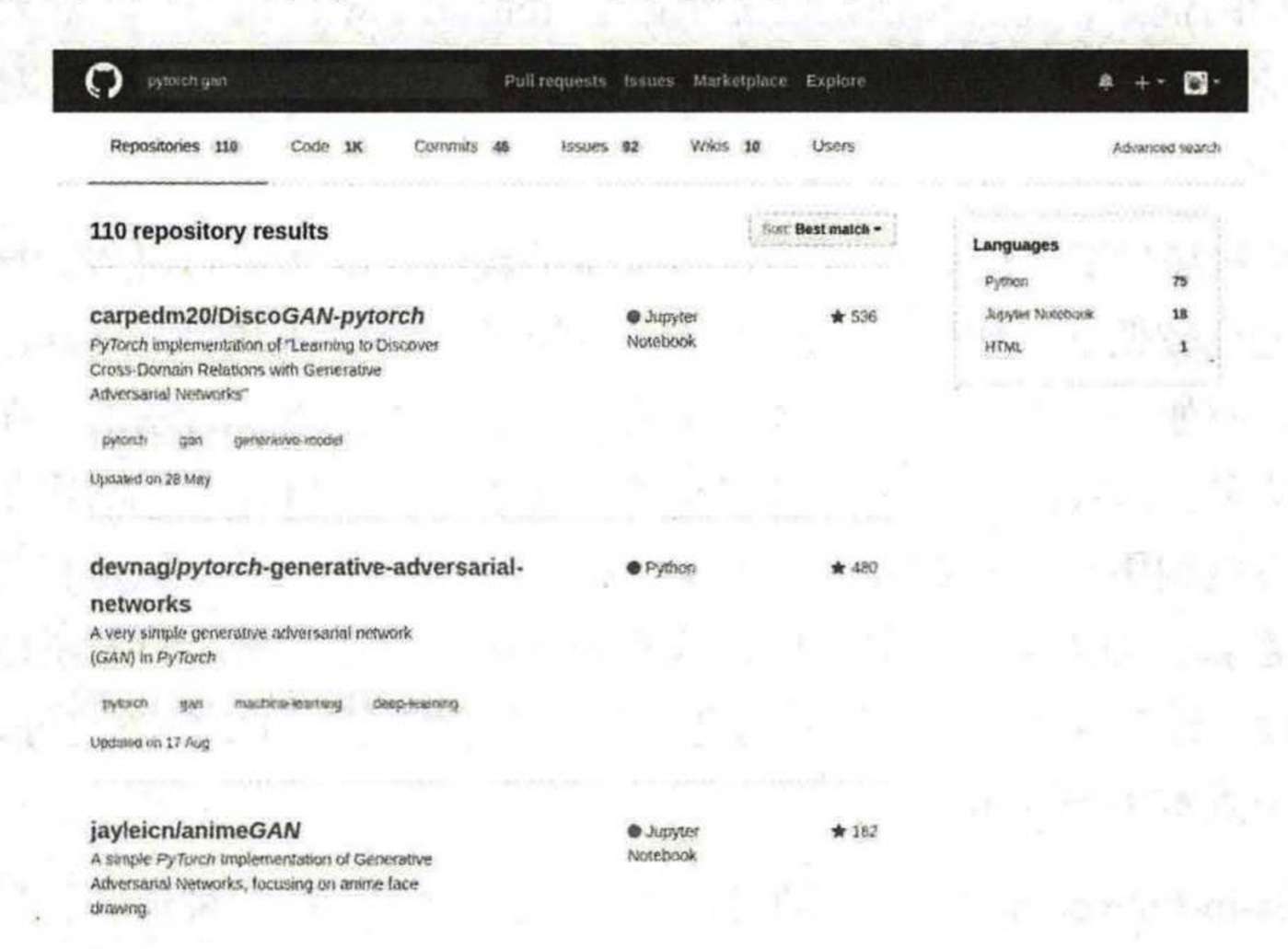


图 11-3 使用 GitHub 搜索优秀的开源代码

在 GitHub 上诸多的示例代码中，质量最高的是官方的示例仓库（pytorch/example），这些样例代码或者由作者亲自编写，或者是经过作者的审核才被收录到官

[①] https://github.com

方仓库中，往往实现很简洁，更符合 PyTorch 的设计思想。

除了利用 GitHub 搜索、参考别人的代码外，还可以到 PyTorch 的官方仓库，追踪 PyTorch 的开发进展，阅读开发者们的讨论内容，理解他们在开发 PyTorch 过程中的取舍。学有余力者还可以试着为 PyTorch 做一些简单的代码贡献。在 issue 中有许多人提问题，其中很多 BUG 或使用不当踩的坑，不论是哪一种，读者都能从中积累许多经验。

- 文档[1]（如图 11-4 所示）。PyTorch 的文档精炼简洁，同时又条理清晰，强烈建议读者抽空通读所有文档，大概花费 3~4 个小时。通读文档可以了解 PyTorch 的所有接口，等到使用时能够自然而然想起来，从而避免陷入重复造轮子或者使用不当等情况。PyTorch 的文档由 PyTorch 的作者编写，严谨而准确，在 IPython 下还能够配合 IPython 的自省机制查看，例如`nn.Conv2d?`就能查看`nn.Conv2d`的文档。PyTorch 的中文社区十分活跃，目前已经有人将 PyTorch 的文档翻译成中文（http://pytorch-cn.readthedocs.io），这份文档虽然稍微落后于最新版的 PyTorch 官方文档，但并不妨碍正常使用，不过建议英语水平不错的读者直接阅读英文文档（如图 11-4 所示）。

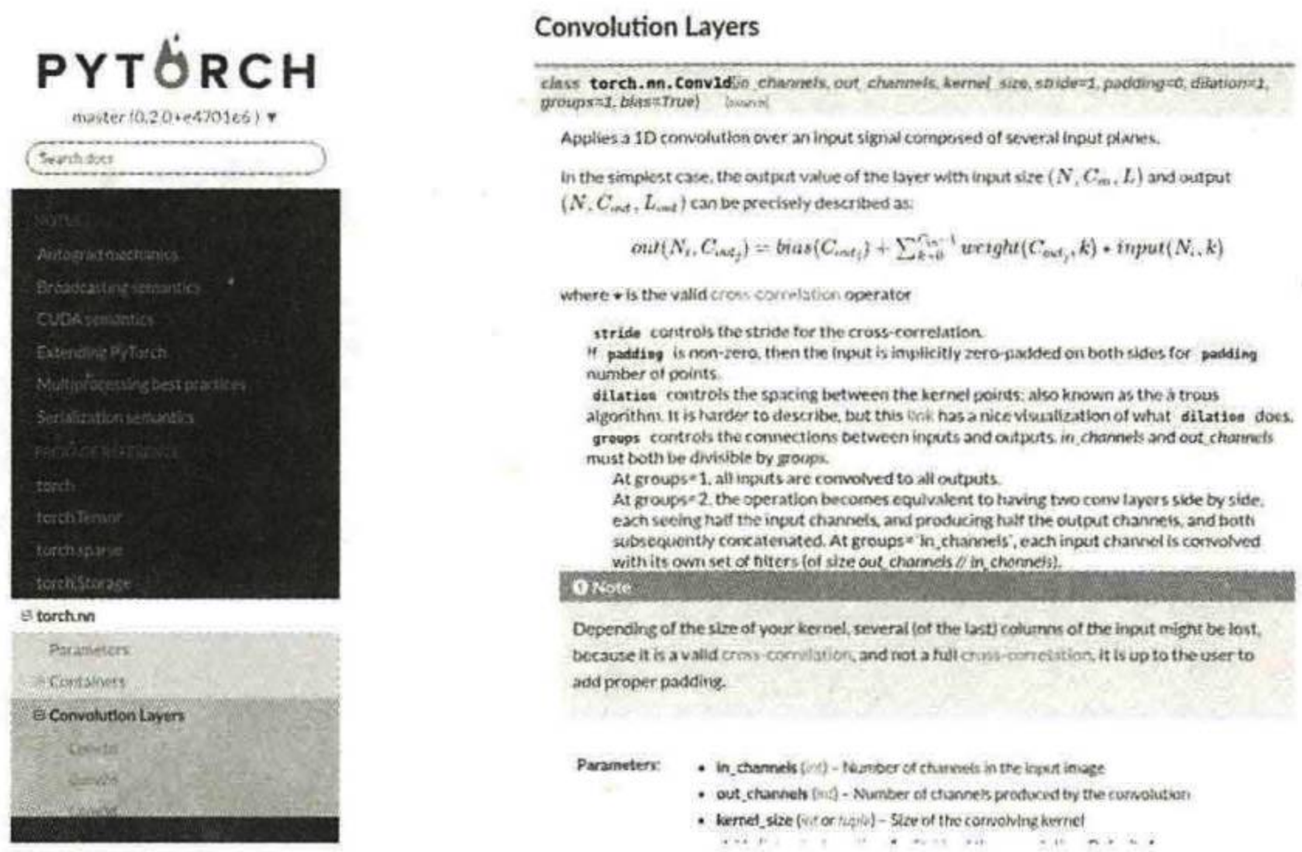


图 11-4 PyTorch 的文档

- 论坛[2]。PyTorch 专门建立论坛供读者交流，这对 PyTorch 的社区构建起到了非常好的效果，如图 11-5 所示。不少初学者就是通过在论坛求教解决了自己遇到的问题。论坛由 PyTorch 的作者亲自维护，用户也很积极热心，绝大多数帖子都能在数小时之内得到回复。许多经典的问题（比如 PyTorch 中如何 finetune，如何从预训练的模型中提取特征）被积极的讨论，也为初学者在后续的学习中起到非常好

[1] http://pytorch.org/docs

[2] https://discuss.pytorch.org

的指导作用。如果读者在学习 PyTorch 的过程中遇到了问题，或者有疑惑，不妨先到论坛上搜索，有很大的概率能够找到解决方案。如果没找到满意的答案，可以直接发帖提问，论坛中的用户都很热心，想必能帮到你。

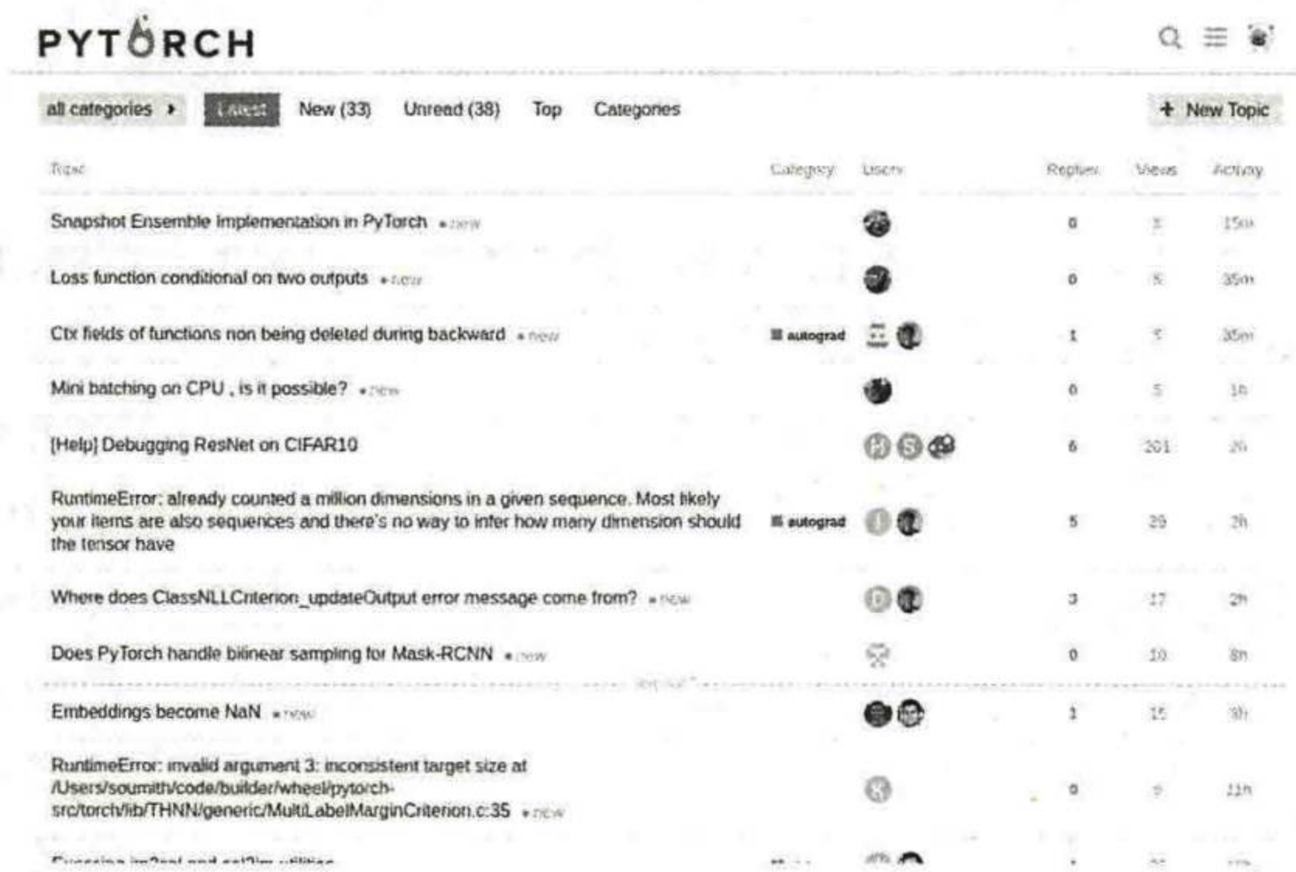


图 11-5　PyTorch 论坛

Happy Coding!

好书分享

《深度学习轻松学：核心算法与视觉实践》

作 者：冯超

出版时间：2017-08

ISBN：9787121317132

专注深度学习的思想本质，结合大量案例夯实技术基础！

知其然，更知其所以然，成为人工智能专家！

本书特色

√理论 + 实践 = 深度学习前沿技术

√夯实入门深度学习必备数学知识（线性代数基础、对称矩阵的性质、概率论、信息论基础、KL 散度、凸函数等）

√梳理深层模型（全连接层、卷积层）的基本知识

√经典深度学习框架入门

√深层模型的数值问题

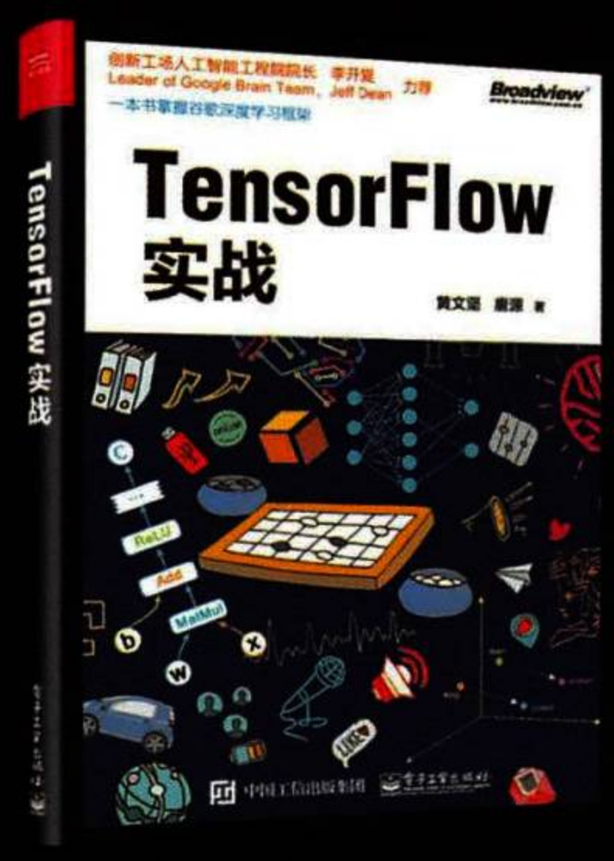
√经典模型能完成哪些任务

√模型的优化算法（方法）

√视觉领域前沿应用（图像分割、生成模型等）

好书分享

《深度学习轻松学：
核心算法与视觉实践》



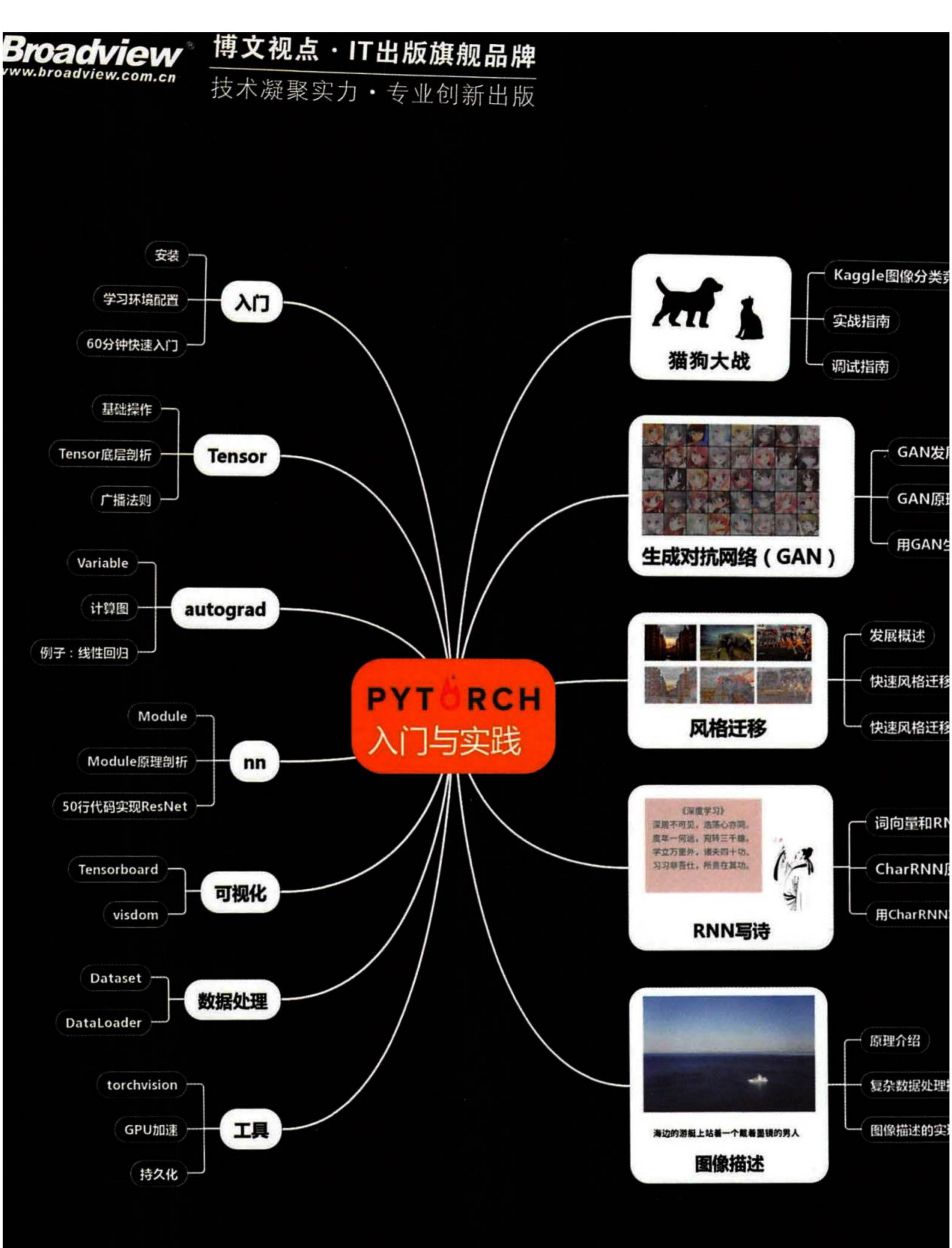
《TensorFlow实战》

欢迎投稿
邮箱：zhenglj@phei.com.cn
微信号：Alinamercy

Broadview®
www.broadview.com.cn

博文视点 · IT出版旗舰品牌

技术凝聚实力 · 专业创新出版



上架建议：计算机

博文视点Broadview

@博文视点Broadview

责任编辑：郑柳洁
封面设计：吴海燕

定价：6