

程序员的自我修炼手册

目录

第 1 章 搭建可塑计算机视觉工具集.....	9
1.1 在机器上搭建一个深度学习环境	9
1.1.1 在 Windows10 上配置软件工具.....	9
1.1.2 在 Ubuntu18.04 上配置软件工具.....	13
1.1.3 设置清华镜像.....	15
1.2 安装深度学习框架.....	16
1.3 桥接 Ubuntu 与 Windows.....	19
1.4 Ubuntu 多用户共享使用深度学习环境	19
1.5 本章小结	21
第 2 章 计算机视觉之利器：vscode.....	21
2.1 本书的注意事项	21
2.2 安装 vscode 必备插件	21
2.3 MPE 使用的必要性.....	23
2.4 Markdown 使用手册	23
2.4.1 Markdown 基本要素	23
2.4.2 常用标记	24
2.4.3 次常用标记	27
2.4.4 不常用标记	28
2.5 公式使用参考	29
2.5.1 如何插入公式.....	29
2.5.2 如何输入公式的上下标.....	30

2.5.3 如何输入括号和分隔符	30
2.5.4 如何输入分数.....	31
2.5.5 如何输入开方.....	31
2.5.6 如何输入省略号.....	31
2.5.7 如何输入矢量.....	32
2.5.8 如何输入积分.....	32
2.5.9 如何输入极限运算.....	33
2.5.10 如何输入累加、累乘运算.....	33
2.5.11 如何输入希腊字母.....	33
2.5.12 大括号和行标的使用	34
2.5.13 字体转换.....	35
2.6 将 Markdown 转换为 docx 文档	35
2.6.1 依据给定的模板输出 word 文档.....	37
2.6.2 设置多文件生成 word 的模板	37
2.6.3 设置自动目录.....	37
2.6.4 自动导出为 word	38
2.7 导入外部文件	38
2.8 其他有趣的功能	39
2.8.1 导入目录	39
2.8.2 使用表情符号 与 Front Icon.....	39
2.8.3 画图.....	39
2.8.4 任务列表	40
2.9 本章小结	40
第 3 章 Python 的基础入门	40
3.1 编写和运行代码	41
3.2 Python 基础知识	44

3.2.1 标识符：我们的描述对象的名称	44
3.2.2 赋值语句	45
3.2.3 输入与输出	45
3.3 数据类型	45
3.3.1 Number 数字	46
3.3.2 String 字符串	46
3.3.3 List 列表	46
3.3.4 Tuple 元组	47
3.3.5 Set 集合	47
3.3.6 Dictionary 字典	47
3.4 Python 数据类型转换	47
3.5 运算符	48
3.5.1 算术运算符 & 赋值算术运算符	48
3.5.2 位运算符	49
3.5.3 比较运算符 & 逻辑运算符	50
3.5.4 成员运算符 & 身份运算符	50
3.6 Python 运算符优先级	51
3.7 从容器角度看待 list, tuple, dict, set 等结构	51
3.7.1 序列的基本操作：索引与切片	52
3.7.2 序列的拼接	52
3.7.3 序列的其他方法	53
3.7.4 序列之列表	53
3.7.5 序列之元组	56
3.7.6 容器之字典（dict）	57
3.7.7 集合	60
3.8 本章小结	63

第 4 章 Python 的简易进阶教程.....	63
4.1 再谈“变量”.....	63
4.1.1 交换赋值.....	64
4.1.2 可变对象的赋值	64
4.1.3 增量赋值.....	65
4.1.4 符号 * 的妙用	66
4.2 流程控制.....	66
4.2.1 真值测试.....	66
4.2.2 if 条件语句	67
4.2.3 循环结构.....	67
4.3 函数.....	69
4.4 类与模块.....	71
4.5 本章小结	72
第 5 章 定制 AI 的专属数据 X.....	72
5.1 数据集 X 的制作的准备工作	72
5.1.1 好用的 Bunch.....	72
5.1.2 MNIST 的处理.....	74
5.1.3 Cifar 的处理	75
5.1.4 如何使用 MNIST 与 Cifar 类	77
5.2 数据集 X 的制作.....	78
5.2.1 数据集的可视化与简介	79
5.2.2 数据集 X 的封装	81
5.2.3 Bunch 转换为 HDF5 文件：高效存储数据集 X	82
5.3 X.h5 的使用说明	83
5.4 本章小结	87

第 6 章 CASIA 脱机和在线手写汉字库.....	87
6.1 CASIA 手写汉字简介.....	87
6.2 手写单字的特征的解码.....	88
6.3 手写单字的特征的序列化与反序列化.....	91
6.4 手写单字的图片解读.....	95
6.4.1 离线手写单字的图片解析.....	95
6.4.2 在线手写单字的图片解析.....	97
6.5 本章小结.....	98
第 7 章 COCO API 的使用.....	98
7.1 COCO API 的配置与简介.....	98
7.1.1 Windows 的配置.....	99
7.1.2 Linux 下的配置.....	100
7.2 改写 COCO API 的初衷.....	100
7.2.1 Why? API 改写的目的.....	100
7.2.2 What? API 可以做什么.....	100
7.2.3 How? API 如何设计.....	100
7.3 ImageZ 的设计和使用.....	101
7.4 AnnZ 的设计和使用.....	103
7.5 COCOZ 的设计和使用.....	105
7.5.1 展示 COCO 的类别与超类	107
7.5.2 通过给定条件获取图片.....	108
7.5.3 将图片的 anns 信息标注在图片上.....	108
7.5.4 关键点检测.....	109
7.5.5 看图说话.....	111
7.6 让 API 更加通用.....	112

7.7 本章小结	115
第 8 章 计算机视觉的项目管理工具 Git.....	115
8.1 Git 基础简介.....	116
8.1.1 配置用户信息	116
8.1.2 创建并操作一个本地仓库.....	116
8.1.3 远程仓库.....	122
8.1.4 标签.....	123
8.1.5 分支.....	124
8.1.6 vscode 与 Git 集成.....	124
8.2 使用 Git 管理项目	125
8.2.1 对 Microsoft Office 进行版本控制.....	125
8.2.2 TortoiseGit 的使用.....	127
8.3 本章小结	133
第 9 章 构建属于自己的计算机视觉开源项目	133
9.1 创建一个项目	133
9.2 修改 README.....	135
9.3 编写贡献者指南	136
9.3.1 准备工作 1：创建 Issue 与 PR 模板	136
9.3.2 准备工作 2：编写行为准则.....	137
9.3.3 贡献指南编写细则	138
9.4 使用 git submodule 管理子项目.....	139
9.5 克隆含有子模块的项目	140
9.6 使用子模块	141
9.7 本章小结	141
第 10 章 从零开始设计计算机视觉软件.....	141

10.1 创建一个项目	141
10.2 项目的准备工作	142
10.3 开发一个小工具：bbox	144
10.3.1 创建数学中的“向量”	144
10.3.2 编写 box.py 代码	146
10.4 Python 中的数学：符号计算	147
10.4.1 使用 SymPy 表示数	147
10.4.2 使用 SymPy 提升您的数学计算能力	147
10.4.3 使用 SymPy 求值	149
10.4.4 使用 SymPy 做向量运算	150
10.5 案例：实现图像局部 resize 保持高宽比不变	152
10.6 本章小结	155
第 11 章 Kaggle 实战：猫狗分类 (Gluon 版)	156
11.1 数据处理	156
11.2 Gluon 实现模型的训练和预测	160
11.2.1 建立并训练模型	162
11.2.2 模型测试	165
11.3 可视化中间层的输出	166
11.4 本章小结	168
第 12 章 利用 GluonCV 学习 Faster RCNN	168
12.1 初识 RPN	168
12.1.1 锚点	169
12.1.2 感受野	171
12.1.3 锚点的计算	171
12.1.4 全卷积 (FCN)：将锚点映射回原图	174

12.2 平移不变性的锚点	178
12.2.1 边界框回归	179
12.2.2 裁剪预测边界框超出原图边界的边界	180
12.2.3 移除小于 min_size 的边界框	182
12.3 NMS (Non-maximum suppression)	182
12.3.1 重构代码	184
12.3.2 标注边界框	187
12.4 RoIHead	188
12.5 本章小结	189
第 13 章 自制一个集数据与 API 于一身的项目	189
13.1 构建项目的骨架	189
13.1.1 datasome 的创建细节	189
13.1.2 loader 的创建细节	190
13.2 创建一个数据与 API 的统一体	193
13.2.1 datasetsome 项目初设	193
13.2.2 在 datasetsome 中编写 loader 使用案例	195
13.3 编写 datasetsome 的使用案例	198
13.4 本章小结	204
第 14 章 TensorFlow 基础教程	204
14.1 使用 tf.keras.Sequential 搭建模型	204
14.2 FASHION-MNIST 可视化实验结果	206
14.3 PyTorch、TensorFlow、MXNet 的数据操作	209
14.4 PyTorch、TensorFlow、MXNet 自动微分	210
14.5 TensorFlow 实现手写汉字分类的实例	211
14.5.1 loader 的制作原理	212
14.5.2 打包多个 zip 文件	213

14.5.3 解析 features.h5	215
14.5.4 使用 TensorFlow 训练 MPF 分类器.....	216
14.6 本章小结.....	221

第一篇 基础篇

第 1 章 搭建可塑计算机视觉工具集

对于计算机视觉，深度学习是一个始终绕不开的话题，本文将告诉你如何从零开始搭建一个友好的可塑性的计算机视觉环境，当然它也支持深度学习。

为什么要从零开始搭建环境呢？因为，别人建立的环境也许并不适合你，而你想要改变的环境也许会花费很多时间还不一定有用。为了解决这个烦恼，本文考虑搭建一个具有可以定制性并可以灵活改变环境的计算机视觉工具集。

因为深度学习在计算机视觉中占据十分重要的地位，所以本文将以深度学习为基础进行工具集组装。本章会分别介绍 Windows10 与 Ubuntu18.04 上是如何搭建环境的。如果不深入了解 Ubuntu18.04 的环境搭建可以直接跳过，不会影响后面章节的阅读。

1.1 在机器上搭建一个深度学习环境

本文考虑分别在 Windows10 与 Ubuntu18.04 这两个系统上搭建深度学习环境。下面提到的软件包的具体功能先不说明，之后的章节再一一阐述其功能。

1.1.1 在 Windows10 上配置软件工具

首先，需要下载一些必备软件，相关下载链接我已放入赠送资料里，读者可自行下载：

- 下载 Anaconda，选择 Anaconda 2019.10 for Windows Installer-Python 3.7 版本。
- 进入 CUDA 下载页面，依次选择 Windows, x86_64, 10, exe (local) 进行下载。具体是操作界面，如图 1.1 所示。
- 下载深度学习加速库 cudnn。依次选择选择 Download cuDNN v7.6.4 (September 27,

2019), for CUDA 10.0, cuDNN Library for Windows 10 进行下载。

- 下载 vscode, 依次选择 Windows, System Installer 64 bit。
- 选择 64-bit Git for Windows Setup 版本下载 Git。

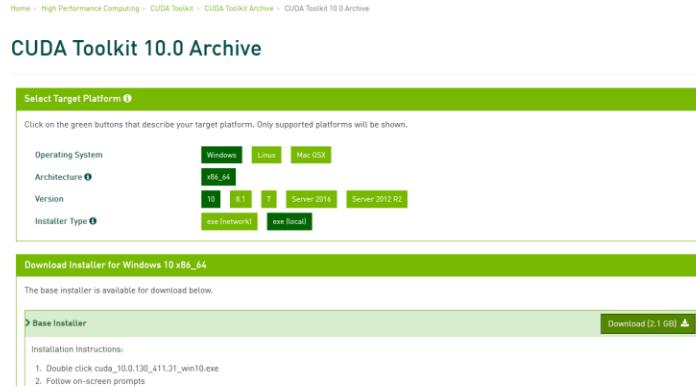


图 1.1 CUDA 下载的选择界面

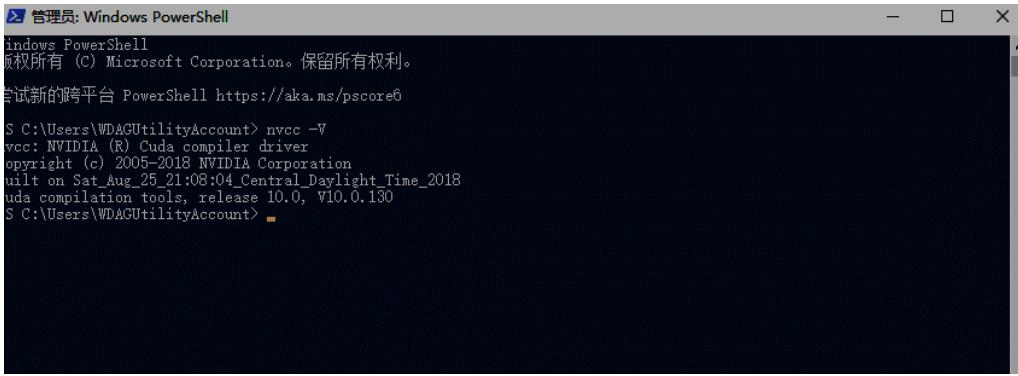
这些软件下载好之后，按照提示进行安装即可。不过，安装过程中可能出现一些选项需要按照本文接下来展示的图示进行操作。

(1) 安装 CUDA 时，需要选择安装选项为自定义（高级），如图 1.2 所示。



图 1.2 安装 CUDA 时需要选择 自定义

(2) 安装 CuDNN 时，只需要解压 cudnn-10.0-windows10-x64-v7.6.4.38.zip 到 C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v10.0 目录之下。接着，设置 Path 的环境变量：C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v10.0\lib\x64。安装完毕之后，打开 Power Shell 输入 nvcc -V 验证是否与图 1.3 显示一致。如果一致，恭喜安装完成！



```
管理员: Windows PowerShell
Windows PowerShell
版权所有 (C) Microsoft Corporation。保留所有权利。
尝试新的跨平台 PowerShell https://aka.ms/pscore6
S C:\Users\WDAGUtilityAccount> nvcc -V
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2018 NVIDIA Corporation
built on Sat_Aug_25_21:08:04_Central_Daylight_Time_2018
cuda compilation tools, release 10.0, V10.0.130
S C:\Users\WDAGUtilityAccount>
```

图 1.3 验证 nvcc 是否正确安装

(3) 安装 Anaconda 需要注意按照图 1.4 选择将 Anaconda 添加到环境变量。安装完成之后，找到开始菜单中的 Anaconda Navigator 图标，如图 1.5 所示便可以使用 Anaconda。

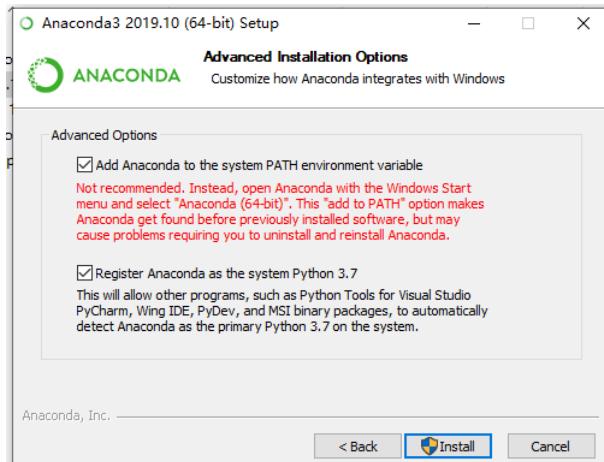


图 1.4 Anaconda 安装

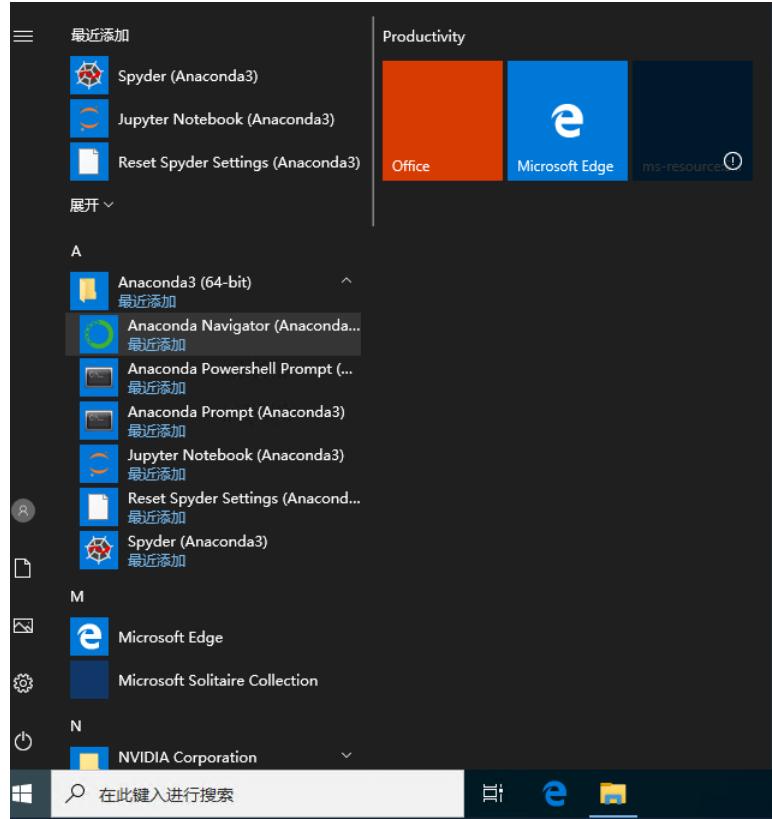


图 1.5 找到 Anaconda 启动的图标

(4) 为了将 Git 与 vscode 紧密结合，需要先安装 vscode，再安装 Git。为了方便 vscode 更好的管理您的文件，安装 vscode 时需要按照图 1.6 进行选择。安装 Git 需要按照图 1.7 进行选择将 vscode 作为 Git 的默认编辑器。

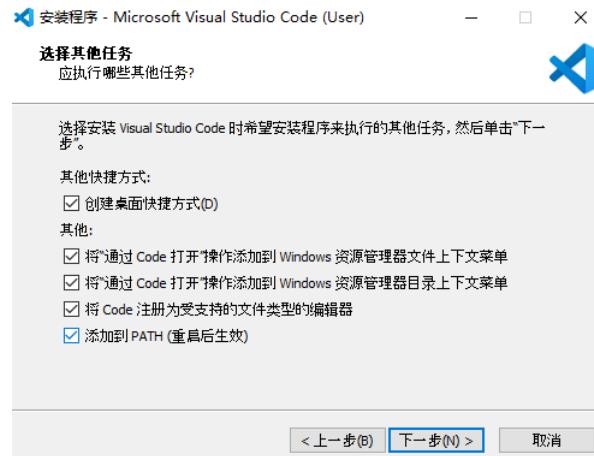


图 1.6 选择方便 vscode 管理的选项

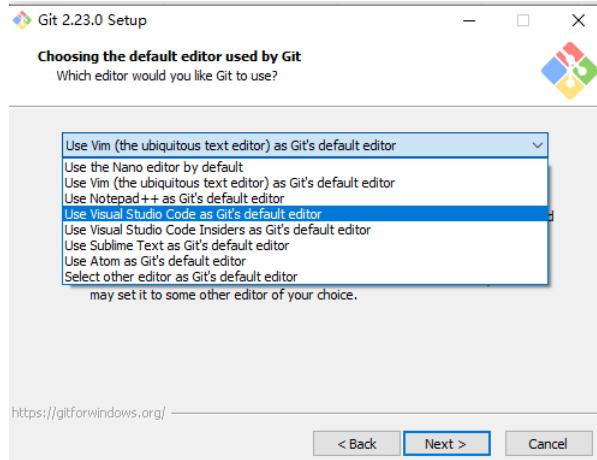


图 1.7 将 vscode 作为 Git 的默认编辑器

这样 vscode 与 Git 紧密结合在一起了。

1.1.2 在 Ubuntu18.04 上配置软件工具

在 Ubuntu 系统上搭建深度学习系统是很多人的噩梦，为了大家少走弯路，赶走噩梦，本文接下来介绍如何从安装 Ubuntu 开始搭建 Ubuntu 深度学习环境。和 Windows10 一样，我们同样需要先下载一些必备软件，读者可在赠送资料找到下载链接：

- 下载 Anaconda，选择 Anaconda 2019.10 for Linux Installer-Python 3.7 版本。
- 下载 Ubuntu 镜像，选择 Ubuntu 18.04.3 LTS 版本进行下载。
- 下载 CUDA，依次选择 Linux, x86_64, 18.04, runfile (local)，之后下载 Base Installer 与 Patch 1。
- 下载 rufs，用于制作 Ubuntu 启动盘
- 下载 cudnn：深度学习加速库。依次选择选择 Download cuDNN v7.6.4 (September 27, 2019), for CUDA 10.0, cuDNN Runtime Library for Ubuntu18.04 (Deb) 进行下载。

下载好软件之后，需要使用 U 盘制作一个 Ubuntu 启动盘。我们使用 rufs 制 Ubuntu 启动盘。

(1) 双击 rufs 软件包，弹出的界面，设置如图 1.8 所示。



图 1.8 rufs 软件界面

制作完毕关闭 rufs 软件，并弹出 U 盘。

(2) 关闭待安装的机器，之后上插入 U 盘，再启动机器，选择 Install Ubuntu，进入安装界面。

(3) 选择系统语言为中文

(5) 不断点击继续，直到安装类型的界面，选择清除整个磁盘并安装 Ubuntu（如果想要安装双系统，可选择安装 Ubuntu，与其他系统共存）。

(6) 接着按照界面的提示进行操作即可。直到提示您重启电脑时，您点击确认，之后等到屏幕关闭拔掉 U 盘，让机器自动重启。如此，您便完成了 Ubuntu 的安装。

电脑重启之后需要配置网络连接，配置好之后，我们需要做一些准备工作。

(1) 智能升级。安装新软件包并删除废弃的软件包：

```
$ sudo apt-get dist-upgrade
$ sudo apt-get autoremove
```

(2) 删除一些不需要的内置软件：

```
$ sudo apt-get remove libreoffice-common
$ sudo apt-get remove unity-webapps-common
$ sudo apt-get autoremove
```

(3) 启用图标点击最小化操作：

```
$ gsettings set org.gnome.shell.extensions.dash-to-dock click-action 'minimize'
```

(4) 更新和升级系统：

```
$ sudo apt update
$ sudo apt upgrade
```

(5) 安装 Git：

```
$ sudo apt-get install git
```

配置 git 的两个重要信息，user.name 和 user.email，终端输入如下命令即可设置：

```
$ git config --global user.name "Your Name"
$ git config --global user.email "email@example.com"
(6) 支持挂载 exfat:
$ sudo apt-get install exfat-fuse
(7) 安装 g++ gcc 开发必备编译库（为之后安装 CUDA 做准备）:
$ sudo apt-get install build-essential
(8) 为了支持 ssh server, 需要:
$ sudo apt-get install openssh-server
$ sudo /etc/init.d/ssh start
$ sudo service ssh start
(9) 为了防止 Ubuntu 系统被破坏了, 我们需要一个可以进行备份和还原的工具:
```

TimeShift:

```
$ sudo apt-add-repository -y ppa:teejee2008/ppa
$ sudo apt update
$ sudo apt install timeshift
```

(10) 接着, 需要安装 Anaconda:

```
$ sh Anaconda3-2019.10-Linux-x86_64.sh
```

安装过程中需要注意选择 conda init 设置为 yes 以方便我们管理 Python 环境并将 Ubuntu 系统的 Python 环境设置为 Anaconda, 如果你还想要使用原来的 Python 环境, 可以在终端输入如下命令:

```
$ conda config --set auto_activate_base false
```

Anaconda 的打开使用命令: anaconda-navigator。

(11) 最后, 还需要安装 vscode:

```
$ sudo dpkg -i code_1.39.2-1571154070_amd64.deb
```

vscode 的打开使用命令 code 即可。

Ubuntu 系统的深度学习基础环境已经搭建完毕! 下面需要安装 CUDA 与 cuDNN。

因为安装 CUDA 是一个很危险的行为, 设置出错很容易把系统玩崩, 所以可以先使用 TimeShift 备份当前系统。做深度学习, 要用到 NVIDIA 的显卡, 因此需要改显卡驱动, 禁用 nouveau。即以管理员是身份打开/etc/modprobe.d/blacklist.conf 文件, 然后添加内容: blacklist nouveau #添加数据用来禁用 nouveau。而打开文件我们可以使用 vscode 进行文件编辑:

```
$ sudo code /etc/modprobe.d/blacklist.conf
```

首先进入 CUDA 安装包所在目录运行:

```
$ sudo ubuntu-drivers autoinstall
$ sudo sh cuda_10.0.130_410.48_linux.run
$ sudo sh cuda_10.0.130.1_linux.run
```

注意: 最好不要选择安装 OpenGL 库, 否则可能无法顺利安装 CUDA。如果下载速度很慢可以修改下载源为阿里云。安装完毕之后, 运行 nvidia-smi 检查 CUDA 是否安装正确。

最后需要安装 cuDNN:

```
sudo dpkg -i libcudnn7_7.6.4.38+cuda10.0_amd64.deb
```

1.1.3 设置清华镜像

为了提高 pip 与 conda 安装软件包的速度, 需要设置清华镜像。对于 pip, Windows10 设置的方法是一样的, 即:

```
$ pip install --upgrade pip -U # -U == --user  
$ pip config set global.index-url https://pypi.tuna.tsinghua.edu.cn/simple
```

而对于 conda，在 Windows10 中这样设置：

```
$ conda config --add channels https://mirrors.tuna.tsinghua.edu.cn/anaconda/pkgs/free/  
$ conda config --add channels https://mirrors.tuna.tsinghua.edu.cn/anaconda/pkgs/main/
```

在 Ubuntu18.04 中设置 conda，需要借助 vscode 修改用户目录下的 .condarc，即 code~/.condarc，然后添加如下内容：

```
channels:  
  - defaults  
show_channel_urls: true  
default_channels:  
  - https://mirrors.tuna.tsinghua.edu.cn/anaconda/pkgs/main  
  - https://mirrors.tuna.tsinghua.edu.cn/anaconda/pkgs/free  
  - https://mirrors.tuna.tsinghua.edu.cn/anaconda/pkgs/r  
custom_channels:  
  conda-forge: https://mirrors.tuna.tsinghua.edu.cn/anaconda/cloud  
  msys2: https://mirrors.tuna.tsinghua.edu.cn/anaconda/cloud  
  bioconda: https://mirrors.tuna.tsinghua.edu.cn/anaconda/cloud  
  menpo: https://mirrors.tuna.tsinghua.edu.cn/anaconda/cloud  
  pytorch: https://mirrors.tuna.tsinghua.edu.cn/anaconda/cloud  
  simpleitk: https://mirrors.tuna.tsinghua.edu.cn/anaconda/cloud
```

1.2 安装深度学习框架

在安装深度学习框架之前，先了解一下 Anaconda，一个用于科学计算的 Python 发行版，支持 Linux、Mac、Windows，包含众多流行的科学计算、数据分析的 Python 包。

Anaconda 提供了十分强大的 Python 环境与包的管理机制，本小节将利用它这一特性来说明如何在同一台机器上创建多个深度学习框架。由于深度学习框架的安装在 Windows10 与 Ubuntu 18.04 上是一样的，所以下面我便不在言明是在哪个系统上进行操作。

如果想要使用和管理多个框架，如果将它们均安装在同一环境之下，往往很容易发生包的冲突问题。因而，为了让深度学习框架之间不发生冲突，需要借助 conda 对 Python 的环境进行管理，下面看看如何创建新的 Python 环境。

(1) 打开 Anaconda Navigator 并依次选择 Environments，Create，接着输入环境的名字并选择 Python 版本，如图 1.9 所示。

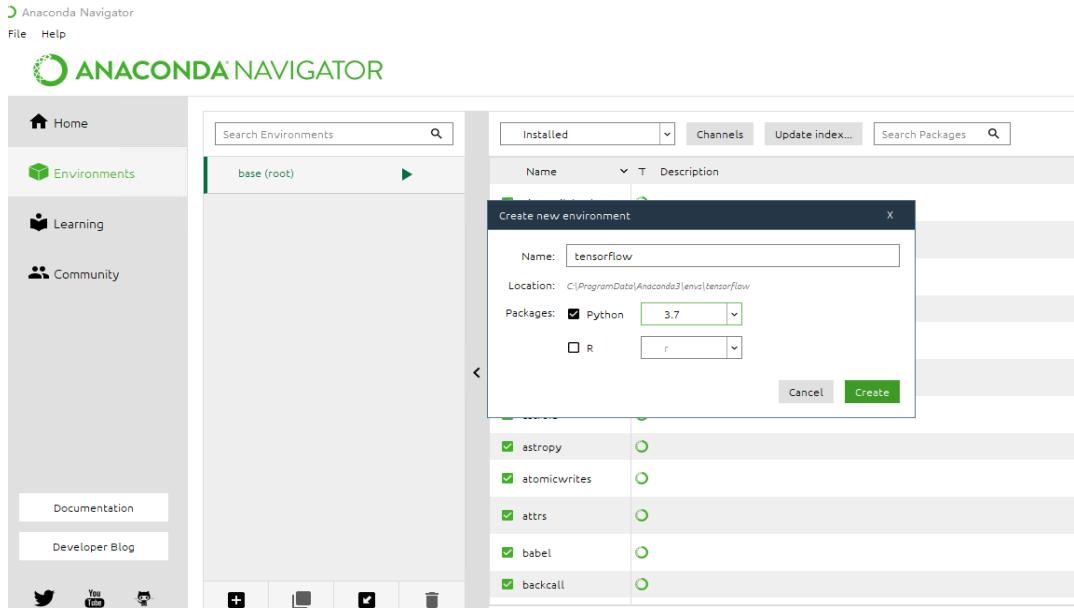


图 1.9 创建一个新环境

下面以 TensorFlow 为例，说明如何安装包。

(2) 在新创建的环境中打开终端，如图 1.10 所示。

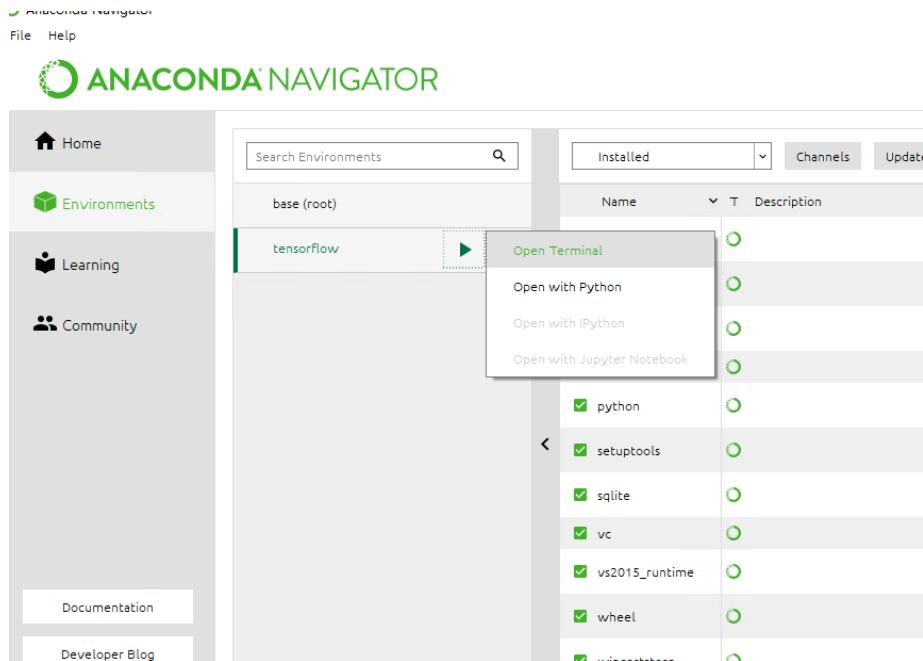


图 1.10 打开新的终端

然后，在终端输入命令：

```
$ pip install tensorflow-gpu
```

完成 TensorFlow 框架的 GPU 版本安装。

我们不仅仅满足于在终端运行 Python 程序，如果想要在 Notebook，也可以运行新创建的环境岂不妙哉！

在 Tensorflow 环境的终端输入：

```
$ conda install ipykernel  
$ python -m ipykernel install --user --name tensorflow --display-name "tensorflow"  
$ pip install jupyter
```

此时，再次打开 Notebook，则会呈现两个环境：tensorflow 与 python 3，如图 1.11 所示。



图 1.11 两个 Python 环境

选择 tensorflow，进入 Notebook 编辑界面，我们测试 GPU 是否可以正常使用，如图 1.12 所示。

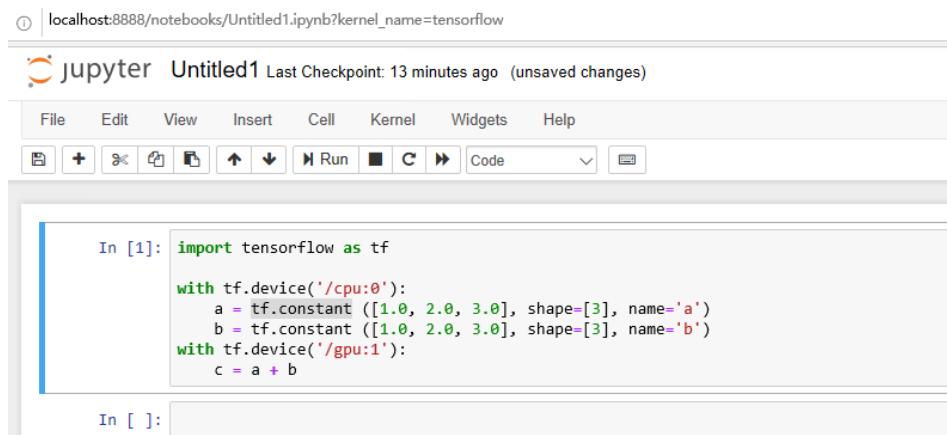


图 1.12 测试 tensorflow gpu 是否使用正常

代码没有报错，说明 GPU 配置完成。需要注意的是在 Ubuntu 系统上如果测试 GPU 失败，可以尝试运行如下命令：

```
$ conda install cudatoolkit=10.0  
$ conda install cudnn=7.6
```

这样你便可以拥有两个互不干扰的 jupyter 环境！为了方便以后切换不同的深度学习框架，按照上述的步骤分别创建 MXNet、Pytorch 深度学习环境。

安装 MXNet 的命令是：

```
$ conda install ipykernel  
$ python -m ipykernel install --user --name mxnet --display-name "MXNet"  
$ pip install jupyter  
$ pip install mxnet-cu100
```

安装 Pytorch 的命令是：

```
$ conda install ipykernel  
$ python -m ipykernel install --user --name torch --display-name "Pytorch"  
$ pip install jupyter
```

```
$ conda install pytorch torchvision cudatoolkit=10.0 -c pytorch
```

1.3 桥接 Ubuntu 与 Windows

对于同一个局域网的两台机器，一台安装了 Windows10，一台安装了 Ubuntu18.04。我们想要利用 SSH 协议桥接这两台机器，对于 Ubuntu 系统我们已经配置好了其 SSH Server，而 Windows10 需要我们做一些工作。

在 Windows Server2019 或 Windows 10 1809 及以上设备上安装 OpenSSH 很简单。只需要通过 PowerShell 输入如下命令安装 OpenSSH 即可。（参考 https://docs.microsoft.com/zh-cn/windows-server/administration/openssh/openssh_install_firstuse）

```
$ Get-WindowsCapability -Online | ? Name -like 'OpenSSH*'  
$ # Install the OpenSSH Server  
$ Add-WindowsCapability -Online -Name OpenSSH.Server~~~~0.0.1.0
```

而对于低版本的系统，则需要下载 <https://github.com/PowerShell/Win32-OpenSSH/releases> 中的代码，然后将其解压并将解压后的文件目录添加 Windows 系统的 Path 环境变量之中。然后，还需要在终端（PowerShell）打开解压后的文件目录并输入如下命令来完成 SSH Server 的配置：

```
$ .\install-sshd.ps1  
$ .\FixHostFilePermissions.ps1
```

配置好 SSH Server 之后，我们便可以利用 SSH 连接这两台机器了。

比如，在 Windows10 系统打开 PowerShell 并使用 SSH 连接 Ubuntu18.04 的机器：

```
$ ssh xinet@192.168.42.7
```

其中 xinet@192.168.42.7 的组成是 用户名@IP 地址。

这样你便可以像使用 Ubuntu 系统的终端进行操作

- (1) conda activate pytorch 启动我们之前创建的 Pytorch 环境
- (2) pip install 或者 conda install 是用来安装 Python 包的命令。
- (3) 待你在终端的操作完成之后，需要使用命令 exit 退出 ssh 连接。

1.4 Ubuntu 多用户共享使用深度学习环境

我们可以使用命令 sudo adduser 用户名 的方式创建新用户。创建新用户之后，便可以令其使用共享环境。比如我们在用户 A 之中配置了深度学习环境，而用户 B 想要使用用户 A 的深度学习环境只需要运行命令 source /home/A/.bashrc 即可激活深度学习环境。如果不想要输入这么长或者不想用户之间的环境进行干扰，可以在 root 权限之下做如下操作：

```
$ su root  
$ cat /home/A/.bashrc >> /home/B/.bashrc
```

之后您只需要运行如下命令即可激活深度学习环境：

```
$ source ~/.bashrc
```

情景：我们不想建立远程桌面且又想要使用 Ubuntu 系统的 Jupyter Notebook，该怎么办？

我们可以使用 MobaXterm 软件来解决该情景问题。具体操作方法：

进入网站 <https://mobaxterm.mobatek.net/download.html> 下载软件。安装好之后，创建一个

Session, 如图 1.13 所示。

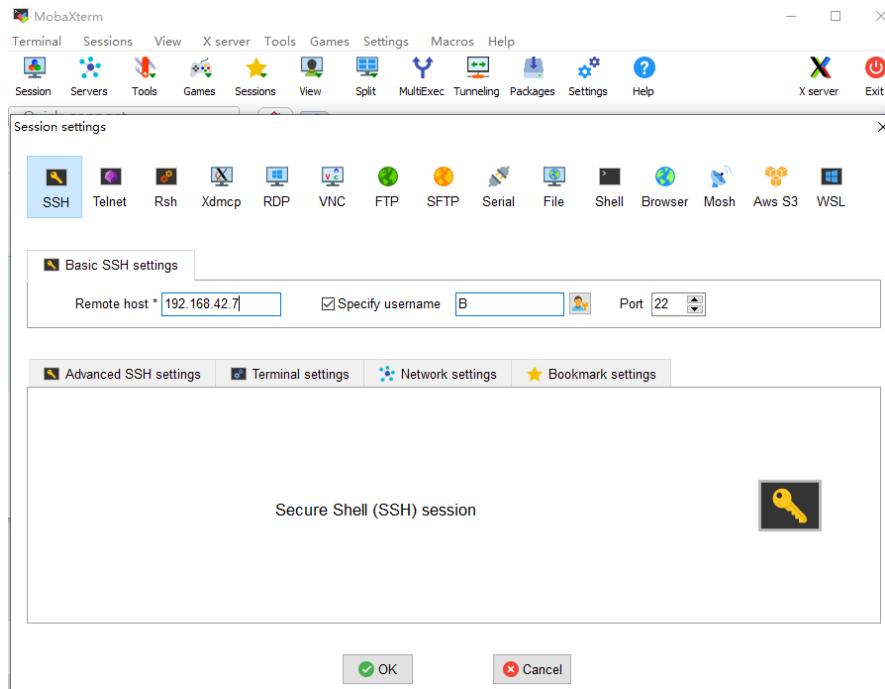


图 1.13 创建一个 Session

点击 OK 之后, 进入 Ubuntu 系统下的用户 B 所在账户下的终端。虽然这是一个终端, 但是此终端还可以做一些仿真工作, 比如打开 Jupyter Notebook, 如图 1.14 所示。

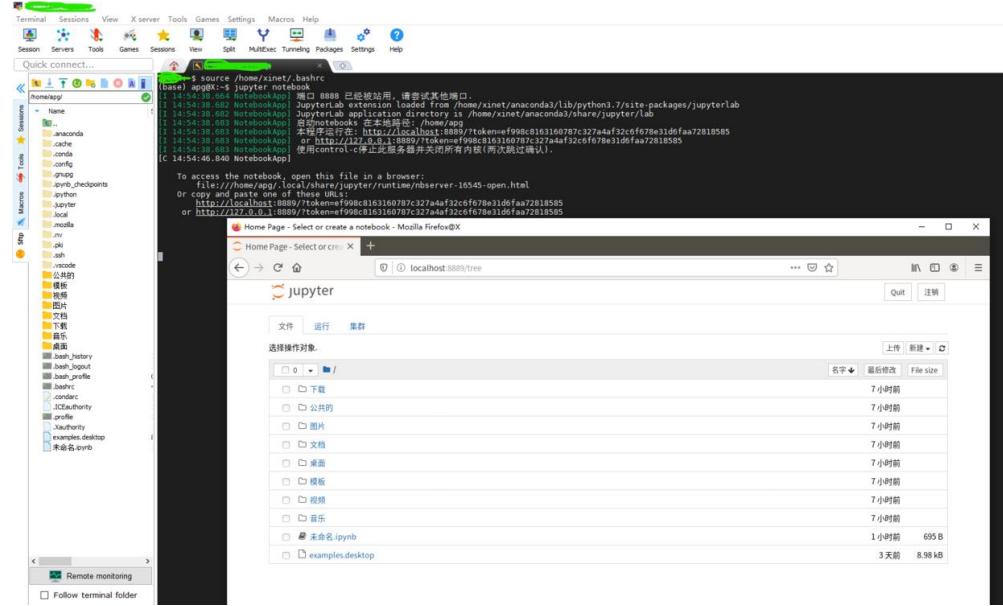


图 1.14 在 windows 系统打开远端的 Ubuntu 系统的 Jupyter Notebook

不仅如此, 这里打开了一个浏览器服务器, 你可以在此浏览器之中下载东西, 然后将下载的东西拖曳回 Windows 系统。更多精彩内容可以查看 MobaXterm 官方网站进行了解。

1.5 本章小结

本章主要介绍如何创建一个具有可塑性的计算机视觉工具集。该工具集集成了 TensorFlow、MXNet、Pytorch 这三种深度学习框架，介绍了如何在 Windows10 与 Ubuntu18.04 上分别搭建深度学习基础环境，以及如何建立二者之间的 SSH 协议。同时介绍如何令不同的深度学习环境进行隔离的策略。

第 2 章 计算机视觉之利器：vscode

本章主要介绍计算机视觉的一个十分强大的利器：vscode。首先介绍 vscode 的一些简单使用方法并安装 Python 插件，最后介绍一个十分强大的文档编辑插件 MPE。关于 Python 插件的使用将在下一章进行介绍。

2.1 本书的注意事项

虽然在第 1 章介绍了计算机视觉的深度学习工具集，但是它对电脑的硬件要求很高，需要你的设备支持 GPU。如果你只是想学习计算机视觉且电脑配置也不高，那么仅仅考虑使用 CPU 也便是足够了。因为我不是计算机专业出身，是由数学专业转行过来的，所以对于计算机是一些名词并不是很了解。故而，本书我是以数学专业的思想，乃至是一个计算机菜鸟的视角解读编程思想。我们需要关心 GPU 与 CPU 是什么，只需要知道它们是用来做运算的工具即可。

由于第 1 章对设备的要求很高，如果您的设备是一个普通的电脑，不需要考虑安装那么多软件，只需要安装 Anaconda，Git，vscode 即可。这三个软件是十分强大的，它们会为您的工作和学习提供极大的便利。在之后的章节会不断的使用到它们，也会被它们的强大逐渐吸引的。从本章开始，将逐步进入编程的世界，使用 vscode 以及 Anaconda 的 Jupyter Notebook 作为我们的编程利器。

本书尽量以 Windows 系统作为主要使用的环境，对于其他系统，大体适用于本书的内容。

2.2 安装 vscode 必备插件

vscode 是微软官方提供的一个十分强大的编辑器，它提供了十分强大的插件库，您只需要安装您需要的插件，便可将 vscode 打造称为史上最强大的工作、学习、写作利器。下面我们逐一揭开 vscode 的神秘面纱。

刚刚安装好的 vscode 的界面如图 2.1 所示是英文的界面。如果不使用英文界

面，可以如图 2.2 所示下载简体中文插件，安装完毕之后的界面将会变成简体中文的界面。

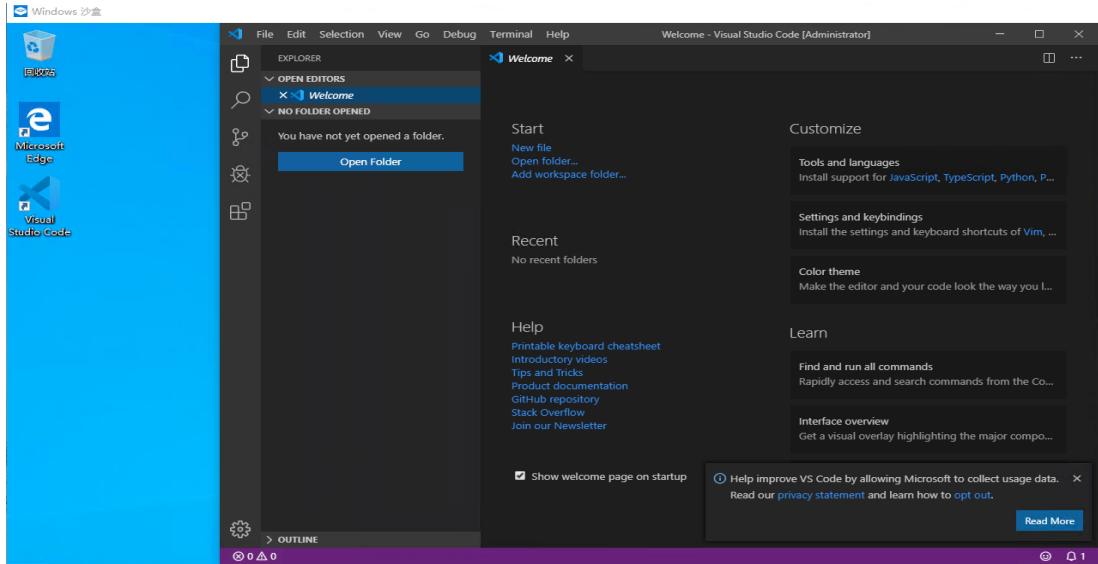


图 2.1 vscode 最初状态

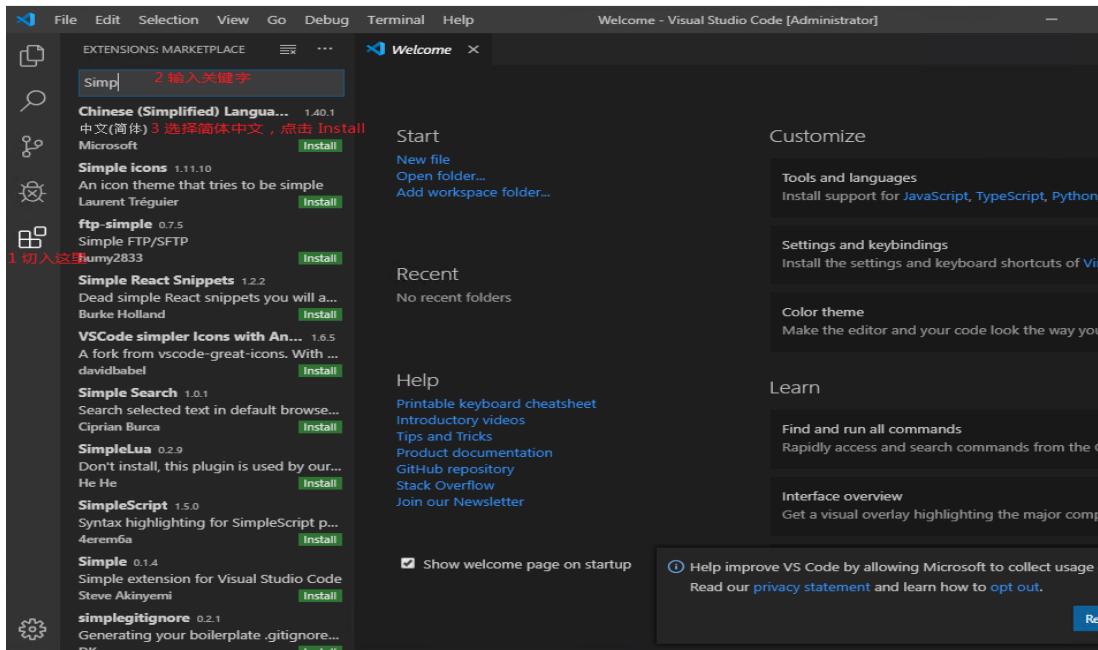


图 2.2 安装简体中文插件

继续像图 2.2 那样搜索 Markdown Preview Enhanced 与 python 插件并分别安装它们，这样便完成了 vscode 的必备插件的安装工作。这里的 python 插件是帮助你进行 Python 代码的编写而做的准备，而 Markdown Preview Enhanced 则为你提供了十分强大的 Markdown 语法支持，提高了文档编辑或者写作效率。vscode 也支持缩放功能：Ctrl+（放大），Ctrl-（缩小）。

vscode 的应用商店 (<https://marketplace.visualstudio.com/>) 提供了许多实用的插

件，可以按需选择安装，本章不做展开。

安装完这 3 个插件，便可以很好的体验编程和写作的乐趣了。下面将详细介绍 Markdown Preview Enhanced 这个插件。

2.3 MPE 使用的必要性

Markdown Preview Enhanced 简称 MPE，其官方提供了一份十分详细的中文教程：<https://shd101wyy.github.io/markdown-preview-enhanced/#/zh-cn/>。MPE 为 Markdown 提供了许多强大的功能。

看到这里，你可能会有疑问：Markdown？本书是关于计算机视觉的书籍，Markdown 好像与本书的主题无关？其实不是这样的。想象这样一个场景：有一天获得了一份关于计算机视觉的源代码，你想要对该源码进行重构让其适用于项目。奈何该源码几乎没有提供文档说明，你盯着代码傻傻分不清楚那个代码是“父”，那个代码是“子”，甚至你想要正常运行某个程序都无处下手。

是不是很可怕？这种场景是很常见的，并不是特例。该场景说明了一份详细的代码说明对后期的代码开发和维护是十分关键的。

那为什么一定要是 Markdown，不能是.docx 等其他格式的文档对代码进行说明呢？因为 Markdown 是轻量型的文件占用空间很小且十分方便 Git（后面章节再详细介绍）对其进行管理。而像.docx 等文件格式是二进制文件，对 Git 并不友好，并且传输过程中极可能出现乱码的情况。不仅仅如此 Markdown 还可以作为您代码的笔记进行保存和分享。下面我将详细阐述 Markdown 是如何做文档编辑工作的。

2.4 Markdown 使用手册

下面我挑选几个比较常用的功能进行介绍。

2.4.1 Markdown 基本要素

二八定律说：百分之二十的知识解决百分之八十的问题。本文将 Markdown 语法分为常用标记，次常用标记和不常用标记。如果您想学习和使用 Markdown，本书建议：

- 常用标记要先花一些时间熟记，后面经常使用的话也会慢慢形成习惯；
- 次常用标记要有基本的印象，能记住也是可以的；
- 不常用标记看看就好，等到使用的时候再百度一下。

markdown 是 html 的一个子集，它通过在文本中插入标示性的符号来构造和排版文章，其源文件是以.md 为后缀名的纯文本格式进行保存。为了排版的可读性考虑，本书建议你遵循段落前后为空行的设定。

2.4.2 常用标记

常用标记要先花一些时间熟记，后面经常使用的话就会形成习惯。

1. 标题

使用#表示标题，一级标题使用一个#，二级标题使用两个##，以此类推，共有六级标题。#和标题之间最好加一个空格。也可以使用<h1>标题名称</h1>的形式定义标题，其中 h1 表示一级标题，以此类推到六级标题。具体示例如下：

```
# 这是 <h1> 一级标题 </h1>
## 这是 <h2> 二级标题 </h2>
### 这是 <h3> 三级标题 </h3>
#### 这是 <h4> 四级标题 </h4>
##### 这是 <h5> 五级标题 </h5>
###### 这是 <h6> 六级标题 </h6>
```

如果想要给标题添加 id 或者 class，请在标题最后添加{#id .class1 .class2}。例如：

```
# 这个标题拥有 1 个 id {#my_id}
# 这个标题有 2 个 classes {.class1 .class2}
```

2. 目录

将[TOC]放在一级标题的前面可用于生成目录。如果你想要在你的 TOC 中排除一个标题，请在你的标题后面添加{ignore=true} 即可。

3. 引用

使用>表示引用，>>表示引用里面再套一层引用，依次类推。

例 1，引用示例：

```
> 这是一级引用
>> 这是二级引用
>>> 这是三级引用
>
> 这是一级引用
```

效果：

```
这是一级引用 > 这是二级引用 >> 这是三级引用
```

```
这是一级引用
```

注意：如果>和>>嵌套使用的话，从>>退到>时，必须之间要加一个空行或者>作为过渡，否则默认认为下一行和上一行是同级别的引用。如示例所示。引用标记里可以使用其他标记，如：有序列表或无序列表标记，代码标记等。

4. 代码块

使用```表示代码块。(要与您的文档上行文字之间空一行)

例 2，展示 Python 代码块的写法：

```
```python
a = range(10)
for i in a:
 print(i)
```
```

注意：`这个符号是在 Esc 键下面，切换到英文下即可。```后面的

`python` 表示此段代码为 python 代码，Markdown 会自行使用 python 代码颜色渲染。

使用 ` 表示行内代码。比如：

这是`javascript`代码

如果你想要你的代码块可以显示代码的行数，只要添加`line-numbers class`就可以了。比如：

```
```python {.line-numbers}
A = 1
B = 2
C = A+ B
...```

```

显示的效果如图 2.3 所示。

```
1 | A = 1
2 | B = 2
3 | C = A+ B
```

图 2.3 代码显示行数

## 5.列表

使用`1. 2. 3.` 表示有序列表，使用`\*`、`-` 或`+` 表示无序列表。下面举例说明。

例 3，有序列表代码示例：

1. 第一点
2. 第二点
3. 第三点

例 4，无序列表代码示例

```
+ 呵呵
 * 嘉嘉
 - 嘻嘻
 - 吼吼
 - 嘎嘎
 + 桀桀
* 哈哈
```

注意：

- 无序列表或有序列表标记和后面的文字之间要有一个空格隔开。
- 有序列表标记不是按照你写的数字进行显示的，而是根据当前有序列表标记所在位置显示的，如示例 3 所示。
- 无序列表的项目符号是按照实心圆、空心圆、实心方格的层级关系递进的，如例 4 所示。通常情况下，同一层级使用同一种标记表示（否则，显示效果不佳），便于自己查看和管理。

## 6.粗体和斜体

- 使用\*\*或者 \_\_(双下划线)表示粗体。
- 使用 \* 或者 \_ 表示斜体。

下面看几个示例。

例 5:

```
粗体 1 _粗体 2_
斜体 1 _斜体 2_
```

效果:

**粗体 1 粗体 2**

*斜体 1 斜体 2*

注意: 前后的\*或\_与要加粗或倾斜的字体之间不能有空格。

## 7. 表格

下面的 - 的个数可以为任意大于 1 的数。

`-----`为右对齐。

`:-----`为左对齐。

`:-----:`为居中对齐。

`` 与 `` 之间不要有空格, 否则对齐会有些不兼容

例 6, 列表示例:

序号	交易名	交易说明	备注
1	prfcfg	菜单配置	可以通过此交易查询到所有交易码和菜单的对应关系
2	gentmo	编译所有交易	
100000	sysdba	数据库表模型汇总	

效果如表 2.1 所示。

表 2.1 列表示意图

序号	交易名	交易说明	备注
1	prfcfg	菜单配置	可以通过此交易查询到所有交易码和菜单的对应关系
2	gentmo	编译所有交易	
100000	sysdba	数据库表模型汇总	

## 8. 分割线

使用---或者\*\*\*或者\* \* \*表示水平分割线。

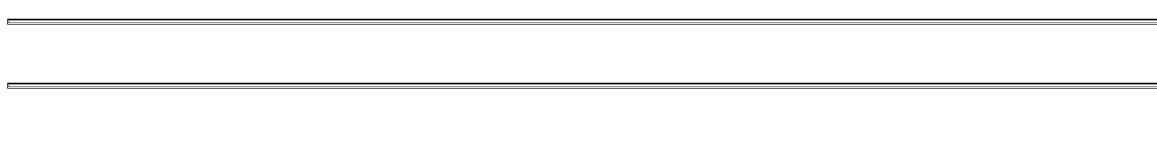
例 7:

```



```

效果:



注意：

- 只要\*或者-大于等于三个就可组成一条平行线。
- 使用---作为水平分割线时，要在它的前后都空一行，防止---被当成标题标记的表示方式。

## 9.链接

使用[](link "Optional title")表示行内链接。其中：[]内的内容为要添加链接的文字。link 为链接地址。Optional title 为显示标题。显示效果为在你将鼠标放到链接上后，会显示一个小框提示，提示的内容就是 Optional title 里的内容。

例 8，行内链接：

这就是我们常用的地址：[简书](http://www.jianshu.com "简书")

效果：

这就是我们常用的地址：简书

例 9，参考式链接：

这就是我们常用的地址：[Baidu][1]

[1]:www.baidu.com "百度一下，你就知道"

注意：参考式链接和行内链接的显示效果是一样的，但是在编辑状态下的使用情况不一样。行内连接紧跟链接文字，可以在看到链接文字的同时清楚的知道链接地址，但是不便于多次重复利用。参考式链接可以重复使用，但一般都是将一些链接放在一起统一管理，如一段文字后面或文章结尾，因此在找到链接和链接文字的对应关系上有些麻烦。

## 10.导入图片

我们使用![Alt text](/path/to/img.jpg Optional title)的方式导入图片。其中：Alt text 为如果图片无法显示时显示的文字；/path/to/img.jpg 为图片所在路径；Optional title 为显示标题。显示效果为在你将鼠标放到图片上后，会显示一个小框提示，提示的内容就是 Optional title 里的内容。导入的图片路径可以使用绝对路径也可以使用相对路径，建议使用相对路径。

## 11.反斜杠

使用\表示反斜杠。在你不想显示 Markdown 标记时可以使用反斜杠。

例 11：

\\*这里不会显示斜体\\*

效果：

\*这里不会显示斜体\*

### 2.4.3 次常用标记

次常用标记要有基本的印象，能记住也是可以的。

#### 1.注脚

在需要添加注脚的文字后加上脚注名字[^注脚名字]被称为加注，然后在文本的任意位置(一般在最后)添加脚注，脚注前必须有对应的脚注名字。

注意：经测试注脚与注脚之间必须空一行，不然会失效。成功后会发现，即使你没有把注脚写在文末，经 Markdown 转换后，也会自动归类到文章的最后。

下面来看一个例子：

使用 Markdown<sup>[^1]</sup>可以效率的书写文档，直接转换成 HTML<sup>[^2]</sup>，你可以使用 Leanote<sup>[^Le]</sup> 编辑器进行书写。

[^1]: Markdown 是一种纯文本标记语言(冒号与文字之间需要有空格)

[^2]: HyperText,超文本标记语言

[^Le]: 开源笔记平台，支持 Markdown 和笔记直接发为博文

## 2.删除线

使用~~表示删除线，示例如下：

~~这是一条删除线~~

效果：

~~这是一条删除线~~

注意：~~ 和要添加删除线的文字之间不能有空格。

## 3.缩略

比如如下的写法：

\*[W3C]: World Wide Web Consortium

再次出现 W3C，当鼠标悬停会显示其全称。

### 2.4.4 不常用标记

不常用标记看看就好，等到使用的时候再百度一下。

#### 1.自动链接

Markdown 支持以比较简短的自动链接形式来处理网址和电子邮件信箱，只要是用<>包起来，Markdown 就会自动把它转成链接。一般网址的链接文字就和链接地址一样，例如：

```
<http://example.com/>
<address@example.com>
```

效果：

<http://example.com/>

[address@example.com](mailto:address@example.com)

引用存储文件形式为[example](.../.../example.md)。

#### 2.语义标记与标签

有时候我们也需要去设置一些特殊的符号，下面的语义标记与标签可能会帮到你。在表 2.2 记录了一些斜体，加粗等标记，而表 2.3 记录了上下标等标记方法。

除了表 2.2 和表 2.3 列出的特殊标记之外，也可以使用 ^ 和 ~ 标记上下标，比如 LATEX 这样的符号，也可以这样输入 L^A^T~E~X。也可以组合各种标记，比如 LATEX 便是由\*L^A^T~E~X\* 组合得到的效果，具有示例如表 2.4 所示。

表 2.2 语义标记

效果	代码
斜体	*斜体*
斜体	_斜体_
加粗	**加粗**

<b>加粗+斜体</b>	***加粗+斜体***
<b>加粗+斜体</b>	**_加粗+斜体_*
删除线	~~删除线~~

表 2.3 语义标签

效果	代码
斜体	<i>斜体</i>
加粗	<b>加粗</b>
强调	<em>强调</em>
Za	Z<sup>a</sup>
Za	Z<sub>a</sub>
Ctrl	<kbd>Ctrl</kbd>

表 2.4 另类的上下标

符号	说明	示例
^ ^	上标	猫 <sup>狗</sup>
~ ~	下标	狗 <sub>猫</sub>

## 2.5 公式使用参考

如果代码量很大，足够组成一个项目，而该项目有涉及到许多数学公式，那么，需要学习如何使用 Markdown 编写数学公式。

### 2.5.1 如何插入公式

- 行中公式(放在文中与其它文字混编)可以用如下方法表示：\$数学公式\$
- 独立公式可以用如下方法表示：\$\$数学公式\$\$

行内公式示例：

```
$
J_{\alpha}(x) = \sum_{m=0}^{\infty} \frac{(-1)^m}{m!} \Gamma(m + \alpha + 1)
\left(\frac{x}{2m + \alpha}\right)^{2m + \alpha} \text{, 行内公式示例}
$
```

显示：

$$J_\alpha(x) = \sum_{m=0}^{\infty} \frac{(-1)^m}{m!\Gamma(m+\alpha+1)} \left(\frac{x}{2}\right)^{2m+\alpha}, \text{ 行内公式示例。}$$

独立公式示例：

```
$$
J_\alpha(x) = \sum_{m=0}^{\infty} \frac{(-1)^m}{m!\Gamma(m+\alpha+1)} \left(\frac{x}{2}\right)^{2m+\alpha}, \text{ 独立公式示例}
$$
```

显示：

$$J_\alpha(x) = \sum_{m=0}^{\infty} \frac{(-1)^m}{m!\Gamma(m+\alpha+1)} \left(\frac{x}{2}\right)^{2m+\alpha}, \text{ 独立公式示例。}$$

## 2.5.2 如何输入公式的上下标

$\wedge$  表示上标,  $\_$  表示下标。如果上下标的内容多于一个字符, 需要用 {} 将这些内容括成一个整体。上下标可以嵌套, 也可以同时使用, 效果如图 2.4 所示。



图 2.4 上下标使用展示

## 2.5.3 如何输入括号和分隔符

)、[] 和 | 表示符号本身, 使用  $\{\}$  来表示 {}。当要显示大号的括号或分隔符时, 要用  $\left.$  和  $\right.$  命令。一些特殊的括号, 如表 2.4 所示。

表 2.4 各种括号的表达

输入	显示
$\$ \$ \langle \rangle$ 表达式 $\rangle$	$\langle \rangle$
$\$ \$ \lfloor \rfloor$ 表达式 $\rfloor$	$\lfloor \rfloor$
$\$ \$ \lfloor \rfloor$ 表达式 $\rfloor$	$\lfloor \rfloor$

$\$\$\backslash lbrace$ 表达式 $\backslash rbrace\$\$$	$\{表达式\}$
--------------------------------------------------------	-----------

再看一个例子:  $\$\$f(x,y,z) = 3y^2z \left( 3 + \frac{7x+5}{1+y^2} \right)$   
显示:

$$f(x, y, z) = 3y^2z \left( 3 + \frac{7x+5}{1+y^2} \right)$$

#### 2.5.4 如何输入分数

通常使用  $\frac{分子}{分母}$  命令产生一个分数，分数可嵌套。便捷情况可直接输入  $\frac{ab}{cd}$  来快速生成一个  $\frac{ab}{cd}$ 。如果分式很复杂，亦可使用 分子 分母 命令，此时分数仅有一层，具体实例如图 2.5 所示。

The screenshot shows a LaTeX editor interface. On the left, the code is displayed in a monospaced font:

```

1 # test
2
3 ``$\frac{a-1}{b-1} ; \text{and} ; \frac{a+1}{b+1}$``
4
5 显示:
6
7 $$
8 \frac{a-1}{b-1} ; \text{and} ; \frac{a+1}{b+1}
9 $$
10

```

On the right, the rendered output is shown in a preview window:

**test**

$\frac{a-1}{b-1}$  and  $\frac{a+1}{b+1}$

图 2.5 输入分数的实例

#### 2.5.5 如何输入开方

使用  $\sqrt[n]{被开方数}$  命令输入开方。比如  $\$\$\\sqrt{2}$   $\backslash quad$  and  $\backslash quad \\sqrt[3]{3}$  显示为:

$$\sqrt{2} \quad \text{and} \quad \sqrt[3]{3}$$

#### 2.5.6 如何输入省略号

数学公式中常见的省略号有两种，表示与文本底线对齐的省略号，表示与文本中线对齐的省略号，实例如图 2.6 所示。

```

1 # 如何输入省略号
2
3 数学公式中常见的省略号有两种, \ldots 表示与文本底线对齐的省略号,
4 \cdots 表示与文本中线对齐的省略号。
5
6 例子:
7
8 \$$f(x_1,x_2,\underbrace{\ldots}_{\text{ldots}},x_n) = x_1^2 +
9 x_2^2 + \underbrace{\cdots}_{\text{cdots}} + x_n^2$$
10
11 显示:
12
13 \$$
14 f(x_1,x_2,\underbrace{\ldots}_{\text{ldots}},x_n) = x_1^2 + x_2^2 + \cdots + x_n^2$$
15
16

```

如何输入省略号

数学公式中常见的省略号有两种, \ldots 表示与文本底线对齐的省略号, \cdots 表示与文本中线对齐的省略号。

例子:

$$f(x_1, x_2, \underbrace{\ldots}_{\text{ldots}}, x_n) = x_1^2 + x_2^2 + \underbrace{\cdots}_{\text{cdots}} + x_n^2$$

图 2.6 如何输入省略号

## 2.5.7 如何输入矢量

使用 \vec{矢量} 来自动产生一个矢量。也可以使用 \overrightarrow 等命令自定义字母上方的符号。

例子:

```
 $$\vec{a} \cdot \vec{b} = 0$$
```

显示:

$$\vec{a} \cdot \vec{b} = 0$$

例子:

```
 $$\overleftarrow{xy} \quad \text{and} \quad \overleftarrow{xy} \quad \text{and} \quad \overrightarrow{xy}$$
```

显示:

$$\overleftarrow{xy} \quad \text{and} \quad \overleftarrow{xy} \quad \text{and} \quad \overrightarrow{xy}$$

## 2.5.8 如何输入积分

使用 \_ 积分下限^积分上限 {被积表达式} 来输入一个积分。参考图 2.7 的实例:

```

1 # 如何输入积分
2
3 使用 \int_ 积分下限^积分上限 {被积表达式} 来输入一个积分。
4
5 例子:
6
7 \$$\int_0^1 {x^2} \, dx$$
8
9 显示:
10 \$$\int_0^1 {x^2} \, dx$$

```

如何输入积分

使用 \int\_ 积分下限^积分上限 {被积表达式} 来输入一个积分。

例子:

$$\int_0^1 x^2 \, dx$$

图 2.7 积分输入

## 2.5.9 如何输入极限运算

使用  $\lim_{\text{变量} \rightarrow \text{表达式}}$  表达式 来输入一个极限。如有需求，可以更改  $\rightarrow$  符号至任意符号。

例子：

```
$$ \lim_{n \rightarrow +\infty} \frac{1}{n(n+1)} \quad \text{and} \quad \lim_{x \leftarrow \text{示例}} \frac{1}{n(n+1)} $$
```

显示：

$$\lim_{n \rightarrow +\infty} \frac{1}{n(n+1)} \quad \text{and} \quad \lim_{x \leftarrow \text{示例}} \frac{1}{n(n+1)}$$

## 2.5.10 如何输入累加、累乘运算

使用  $\sum_{\text{下标表达式}}^{\text{上标表达式}}$  {累加表达式} 来输入一个累加。与之类似，使用  $\prod \bigcup \bigcap$  来分别输入累乘、并集和交集。此类符号在行内显示时上下标表达式将会移至右上角和右下角。

例子：

```
$$ \sum_{i=1}^n \frac{1}{i^2} \quad \text{and} \quad \prod_{i=1}^n \frac{1}{i^2} \quad \text{and} \quad \bigcup_{i=1}^2 R $$
```

显示：

$$\sum_{i=1}^n \frac{1}{i^2} \quad \text{and} \quad \prod_{i=1}^n \frac{1}{i^2} \quad \text{and} \quad \bigcup_{i=1}^2 R$$

## 2.5.11 如何输入希腊字母

输入  $\backslash$  小写希腊字母英文全称和  $\backslash$  首字母大写希腊字母英文全称来分别输入小写和大写希腊字母。

对于大写希腊字母与现有字母相同的，直接输入大写字母即可，希腊字母一览表如表 2.5 所示。

表 2.5 希腊字母一览表

输入	显示	输入	显示
$\$\\alpha$$	$\alpha$	$\$A$$	$A$
$\$\\beta$$	$\beta$	$\$B$$	$B$
$\$\\gamma$$	$\gamma$	$\$\\Gamma$$	$\Gamma$
$\$\\delta$$	$\delta$	$\$\\Delta$$	$\Delta$
$\$\\epsilon$$	$\epsilon$	$\$E$$	$E$
$\$\\zeta$$	$\zeta$	$\$Z$$	$Z$
$\$\\eta$$	$\eta$	$\$H$$	$H$
$\$\\theta$$	$\theta$	$\$\\Theta$$	$\Theta$
$\$\\iota$$	$\iota$	$\$I$$	$I$
$\$\\kappa$$	$\kappa$	$\$K$$	$K$
$\$\\lambda$$	$\lambda$	$\$\\Lambda$$	$\Lambda$

$\$\\nu$$	$\nu$	$\$N$$	$N$
$\$\\mu$$	$\mu$	$\$M$$	$M$
$\$\\xi$$	$\xi$	$\$\\Xi$$	$\Xi$
$\$o$$	$o$	$\$O$$	$O$
$\$\\pi$$	$\pi$	$\$\\Pi$$	$\Pi$
$\$\\rho$$	$\rho$	$\$P$$	$P$
$\$\\sigma$$	$\sigma$	$\$\\Sigma$$	$\Sigma$
$\$\\tau$$	$\tau$	$\$T$$	$T$
$\$\\upsilon$$	$\upsilon$	$\$\\Upsilon$$	$\Upsilon$
$\$\\phi$$	$\phi$	$\$\\Phi$$	$\Phi$
$\$\\chi$$	$\chi$	$\$X$$	$X$
$\$\\psi$$	$\psi$	$\$\\Psi$$	$\Psi$
$\$\\omega$$	$\omega$	$\$\\Omega$$	$\Omega$

### 2.5.12 大括号和行标的使用

使用``\left` 和`\right` 来创建自动匹配高度的 (圆括号), [方括号]和{花括号}。在每个公式末尾前使用`\tag{行标}` 来实现行标的设定。下面来看一个例子:`

```
$$
\left(\frac{1 + \left(x, y \right)}{\left(\frac{x}{y} + \frac{y}{x} \right) \left(u + 1 \right)} + a \right)^{3/2}
\tag{行标}
$$
```

显示如图 2.8 所示。

$$f \left( \left[ \frac{1 + \{x, y\}}{\left( \frac{x}{y} + \frac{y}{x} \right) (u + 1)} + a \right]^{3/2} \right)$$

§
○
↑
  
(行标)

图 2.8 大括号和行标的使用

小技巧, 如图 2.9 所示。

1. `$\smash{\displaystyle \max_{\{0 \leq q \leq n-1\}} f(q) \leq n}$` 显示:  

$$\max_{0 \leq q \leq n-1} f(q) \leq n$$
2. `$f(x + \epsilon) \approx f(x) + f'(x) \epsilon + \mathcal{O}(\epsilon^2)$`  

$$(\epsilon^2).$, 显示:  $f(x + \epsilon) \approx f(x) + f'(x)\epsilon + \mathcal{O}(\epsilon^2).$$$
3. 求导符号使用 `$\text{d}x$`, 即:  $\text{d}x$

图 2.9 使用小技巧

### 2.5.13 字体转换

若要对公式的某一部分字符进行字体转换, 可以用 `{\字体 {需转换的部分字符}}` 命令, 其中 `\字体` 部分可以参照下表选择合适的字体。一般情况下, 公式默认设置为意大利体, 数学公式字体转换, 如图 2.10 所示。

输入	说明	显示实例
<code>\rm</code>	罗马体	<b>D</b>
<code>\cal</code>	花体	$\mathcal{D}$
<code>\it</code>	意大利体	<i>D</i>
<code>\Bbb</code>	黑板粗体	$\mathbb{D}$
<code>\bf</code>	粗体	<b>D</b>
<code>\mit</code>	数学斜体	$\mathit{D}$
<code>\sf</code>	等线体	$\mathsf{D}$
<code>\scr</code>	手写体	$\mathscr{D}$
<code>\tt</code>	打字机体	$\mathfrak{D}$
<code>\frak</code>	旧德式字体	$\mathfrak{D}$
<code>\boldsymbol</code>	黑体	<b>X</b> , $\mathbf{x}$

图 2.10 数学公式字体转换

## 2.6 将 Markdown 转换为 docx 文档

在 2.4 与 2.5 了解了 Markdown 的基本使用, 但是, 如果想要将其分享给一个不懂 Markdown 的人, 他们也没有可以预览 Markdown 的工具, 那么将 Markdown 转换为其他格式的文档是十分有必要的。

继续往下读, 也许会惊异, 转换文档格式尽然这么简单! 是的, Markdown 搭配上 Pandoc 之后, 文档的转换将十分轻松。下面来看看如何操作。本章仅仅介绍如何转换为 docx 文档。

首先需要下载 Windows 版的安装包 Pandoc, 并按照提示进行安装。如果想要

更好的支持 latex 数学公式，还需要安装 Tex Live。安装完 Pandoc 之后，可用命令 pandoc -v 查看 Pandoc 的版本。

如何需要将 markdown 出为 word，那么，可以在文章的开头填入如下内容：

```

```

title: 构建属于自己的项目  
author: xinetzone  
date: 2019/10/17  
output:  
word\_document:  
highlight: "tango"

```

```

其中 highlight 用于设置代码的高亮的主题。上面的设置便可以输出一个十分美观的 word 版本的文档。如图 2.11 所示，是对比代码高亮与不高亮的区别。

第 2 步：将远端的项目克隆到本地。首先，按照下图操作获取项目的远程仓库地址 (<https://github.com/xinetzone/cv-actions.git>) :

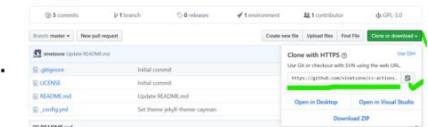


图 4 获取远程仓库地址。

第 3 步：在本地电脑端打开 [vscode](#) 并在终端输入：  
\$ git clone https://github.com/xinetzone/cv-actions.git  
\$ cd cv-actions/-

第 2 步：将远端的项目克隆到本地。首先，按照下图操作获取项目的远程仓库地址 (<https://github.com/xinetzone/cv-actions.git>) :



图 4 获取远程仓库地址。

第 3 步：在本地电脑端打开 [vscode](#) 并在终端输入：  
\$ git clone https://github.com/xinetzone/cv-actions.git  
\$ cd cv-actions/-

图 2.11 代码高亮对比

图 2.11 的左边是代码没有高亮的，而右边是代码高亮的。两者的优劣一眼便可看出。

为了获取更好的观感体验您可以设置高亮主题为 zenburn，效果图见图 2.12。好了，效果图有了那么具体该怎么转换呢？看看图 2.13，按照图示进行类似的操作即可。

第 2 步：将远端的项目克隆到本地。首先，按照下图操作获取项目的远程仓库地址 (<https://github.com/xinetzone/cv-actions.git>) :

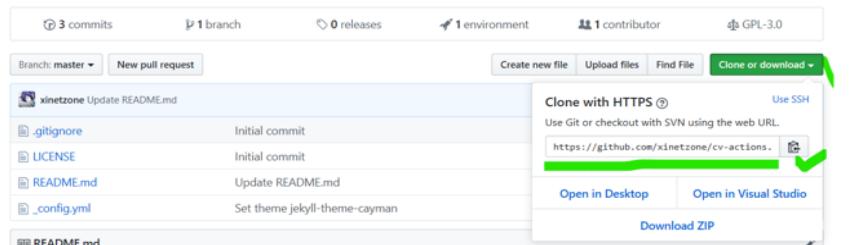


图 4 获取远程仓库地址。

第 3 步：在本地电脑端打开 [vscode](#) 并在终端输入：

```
$ git clone https://github.com/xinetzone/cv-actions.git
$ cd cv-actions/-
```

图 2.12 高亮主题设置为 zenburn

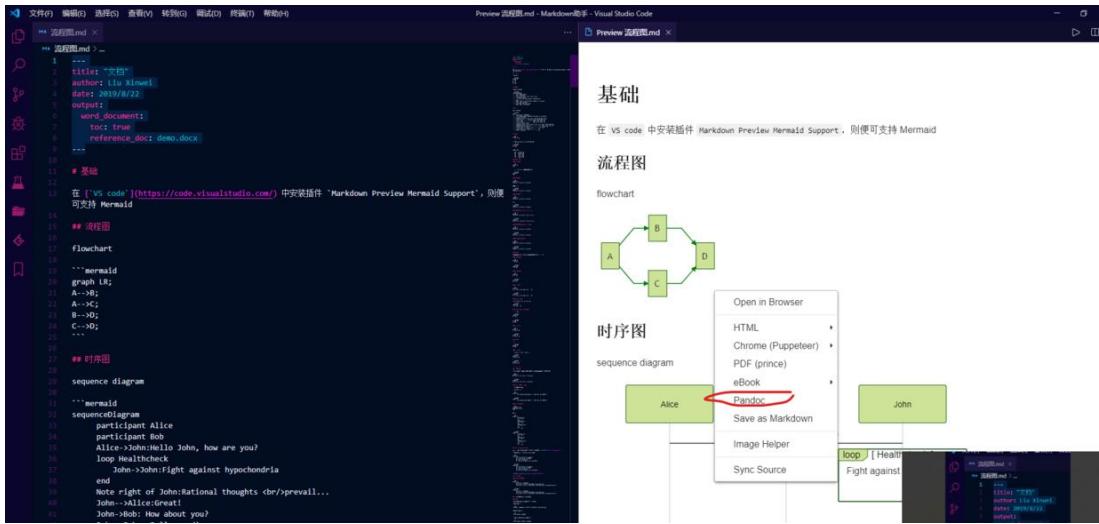


图 2.13 pandoc 转换为 docx 的示意图

### 2.6.1 依据给定的模板输出 word 文档

有时，我们想要依据给定的模板 mystyles.docx 来输出 word 文档，可以这样设置：

```

title: 构建属于自己的项目
author: xinetzone
date: 2019/10/17
output:
 word_document:
 highlight: "tango"
 reference_doc: mystyles.docx

```

其中 reference\_doc 参数指定了 docx 模板。

### 2.6.2 设置多文件生成 word 的模板

如果想要在同一个文件夹下的 markdown 文档以相同的模板生成 word 文档，可以在该目录下设置文件 \_output.yaml 并输入模板配置，比如：

```
output:
 word_document:
 highlight: "tango"
```

这样，不必在 markdown 中再次设置上述参数，即可达到设置 \_output.yaml 中参数的效果。

### 2.6.3 设置自动目录

如果需要设置自动目录，可以这样：

```

title: 构建属于自己的项目
author: xinetzone
date: 2019/10/17
output:
 word_document:
 highlight: zenburn
 reference_docx: mystyles.docx
 toc: true
 toc_depth: 2

```

#### 2.6.4 自动导出为 word

如果不想每次都在预览 markdown 的时候，手动生成 word，可以设置 `export_on_save` 自动生成 word：

```

output: word_document
export_on_save:
 pandoc: true

```

### 2.7 导入外部文件

有时有这样一种需求，需要将某个文件的内容导入到另一个文件之中，省去复制粘贴的操作。可以使用如下语法：`@import "你的文件"` 或者 `<!-- @import "your_file" -->`

导入的外部文件支持如下的类型：

- `.jpeg(.jpg), .gif, .png, .apng, .svg, .bmp` 文件将会直接被当作 markdown 图片被引用。
- `.csv` 文件将会被转换成 markdown 表格。
- `.mermaid` 将会被 mermaid 渲染。
- `.dot` 文件将会被 viz.js (graphviz) 渲染。
- `.plantuml(.puml)` 文件将会被 PlantUML 渲染。
- `.html` 将会直接被引入。
- `.js` 将会被引用为 `<script src="你的 js 文件"></script>`。
- `.less` 和 `.css` 将会被引用为 style。目前 less 只支持本地文件。`.css` 文件将会被引用为。
- `.pdf` 文件将会被 pdf2svg 转换为 svg 然后被引用。
- markdown 将会被分析处理然后被引用。
- 其他所有的文件都将被视为代码块。

导入图片的示例：

```
@import "test.png" {width="300px" height="200px" title="图片的标题" alt="我的 alt"}
```

引用文件作为 Code Chunk（下一章会介绍）的示例：

```
@import "test.py" {cmd="python3", class="line-numbers"}
```

将外部文件作为代码块：

```
@import "test.json" {as="vega-lite"}
```

当然可以插入 Markdown，这样很有意思。除了把多个小文档合并成一个大文档这种无脑的用法之外，我觉得最有潜力的用法是——非线性写作啊！

如果把一些小文件当作卡片的话，善用导入文件功能就可以实现文章分块化管理。

## 2.8 其他有趣的功能

vscode 的 MPE 插件还提供了许多有趣的功能。下面挑选几个实用的让大家研究。

### 2.8.1 导入目录

在 vscode 中，只需要在需要生成目录的地方放入 [TOC] 便会自动生成目录。对于[TOC]不生效的情况，可以使用<!-- @import "[TOC]" {cmd="toc" depthFrom=2 depthTo=3 orderedList=false} --> 代替其自动生成目录。

### 2.8.2 使用表情符号与 Front Icon

参考：GitHub Help：Emoji（<https://help.github.com/cn/github/writing-on-github/basic-writing-and-formatting-syntax#content-attachments>）。

通过键入:EMOJICODE:可在写作中添加表情符号，效果如图 2.14 所示。

```
@octocat :+1: 这个 PR 看起来很棒 - 可以合并了! :shipit:
```

 This PR looks great - it's ready to merge! 

图 2.14 渲染的表情符号

键入:将显示建议的表情符号列表。列表将在你键入时进行过滤，因此一旦找到所需表情符号，按 Tab 或 Enter 键以填写选中的结果。

有关可用表情符号和代码的完整列表，请查看 [emoji-cheat-sheet.com](http://emoji-cheat-sheet.com) ([emoji-cheat-sheet.com](http://emoji-cheat-sheet.com))。您也可以参考 Font Awesome ([https://fontawesome.com/icons?d=gallery](http://fontawesome.com/icons?d=gallery)) 设置 Front Icon。

### 2.8.3 画图

MPE 提供了许多画图的工具，本节仅展示 Mermaid 的画图效果，如图 2.15 所示。更多与 Mermaid 相关的使用语法，可以[我的博文 Mermaid 学习 \(Mermaid](#)

学习)。

更多与画图相关的内容可以参阅 MPE 的图像部分。

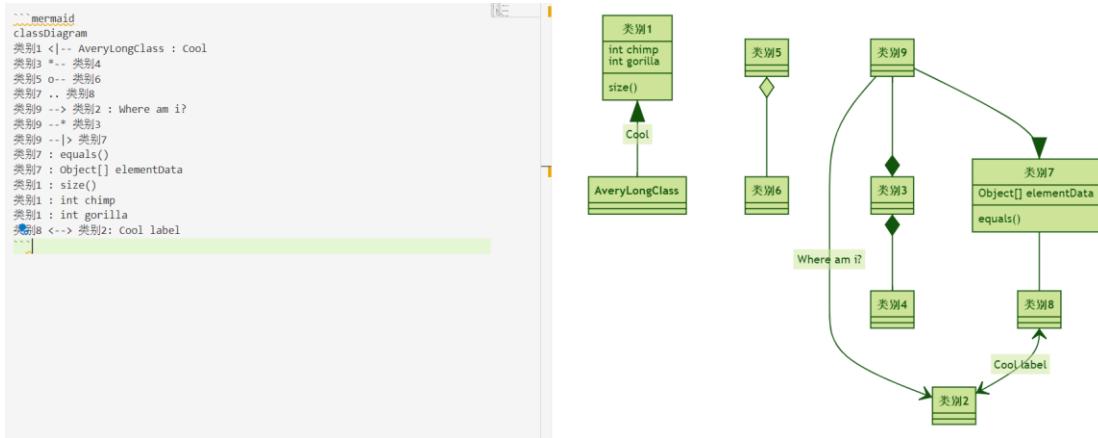


图 2.15 mermaid 画类图

#### 2.8.4 任务列表

可以使用`- [ ]`表示未完成的任务，而`- [x]`表示已经完成的任务，如图 2.16 所示。

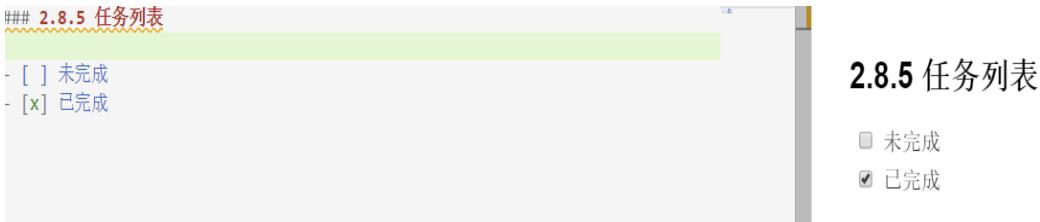


图 2.16 任务列表预览

#### 2.9 本章小结

本章先介绍了 vscode 的一些基本用法，然后着重介绍 MPE 插件的使用。虽然，本章介绍的重点是 Markdown，而没有直接涉及到计算机视觉相关内容，但是，本章也是很重要的，它为我们提供了一份详尽的用于编写代码的说明文档的工具。

## 第 3 章 Python 的基础入门

在前两章已经介绍了计算机视觉的环境搭建以及代码的文档编辑器 vscode。从本章开始我

们将利用 Python 逐步揭开计算机视觉的神秘面纱。因为本书是以 Python 作为编程语言的，所以我们需要先了解 Python，之后再利用 Python 进行计算机视觉的操作。前面的两章可以看作是进入计算机视觉世界的准备工作，从本章开始才真正进入计算机视觉的世界。本章也可以看作是一个先导章节，主要介绍 Python 的基本数据类型。

注意：本章介绍的 Python 是 Python3.7 版本的语法，本书不是专业介绍 Python 的，仅仅展示一些在计算机视觉中可能会用到的知识点。

## 3.1 编写和运行代码

名词解释：

- 程序：使用精确的形式和语义，让计算机实现某项特定功能的软件。
- 终端：接收用户输入的命令（有固定的语法的语句），然后让计算机做出相应操作的一个平台。vscode 可以直接打开终端。

下面以一个实例作为 Python 的开篇，您不需要想该例子的具体细节，只需要了解 Python 的大体运行方式即可。设计一个计算器用于计算商品的价格。即 总价 = 单价 × 数量。用 Python 可以这样设计：

首先，创建一个名为商品价格计算.py 的文件并写入如下内容：

```
商品价格计算.py
这个程序可以用来计算商品的总价。
by: xinetzone
def 商品总价():
 单价, 数量 = eval(input("请输入逗号隔开您想要计算的商品的单价与数量: \n"))
 return 单价 * 数量
调用函数
商品总价()
```

接着，您需要使用终端打开商品价格计算.py，具体操作见如图 3.1 所示。



图 3.1 使用终端打开 商品价格计算.py

最后，在终端输入：python 商品价格计算.py，即可计算商品价格，如图 3.2 所示。

```
xz@Kinet MINGW64 /e/cvsome/datasetsome/docs/draft (master)
$ python 商品价格计算.py
请输入逗号隔开您想要计算的商品的单价与数量:
30,10
商品总价是: 300
xz@Kinet MINGW64 /e/cvsome/datasetsome/docs/draft (master)
$ python 商品价格计算.py
请输入逗号隔开您想要计算的商品的单价与数量:
27,86
商品总价是: 2322
xz@Kinet MINGW64 /e/cvsome/datasetsome/docs/draft (master)
$
```

图 3.2 运行程序

从图 3.2 可以看出，只需要输入 python 商品价格计算.py 命令，那么立马终端便可以计算商品的总价了。这里是直接在 vscode 启动终端来运行程序的一种方式（一般被称为命令行形式）。下面介绍一个更加强大的方式使用 Jupyter Notebook 运行程序。

首先，需要打开 PowerShell（也是终端的一种），如图 3.3 所示。

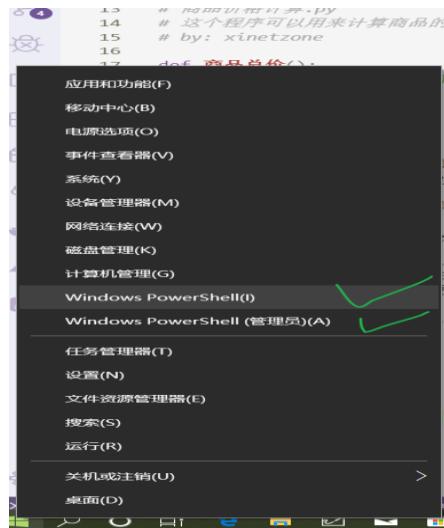


图 3.3 打开 Windows 系统的 PowerShell

然后，您在终端键入 D: 将 PowerShell 切换到磁盘 D 之下，最后输入命令 jupyter notebook 便可以启动 Jupyter Notebook 程序，如图 3.4 所示。

```
Windows PowerShell
版权所有 (C) Microsoft Corporation。保留所有权利。

尝试新的跨平台 PowerShell https://aka.ms/pscore6

PS C:\Users\xz> D:
PS D:>\ jupyter notebook
[I: 16:29:53.620 NotebookApp] The port 8888 is already in use, trying another port.
[I: 16:29:53.621 NotebookApp] The port 8889 is already in use, trying another port.
[I: 16:29:53.764 NotebookApp] [jupyter_nbextensions_configurator] enabled 0.4.1
[I: 16:29:53.852 NotebookApp] JupyterLab extension loaded from C:\ProgramData\Anaconda3\lib\site-packages\jupyterlab
[I: 16:29:53.852 NotebookApp] JupyterLab application directory is C:\ProgramData\Anaconda3\share\jupyter\lab
[I: 16:29:53.855 NotebookApp] Serving notebooks from local directory: D:\
[I: 16:29:53.855 NotebookApp] The Jupyter Notebook is running at:
[I: 16:29:53.856 NotebookApp] http://localhost:8890/
[I: 16:29:53.856 NotebookApp] Use Control-C to stop this server and shut down all kernels (twice to skip confirmation).
```

图 3.4 打开 Jupyter Notebook

在 Python 中文件 python 商品价格计算.py 一般被称为模块，一个可以反复使用的程序文件。打开的 Jupyter Notebook 的界面内可以像图 3.5 那样操作。

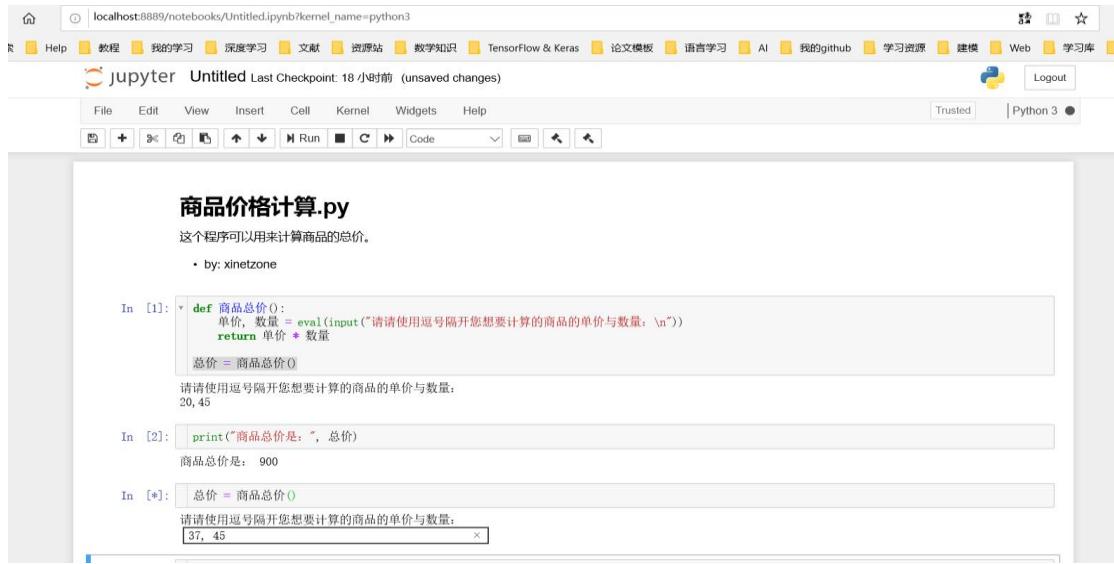


图 3.5 使用 Jupyter Notebook 运行程序

从图 3.5 可以看出 Jupyter Notebook 比直接使用 vscode 运行代码更友好，且支持文本编辑，因而，Jupyter Notebook 更适合做笔记，学习使用。至此，应该对 Python 有了一定的认识。下面将详细阐述 Python 的形式与语义。

## 3.2 Python 基础知识

上面的例子中我们使用“商品价格计算”，“商品总价”，“单价”，“数量”这些名称来表示我们想要做的功能。这种操作在 Python 之中是很常见的。在 Python 之中一般称其为标识符。

### 3.2.1 标识符：我们的描述对象的名称

在 Python 中对标识符有一些约定规则：每个字符必须以字母或下划线（即 \_ 字符）开头，后可跟字母、数字或者下划线的任意序列。这里的字母可以是英文字母，也可以是汉字，日文，韩文等其他语言的字符，比如下面的代码（程序有时也被称为代码）在 Python 中也是合法的：

```
め 3 = 5
かんとう er = 7
print(め 3, かんとう er)
```

不过，人们通常还是习惯使用英文字母，故而，本书也一样使用英文字母。如果您想要使用您喜欢的字符，也是可以的。但是 Python 中一些约定的符号一般都是英文的，并且它们也是 Python 本身的一部分，不能作为普通标识符，一般被称为关键字。本章已经出现的 Python 的关键字有 `def`、`print`、`return`，在之后的章节您会见到其他的关键字的，本章便不做展开了。

注意：

- 关于 Python 的标识符，您还需要知道的是英文字母是区分大小写的。
- 为了统一本书的符号，本书之后的章节的标识符的字母仅使用英文字母。

- 标识符不能使用数字作为开头。

### 3.2.2 赋值语句

在 Python 中标识符标记了我们需要描述的对象的名称，类似于我们为各种比较复杂的符号或者现实生活中的一个物体的抽象起名字，为其赋能。在 3.1 的例子中我们见到了模块的标识符，本小节将介绍如何操作更多的已经创建好的实体（有了一定语义的对象）的标识符。

在 Python 中已经定义了一些有实际语义的对象，比如数字（比如，1, 2.5 等），字符串（用于描述文字的对象，比如“Tom”）等。如果这些对象很复杂，比如 12313334212323146645846346396，您不可能每次都去输入这么长一串数字吧？为了简化，我们可以给这个数字起一个“名字”（即标识符），即：

```
a = 12313334212323146645846346396
```

这样，您只需要记住和使用 a 便可。因为 a 就是数字 12313334212323146645846346396 的代表。

更加正规的说法是 a 被称为变量，而 = 被称为赋值，= 之后的数字被称为值。更一般的形式是 <变量> = <表达式>（表示将表达式赋值给变量）。其中 <> 表示由 Python 代码的其他片段填充的槽。表达式即产生或者计算 Python 对象的程序，比如：1+7, 4\*3, 7 / 5, "a" + "v", a \* 2 等。当然，数字和字符串也属于表达式。

关于赋值语句还有一些强大的功用，在之后的章节在详细介绍。

### 3.2.3 输入与输出

在 Python 中是通过两个函数来操控输入（即您通过键盘等键入的信息）与输出（打印出来信息）的。下面给出它们的具体形式：

输入：<变量> = input(<用于提示用户的字符串表达式>)

输出：print(<表达式>, <表达式>, ..., <表达式>)

关于输入与输出的例子，可以参考 3.1 节的示例。

## 3.3 数据类型

考虑到 Python 2 已经慢慢地淡出人们的视野，故而仅仅考虑 Python 3 即可（之后的说明中不再明确指出是 Python 3，出现的 Python 一律视为 Python 3）。Python 中有六个标准的数据类型：Number（数字）、String（字符串）、List（列表）、Tuple（元组）、Sets（集合）、Dictionary（字典）（注意：本书中涉及的 Python 是 Python 3.6.X~3.7.X 版，暂不考虑 Python3.8）。

在介绍 Python 的数据类型之前，先了解变量。Python 中的变量不需要声明。每个变量在使用前都必须赋值，变量赋值以后该变量才会被创建（换言之，Python 的变量是动态的，可以使用 del 语句删除一些对象引用）。变量是通过 = 进行创建的，比如：

```
a = 1
```

= 运算符左边是一个变量名，右边是存储在变量中的值（该值是有类型的）。

再次回到数据类型的话题上来，在 Python 中，变量就是变量，它没有类型，我们所说的“类型”是变量所指代的内存中对象的类型。等号 (=) 用来给变量赋值。下面将逐一展开 Python 3 的数据类型。

### 3.3.1 Number 数字

Python 3 支持 int、float、bool、complex（复数）。需要注意的是在 Python 3 里，只有一种整数类型 int，表示为长整型，没有 Python 2 中的 Long（不包括 Python 3.8 及其以后的版本）。如果需要查看数据类型，可以通过内置的 type() 函数来查询变量所指的对象类型。

```
a, b, c, d = 204, 5.534, True, 43+3j
print(type(a), type(b), type(c), type(d))
```

显示输出：

```
<class 'int'> <class 'float'> <class 'bool'> <class 'complex'>
```

### 3.3.2 String 字符串

Python 中的字符串用单引号(')或双引号(")括起来，同时使用反斜杠(\)转义特殊字符。

在 Python 3.X 中，有 4 种字符串类型：

- str：用于 Unicode 文本 (ASCII 或其他更宽的); (ASCII 看作 Unicode 的一种简单类型)
- bytes：用于二进制数据 (包括编码的文本);
- bytearray：是一种可变的 bytes 类型。
- raw 字符串：原始字符串，保留字符串的原始格式，包括转义字符。

在字符串的处理中有两个重要的概念：

- 编码：是根据一个想要的编码名称，把一个字符串翻译为原始字节形式。
- 解码：是根据其编码名称，把一个原始字节串翻译为字符串形式的过程。

举个例子：

```
S = 'ni'
S.encode('ascii'), S.encode('latin1'), S.encode('utf8')
```

输出结果：

```
(b'n\u00f1', b'n\u00f1', b'n\u00f1')
```

更多详细内容，可以查阅我的博客：

<https://www.cnblogs.com/q735613050/p/7341004.html>

### 3.3.3 List 列表

List（列表）是 Python 中使用最频繁的数据类型。列表中元素的类型可以不相同，它支持数字，字符串甚至可以包含列表（即所谓的列表嵌套）。列表是写在方括号([])之间并用逗号分隔开的元素列表。

### 3.3.4 Tuple 元组

元组（tuple）与列表类似，不同之处在于元组的元素不能修改。元组写在小括号(())里，元素之间用逗号隔开。一般来说，函数的返回值一般为一个，而函数返回多个值的时候，是以元组的方式返回的。python 中的函数还可以接收可变长参数，以 “\*” 开头的参数名，会将所有的参数收集到一个元组上。

### 3.3.5 Set 集合

集合（set）是一个无序不重复元素的序列。基本功能是进行成员关系测试和删除重复元素。可以使用大括号 {} 或者 set() 函数创建集合，但是需要注意：创建一个空集合，必须用 set() 而不是 {}，因为 {} 是用来创建一个空字典的。

### 3.3.6 Dictionary 字典

字典（dictionary）是 Python 中另一个非常有用的内置数据类型。两者之间的区别在于：字典当中的元素是通过键来存取的，而不是通过偏移存取。字典是一种映射类型，字典用 “{}” 标识，它是一个键(key): 值(value)对组成的集合。键(key)必须使用不可变类型。在同一个字典中，键(key)必须是唯一的。

上述介绍的 Python 数据类型在一定的条件下是可以相互转换的。

## 3.4 Python 数据类型转换

如表 3.1 所示的几个内置函数可以执行数据类型之间的转换。这些函数返回一个新的对象，表示转换的值。

表 3.1 常见的 Python 数据类型转换函数

函数	描述
int(x [,base])	将 x 转换为一个整数
float(x)	将 x 转换到一个浮点数
complex(real [,imag])	创建一个复数
str(x)	将对象 x 转换为字符串
repr(x)	将对象 x 转换为表达式字符串
eval(str)	用来计算在字符串中的有效 Python 表达式，并返回一个对象
tuple(s)	将序列 s 转换为一个元组
list(s)	将序列 s 转换为一个列表

set(s)	转换为可变集合
dict(d)	创建一个字典。D 必须是一个序列 (key,value)元组。
Frozenset(s)	转换为不可变集合
chr(x)	将一个整数转换为一个字符
unichr(x)	将一个整数转换为 Unicode 字符
ord(x)	将一个字符转换为它的整数值
hex(x)	将一个整数转换为一个十六进制字符串
oct(x)	将一个整数转换为一个八进制字符串

## 3.5 运算符

仅仅有了数据类型还不能满足于计算机视觉的任务，还需要可以进行操作的运算符，接下来进行详细学习。

### 3.5.1 算术运算符 & 赋值算术运算符

算术运算符 & 赋值算术运算符，如表 3.2 所示。

表 3.2 算术运算符 & 赋值算术运算符

算术运算符	描述	赋值算术运算符	描述	示例
+	加(两个对象相加，或者序列的拼接)	+=	加法赋值运算符	c += a 等效于 c = c + a
-	减	-=	减法赋值运算符	c -= a 等效于 c = c - a
*	乘(两个数相乘或是返回一个被重复若干次的序列)	*=	乘法赋值运算符	c *= a 等效于 c = c * a
/	除(两个数相除)	/=	除法赋值运算符	c /= a 等效于 c = c / a
%	取模(返回除法的余数)	%=	取模赋值运算符	c %= a 等效于 c = c % a
**	幂	**=	幂赋值运算符	c **= a 等效于 c = c ** a
//	整除	//=	取整除赋值运算符	c //= a 等效于 c = c // a
		=	简单的赋值运算符	c = a + b 将 a + b 的运算结果赋值为c

### 3.5.2 位运算符

按位运算符是把数字看作二进制来进行计算的（此类运算符一般很少使用），如表 3.3 所示。

表 3.3 位运算符

运算符	描述
&	按位与运算符：参与运算的两个值，如果两个相应位都为 1，则该位的结果为 1，否则为 0
	按位或运算符：只要对应的二个二进位有一个为 1 时，结果位就为 1。
^	按位异或运算符：当两对应的二进位相异时，结果为 1
~	按位取反运算符：对数据的每个二进制位取反，即把 1 变为 0，把 0 变为 1。 $\sim x$ 类似于 $x-1$
<<	左移动运算符：运算数的各二进位全部左移若干位，由“<<”右边的数指定移动的位数，高位丢弃，低位补 0。
>>	右移动运算符：把“>>”左边的运算数的各二进位全部右移若干位，“>>”右边的数指定移动的位数

下面简单举几个例子：

```
a = 60 # 60 = 0011 1100
b = 13 # 13 = 0000 1101
c = 0
& 运算
c = a & b # 12 = 0000 1100
print ("a & b 的值为: ", c)
| 运算
c = a | b # 61 = 0011 1101
print ("a | b 的值为: ", c)
^ 运算
c = a ^ b # 49 = 0011 0001
print ("a ^ b 的值为: ", c)
~ 运算
c = ~a # -61 = 1100 0011
print ("~a 的值为: ", c)
<< 运算
c = a << 2 # 240 = 1111 0000
print ("a << 2 的值为: ", c)
>> 运算
c = a >> 2 # 15 = 0000 1111
print ("a >> 2 的值为: ", c)
```

输出结果：

```
a & b 的值为: 12
a | b 的值为: 61
a ^ b 的值为: 49
~a 的值为: -61
a << 2 的值为: 240
```

```
a >> 2 的值为: 15
```

### 3.5.3 比较运算符 & 逻辑运算符

在 Python 中比较两个变量的值一般有比较运算符提供支持: ==、<=、<、>=、>、!=。比较运算符返回的是布尔值 (True/False)。而逻辑运算符需要注意 and 与 or 的惰性求值, 如表 3.4 所示。

表 3.4 Python 逻辑判断运算符

逻辑运算符	逻辑表达式	描述
and	x and y	布尔“与”(如果 x 为 False, x and y 返回 False, 否则它返回 y 的计算值。)
or	x or y	布尔“或”(如果 x 是 True, 它返回 x 的值, 否则它返回 y 的计算值。)
not	not x	布尔“非”(如果 x 为 True, 返回 False。如果 x 为 False, 它返回 True。)

### 3.5.4 成员运算符 & 身份运算符

成员运算符用于搜索成员 (或者元素) 是否存在于某一个序列或集合等结构之中, 如表 3.5 所示。

表 3.5 Python 成员运算符

成员运算符	描述
in	如果在指定的序列中找到值返回 True, 否则返回 False。
not in	如果在指定的序列中没有找到值返回 True, 否则返回 False。

身份运算符用于比较两个对象的存储单元, 如表 3.6 所示。

表 3.6 Python 身份运算符

身份运算符	描述	实例
is	判断两个标识符是不是引用自一个对象	x is y, 类似 id(x) == id(y), 如果引用的是同一个对象则返回 True, 否则返回 False
is not	判断两个标识符是不是引用自不同对象	x is not y, 类似 id(a) != id(b)。如果引用的不是同一个对象则返回结果 True, 否则返回 False。

注意: id()函数用于获取对象内存地址。

## 3.6 Python 运算符优先级

运算符也需要有一个运算的先后，如表 3.7 所示列出了从最高到最低优先级的所有运算符。

表 3.7 Python 运算符优先级排序

运算符	描述
$**$	指数 (最高优先级)
$\sim + -$	按位翻转, 一元加号和减号 (最后两个的方法名为 $+@$ 和 $-@$ )
$* / \% //$	乘, 除, 取模和取整除
$+ -$	加法 减法
$>> <<$	右移, 左移运算符
$\&$	位 ‘AND’ 运算符
$\wedge \mid$	位运算符
$<= <> >=$	比较运算符
$\diamond == !=$	等于运算符
$= \% = /= // = -= += *= ** =$	赋值运算符
$is is not$	身份运算符
$in not in$	成员运算符
$not or and$	逻辑运算符

## 3.7 从容器角度看待 list, tuple, dict, set 等结构

我们可以将 Python 中的 list, tuple, dict, set 以图 3.6 所示的形式进行划分。

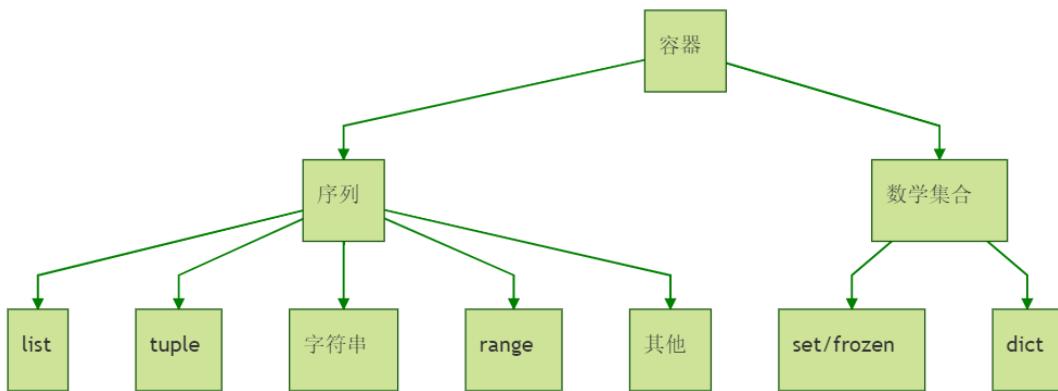


图 3.6 从容器角度看待 list, tuple, dict, set 等结构

容器表示需要处理的对象是一组或者多组而不是单个。如果一个组数据是有序的并且支持切片和索引功能，我们可以将其视为序列：每个元素可以是任何类型(也可以是序列)，每个元素被分配一个序号(从 0 开始) (序号，也叫索引，表示元素的位置)。Python 中可以视为序列

的数据结构有：元组，列表，字符串，buffer 对象，range 对象等。

Python 中的序列可以类比数学中的数列进行理解，而 Python 中的 set/frozen 与 dict 亦是完全可以借助数学中的“集合”进行理解。

### 3.7.1 序列的基本操作：索引与切片

索引使用方法：序列[编号]。

下面以列表为示例进行说明：

```
a = [1,2,3,4,5]
a[0] # 取出索引为 0 的元素
```

输出：

```
1
```

上述的代码中列表 a 等价于数学中的数列  $a = \{a_0, a_1, a_2, a_3, a_4\}$ ，而其索引 a[0]即为  $a_0$ 。

切片使用方法：序列[开始编号:结束编号后一个:步长(默认为 1)]。

示例：

```
a[1:3]
```

输出：

```
[2, 3]
```

这里 a[1:3]指代数列 a 中的  $a_1, a_2$ 。切片支持负索引：

```
a[1:-1]
```

输出：

```
[2, 3, 4]
```

即负索引是从右边开始算起以 -1 作为起点的。切片操作中的步长指代移动的间隔，可以将其视为数列的下标以步长为等差的数列。详细点说，序列  $a[i:j:k]$  表示选取数列  $\{a_0, \dots, a_n\}$  的子列，该子列分别以  $a_i, a_j$  作为起点与终点，以 k 为等差。需要注意的是 Python 中 i, j, k 可以缺省。缺省 i 表示起点为  $a_0$ ，缺省 j 表示终点为  $a_n$ ，缺省 k 表示步长为 1。

举例如下：

```
print(a[1:])
print(a[:-1])
a[:]
```

输出：

```
[2, 3, 4, 5]
[1, 2, 3, 4]
[1, 2, 3, 4, 5]
```

### 3.7.2 序列的拼接

拼接有两种方法：+ 与 \*。其中 + 类似于如下功能：一篮苹果+一篮香蕉=两篮水果。

使用方法：序列+序列。

示例：

```
a=[1,2,3]
b=[2,3,4]
a+b
```

输出：

```
[1, 2, 3, 2, 3, 4]
```

需要注意的是 + 并没有改变原有序列:

```
a
```

输出仍然不变:

```
[1,2,3]
```

\* 类似于: 一个苹果\*n=n 个苹果。

使用方法: 序列\*n(n 表示重复次数)。

示例:

```
a=[1,2,3]
```

```
a*3
```

输出:

```
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

### 3.7.3 序列的其他方法

`in`: 判断某个元素是否存在。

使用方法:

```
a in 序列 A
```

若  $a \in A$ , 则返回 True, 否则返回 False。

`len,max,min` 分别返回序列的长度, 最大值, 最小值。

使用方法:

```
len(序列)
```

```
max(序列)
```

```
min(序列)
```

### 3.7.4 序列之列表

前面几个小结介绍的是序列的通用方法, 下面着重介绍一下列表。

#### 1.列表的修改

我们先创建一个列表 `a`:

```
a=[1,2,3,4]
```

可以直接对列表中某个元素进行修改:

```
a[0]=2
```

```
a
```

输出结果:

```
[2, 2, 3, 4]
```

也可以对列表的某个子列进行修改:

```
a[1:3]=[5,6]
```

```
a
```

输出结果为:

```
[2, 5, 6, 4]
```

当然, 也可以通过赋值删除列表中的部分元素:

```
a[1:3]=""
```

```
a
```

输出结果为:

```
[2, 4]
```

可以通过赋值扩展列表元素:

```
a[0:0]=[99,88]
```

```
a
```

输出结果为:

```
[99, 88, 2, 4]
```

可以通过 list 函数直接将字符串转换为列表:

```
a[1:3]=list('hello')
```

```
a
```

输出结果为:

```
[99, 'h', 'e', 'l', 'l', 'o', 4]
```

删除列表中的元素，也可以使用 del 语句:

```
del a[0]
```

```
a
```

输出结果为:

```
['h', 'e', 'l', 'l', 'o', 4]
```

删除列表中多个元素:

```
del a[0:3]
```

```
a
```

输出结果为:

```
['l', 'o', 4]
```

## 2.列表的 append 方法

使用调用对象的方法: 对象.方法(参数)，可以在列表的末尾追加新的元素:

```
a=[1,2,3]
```

```
a.append(4)
```

```
a
```

输出结果为:

```
[1, 2, 3, 4]
```

亦可以追加一个列表:

```
a.append([1,5,'fr'])
```

```
a
```

输出结果为:

```
[1, 2, 3, 4, [1, 5, 'fr']]
```

## 3.列表的 count 方法

查看某个元素在列表中出现的次数:

```
a=[1,2,3,4,1,2]
```

```
a.count(1)
```

查看 1 在 a 列表中出现的次数:

```
2
```

## 4.列表的 extend 方法

使用其他列表拓展原有列表，其他列表的元素被添加到原有列表的末尾:

```
a=[1,2,3]
```

```
a.extend([4,5,6])
```

```
a
```

输出结果为:

```
[1, 2, 3, 4, 5, 6]
```

## 5.列表的 index 方法

返回某个元素的索引，如若此元素不存在，则会引发异常。

```
a=[1,2,3,45,5,45,7,84]
```

```
a.index(45)
```

输出结果为:

```
3
```

## 6.列表的 insert 方法

在序列的某个位置插入一个元素:

```
a=[1,2,3,4]
```

```
a.insert(2,'hello')
```

```
a
```

输出结果为:

```
[1, 2, 'hello', 3, 4]
```

插入超出边界位置的元素不会报错:

```
20 号位置不存在，直接插到列表末尾
```

```
a.insert(20,'world')
```

```
a
```

输出结果符合预期:

```
[1, 2, 'hello', 3, 4, 'world']
```

## 7.列表的 pop 方法

移除列表某个位置的元素并返回该元素，如若没有指定要移除元素的位置，则默认移除末项。

```
a=[1,2,3,4]
```

```
a.pop()
```

移除末尾的元素，并返回:

```
4
```

现在查看 a:

```
a
```

输出结果为:

```
[1,2,3]
```

a 确实移除了末尾元素，下面移除首位元素:

```
a.pop(0)
```

输出结果为:

```
1
```

再次查看 a:

```
a
```

输出结果为:

```
[2,3]
```

## 8.列表的 remove 方法

移除序列中第一个与参数匹配的元素:

```
a=[1,2,88,3,4,88,2]
```

```
a.remove(88)
```

```
a
```

输出结果为:

```
[1, 2, 3, 4, 88, 2]
```

## 9.列表的 reverse 方法

将列表改为倒序:

```
a=[1,2,3,4,2,5,3]
a.reverse()
a
```

输出结果为:

```
[3, 5, 2, 4, 3, 2, 1]
```

## 10.列表的 sort 方法

默认为升序:

```
a=[4,6,2,1,7,9,6]
a.sort()
a
```

输出结果为:

```
[1, 2, 4, 6, 6, 7, 9]
```

通过参数修改排序:

参数 key 用来为每个元素提取比较值(默认值为 None); reverse 为 True 时是反序(默认值为 False)。

```
names=['Judy','Perter','Perkins']
names.sort(key=len)
names
```

输出结果为:

```
['Judy', 'Perter', 'Perkins']
```

## 3.7.5 序列之元组

元组与字符串是序列中的不可修改的，关于字符串有点复杂，本章便不作展开，仅仅考虑元组。“元组的不可修改”是指元组的每个元素的指向永远不变：

```
b=('python','20150101',['a','b'])
b
```

输出结果为:

```
('python', '20150101', ['a', 'b'])
```

但是元组的元素如果是可变的列表，则可以修改：

```
b[2][0]='c'
b
```

赋值的输出结果为:

```
('python', '20150101', ['c', 'b'])
```

当然也可以添加元素：

```
b[2].append('d')
b
```

输出结果为:

```
('python', '20150101', ['c', 'b', 'd'])
```

### 3.7.6 容器之字典 (dict)

从数学的角度来看，dict 便是映射，其形式为：

```
{key:value,...}
```

其中 key 是唯一的，具有排它性，只能是具有不可变的字符串元组之流，而 value 则没有限制，可以是任意 Python 对象。

#### 1.字典的创建

以例子说明：

```
d={'wang':'111','U':'123'}
d
```

输出结果如图 3.7 所示，代码如下：

```
{'U': '123', 'wang': '111'}
```

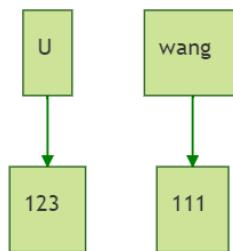


图 3.7 字典比作映射

可以直接通过 key 来获取 value，有点类似与“查字典”。当然，也可以通过 dict 函数依据元组对创建字典：

```
d1=dict([('A',1),('B',2)])
d1
```

输出结果为：

```
{'A': 1, 'B': 2}
```

也可以通过 dict 以传关键字参数的形式创建字典：

```
d2=dict(A=1,B=2)
d2
```

输出结果为：

```
{'A': 1, 'B': 2}
```

#### 2.通过键查找或修改字典

查找字典的示例：

```
d={'wang':'111','U':'123'}
d['U']
```

输出结果为：

```
'123'
```

修改字典：

```
d['wang']=111
d
```

输出结果为：

```
{'U': '123', 'wang': 111}
```

dict 也有 del, len, in 的操作，这里就不展开了，下面简要介绍 dict 的一些特殊方法。

### 3.字典之 clear()

清除字典中所有的项:

```
d={'wang':'111','U':'123'}
d.clear()
d
```

输出结果为:

```
{}
```

### 4.字典之 copy()

浅复制，得到一个键的指向完全相同原字典的副本。

```
d={'wang':'111','U':[1,2,3,4]}
d1=d.copy()
d1
```

输出结果为:

```
{'U': [1, 2, 3, 4], 'wang': '111'}
```

原地修改原字典 d,相应的 d1 也会被修改,反之亦然。

```
d1['U'].append('lr')
d1
```

输出结果为:

```
{'U': [1, 2, 3, 4, 'lr'], 'wang': '111'}
```

查看 d:

```
d
```

输出结果为:

```
{'U': [1, 2, 3, 4, 'lr'], 'wang': '111'}
```

可以看出 d 也被修改了，但如果使用 deepcopy() 函数则可以避免上述情况发生。

```
d={'wang':'111','U':[1,2,3,4]}
from copy import deepcopy
d1=deepcopy(d)
d1
```

输出的结果为:

```
{'U': [1, 2, 3, 4], 'wang': '111'}
```

此时修改 d1:

```
d1['U'].append('lr')
d1
```

输出结果为:

```
{'U': [1, 2, 3, 4, 'lr'], 'wang': '111'}
```

查看 d:

```
d
```

输出结果为:

```
{'U': [1, 2, 3, 4], 'wang': '111'}
```

可以看出 d 没有随 d1 的变化而变化。

### 5.字典之 get 方法，查找元素

如若元素不存在，可以自定义返回的内容（默认为 None）：

```
d={}
d.get('name')
```

此时便返回 None，而下面赋值语句则会添加字典元素：

```
d['name']='Tom'
```

```
d
```

输出结果为:

```
{'name': 'Tom'}
```

再使用 get 方法查看:

```
d.get('name')
```

输出结果便不是空（None）：

```
'Tom'
```

也可以以如下方式赋值:

```
d.get('phone','Unknown')
```

输出结果为:

```
'Unknown'
```

## 6.字典之 setdefault 方法, 查找元素

与 get 方法不同的是, 当键不存在时, 自定义的值和该键会组成一个新项被加入字典。

```
d
```

输出结果为:

```
{'name': 'Tom'}
```

使用 setdefault 方法:

```
d.setdefault('phone','119')
```

输出结果为:

```
'119'
```

此时 d 也被改变了:

```
d
```

```
{'name': 'Tom', 'phone': '119'}
```

## 7.字典之 items(),keys(),values()

均以列表的形式返回 a set-like object, 其中的元素分别为“项”, “键”, “值”

```
d={'wang':'111','U':[1,2,3,4]}
```

```
d.items()
```

返回项:

```
dict_items([('wang', '111'), ('U', [1, 2, 3, 4])])
```

返回关键字:

```
d.keys()
```

输出结果为:

```
dict_keys(['wang', 'U'])
```

返回值:

```
d.values()
```

输出结果为:

```
dict_values(['111', [1, 2, 3, 4]])
```

## 8.字典之 pop(键)

返回键对应的值, 并删除字典中这个键对应的项:

```
d={'wang':'111','U':[1,2,3,4]}
```

```
d.pop('U')
```

输出结果为:

```
[1, 2, 3, 4]
```

此时查看 d:

```
d
```

输出结果为:

```
{'wang': '111'}
```

可以看出 U 所在项已经被删除。

## 9.字典之 popitem()

随机返回字典中的项，并从字典中删除:

```
d={'wang':'111','U':[1,2,3,4]}\n\n\nd.popitem()
```

输出结果为:

```
('U', [1, 2, 3, 4])
```

此时查看 d:

```
d
```

输出结果为:

```
{'wang': '111'}
```

## 10.字典之 update()

使用新字典更新旧字典：新字典中有而旧字典中没有的项会被加入到旧字典中；新字典中有而旧字典中也有的值会被新字典的值所代替。

```
d1={'n':'xx','p':'110'}\n\nd2={'p':'120','a':'A'}\n\nd1.update(d2)\n\nd1
```

输出结果为:

```
{'a': 'A', 'n': 'xx', 'p': '120'}
```

查看 d2:

```
d2
```

d2 没有被改变:

```
{'a': 'A', 'p': '120'}
```

## 3.7.7 集合

集合分为：可变集合 set() 与 不可变集合 frozen()。集合的元素必须是不可变对象（如字符串，元组等），且元素间有互异性（与数学中集合的定义一致）。

```
set('Hello')
```

输出结果为:

```
{'H', 'e', 'l', 'o'}
```

## 1.集合的基本操作

创建集合:

```
s=set(['Python','is','a','magic','language'])\nprint(s)
```

输出结果为:

```
{'a', 'magic', 'Python', 'language', 'is'}
```

add 方法，添加元素:

```
s.add('!')\ns
```

输出结果为:

```
{'!', 'Python', 'a', 'is', 'language', 'magic'}
```

也像 dict 一样支持更新（update）

```
a=set([1,2,3,4])
b=set([3,4,5,6])
a.update(b)
a
```

输出结果为：

```
{1, 2, 3, 4, 5, 6}
```

remove 方法删除集合中元素：

```
s=set('hello')
s
```

输出结果为：

```
{'e', 'h', 'l', 'o'}
```

查看删除后的集合：

```
s.remove('h')
s
```

输出结果为：

```
{'e', 'l', 'o'}
```

注意：删除集合元素，使用 remove 方法，若元素不存在，则会引发错误，而 discard 则不会。

```
s.discard('om')
```

下面再看看集合的特殊操作。

## 2. 集合之等价(==)与不等价(!=)

```
set('Python') == set('python')
```

输出结果为：

```
False
```

而：

```
set('Python') != set('python')
```

输出结果为：

```
True
```

可以看出集合对元素的大小写是敏感的。

## 3. 子集与超集

判断前面一个集合是否是后面一个集合的严格子集，子集，严格超集，超集： $<$ ,  $\leq$ ,  $>$ ,  $\geq$ :

```
set('Hello') < set('HelloWorld')
```

输出结果为：

```
True
```

而：

```
set('Hello') <= set('Hello')
```

输出结果为

```
True
```

## 4. 集合之交并补差

集合的并运算(U):

```
set('Hello') | set('world')
```

输出结果为：

```
{'H', 'd', 'e', 'l', 'o', 'r', 'w'}
```

集合的交( $\cap$ ) 使用 &:

```
set('Hello') & set('world')
```

输出结果为:

```
{'l', 'o'}
```

集合的差(-)运算使用 -:

```
set('Hello') - set('world')
```

输出结果为:

```
{'H', 'e'}
```

集合的对称差使用 ^:

```
set([1,2,3,4])^set([3,4,5,6])
```

输出结果为:

```
{1, 2, 5, 6}
```

## 5.集合的注意事项

如是可变集合(set)与不可变集合 (frozenset) 进行运算，得到的新集合的类型与左操作数相同。对于可变集合(set)可以进行就地修改，操作符为: |=,&=,-=,^=

集合只能包含不可变的(即可散列的)对象类型。

```
a=set('Hello')
```

```
a |= set('Python')
```

```
a
```

输出结果为:

```
{'H', 'P', 'e', 'h', 'l', 'n', 'o', 't', 'y'}
```

就地交运算:

```
a=set('Hello')
```

```
a &= set('Python')
```

```
a
```

输出结果为:

```
{'o'}
```

就地差运算:

```
a=set('Hello')
```

```
a -= set('Python')
```

```
a
```

输出结果为:

```
{'H', 'e', 'l'}
```

就地对称差运算:

```
a=set('Hello')
```

```
a ^= set('Python')
```

```
a
```

输出结果为:

```
{'H', 'P', 'e', 'h', 'l', 'n', 't', 'y'}
```

set 与 frozenset 也可以进行运算:

```
b=set('Hello')|frozenset('Python')
```

```
b
```

输出结果为:

```
{'H', 'P', 'e', 'h', 'l', 'n', 'o', 't', 'y'}
```

但是 set 与 frozenset 的顺序不同，运算的结果也不相同:

```
c=frozenset('Python')|set('Hello')
c
```

输出结果为：

```
frozenset({'H', 'P', 'e', 'h', 'l', 'n', 'o', 't', 'y'})
```

## 3.8 本章小结

本章主要简要的介绍了 Python 的基础知识，为计算机视觉的编程提供了编程基础。同时，本章最后从容器的角度进一步阐述 Python 的列表、元组、字典以及集合等数据结构。

# 第 4 章 Python 的简易进阶教程

在第 3 章了解了 Python 的基础知识，认识了什么是标识符，什么是变量与表达式，同时详细的介绍了 Python 支持的基本数据类型。本章开始我们将接触如何使用 Python 操作循环流程，执行条件判断等一系列高级的操作。

## 4.1 再谈“变量”

本节我们将讨论“变量”的高级操作。Python 的对象一般可以分为可变对象与不可变对象这两类。具体解释如下：

**可变对象：**对象存放的地址的值会原地改变，即销毁原来的值，赋予新的值。有 `list`、`set`、`dict`。

**不可变对象：**对象存放的地址的值不会被改变。有 `tuple`、`str`、`frozenset`、`int`、`float`、`bool`（即 `True` 和 `False`）。

变量可以看作是 Python 对象的标签或引用，又被称为对象引用。在 Python 中可以使用 `id()` 函数查看变量的引用对象的地址，可以使用 `==` 判断两个引用对象的值是否相等，可以使用 `is` 判断两个对象是否是同一个对象。下面看几个例子。

我们仅以数字对象为例：

```
a = 1
print(id(a))
```

输出结果为 140728035942800，即数字对象 1 的地址。下面改变数字对象 1 的引用变量：

```
b = 1
print(id(b))
```

输出的结果依然为 140728035942800，这说明了什么？说明数字对象是不可变的，且变量仅仅是一个“引用”，通俗点说，就是数字 1 被贴了两个标签 `a` 与 `b`。看下个例子：

```
a = 1
```

```
b = 1
print(id(a) == id(b), a is b, a == b)
```

输出了 True True True，更进一步的说明 a 与 b 引用了同一个对象。想要达到上述同样效果，我们也可以这样操作：

```
a = 1
b = a
```

或者：

```
a = b = 1
```

### 4.1.1 交换赋值

我们再看一个比较有意思的变量操作：交换两个变量的赋值。在此之前，需要知道 Python 支持“联合”赋值，即：

```
a = 1
b = 2
```

等价于：

```
a, b = 1, 2
```

因为数字对象是不可变的，所以 a 与 b 的引用是不同的：

```
a, b = 1, 2
print(id(a) == id(b))
```

输出结果为 False 符合我们之前讨论的逻辑。如何交换 a 与 b 的赋值呢？像下面的操作吗？

```
a, b = 1, 2
a = b
b = a
```

这样是不行的。因为数字对象是不可变的，当执行 a=b 的操作时就已经将 b 的值赋值给 a，并且二者牢牢的绑在一起了，再次执行 b=a 将是在同一对象的引用之间进行赋值的，故而没有改变二者的值。

那该怎么办呢？人们往往会想到引入第三个变量，即：

```
a, b = 1, 2
temp = a
a = b
b = temp
```

问题得以解决！但是有点太繁琐了，有没有更便捷的方法？是有的，在 Python 中同时赋值提供了一个更优雅的选择，即：

```
a, b = 1, 2
b, a = a, b
```

因为赋值是同时的，它避免了擦除一个原始值，同时也实现了交换赋值的目的。

### 4.1.2 可变对象的赋值

前面我们已经了解不可变对象的赋值原理，那不可变对象是怎样的呢？先看例子：

```
a = [2, 5, 7]
b = [2, 5, 7]
print(id(a) == id(b), a is b, a == b)
```

输出结果是 False False True，那两个 False 表明列表对象[2, 5, 7]是可变的。因为相同的值（True 说明值相同）的可变对象，却拥有不同的地址。该例子说明了可变对象的“变”，但是可变对象不仅仅在于“变”，它也有不变的一面：

```
a = [2, 5, 7]
print(id(a))
a[0] = 77
print(id(a))
```

输出的结果是：

```
1754482627016
1754482627016
```

这个结果表明，您虽然改变了可变对象的内部赋值，但是可变对象的引用并没有被改变。

这里有一个误区，务必注意：`a = b = [2, 3, 4]` 与 `a, b = [2, 3, 4], [2, 3, 4]` 并不等效。前者表示将[2, 3, 4]赋值给 b，然后再将 b 与 a 进行绑定，换言之，此时 a 与 b 是同一个对象的两个引用。而后者则是两个有相同值的不同对象的引用。

### 4.1.3 增量赋值

在上一章我们了解了赋值算术运算符（或被称为增量运算符）。接下来讨论增量赋值到底是怎么一回事？先看一个增量赋值的例子：

```
a = 1
print(id(a))
a += 1
print(id(a))
```

输出了两个不同的 id：

```
140728035942800
140728035942832
```

再看看普通赋值的例子：

```
a = 1
print(id(a))
a = a + 1
print(id(a))
```

输出结果与上个例子完全一致：

```
140728035942800
140728035942832
```

这两个例子说明对于不可变对象的增量赋值仅仅是缩短了代码的长度而已。接下来看看可变对象的增量赋值：

```
a = [1, 2]
print(id(a))
a += [3]
print(id(a))
```

输出结果为：

```
3018037481928
3018037481928
```

id 为什么没有改变？那么再看看普通赋值又如何：

```
a = [1, 2]
```

```
print(id(a))
a = a + [3]
print(id(a))
```

输出结果为:

```
1403910443464
1403911725000
```

此时，id 竟然又发生改变了？通过这两个例子，我们知道：对于可变对象，普通赋值会创建新的对象，而增量赋值会就地修改原对象。

#### 4.1.4 符号 \* 的妙用

如果想要使用一个变量打包赋值某些特定位置的多个对象，那么需要借用\*。看例子：

```
a, *e, s = 1, 2, 3, 4, 5
print(e)
```

输出结果为:

```
[2, 3, 4]
```

这里变量 e 使用列表的形式打包了多个数据。如果直接打包多个数据，即：

```
a = 1, 2, 3
print(a)
```

输出结果为(1, 2, 3)，即以元组的形式进行打包。有时，我们需要合并两个字典，一般的操作方法是这样的：

```
x = {'a':1, 'b':3}
y = {'c':7}
x.update(y)
```

此操作将 y 合并进了 x。改变了 x，为了不改变 x，我们可以借用\*：

```
x = {'a':1, 'b':3}
y = {'c':7}
z = dict(x,**y)
```

## 4.2 流程控制

前面的章节，已经使用了许多的逻辑判断语句，比如 a is b, a==b 等。它们返回的结果是布尔值（即 True 或 False），用于判断是否满足条件。

#### 4.2.1 真值测试

前面提到的逻辑判断都是单个的，如果想要判断多个条件，可能会用到：and、or、not，它们的逻辑如表 4.1 所示。

表 4.1 and、or、not 使用逻辑

真值判断	结果
------	----

X and Y	从左到右求算操作对象，然后返回第一个为假的操作对象
X or Y	从左到右求算操作对象，然后返回第一个为真的操作对象
not X	返回 X 的否定值

还有一种特殊的判断方式，断言（assert），看例子：

```
num = -1
assert num > 0, 'num should be positive!'
```

该例子用于判断 num 是否大于 0，如果不满足，则触发报错机制，并给出引发的错误信息 'num should be positive!'。

### 4.2.2 if 条件语句

为了给出多种选择，您可以使用 if 条件语句，看下面的例子：

```
year = int(input('请输入年份：'))
if (year % 4 == 0 and year % 100 != 0) or year % 400 == 0:
 print('闰年')
else:
 print('平年')
```

该例子给出了判断是否是“闰年”的选择结构，不同的选择输出不同的结果。这是一个二项选择题，也可以设置为多项选择：if-elif-elif-...-elif-else。

### 4.2.3 循环结构

在 Python 中有两种常用循环结构：for 语句与 while 语句。for 语句可以用于可迭代对象（即可以一个挨着一个取出的数据对象，正式点说就是内部实现了\_\_iter\_\_方法的 Python 对象）的数据的遍历。看个例子：

```
for k in [2, 3, 4]:
 print(k)
```

该例子将会把列表[2, 3, 4] 中的内容逐个取出并打印。可迭代对象有但不限于：range()，列表，字典，集合，元组等。for 语句是一个确定的循环体（循环之前已经知道其循环的次数），而 while 语句则是一个不定循环，看例子：

```
k = 20
while k >= 10: # 直到不满足条件时，终止循环
 print(k)
 k -= 1
```

从这个例子可以看出 while 语句并不能提前直接预知可能的循环次数，需要借助条件判断来终止循环。while 语句是依据条件判断进行的循环体（也被称为条件循环），它可以实现一些有趣的功能，比如：

```
x = range(200, 100, -25)
while x:
 print(x, end=' ')
 x = x[1:]
```

输出结果为：

```
range(200, 100, -25) range(175, 100, -25) range(150, 100, -25) range(125, 100, -25)
```

这个例子实现将一个 `range` 对象，不断改变起点打印输出的功能。再来看看 `while` 如何简单测试卡拉兹(Callatz)猜想：对任何一个自然数 $n$ ，如果它是偶数，那么把它砍掉一半；如果它是奇数，那么把 $3n + 1$ 砍掉一半。这样一直反复砍下去，最后一定在某一步得到 $n = 1$ 。

```
卡拉兹猜想
num = int(eval(input('请输入初始值: ')))
while num != 1:
 if num % 2 == 0:
 num /= 2
 else:
 num = num*3+1
 print(num)
```

`while` 语句不仅限于这些功用，它可以通过 `continue` 直接跳到最近所在循环的开头处，来到循环的首行，而不继续往下执行。比如，下面一个例子将打印 0-10 之间的除 10 之外的所有偶数：

```
x = 10
while x:
 x -= 1
 if x % 2 != 0:
 continue # 跳过打印
 else:
 print(x, end=' ')
```

在 Python 中还提供了一种叫 `else` 子句的机制，用于完成 `while` 循环之后的后续操作。看一个判断质数的例子：

```
y = int(input('输入数字: '))
x = y // 2
while x > 1:
 if y % x == 0:
 print(y, '有因子', x)
 break
 x -= 1
else: # 没有碰到 break 才会执行
 print(y, '是质数! ')
```

该例子出现两个新的关键字 `break` 与 `else` 子句。`break` 用于终止当前所在的循环。`else` 子句用于处理当 `break` 没有执行的情况。这里例子不太直观，可以比较接下来两个例子来理解。假设你要写一个循环用于搜索列表的值，而且需要知道在离开循环后该值是否已经找到，可能会用下面的方式编写该任务：

```
found = False
while x and not found:
 if match(x[0]):
 print('Hi')
 found = True
 else:
 x = x[1:]
if not found:
 print('not found')
```

这个任务，您可以使用 `else` 子句进行简化：

```
while x:
 if match(x[0]):
 print('Hi')
 break
 x = x[1:]
else:
 print('not found')
```

可以看出 else 子句不仅仅简化代码量，而且代码的逻辑更加清晰。

## 4.3 函数

为了简化和重构代码，Python 使用关键字 def 定义函数。形式是：

```
def 函数名(参数):
 函数体
```

具体该如何使用函数，暂且不提，看看 Python 如何实现斐波那契数列，即  $a_0 = 0, a_1 = 1, a_{n+2} = a_n + a_{n+1}$ ：

```
n = 10
a, b = 0, 1
while a < n:
 print(a, end=' ')
 a, b = b, a+b
print()
```

该代码输出结果为 10 以内的斐波那契数列：

```
0 1 1 2 3 5 8
```

符合预期结果。但是，如果我们想要输出 20 以内的斐波那契数列，我们需要复制代码并修改终值 n：

```
n = 20
a, b = 0, 1
while a < n:
 print(a, end=' ')
 a, b = b, a+b
print()
```

这样也太麻烦了，且极大的增加了代码量，有没有一种简洁的方式呢？当然有了，Python 提供了函数这一概念用于封装上述的代码即：

```
def fib(n):
 """打印 n 以内的斐波那契数列序列"""
 a, b = 0, 1
 while a < n:
 print(a, end=' ')
 a, b = b, a+b
 print()
```

这样可以使用 fib(40) 打印 40 以内的斐波那契数列。

下面来拆解一下 fib 函数：其中的 """ 表示文档字符串（可以使用 fib.\_\_doc\_\_ 或者 help(fib) 来查看），用于注解函数，辅助函数的开发者与使用者了解函数的细节。n 是函数 fib 的参数。

此函数是使用 `print` 打印结果的，返回值是 `None`。为了能够直接获得返回值，需要借助 `return` 关键字：

```
def fib(n):
 "作为一个容器返回 n 以内的斐波那契数列序列"
 res = []
 a, b = 0, 1
 while a < n:
 res.append(a)
 a, b = b, a+b
 return res
```

函数的参数也可以设置默认值。为了说明默认值的便利，我们可以这样定义数的加法：

```
def add(x,y):
 "计算 x+y"
 return x + y
```

```
测试
print(add(1,2))
print(add(2,2))
print(add(3,2))
```

此代码测试了 `add` 函数的运算，而这些运算有一个特点：`y` 的取值均为 2，可以这样：

```
def add(x,y=2):
 "计算 x+y
 y 的默认值为 2
 "
 return x + y
```

```
测试
print(add(1))
print(add(2))
print(add(3))
```

该代码同样实现了上述加法运算，但是函数的参数传入更少了，有部分缺省（因为有默认值，故而可以不传参数）。像 `y=2` 这种形式的参数一般被称为关键字参数，而没有默认值的参数 `x` 被称为位置参数。位置参数在传参数时不能写错位置，且位置参数一定要放在关键字参数的前面，否则该函数的定义是不合法的。

注意：关键字参数的默认值只会执行一次。比如在下面的函数调用中，`L` 的值在不断的改变：

```
def f(a, L=[]):
 L.append(a)
 return L
测试
print(f(1))
print(f(2))
print(f(3))
```

输出结果为：

```
[1]
[1, 2]
[1, 2, 3]
```

如果您想要共享默认值，可以这样：

```
def f(a, L=None):
 if L is None:
 L = []
 L.append(a)
 return L
```

函数还有一种参数设计：可变参数。即类似于：

```
def func(a, b = "", *args, **kwargs):
 ...
```

其中 `args` 表示可变的位置参数列表，`kwargs` 为可变的关键字参数字典。在变量那一节了解了 `*` 的妙用，这里需要重申一个知识点：

```
a, *b, c = range(7)
print(type(b))
```

输出结果：

```
<class 'list'>
```

即赋值的打包操作是以 `list` 来存储数据的，而函数则是以 `tuple` 进行打包的：

```
def sum_str(*strings):
 print(type(strings))
 res = ""
 for s in strings: res += s
 return res
测试
print(sum_str('he','llo'))
```

输出结果：

```
<class 'tuple'>
hello
```

\*\* 用于传入多个关键字参数：

```
def test(**D):
 print(type(D))
测试
test(a='4')
```

输出结果为：

```
<class 'dict'>
```

即\*\*是以 `dict` 的形式进行打包的。

## 4.4 类与模块

可以说 Python 中的函数定义了功能，行为而类不仅仅定义了功能同时也定义了数据。类还有继承与多态的思想，使用关键字 `class` 定义。

模块可以将类与函数进行再次封装并保存在 `.py` 文件之中，不同的模块之间又可以通过 `__init__.py` 文件组织成为包。

关于 Python 的类与模块以及包比较抽象，限于篇幅，本章便不做展开了，您自行查阅相关资料即可。本书后续章节会再次展开讨论。

## 4.5 本章小结

本章简要的介绍了 Python 中的变量、流程控制与函数，略略提及类、模块与包。本章以及前一章仅仅是抛砖引玉，在后续的章节本书将以实例来说明如何使用 Python 玩转计算机视觉。

# 第二篇 数据篇

## 第 5 章 定制 AI 的专属数据 X

前面的章节已经介绍了本书使用的编程基础，本章将介绍如何利用之前的知识处理数据集。我们知道诸如 Keras、MXNet、TensorFlow、Pytorch 各大深度学习平台都封装了自己的基础数据集，如 MNIST、Cifar 等。如果要在不同平台使用这些数据集，还需要了解它们是如何组织这些数据集的，需要花费一些不必要的时间学习它们的 API。我们为何不去创建属于自己的数据集呢？本章使用了 Numpy 与 Pytables 封装了一些深度学习必备数据集。

### 5.1 数据集 X 的制作的准备工作

本小节将探讨如何将 MNIST、Fashion MNIST、Cifar10、Cifar100 封装进 X.h5 之中。这些数据集的简介和下载链接如下：

- MNIST: <http://yann.lecun.com/exdb/mnist>。
- Fashion MNIST: <https://github.com/zalandoresearch/fashion-mnist>。
- Cifar10 与 Cifar100: <https://www.cs.toronto.edu/~kriz/cifar.html>。
- 需要一些必备的包：struct、numpy、gzip、tarfile、os、time、pickle。

本小节的数据封装大都基于 Bunch 结构，下面来简单了解一下它的使用。

#### 5.1.1 好用的 Bunch

Bunch 是一个十分好用的 python 结构，具体是代码实现很简单：

```
class Bunch(dict):
 def __init__(self, *args, **kwds):
 super().__init__(*args, **kwds)
 self.__dict__ = self
```

Bunch 主要用于存储松散的数据，它能让我们以命令行参数的形式创建相关对象，并设置

对象的相关属性。下面来看看 Bunch 的魅力，Bunch 的定义利用了 dict 的特性。

下面我们构建一个 Bunch 的实例，Tom 它代表一个住在北京的 54 岁的人。

```
Tom = Bunch(age="54", address="Beijing") # 存储一个人的年龄、居住地址的 Bunch 实例
print('Tom 的年龄是 {}，他住在 {}'.format(Tom.age, Tom.address))
#print(x.age) # 获取该人的年龄
#x.address # 获取该人的居住地址
```

结果显示：

```
Tom 的年龄是 54，他住在 Beijing.
```

由于 Bunch 继承自 dict 类，我们可以自然而然获得大量与 dict 相关操作，如对于键值/属性值的遍历，或者简单查询一个属性是否存在。同时我们还可以直接对 Tom 增加属性，比如：

```
Tom.sex = 'male'
print(Tom)
```

结果为：

```
{'age': '54', 'address': 'Beijing', 'sex': 'male'}
```

你也许会奇怪，Bunch 结构与 dict 结构好像没有太大的的区别，只不过是多了一个点号运算，那么，Bunch 到底有什么神奇之处呢？下面以一个树式结构的数据来说明：

```
T = Bunch # 类重命名以简化书写
t = T(left=T(left='a',right='b'), right=T(left='c')) # 树数据
```

t 是一个树数据结构，它的结构见图 5.1 所示。

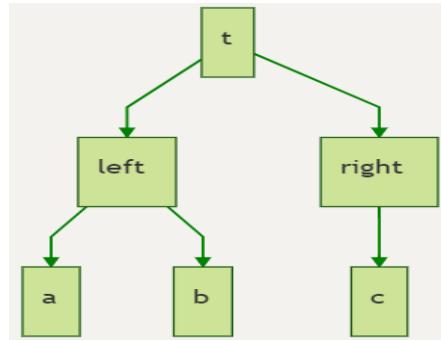


图 5.1 树数据

接着，便可以通过如下方式来使用 t：

```
t.left # 获取 t 的左边数据
```

结果如下：

```
{'left': 'a', 'right': 'b'}
```

我们也可以直接像下面的方式调取更多信息：

```
t.left.right
```

即：

```
'b'
```

除此之外，我们也可以通过键的方式获取数据：

```
t['left']['right']
```

为了判断某一个数据是否存在于 t 中，可以使用如下命令：

```
"left" in t.right
```

当然你也可以通过如下方式打印所有信息：

```
for first in t:
 print('第一层的节点：', first)
```

```
for second in t[first]:
 print('\t 第二层的节点: ', second)
 for node in t[first][second]:
 print('\t\t 第三层的节点: ', node)
```

结果:

```
第一层的节点: left
 第二层的节点: left
 第三层的节点: a
 第二层的节点: right
 第三层的节点: b
第一层的节点: right
 第二层的节点: left
 第三层的节点: c
```

介绍了这么多，我想大家应该对 Bunch 有了一个大体的印象，这些印象可以帮助我们理解接下来我要介绍的主菜：X.h5。

### 5.1.2 MNIST 的处理

接下来先定义一个类，用来处理 MNIST 与 Fashion\_MNIST 数据集。这两个数据集都是以如下形式来存储的：train-images-idx3-ubyte.gz，train-labels-idx1-ubyte.gz，t10k-images-idx3-ubyte.gz，t10k-labels-idx1-ubyte.gz。依据压缩包的文件名我们便可以很容易猜到它们各自代表的含义。为了代码的可复用性，将创建文件 mnist.py 并写入如下内容：

```
import struct
import numpy as np
import gzip
MNIST 类
class MNIST:
 def __init__(self, root, namespace, train=True):
 """
 (MNIST handwritten digits dataset from http://yann.lecun.com/exdb/mnist)
 (A dataset of Zalando's article images consisting of fashion products,
 a drop-in replacement of the original MNIST dataset from
 https://github.com/zalandoresearch/fashion-mnist)
 Each sample is an image (in 2D NDArray) with shape (28, 28).
 参数

 root : 数据根目录，如 'E:/Data/Zip/'
 namespace : 'mnist' or 'fashion_mnist'
 train : bool, default True
 Whether to load the training or testing set.
 ...
 self._train = train # 以此判断是否是训练数据集
```

```

self.namespace = namespace # 'mnist' 或者 'fashion_mnist'
root = root + namespace
self._train_data = f'{root}/train-images-idx3-ubyte.gz' # 训练数据集文件名
self._train_label = f'{root}/train-labels-idx1-ubyte.gz' # 训练数据集的标签文件名
self._test_data = f'{root}/t10k-images-idx3-ubyte.gz' # 测试数据集文件名
self._test_label = f'{root}/t10k-labels-idx1-ubyte.gz' # 测试数据集的标签文件名
self._get_data()
获取数据信息的函数
def _get_data(self):
 """
 官方网站是以 `[offset][type][value][description]` 的格式封装数据的，因而我们使用 `struct.unpack`
 ...
 if self._train:
 data, label = self._train_data, self._train_label # 获得训练数据集名称及其标签名称
 else:
 data, label = self._test_data, self._test_label # 获得测试数据集名称及其标签名称
 # 获取标签信息
 with gzip.open(label, 'rb') as fin:
 struct.unpack(">ll", fin.read(8)) # 参考数据集的网站，即 offset=8
 self.label = np.frombuffer(
 fin.read(), dtype=np.uint8).astype(np.int32) # 获得数据集的标签
 # 获取图像信息
 with gzip.open(data, 'rb') as fin:
 shape = struct.unpack(">IIII", fin.read(16))[1:]
 data = np.frombuffer(fin.read(), dtype=np.uint8)
 self.data = data.reshape(shape)
 """

```

代码的实现十分简单，具体细节可参考数据集的官方介绍。下一小节接着处理 Cifar 数据集。

### 5.1.3 Cifar-10的处理

Cifar 数据集的压缩包是 tar.gz 文件，此时需要将其解压方可进一步处理，为此定义函数：

```

import tarfile # 载入压缩包处理库
解压函数
def extractall(tar_name, root):
 """
 解压 tar 文件并返回路径
 root: 解压的根目录
 ...
 with tarfile.open(tar_name) as tar:
 tar.extractall(root) # 解压全部文件
 """

```

```
 names = tar.getnames() # 获取解压后的文件所在目录
 return names
```

同样，我们将其封装进 cifar.py 文件中，其中 Cifar 类的定义如下：

```
class Cifar(dict):
 def __init__(self, root, namespace, *args, **kwds):
 """CIFAR image classification dataset from https://www.cs.toronto.edu/~kriz/cifar.html
 Each sample is an image (in 3D NDArray) with shape (3, 32, 32).
 Parameters

 meta : 保存了类别信息
 root : str, 数据根目录
 namespace : 'cifar-10' 或 'cifar-100'
 train : bool, default True , 是否载入训练集
 """
 super().__init__(*args, **kwds)
 self._dict_ = self
 self.root = root
 self.namespace = namespace # 名称空间
 self._extract() # 解压数据集并载入到内存
 self._read_batch() # 获取我们需要的数据形式

 def _extract(self):
 tar_name = f'{self.root}{self.namespace}-python.tar.gz' # 文件名
 names = extractall(tar_name, self.root) # 解压后数据所在目录
 print('载入数据的字典信息：')
 start = time.time() # 开始计时
 for name in names:
 path = f'{self.root}{name}' # 获取子文件名或者子目录名
 if os.path.isfile(path): # 判断是否为文件
 if not path.endswith('.html'): # 是否为 html 文件
 k = name.split('/')[-1] # 拆分 path
 if path.endswith('.meta'): # 元数据
 k = 'meta'
 elif path.endswith('.txt~'): # 跳过 .txt~ 文件
 continue
 with open(path, 'rb') as fp: # 打开文件
 self[k] = pickle.load(fp, encoding='bytes') # 载入数据到内存
 # time.sleep(0.2)
 t = int(time.time() - start) * '-' # 计时结束
 print(t, end="")
 print('\n载入数据的字典信息完毕！')
 def _read_batch(self):
```

```

if self.namespace == 'cifar-10': # cifar 10
 self.trainX = np.concatenate([
 self[f'data_batch_{str(i)}'][b'data'] for i in range(1, 6)].reshape(-1, 3, 32, 32) # 训练集图片
 self.trainY = np.concatenate(
 [np.asarray(self[f'data_batch_{str(i)}'][b'labels']) for i in range(1, 6)]) # 训练集标签
测试集图片
 self.testX = self.test_batch[b'data'].reshape(-1, 3, 32, 32)
 self.testY = np.asarray(self.test_batch[b'labels']) # 测试集标签
elif self.namespace == 'cifar-100': # cifar 100
 self.trainX = self.train[b'data'].reshape(-1, 3, 32, 32) # 训练集图片
 self.train_fine_labels = np.asarray(
 self.train[b'fine_labels']) # 子类标签
 self.train_coarse_labels = np.asarray(
 self.train[b'coarse_labels']) # 超类标签
 self.testX = self.test[b'data'].reshape(-1, 3, 32, 32) # 测试集图片
 self.test_fine_labels = np.asarray(
 self.test[b'fine_labels']) # 子类标签
 self.test_coarse_labels = np.asarray(
 self.test[b'coarse_labels']) # 超类标签

```

在 Cifar 类中引用了 Bunch 结构，方便我们更加友好的处理数据集。至此各个数据集我们都已经处理好了，先看看具体的效果如何？

#### 5.1.4 如何使用 MNIST 与 Cifar 类

为了让代码的复用性更高，将上面的 mnist.py 与 cifar.py 文件一起放入 utils/tools 文件夹下以此来创建一个工具包。下面便可很容易导入它们并加以使用：

```

from utils.tools.mnist import MNIST # 载入 MNIST 类
from utils.tools.cifar import Cifar # 载入 Cifar 类

```

将 MNIST、Fashion MNIST、Cifar 10、Cifar 100 均放置在同一目录下：

```
root = 'E:/Data/Zip/' # 数据所在目录
```

下面来看看如何使用这两个类？首先是 MNIST：

```

mnist_train = MNIST(root, 'mnist', True) # 训练数据集
mnist_test = MNIST(root, 'mnist', True) # 测试数据集

```

接着，便可以查看数据的 shape 与 标签（以训练集为例）：

```
mnist_test.data.shape, mnist_train.label
```

结果为：

```
((60000, 28, 28), array([5, 0, 4, ..., 5, 6, 8]))
```

可以看出训练集有 60000 个样本，每张图片的尺寸均为 28x28 的灰度图片。关于图片的可视化与计算这里暂且不提。Fashion MNIST 的使用与 MNIST 相似，仅仅需要将 namespace 改为 fashion\_mnist。MNIST 的类很简单，但是 Cifar 类又该如何操作呢？

其实 Cifar 类的使用也是很简单的：

```
cifar10 = Cifar(root, 'cifar-10') # cifar10
```

结果为：

载入数据的字典信息：

载入数据的字典信息完毕！

由于 Cifar 是 Bunch 结构的数据，所以有：

```
cifar10.keys()
```

结果为：

```
dict_keys(['root', 'namespace', 'data_batch_4', 'test_batch', 'data_batch_3', 'meta', 'data_batch_2',
'data_batch_5', 'data_batch_1', 'trainX', 'trainY', 'testX', 'testY'])
```

可以看出这里面有许多我们需要的信息，但是我们仅仅考虑：

- 'trainX', 'trainY': 训练数据的图片与标签
- 'testX', 'testY': 测试数据的图片与标签
- 'meta': 标签的名称列表等

调取这些信息十分方便（可以参考 5.1.1 节）：

```
cifar10.meta
```

结果为：

```
{b'num_cases_per_batch': 10000,
 b'label_names': [b'airplane',
 b'automobile',
 b'bird',
 b'cat',
 b'deer',
 b'dog',
 b'frog',
 b'horse',
 b'ship',
 b'truck'],
 b'num_vis': 3072}
```

通过英文名称很容易判断它们各自的含义，不过我们仅仅需要知道标签名称列表，其他的可以忽略：

```
label_names = cifar10.meta[b'label_names']
```

至此准备工作已经完成，接下来进入 X 数据集的制作环节！

## 5.2 数据集 X 的制作

为了方便管理和调用数据集，将上面的 MNIST 与 Cifar 做了一定的调整（详细见：<https://github.com/DataLoaderX/datazone/tree/master/lab/utils/tools>），将 cifar 的图片(3, 32, 32) 存储形式转换为(32, 32, 3)，并定义一个 DataBunch 类来打包上面介绍的所有数据集：

```
class DataBunch(Bunch):
```

```
 ...
```

```
将多个数据集打包为 Bunch
...
def __init__(self, root, *args, **kwds):
 super().__init__(*args, **kwds)
 B = Bunch # 重命名
 self.mnist = B(MNIST(root, 'mnist')) # mnist
 self.fashion_mnist = B(MNIST(root, 'fashion_mnist')) # fashion_mnist
 self.cifar10 = B(Cifar(root, 'cifar-10'))
 self.cifar100 = B(Cifar(root, 'cifar-100'))
```

下面便可以直接利用 DataBunch 类来调用上述介绍的数据集了。

```
db = DataBunch(root)
```

我们可以通过`key` 查看封装的数据集:

```
db.keys()
```

结果:

```
dict_keys(['mnist', 'fashion_mnist', 'cifar10', 'cifar100'])
```

### 5.2.1 数据集的可视化与简介

前面的内容一直都是如何处理数据的，但是数据具体长什么样子，可以通过 matplotlib 来可视化。先定义一个可视化的函数：

```
from matplotlib import pyplot as plt
import numpy as np
def show_imgs(imgs):
 ...
 展示 多张图片
 ...
 n = imgs.shape[0] # 图片的个数
 h, w = 4, int(n / 4) # 图片队列的行数与列数
 figs = plt.subplots(h, w, figsize=(5, 5))
 K = np.arange(n).reshape((h, w))
 for i in range(h):
 for j in range(w):
 img = imgs[K[i, j]] # 取出一张图片并放入指定位置
 figs[i][j].imshow(img) # 显示图片
 figs[i][j].axes.get_xaxis().set_visible(False) # 隐藏坐标轴 x
 figs[i][j].axes.get_yaxis().set_visible(False) # 隐藏坐标轴 y
 plt.show()
```

先从 MNIST 的训练集中取出 16 张图片进行可视化：

```
imgs = db.mnist.trainX[16] # 取出 16 张图片
show_imgs(imgs)
```

结果，如图 5.2 所示。

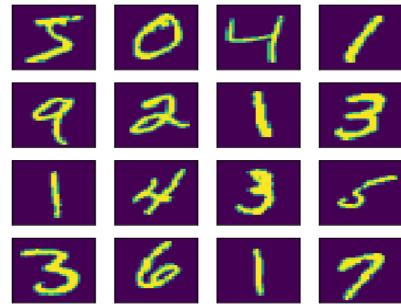


图 5.2 MNIST 示例

同样，对 FASHION MNIST 进行可视化：

```
imgs = db.fashion_mnist.testX[:16] # 取出 16 张图片
show_imgs(imgs)
```

结果，如图 5.3 所示。

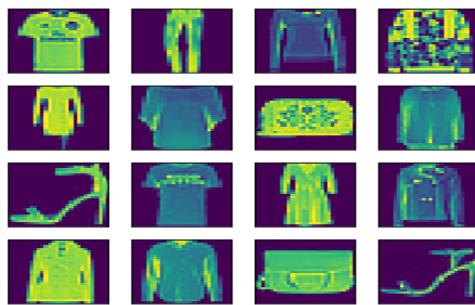


图 5.3 fashion mnist 示例

Cifar10 的可视化：

```
imgs = db.cifar10.testX[:16] # 取出 16 张图片
show_imgs(imgs)
```

结果，如图 5.4 所示。

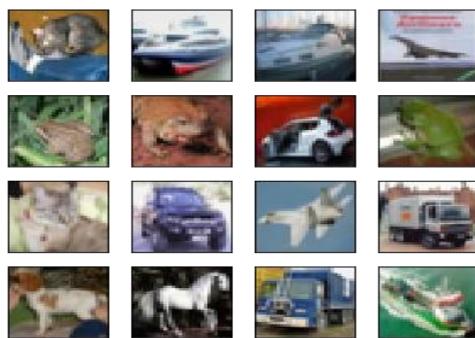


图 5.4 cifar10 示例

Cifar100 的可视化：

```
imgs = db.cifar100.testX[:16] # 取出 16 张图片
show_imgs(imgs)
```

结果，如图 5.5 所示。

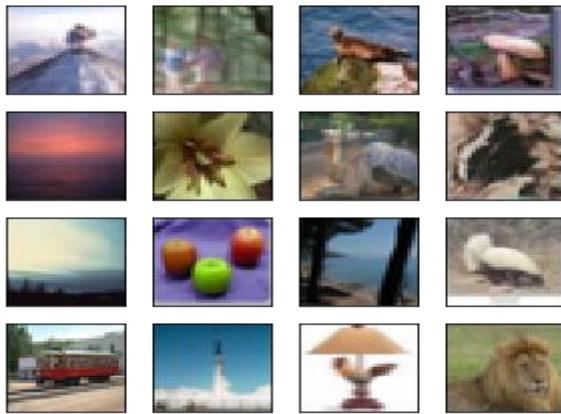


图 5.5 cifar100 示例

虽然我们一瞥了数据集的样子，但是对其做简要的介绍还是十分有必要的。MNIST 数据集可以说是深度学习中的 hello world 级别的数据集，很多教程都是把它作为入门级的数据集。不过有些人可能对它还不是很了解，下面简单的了解一下。

MNIST 数据集来自美国国家标准与技术研究所（National Institute of Standards and Technology, NIST）训练集(training set)来自 250 个不同人（其中 50% 是高中学生，剩余的来自人口普查局(the Census Bureau)的工作人员）手写的数字构成。测试集(test set)也是同样比例的手写数字数据 MNIST 是 NIST 的子集，60000 个样本作为训练集，10000 个样本作为测试集。同时 MNIST 的数字图像已被大小规范化，并以固定大小的图像居中。

虽然图像分类数据集中最常用的是手写数字识别数据集 MNIST，但大部分模型在 MNIST 上的分类精度都超过了 95%。为了更直观地观察算法之间的差异，我们可以使用一个图像内容更加复杂的数据集 Fashion-MNIST。Fashion-MNIST 和 MNIST 一样，也包括了 10 个类别，分别为：t-shirt (T 恤)、trouser (裤子)、pullover (套衫)、dress (连衣裙)、coat (外套)、sandal (凉鞋)、shirt (衬衫)、sneaker (运动鞋)、bag (包) 和 ankle boot (短靴)。Fashion-MNIST 的存储方式和 MNIST 是一样的，故而，我们可以使用相同的方式对其进行处理。

cifar-10 和 CIFAR-10 是由 Alex Krizhevsky, Vinod Nair, 和 Geoffrey Hinton 收集的 8000 万个微小图像的子集(<http://people.csail.mit.edu/torralba/tinyimages/>)。其中 cifar-10 数据集由 10 类  $32 \times 32$  彩色图像组成，每类有 6000 张图像。被划分为 50 000 张训练图像和 10 000 张测试图像。而 Cifar100 则分为 10 个粗糙类别和 100 个精细类别。

## 5.2.2 数据集 X 的封装

虽然定义了 DataBunch 类，但是，这样并没有解决本章提出的主题：让数据集更加友好？为此，我们需要将数据集进行序列化，并将其命名为数据集 X。对于 Python 来说 pickle 是一个很不错的序列化工具：

```
import pickle
def write_bunch(path):
 # path:: 写入数据集的文件路径
```

```

with open(path, 'wb') as fp:
 pickle.dump(db, fp)
root = 'E:/Data/Zip/' # # 数据集序列化的目录
path = f'{root}X.json' # 写入数据集的文件路径, 也可以是 .dat 文件
write_bunch(path)

```

这样以后就可以直接复制 `f'{root}X.dat` 或 `f'{root}X.json'` 到你可以放置的任何地方, 然后你就可以通过 `load` 函数来调用 MNIST、Fashion MNIST、Cifar 10、Cifar 100 这些数据集。即反序列化:

```

def read_bunch(path):
 with open(path, 'rb') as fp:
 bunch = pickle.load(fp) # 即为上面的 DataBunch 的实例
 return bunch
db = read_bunch(path) # path 即你的数据集所在的路径

```

考虑到 JSON 或者 dat 对于其他编程语言的不太友好, 下面将介绍如何将 Bunch 数据集存储为 HDF5 格式的数据。

### 5.2.3 Bunch 转换为 HDF5 文件: 高效存储数据集 X

PyTables 是 Python 与 HDF5 数据库/文件标准的结合。它专门为优化 I/O 操作的性能、最大限度地利用可用硬件而设计, 并且它还支持压缩功能。Pytables 为数据的可移植提供了极大的便利, 下面我们来见识它的魅力! 下面的代码均是在 Jupyter NoteBook 下完成的:

```

import tables as tb
import numpy as np
def bunch2hdf5(root):
 ...
 这里我仅仅封装了 Cifar10、Cifar100、MNIST、Fashion MNIST 数据集,
 使用者还可以自己追加数据集。
 ...
 db = DataBunch(root)
 filters = tb.Filters(complevel=7, shuffle=False)
 # 这里我采用了压缩表, 因而保存为 '.h5c' 但也可以保存为 '.h5'
 with tb.open_file(f'{root}X.h5c', 'w', filters=filters, title='XinetW's dataset') as h5:
 for name in db.keys():
 h5.create_group('/', name, title=f'{db[name].url}')
 if name != 'cifar100':
 h5.create_array(h5.root[name], 'trainX', db[name].trainX, title='训练数据')
 h5.create_array(h5.root[name], 'trainY', db[name].trainY, title='训练标签')
 h5.create_array(h5.root[name], 'testX', db[name].testX, title='测试数据')
 h5.create_array(h5.root[name], 'testY', db[name].testY, title='测试标签')
 else:
 h5.create_array(h5.root[name], 'trainX', db[name].trainX, title='训练数据')

```

```

h5.create_array(h5.root[name], 'testX', db[name].testX, title='测试数据')
h5.create_array(h5.root[name], 'train_coarse_labels', db[name].train_coarse_labels, title='超类
训练标签')
h5.create_array(h5.root[name], 'test_coarse_labels', db[name].test_coarse_labels, title='超类测
试标签')
h5.create_array(h5.root[name], 'train_fine_labels', db[name].train_fine_labels, title='子类训练
标签')
h5.create_array(h5.root[name], 'test_fine_labels', db[name].test_fine_labels, title='子类测试标
签')

for k in ['cifar10', 'cifar100']:
 for name in db[k].meta.keys():
 name = name.decode()
 if name.endswith('names'):
 label_names = np.asarray([label_name.decode() for label_name in
 db[k].meta[name.encode()]])
 h5.create_array(h5.root[k], name, label_names, title='标签名称')

```

我们可以自如的使用数据集 X 了。

```

root = 'E:/Data/Zip/' # X 所在目录
bunch2hdf5(root) # 将 db(Bunch) 转换为 HDF5 文件
h5c = tb.open_file('E:/Data/Zip/X.h5c') # 载入 数据集 X

```

看看 JSON 与 HDF5 文件大小:

```

import os
h5_size = os.path.getsize('E:/Data/Zip/X.h5')/1e6 # 文件的大小
json_size = os.path.getsize('E:/Data/Zip/X.json')/1e6 # 单位为 M
h5_size, json_size

```

结果:

```
(479.697264, 852.202724)
```

保存相同的内容，此时 JSON 比 HDF5 大了近 1.8 倍！不仅如此，HDF5 载入数据的速度也是很快的：

```

%%time
arr = h5.root.cifar100.trainX.read() # 读取数据十分快速

```

结果:

```
Wall time: 125 ms
```

如果你想要像之前 db 那样获取你想要的数据信息，也是十分简单的。

### 5.3 X.h5 的使用说明

下面以 Cifar10 为例来展示本章自创的数据集 X.h5（我将 X 数据集上传到了百度云盘「链接：[https://pan.baidu.com/s/12jzaJ2d2kvHCXbQa\\_HO6YQ](https://pan.baidu.com/s/12jzaJ2d2kvHCXbQa_HO6YQ) 提取码：2clg」可以下载直接使用；亦可你自己生成，不过我推荐自己生成，可以加深理解如何使用 Python 处理数据集）。

```
root = 'datasets/X.h5' # X 数据集所在路径
h5 = tb.open_file(root) # 获取 X 数据集
h5.root # 查看 X 所包含的数据集
```

结果为：

```
/ (RootGroup) "Xinet's dataset" children := ['cifar10' (Group), 'cifar100' (Group), 'fashion_mnist' (Group),
'mnist' (Group)]
```

以相对复杂的 cifar100 数据集为例来说明如何使用数据集 X？

```
cifar100 = h5.root.cifar100
cifar100
```

结果为：

```
/cifar100 (Group) 'https://www.cs.toronto.edu/~kriz/cifar.html'
 children := ['coarse_label_names' (Array), 'fine_label_names' (Array), 'testX' (Array), 'test_coarse_labels'
(Array), 'test_fine_labels' (Array), 'trainX' (Array), 'train_coarse_labels' (Array), 'train_fine_labels' (Array)]
```

'coarse\_label\_names'指的是粗粒度或超类标签名，'fine\_label\_names'则是细粒度标签名。可以使用 read()方法直接获取你需要的信息，也可以使用索引的方式获取：

```
coarse_label_names = cifar100.coarse_label_names[:]
或者
coarse_label_names = cifar100.coarse_label_names.read()
coarse_label_names.astype('str')
```

结果为：

```
array(['aquatic_mammals', 'fish', 'flowers', 'food_containers',
 'fruit_and_vegetables', 'household_electrical_devices',
 'household_furniture', 'insects', 'large_carnivores',
 'large_man-made_outdoor_things', 'large_natural_outdoor_scenes',
 'large_omnivores_and_herbivores', 'medium_mammals',
 'non-insect_invertebrates', 'people', 'reptiles', 'small_mammals',
 'trees', 'vehicles_1', 'vehicles_2'], dtype='|<U30')
```

'testX'与'trainX'分别代表测试数据和训练数据，而其他的节点所代表的含义也是类似的。

例如，可以看看训练集的数据和标签：

```
trainX = cifar100.trainX
train_coarse_labels = cifar100.train_coarse_labels[:]
```

结果为：

```
array([11, 15, 4, ..., 8, 7, 1])
```

数据也可以通过下面的形式获取：

```
h5.get_node(h5.root.cifar100, 'trainX')
```

更甚者，我们可以直接定义迭代器来获取数据：

```
def data_iter(X, Y, batch_size): # 定义一个迭代器
 n = X.nrows # 样本数
 idx = np.arange(n) # 样本索引
 if X.name.startswith('train'): # 判断是否为训练集
 np.random.shuffle(idx) # 训练集需要打乱
```

```
for i in range(0, n ,batch_size):
 k = idx[i: min(n, i + batch_size)].tolist() # 批量的索引
 yield np.take(X, k, 0), np.take(Y, k, 0) # 取出一个批量数据
```

使用迭代器：

```
for x, y in data_iter(trainX, train_coarse_labels, 8):
 print(x.shape, y)
 break
```

结果为：

```
(8, 32, 32, 3) [7 7 0 15 4 8 8 3]
```

更多使用详情见：使用迭代器获取Cifar等常用数据集：

<https://yq.aliyun.com/articles/614332?spm=a2c4e.11155435.0.0.30543312vFsboY>

为了更加形象的说明该数据集，我们将其可视化：

```
import numpy as np
from matplotlib import pyplot as plt
%matplotlib inline
plt.rcParams['font.sans-serif'] = ['SimHei'] # 指定默认字体
plt.rcParams['axes.unicode_minus'] = False # 解决保存图像是负号 '-' 显示为方块的问题
def show_imgs(imgs, labels):
 # 展示多张图片
 n = imgs.shape[0]
 h, w = 4, int(n / 4)
 fig, ax = plt.subplots(h, w, figsize=(7, 7))
 K = np.arange(n).reshape((h, w)) # 显示的图片规格
 names = np.asarray([cifar.fine_label_names[label]
 for label in labels], dtype='U')
 names = names.reshape((h, w)) # 标签名称
 for i in range(h):
 for j in range(w):
 img = imgs[K[i, j]]
 ax[i][j].imshow(img)
 ax[i][j].axes.get_yaxis().set_visible(False)
 ax[i][j].axes.set_xlabel(names[i][j])
 ax[i][j].set_xticks([])
 plt.show()
```

为了高效使用数据集 X.h5，使用迭代器的方式来获取它。一般地，在深度学习领域，大都将数据集以批量的形式读取：

```
class Loader:
 """
 方法
 =====
 L 为该类的实例
```

```

len(L)::返回 batch 的批数
iter(L)::即为数据迭代器
Return
=====
可迭代对象 (numpy 对象)
"""

def __init__(self, X, Y, batch_size, shuffle):
 """
 X, Y 均为类 numpy
 """

 self.X = X # 特征
 self.Y = Y # 标签
 self.batch_size = batch_size # 批量大小
 self.shuffle = shuffle # 是否打乱

def __iter__(self):
 n = len(self.X) # 样本数目
 idx = np.arange(n) # 样本索引
 if self.shuffle:
 np.random.shuffle(idx)
 for k in range(0, n, self.batch_size):
 K = idx[k:min(k + self.batch_size, n)].tolist()
 yield np.take(self.X, K, 0), np.take(self.Y, K, 0) # 取出批量

def __len__(self):
 return int(np.ceil(len(self.X) / self.batch_size))

```

最后，看看效果如何：

```

import tables as tb
import numpy as np
batch_size = 512
cifar = h5.root.cifar100
train_cifar = Loader(cifar.trainX, cifar.train_fine_labels, batch_size, True)
for imgs, labels in iter(train_cifar):
 break
show_imgs(imgs[:16], labels[:16])

```

结果展示，如图 5.6 所示。



图 5.6 展示一个批量的图片

## 5.4 本章小结

上面的 API 设计和不断改进的过程中，可以获得学习和创造的喜悦。上面所介绍的 X.h5 数据集不仅仅是那些数据集的封装，你还可以继续添加自己的数据集到数据 X 中。同时，类 Loader 十分有用，它定义了一个标准，一个可以延拓到处理其他深度学习的数据集中去。

# 第 6 章 CASIA 脱机和在线手写汉字库

本章通过 Python 解析 CASIA 脱机和在线手写汉字库中的单字库，主要从以下两个方面进行展开：

- 单字库的特征的解码、序列化与反序列化；
- 单字库的图片的解码、序列化与反序列化。

## 6.1 CASIA 手写汉字简介

CASIA-HWDB (CASIA-OLHWDB) 数据库由中科院自动化研究所在 2007-2010 年间收集，包含 1020 人书写的脱机（联机）手写中文单字样本和手写文本，用 Anoto 笔在点阵纸上书写后扫描、分割得到。

本数据库经签约授权后可免费用于学术研究目的，但用于商业目的需付费。学术研究的用途包括手写文档分割、字符识别、字符串识别、文档检索、书写人适应、书写人鉴别等。

如果需要使用该数据集需要提交申请书：

- CASIA-HWDB:<http://www.nlpr.ia.ac.cn/databases/download/CASIA-HWDB-Chinese.pdf>
- CASIA-OLHWDB:<http://www.nlpr.ia.ac.cn/databases/download/CASIA-OLHWDB-Chinese.pdf>

数据集的下载网址: <http://www.nlpr.ia.ac.cn/databases/handwriting/Download.html>。

在申请书中介绍了数据集的基本情况:

CASIA-HWDB 手写单字样本分为三个数据库: HWDB1.0~1.2。手写文本分为三个数据库: HWDB2.0~2.2。

CASIA-OLHWDB 手写单字样本分为三个数据库: OLHWDB1.0~1.2, 手写文本也分为三个数据库: OLHWDB2.0~2.2。

仅仅考虑 CASIA Online and Offline Chinese Handwriting Databases (<http://www.nlpr.ia.ac.cn/databases/handwriting/Download.html>) 下载页提供的手写单字数据, 并且下载到了 root 目录下:

```
import os
root = 'E:/OCR/CASIA/data' # CASI 数据集所在根目录
os.listdir(root)
```

输出结果:

```
['HWDB1.0trn.zip',
 'HWDB1.0trn_gnt.zip',
 'HWDB1.0tst.zip',
 'HWDB1.0tst_gnt.zip',
 'HWDB1.1trn.zip',
 'HWDB1.1trn_gnt.zip',
 'HWDB1.1tst.zip',
 'HWDB1.1tst_gnt.zip',
 'OLHWDB1.0test_pot.zip',
 'OLHWDB1.0train_pot.zip',
 'OLHWDB1.0trn.zip',
 'OLHWDB1.0tst.zip',
 'OLHWDB1.1trn.zip',
 'OLHWDB1.1trn_pot.zip',
 'OLHWDB1.1tst.zip',
 'OLHWDB1.1tst_pot.zip']
```

CASIA 单字数据库不仅仅提供了单字数据的图片还提供了这些单字数据的特征, 并依 fileFormat-mpf.pdf([http://www.nlpr.ia.ac.cn/databases/download/feature\\_data/FileFormat-mpf.pdf](http://www.nlpr.ia.ac.cn/databases/download/feature_data/FileFormat-mpf.pdf)) 格式来保存其特征。简单点说, 每个单字的特征均以.mpf 形式保存手工特征。以\_pot 结尾的压缩文件保存了在线单字的图片信息, 而以\_gnt 结尾的压缩文件则保存了离线单字的图片信息。

## 6.2 手写单字的特征的解码

对照 fileFormat-mpf.pdf 给出的约定可以得出如下的解码方式:

```
class MPF:
 # MPF 文件的解码器
 def __init__(self, fp):
```

```

self._fp = fp
解码文件头
header_size = struct.unpack('l', self._fp.read(4))[0]
文件保存的形式，如 “MPF”
self.code_format = self._fp.read(8).decode('ascii').rstrip('\x00')
文本说明
self.text = self._fp.read(header_size - 62).decode().rstrip('\x00')
编码类型，如 “ASCII” , “GB” , etc
self.code_type = self._fp.read(20).decode('latin-1').rstrip('\x00')
编码长度
self.code_length = struct.unpack('h', self._fp.read(2))[0]
self.dtype = self._fp.read(20).decode('ascii').rstrip('\x00')
if self.dtype == 'unsigned char':
 self.dtype = np.uint8
else:
 self.dtype = np.dtype(self.dtype)
样本数
self.nrows = struct.unpack('l', self._fp.read(4))[0]
特征的维度
self.ndims = struct.unpack('l', self._fp.read(4))[0]
def __iter__(self):
 m = self.code_length + self.ndims
 for i in range(0, m * self.nrows, m):
 # 样本的标签
 label = self._fp.read(self.code_length).decode('gb2312-80')
 # 样本的特征
 data = np.frombuffer(self._fp.read(self.ndims), self.dtype)
 yield data, label 先以 'HWDB1.0trn.zip' 数据集为例来说明 MPF:
z = zipfile.ZipFile(f'{root}/HWDB1.0trn.zip')
z.namelist()[1:5] # 查看前 4 个人写的 MPF

```

结果如下：

```

['HWDB1.0trn/001.mpf',
'HWDB1.0trn/002.mpf',
'HWDB1.0trn/003.mpf',
'HWDB1.0trn/004.mpf']

```

下面以 'HWDB1.0trn/001.mpf' 为例展示如何使用 MPF:

```

import pandas as pd
fp = z.open('HWDB1.0trn/001.mpf') # 查看第一个写手
mpf = MPF(fp) # 解码
df = pd.DataFrame.from_records([(label, data) for data, label in mpf])
将 records 转换为 pd.DataFrame
df = pd.DataFrame(data=np.column_stack(df[1]).T, index=df[0])
df.head() # 查看前 5 个字

```

输出结果，如图 6.1 所示。

0	1	2	3	4	5	6	7	8	9	...	502	503	504	505	506	507	508	509	510	511
0																				
𢃠	11	11	0	3	8	1	0	0	16	4	...	0	0	41	7	8	4	14	10	0
𢃡	5	8	25	2	2	22	7	0	32	20	...	10	0	20	28	24	18	9	5	3
鄂	27	10	5	35	7	1	8	12	57	25	...	1	0	9	7	2	0	5	15	1
餓	21	1	1	11	2	2	3	13	34	1	...	25	3	16	1	15	20	11	2	35
恩	26	4	2	4	1	26	12	0	60	9	...	1	3	9	5	42	31	5	0	4

5 rows × 512 columns

图 6.1 一个 mpf 样例

由上图 6.1 可以看出，每个字均有 512 个特征，而这些特征的具体信息，可见：

mpf.text

结果如下：

```
'Character features extracted from grayscale images. #ftrtype=ncg, #norm=ldi, #aspect=4, #dirn=8,
#zone=8, #zstep=8, #fstep=8, $deslant=0, $smooth=0, $nmdir=0, $multisc=0'
```

具体含义暂时还没有深入，暂且先以 text 记录下来。虽然 MPF 很好的解码了.mpf 文件，但是不免有点繁琐，为了更加人性化，考虑树结构：

```
class Bunch(dict):
 def __init__(self, *args, **kwargs):
 # 树结构
 super().__init__(*args, **kwargs)
 self.__dict__ = self
```

具体使用方法如下：

```
mb = Bunch() # 初始化树结构，生成树根
for name in z.namelist():
 if name.endswith('.mpf'): # 排除根目录名称
 # 树的枝桠名称
 writer_ = f"writer{os.path.splitext(name)[0].split('/')[1]}"
 with z.open(name) as fp:
 mpf = MPF(fp)
 df = pd.DataFrame.from_records([(label, data) for data, label in mpf])
 df = pd.DataFrame(data=np.column_stack(df[1].T, index=df[0]))
 # writer_ 枝桠上的两片叶子
 mb[writer_] = Bunch({"text":mpf.text, 'features':df})
```

如此以来，便可以通过树结构来获取自己想要的信息，比如，查看第 001 人写的字，可以使用如下命令：

mb.writer001.features.index

显示结果：

```
Index(['𢃠', '𢃡', '鄂', '餓', '恩', '而', '儿', '耳', '尔', '餌',
 ...,
 '墮', '蛾', '峨', '鹅', '俄', '额', '讹', '娥', '恶', '厄'],
 dtype='object', name=0, length=3728)
```

可以看出第 001 人（写手的 ID）写了 3728 个字。

## 6.3 手写单字的特征的序列化与反序列化

废了这么多力气，解析的数据如果不序列化（写入磁盘）是十分不智的行为。对于序列化，Python 提供了两种通用方式：JSON 与 HDF5。它们的实现方法均很简单：

(1) 序列化为 HDF5：

```
序列化的路径
save_path = 'E:/OCR/CASIA/datasets/features.h5'
for writer_id in mb.keys():
 mb[writer_id].features.to_hdf(save_path, key = writer_id, complevel = 7)
重新读取已经序列化的数据也是十分简单的：
df = pd.read_hdf(save_path, 'writer1001') # 通过 key 获取
```

(2) 序列化为 JSON：

```
import pickle
def bunch2json(bunch, path):
 # bunch 序列化为 JSON
 with open(path, 'wb') as fp:
 pickle.dump(bunch, fp)
```

反序列化亦很简单：

```
def json2bunch(path):
 # JSON 反序列化为 bunch
 with open(path, 'rb') as fp:
 X = pickle.load(fp)
 return X
```

具体操作：

```
path = 'E:/OCR/CASIA/datasets/features.json'
bunch2json(mb, path) # 序列化
X = json2bunch(path) # 反序列化
```

下面为了代码的简洁，定义一个函数：

```
def MPF2Bunch(set_name):
 # 将 MPF 转换为 bunch
 Z = zipfile.ZipFile(set_name)
 mb = Bunch()
 for name in Z.namelist():
 if name.endswith('.mpf'):
 writer_ = f"writer{os.path.splitext(name)[0].split('/')[1]}"
 with Z.open(name) as fp:
 mpf = MPF(fp)
 df = pd.DataFrame.from_records([(label, data) for data, label in mpf])
 df = pd.DataFrame(data=np.column_stack(df[1]).T, index=df[0])
 mb[writer_] = Bunch({"text":mpf.text, 'features':df})
 return mb
```

这里还有一个问题：HDF5 文件丢弃了 text 信息，显然并不是我们想要的效果。为了将 text 加入 HDF5 文件中，需要引入新的库 tables：

```
import tables as tb
```

具体如何使用 tables 暂且放下，先来考虑所有单字数据的特征数据：

```
feature_paths = {
```

```
os.path.splitext(name)[0].replace('.', ''):
 f'{root}/{name}' for name in os.listdir(root) if '_' not in name}
feature_paths
```

结果显示：

```
{'HWDB10trn': 'E:/OCR/CASIA/data/HWDB1.0trn.zip',
'HWDB10tst': 'E:/OCR/CASIA/data/HWDB1.0tst.zip',
'HWDB11trn': 'E:/OCR/CASIA/data/HWDB1.1trn.zip',
'HWDB11tst': 'E:/OCR/CASIA/data/HWDB1.1tst.zip',
'OLHWDB10trn': 'E:/OCR/CASIA/data/OLHWDB1.0trn.zip',
'OLHWDB10tst': 'E:/OCR/CASIA/data/OLHWDB1.0tst.zip',
'OLHWDB11trn': 'E:/OCR/CASIA/data/OLHWDB1.1trn.zip',
'OLHWDB11tst': 'E:/OCR/CASIA/data/OLHWDB1.1tst.zip'}
```

这样所有的单字数据的特征可以使用一个 dict 表示：

```
root_dict = Bunch({
 name: MPF2Bunch(feature_paths[name]) for name in feature_paths
})
```

由于总共有8个数据块，故而需要为 HDF5 文件生成8个“头”：

```
save_path = 'E:/OCR/CASIA/datasets/features.h5' # 数据保存的路径
filters = tb.Filters(complevel=7, shuffle=False) # 过滤信息，用于压缩文件
h = tb.open_file(save_path, 'w', filters=filters, title='Xinet's dataset')
for name in root_dict:
 h.create_group('/', name = name, filters=filters) # 生成数据集"头"
 for writer_id in root_dict[name].keys():
 h.create_group('/'+name, writer_id) # 生成写手"头"
```

查看数据集：

```
h.root
```

结果显示为：

```
/ (RootGroup) "Xinet's dataset"
 children := ['HWDB10trn' (Group), 'HWDB10tst' (Group),
 'HWDB11trn' (Group), 'HWDB11tst' (Group), 'OLHWDB10trn' (Group),
 'OLHWDB10tst' (Group), 'OLHWDB11trn' (Group), 'OLHWDB11tst' (Group)]
```

如果要查看'HWDB1.0trn'数据集下某个人可以这样操作：

```
h.root.HWDB10trn.writer001
```

结果显示：

```
/HWDB10trn/writer001 (Group)
 children := []
```

当然，h.root.HWDB10trn.writer001 现在里面什么也没有，因为还没有为它添加内容：

```
def bunch2hdf(root_dict, save_path):
 filters = tb.Filters(complevel=7, shuffle=False) # 过滤信息，用于压缩文件
 h = tb.open_file(save_path, 'w', filters=filters, title='Xinet's dataset')
 for name in root_dict: # 生成数据集"头"
 h.create_group('/', name = name, filters=filters)
 for writer_id in root_dict[name].keys(): # 生成写手
 h.create_group('/'+name, writer_id)
 h.create_array(f"/{name}/{writer_id}", 'text', root_dict[name][writer_id]['text'].encode())
 features = root_dict[name][writer_id]['features']
 h.create_array(f"/{name}/{writer_id}", 'labels', " ".join([l for l in features.index]).encode())
 h.create_array(f"/{name}/{writer_id}", 'features', features.values)
```

```
h.close() # 防止资源泄露
```

为了后续与 JSON 进行比较，下面将时间因素考虑进去：

```
%%time
save_path = 'E:/OCR/CASIA/datasets/features.h5'
bunch2hdf(save_path, root_dict)
```

输出结果：

```
Wall time: 57.4 s
```

而将 bunch 序列化为 JSON 仅仅使用函数 bunch2json 即可：

```
%%time
json_path = 'E:/OCR/CASIA/datasets/features.json'
bunch2json(root_dict, json_path)
```

输出结果：

```
Wall time: 58.5 s
```

HDF5 与 JSON 的反序列化很简单：

```
%%time
h = tb.open_file(save_path)
```

使用时间为：

```
Wall time: 158 ms
```

而 JSON 载入的时间就有点长：

```
%%time
j = json2bunch(json_path)
```

使用时间：

```
Wall time: 32.3 s
```

从上面的操作可以看出，虽然序列化为 JSON 或 HDF5 花费的时间都很短，不足 1 分钟，但生成 HDF5 相对快一点。反序列化时 JSON 花费的时间远多于 HDF5 花费的时间。

下面再看看二者在存储上的差距：

```
from sys import getsizeof
source_size = 0
for path in feature_paths.values():
 source_size += os.path.getsize(path)
print("源数据文件总大小", source_size/1e9)
print("JSON Python 对象占用空间大小为：", getsizeof(j), '文件大小为', os.path.getsize(json_path)/1e9)
print("HDF5 Python 对象占用空间大小为：", getsizeof(h), '文件大小为', os.path.getsize(save_path)/1e9)
```

输出结果（单位是 GB）：

```
源数据文件总大小 1.718896862
JSON Python 对象占用空间大小为： 368 文件大小为 2.820918817
HDF5 Python 对象占用空间大小为： 80 文件大小为 2.775279132
```

可以看出使用 JSON 与 HDF5 编码的数据均比源数据占用的空间要大 1.5 倍。虽然牺牲了存储空间，但是 JSON 与 HDF5 在数据的解码和读取上更为高效。

在解析的方式上，JSON 与 HDF5 极为相似，都是“树”结构。查看数据集：

```
h.root
```

输出结果为：

```
/ (RootGroup) "Xinet's dataset"
children := ['HWDB10trn' (Group), 'HWDB10tst' (Group),
'HWDB11trn' (Group), 'HWDB11tst' (Group), 'OLHWDB10trn' (Group),
```

```
'OLHWDB10tst' (Group), 'OLHWDB11trn' (Group), 'OLHWDB11tst' (Group)]
```

而 JSON 使用 dict 的方式获取：

```
j.keys()
```

输出结果为：

```
dict_keys(['HWDB10trn', 'HWDB10tst', 'HWDB11trn', 'HWDB11tst',
'OLHWDB10trn', 'OLHWDB10tst', 'OLHWDB11trn', 'OLHWDB11tst'])
```

获取某个数据集的写手亦是通过点运算：h.root.HWDB10tst 与 j.HWDB10tst.keys(); 同样获取写手的信息也是通过点运算：h.root.HWDB10trn.writer001 与 j.HWDB10trn.writer001.keys()。而查看一个写手写的具体汉字，则略有不同：

查看文本信息，h.root.HWDB10trn.writer007.text.read().decode() 或者 j.HWDB10trn.writer007.text 均可输出：

```
'Character features extracted from grayscale images.
#ftrtype=ncg, #norm=ldi, #aspect=4, #dirn=8, #zone=8,
#zstep=8, #fstep=8, $deslant=0, $smooth=0, $nmdir=0, $multisc=0'
```

查看特征，JSON 比较直接：

```
j.HWDB10trn.writer007.features.head() # 前 5 特征
```

输出结果，如图 6.2 所示。

0	1	2	3	4	5	6	7	8	9	...	502	503	504	505	506	507	508	509	510	511
<b>0</b>																				
昌	21	8	6	1	0	5	5	0	28	12	...	0	0	31	41	18	11	5	0	0
屹	2	25	9	0	7	6	1	0	26	43	...	0	1	13	2	14	46	16	1	0
亿	4	13	11	1	0	0	0	0	3	3	...	0	1	11	0	4	43	29	7	1
役	11	7	3	19	6	4	10	0	13	17	...	3	0	11	6	7	1	13	51	46
臆	8	2	6	3	9	5	4	0	11	3	...	1	7	6	31	15	0	14	12	6

5 rows × 512 columns

图 6.2 JSON 格式的数据特征

但是，HDF5 将特征与标签分开，查看 HDF5 的特征：

```
c = h.root.HWDB10trn.writer007.features
```

```
c[:5] # 前 5 特征
```

输出结果：

```
array([[21, 8, 6, ..., 0, 0, 0],
 [2, 25, 9, ..., 1, 0, 0],
 [4, 13, 11, ..., 7, 1, 0],
 [11, 7, 3, ..., 51, 46, 5],
 [8, 2, 6, ..., 12, 6, 5]], dtype=uint8)
```

查看 HDF5 的特征对应的标签：

```
b = h.root.HWDB10trn.writer007.labels.read().decode().split(' ')
```

```
b[:5]
```

输出结果为：

```
['昌', '屹', '亿', '役', '臆']
```

JSON 与 HDF5 各有各的特色，具体使用那一个可以自行选择。手写单字的手工特征已经解读完成，接下来的内容将着眼于其图片的解读。

## 6.4 手写单字的图片解读

手写单字的离线与在线的图片是分别以.gnt 与.pot 格式进行编码的。下面先看看离线的手写单字长什么样？

### 6.4.1 离线手写单字的图片解析

首先要获取离线手写单字的图片文件：

```
image_s = [f'{root}/{name}' for name in os.listdir(root) if 'gnt' in name] # 图片的源文件
image_s
```

输出结果：

```
['E:/OCR/CASIA/data/HWDB1.0trn_gnt.zip',
'E:/OCR/CASIA/data/HWDB1.0tst_gnt.zip',
'E:/OCR/CASIA/data/HWDB1.1trn_gnt.zip',
'E:/OCR/CASIA/data/HWDB1.1tst_gnt.zip']
```

先定义一个 gnt 解码器：

```
class GNT:
 # GNT 文件的解码器
 def __init__(self, Z, set_name):
 self.Z = Z
 self.set_name = set_name # 数据集名称
 def __iter__(self):
 with self.Z.open(self.set_name) as fp:
 head = True
 while head:
 head = fp.read(4)
 if not head: # 判断文件是否读到结尾
 break # 读到文件结尾立即结束
 head = struct.unpack('!I', head)[0]
 tag_code = fp.read(2).decode('gb2312-80')
 width, height = struct.unpack('2H', fp.read(4))
 bitmap = np.frombuffer(fp.read(width*height), np.uint8)
 img = bitmap.reshape((height, width))
 yield img, tag_code
```

选择 HWDB1.0trn\_gnt.zip 数据子集作为示范来说明 GNT 的使用：

```
Z = zipfile.ZipFile(f'{root}/HWDB1.0trn_gnt.zip')
Z.namelist()
```

输出结果：

```
['1.0train-gb1.gnt']
```

由输出结果知道 HWDB1.0trn\_gnt.zip 仅仅封装了一个数据'1.0train-gb1.gnt'，下面直接传入 GNT 类：

```
set_name = '1.0train-gb1.gnt'
gnt = GNT(Z, set_name)
for imgs, labels in gnt: # 仅仅查看一个字
 break
```

为了更加直观，引入可视化包：

```
%matplotlib inline
from matplotlib import pyplot as plt
这样便可以查看图片:
plt.imshow(imgs)
plt.title(labels)
plt.show()
```

输出结果，如图 6.3 所示。

```
C:\Users\xz\conda\envs\mxnet\lib\site-packages\matplotlib\backends\backend_agg.py:211: RuntimeWarning: Glyph 25212 missing from current font.
font.set_text(s, 0, 0, flags=flags)
C:\Users\xz\conda\envs\mxnet\lib\site-packages\matplotlib\backends\backend_agg.py:176: RuntimeWarning: Glyph 25212 missing from current font.
font.load_char(ord(s), flags=flags)
```

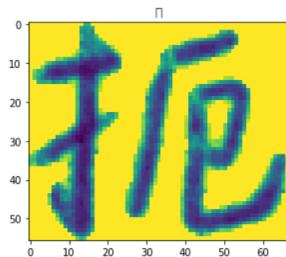


图 6.3 离线手写字示例图

可以看出，此时报错了，说缺少字体。实际上，这是 matplotlib 的默认设置不支持汉字，为了让其支持汉字，需要如下操作：

```
plt.rcParams['font.sans-serif'] = ['SimHei'] # 用来正常显示中文标签
plt.rcParams['axes.unicode_minus'] = False # 用来正常显示负号
```

接下来便可以正常显示了：

```
plt.imshow(imgs)
plt.title(labels);
```

显示截图，如图 6.4 所示。

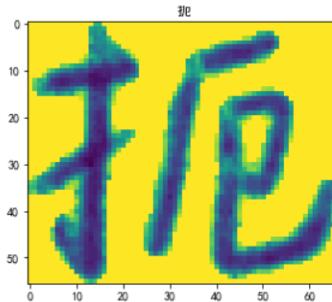


图 6.4 修正的离线手写字示例图

可以查看'1.0train-gb1.gnt'总有多少字符？

```
labels = np.asarray([l for _, l in gnt])
labels.shape[0]
```

输出：

```
1246991
```

故而，'1.0train-gb1.gnt' 总有 1246991 个字符，与官网提供的信息一致。

## 6.4.2 在线手写单字的图片解析

同样，需要获取在线手写单字的图片文件：

```
image_s1 = [f'{root}/{name}' for name in os.listdir(root) if 'pot' in name] # POT 图片的源文件
image_s1
```

输出结果为：

```
[E:/OCR/CASIA/data/OLHWDB1.0test_pot.zip',
'E:/OCR/CASIA/data/OLHWDB1.0train_pot.zip',
'E:/OCR/CASIA/data/OLHWDB1.1trn_pot.zip',
'E:/OCR/CASIA/data/OLHWDB1.1tst_pot.zip']
```

与 GNT 一样，先写一个 POT 的解码器：

```
class POT:
 # POT 解码器
 def __init__(self, Z, set_name):
 self.Z = Z
 self._fp = Z.open(set_name)
 def __iter__(self):
 size = struct.unpack('H', self._fp.read(2))[0] # Sample size
 tag = {} # 记录字符与笔画
 sizes = []
 tag_id = 0
 while size:
 sizes.append(size)
 tag_code = self._fp.read(4).decode(
 'gb18030').strip('\x00') # 字符解码
 stroke_num = struct.unpack('H', self._fp.read(2))[0] # 笔画数
 strokes = {k: [] for k in range(stroke_num)}
 k = 0
 while k <= stroke_num:
 xy = struct.unpack('2h', self._fp.read(4))
 if xy == (-1, 0):
 k += 1
 elif xy == (-1, -1):
 tag.update({tag_id:{tag_code: strokes}}) # 更新字典
 tag_id += 1
 size = self._fp.read(2)
 if size == b"": # 判断是否解码完成
 print('解码结束！')
 else:
 size = struct.unpack('H', size)[0] # Sample size
 break
 else:
 strokes[k].append(xy) # 记录笔迹坐标
 yield tag, sizes
```

以 OLHWDB1.1trn\_pot.zip 为例来说明如何使用 POT 类：

```
Z = zipfile.ZipFile(f'{root}/OLHWDB1.1trn_pot.zip')
ss = 0 # 统计该数据集总共的样本数目
Z = zipfile.ZipFile('E:/OCR/CASIA/data/OLHWDB1.1trn_pot.zip')
```

```
for set_name in Z.namelist():
 pot = POT(Z, set_name) # 实例化
 for tags, _ in pot:
 ... # 等价于 pass
 ss += len(tags.keys())
print(ss == 898573)
```

输出结果为：

```
True
```

即输出的结果符合官方提供的统计。更多的细节这里就不展开了，后面用到时再详细讲解。

## 6.5 本章小结

本章主要介绍了 CASIA 脱机与在线手写字符的人工特征解码以及其图片的解码。虽然，写了不少 API 作为解码器，但是仍然存在不少细节内容没有展开。本章这样做的目的是为了减少理解负担，在之后的章节若有涉及 CASIA 的数据集的使用，那时再展开将更容易理解。

# 第 7 章 COCO API 的使用

COCO 数据库是由微软发布的一个大型图像数据集，该数据集专为对象检测、分割、人体关键点检测、语义分割和字幕生成而设计。

本章快报：

- 介绍和使用官方 API：详细说明如何在 Linux 和 Windows 系统下使用 cocoapi。
- 改写官方 API：利用 Python 的特性对 API 进行改写，同时支持直接读取压缩文件。
- API 扩展：将 API 推广至其他数据集。

下面来探讨一下如何利用 Python 来使用 COCO 数据集。

## 7.1 COCO API 的配置与简介

如果你要了解 COCO 数据库的一些细节，可以参考：

- 我改写的 COCO API 网址：<https://github.com/Xinering/cocoapi>。
- 数据下载：<http://mscoco.org/dataset/#download>。

COCO API (<https://github.com/cocodataset/cocoapi>) 提供了 Matlab, Python 和 Lua 的 API 接口。该 API 接口提供完整的图像标签数据的加载，解析和可视化的工作。此外，网站还提供了与数据相关的文章、教程等。

在使用 COCO 数据库提供的 API 和 demo 之前，首先需要下载 COCO 的图像和标签数据：

- 图像数据下载到 coco/images/ 文件夹中。
- 标签数据下载到 coco/annotations/ 文件夹中。

为了方便操作，先 fork 官方 COCO API，然后下载到本地，并切换到 API 所在目录，如 D:\API\cocoapi\PythonAPI。直接在 Shell 中调取：

```
cd D:\API\cocoapi\PythonAPI
```

打开当前目录下的 Makefile 可以看到 API 的安装和使用说明。

### 7.1.1 Windows 的配置

Windows 中（一般需要安装 visual studio）有许多的坑，暴力删掉参数 Wno-cpp 和 Wno-unused-function 可以解决一部分问题，代码如下所示：

```
from distutils.core import setup
from Cython.Build import cythonize
from distutils.extension import Extension
import numpy as np
To compile and install locally run "python setup.py build_ext --inplace"
To install library to Python site-packages run "python setup.py build_ext install"
ext_modules = [
 Extension(
 'pycocotools._mask',
 sources=['./common/maskApi.c', 'pycocotools/_mask.pyx'],
 include_dirs = [np.get_include(), './common'],
 extra_compile_args=['-std=c99'],
)
]
setup(name='pycocotools',
 packages=['pycocotools'],
 package_dir = {'pycocotools': 'pycocotools'},
 version='2.0',
 ext_modules=
 cythonize(ext_modules)
)
```

这样，便可以在 Python 中使用 pycocotools，不过每次你想要调用 pycocotools 需要先载入局部环境，如下代码所示：

```
import sys
sys.path.append('D:\API\cocoapi\PythonAPI') # 将你的 `pycocotools` 所在路径添加到系统环境
```

如果你不想这么麻烦，你可以直接将 pycocotools 安装在你的主环境下，如下所示：

```
cd D:\API\cocoapi\PythonAPI
python setup.py build_ext install
rd build # 删除
```

但是，这样并没有解决根本问题，还有许多 bug 需要你自己调，因而在第 7.2 节介绍了 cocoapi 对 Windows 系统更加的友好实现。

## 7.1.2 Linux 下的配置

在 Linux 下，不需要上面这么多编译步骤，可以直接在终端输入下列命令即可正常使用 COCO API，代码入下所示：

```
pip3 install -U Cython
pip3 install -U pycocotools
```

当然，你也可以使用和 Windows 系统同样的处理方法，具体操作方法也可以参考 Makefile：<http://cocodataset.org/#format-data>。

## 7.2 改写 COCO API 的初衷

前文我一直在说 cocoapi Windows 系统不友好，相信在 Windows 系统下使用过 cocoapi 的朋友一定会十分赞同的。

### 7.2.1 Why? API 改写的目的

为了在 Windows 系统下更加友好的使用 cocoapi，抛去各种调 bug 的烦恼，十分有必要对 cocoapi 进行改写。但是，完全改写源码是有点让人感到恐惧的事情，而 Python 是一个十分强大的语言，利用它的继承机制可以无压力改写代码。

### 7.2.2 What? API 可以做什么

读者朋友是不是感觉改写 API 在做无用功，直接在 Linux 系统使用 cocoapi 也没有这么多的烦恼，为什么一定要改写？因为，改写后的 API 除了可以直接在 Windows 系统友好使用之外，它还提供了无需解压（直接跳过解压）直接获取标注信息和图片的功能。

### 7.2.3 How? API 如何设计

在 cocoapi 所在目录 D:\API\cocoapi\PythonAPI\pycocotools 下创建 cocoz.py 文件。下面来一步一步的填充 cocoz.py。为了方便调试，先在 Notebook 模式下设计该 API，设计好之后，再封装到 cocoz.py 文件中。为了令 cocoapi 可以使用，需要先载入环境，代码如下所示：

```
import sys
sys.path.append(r'D:\API\cocoapi\PythonAPI')
from pycocotools.coco import COCO
```

由于需要直接读取压缩文件，因而需要载入 zipfile，为了减少代码编写的工作量，下面直接借用 cocoapi 的 COCO 类。又因为标注信息是以.json 形式存储的，所以载入 json 也是必要的，而 numpy 和 cv2 处理图片数据的重要工具当然也需要，所涉及代码如下所示：

```
载入必备包
```

```
import os
import zipfile
import numpy as np
import cv2
import json
import time
```

为了更加方便的查看某个函数运行时间，同时需要一个计时器来看看效果，如下所示：

```
def timer(func):
 # 定义一个计时器，传入一个需要修饰的函数，返回附加了计时功能的另一种方法
 def wrapper(*args):
 start = time.time() # 开始计时
 print('Loading json in memory ...')
 value = func(*args)
 end = time.time() # 终止时间
 print('used time: {0:g} s'.format(end - start))
 return value
 return wrapper
```

我将 COCO 的所有数据都下载到了磁盘，可以通过如下代码查看：

```
root = r'E:\Data\coco' # COCO 数据根目录
dataDir = os.path.join(root, 'images') # 图片所在目录
annDir = os.path.join(root, 'annotations') # 标注信息所在目录
print('images:\n', os.listdir(dataDir))
print('*'*50) # 为了区分
print('annotations:\n', os.listdir(dataDir))
print(os.listdir(annDir))
```

输出结果如下所示：

```
images:
['test2014.zip', 'test2015.zip', 'test2017.zip', 'train2014.zip', 'train2017.zip', 'unlabeled2017.zip',
 'val2014.zip', 'val2017.zip']
=====
annotations:
['test2014.zip', 'test2015.zip', 'test2017.zip', 'train2014.zip', 'train2017.zip', 'unlabeled2017.zip',
 'val2014.zip', 'val2017.zip']
['annotations_trainval2014.zip', 'annotations_trainval2017.zip', 'image_info_test2014.zip',
 'image_info_test2015.zip', 'image_info_test2017.zip', 'image_info_unlabeled2017.zip',
 'panoptic_annotations_trainval2017.zip', 'stuff_annotations_trainval2017.zip']
```

从上代码可以看出，所有数据我都没有解压，下面将动手设计一个无需解压便可获取数据信息的接口。

### 7.3 ImageZ 的设计和使用

先设计一个用来处理 coco/images/ 文件夹下的图片数据集的类。dataType 可以是 'test2014', 'test2015', 'test2017', 'train2014', 'train2017', 'unlabeled2017', 'val2014', 'val2017'，代码如下所示：

```
class ImageZ(dict):
```

```

处理 images 下的压缩文件
def __init__(self, root, dataType, *args, **kwds): # root:: root 目录
 super().__init__(*args, **kwds)
 self.__dict__ = self
 self.shuffle = True if dataType.startswith('train') else False
 self.Z = self.__get_Z(root, dataType) # ZipFile 对象
 self.names = self.__get_names(self.Z)
 self.dataType = self.Z.namelist()[0]
@staticmethod
def __get_Z(root, dataType):
 # 获取 images 下压缩文件的文件名
 dataType = dataType + '.zip'
 img_root = os.path.join(root, 'images')
 return zipfile.ZipFile(os.path.join(img_root, dataType))
@staticmethod
def __get_names(Z):
 names = [
 name.split('/')[1] for name in Z.namelist()
 if not name.endswith('/')
]
 return names
def buffer2array(self, image_name):
 # 直接获取图片数据, 无需解压缩
 image_name = self.dataType + image_name
 buffer = self.Z.read(image_name)
 image = np.frombuffer(buffer, dtype='B') # 将 buffer 转换为 np.uint8 数组
 img_cv = cv2.imdecode(image, cv2.IMREAD_COLOR) # BGR 格式或者 Gray
 img = cv2.cvtColor(img_cv, cv2.COLOR_BGR2RGB)
 return img

```

代码这么长看着是不是有点懵，具体细节大家自己琢磨，接下来直接看看它有什么神奇之处，代码如下所示：

```

dataDir = r'E:\Data\coco' # COCO 数据根目录
dataType = 'val2017'
imgZ = ImageZ(dataDir, dataType)

```

由于 imgZ 继承自 dict，所以它拥有字典的几乎所有属性和功能，如下所示：

```
imgZ.keys()
```

输出结果如下所示：

```
dict_keys(['shuffle', 'Z', 'names', 'dataType'])
```

- names: 存储了 val2017.zip 的所有图片的文件名
- shuffle: 判断是否是训练数据集
- Z: ZipFile 对象，用来操作整个 val2017.zip 文件

还有一个实例方法 buffer2array 可以直接通过图片的文件名获取其像素级特征，如下代码所示：

```

fname = imgZ.names[77] # 一张图片的文件名
img = imgZ.buffer2array(fname) # 获取像素级特征

```

由于 img 是 Numpy 数组，这样就可以对其进行各种我们熟悉的操作，如图片 6.1 所示，所涉及的代码如下：

```
from matplotlib import pyplot as plt
plt.imshow(img)
plt.show()
```

输出如图 7.1 所示。

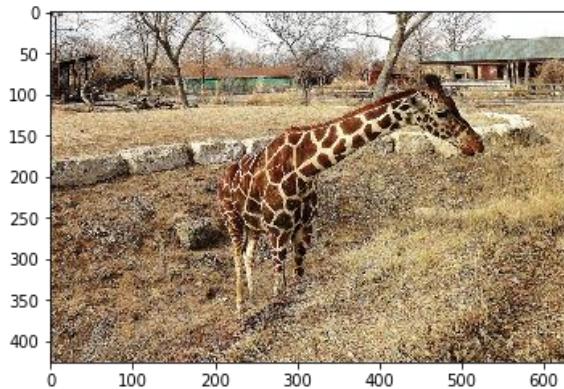


图 7.1 buffer2array 获取图片

至此，已经完成无需解压直接读取图片的工作。

## 7.4 AnnZ 的设计和使用

为了直接获取标注信息而不解压，可以设计如下代码：

```
class AnnZ(dict):# 处理 annotations 下的压缩文件
 def __init__(self, root, annType, *args, **kwds):
 #annType 可以是
 #'annotations_trainval2014','annotations_trainval2017','image_info_test2014',
 #'image_info_test2015','image_info_test2017','image_info_unlabeled2017',
 #'panoptic_annotations_trainval2017', 'stuff_annotations_trainval2017'
 super().__init__(*args, **kwds)
 self.__dict__ = self
 self.Z = self.__get_Z(root, annType)
 self.names = self.__get_names(self.Z)
 @staticmethod
 def __get_Z(root, annType):
 # 获取 ann 下压缩文件的文件名
 annType = annType + '.zip'
 annDir = os.path.join(root, 'annotations')
 return zipfile.ZipFile(os.path.join(annDir, annType))
 @staticmethod
 def __get_names(Z):
 names = [name for name in Z.namelist() if not name.endswith('/')]
 return names
 @timer
 def json2dict(self, name):
 with self.Z.open(name) as fp:
```

```
dataset = json.load(fp) # 将 json 转换为 dict
return dataset
```

直接调用代码如下所示：

```
root = r'E:\Data\coco' # COCO 数据集所在根目录
annType = 'annotations_trainval2017' # COCO 标注数据类型
annZ = AnnZ(root, annType)
```

来查看一下该标注数据所包含的标注种类：

```
annZ.names
```

输出如下所示：

```
['annotations/instances_train2017.json',
'annotations/instances_val2017.json',
'annotations/captions_train2017.json',
'annotations/captions_val2017.json',
'annotations/person_keypoints_train2017.json',
'annotations/person_keypoints_val2017.json']
```

下面以 dict 的形式载入 'annotations/instances\_train2017.json' 的具体信息：

```
annFile='annotations/instances_val2017.json'
dataset = annZ.json2dict(annFile)
```

输出：

```
Loading json in memory ...
used time: 1.052 s
```

可以查看 dataset 的关键字如下所示：

```
dataset.keys()
```

输出如下：

```
dict_keys(['info', 'licenses', 'images', 'annotations', 'categories'])
```

这样，可以很方便的使用 dict 的相关操作获取想要的一些信息，代码如下所示：

```
dataset['images'][7] # 查看一张图片的一些标注信息
```

输出结果如下代码所示：

```
{'license': 6,
'file_name': '000000480985.jpg',
'coco_url': 'http://images.cocodataset.org/val2017/000000480985.jpg',
'height': 500,
'width': 375,
'date_captured': '2013-11-15 13:09:24',
'flickr_url': 'http://farm3.staticflickr.com/2336/1634911562_703ff01cff_z.jpg',
'id': 480985}
```

可以利用 'coco\_url' 直接从网上获取图片，如下图 6.2 所示，代码如下：

```
from matplotlib import pyplot as plt
import skimage.io as sio
coco_url = dataset['images'][7]['coco_url']
use url to load image
I = sio.imread(coco_url)
plt.axis('off')
plt.imshow(I)
plt.show()
```

输出结果，如图 7.2 所示。



图 7.2 coco\_url 获取图片

借助 ImageZ 从本地读取图片 7.3，代码如下所示：

```
from matplotlib import pyplot as plt
imgType = 'val2017'
imgZ = ImageZ(root, imgType)
I = imgZ.buffer2array(dataset['images'][100]['file_name'])
plt.axis('off')
plt.imshow(I)
plt.show()
```

输出结果，如图 7.3 所示。



图 7.3 ImageZ 从本地读取图片

## 7.5 COCOZ 的设计和使用

ImageZ 和 AnnZ 虽然很好用，但是它们的灵活性太大，并且现在的开源代码均是基于 COCO 类进行设计的。为了更加契合 cocoapi 需要一个中转类 COCOZ 去实现和 COCO 几乎一样的功能，并且使用方法也尽可能的保留。具体是代码如下：

```
class COCOZ(COCO, dict):
 def __init__(self, annZ, annFile, *args, **kwds):
 # print(coco):: 预览 COCO 的 'info'
 super().__init__(*args, **kwds)
```

```

self.__dict__ = self # Bunch 结构
self.dataset = annZ.json2dict(annFile)
self.createIndex()
@timer
def createIndex(self):
 # 创建 index
 print('creating index...')
 cats, anns, imgs = {}, {}, {}
 imgToAnns, catTolmgs = {}, {}
 if 'annotations' in self.dataset:
 for ann in self.dataset['annotations']:
 imgToAnns[ann['image_id']] = imgToAnns.get(
 ann['image_id'], []) + [ann]
 anns[ann['id']] = ann
 if 'images' in self.dataset:
 for img in self.dataset['images']:
 imgs[img['id']] = img
 if 'categories' in self.dataset:
 for cat in self.dataset['categories']:
 cats[cat['id']] = cat
 if 'annotations' in self.dataset and 'categories' in self.dataset:
 for ann in self.dataset['annotations']:
 catTolmgs[ann['category_id']] = catTolmgs.get(
 ann['category_id'], []) + [ann['image_id']]
 print('index created!')
 # 创建类别
 self.anms = anns
 self.imgToAnns = imgToAnns
 self.catTolmgs = catTolmgs
 self.imgs = imgs
 self.cats = cats
 def __str__(self):
 """
 Print information about the annotation file.
 """
 S = [
 '{}: {}'.format(key, value)
 for key, value in self.dataset['info'].items()
]
 return '\n'.join(S)

```

具体如何使用 COCOZ，代码如下所示：

```

root = r'E:\Data\coco' # COCO 数据集所在根目录
annType = 'annotations_trainval2017' # COCO 标注数据类型
annFile = 'annotations/instances_val2017.json'
annZ = AnnZ(root, annType)
coco = COCOZ(annZ, annFile)

```

输出如下所示：

```

Loading json in memory ...
used time: 1.036 s

```

```
Loading json in memory ...
creating index...
index created!
used time: 0.421946 s
```

如果你需要预览你载入的 COCO 数据集，可以使用 print() 来实现代码如下所示：

```
print(coco)
```

输出代码如下所示：

```
description: COCO 2017 Dataset
url: http://cocodataset.org
version: 1.0
year: 2017
contributor: COCO Consortium
date_created: 2017/09/01
```

再次查看：

```
coco.keys()
```

输出：

```
dict_keys(['dataset', 'anns', 'imgToAnns', 'catToImgs', 'imgs', 'cats'])
```

### 7.5.1 展示 COCO 的类别与超类

一般地，对于图片分类和识别，需要考虑类别的分布问题。比如：有一个数据集拥有类别是“狗”和“猫”这两类，而另一个数据集拥有类别“黑狗”、“白狗”、“黑猫”和“白猫”。

如果直接建立一个模型来训练这两个数据集，很显然是不合理的。为此，可以将“狗”和“猫”的标签作为超类标签，而“黑狗”、“白狗”视为“狗”的子类；同理，“黑猫”和“白猫”视为“猫”的子类。

可以通过 loadCats 直接获取图片的类别信息代码如下所示：

```
cats = coco.loadCats(coco.getCatIds())
nms = set([cat['name'] for cat in cats]) # 获取 cat 的 name 信息
print('COCO categories: \n{}\n'.format(' '.join(nms)))
=====
snms = set([cat['supercategory'] for cat in cats]) # 获取 cat 的 name 信息
print('COCO supercategories: \n{}\n'.format(' '.join(snms)))
```

其中'name'代表图片的类别名字，'superCategory' 代表超类名字。输出结果如下所示：

```
COCO categories:
kite sports ball horse banana toilet mouse frisbee bed donut clock sheep keyboard tv cup elephant
cake potted plant snowboard train zebra fire hydrant handbag cow wine glass bowl sink parking meter
umbrella giraffe suitcase skis surfboard stop sign bear cat chair traffic light fork truck orange carrot
broccoli couch remote hair drier sandwich laptop tie person tennis racket apple spoon pizza hot dog
bird refrigerator microwave scissors backpack airplane knife baseball glove vase toothbrush book
bottle motorcycle bicycle car skateboard bus dining table cell phone toaster boat teddy bear dog
baseball bat bench oven
```

```
COCO supercategories:
```

```
animal kitchen food appliance indoor accessory person sports furniture outdoor electronic vehicle
```

## 7.5.2 通过给定条件获取图片

如何快速搜索数据集集中的数据是很关键的。由于数据量有点大，故而需要一个快速查找图片的方式。下面的代码提供了依据给定条件来获取包含给定类别下的所有图片，代码如下：

```
获取包含给定类别的所有图像, 随机选择一个
catIds = coco.getCatIds(catNms=['cat', 'dog', 'snowboard']) # 获取 Cat 的 Ids
imgIds = coco.getImgIds(catIds=catIds) #
img = coco.loadImgs(imgIds)
随机选择一张图片的信息
img = coco.loadImgs(imgIds[np.random.randint(0,len(imgIds))])[0]

img
```

输出代码如下所示：

```
{'license': 3,
'file_name': '000000179392.jpg',
'coco_url': 'http://images.cocodataset.org/val2017/000000179392.jpg',
'height': 640,
'width': 480,
'date_captured': '2013-11-18 04:07:31',
'flickr_url': 'http://farm5.staticflickr.com/4027/4329554124_1ce02506f8_z.jpg',
'id': 179392}
```

## 7.5.3 将图片的 anns 信息标注在图片上

为了提取图片是高级语义特征，比如将图片传入分割算法中来训练模型，训练好之后，又需要查看训练的效果如何，为此，需要一个工具来将图片的标注信息可视化。下面先从本地磁盘获取一张图片，如下图 7.4，代码如下：

```
from matplotlib import pyplot as plt
imgType = 'val2017'
imgZ = ImageZ(root, imgType)
读取图片
I = imgZ.buffer2array(dataset['images'][55]['file_name'])
可视化
plt.axis('off')
plt.imshow(I)
plt.show()
```

输出结果如下图 7.4 所示。



图 7.4 原图

将标注信息加入图片，如下图 7.5，代码如下：

```
载入和展示 annotations
plt.imshow(I)
plt.axis('off')
annIds = coco.getAnnIds(imgIds=img['id'], catIds=catIds, iscrowd=None)
载入标注信息
anns = coco.loadAnns(annIds)
coco.showAnns(anns)
```

输出结果，如图 7.5 所示。



图 7.5 将图片的 anns 信息标注在图片上

#### 7.5.4 关键点检测

关键点检测主要有人脸键点检测和人体骨骼关键点检测。接下来看看人体骨骼关键点检测

的一些简单操作，载入标注信息，代码如下所示：

```
initialize COCO api for person keypoints annotations
root = r'E:\Data\coco' # COCO 数据集所在根目录
annType = 'annotations_trainval2017' # COCO 标注数据类型
annFile = 'annotations/person_keypoints_val2017.json'

annZ = AnnZ(root, annType)
coco_kps = COCOZ(annZ, annFile)
```

输出如下代码：

```
Loading json in memory ...
used time: 0.924155 s
Loading json in memory ...
creating index...
index created!
used time: 0.378003 s
```

先选择一张带有 person 的图片，如下图 7.6 所示。

```
from matplotlib import pyplot as plt
catIds = coco.getCatIds(catNms=['person']) # 获取 Cat 的 Ids
imgIds = coco.getImgIds(catIds=catIds)
img = coco.loadImgs(imgIds)[99]
use url to load image
I = sio.imread(img['coco_url'])
plt.axis('off')
plt.imshow(I)
plt.show()
```

输出结果，如图 7.6 所示。



图 7.6 原图

将标注加到图片 7.7 上，代码如下：

```
load and display keypoints annotations
plt.imshow(I); plt.axis('off') # 关闭网格
ax = plt.gca()
annIds = coco_kps.getAnnIds(imgIds=img['id'], catIds=catIds, iscrowd=None)
anns = coco_kps.loadAnns(annIds) # 载入标注
coco_kps.showAnns(anns)
```

输出结果，如图 7.7 所示。



图 7.7 关键点标注

### 7.5.5 看图说话

还有一个比较有有意思的任务是给定图片，然后输出该图片的描述性文字，操作如下。  
载入标注信息，代码如下所示：

```
initialize COCO api for person keypoints annotations
root = r'E:\Data\coco' # COCO 数据集所在根目录
annType = 'annotations_trainval2017' # COCO 标注数据类型
annFile = 'annotations/captions_val2017.json'
annZ = AnnZ(root, annType)
coco_caps = COCOZ(annZ, annFile)
```

输出代码如下所示：

```
Loading json in memory ...
used time: 0.0760329 s
Loading json in memory ...
creating index...
index created!
used time: 0.0170002 s
```

将标注加到图片 7.8 上，如下所示：

```
A man riding on a skateboard on the sidewalk.
a kid riding on a skateboard on the cement
There is a skateboarder riding his board on the sidewalk
A skateboarder with one fut on a skateboard raising it up.
A pavement where a person foot are seen having skates.
```



图 7.8 字幕生成

至此，达到我们的预期目标。

## 7.6 让 API 更加通用

虽然完成了预期目标，但是 `cocoz.py` 还有很大的改进余地。比如可以令 `ImageZ` 变得像列表一样，支持索引和切片。为了优化结构，可以将其封装为生成器。基于这些想法便得到一个改进的 `ImageZ`，只需添加几个实例方法即可，代码如下：

```
def __getitem__(self, item):
 """令类对象支持索引和切片"""
 names = self.names[item]
 if isinstance(item, slice): # 切片
 return [self.buffer2array(name) for name in names]
 else:
 return self.buffer2array(names)
def __len__(self):
 return len(self.names)
def __iter__(self):
 for name in self.names:
 yield self.buffer2array(name)
```

详细代码被我放在：

<https://github.com/Xinering/cocoapi/blob/master/PythonAPI/pycocotools/cocoz.py>。

同时，为了令其他数据也可以使用 `ImageZ` 类，将 `ImageZ` 的输入参数改为 `images` 所在路径，其他不变。由于已经将上述的 `ImageZ`、`AnnZ`、`COCOZ` 封装进了 `cocoz.py`，所以如下代码，可以直接调用它们：

```
import sys
将 cocoapi 添加进入环境变量
sys.path.append('D:\API\cocoapi\PythonAPI')
from pycocotools.coco import AnnZ, ImageZ, COCOZ
为了避免重复，下面我们查看一张 train2017.zip 的图片：
dataDir = 'E:\Data\coco\images' # COCO 数据根目录
dataType = 'train2017'
```

```
imgZ = ImageZ(dataDir, dataType)
```

下面通过索引的方式查看，代码如下：

```
from matplotlib import pyplot as plt
img = imgZ[78]
plt.imshow(img)
plt.show()
```

显示如图 7.9 所示。

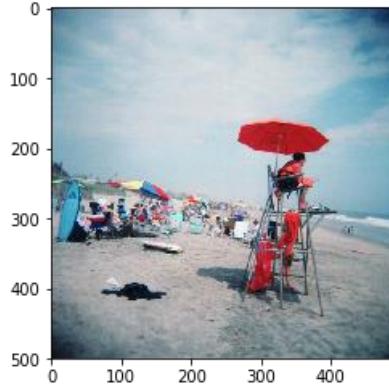


图 7.9 通过索引获取图片

也可以通过切片的方式获取多张图片，为了可视化的方便，先定义一个用来可视化的函数，代码如下所示：

```
from IPython import display
from matplotlib import pyplot as plt
def use_svg_display():
 # 用矢量图显示，效果更好
 display.set_matplotlib_formats('svg')
def show_imgs(imgs, is_first_channel=False):
 # 展示多张图片
 if is_first_channel:
 imgs = imgs.transpose((0, 2, 3, 1))
 n = len(imgs)
 h, w = 4, int(n / 4)
 use_svg_display()
 _, ax = plt.subplots(h, w, figsize=(5, 5)) # 设置图的尺寸
 K = np.arange(n).reshape((h, w))
 for i in range(h):
 for j in range(w):
 img = imgs[K[i, j]]
 ax[i][j].imshow(img)
 ax[i][j].axes.get_yaxis().set_visible(False)
 ax[i][j].set_xticks([])
 plt.show()
```

下面看看其中的 16 张图片，如图 7.10，代码如下：

```
show_imgs(imgZ[100:116])
```

显示结果如图 7.10 所示。

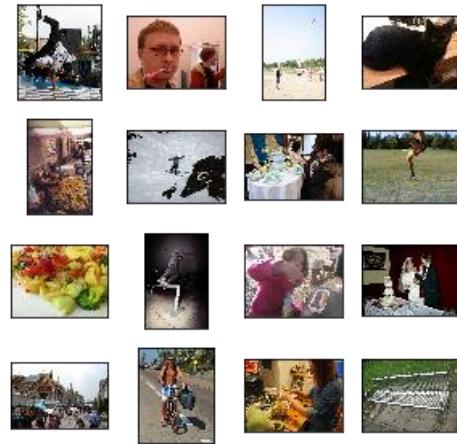


图 7.10 通过切片获取图片

到目前为止，都是仅仅关注了 COCO 数据集，而 ImageZ 不仅仅可以处理 COCO 数据集，它也能处理其它以.zip 形式压缩的数据集。比如 Kaggle 上的一个比赛 Humpback Whale Identification： <https://www.kaggle.com/c/humpback-whale-identification>，提供的关于座头鲸的数据集。该数据集大小为 5G 左右，如果直接解压然后再处理也是很麻烦，可以直接使用 ImageZ 来读取图片。

首先，先将下载好的 all.zip 进行解压：

```
import zipfile
import os
dataDir = r'E:\Data\Kaggle'
fname = 'all.zip'
with zipfile.ZipFile(os.path.join(dataDir, fname)) as z:
 z.extractall(os.path.join(dataDir, 'HumpbackWhale')) # 解压
```

解压好之后，查看一下该数据集的组成，代码如下所示：

```
dataDir = r'E:\Data\Kaggle\HumpbackWhale'
os.listdir(dataDir)
```

输出如下代码：

```
['sample_submission.csv', 'test.zip', 'train.csv', 'train.zip']
```

可以看出：该数据集有图片数据集 'test.zip' 与 'train.zip'。这样直接借助 ImageZ 来读取图片，如下代码所示：

```
dataDir = r'E:\Data\Kaggle\HumpbackWhale'
dataType = 'train'
imgZ = ImageZ(dataDir, dataType)
```

也看看其中的 16 张图片如图 7.11，代码如下所示：

```
show_imgs(imgZ[100:116])
```

显示如图 7.11 所示。

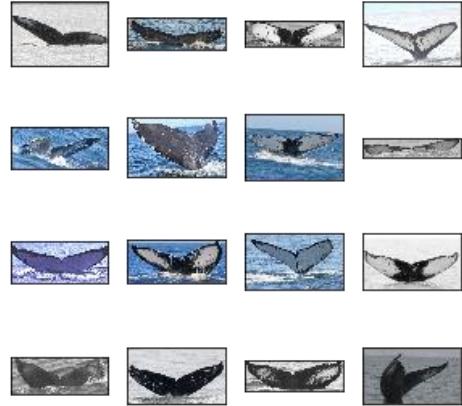


图 7.11 非 COCO 数据集使用 cocoz

## 7.7 本章小结

本章主要介绍了 COCO 数据及其 API cocoapi，同时为了更好的使用 cocoapi，又自制了一个可以直接读取.zip 数据集的接口 cocoz.py。同时，cocoz.py 也可以用来直接读取以 COCO 标注数据形式封装的其他数据集。

## 第三篇 工具篇

# 第 8 章 计算机视觉的项目管理工具 Git

一般地，学习深度学习技术需要具备以下能力：

- 编程语言：当前，大多数深度学习框架均以 Python 为主流编程语言。您可以查阅前面章节的 Python 知识清单简要了解 Python 的基础语法，如果您想要精通 Python 请阅读 Python 官方提供的中文教程：<https://docs.python.org/zh-cn/3/>。

- 版本控制：一个深度学习项目不是一天建成，需要不断的调试和修复，因而学习版本控制是十分重要的任务。本文仅仅介绍 Git。

- 解读和使用优质的 GitHub 资源：如果所有的项目都从零开始构建是完全没有必要的，将项目建立在优秀的 GitHub 项目的基础上，继续开发，将会为您节省大量的时间和经历。本文将详细介绍如何使用 GitHub 资源。

本章主要介绍如何使用项目管理工具 Git 来管理计算机视觉项目（当然包括深度学习项目）。

## 8.1 Git 基础简介

所谓的版本控制，就是用来某种工具记录文件内容的改变，方便未来查询或者恢复文件的内容的一种系统。版本控制的工具有很多，但是，我比较倾向于使用 Git。因为 Git 是分布式的，即使服务器上的版本库坏掉了，您也可以从其他非服务器的电脑上进行版本更新。

您完全不用担心文件的丢失或者损坏，版本控制工具 Git 已经帮你准备好了时光机器，借由该工具可以追溯文件的前世、今生以及未来，文件的时间轴完全由您掌控，甚至于，您可以借助 Git 的分支功能，跳跃不同的“平行宇宙”（同一个时间节点的不同改动）。

Git 数据库中保存的信息都是以文件内容的哈希值来索引，而不是文件名。Git 有三种状态：已提交（committed）、已修改（modified）和已暂存（staged）。

- **已提交**表示数据已经安全的保存在本地 Git 仓库目录（即 .git 目录，是 Git 用来保存项目的元数据和对象数据库的地方，可供其它计算机拷贝数据）中。可以看作是记录了一个时间节点的状态。

- **已修改**表示已经被 Git 管理的文件被修改了，但还没保存修改内容到 Git 仓库目录中。

- **已暂存**表示在工作目录（即您磁盘上存储的你看到的数据）对一个已修改文件的当前版本做了标记（即记录在 .git/index 中，一般地，将其称为暂存区域），使之包含在下次提交的快照中。

这些到底在说什么？您可能会有如此疑问，下面我们将详细了解这些概念。

### 8.1.1 配置用户信息

一般情况下，安装好 Git 之后需要配置用户信息，用于记录提交者的信息。下面的命令是针对您使用的机器的全局设置，如果您不需要针对全局进行设置，只需要删除参数--global 即可。

```
$ git config --global user.name 用户名
$ git config --global user.email 邮箱
```

配置好了用户信息，之后您便可以使用 Git 了。如果您不知道某一个命令如何使用，您可以使用 git help 来获取帮助，比如：

```
$ git help config
```

您便轻松的获取 git config 的离线帮助文档。

### 8.1.2 创建并操作一个本地仓库

创建一个本地仓库，只需要您在工作目录下使用命令 git init 即可。此命令会在工作目录的根目录下创建.git 目录，即为本地 Git 仓库详情，如图 8.1 所示。但是，此时，您的 Git 本地仓库是空的，我们需要将工作目录中的文件加入 Git 的跟踪。我们添加一个文件 A.md，并写入内容：# 测试，便会在 vscode 中看到图 8.2 的状态。我们可以看到画圈的部分，文件 A.md 被

标记为 U 表示该文件没有被 Git 跟踪管理，即不在 Git 的管理范围内。且左边的状态栏也显示了该 Git 仓库有一个文件被改动。为了让 Git 管理文件 A.md，我们需要执行命令：

```
$ git add A.md
```

The screenshot shows the Visual Studio Code interface. In the terminal tab, the command `git init` is run, resulting in the message: `Initialized empty Git repository in D:/study/git_study/.git/`. The file tree on the left shows a folder structure under `STUDY (工作区)`, including `study` and `git_study` folders, and a `.git` folder which is highlighted. A handwritten note "git 本地仓库" is written over the terminal area.

图 8.1 git init 初始化本地仓库

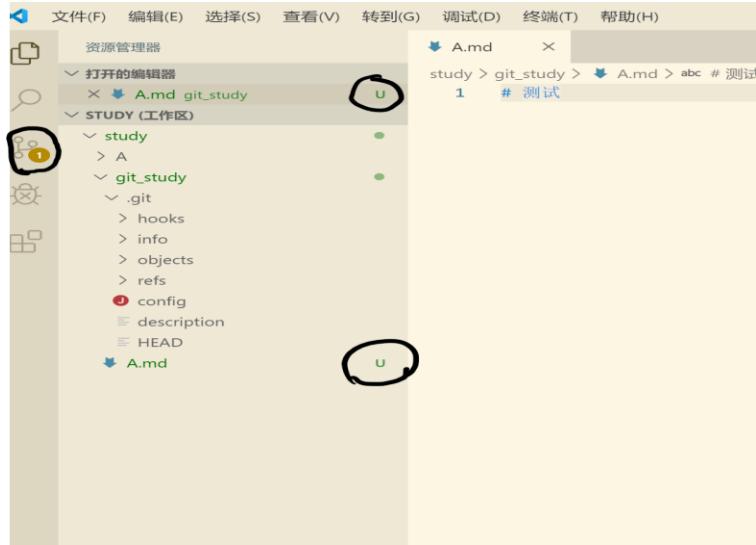


图 8.2 vscode 显示 git 的状态

该命令将文件 A.md 的内容添加到了 Git 仓库的索引区，即 `^.git/index` 文件之中。在 vscode 中的显示状态可如图 8.3 所示。

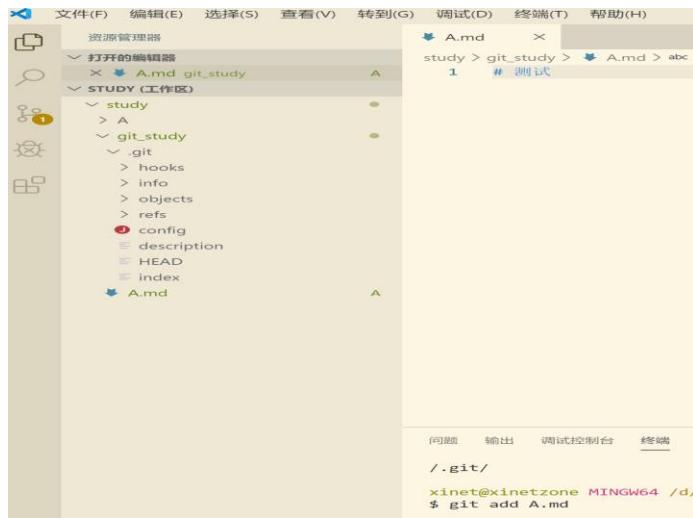


图 8.3 git add A.md 跟踪文件内容

从图 8.3 可以看出，文件 A.md 的状态由 U 变为 A，表示该文件被 Git 跟踪并添加到了索引区。并且与图 8.2 相比，Git 仓库在图 8.3 中多了一个文件 .git/index，该文件即为 Git 的索引区，也叫暂存区。如果工作目录中不仅仅有 A.md 一个文件，您可以使用命令 git add . 将工作目录的所有文件由未跟踪状态转换为跟踪状态，并存储在暂存区（注意，这里说“存储”并不准确，应该说是将文件的内容的指针放入 .git/index 中进行存储），以备下次提交。

您可以使用命令 git status -s 查看仓库中文件存储的状态，具体如图 8.4 所示。其中-s 是 short 的缩写，让 Git 显示关键的状态信息。图 8.4 中可以看出 A.md 的状态是在索引区。

```
$ git add A.md

xinet@xinetzone MINGW64 /d/study/git_study (master)
$ git status -s
A A.md
```

图 8.4 查看仓库文件状态

为了让 Git 仓库记录下 A.md 的内容，您需要将索引区的暂存提交到 Git 仓库之中，输入如下命令即可：

```
$ git commit -m "添加 A.md"
```

效果如图 8.5 所示，可以看到在左边的状态栏的标记都消失了，说明该仓库已经“干净了”。

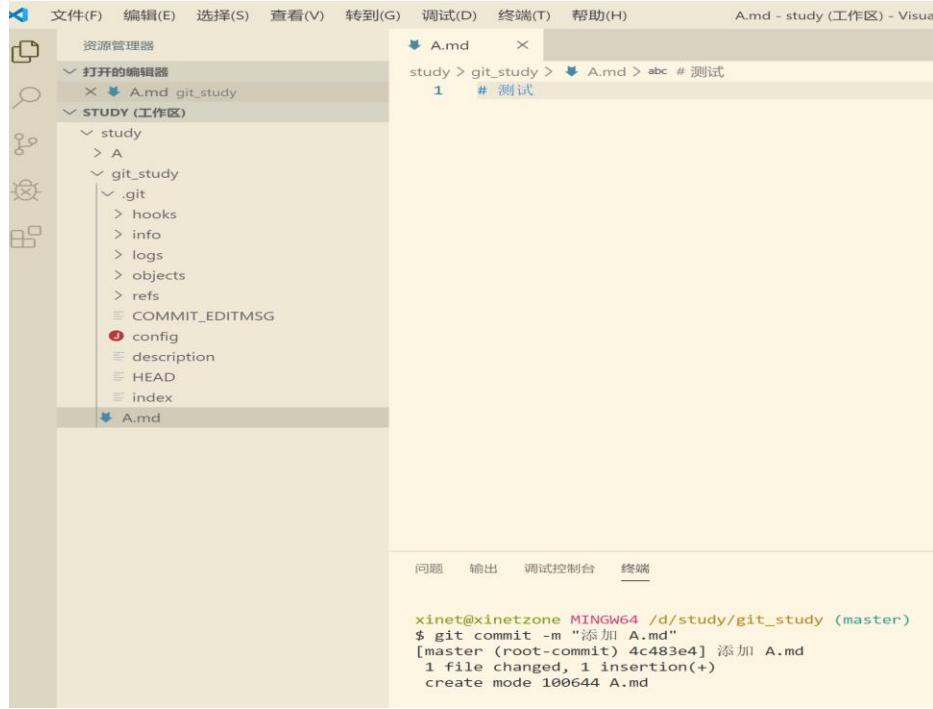


图 8.5 提交修改的内容到仓库

可以再次使用命令 `git status`, 查看此时仓库的状态, 如图 8.6 所示。可以看出此时的工作目录是干净的。

```
问题 输出 调试控制台 终端

xinet@xinetzone MINGW64 /d/study/git_study (master)
$ git status
On branch master
nothing to commit, working tree clean

xinet@xinetzone MINGW64 /d/study/git_study (master)
$
```

图 8.6 查看提交后的仓库的状态

下面我们修改 `A.md` 的内容, 即添加内容: 这是一个测试文件。此时可由图 8.7 看出文件 `A.md` 的状态显示为 `M`, 表示此时该文件有内容被修改 (`modified`) :



图 8.7 vscode 显示文件被修改的状态为 M

接着同样使用 git status 查看此时仓库的状态：

```
问题 输出 调试控制台 终端
xinet@xinetzone MINGW64 /d/study/git_study (master)
$ git status
On branch master
Changes not staged for commit:
 (use "git add <file>..." to update what will be committed)
 (use "git restore <file>..." to discard changes in working directory)
 modified: A.md

no changes added to commit (use "git add" and/or "git commit -a")
```

图 8.8 git status 查看修改的文件的状态

从图 8.8 可以看出此时工作目录的文件 A.md 被修改了。但是 git status 仅仅可以看出文件级别的（与暂存区的）不同状态，而如果您想要获悉文件的内容之间的不同，可以使用 git diff 获取工作目录中当前文件和暂存区域快照之间的差异：

```
问题 输出 调试控制台 终端
$ git diff
diff --git a/A.md b/A.md
index 9639844..4eb9f2e 100644
--- a/A.md
+++ b/A.md
@@ -1 +1,3 @@
-# 测试
\ No newline at end of file
+# 测试
+
+这是一个测试文件。
\ No newline at end of file
```

图 8.9 git diff 找不同

git diff 本身只显示尚未暂存的改动，而不是自上次提交以来所做的所有改动，若要查看已暂存的将要添加到下次提交的内容，可以用 git diff --staged 命令；命令 git diff --check 可以用来检测可能存在由空白符引起的问题；Git 的 ... 语法：git diff A...B 用于查找分支（之后再介绍）A 同 A 与 B 的共同跟节点的不同。如图 8.10 所示，简要的展示了 git diff A...B 命令的使用。

图 8.10 git diff A...B

当有一些文件不希望被 Git 追踪（比如一些机密文件），此时可以创建文件.gitignore 并将那些不希望被追踪的文件名称写入到.gitignore 中。

在 <https://github.com/github/gitignore> 中提供了一些.gitignore 模板可供参考，本章简要介绍文件.gitignore 的格式规范：

- 所有空行或者以 # 开头的行都会被 Git 忽略。
- 可以使用标准的 glob 模式匹配。
- 以 (/) 开头防止递归。
- 以 (/) 结尾指定目录。
- 惊叹号 (!) 取反要忽略指定模式以外的文件或目录。

当通过 git status 查看文件的状态为均已经暂存时，便可以使用 git commit -m "提交的信息" 命令将所有通过 git add 暂存的文件内容在数据库中创建一个持久的快照，然后将当前分支上的分支指针（即 HEAD）移到其之上。提交时记录的是放在暂存区域的快照，并以 SHA-1 校验码进行标识方便之后的版本转换。更多命令解读，如图 8.11 所示。

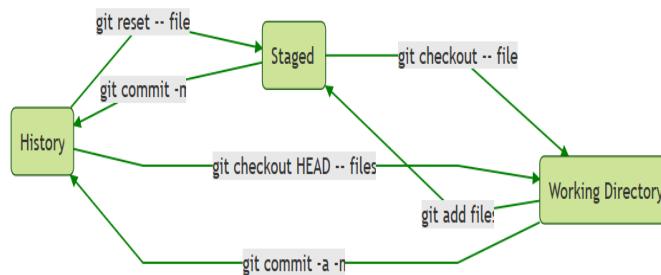


图 8.11 Git 常用命令

图中箭头指向表示文件的状态转换方向

如果已经做了多次提交，那么 Git 会将提交的顺序进行记录，通过 git log 命令可以查看提交历史。（更多关于 git log 的用法请参阅 <https://git-scm.com/book/zh/v2/Git-基础-查看提交历史>）。

每次提交均被 Git 以 SHA-1 码记录，这样一来，便可以依据其将工作目录恢复到某个过去的时间节点。假如，已经回到工作目录过去的某个时间节点，而又想回到未来（相对于当前时

间节点），那么，可以借助 git reflog 来查找“未来”的时间节点，然后使用 git checkout ID 进行穿梭（ID 指的是 SHA-1 码）。

工作目录的时间节点的状态是由 HEAD 进行指示的，故而，您也可以使用 git reset --hard commit\_id 不断的切换 HEAD 达到穿梭时间的效果。

有时候提交完了才发现漏掉了几个文件没有添加，或者提交信息写错了。此时，可以运行 git commit -a -m 提交命令尝试重新提交（其中 -a 表示 --amend）。

### 8.1.3 远程仓库

上一节已经了解了然后创建一个本地仓库，并对该本地仓库进行管理。本节我们探讨远程仓库（远程仓库是指托管在服务器或其他网络中的你的项目的版本库）。因为该仓库仅仅作为合作媒介，不需要从磁盘检查快照，所以一个远程仓库通常只是一个裸仓库（bare repository，一个没有当前工作目录的仓库，即工程目录内的 .git 目录）。

创建一个裸仓库，只需要：

```
$ git clone --bare my_project my_project.git
```

该命令实现了将 my\_project/.git 复制到 my\_project.git 目录中的作用，而 my\_project.git 便是一个裸仓库。为了让裸仓库发挥分布式的作用，需要将其放到服务器上并设置你的协议。其他拥有服务器的访问或读写权限的电脑将可以通过如下方式进行复刻：

```
$ git clone user@git.example.com:/opt/git/my_project.git
```

其中，user@git.example.com 代表服务器的地址，而 /opt/git/my\_project.git 代表服务器上裸仓库所在路径。本文不展开说明如何构建服务器，如果您想要了解构建服务器的详细信息，可以查看：服务器上的 Git - 在服务器上搭建 Git (<https://git-scm.com/book/zh/v2/服务器上的-Git-在服务器上搭建-Git>) 或者 GitBlit (<http://gitblit.com/>)。本文将使用目前最大的 Git 托管平台——GitHub 这一 Git 服务器（详细内容见 <https://git-scm.com/book/zh/v2/GitHub-账户的创建和配置>）。

假如从 GitHub 上搜寻到一个不错的仓库，想要将其加入到自己的项目中去，可以这样做，如图 8.12 所示。

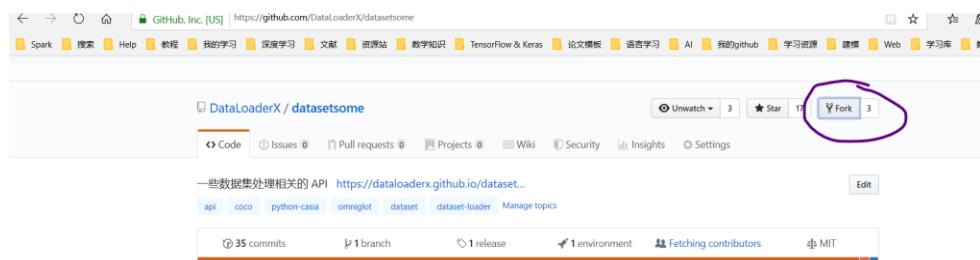


图 8.12 GitHub fork

点击右边的 fork 按钮，然后，选择您的用户名或者组织，将其复刻下来。这样，便有了对该仓库的读写以及推送（将本地仓库同步到远端仓库）的权限。使用 git clone 的命令将远端仓库克隆到本地，克隆的网址可以如图 8.13 所示获取。

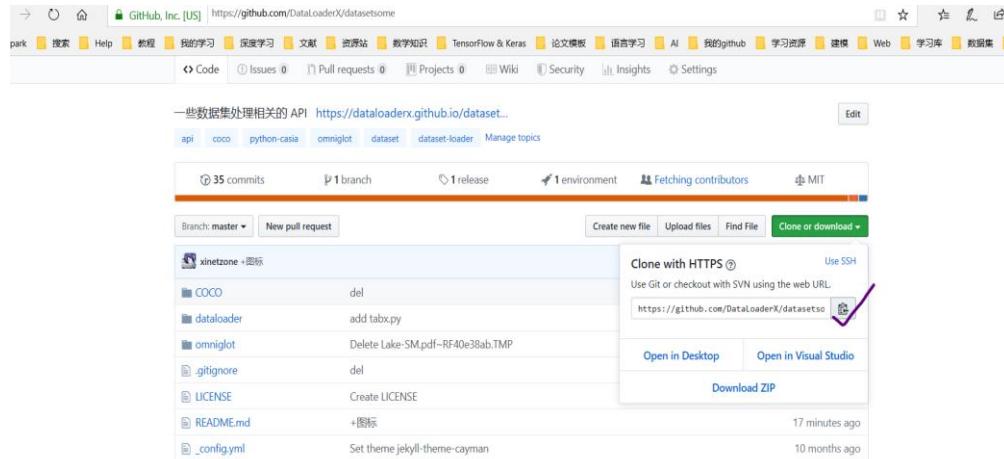


图 8.13 git clone

克隆到本地后，可以使用 `git remote -v` 命令查看远程仓库使用的 Git 保存的简写与其对应的 URL，比如：

```
origin https://github.com/xinetzone/xinet-matery.git (fetch)
origin https://github.com/xinetzone/xinet-matery.git (push)
```

其中 `origin` 便是别名。如果您想要拥有多个远程仓库，可以运行 `git remote add <shortname> <url>` 添加一个新的远程 Git 仓库，同时指定一个你可以轻松引用的简写。

为了方便说明我们创建两个本地仓库 `server/D.git` 与 `server/R.git`，然后在目录 T 中添加这两个仓库。目录 T 下有两个远程仓库，它们的别名分别为：D 与 R。现在可以在命令行中使用字符串 D 来代替整个 URL。例如，如果你想拉取 D 所指代的仓库，可以运行 `git fetch D`，这个命令会访问远程仓库，从中拉取所有你还没有的数据。执行完成后，你将会拥有那个远程仓库中所有分支的引用，接着，运行 `git merge D/master` 命令便可以将 D 合并到当前目录。

如果你使用 `git clone` 命令克隆了一个仓库，该命令会自动将其添加为远程仓库并默认以“`origin`”为简写。

如果在当前工作目录下做了修改，并且，想要将其分享，只需要执行 `git push [remote-name] [branch-name]` 命令即可。比如：

```
$ git push D master
```

如果您想查看远程仓库的更多详细信息，可以运行 `git remote show [remote-name]`；如果您想要重命名引用的名称可以运行 `git remote rename`；如果您想要删除远程仓库可以运行 `git remote rm` 或者 `git push origin --delete branch_name`。

#### 8.1.4 标签

在切换不同的时间节点时，可以借助 `commit` 的 ID，但是，此 ID 太长了并且不好记忆。为了人们可以更好的管理历史节点，我们需要给那些重要的历史节点打上标签，这样一来，人们通过这些有实际语义的标签进行管理将更加方便。

Git 主要使用两种类型的标签：轻量标签（lightweight）与附注标签（annotated）。

##### 1.附注标签

附注标签的创建可以这样:

```
$ git tag -a v1.4 -m "我的版本 1.4"
```

使用 `git tag` 或者 `git tag -l 'v1.4.1*'` 列出当前的标签。或者使用 `git show v1.4` 命令查看标签信息与其对应的提交信息。

## 2. 轻量标签

轻量标签不需要提供 commit 信息，只需要:

```
$ git tag v1.4-1q
```

亦可以对历史提交的某个 commit ID 进行打标，比如:

```
$ git tag -a v1.1 ead28ed
```

## 3. 管理标签

可以使用命令 `git push origin [tagname]` 将标签推送到远端；如果想要一次推送多个标签，可以是使用 `git push origin --tags`。

当然，也可以使用命令 `git tag -d <tagname>` 删除标签。如果也要将远端的标签删除，可以使用命令 `git push <remote> :refs/tags/<tagname>`，比如:

```
$ git push origin :refs/tags/v1.4-1q
```

## 8.1.5 分支

分支可以想象为不同维度的平行宇宙，在同一个时间节点可以并行的存在不同的支线来发展项目的不同功能。专业点的说法：Git 的分支仅仅是指向提交对象的可变指针，（Git 的默认分支名称为 `master`）每次进行 `git commit` 操作，都将会移动该指针。分支就好比河流拥有的支流，只不过在这些支流上面存在着无法移动的“指针”（`git tag`）与可以移动的“指针”（`git branch`）。

分支的创建需要使用 `git branch`，比如:

```
$ git branch develop
```

可以移动？那么我们该如何判断工作目录所处的状态呢？（简单点说，我们该如何判断我们“now”的位置。）这个很简单，Git 中还有一个特殊的指针 `HEAD`，它总是指向当前所在的分支所在的时间节点。您可以使用 `git log --oneline --decorate` 命令查看各个分支的指针。

您若要在不同的分支之间进行跳转，可以使用 `git checkout [分支名]` 进行切换。有没有命令将创建分支与切换分支进行合并的呢？当然有了，它就是：`git branch -b <分支名>`。

关于分支的命令还有：

- 合并: `git merge branch_name`
- 删除: `git branch -d branch_name`

## 8.1.6 vscode 与 Git 集成

为了让 Git 更好的与 vscode 集成，提供更丰富的 commit 信息，您可以这样:

```
$ git config --global core.editor "code --wait"
```

将 vscode 作为 Git 的内核编辑器。

## 8.2 使用 Git 管理项目

前面介绍了 Git 的基础知识，本文剩余部分将介绍如何利用 Git 更好的管理您的项目。

### 8.2.1 对 Microsoft Office 进行版本控制

最新版的 Git，已经支持对.docx 的控制。对于使用命令行，大多数人都是有点抵触的，好在有需求就有市场，在 Windows 系统下使用 Git，我们可以借助一个十分强大的 GUI 软件：TortoiseGit。

首先，我们需要下载并安装 TortoiseGit。下载网址：<https://tortoisegit.org/download/>，我们选择 64 位进行下载，并且需要下载中文语言包，如图 8.14 所示。

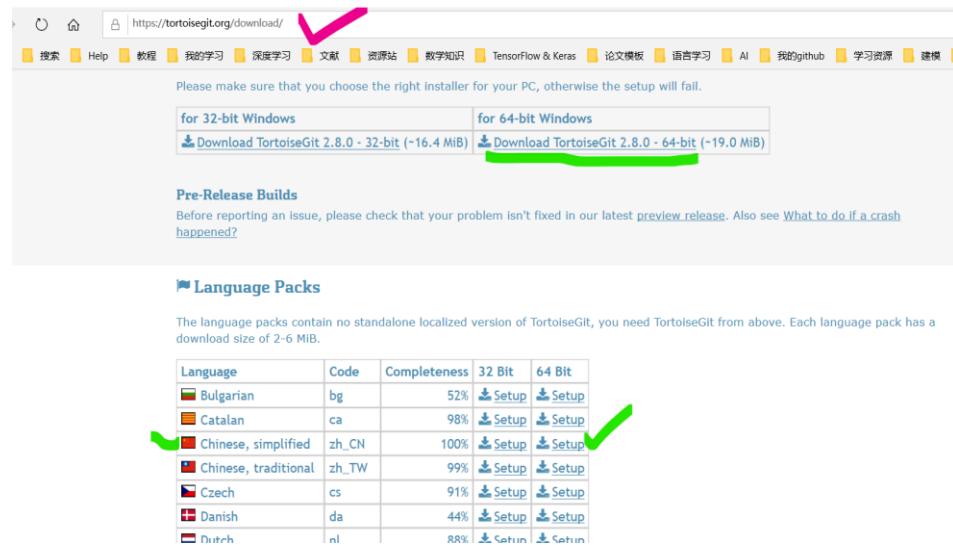


图 8.14 TortoiseGit 下载

下载好之后，按照提示默认安装即可，但是，安装中文包的最后一步需要如图 8.15 所示选择打勾。



图 8.15 安装 TortoiseGit 中文包

关于 TortoiseGit 的使用细节可参考其手册：<https://tortoisegit.org/docs/>。其实，TortoiseGit 被集成在了右键快捷键中，如图 8.16 所示。

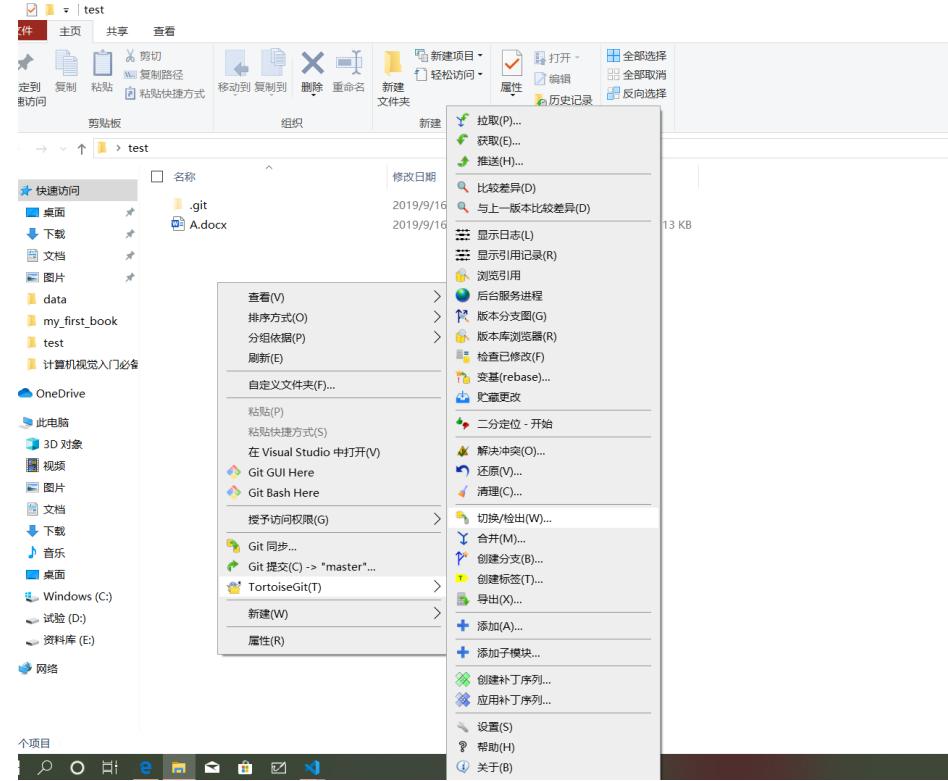


图 8.16 TortoiseGit 集成于右键

从图 8.16 可以清晰的看到 Git 的大部分功能都被集成到了右键快捷键中。使用 TortoiseGit 比较差异将比较简单，如图 8.17 所示。

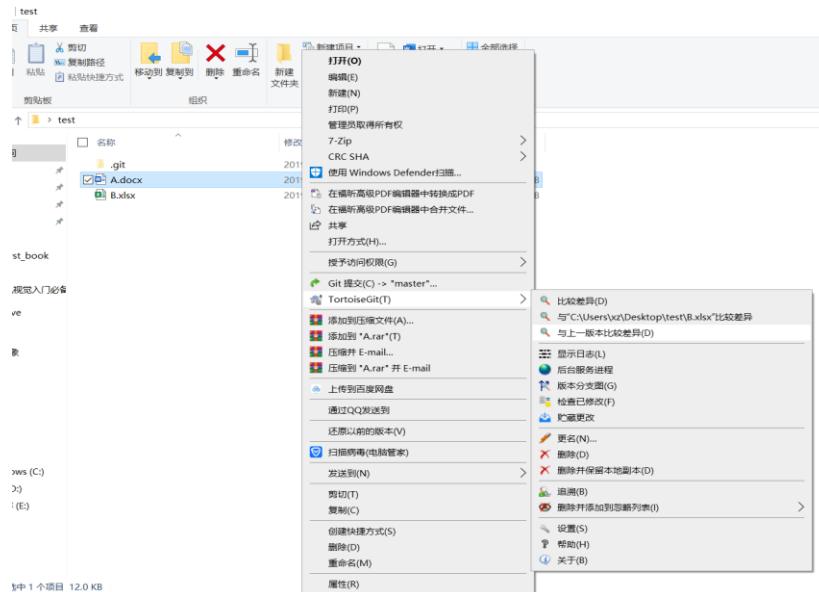


图 8.17 TortoiseGit 比较 docx

接着，将打开 Word，界面如图 8.18 所示。

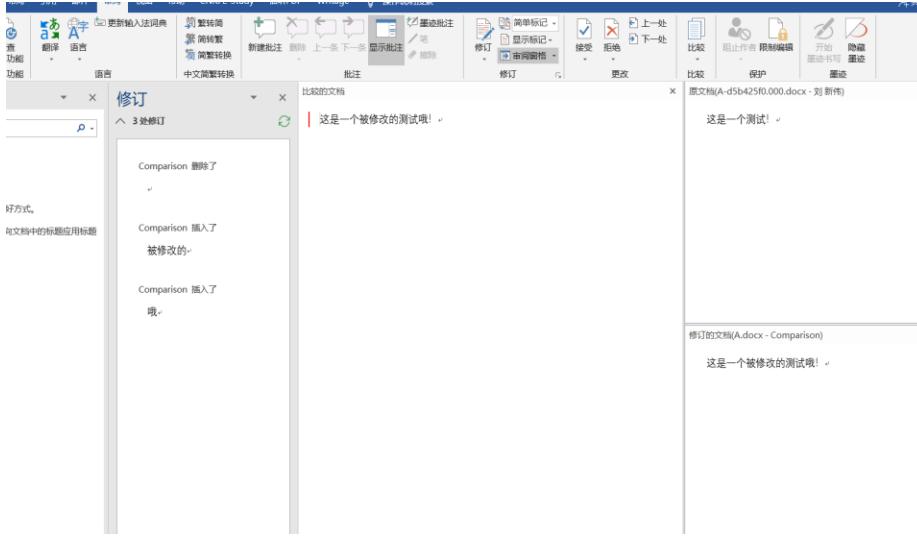


图 8.18 TortoiseGit 比较 docx 界面

这样的界面将会更加直观！

TortoiseGit 的强大不仅仅如此，它还可以直接比较 .xlsx、.pptx 等。换句话说，TortoiseGit 与 Microsoft office 完美的结合在了一起，为 office 管理提供了一个十分高大上的 Git 支持。

## 8.2.2 TortoiseGit 的使用

下面我们演示如何使用 TortoiseGit 操作项目？首先，从服务端（如局域网架设的 Git 服务器，GitHub 等）获取远程项目的地址：ssh://lxw@192.168.42.30:29418/test.git（此地址是我通过 GitBlit 架设的服务器创建的一个空的裸库）。其中，lxw 表示远程裸库的用户名，如图 8.19 所示。

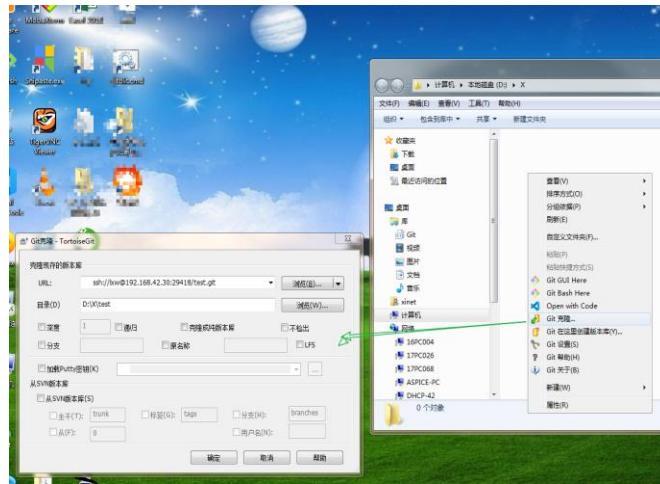


图 8.19 TortoiseGit clone

需要输入 lxw 对应的密码，如图 8.20 所示。

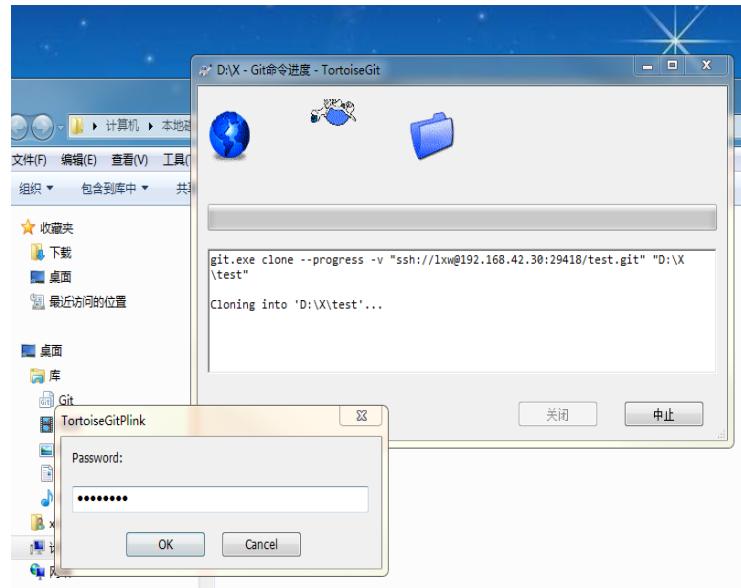


图 8.20 TortoiseGit clone 密码

因为是刚刚克隆（git clone）的仓库，其中的暂存区与工作目录均是干净的，所以，目录图标上会有蓝色的对勾，如图 8.21 所示。

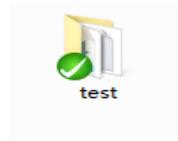


图 8.21 test

下面使用 vscode 打开工作目录 test 并创建一个 git bash 终端。该项目已经存在自述文档（README.md）与.gitignore（令 Git 忽视的文件列表）。为了更好的展示 TortoiseGit 与的集成效果，下面我们将演示如何利用 TortoiseGit 管理.docx 文档。

先创建 A.docx 并写入内容：我是 A。由于 A.docx 没有加入到 Git 管理，所以其图标为空，如图 8.22 所示。

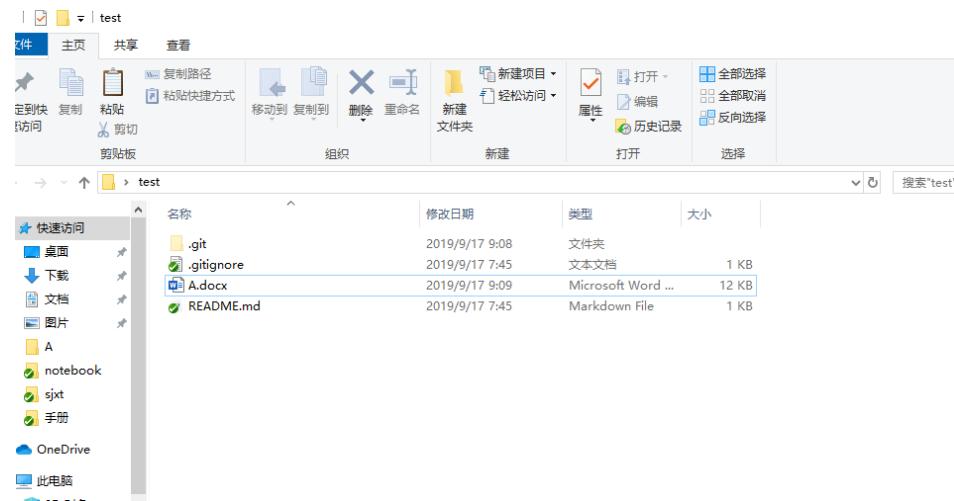


图 8.22 创建 A.docx

接着，将 A.docx 纳入 Git 管理，如图 8.23 所示。

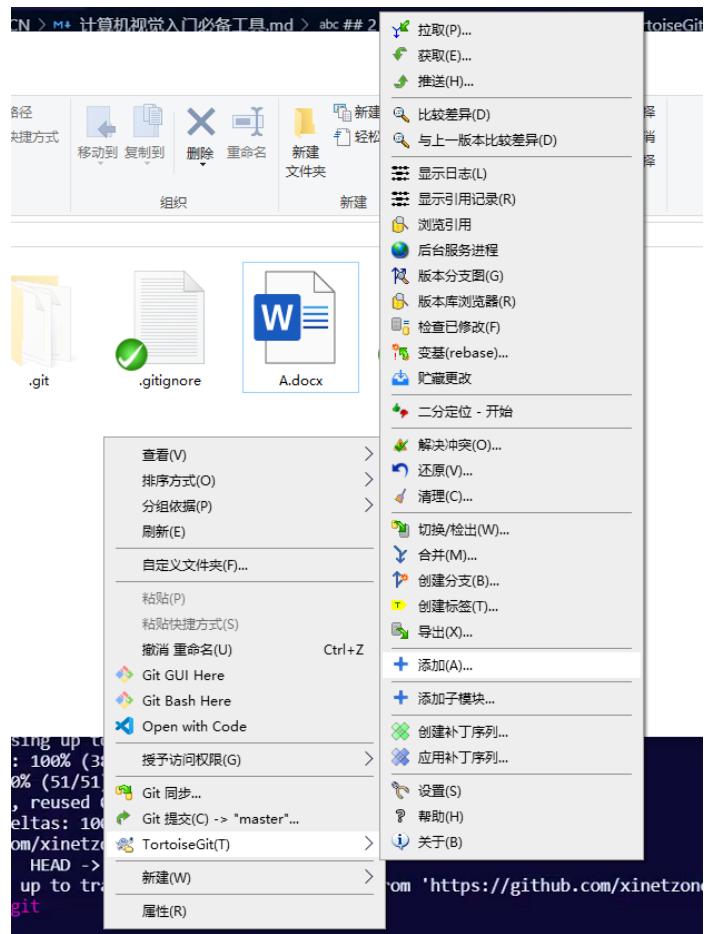


图 8.23 右键添加

接着弹出一个窗口，列出所有未被加入到 Git 管理的文件（即 git add 操作），如图 8.24-8.25 所示。

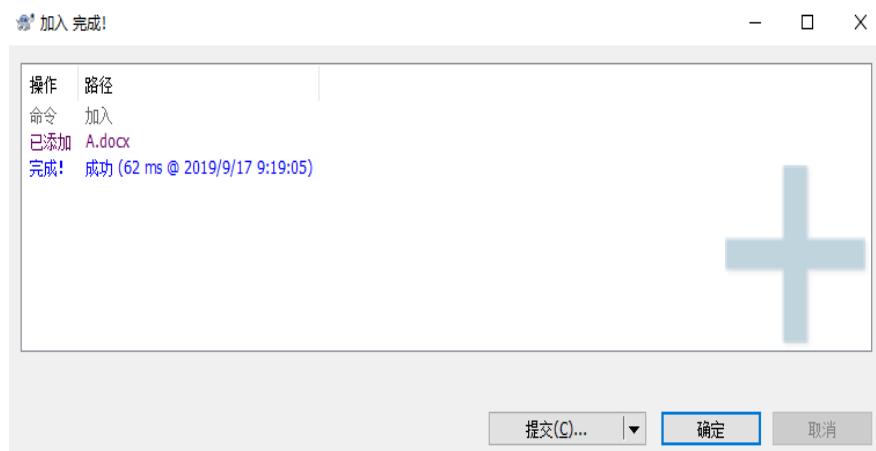


图 8.24 Tortoise add (一)

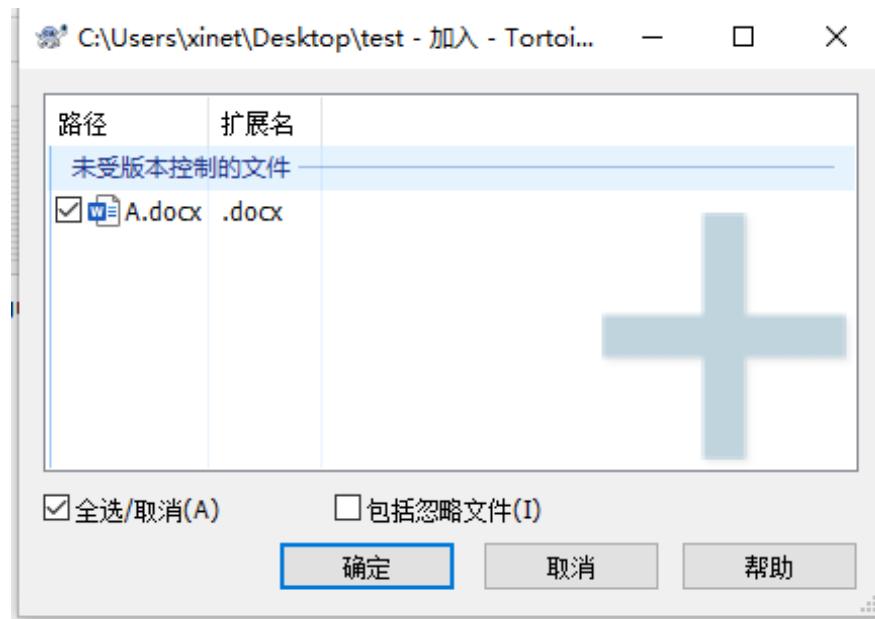


图 8.25 Tortoise add

自此，完成了将 A.docx 纳入 Git 的 index（即暂存区）的操作，刷新目录，可以看到 A.docx 多了一个蓝色的 + 图标，如图 8.26 所示。

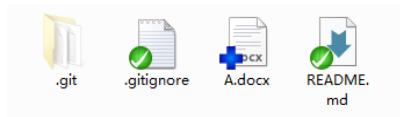


图 8.26 完成 git add

至此，如果没有其他需要加入到暂存区的文档，可以将这些暂存区的信息提交（git commit）到本地仓库，并写上简要的说明，如图 8.27-8.28 所示。



图 8.27 提交到本地 master

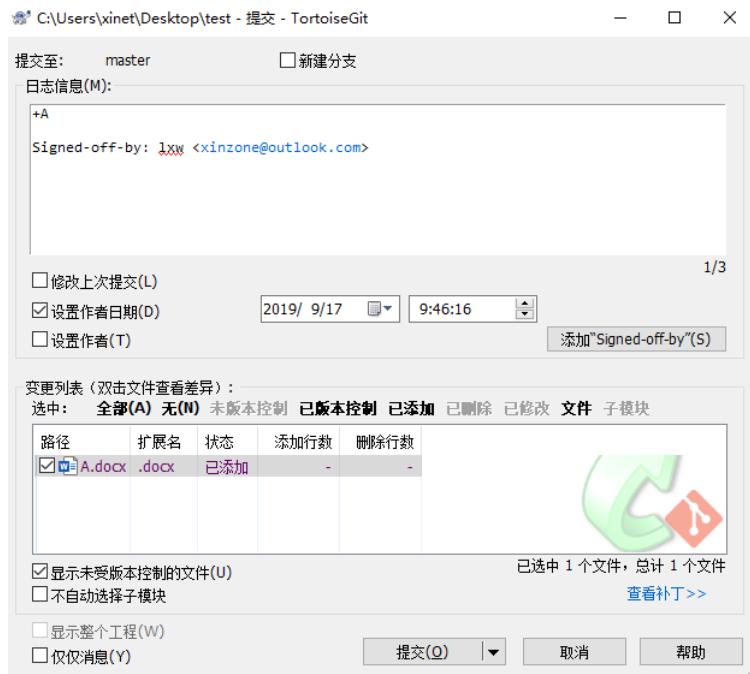


图 8.28 写上简要的说明

如果还想要与其他人共享编写 A.docx，需要将其推送（git push）到远端服务器，如图 8.29 所示。

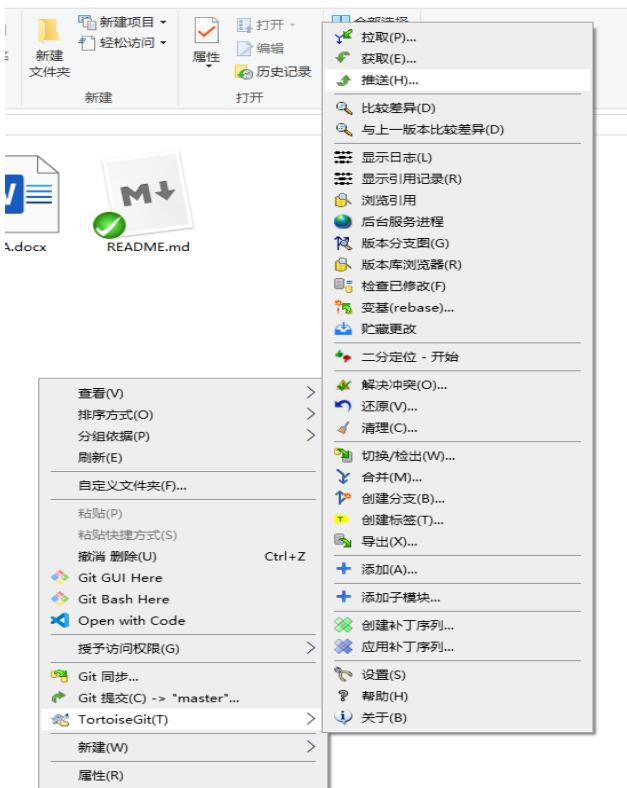


图 8.29 git push

其他人只需要在他们克隆的副本进行拉取（git pull）便可获得更新，如图 8.30 所示。



图 8.30 git pull

如果某人 K 拉取之后对 A.docx 进行了修改，添加内容：

做了 test1。

而自己也做了修改，添加内容：

学习了 M。

之后，先将变动推送到远端。紧接着 K 也将其做的改动推送到远端，由于你们所做的改动有冲突，所以推送失败，如图 8.31 所示。

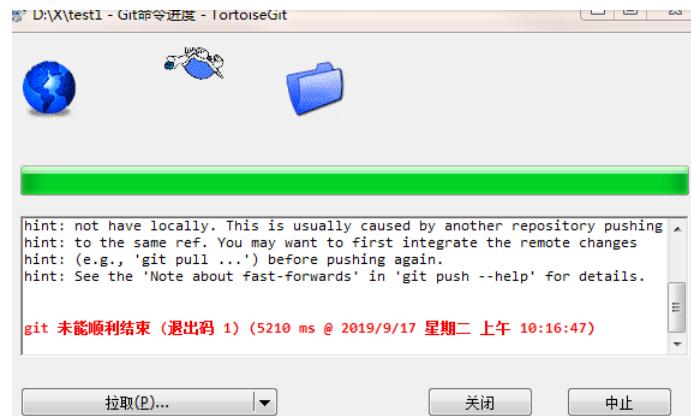


图 8.31 存在冲突

解决冲突的办法有两种：

- (1) 创建新的分支 (git branch) 与发生冲突的分支分开，解决冲突交给管理员（负责管理整个项目的人）；
- (2) 先将远端的更新获取 (git fetch) 后，如图 8.32 所示，之后再解决冲突。



图 8.32 git fetch

更多的关于 TortoiseGit 的使用可以参考其官方文档 <https://tortoisegit.org/docs/>，本文便不再引申。

### 8.3 本章小结

本章主要介绍学习计算机视觉需要准备的工具：Git 与 GitHub。但考虑到部分人不喜欢命令行操作，本章引入 Git 的 GUI 工具：TortoiseGit。需要学习 Git 是一个有点儿痛苦的任务，但是，如果您学会 Git，那么您会发现您将会离不开它，它为您的学习和工作带来极大的便利。

## 第 9 章 构建属于自己的计算机视觉开源项目

本章以 GitHub 作为 Git 的远端服务器来构建个人项目，而不考虑个人或者公司的私有服务器。但是如果私有服务器已经搭建好了，可以直接把它当作 GitHub 进行操作即可。因为，本文介绍的创建项目的方法同样适用于私有服务器，主要区别是提供的服务器的地址不同和仓库的使用权限不同而已。本节内容参考 GitHub 官方提供的开源项目的教程：<https://opensource.guide/zh-cn/>。

### 9.1 创建一个项目

(1) 在 GitHub (<https://github.com/>) 上创建一个仓库，取名为 cv-actions，接着选择一个 LICENSE (如 GNU GPL v3.0) 和 gitignore (选择 Python)。然后如图 9.1 所示，选择 Settings。

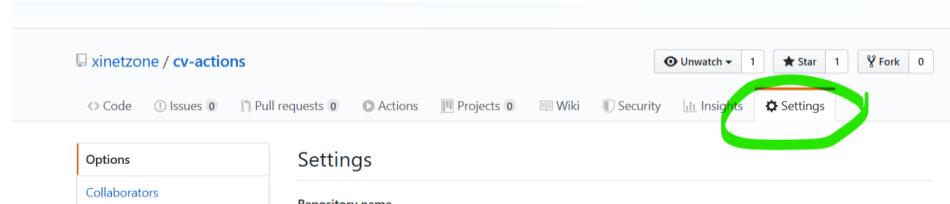


图 9.1 GitHub 的 Settings

接着在该页面往下滑动鼠标，找到 GitHub Pages，选择主题，随后便会生成一个网址，示例如图 9.2 所示。

## GitHub Pages

[GitHub Pages](#) is designed to host your personal, organization, or project pages from a GitHub repository.

✓ Your site is published at <https://xinetzone.github.io/cv-actions/>

### Source

Your GitHub Pages site is currently being built from the `master` branch. [Learn more](#).

`master branch` ▾

### Theme Chooser

Select a theme to publish your site with a Jekyll theme. [Learn more](#).

Your site is currently using the Cayman theme.

[Change theme](#)

图 9.2 GitHub Pages 用于创建项目网站

接着，点击图 9.3 中的 Edit，填写项目的简介：

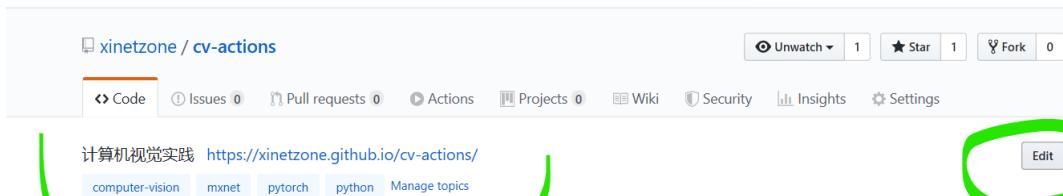


图 9.3 编写项目简介

至此，便完成了 GitHub 网站的初建。

(2) 将远端的项目克隆到本地。首先，按照图 9.4 所示的操作获取项目的远程仓库地址 (<https://github.com/xinetzone/cv-actions.git>)：

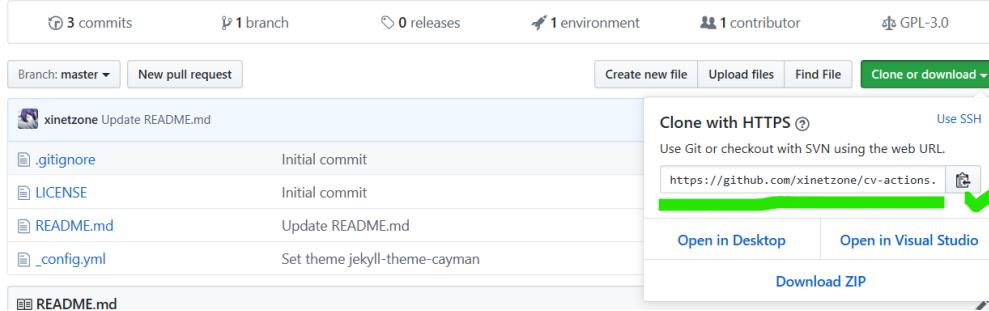


图 9.4 获取远程仓库地址

(3) 在本地电脑端打开 vscode 并在终端输入:

```
$ git clone https://github.com/xinetzone/cv-actions.git
$ cd cv-actions/
```

这样本地端便拥有一个 cv-actions 的副本。

## 9.2 修改 README

对于一个项目来说，README（自述文档）是十分重要的。自述文件不仅仅是用于展示说明人们如何使用你的项目，还解释了你的项目为什么重要，以及它可以为你的用户做什么。在自述文件中，尝试回答以下问题：

- What: 这个项目做什么？
- Why: 为什么这个项目有用？
- How: 如何开始？
- Help: 如果需要，我可以在哪里获得更多的帮助？
- Who: 谁维护和参与项目？

参考[@PurpleBooth](<https://github.com/PurpleBooth>) 的 README 模板<sup>1</sup> 编写属于自己项目的自述文档 README<sup>2</sup>。其中的图标可以参考 github 编写 README 时如何获取 fork 等图标 (<https://xinetzone.github.io/zh-CN/f7c6a6b8.html>) 进行设计。

接着创建一些目录，这些目录均代表项目的不同功能，即在根目录打开终端并输入：

```
$ mkdir data draft models outputs app notebook
```

- data/: 数据的存放
- models/: 模型的参数存储
- outputs/: 模型的输出结果
- app/: 存放相关的 API
- notebook/: 存放 jupyter notebook 等相关的文件
- draft/: (可以放置在任何位置)存放一些不成熟的或者未开发完成的一些相关内

<sup>1</sup> <https://gist.github.com/PurpleBooth/109311bb0361f32d87a2>

<sup>2</sup> <https://github.com/xinetzone/cv-actions/blob/master/README.md>

容，不被上传到 github

其中 data/, models/, draft/ 加入到 .gitignore 不被 git 管理。

进入项目的 Insights 栏目，选择 Community，可以看到如图 9.5 所示的结果。

The screenshot shows the GitHub Project Insights interface for the 'cv-actions' repository. On the left, there's a sidebar with links like 'Community', 'Traffic', 'Commits', 'Code frequency', 'Dependency graph', 'Network', and 'Forks'. The main area is titled 'Checklist' and contains a table with rows for 'Description' (green checkmark), 'README' (green checkmark), 'Code of conduct' (yellow dot), 'Contributing' (yellow dot), 'License' (green checkmark), 'Issue templates' (yellow dot), and 'Pull request template' (yellow dot). Each row has an 'Add' button on the right.

图 9.5 项目配置检查列表

图 9.5 列出了一个比较规范的项目需要满足的文件的清单。本文将逐渐填充检查清单。

### 9.3 编写贡献者指南

编写 CONTRIBUTING.md（贡献者指南）文档有助于贡献者们快速了解如何加入本项目的贡献之中。下面介绍如何编写贡献者指南？

一般地，使用热情友好的语气并提供具体的贡献建议可以大大提高新人的参与度。比如，可以在开头这样引入：

首先，非常感谢您为 cv actions 提供新鲜血液。正是更多像您这样的人，使得 cv actions 将逐渐发展为一个强大的计算机视觉社区。

#### 9.3.1 准备工作 1：创建 Issue 与 PR 模板

对于一个比较健全的社区，需要拥有一个 Issue templates，如图 9.5 所示。它可以令 Issue 的参与者有着一个统一而清晰的框架，同时，也方便其他人参与 Issue 的讨论之中。Issue templates 的创建可以直接从 GitHub 中需要选择一个模板，之后点击绿色按钮 Propose changes 提交修改即可，如图 9.6 所示。

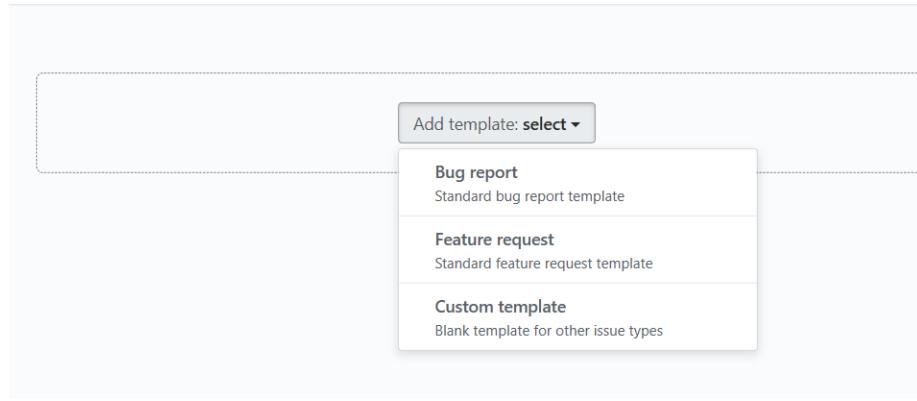


图 9.6 添加 Issue templates

当然，也可以添加多个模板甚至自定义模板具体细节见如下网站：  
<https://help.github.com/cn/articles/creating-a-pull-request-template-for-your-repository>。

这些模板仅仅是一个雏形，还需要与后续的工作进行匹配。所以，刚开始先不用管这么多，先生成这些模板再说。如果需要创建有多个作者的提交，可以参考  
<https://help.github.com/cn/articles/creating-a-commit-with-multiple-authors>。

还需要在`./.github`下创建文档 `PULL_REQUEST_TEMPLATE.md` 用于 Pull request（拉取请求，PR）的生成。因为编写良好的 PR 说明可以加速审阅者了解代码的预期结果的进程。它们也是帮助跟踪并应对每次的 commit（如测试、添加单元测试和更新文档）应执行的工作的好方法。更多优秀模板参考：<https://github.com/XNoteW/github-issue-templates>。下面以一个简单的例子来说明 PR 模板的样式：

非常感谢您参与到 cv actions 的项目中来，但为了创建一个良好的社区环境，在提交 PR 之前，请确保已经满足下列要求：

- [ ] code 是否已经还存在错误或者警告信息
- [ ] 正在使用的术语（terminology）是否是社区约定的或者已经被公认的
- [ ] 是否已经做过单元测试（unit tests）

一个相对优秀的 PR 应该包含如下问题：

- 做了什么改变？
- 修复了什么问题？
- 你发起的是一个 bug 修复(bug fix) 还是创建了一个新的功能？对旧的版本是否有重大影响？对已经存在的函数或者模块是否进行重构？
- 是如何验证它的？

更多关于 PR 的细节可参考：

[https://github.com/embeddedartistry/templates/blob/master/oss\\_docs/PULL\\_REQUEST\\_TEMPLATE.md](https://github.com/embeddedartistry/templates/blob/master/oss_docs/PULL_REQUEST_TEMPLATE.md)。

### 9.3.2 准备工作 2：编写行为准则

一个社区想要可持续地且健康地发展需要拥有一份约定的行为准则：`CODE_OF_CONDUCT.md`。顾名思义，行为准则即是参与社区时的一些礼仪、说话方式、行

为等，它帮助社区形成一种友好的氛围，且在某种程度上它也是展示项目对于贡献者的友好程度。

当前已经不需要我们自己创建行为准则，在贡献者公约(<https://www.contributor-covenant.org/>)之中已经提供了一份行之有效的行为规范，已经被用在包括 Kubernetes, Rails, 以及 Swift 等超过 4000 个开源项目(<https://www.contributor-covenant.org/adopters>)之中。参考图 9.5 点击 Add Code of conduct，便会弹出图 9.7 的结果：

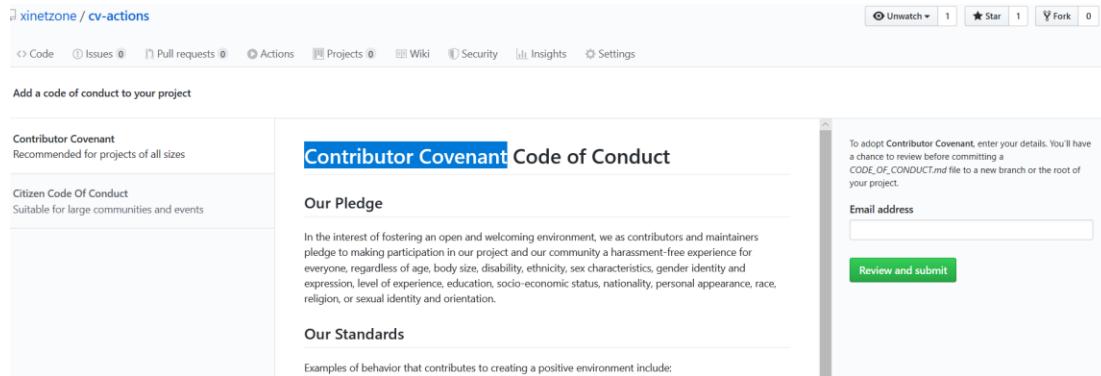


图 9.7 添加行为准则

选择贡献者公约（Contributor Covenant）并在右侧填写电子邮箱用于处理违反公约的行为和反馈。最后，提交修改即可。

### 9.3.3 贡献指南编写细则

考虑到项目的可读性，需要设计一些贡献指南编写的细则。如果您想要获得更多有关编写贡献指南的方式，可以查阅 @nayafia 的贡献指南模板或者 @mozilla 的“如何构建 CONTRIBUTION.md”。下面给出 cv actions 的贡献指南：

```
贡献指南
非常感谢您乐意为 cv actions 提供新鲜血液。正是更多像您这样的人，使得 cv actions 将逐渐发展为一个强大的计算机视觉社区。

为了创建一个良好的社区环境，请您遵守我们社区的《[行为准则](CODE_OF_CONDUCT.md)》。如果您想要发起一个 Bug report 或者 Feature request，请您遵循 Issue 的模板进行贡献。同时也欢迎您贡献新的且实用的 Issue 与 Pull request templates。

请您仔细阅读这份指南，因为，如果您这样做了，既可以节省您的宝贵时间，同时也对其他贡献者或者审阅者们的极大的尊重。

本项目的**目标**是**打造一个健康且可持续发展的计算机视觉社区。**

贡献的类型

本项目不仅仅欢迎您贡献代码，同时也欢迎您贡献文档（请在 `/docs/` 中进行），教程或者本项目的使用手册等内容。
```

```
贡献的必要条件
1. 如果您不了解如何对项目进贡献, 请您参考 [如何为开源做贡献](https://opensource.guide/zh-cn/how-to-contribute/)。
2. 为了方便项目的管理, 请您使用 Git Flow 进行管理。当您将项目克隆到您的本地电脑后, 请您运行: `git flow init` 并随之切换到 develop。具体是使用细节请 https://xinetzone.github.io/projects/。
3. 本项目暂定以 Python 作为代码的开发语言, 而文档的编写请使用 Markdown 进行工作。
4. 本项目在 GitHub 上维护的 master 分支作为预发布版本, 而 develop 分支作为开发版本。其他功能版本请以 git flow 的规则进行管理。
5. 您贡献的 Python 代码需要支持 pep8, 数学公式最好使用 markdown 的标准语法, 比如 `\$x^2=1` 等。
Issue 与 Pull request 的行为准则
考虑到可读性与可理解性, 在您创建 Issue 或者 Pull request (简写为 PR) 时, 请遵循如下约定:
1. **请给出您的 issue 或者 PR 的 context**: 即给出上下文语境。比如当您运行程序时遇到一个错误, 请解释你是如何操作的, 并描述该错误现象是否具有再现性; 当您提交一个新的 idea, 请您解释该 idea 您是如何想到的, 并且说明您的 idea 的适用范围, 是否可以将您的 idea 进行推广或者拓展。
2. **您是否已经完成准备工作**: 请确认您是否阅读了项目的 README、文档、问题 (开放的和关闭的), 检查您的 issue 或者 PR 是否已经有人做过或者正在做, 避免提交重复的议题或者请求。
3. 请您**保持请求内容的短小而直接**: 每个人的时间和精力都是有限的, 短小而直接的请求对大家都有益处。
4. **请保持您的优雅**: 本社区禁止发布不良信息, 注意用语文明而优雅, 杜绝各种辱骂和打口水仗的行为。
获得联系
- 您可以在 Gitter @[cv-actions](https://gitter.im/cv-actions/community) 上讨论 Issue
- 如果有疑问, 请您 xinzone@outlook.com
TODOS
- [] 利用 [pre-commit](https://pre-commit.com/) 自动部署代码的格式, 使其自动化支持 pep8 等格式。
```

至此, 我们便在 GitHub 上完成了项目的构建工作。

## 9.4 使用 git submodule 管理子项目

参考资料: <https://git-scm.com/book/en/v2/Git-Tools-Submodules>。

情景: 在正在维护的项目 (可记为 A) 中存在一些子项目或者第三方库 (可记为 B)。你不想在 A 中直接维护 B, 希望 B 作为一个单独的项目进行管理, 这种情况下便需要借助 Git 子模块工具了。Git 子模块工具将 B 作为 A 的一个子目录进行管理, 使用 git submodule 来实现。下面以一个例子来说明如何使用子模块的。

前提: 在 Github 上单独创建一个独立的项目 xinetzone/image, 该项目是一个数字图像处理的工具包 (细节见 <https://xinetzone.github.io/image/>)。

目标: 在 cv-actions 项目中添加子项目 image。

(1) 克隆 [cv-actions](#) 到个人电脑。即在终端输入:

```
git clone https://github.com/xinetzone/image.git
```

(2) git glow 初始化, 即在终端输入如下命令:

```
$ cd cv-actions
```

```
$ git flow init
```

(3) 使用 `git flow` 创建并切换到新的分支 `feature/image`, 即在命令行输入:

```
$ git flow feature start image
```

(4) 使用 `git submodule` 添加子项目 `cv-actions` 到目录 `app/` 之下, 即在命令行输入:

```
$ cd app/
```

```
$ git submodule add https://github.com/xinetzone/image
```

其中 `https://github.com/xinetzone/image` 是子项目 `image` 的网址。

可以看到此时在 `app/` 下新增目录 `image`, 且在根目录新增文件 `.gitmodules`。

其中 `.gitmodules` 文件记录了配置文件保存的项目 `image` 的 URL 与已经拉取的本地目录之间的映射:

```
[submodule "app/image"]
path = app/image
url = https://github.com/xinetzone/image
```

如果有多个子模块需要被管理, `.gitmodules` 文件中就会有多条记录。虽然 `image/` 是 `cv-actions` 的工作目录的一个子目录, 但是, `git` 并不会将其视作普通文件进行管理, 而是将其作为一个整体当作子模块中的某个具体的提交进行管理。

(5) 提交更改。为了让子模块 `image` 加入到 `git` 的历史 (`commit`) 之中, 您需要将其提交:

```
$ cd .. # 回到根目录
```

```
$ git commit -am '添加子模块 image'
```

其中 `-am` 是 `-a -m` 的组合。

(6) 推送项目到远端分支 `feature/image`。

```
$ git push origin feature/image
```

## 9.5 克隆含有子模块的项目

克隆含有子模块的项目一般有两种方法:

方法 1: `git clone URL`

方法 2: `git clone --recursive URL`

下面先讨论如何使用方法 1 克隆含有子模块的项目, 首先使用语法 `git clone -b 分支名 URL` 克隆项目的指定分支。在终端运行:

```
$ git clone -b feature/image https://github.com/xinetzone/cv-actions
```

```
$ cd cv-actions/
```

此时, 进入 `app/image` 会发现其是一个空目录, 这是因为含有子模块的项目需要进行初始化。初始化需要两步, 具体操作是在 `app/image` 下面运行:

```
$ git submodule init
```

```
$ git submodule update
```

其中 `git submodule init` 用于初始化本地配置, 而 `git submodule update` 用于从项目 `image` 中抓取所有数据并检出 (`checked out`) 父项目中列出的合适的提交。

对于方法 2, 仅仅运行如下命令即可达到上述的同样效果:

```
$ git clone --recursive -b feature/image https://github.com/xinetzone/cv-actions
```

## 9.6 使用子模块

如果子模块的上游有的新的提交（commit），想要获取更新，有两种方法：

直接进入子模块所在目录 app/image 通过运行 git fetch 与 git merge，合并上游分支来更新本地代码。

直接运行 git submodule update --remote image 获取上游分支来更新本地代码。

## 9.7 本章小结

本文主要讨论了如何利用 Git 与 Github 创建属于自己的计算机视觉项目，同时也介绍了如何利用 git submodule 在大项目中管理小项目。虽然，本项目介绍的计算机视觉项目的创建，但是它同样适用于其他领域的项目的创建。

# 第 10 章 从零开始设计计算机视觉软件

本章导航：

- 说明如何使用 GitHub 创建一个项目。
- 使用 Python 如何创建一个 bbox 处理的工具。
- 介绍 SymPy 做符号运算。

## 10.1 创建一个项目

在介绍本章主题之前，需要了解本章需要的背景知识：

- 了解 Git 与 vscode。
- 熟悉 Python。
- Shell（命令行命令）基本命令，本书均以\$开头用于标识。
- 本章的代码内容均是在 Jupyter Notebook 环境下运行的。

本章尽可能地从零基础说明如何在 <https://github.com> 创建一个项目，并在本地进行项目开发。创建项目的步骤很简单：

(1) 进入自己的 GitHub 主页，比如：<https://github.com/xinetzone>，如图 10.1 所示，点击页面右上角的 +，选择 New repository。

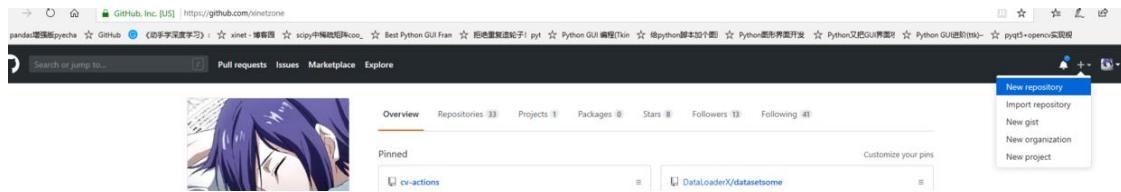


图 10.1 选择 New repository

填写必需项目信息，如图 10.2 所示。

Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository.](#)

Repository template

Start your repository with a template repository's contents.

No template ▾

Owner      Repository name \*

xinetzone /

Great repository names are short and memorable. Need inspiration? How about bookish-octo-funicular?

Description (optional)

Public      Anyone can see this repository. You choose who can commit.

Private      You choose who can see and commit to this repository.

Skip this step if you're importing an existing repository.

Initialize this repository with a README

This will let you immediately clone the repository to your computer.

Add .gitignore: None | Add a license: None | ⓘ

Grant your Marketplace apps access to this repository

You are subscribed to 2 Marketplace apps

GitHub Learning Lab

Your interactive guide to learning the skills you need without leaving GitHub

todo

Create new issues from actionable comments in your code

**Create repository**

图 10.2 填写必需项目信息

表单上的 Initialize this repository with a README 需要 ，.gitignore 选择在项目中需要使用的编程语言，比如，Python，license 选择一个需要的即可。

如果想要将此项目打造为一个社区，可以参考我的博客：构建属于自己的项目 <https://xinetzone.github.io/zh-CN/e6d6f9e7.html>。至此，完成了项目的创建工作。

## 10.2 项目的准备工作

在电脑磁盘上创建一个名为 projects 的文件夹，然后使用 vscode 打开该文件夹。接着，创建一个终端（快捷键 **Ctrl+shift+`**），并使用 git clone URL 克隆您的项目，如图 10.3 所示。

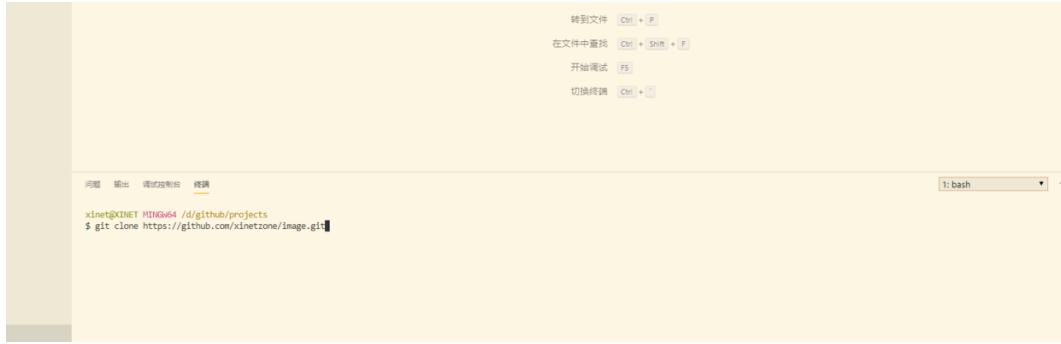


图 10.3 克隆项目到本地

转到项目目录，编辑 README.md 文件，通过网站 <https://img.shields.io> 为您的项目添加图标信息，比如：

```
数字图像处理
[![GitHub
issues](https://img.shields.io/github/issues/xinetzone/image)](https://github.com/xinetzone/image/issu
es)
[![GitHub
forks](https://img.shields.io/github/forks/xinetzone/image)](https://github.com/xinetzone/image/network)
[![GitHub
stars](https://img.shields.io/github/stars/xinetzone/image)](https://github.com/xinetzone/image/stargaz
ers)
[![GitHub
license](https://img.shields.io/github/license/xinetzone/image)](https://github.com/xinetzone/image/blob
/master/LICENSE) ![
GitHub repo size](https://img.shields.io/github/repo-size/xinetzone/image)
```

显示的效果如图 10.4 所示。

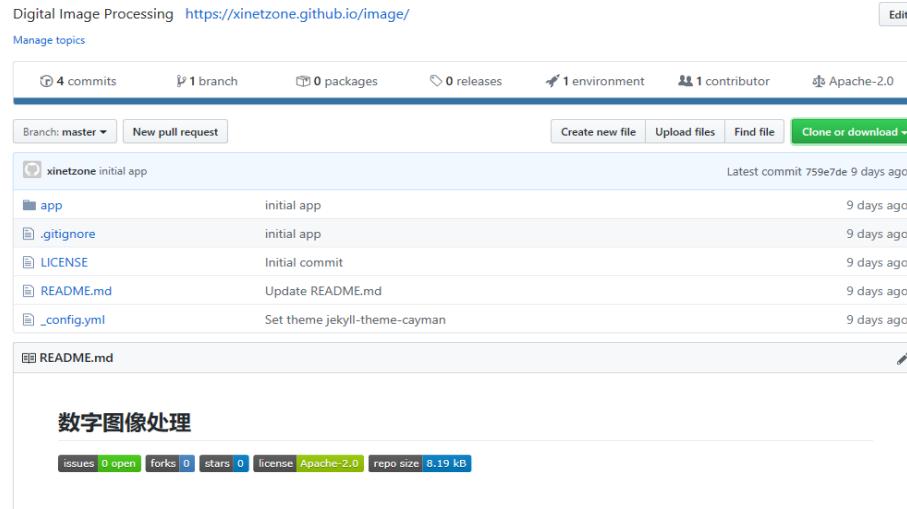


图 10.4 创建图标

下面就以我自己的项目 image（在 GitHub 中搜索 xinetzone/image 即可）为例展示如何开发项目。

## 10.3 开发一个小工具：bbox

对于目标检测任务，总是会涉及到对边界框（Bounding Box，简称 bbox）的处理，因而，开发一个专门处理边界框的 API 是十分有必要的。在创建小工具 bbox 之前，我们需要了解一些 Python 的基础知识并创建一个数学的向量实例。

### 10.3.1 创建数学中的“向量”

Python 中存在一个十分强大的标准库：dataclass。dataclass 的定义位于 PEP-557，一个 dataclass 是指“一个带有默认值的可变的 namedtuple”，广义的定义就是有一个类，它的属性均可公开访问，可以带有默认值并能被修改，而且类中含有与这些属性相关的类方法，那么这个类就可以称为 dataclass，再通俗点讲，dataclass 就是一个含有数据及操作数据方法的容器。

关于 dataclass 的更多精彩内容可参阅 Python 官方网站搜索 dataclasses 与 typing（详细的介绍可参考搜索我的博文：“xinetzone + Python 3 之类型注解”）。

为了更好的说明 dataclass 的魅力。先从“整点”开始的定义开始。即定义点 A 为  $A = x, x \in Z$ 。先看看 typing 的方法如何定义？

```
from typing import Any
class Point:
 def __init__(self, name: str, x: int) -> Any:
 self.x = x # 坐标值
 self.name = name # 名称
 def __repr__(self) -> Any:
 print("name={self.name}, x={self.x}")
```

再看看 dataclass 如何定义？

```
from dataclasses import dataclass
@dataclass
class Point:
 name: str
 x: int
```

在数学中一般使用向量来表示点。即设  $x_i \in \mathbb{R}^n, 0 \leq i \leq n$ ，则可以使用  $A = (x_1, \dots, x_n)$  代表点A。这里的 n 被称为点或者向量的纬度，我们使用 Python 实现向量的构建工作。

```
from typing import Sequence
from dataclasses import dataclass
import numpy as np
@dataclass
class Vector:
 """
 数学中的向量
 """
 name: str
 # 矢量的分量
 components: Sequence[float]
 @property
 def toArray(self):
 return np.asarray(self.components)
```

```

def __len__(self):
 """
 向量的维度
 """
 return len(self.components)
def __add__(self, other):
 """
 向量加法运算
 """
 assert len(self) == len(other), "向量的维度不相同"
 out = self.toArray + other.toArray
 return out
def __sub__(self, other):
 """
 向量减法运算
 """
 assert len(self) == len(other), "向量的维度不相同"
 out = self.toArray - other.toArray
 return out
def scale(self, scalar):
 """
 向量的数乘运算
 """
 return scalar * self.toArray
def __mul__(self, other):
 """
 向量的内积运算
 """
 assert len(self) == len(other), "向量的维度不相同"
 return np.dot(self.toArray, other.toArray.T)
@property
def norm(self):
 """
 计算模长
 """
 # S = sum(component**2 for component in self.components)
 mod = self * self
 return np.sqrt(mod)
def __getitem__(self, index):
 """
 向量的索引与切片
 """
 return self.components[index]

```

将该代码保存到 /app/vector.py 文件之中。下面看一个实例：

```

a = Vector('a', range(2,7))
a1 = Vector('a1', range(5,10))
a2 = Vector('a2', range(7,12))
a + a1, a - a1, a.scalar(4), a.norm

```

创建了 3 个向量  $a, a_1, a_2$ ，接着，分别计算其运算：向量加法，减法，数乘以及模。输出

结果：

```
(array([7, 9, 11, 13, 15]),
 array([-3, -3, -3, -3, -3]),
 array([8, 12, 16, 20, 24]),
 9.486832980505138)
```

---

### 10.3.2 编写 box.py 代码

Vector 定义了“点”，下面需要定义：有向线段  $[x_1, x_2] = x_2 - x_1$ 。使用 Python 定义很简单：

```
x1 = Vector('x1', [1, 2])
x2 = Vector('x2', [3, 4])
LS = Vector('x2 - x1', x2 - x1) # 线段 x2 - x1
```

则点  $x_2$  与  $x_1$  之间的线段，可以使用  $x_1 + t[x_1, x_2]$ ,  $0 \leq t \leq 1$  进行表示。这样一来，以  $x_2$  与  $x_1$  连接的线段为对角线的矩形框便由  $x_2$  与  $x_1$  唯一确定。

由于大多数情况，矩形框是二维平面图形，所以，接下来仅仅考虑二维向量生成的矩形框。使用 Python 可以这样定义：

```
from dataclasses import dataclass
from typing import Sequence
import numpy as np
from vector import Vector
@dataclass
class Rectangle:
 box: Sequence[Vector]
 def __post_init__(self):
 self.w, self.h = np.abs(self.box[1] - self.box[0])
 def __lt__(self, size):
 ""
 判断矩形的尺寸是否是小于 size
 ""
 w_cond = self.w < size # 宽是否小于 size
 h_cond = self.h < size # 高是否小于 size
 cond = w_cond or h_cond # 宽或者高是否均小于 size
 return cond
 def __ge__(self, size):
 min_size = self < size
 return not min_size
 @property
 def area(self):
 ""
 计算矩形面积
 ""
 return self.w * self.h
```

该代码保存在 /app/image/box.py 之中。Rectangle 完成了矩形边界框的基础构建工作。同时提供了宽、高、面积计算以及判断是否为最小边界框的实现。

我们依然看一个例子：

设存在一个矩形框  $a = (x_1, y_1, x_2, y_2)$ , 其中,  $(x_1, y_1)$ ,  $(x_2, y_2)$  分别表示矩形框的左上角、右下角坐标。这样, 利用 Rectangle 便可定义此矩形框:

```
a1 = Vector('a_1', [2, 3])
a2 = Vector('a_1', [7, 13])
bbox = Rectangle([a1, a2])
```

接着, 可以计算 bbox 的面积、高、以及宽:

```
bbox.area, bbox.h, bbox.w
```

输出结果是:

```
(66, 10, 5)
```

至此, 一个简单的边界框模型搭建完成了。

可以说数学是计算机视觉的核心, 我们需要使用数学理论分析大量的视觉数据。学习计算机视觉相关知识, 总是绕不开数学的。

## 10.4 Python 中的数学: 符号计算

Python 有许多优秀的处理数学的工具, 本章仅仅介绍符号计算库: SymPy。

### 10.4.1 使用 SymPy 表示数

Sympy 库完全由 Python 写成, 支持符号计算、高精度计算、模式匹配、绘图、解方程、微积分、组合数学、离散数学、几何学、概率与统计、物理学等方面的功能。

我们来简单回顾一下数学中的“数”:

$$\left\{ \begin{array}{l} \text{实数} \\ \text{虚数} \end{array} \right\} \left\{ \begin{array}{l} \text{有理数} \\ \text{无理数} \end{array} \right\} \left\{ \begin{array}{l} \text{整数} \\ \text{分数} \end{array} \right\}$$

虚数一般使用  $a + bi$  的形式进行表示。其中  $a, b \in \mathbb{R}$ , 而  $i$  是虚数的单位。在 SymPy 中针对这些特定的“数”, 有专门的符号表示。

使用 I 表示  $i$ , 使用 Rational 表示分数。

```
from sympy import I, Rational, sqrt, pi, E, oo
3 + 4I, Rational(1, 3), sqrt(8), sqrt(-1), pi, E**2, oo
```

输出的结果是:

$$3 + 4i, \frac{1}{3}, 2\sqrt{2}, i, \pi, e^2, \infty$$

可以看出, SymPy 可以完美的表示我们熟悉的各种数学中定义的“数”。

### 10.4.2 使用 SymPy 提升您的数学计算能力

为什么要使用 SymPy 呢? 直接使用 Python 的 math 库不就可以了吗? 您也许会有诸如此类的疑问。在解释原因之前, 我们来看一个例子:

```
import math
math.sqrt(8)
```

输出的结果是:

```
2.8284271247461903
```

2.8284271247461903? 您没有看出, 由于计算机是以二进制方式进行数学运算的, 故而计算虚数的值总是不精确的, 而 SymPy 的计算结果便是精确的。

```
import math, sympy
math.sqrt(8) ** 2, sympy.sqrt(8) ** 2
```

输出结果是:

```
(8.000000000000002, 8)
```

## 1. 对数学表达式进行简化与展开

在 SymPy 中使用 symbols 定义数学中一个名词: 变量。使用 symbols 创建多个变量名时, 请使用空格或者逗号进行隔开, 即 symbols('x y') 或者 symbols('x,y') 的形式。

```
from sympy import symbols
x, y = symbols('x y')
expr = x + 2*y
expr
```

输出结果:

```
x + 2y
```

这样, 我们便可以将 expr 当作数学中的表达式了:

```
expr + 1, expr - x
```

输出结果:

```
x+2y+1,2y
```

既然谈到数学表达式, 总会涉及的其的展开与化简。比如  $x(x + 2y) = x^2 + 2xy$ , 可以这样:

```
from sympy import expand, factor
expanded_expr = expand(x*expr)
expanded_expr
```

输出结果是:

```
x2 + 2xy
```

我们还可以将其展开式转换为因子的乘积的形式:

```
factor(expanded_expr)
```

输出的结果是:

```
x(x + 2y)
```

SymPy 不仅仅如此, 它还可以支持 LATEX 公式:

```
alpha, beta, nu = symbols('alpha beta nu')
alpha, beta, nu
```

输出结果是:

```
 α, β, ν
```

需要注意的是 symbols 定义的变量 x, y 等不再是字符串, 您将其看作是数学中的变量即可。

## 2. 求导, 微分, 定积分, 不定积分, 极限

先载入一些会用到的函数, 方法, 符号:

```
from sympy import symbols, sin, cos, exp, oo
from sympy import expand, factor, diff, integrate, limit
x, y, t = symbols('x y t')
```

在 SymPy 中使用 diff 对数学表达式求导，比如： $(\sin(x)e^x)' = e^x \sin(x) + e^x \cos(x)$ ，使用 SymPy，则有：

```
diff(sin(x)*exp(x), x)
```

输出结果是：

```
exsin(x) + excos(x)
```

在 SymPy 中使用 integrate 求解不定积分与定积分。比如， $\int e^x \sin(x) + e^x \cos(x) dx = \sin(x)e^x$ ，即：

```
integrate(exp(x)*sin(x) + exp(x)*cos(x), x)
```

输出结果：

```
sin(x)ex
```

定积分  $\int_{-\infty}^{\infty} \sin(x^2) dx = \frac{\sqrt{2\pi}}{2}$

```
integrate(sin(x**2), (x, -oo, oo))
```

输出结果：

```
sqrt(2)*sqrt(pi)

2
```

在 SymPy 中使用 limit 求极限，比如  $\lim_{x \rightarrow 0} \frac{\sin(x)}{x} = 1$ ，即：

```
limit(sin(x)/x, x, 0)
```

输出结果：

```
1
```

### 3.解方程

```
from sympy import solve, dsolve, Eq, Function
```

可以使用 SymPy 的 solve 求解  $f(x) = 0$  这种类型的方程。比如，求解  $x^2 - 2 = 0$ ：

```
solve(x**2 - 2, x)
```

输出结果是：

```
[-sqrt(2), sqrt(2)]
```

还可以使用 SymPy 的 dsolve 函数求解微分方程：

```
y = Function('y')
```

```
dsolve(Eq(y(t).diff(t, t) - y(t), exp(t)), y(t))
```

输出  $y'' - y = e^t$  的结果：

```
C_2e-t + (C_1 + t/2)et
```

小贴士：可以使用 sympy.latex 将您的运算结果使用 LATEX 的形式打印出来：

```
from sympy import latex
```

```
latex(Integral(cos(x)**2, (x, 0, pi)))
```

输出结果：

```
\intlimits_{0}^{\pi} \cos^2(x) dx
```

### 10.4.3 使用 SymPy 求值

在 10.4.2 中我们已经介绍了 SymPy 的大多数符号运算机制，接下来将讨论如何通过“赋值”（数学中的变量赋值）来获取表达式的值。如果您想要为表达式求值，可以使用 subs 函数（Substitution，即置换）：

```
x = symbols('x')
```

```
expr = x + 1
```

```
expr.subs(x, 2)
```

输出的结果为:

```
3
```

正是我们想要得到的预期结果。

在 10.4.2 中出现的符号 Eq 是用来表示两个数学表达式的相等关系。比如,  $x + 1 = 4$ , 可以这样:

```
Eq(x+1, 4)
```

输出结果:

```
x + 1 == 4
```

又比如,  $(x + 1)^2 = x^2 + 2x + 1$ , 可以这样:

```
Eq((x + 1)**2, x**2 + 2*x + 1)
```

输出结果:

```
(x + 1)**2 == x**2 + 2*x + 1
```

如果想要判断两个数学表达式是否相等, 有两种方式。下面我们来判断  $\cos^2(x) - \sin^2(x) = \cos(2x)$

方式一: 使用 equals。

```
a = cos(x)**2 - sin(x)**2
```

```
b = cos(2*x)
```

```
a.equals(b)
```

输出结果为 True 符合预期。

方式二: 使用 simplify 化简等式:

```
simplify(a - b)
```

输出结果为 0 也符合预期。

#### 10.4.4 使用 SymPy 做向量运算

在 SymPy 中分别使用 Point, Segment 来表示数学中的点与线段实体 (entity)。

```
from sympy import Segment, Point
from sympy.abc import x
a = Point(1, 2, 3)
b = Point([2, 3])
c = Point(0, x)
a, b, c
```

输出结果是:

```
(Point3D(1, 2, 3), Point2D(2, 3), Point2D(0, x))
```

即 Point 用来表示数学中的点 (向量) :  $x = (x_1, \dots, x_n) \in \mathbb{R}^n$ , 其中  $x_j \in \mathbb{R}$ ,  $1 \leq j \leq n$ 。对于 Segment(x1,x2)中的参数 x1, x2 可以是 Point 实例, 也可以是元组或者列表, array 等。但是 x1 与 x2 代表的点的维度必须一样。Segment(x1,x2)表示一个有向的线段, 即矩形的对角线方向为  $x_2 - x_1$ 。比如, 定义下图 10.5 的实体:

```
x1 = Point(1,1)
x2 = Point(2,2)
s = Segment(x1,x2)
```

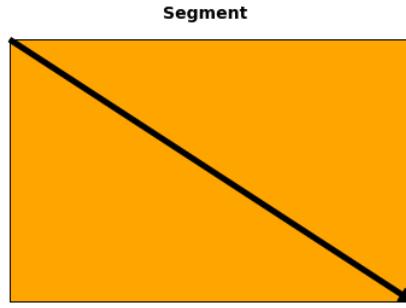


图 10.5 Segment 示意图

图中的有向对角线（向量）可以通过 `s.direction` 进行计算：

`s.direction`

输出结果为一个向量：

`Point2D(1,1)`

即向量  $x_2 - x_1$ 。在许多领域中，一般将水平向右作为 X 轴，水平向下作为 Y 轴。左上角作为原点 (0,0)。

有了对角线的方向，便可以计算矩形框的宽与高：

`w, h = s.direction`

由于 Point 指代数学中的向量，故而其存在加法、减法、数乘及内积运算，它们均被重载为 Python 的运算符。比如，我们定义向量 `a, b, c`，且  $a + b = c$ ，可有：

`a = Point(0,1)`

`b = Point(1,0)`

`c = Point(1,1)`

下面对它们进行加法运算：

`a + b == c`

输出结果为 True 符合预期。同样有：

`a - b, c * 2, a.dot(c)`

输出结果为：

`(Point2D(-1, 1), Point2D(2, 2), 1)`

所有结果均符合预期。下面再来看看 Segment：

`ab = Segment(a, b)`

`ac = Segment(a, c)`

`ab.direction, ac.direction`

输出的结果是：

`(Point2D(1, -1), Point2D(1, 0))`

我们可以计算 ab 与 ac 的方向向量的夹角余弦：

`ab.direction.dot(ac.direction) / (ab.length * ac.length)`

输出结果为  $\frac{\sqrt{2}}{2}$ ，即  $\cos(\theta) = \frac{a \cdot b}{|a||b|} = \frac{\sqrt{2}}{2}$ 。很容易求得  $\theta = \frac{\pi}{4}$ 。更方便的是，Segment 提供了函数

直接计算夹角：

`ab.angle_between(ac)`

输出结果也是  $\frac{\pi}{4}$ 。是不是很方便？如果您想要获取 Segment 实例的全部点可以调用 `ab.points`，分别调取各点则可以使用 `ab.p1` 与 `ab.p2`。

## 10.5 案例：实现图像局部 resize 保持高宽比不变

下面利用 9.4 节学习的东西构造一个实现图像局部 resize 保持高宽比不变的模型。

题设：假设有一张图片 I 的尺寸为  $(W_I, H_I)$ , I 存在一个目标物 O, O 的尺寸为  $(W, H)$ 。

目标：将 O resize 为尺寸是  $(w, h)$ , 而约束条件是保证 resize 之后的边界为  $w_m, h_m$  (注意：这里的  $h_m$  表示下边界长度)。需要尽可能保持 resize 之后的图片块宽高比不变，即满足公式 10.1：

$$\frac{W}{H} = \frac{w - 2w_m}{h - h_m - h_p} \quad (\text{公式 10.1})$$

其中  $h_p$  指的是上边界的长度。这里只有  $h_p$  的未知量，我们可以直接求出：

$$h_p = h - h_m - \frac{W - 2w_m}{W} H \quad (\text{公式 10.2})$$

为了保证 resize 的一致性，需要在原图补边  $H_m, W_m, H_p$  对应于  $h_m, w_m, h_p$ 。下面给出它们的计算公式：

$$\frac{w}{h} = \frac{W + 2W_m}{H + H_m + H_p} \quad (\text{公式 10.3})$$

同时还需要保证 resize 前后的宽高与边界的比例不变：

$$\frac{W}{W_m} = \frac{w}{w_m} \quad (\text{公式 10.4})$$

$$\frac{H}{H_m} = \frac{h}{h_m} \quad (\text{公式 10.5})$$

联合上述公式，便可以求出所有未知量。至此，模型建立完毕。下面考虑使用 Python 实现。

```
from sympy import Point, Segment, symbols, Eq, solve
from dataclasses import dataclass
@dataclass
class Resize:
 W: int
 H: int
 w: int
 h: int
 w_m: int
 h_m: int
 def model(self):
 H_m, W_m, H_p, h_p = symbols("H_m, W_m, H_p, h_p")
 eq1 = Eq(self.W/self.H, (self.w - 2 * self.w_m)/(self.h - self.h_m - h_p))
 eq2 = Eq(self.w/self.h, (self.W + 2 * W_m)/(self.H + H_m + H_p))
 eq3 = Eq(self.W/W_m, self.w/w_m)
```

```

eq4 = Eq(self.H/H_m, self.h/self.h_m)
R = solve([eq1, eq2, eq3, eq4], [H_m, W_m, H_p, h_p])
return R
传入已知量
W, H = 100, 100
w, h = 32, 32
w_m, h_m = 5, 5
模型建立
r = Resize(W, H, w, h, w_m, h_m)
模型求解
R = r.model()
查看求解结果
R

```

输出结果为：

```

{H_m: 15.62500000000000,
 W_m: 15.62500000000000,
 H_p: 15.62500000000000,
 h_p: 5.00000000000000}

```

这样我们完成了设定的目标。为了更直观，从网上找一张蛇豆的图片作为例子。为了可视化的方便，我们需要定义一个画边界框的函数：

```

def bbox_to_rect(bbox, color):
 # 将边界框(左上 x, 左上 y, 右下 x, 右下 y)格式转换成 matplotlib 格式：
 # ((左上 x, 左上 y), 宽, 高)
 return plt.Rectangle(
 xy=(bbox[0], bbox[1]), width=bbox[2]-bbox[0], height=bbox[3]-bbox[1],
 fill=False, edgecolor=color, linewidth=1)

```

该函数的参数 `bbox` 取值为  $(x_{\text{左上}}, y_{\text{左上}}, x_{\text{右下}}, y_{\text{右下}})$ ，`color` 指定框的颜色。

```

from matplotlib import pyplot as plt
%matplotlib inline
读取图片
img = plt.imread("蛇豆.jpg")
bbox = [75, 21, 160, 206] # 框的坐标
fig = plt.imshow(img)
fig.axes.add_patch(bbox_to_rect(bbox, 'red'))
plt.show()

```

输出结果，如图 10.6 所示。

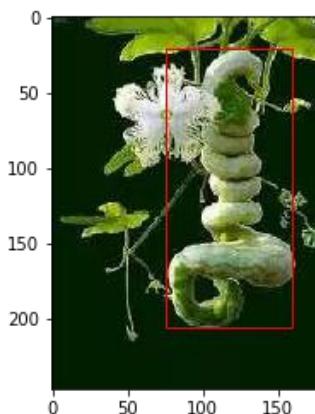


图 10.6 一张蛇豆的图片

数组 `img` 指代 `I`, 而图中的“蛇豆”(即 `O`)我们可以使用切片的方式获取, 即 `img[bbox[1]:bbox[3]+1,bbox[0]:bbox[2]+1]`(这里需要注意, 切片的第一维度为高), 即:

```
patch = img[bbox[1]:bbox[3]+1,bbox[0]:bbox[2]+1]
plt.imshow(patch)
plt.show()
```

输出结果, 如图 10.7 所示。

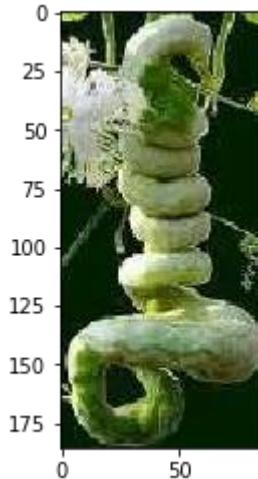


图 10.7 蛇豆的切片, 简称蛇豆片, 即图像块

可以直接求得蛇豆片的宽和高: `h, w = patch.shape[:2]`, 但是, `patch` 是由 `bbox` 获取的, 故而, 我们也可以直接求得:

```
def get_aspect(bbox):
 """
 计算 bbox 的宽和高
 """
 # 加 1 是因为像素点是的尺寸为 1x1
 w = bbox[2] - bbox[0] + 1
 h = bbox[3] - bbox[1] + 1
 return w, h
获取宽高
W, H = get_aspect(bbox)
```

我们想要将蛇豆片 `resize` 为 `(124, 270)`, 则可以设定:

```
w, h = 124, 270
w_m, h_m = 5, 5
```

接着, 计算边界:

```
r = Resize(W, H, w, h, w_m, h_m)
R = r.model()
H_m, W_m, H_p, h_p = R.values()
```

由于这里获取的值是浮点数, 我们需要将其转换为整数:

```
def float2int(*args):
 return [int(arg)+1 for arg in args]
获取整型数据
```

```
H_m, W_m, H_p, h_p = float2int(*R.values())
```

最后，原图与 resize 之后的图片展示如下：

```
import cv2
patch = img[bbox[1]-H_p:bbox[3]+1+H_m,bbox[0]-W_m:bbox[2]+1+W_m]
resize = cv2.resize(patch, (w, h))
plt.imshow(patch)
plt.title("origin")
plt.show()
plt.imshow(resize)
plt.title("resize")
plt.show()
```

输出结果见图 10.8 所示，我们可以看出 resize 的效果十分不错。

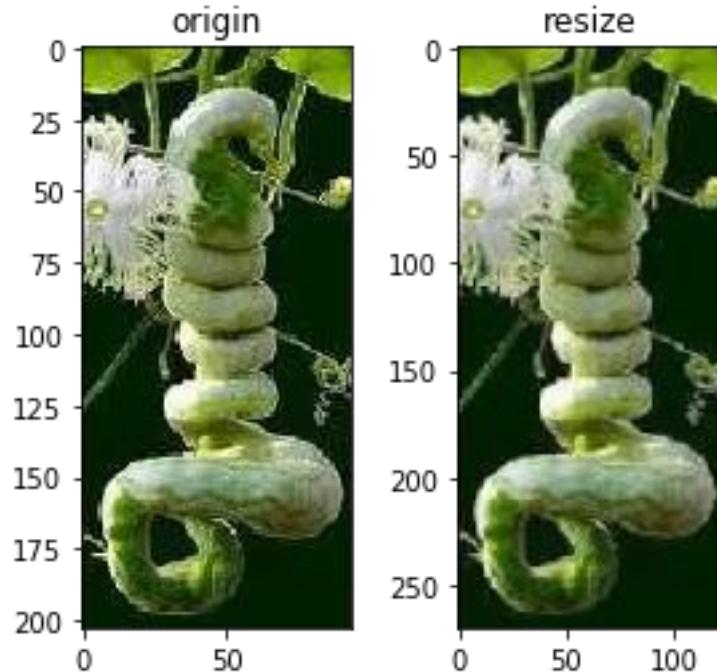


图 10.8 resize 前后对比图

本小结主要介绍如何利用 SymPy 实现一个 resize 保持宽高比不变的模型。在模型的建立过程中，我们发现 SymPy 为我们提供了十分便利的公式推理工具，因而，好好的利用 SymPy 您一定会创建一个十分强大的属于自己的工具包。

## 10.6 本章小结

本章主要介绍如何从零开始设计计算机视觉软件。首先介绍了创建项目的一般步骤，之后简述 SymPy 的基础知识点，最后，以 SymPy 为基础创建了一个模型，该模型可以实现将图片 resize 之后保持宽高比（aspect rate）不变的特性。

本章的代码逻辑并不是很严谨，主要是提供如何一个创建计算机视觉软件的思路，仔细研究本章的软件设计思路将会收获很多的。

# 第四篇 案例篇

## 第 11 章 Kaggle 实战：猫狗分类（Gluon 版）

本章利用 Kaggle 上的数据集：<https://www.kaggle.com/c/dogs-vs-cats/data> 来学习卷积神经网络。利用了第 7 章的模块：ImageZ 等。它们被封装进了一个名字叫 zipimage.py 的文件中。接下的几章也会利用该模块。

本章快报：

- 自定义数据处理 API。
- 利用 MXNet 来创建和训练模型。
- 利用迁移学习进一步提升模型的泛化性。

卷积神经网络是什么？它到底有什么神奇之处，使得其超越传统的机器学习方法？下面让我们一步一步地揭开它的神秘面纱！

### 11.1 数据处理

一般地，数据是一个模型的灵魂，为了让模型变得更好，下面需要先花点时间处理数据。先载入一些必备的包，代码如下所示：

```
import zipfile # 处理压缩文件
import os
import pandas as pd # 处理 csv 文件
import numpy as np
from matplotlib import pyplot as plt
----- 自定义模块
from utils.zipimage import ImageZ
%matplotlib inline # 使得 Notebook 可以显示图片
```

查看 <https://www.kaggle.com/c/dogs-vs-cats/data> 来了解数据的基本信息。从该网页可以知道：训练数据 train.zip 包括 50 000 个样本，其中猫和狗各半。而其任务是预测 test1.zip 的标签（1 = dog, 0 = cat）。为了方便，将数据下载到本地，然后做如下操作。代码如下所示：

```
def unzip(root, NAME): # 函数 unzip 被封装到了 kaggle/helper.py
 source_path = os.path.join(root, 'all.zip')
 dataDir = os.path.join(root, NAME)
 with zipfile.ZipFile(source_path) as fp:
 fp.extractall(dataDir) # 解压 all.zip 数据集到 dataDir
 os.remove(source_path) # 删除 all.zip
 return dataDir # 返回数据所在目录
具体设置
root = 'data/'
```

```
NAME = 'dog_cat'
dataDir = unzip(root, dataDir)
```

为了以后处理其他的 Kaggle 提供的数据，将 `unzip` 函数封装到了 `kaggle` 包下的 `helper` 模块中。`unzip` 实现的功能是将 `root` 下的 '`all.zip`' 文件解压并返回解压后数据所在的目录，同时也将 '`all.zip`' 删除（之后用不到了）。先看看 `dataDir` 目录下面都是什么文件？

代码如下所示：

```
dataDir = 'data/dog_cat'
os.listdir(dataDir)
```

代码如下所示：

```
['sampleSubmission.csv', 'test1.zip', 'train.zip']
```

文件 '`sampleSubmission.csv`' 是 `kaggle` 比赛提交预测结果的模板样式。代码如下所示：

```
submit = pd.read_csv(os.path.join(dataDir, 'sampleSubmission.csv'))
submit.head()
```

图片如下图 11.1 所示。

	<b>id</b>	<b>label</b>
0	1	0
1	2	0
2	3	0
3	4	0
4	5	0

图 11.1 Kaggle 提供的标签样例

其中 `id` 表示 `test.zip` 中图片的文件名称，比如 `id = 1` 代表图片文件名为 `1.jpg`。`label` 表示图片的标签（`1=dog,0=cat`）。训练集和测试集都是 `.zip` 压缩文件，下面直接利用类 `ImageZ` 来读取数据。代码如下所示：

```
testset = ImageZ(dataDir, 'test1') # 测试数据
trainZ = ImageZ(dataDir, 'train') # 训练数据
```

你也许会疑惑，训练数据的标签呢？其实，在 <https://www.kaggle.com/c/dogs-vs-cats/data> 中你查看 `train.zip` 便可以发现其类别信息隐藏在文件名中，为此，可以直接查看 `trainZ` 的 `names` 属性。代码如下所示：

```
trainZ.names[:5] # 查看其中的 5 个文件名
```

代码如下所示：

```
['train/cat.0.jpg',
'train/cat.1.jpg',
'train/cat.10.jpg',
'train/cat.100.jpg',
'train/cat.1000.jpg']
```

因而，对于训练数据可以通过文件名来获知其所属于的类别。为了与 `1=dog,0=cat` 对应，下面定义，代码如下所示：

```
class_names = ('cat', 'dog')
class_names[1], class_names[0]
```

代码如下所示：

```
('dog', 'cat')
```

为了后期处理方便，定义 DataSet 类。代码如下所示：

```
class DataSet(ImageZ):
 def __init__(self, dataDir, dataType):
 super().__init__(dataDir, dataType)
 self.class_names = ('cat', 'dog') # 数据集的类名称
 self._get_name_class_dict()
 def _get_name_class_dict(self):
 self.name_class_dict = {} # 通过文件名获取图片的类别
 class_dict = {
 class_name: i
 for i, class_name in enumerate(self.class_names)
 }
 for name in self.names:
 class_name = name.split('.')[0].split('/')[-1]
 self.name_class_dict[name] = class_dict[class_name]
 def __iter__(self):
 for name in self.names:
 # 返回 (data, label) 数据形式
 yield self.buffer2array(name), self.name_class_dict[name]
```

下面看看如何使用 DataSet 类。代码如下所示：

```
dataset = DataSet(dataDir, 'train')
for img, label in dataset:
 print("大小: ", img.shape, '标签: ', label) # 查看一张图片
 plt.imshow(img)
 plt.show()
break
```

此时有图片输出，如图 11.2 所示。

```
大小: (374, 500, 3) 标签: 0
```



图 11.2 训练集中的一张图片

为了查看模型的泛化性，需要将 dataset 划分为 trainset 与 valset，为此需要将 dataset 的 names 属性进行划分。由于 dataset 的特殊性，下面对 trainZ 进行处理比较好。代码如下所示：

```
cat_rec = []
dog_rec = []
for name in trainZ.names:
 if name.startswith('train/cat'):
 cat_rec.append((name, 0))
 elif name.startswith('train/dog'):
```

```
 dog_rec.append((name, 1))
```

为了避免模型依赖于数据集的顺序，下面将会打乱原数据集的顺序。代码如下所示：

```
import random
random.shuffle(cat_rec) # 打乱 cat_names
random.shuffle(dog_rec) # 打乱 dog_names
train_rec = cat_rec[:10000] + dog_rec[:10000] # 各取其中的 10000 个样本作为训练
val_rec = cat_rec[10000:] + dog_rec[10000:] # 剩余的作为测试
random.shuffle(train_rec) # 打乱类别的分布，提高模型的泛化能力
random.shuffle(val_rec)
len(train_rec), len(val_rec)
```

代码如下所示：

```
(20000, 5000)
```

从上面的代码我们可以知道，训练数据和验证数据的样本个数分别为：20000 和 5000。下面为了可以让模型能够使用该数据集，需要定义一个生成器，代码如下所示：

```
class Loader:
 def __init__(self, imgZ, rec, shuffle=False, target_size=None):
 if shuffle:
 random.shuffle(rec) # 训练集需要打乱
 self.shuffle = shuffle
 self.imgZ = imgZ
 self.rec = rec
 self.__target_size = target_size
 def name2array(self, name):
 # 将 name 转换为 array
 import cv2
 img = self.imgZ.buffer2array(name)
 if self.__target_size: # 将图片 resize 为 self.target_size
 return cv2.resize(img, self.__target_size)
 else:
 return img
 def __getitem__(self, item):
 rec = self.rec[item]
 if isinstance(item, slice):
 return [(self.name2array(name), label) for (name, label) in rec]
 else:
 return self.name2array(rec[0]), rec[1]
 def __iter__(self):
 for name, label in self.rec:
 yield self.name2array(name), label # 返回 (data, label)
 def __len__(self):
 return len(self.rec) # 返回数据样本数
```

这样，便可以利用 Loader 得到随机划分后的 trainset 和 valset。

```
trainset = Loader(trainZ, train_rec, True) # 训练集
valset = Loader(trainZ, val_rec) # 验证集
for img, label in trainset:
 plt.imshow(img) # 显示出图片
 plt.title(str(class_names[label])) # 将类别作为标题
 plt.show()
```

```
break
```

图片如图 11.3 所示。

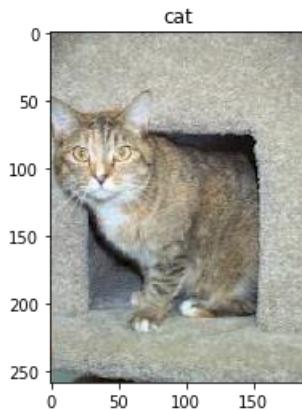


图 11.3 trainset 中的一张图片

数据处理好之后，看看图片的大小，代码如下所示：

```
name2size = {} # 获得图片的 size
for name in trainZ.names:
 name2size[name] = trainZ.buffer2array(name).shape[:-1]
min({w for h, w in set(name2size.values())}), min({h for h, w in set(name2size.values())})
```

代码如下所示：

```
(42, 32)
```

从上述代码可以看出：图片的最小高和宽分别为 32 和 42，基于此，不妨将所有的图片均 resize 为 (150, 150)。

## 11.2 Gluon 实现模型的训练和预测

Gluon 的学习可以参考我的学习笔记：<https://github.com/xinetzone/XinetStudio>。为了提高模型的泛化能力，数据增强技术是十分有必要的，Gluon 提供了一个十分方便的模块 transforms，下面我们来看看它的具体使用。代码如下所示：

```
from mxnet.gluon.data.vision import transforms as gtf
transform_train = gtf.Compose([
 # 随机对图像裁剪出面积为原图像面积 0.08~1 倍、且高和宽之比在 3/4~4/3 的图像，再放缩为高和
 # 宽都是为 150 的新图
 gtf.RandomResizedCrop(
 150, scale=(0.08, 1.0), ratio=(3.0 / 4.0, 4.0 / 3.0)),
 gtf.RandomFlipLeftRight(),
 # 随机变化亮度、对比度和饱和度
 gtf.RandomColorJitter(brightness=0.4, contrast=0.4, saturation=0.4),
 # 随机加噪声
 gtf.RandomLighting(0.1),
 gtf.ToTensor(),
 # 对图像的每个通道做标准化
 gtf.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])]
```

```
])
transform_test = gtf.Compose([
 gtf.Resize(256),
 # 将图像中央的高和宽均为 150 的正方形区域裁剪出来
 gtf.CenterCrop(150),
 gtf.ToTensor(),
 gtf.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
])
```

由于 Gluon 的数据输入是 `mxnet.ndarray.ndarray.NDArray` 类型的数据，而 `Loader` 类的输出是 `numpy.ndarray` 类型的数据，因而，需要改写 `Loader` 类。

```
from mxnet import nd, gluon
from mxnet.gluon import data as gdata
class GluonLoader(Loader, gdata.Dataset): # gdata.Dataset 是 gluon 处理数据的基类之一
 def __init__(self, imgZ, rec):
 super().__init__(imgZ, rec)

 def name2array(self, name):
 return nd.array(self.imgZ.buffer2array(name)) # 将 name 转换为 array
```

下面可以看看 `GluonLoader` 类实现了哪些有趣的功能？

```
train_ds = GluonLoader(trainZ, train_rec)
valid_ds = GluonLoader(trainZ, val_rec)
for img, label in train_ds:
 print(type(img), img.shape, label)
 break
```

输出：

```
<class 'mxnet.ndarray.ndarray.NDArray'> (391, 500, 3) 0
```

此时，`img` 已经满足 Gluon 所创建的模型的输入数据要求。对于图像分类任务，卷积神经网络无疑是一个比较好的模型。对于卷积神经网络来说，一般会采用小批量随机梯度下降优化策略来训练模型。我们需要将数据转换为批量形式，而 `gluon.data.DataLoader` 为我们实现该功能提供了便利，代码如下所示：

```
from mxnet.gluon import data as gdata
batch_size = 16 # 批量大小
train_iter = gdata.DataLoader(# 每次读取一个样本数为 batch_size 的小批量数据
 train_ds.transform_first(transform_train), # 变换应用在每个数据样本（图像和标签）的第一个元素，即图像之上
 batch_size,
 shuffle=True, # 训练数据需要打乱
 last_batch='keep') # 保留最后一个批量
valid_iter = gdata.DataLoader(
 valid_ds.transform_first(transform_test),
 batch_size,
 shuffle=True, # 验证数据需要打乱
 last_batch='keep')
```

`train_ds.transform_first(transform_train)` 和 `valid_ds.transform_first(transform_test)` 均将原数据集应用了数据增强技术，并将其转换为了适合 Gluon 所创建卷积神经网络的输入形式。

### 11.2.1 建立并训练模型

在建立本章所需要的模型之前，先了解一些卷积神经网络的必备知识。卷积神经网络简单的说就是含有卷积层的神经网络。神经网络就是具有层级结构的神经元组成的具有提取数据的多级特征的机器学习模型。

一个神经元由仿射变换（矩阵运算）+非线性变换（被称为激活函数）组成。在 Gluon 中卷积运算由 `nn.Conv2D` 来实现。

二维卷积层输出的数组可以看作是数据在宽和高维度上某一级的特征表示（也叫特征图（feature map）或表征）或是对输入的响应（称作响应图（response map）），代码如下所示：

```
载入一些必备包
from mxnet import autograd, init, gluon
from mxnet.gluon import model_zoo
import d2lzh as d2l
from mxnet.gluon import loss as gloss, nn
from gluoncv.utils import TrainingHistory # 可视化
from mxnet import metric # 计算度量的模块
```

`nn.Conv2D` 常用参数：

- `channels`: 下一层特征图的通道数，即卷积核的个数。
- `kernel_size`: 卷积核的尺寸。
- `activation`: 激活函数。

还有一点需要注意：卷积层的输出可由下面的公式来计算：

$$\begin{cases} h_{out} = \lfloor \frac{h_{in} - f_h + 2p}{s} \rfloor + 1 \\ w_{out} = \lfloor \frac{w_{in} - f_w + 2p}{s} \rfloor + 1 \end{cases}$$

- $h_{out}, w_{out}$  表示卷积层的输出的特征图尺寸。
- $h_{in}, w_{in}$  表示卷积层的输入的特征图尺寸。
- $f_h, f_w$  表示卷积核的尺寸，即 `nn.Conv2D` 的参数 `kernel_size`。
- $s$  表示卷积核移动的步长，即 `nn.Conv2D` 的参数 `strides`。
- $p$  表示卷积层的输入的填充大小，即 `nn.Conv2D` 的参数 `padding`，减小边界效应带来的影响。

虽然，使用步进卷积（`strides` 大于 1）可以对图像进行下采样，但是，一般很少使用它。一般地，可以使用池化操作来实现下采样。在 Gluon 中使用 `nn.MaxPool2D` 和 `nn.AvgPool2D` 实现。

下面创建模型并训练：

```
创建模型
model = nn.HybridSequential()
model.add(
 nn.BatchNorm(), nn.Activation('relu'), # 批量归一化
 nn.MaxPool2D((2, 2)), # 最大池化
 nn.Conv2D(32, (3, 3)), # 32 个卷积核尺寸为 (3, 3)
 nn.BatchNorm(), nn.Activation('relu'),
 nn.MaxPool2D((2, 2)), # 最大池化
 nn.Conv2D(64, (3, 3)), # 64 个卷积核尺寸为 (3, 3)
```

```

 nn.BatchNorm(), nn.Activation('relu'),
 nn.MaxPool2D((2, 2)), # 最大池化
 nn.Conv2D(128, (3, 3)), # 128 个卷积核尺寸为 (3, 3)
 nn.BatchNorm(), nn.Activation('relu'),
 nn.MaxPool2D((2, 2)), # 最大池化
 nn.Conv2D(512, kernel_size=1), # 1 x 1 卷积 进行降维
 nn.BatchNorm(), nn.Activation('relu'),
 nn.GlobalAvgPool2D(), # 全局平均池化
 nn.Dense(2) # 输出层
)
 return model
def evaluate_loss(data_iter, net, ctx):
 l_sum, n = 0.0, 0
 for X, y in data_iter:
 y = y.as_in_context(ctx).astype('float32') # 模型的输出是 float32 类型数据
 outputs = net(X.as_in_context(ctx)) # 模型的输出
 l_sum += loss(outputs, y).sum().asscalar() # 计算总损失
 n += y.size # 计算样本数
 return l_sum / n # 计算平均损失
def test(valid_iter, net, ctx):
 val_metric = metric.Accuracy()
 for X, y in valid_iter:
 X = X.as_in_context(ctx)
 y = y.as_in_context(ctx).astype('float32') # 模型的输出是 float32 类型数据
 outputs = net(X)
 val_metric.update(y, outputs)
 return val_metric.get()
def train(net, train_iter, valid_iter, num_epochs, lr, wd, ctx, model_name):
 import time
 trainer = gluon.Trainer(net.collect_params(), 'rmsprop', {
 'learning_rate': lr, 'wd': wd}) # 优化策略
 train_metric = metric.Accuracy()
 train_history = TrainingHistory(['training-error', 'validation-error'])
 best_val_score = 0
 for epoch in range(num_epochs):
 train_l_sum, n, start = 0.0, 0, time.time() # 计时开始
 train_acc_sum = 0
 train_metric.reset()
 for X, y in train_iter:
 X = X.as_in_context(ctx)
 y = y.as_in_context(ctx).astype('float32') # 模型的输出是 float32 类型数据
 with autograd.record(): # 记录梯度信息
 outputs = net(X) # 模型输出
 l = loss(outputs, y).sum() # 计算总损失
 l.backward() # 反向传播
 trainer.step(batch_size)
 train_l_sum += l.asscalar() # 计算该批量的总损失
 train_metric.update(y, outputs) # 计算训练精度
 n += y.size
 train_history.append('training-error', train_l_sum / n)
 train_l_sum, n, start = 0.0, 0, time.time()
 val_score = test(valid_iter, net, ctx)
 if val_score > best_val_score:
 best_val_score = val_score
 if model_name:
 save_parameters(net, model_name)
 train_history.append('validation-error', val_score)

```

```

 _, train_acc = train_metric.get()
 time_s = "time {:.2f} sec".format(time.time() - start) # 计时结束
 valid_loss = evaluate_loss(valid_iter, net, ctx) # 计算验证集的平均损失
 _, val_acc = test(valid_iter, net, ctx) # 计算验证集的精度
 epoch_s = ("epoch {:d}, train loss {:.5f}, valid loss {:.5f}, train acc {:.5f}, valid acc {:.5f}, ".format(
 epoch, train_l_sum / n, valid_loss, train_acc, val_acc))
 print(epoch_s + time_s)
 train_history.update([1-train_acc, 1-val_acc]) # 更新图像的纵轴
 train_history.plot(
 save_path='{}_{}/{}_history.png'.format('images', model_name)) # 实时更新图像
 if val_acc > best_val_score: # 保存比较好的模型
 best_val_score = val_acc
 net.save_parameters(
 '{}_{:.4f}-{:d}-best.params'.format('models', best_val_score, model_name, epoch))
 loss = gloss.SoftmaxCrossEntropyLoss() # 交叉熵损失函数
 ctx, num_epochs, lr, wd = d2l.try_gpu(), 30, 1e-3, 1e-5
 net = get_net()
 # 在 ctx 上训练 (如果有 gpu, 那么在 gpu 上训练)
 net.initialize(init.Xavier(), ctx=ctx) # 模型初始化
 net.hybridize() # 转换为符号式编程
 train(net, train_iter, valid_iter, num_epochs, lr, wd, ctx, 'dog_cat')

```

其输出结果, 如图 11.4 所示, Out [7.20]:

```

def get_net(ctx):
 epoch 0, train loss 0.69837, valid loss 0.65844, train acc 0.55850, valid acc 0.60940, time 185.46 sec
 epoch 1, train loss 0.66408, valid loss 0.60767, train acc 0.60845, valid acc 0.67960, time 182.20 sec
 epoch 2, train loss 0.64790, valid loss 0.61149, train acc 0.62825, valid acc 0.67880, time 169.60 sec


```

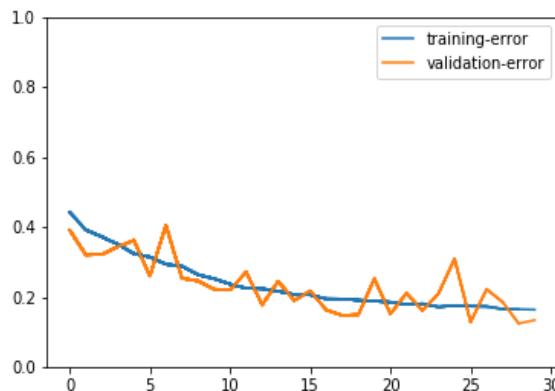


图 11.4 epochs=30 训练结果

上面创建的模型是依据 VGG 结构进行改写的, 同时为了加快模型训练加入了 nn.BatchNorm, 即批量归一化。训练次数有点少, 加大训练次数到 100, 得到如图 11.5 所示。

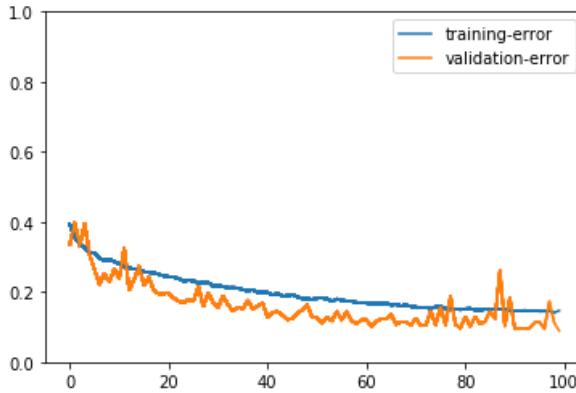


图 11.5 epochs=100 训练结果

可以看出已经有了十分不错的效果，在验证集的准确度在 0.9 附近。

### 11.2.2 模型测试

在此之前，本章是在 `train.zip`（训练数据集）上进行训练和验证，而最终的目的是在 `test1.zip`（测试集）上预测其分类结果。下面重新载入之前已经训练好的模型（第 100 个 epoch 验证集的准确度为 0.91112）。

```
net = get_net() # 定义网络结构
ctx = d2l.try_gpu() # 如果有 gpu, 那么在 gpu 上训练
net.load_parameters(filename='models/0.9112-dog_cat-99-best.params', ctx=ctx) # 在 ctx 上测试
net.hybridize() # 转换为符号式编程
class Testset(ImageZ):
 def __init__(self, root, dataType, ctx):
 super().__init__(root, dataType)
 self.ctx = ctx
 def name2label(self, name):
 img = self.buffer2array(name)
 X = transform_test(nd.array(img)).expand_dims(
 axis=0).as_in_context(self.ctx) # 将 NumPy 转换为 Gluon 定义的网络需要的格式
 return int(net(X).argmax(axis=1).asscalar()) # 返回预测结果
 def __getitem__(self, item):
 # 数据集切片
 names = self.names[item]
 if isinstance(item, slice):
 return [(name, self.name2label(name)) for name in names]
 else:
 return names, self.name2label(names)
 def __iter__(self):
 for name in self.names: # 迭代
 yield name, self.name2label(name) # 返回 (name, label)
```

```
载入 pandas 处理标签
import pandas as pd
_testset = Testset(dataDir, 'test1', ctx) # 实例化
df = pd.DataFrame.from_records(
 (name.split('/')[-1].split('.')[0], label) for name, label in _testset)
df.columns = ['id', 'label'] # 定义数据的表头
df.to_csv('data/dog_cat/results.csv', index=False) # 预测结果保存到本地
```

至此，完成测试集的预测，由于该比赛已经结束，无法提交结果，来查看其测试集的准确度，所以，我们仅仅将其保存到本地，而没有提交最终结果到 Kaggle 上。

### 11.3 可视化中间层的输出

下面直接借助前面训练好的模型，看看卷积神经网络到底训练出什么东西？

```
testset = ImageZ(dataDir, 'test1') # 测试数据
class_names = ('cat', 'dog') # 类别名称
ctx = d2l.try_gpu() # 训练的设备
net = get_net(ctx) # 在 ctx 上训练
加载模型参数
net.load_parameters('models/0.9112-dog_cat-99-best.params')
net.collect_params().reset_ctx(ctx) # 加载网络到 ctx
net.hybridize() # 符号化
```

先看看其中一张图片，如图 11.6 所示。

```
for img in testset: # 从测试集取出一张图片
 # 转换格式以适用模型
 X = loader.transform_test(nd.array(img)).expand_dims(axis=0).as_in_context(ctx)
 y = net(X) # 获取网络的输出
 label = class_names[y.argmax(1).astype('int').asscalar()]
 # 可视化
 plt.imshow(img)
 plt.title(label)
 plt.show()
 break
```

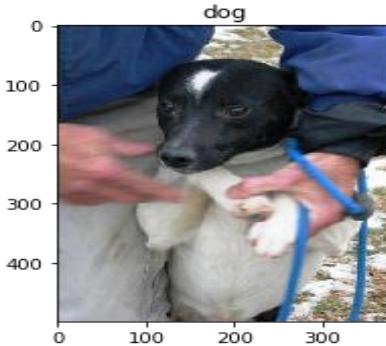


图 11.6 一张测试集的图片

在此，标注了图片预测的类别信息，可以看出模型没有预测错误。下面再来看看中间层的输出情况：

```
取出测试集的第 10 张图片，并将其转换为模型的输出格式
X = loader.transform_test(nd.array(testset[10])).expand_dims(axis=0).as_in_context(ctx)
for layer in net: # 遍历中间层
 X = layer(X) # 获取中间层的输出
 # 筛选出 pooling 层与卷积层
 if layer.name.startswith('pool') or layer.name.startswith('conv'):
 K = X.asnumpy()[0] # 转换为 plt 的输入形式
 size = K.shape[0] # 获取通道大小
 if size == 3: # 此时是原图最初的运算
 imgs = np.concatenate(K, axis=-1)
 else:
 i = 0
 while i <= size // 8: # 以 8 为单位平铺该层的所有输出
 imgs = np.concatenate(K[i:i+8], axis=-1)
 i += 8
 plt.matshow(imgs, cmap='viridis') # 画出特征图
 plt.title(layer.name) # 标注该层的名称
 plt.show()
```

从图 11.7 可以看出，随着层数的加深，该层所表达的信息逐渐丰富起来：刚开始表达的是图像的边缘与轮廓，最后可以表达图像的内容或者语义信息（“猫”这一概念）。

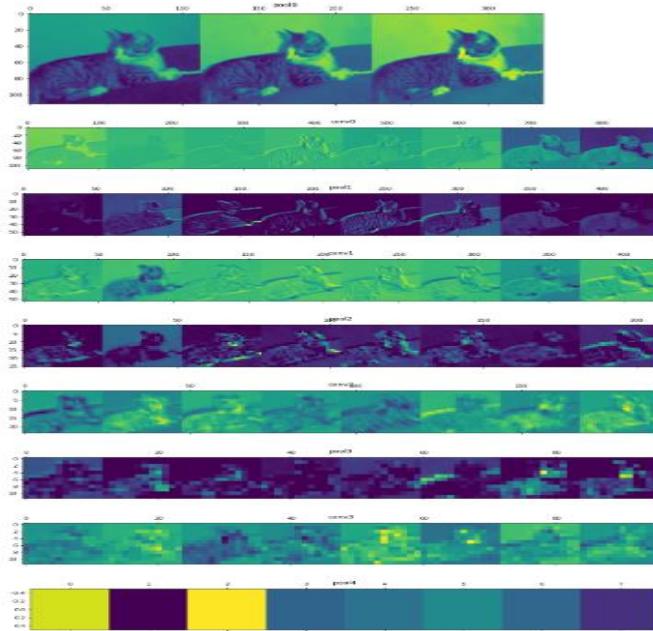


图 11.7 中间层的可视化

## 11.4 本章小结

本章利用 Kaggle 上的一个数据集设计了一个简单的分类模型。在设计该模型的过程中借助于 ImageZ 载入数据，之后利用 MXNet 设计一个类似于 VGG 的网络结构用来训练数据，最后对测试数据进行预测，并通过一张测试图片来查看本章设计的网络的中间层的输出所表达的语义信息。

# 第 12 章 利用 GluonCV 学习 Faster RCNN

本章的目标是理解和学习 Faster RCNN，所以不会介绍如何跑模型。Faster RCNN 主要分为两个部分：

- RPN (Region Proposal Network) 生成高质量的 region proposal。
- Fast R-CNN 利用 region proposal 做出检测与识别。

为了更好的理解 Faster RCNN，下面简要的叙述论文的关键内容。

## 12.1 初识 RPN

Faster RCNN 的作者将 PN 比作神经网络的注意力机制（“attention” mechanisms），告诉

了网络看向哪里。RPN 并不复杂，它的输入与输出：

- Input: 任意尺寸的图像。
- Output: 一组带有目标得分的目标矩形 proposals。

RPN 为了生成 region proposals，在基网络的最后一个卷积层上滑动一个小网络。该小网络由一个 $3 \times 3$  卷积 conv1 和一对兄弟卷积（并行的） $1 \times 1$  卷积 loc 和 score 组成。其中，conv1 的参数 padding=1, stride=1 以保证其不会改变输出的特征图的尺寸。

loc 作为 box-regression 用来编码 box 的坐标，score 作为 box-classification 用来编码每个 proposal 是目标的概率。详细内容见我的博客：我的目标检测笔记（<https://www.cnblogs.com/q735613050/p/10573794.html>）。一般地，不同 scale（尺度）和 aspect ratio（长宽比）组合的 k 个 reference boxes（参数化的 proposal）称作 anchors（锚点），同时锚点也是滑块（Window）的中心。

为了更好的理解 anchors，下面以 Python 来展示其内涵。

### 12.1.1 锚点

首先利用第 7 章的 COCO 中介绍的 API 来获取一张 COCO 数据集的图片及其标注。先载入一些必备的包：

```
import cv2
from matplotlib import pyplot as plt
import numpy as np
载入 coco 相关 api
import sys
sys.path.append('D:/API/cocoapi/PythonAPI') # cocoapi 所在路径
from pycocotools.dataset import Loader # 载入 Loader
%matplotlib inline
```

利用 Loader 载入 val2017 数据集，并选择包含‘cat’，‘dog’，‘person’的图片：

```
dataType = 'val2017' # 选取 2017 的验证集
root = 'E:/Data/coco' # coco 的根目录
catNms = ['cat', 'dog', 'person'] # 类别名称
annType = 'annotations_trainval2017' # 标注文件的名称
loader = Loader(dataType, catNms, root, annType) # 载入数据集的图片及其标签
```

输出结果：

```
Loading json in memory ...
used time: 0.762376 s
Loading json in memory ...
creating index...
index created!
used time: 0.401951 s
```

可以看出，Loader 载入数据的速度很快。为了更加详细的查看 loader，下面打印出一些相关信息：

```
print(f'总共包含 {len(loader)} 张')
for i, ann in enumerate(loader.images):
 w, h = ann['height'], ann['width']
 print(f'第 {i+1} 张图片的高和宽分别为: {w, h}')
```

显示：

```
总 共 包 含 图 片 片
第 1 张 图 片 的 高 和 宽 分 别 为 : 2 张
第 2 张图片的高和宽分别为: (500, 333)
```

下面以第 1 张图片为例来探讨 anchors。先可视化：

```
img, labels = loader[0] # 选择数据集的第一张图片及其标签
plt.imshow(img); # 可视化
```

输出如图 12.1 所示。

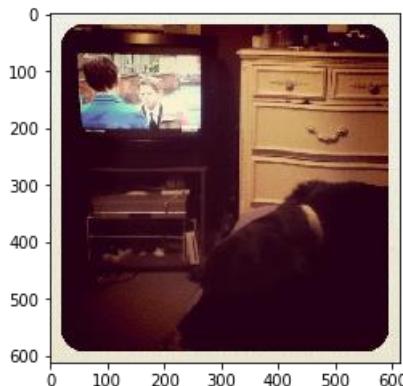


图 12.1 COCO 中的一张图片

为了特征图的尺寸大一点，可以将其 resize 改为(800, 800, 3)：

```
img = cv2.resize(img, (800, 800))
print(img.shape)
```

输出：

```
(800, 800, 3)
```

下面借助 MXNet 来完成接下来的代码编程，为了适配 MXNet，需要将图片由(h, w, 3)转换为(3, w, h)形式：

```
img = img.transpose(2, 1, 0)
print(img.shape)
```

输出：

```
(3, 800, 800)
```

由于卷积神经网络的输入是四维数据，所以还需要：

```
img = np.expand_dims(img, 0)
print(img.shape)
```

输出：

```
(1, 3, 800, 800)
```

为了和论文一致，我们也采用 VGG16 网络（载入 gluoncv 中的权重）：

```
from gluoncv.model_zoo import vgg16
net = vgg16(pretrained=True) # 载入权重
```

仅仅考虑直至最后一层卷积层(去除池化层)的网络，下面查看网络的各个卷积层的输出情况：

```
from mxnet import nd
imgs = nd.array(img) # 转换为 mxnet 的数据类型
x = imgs
for layer in net.features[:29]:
 x = layer(x)
```

```

if "conv" in layer.name:
 print(layer.name, x.shape) # 输出该卷积层的 shape

```

结果为：

				layer.name:
vgg0_conv0	(1,	64,	800,	800)
vgg0_conv1	(1,	64,	800,	800)
vgg0_conv2	(1,	128,	400,	400)
vgg0_conv3	(1,	128,	400,	400)
vgg0_conv4	(1,	256,	200,	200)
vgg0_conv5	(1,	256,	200,	200)
vgg0_conv6	(1,	256,	200,	200)
vgg0_conv7	(1,	512,	100,	100)
vgg0_conv8	(1,	512,	100,	100)
vgg0_conv9	(1,	512,	100,	100)
vgg0_conv10	(1,	512,	50,	50)
vgg0_conv11	(1,	512,	50,	50)
vgg0_conv12	(1, 512, 50, 50)			

由此，可以看出尺寸为(800, 800)的原图变为了(50, 50)的特征图（比原来缩小了 16 倍）。

### 12.1.2 感受野

上面的 16 不仅仅是针对尺寸为(800, 800)，它适用于任意尺寸的图片，因为 16 是特征图的一个像素点的感受野（receptive field）。

感受野的大小是如何计算的？我们回忆卷积运算的过程，便可发现感受野的计算恰恰是卷积计算的逆过程（参考感受野计算）。

记  $F_k, S_k, P_k$  分别表示第  $k$  层的卷积核的高(或者宽)、移动步长（stride）、Padding 个数；记  $i_k$  表示第  $k$  层的输出特征图的高（或者宽）。这样，很容易得出如式 12.1 所示的递推公式：

$$i_{k+1} = \lfloor \frac{i_k - F_k + 2P_k}{S_k} \rfloor + 1 \quad (\text{式 12.1})$$

其中  $k \in \{1, 2, \dots\}$ ，且  $i_0$  表示原图的高或者宽。令  $t_k = \frac{F_k - 1}{2} - P_k$ ，上式可以转换为式 12.2：

$$(i_{k-1} - 1) = (i_k - 1)S_k + 2t_k \quad (\text{式 12.2})$$

反推感受野，令  $i_1 = F_1$ ，且  $t_k = \frac{F_k - 1}{2} - P_k$ ，且  $1 \leq j \leq L$ ，则有式 12.3：

$$i_0 = (i_L - 1)\alpha_L + \beta_L \quad (\text{式 12.3})$$

其中  $\alpha_L = \prod_{p=1}^L S_p$ ，且有式 12.4：

$$\beta_L = 1 + 2 \sum_{p=1}^L (\prod_{q=1}^{p-1} S_q) t_p \quad (\text{式 12.4})$$

由于 VGG16 的卷积核的配置均是  $\text{kernel\_size}=(3,3)$ ,  $\text{padding}=(1,1)$ ，同时只有在经过池化层才使得  $S_j = 2$ ，故而  $\beta_j = 0$ ，且有  $\alpha_L = 2^4 = 16$ 。

### 12.1.3 锚点的计算

在编程实现的时候，将感受野的大小使用  $\text{base\_size}$  来表示。下面讨论如何生成锚框？为了计算的方便，先定义一个 Box：

```

import numpy as np
class Box:
 # corner: Numpy, List, Tuple, MXNet.nd, torch.tensor
 def __init__(self, corner):
 self._corner = corner
 @property
 def corner(self):
 return self._corner
 @corner.setter
 def corner(self, new_corner):
 self._corner = new_corner
 @property
 def w(self):
 # 计算 bbox 的 宽
 return self.corner[2] - self.corner[0] + 1
 @property
 def h(self):
 # 计算 bbox 的 高
 return self.corner[3] - self.corner[1] + 1
 @property
 def area(self):
 # 计算 bbox 的 面积
 return self.w * self.h
 @property
 def whctr(self):
 # 计算 bbox 的 中心坐标
 assert isinstance(self.w, (int, float)), 'need int or float'
 xctr = self.corner[0] + (self.w - 1) * .5
 yctr = self.corner[1] + (self.h - 1) * .5
 return xctr, yctr
 def __and__(self, other):
 # 运算符 &，实现两个 box 的 交集运算
 xmin = max(self.corner[0], other.corner[0]) # xmin 中的大者
 xmax = min(self.corner[2], other.corner[2]) # xmax 中的小者
 ymin = max(self.corner[1], other.corner[1]) # ymin 中的大者
 ymax = min(self.corner[3], other.corner[3]) # ymax 中的小者
 w = xmax - xmin
 h = ymax - ymin
 if w < 0 or h < 0: # 两个边界框没有交集
 return 0
 else:
 return w * h
 def __or__(self, other):
 # 运算符 |，实现两个 box 的 并集运算
 l = self | other
 if l == 0:
 return 0
 else:
 return self.area + other.area - l

```

```

def IoU(self, other):
 # 计算 self & other
 if self == other:
 return 0
 else:
 U = self | other
 return I / U

```

类 Box 实现了 bbox 的交集、并集运算以及 IoU 的计算。下面举一个例子来说明：

```

bbox = [0, 0, 15, 15] # 边界框
bbox1 = [5, 5, 12, 12] # 边界框
A = Box(bbox) # 一个 bbox 实例
B = Box(bbox1) # 一个 bbox 实例

```

下面便可以输出 A 与 B 的高宽、中心、面积、交集、并集、IoU：

```

print('A 与 B 的交集', str(A & B))
print('A 与 B 的并集', str(A | B))
print('A 与 B 的 IoU', str(A.IoU(B)))
print('A 的中心、高、宽以及面积', str(A.whctrs), A.h, A.w, A.area)

```

输出结果：

```

A 与 B 的交集 49
A 与 B 的并集 271
A 与 B 的 IoU 0.18081180811808117
A 的中心、高、宽以及面积 (7.5, 7.5) 16 16 256

```

考虑到代码的可复用性，将 Box 封装进入 app/detection/bbox.py 中。下面重新考虑 load，首先定义一个转换函数：

```

def getX(img):
 # 将 img (h, w, 3) 转换为 (1, 3, w, h)
 img = img.transpose((2, 0, 1, 3))
 return np.expand_dims(img, 0)

```

函数 getX 将图片由(h, w, 3)转换为(1, 3, w, h)：

```

img, label = loader[0]
img = cv2.resize(img, (800, 800)) # resize 为 800 x 800
X = getX(img) # 转换为 (1, 3, w, h)
img.shape, X.shape

```

输出结果：

```
((800, 800, 3), (1, 3, 800, 800))

```

与此同时，获取特征图的数据：

```

features = net.features[:29]
F = features(imgs)
F.shape

```

输出：

```
(1, 512, 50, 50)

```

接着需要考虑如何将特征图 F 映射回原图？

#### 12.1.4 全卷积 (FCN)：将锚点映射回原图

faster R-CNN 中的 FCN 仅仅是有着 FCN 的特性，并不是真正意义上的卷积。faster R-CNN 仅仅是借用了 FCN 的思想来实现将特征图映射回原图的目的，同时将输出许多锚框。

特征图上的 1 个像素点的感受野为  $16 \times 16$ ，换言之，特征图上的锚点映射回原图的感受区域为  $16 \times 16$ ，论文称其为 reference box。下面相对于 reference box 依据不同的尺度与高宽比例来生成不同的锚框。

```
base_size = 2**4 # 特征图的每个像素的感受野大小
scales = [8, 16, 32] # 锚框相对于 reference box 的尺度
ratios = [0.5, 1, 2] # reference box 与锚框的高宽的比率 (aspect ratios)
```

其实 reference box 也对应于论文描述的 window（滑动窗口），这个之后再解释。我们先看看 scales 与 ratios 的具体含义。

为了更加一般化，假设 reference box 图片高宽分别为  $h, w$ ，而锚框的高宽分别为  $h_1, w_1$ ，形式化 scales 与 ratios 为公式 12.5：

$$\begin{cases} \frac{w_1 h_1}{wh} = s^2 \\ \frac{h_1}{w_1} = \frac{h}{w} r \Rightarrow \frac{h_1}{h} = \frac{w_1}{w} r \end{cases} \quad (\text{式 12.5})$$

可以将上式转换为公式 12.6：

$$\begin{cases} \frac{w_1}{w} = \frac{s}{\sqrt{r}} \\ \frac{h_1}{h} = \frac{w_1}{w} r = s\sqrt{r} \end{cases} \quad (\text{式 12.6})$$

同样可以转换为公式 12.7：

$$\begin{cases} w_s = \frac{w_1}{s} = \frac{w}{\sqrt{r}} \\ h_s = \frac{h_1}{s} = h\sqrt{r} \end{cases} \quad (\text{式 12.7})$$

基于公式 12.5 与公式 12.7 均可以很容易计算出  $w_1, h_1$ 。一般地， $w = h$ ，公式 12.7 亦可以转换为公式 12.8：

$$\begin{cases} w_s = \sqrt{\frac{wh}{r}} \\ h_s = w_s r \end{cases} \quad (\text{式 12.8})$$

gluoncv 结合公式 12.8 来编程，本文依据 12.7 进行编程。无论原图的尺寸如何，特征图的左上角第一个锚点映射回原图后的 reference box 的  $bbox = (x_{min}, y_{min}, x_{max}, y_{max})$  均为  $(0, 0, base\_size-1, base\_size-1)$ ，为了方便称呼，我们称其为 base\_reference box。基于 base\_reference box 依据不同的  $s$  与  $r$  的组合生成不同尺度和高宽比的锚框，且称其为 base\_anchors。编程实现：

```
class MultiBox(Box):
 def __init__(self, base_size, ratios, scales):
 if not base_size:
 raise ValueError("Invalid base_size: {}".format(base_size))
```

```

if not isinstance(ratios, (tuple, list)):
 ratios = [ratios]
if not isinstance(scales, (tuple, list)):
 scales = [scales]
super().__init__([0]*2+[base_size-1]*2) # 特征图的每个像素的感受野大小为 base_size
reference box 与 锚 框 的 高 宽 的 比 率 (aspect ratios)
self._ratios = np.array(ratios)[:], None]
self._scales = np.array(scales) # 锚 框 相 对 于 reference box 的 尺 度
@property
def base_anchors(self):
 ws = np.round(self.w / np.sqrt(self._ratios))
 w = ws * self._scales
 h = w * self._ratios
 wh = np.stack([w.flatten(), h.flatten()], axis=1)
 wh = (wh - 1) * .5
 return np.concatenate([self.whctrs - wh, self.whctrs + wh], axis=1)
def _generate_anchors(self, stride, alloc_size):
 # propagete to all locations by shifting offsets
 height, width = alloc_size # 特 征 图 的 尺 寸
 offset_x = np.arange(0, width * stride, stride)
 offset_y = np.arange(0, height * stride, stride)
 offset_x, offset_y = np.meshgrid(offset_x, offset_y)
 offsets = np.stack((offset_x.ravel(), offset_y.ravel()), axis=1)
 # broadcast_add (1, N, 4) + (M, 1, 4)
 anchors = (self.base_anchors.reshape((1, -1, 4)) + offsets.reshape((-1, 1, 4)))
 anchors = anchors.reshape((1, 1, height, width, -1)).astype(np.float32)
 return anchors

```

下面看看具体效果：

```

base_size = 2**4 # 特 征 图 的 每 个 像 素 的 感 受 野 大 小
scales = [8, 16, 32] # 锚 框 相 对 于 reference box 的 尺 度
ratios = [0.5, 1, 2] # reference box 与 锚 框 的 高 宽 的 比 率 (aspect ratios)
A = MultiBox(base_size,ratios, scales)
A.base_anchors

```

输出结果：

```

array([[-84., -38., 99., 53.],
 [-176., -84., 191., 99.],
 [-360., -176., 375., 191.],
 [-56., -56., 71., 71.],
 [-120., -120., 135., 135.],
 [-248., -248., 263., 263.],
 [-36., -80., 51., 95.],
 [-80., -168., 95., 183.],
 [-168., -344., 183., 359.]])

```

接着考虑将 base\_anchors 在整个原图上进行滑动。比如，特征图的尺寸为 (5, 5) 而感受野的大小为 50，则 base\_reference\_box 在原图滑动的情况（移动步长为 50）如下图：

```

x, y = np.mgrid[0:300:50, 0:300:50]
plt.pcolor(x, y, x+y); # x 和 y 是网格,z 是(x,y)坐标处的颜色值 colorbar()

```

输出结果如图 12.2 所示。

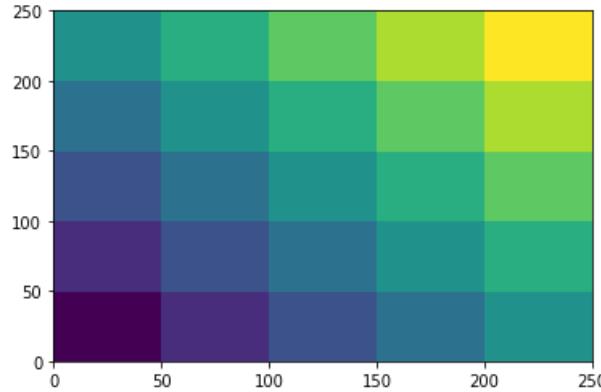


图 12.2 np.mgrid 的使用示例图

原图被划分为了 25 个 block，每个 block 均代表一个 reference box。若 base\_anchors 有 9 个，则只需要按照 stride = 50 进行滑动便可以获得这 25 个 block 的所有锚框（总计  $5 \times 5 \times 9 = 225$  个）。针对前面的特征图 F 有：

```
stride = 16 # 滑 动 的 步 长
alloc_size = F.shape[2:] # 特 征 图 的 尺 寸
anchors = A_.generate_anchors(stride, alloc_size).shape
输出结果:
```

```
(1, 1, 50, 50, 36)
```

即总共  $50 \times 50 \times 9 = 22500$  个锚点（anchors 数量庞大且必然有许多的高度重叠的框）。至此，我们生成初始锚框的过程便结束了，同时很容易发现，anchors 的生成仅仅借助 Numpy 便完成了，这样做十分有利于代码迁移到 Pytorch、TensorFlow 等支持 Numpy 作为输入的框架。下面仅仅考虑 MXNet，其他框架以后再讨论。下面先看看 MultiBox 的设计对于使用 MXNet 进行后续的开发有什么好处吧！

由于 base-net（基网络）的结构一经确定便是固定的，针对不同尺寸的图片，如果每次生成 anchors 都要重新调用 A\_.generate\_anchors()一次，那么将会产生很多的不必要的冗余计算，gluoncv 提供了一种十分不错的思路：先生成 base\_anchors，然后选择一个比较大的尺度 alloc\_size（比如  $128 \times 128$ ）用来生成锚框的初选模板；接着把真正的特征图传入到 RPNAnchorGenerator 并通过前向传播计算得到特征图的锚框。具体操作细节见如下代码：

```
class RPNAncorGenerator(gluon.HybridBlock):
 """ 生 成 RPN 的 锚 框
参

stride : int
 特征图相对于原图的滑动步长，或是说是特征图上单个像素点的感受野。
base_size : int
 reference anchor box 的 宽 或 者 高
ratios : iterable
 anchor boxes 的 aspect ratios (高宽比)。我们期望它是 tuple 或者 list
scales : iterable
 锚框相对于 reference anchor boxes 的 尺度
 采 用 如 下 形 式 计 算 锚 框 的 高 和 宽 :
```

```

 ..
 width_{anchor} = size_{base} \times scale \times \sqrt{1 / ratio}
 height_{anchor} = width_{anchor} \times ratio
 alloc_size : tuple of int
 预设锚框的尺寸为 (H, W)，通常用来生成比较大的特征图（如 128x128）。
 在推断的后期，我们可以有可变的输入大小，在这个时候，我们可以从这个大的 anchor map 中直接裁剪出对应的 anchors，以便我们可以避免在每次输入都要重新生成锚点。
 """
 def __init__(self, alloc_size, base_size, ratios, scales, **kwargs):
 super().__init__(**kwargs)
 # 生成锚框初选模板，之后通过切片获取特征图的真正锚框
 anchors = MultiBox(base_size, ratios, scales).generate_anchors(
 base_size, alloc_size)
 self.anchors = self.params.get_constant('anchor_', anchors)
 # pylint: disable=arguments-differ
 def hybrid_forward(self, F, x, anchors):
 """Slice anchors given the input image shape.
 Inputs:
 - **x**: input tensor with (1 x C x H x W) shape.
 Outputs:
 - **out**: output anchor with (1, N, 4) shape. N is the number of anchors.
 """
 a = F.slice_like(anchors, x, *0, axes=(2, 3))
 return a.reshape((1, -1, 4))

```

上面的 RPNArchGenerator 直接改写自 gluoncv。看看 RPNArchGenerator 的魅力所在：

```

base_size = 2**4 # 特征图的每个像素的感受野大小
scales = [8, 16, 32] # 锚框相对于 reference box 的尺度
ratios = [0.5, 1, 2] # reference box 与锚框的高宽的比率 (aspect ratios)
stride = base_size # 在原图上滑动的步长
alloc_size = (128, 128) # 一个比较大的特征图的锚框生成模板
调用 RPNArchGenerator 生成 anchors
A = RPNArchGenerator(alloc_size, base_size, ratios, scales)
A.initialize()
A(F) # 直接传入特征图 F, 获取 F 的 anchors

```

输出结果：

```

[[[-84. -38. 99. 53.]
 [-176. -84. 191. 99.]
 [-360. -176. 375. 191.]
 ...
 [748. 704. 835. 879.]
 [704. 616. 879. 967.]
 [616. 440. 967. 1143.]]]
<NDArray 1x22500x4 @cpu(0)>

```

shape = 1x22500x4 符合我们的预期。如果我们更改特征图的尺寸：

```

x = nd.zeros((1, 3, 75, 45))
A(x).shape

```

输出结果：

```
(1, 30375, 4)
```

这里 $30375 = 75 \times 45 \times 9$ 也符合我们的预期。至此，我们完成了将特征图上的所有锚点映射回原图生成锚框的工作！

## 12.2 平移不变性的锚点

反观上述的编程实现，很容易便可理解论文提到的锚点的平移不变性。无论是锚点的生成还是锚框的生成都是基于 `base_reference_box` 进行平移和卷积运算（亦可看作是一种线性变换）的。为了叙述方便下文将 `RPNAnchorGenerator`（被放置在 `app/detection/anchor.py`）生成的 anchor boxes 由 `corner`（记作 A 坐标形式：(xmin,ymin,xmax,ymax)）转换为 `center`（形式为：(xctr,yctr,w,h)）后的锚框记作 B。

其中(xmin,ymin),(xmax,ymax)分别表示锚框的最小值与最大值坐标；(xctr,yctr)表示锚框的中心坐标，w,h 表示锚框的宽和高。且记  $a = (x_a, y_a, w_a, h_a) \in B$ ，即使用下标a来标识锚框。A 与 B 是锚框的两种不同的表示形式。

在 `gluoncv.nn.bbox` 中提供了将A转换为B的模块：`BBoxCornerToCenter`。下面便利用其进行编程。先载入环境：

```
cd ./app/
```

接着载入本小节所需的包：

```
from mxnet import init, gluon, autograd
from mxnet.gluon import nn
from gluoncv.nn.bbox import BBoxCornerToCenter
自定义包
from detection.bbox import MultiBox
from detection.anchor import RPNAnchorGenerator
```

为了更加容易理解 A 与 B 的处理过程，下面先自创一个类(之后会抛弃)：

```
class RPNNProposal(nn.HybridBlock):
 def __init__(self, channels, stride, base_size, ratios, scales, alloc_size, **kwargs):
 super().__init__(**kwargs)
 weight_initializer = init.Normal(0.01)
 with self.name_scope():
 self.anchor_generator = RPNAnchorGenerator(
 stride, base_size, ratios, scales, alloc_size)
 anchor_depth = self.anchor_generator.num_depth
 # conv1 的创建
 self.conv1 = nn.HybridSequential()
 self.conv1.add(nn.Conv2D(channels, 3, 1, 1,
 weight_initializer=weight_initializer))
 self.conv1.add(nn.Activation('relu'))
 # score 的创建
 # use sigmoid instead of softmax, reduce channel numbers
 self.score = nn.Conv2D(anchor_depth, 1, 1, 0,
 weight_initializer=weight_initializer)
 # loc 的创建
 self.loc = nn.Conv2D(anchor_depth * 4, 1, 1, 0,
 weight_initializer=weight_initializer)
 # 具体的操作
 # 如下
```

```

channels = 2**4 # 特征图的每个像素的感受野大小
base_size = [8, 16, 32] # 锚框相对于 reference box 的尺度
ratios = [0.5, 1, 2] # reference box 与锚框的高宽的比率 (aspect ratios)
stride = base_size # 在原图上滑动的步长
alloc_size = (128, 128) # 一个比较大的特征图的锚框生成模板
self = RPNProposal(channels, stride, base_size, ratios, scales, alloc_size)
self.initialize()

```

下面便可以看看如何将A转换为B:

```

img, label = loader[0] # 载入图片和标注信息
img = cv2.resize(img, (800, 800)) # resize 为 (800, 800)
imgs = nd.array(getX(img)) # 转换为 MXNet 的输入形式
xs = features(imgs) # 获取特征图张量
F = nd
A = self.anchor_generator(xs) # (xmin,ymin,xmax,ymax) 形式的锚框
box_to_center = BBoxCornerToCenter()
B = box_to_center(A) # (x,y,w,h) 形式的锚框

```

### 12.2.1 边界框回归

手动设计的锚框 B 并不能很好的满足后续的 Fast R-CNN 的检测工作，还需要借助论文介绍的 3 个卷积层：conv1、score、loc。对于论文中的  $3 \times 3$  的卷积核 conv1 我的理解是模拟锚框的生成过程：通过不改变原图尺寸的卷积运算达到降维的目标，同时有着在原图滑动尺寸为 base\_size 的 reference box 的作用。

换言之，conv1 的作用是模拟生成锚点。假设通过 RPN 生成的边界框 bbox 记为  $G = \{p: (x, y, w, h)\}$ ，利用  $1 \times 1$  卷积核 loc 预测  $p$  相对于每个像素点（即锚点）所生成的  $k$  个锚框的中心坐标与高宽的偏移量，利用  $1 \times 1$  卷积核 score 判别锚框是目标（objectness, foreground）还是背景（background）。

记真实的边界框集合为  $G^* = \{p^*: (x^*, y^*, w^*, h^*)\}$ 。其中， $(x, y), (x^*, y^*)$  分别代表预测边界框、真实边界框的中心坐标； $(w, h), (w^*, h^*)$  分别代表预测边界框、真实边界框的的宽与高。论文在 Training RPNs 中提到，在训练阶段 conv1、loc、score 使用均值为 0，标准差为 0.01 的高斯分布来随机初始化。接着，看看如何使用 conv1、loc、score：

```

x = self.conv1(xs)
score 的输出
raw_rpn_scores = self.score(x).transpose(axes=(0, 2, 3, 1)).reshape((0, -1, 1))
rpn_scores = F.sigmoid(F.stop_gradient(raw_rpn_scores)) # 转换为概率形式
loc 的输出
rpn_box_pred = self.loc(x).transpose(axes=(0, 2, 3, 1)).reshape((0, -1, 4))

```

卷积核 loc 的作用是用来学习偏移量的，在论文中给出了公式 12.9：

$$\begin{cases} t_x = \frac{x - x_a}{w_a} & t_y = \frac{y - y_a}{h_a} \\ t_x^* = \frac{x^* - x_a}{w_a} & t_y^* = \frac{y^* - y_a}{h_a} \\ t_w = \log\left(\frac{w}{w_a}\right) & t_h = \log\left(\frac{h}{h_a}\right) \\ t_w^* = \log\left(\frac{w^*}{w_a}\right) & t_h^* = \log\left(\frac{h^*}{h_a}\right) \end{cases} \quad (\text{式 12.9})$$

这样可以很好的消除图像尺寸的不同带来的影响。为了使得修正后的锚框  $G$  具备与真实边界框  $G^*$  有相同的均值和标准差，还需要设定： $\sigma = (\sigma_x, \sigma_y, \sigma_w, \sigma_h)$ ,  $\mu = (\mu_x, \mu_y, \mu_w, \mu_h)$  表示  $G$  的  $(x, y, w, h)$  对应的标准差与均值。故而，为了让预测的边界框的偏移量的分布更加均匀还需要将坐标转换一下，如式 12.10 所示。

$$\begin{cases} t_x = \frac{\frac{x - x_a}{w_a} - \mu_x}{\sigma_x} \\ t_y = \frac{\frac{y - y_a}{h_a} - \mu_y}{\sigma_y} \\ t_w = \frac{\log\left(\frac{w}{w_a}\right) - \mu_w}{\sigma_w} \\ t_h = \frac{\log\left(\frac{h}{h_a}\right) - \mu_h}{\sigma_h} \end{cases} \quad (\text{式 12.10})$$

对于  $G^*$  也是一样的操作。（略去）一般地， $\sigma = (0.1, 0.1, 0.2, 0.2)$ ,  $\mu = (0, 0, 0, 0)$ 。

由于 loc 的输出便是  $\{(t_x, t_y, t_w, t_h)\}$ ，下面需要反推  $(x, y, w, h)$ ，如式 12.11 所示。

$$\begin{cases} x = (t_x \sigma_x + \mu_x) w_a + x_a \\ y = (t_y \sigma_y + \mu_y) h_a + y_a \\ w = w_a e^{t_w \sigma_w + \mu_w} \\ h = h_a e^{t_h \sigma_h + \mu_h} \end{cases} \quad (\text{式 12.11})$$

通常情况下，A 形式的边界框转换为 B 形式的边界框被称为编码（encode），反之，则称为解码（decode）。在 gluoncv.nn.coder 中的 NormalizedBoxCenterDecoder 类实现了上述的转换过程，同时也完成了 G 解码工作。

```
from gluoncv.nn.coder import NormalizedBoxCenterDecoder
stds = (0.1, 0.1, 0.2, 0.2)
means = (0., 0., 0., 0.)
box_decoder = NormalizedBoxCenterDecoder(stds, means)
roi = box_decoder(rpn_box_pred, B) # 解码后的 G
```

## 12.2.2 裁剪预测边界框超出原图边界的边界

为了保持一致性，需要重写 `getX`:

```
def img = img.transpose((2, 0, 1)) # 将 img (h, w, 3) 转换为 (1, 3, h, w)
```

```
 return np.expand_dims(img, 0)
```

考虑到 RPN 的输入可以是批量数据:

```
imgs = []
labels = []
for img, label in zip(imgs, labels):
 img = cv2.resize(img, (600, 800)) # resize 宽高为 (600, 800)
 imgs.append(getX(img))
 labels.append(label)
imgs = nd.array(np.concatenate(imgs)) # 一个批量的图片
labels = nd.array(np.stack(labels)) # 一个批量的标注信息
```

这样便有:

```
from gluoncv.nn.coder import NormalizedBoxCenterDecoder
from gluoncv.nn.bbox import BBoxCornerToCenter
stds = (0.1, 0.1, 0.2, 0.2)
means = (0., 0., 0., 0.)

fs = features(imgs) # 获取特征图张量
F = nd
A = self.anchor_generator(fs) # (xmin,ymin,xmax,ymax) 形式的锚框
box_to_center = BBoxCornerToCenter()
B = box_to_center(A) # (x,y,w,h) 形式的锚框
x = self.conv1(fs) # conv1 卷积
raw_rpn_scores = self.score(x).transpose(axes=(0, 2, 3, 1)).reshape((0, -1, 1)) # 激活之前的 score
rpn_scores = F.sigmoid(F.stop_gradient(raw_rpn_scores)) # 激活后的 score
rpn_box_pred = self.loc(x).transpose(axes=(0, 2, 3, 1)).reshape((0, -1, 4)) # loc 预测偏移量
(tx,ty,tw,yh)
box_decoder = NormalizedBoxCenterDecoder(stds, means)
roi = box_decoder(rpn_box_pred, B) # 解码后的 G
print(roi.shape)
```

此时，便有两张图片的预测 G:

```
(2, 16650, 4)
```

因为此时生成的 RoI 有许多超出边界的框，所以，需要进行裁减操作。先裁剪掉所有小于 0 的边界:

```
x = F.maximum(roi, 0.0)
```

nd.maximum(x) 的作用是  $\max\{0, x\}$ 。接下来裁剪掉大于原图的边界的边界:

```
shape = F.shape_array(imgs) # imgs 的 shape
size = shape.slice_axis(axis=0, begin=2, end=None) # imgs 的尺寸
window = size.expand_dims(0)
```

输出结果:

```
[[800 600]
 <NDArray 1x2 @cpu(0)>
```

此时是(高, 宽)的形式，而锚框的是以(宽, 高)的形式生成的，故而还需要:

```
F.reverse(window, axis=1)
```

结果:

```
[[600 800]
 <NDArray 1x2 @cpu(0)>
```

因而，下面的 m 可以用来判断是否超出边界：

```
m = F.tile(F.reverse(window, axis=1), reps=(2,)).reshape((0, -4, 1, -1))
m
```

结果：

```
[[[600 800 600 800]]]
<NDArray 1x1x4 @cpu(0)>
```

接着，便可以获取裁剪之后的 RoI：

```
roi = F.broadcast_minimum(x, F.cast(m, dtype='float32'))
```

整个裁剪工作可以通过如下操作简单实现：

```
from gluoncv.nn.bbox import BBoxClipToImage
clipper = BBoxClipToImage()
roi = clipper(roi, imgs) # 裁剪超出边界的边界
```

### 12.2.3 移除小于 min\_size 的边界框

移除小于 min\_size 的边界框进一步筛选边界框：

```
min_size = 5 # 最小锚框的尺寸
xmin, ymin, xmax, ymax = roi.split(axis=-1, num_outputs=4) # 拆分坐标
width = xmax - xmin # 锚框宽度的集合
height = ymax - ymin # 锚框高度的集合
invalid = (width < min_size) + (height < min_size) # 所有小于 min_size 的高宽
```

由于张量的<运算有一个特性：满足条件的设置为 1，否则为 0。这样一来两个这样的运算相加便可筛选出同时不满足条件的对象：

```
cond = invalid[0,:10]
cond.T
```

结果：

```
[[1. 0. 0. 0. 1. 0. 1. 1. 0. 2.]]
<NDArray 1x10 @cpu(0)>
```

可以看出有 2 存在，代表着两个条件都满足，我们可以做筛选如下：

```
F.where(cond, F.ones_like(cond)* -1, rpn_scores[0,:10]).T
```

结果：

```
[-1. 0.999997 0.0511509 0.9994136 -1. 0.00826993 -1. -1. 0.99783903 -1.]
<NDArray 1x10 @cpu(0)>
```

由此可以筛选出所有不满足条件的对象。更进一步，筛选出所有不满足条件的对象：

```
score = F.where(invalid, F.ones_like(invalid) * -1, rpn_scores) # 筛选 score
invalid = F.repeat(invalid, axis=-1, repeats=4)
roi = F.where(invalid, F.ones_like(invalid) * -1, roi) # 筛选 RoI
```

## 12.3 NMS (Non-maximum suppression)

先总结 RPN 的前期工作中 Proposal 的生成阶段：

- (1) 利用 base\_net 获取原图 I 对应的特征图 X；
- (2) 依据 base\_net 的网络结构计算特征图的感受野大小为 base\_size；
- (3) 依据不同的 scale 和 aspect ratio 通过 MultiBox 计算特征图 X 在 (0,0) 位置的锚点对应

的k个锚框 base\_anchors:

- (4) 通过 RPNAnchorGenerator 将X映射回原图并生成 corner 格式(xmin,ymin,xmax,ymax) 的锚框A;
- (5) 将锚框A转换为 center 格式(x,y,w,h), 记作B;
- (6) X通过卷积 conv1 获得模拟锚点, 亦称之为 rpn\_features;
- (7) 通过卷积层 score 获取 rpn\_features 的得分 rpn\_score;
- (8) 与 7 并行的通过卷积层 loc 获取 rpn\_features 的边界框回归的偏移量 rpn\_box\_pred;
- (9) 依据 rpn\_box\_pred 修正锚框B并将其解码为G;
- (10) 裁剪掉G的超出原图尺寸的边界, 并移除小于 min\_size 的边界框。

虽然上面的步骤移除了许多无效的边界并裁剪掉超出原图尺寸的边界, 但是, 可以想象到 G 中必然存在许多的高度重叠的边界框, 此时若将G当作 Region Proposal 送入 PoI Pooling 层将给计算机带来十分庞大的负载, 并且G中的背景框远远多于目标极为不利于模型的训练。

论文中给出了 NMS 的策略来解决上述难题。根据我们预测的 rpn\_score, 对 G 进行非极大值抑制操作 (NMS), 去除得分较低以及重复区域的 RoI。在 MXNet 提供了 nd.contrib.box\_nms 来实现此次目标:

```
nms_thresh = 0.7
n_train_pre_nms = 12000 # 训练时 nms 之前的 bbox 的数目
n_train_post_nms = 2000 # 训练时 nms 之后的 bbox 的数目
n_test_pre_nms = 6000 # 测试时 nms 之前的 bbox 的数目
n_test_post_nms = 300 # 测试时 nms 之后的 bbox 的数目
pre = F.concat(scores, rois, dim=-1) # 合并 score 与 roi
非极 大 值 抑制
tmp = F.contrib.box_nms(pre, overlap_thresh=nms_thresh, topk=n_train_pre_nms,
coord_start=1, score_index=0, id_index=-1, force_suppress=True)
slice post_nms number of boxes
result = F.slice_axis(tmp, axis=1, begin=0, end=n_train_post_nms)
rpn_scores = F.slice_axis(result, axis=-1, begin=0, end=1)
rpn_bbox = F.slice_axis(result, axis=-1, begin=1, end=None)
```

上述的封装比较彻底, 无法获取具体的实现, 并且也不利于我们理解 NMS 的具体实现原理。为了更好的理解 NMS, 自己重新实现 NMS 是十分有必要的。

将 scores 按照从大到小进行排序, 并返回其索引:

```
scores = scores.flatten() # 去除无效维度
将 scores 按照从大到小进行排序, 并返回其索引
orders = scores.argsort()[:,::-1]
```

由于 loc 生成的锚框实在是太多了, 为此, 仅仅考虑得分前 n\_train\_pre\_nms 的锚框:

```
keep = orders[:n_train_pre_nms]
```

下面先考虑一张图片, 之后再考虑多张图片一起训练的情况:

```
order = keep[0] # 第一张图片的得分降序索引
score = scores[0][order] # 第一张图片的得分预测降序排列
roi = rois[0][order] # 第一张图片的边界框预测按得分降序排列
label = labels[0] # 真实边界框
```

虽然 Box 支持 nd 作为输入, 但是计算多个 IoU 效率并不高:

```
%%timeit
GT = [Box(corner) for corner in label] # 真实边界框实例化
G = [Box(corner) for corner in roi] # 预测边界框实例化
```

```

ious = nd.zeros((len(G), len(GT))) # 初始化 IoU 的计算
for i, A in enumerate(G):
 for j, B in enumerate(GT):
 iou = A.IoU(B)
 ious[i, j] = iou

```

输出计时结果:

```
1min 10s ± 6.08 s per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

先转换为 Numpy 再计算 IoU 效率会更高:

```

%%timeit
GT = [Box(corner) for corner in label.asarray()] # 真实边界框实例化
G = [Box(corner) for corner in roi.asarray()] # 预测边界框实例化
ious = nd.zeros((len(G), len(GT))) # 初始化 IoU 的计算
for i, A in enumerate(G):
 for j, B in enumerate(GT):
 iou = A.IoU(B)
 ious[i, j] = iou

```

输出计时结果:

```
6.88 s ± 410 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

比使用 nd 快了近 10 倍！但是如果全部使用 Numpy，会有什么变化？

```

%%timeit
GT = [Box(corner) for corner in label.asarray()] # 真实边界框实例化
G = [Box(corner) for corner in roi.asarray()] # 预测边界框实例化
ious = np.zeros((len(G), len(GT))) # 初始化 IoU 的计算
for i, A in enumerate(G):
 for j, B in enumerate(GT):
 iou = A.IoU(B)
 ious[i, j] = iou

```

输出计时结果:

```
796 ms ± 30.5 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

速度又提升了 10 倍左右！为此，仅仅考虑使用 Numpy 来计算。将其封装进 group\_iou 函数：

```

def group_iou(pred_bbox, true_bbox):
 # 计算 pred_bbox 与 true_bbox 的 IoU 组合
 GT = [Box(corner) for corner in true_bbox] # 真实边界框实例化
 G = [Box(corner) for corner in pred_bbox] # 预测边界框实例化
 ious = np.zeros((len(G), len(GT))) # 初始化 IoU 的计算
 for i, A in enumerate(G):
 for j, B in enumerate(GT):
 iou = A.IoU(B)
 ious[i, j] = iou
 return ious

```

不过，还有更加友好的计算 ious 的方法，将会在下节介绍！

### 12.3.1 重构代码

前面的代码有点混乱，为了后续开发的便利，我们先重新整理一下代码。先仅仅考虑一张图片，下面先载入设置 RPN 网络的输入以及部分输出：

```

PRN 前期的设定
channels = 256 # conv1 的输出通道数
base_size = 2**4 # 特征图的每个像素的感受野大小
scales = [8, 16, 32] # 锚框相对于 reference box 的尺度
ratios = [0.5, 1, 2] # reference box 与锚框的高宽的比率 (aspect ratios)
stride = base_size # 在原图上滑动的步长
alloc_size = (128, 128) # 一个比较大的特征图的锚框生成模板
用 来 辅 助 理 解 RPN 的类
self = RPNProposal(channels, stride, base_size, ratios, scales, alloc_size)
self.initialize() # 初始化卷积层 conv1, loc, score
stds = (0.1, 0.1, 0.2, 0.2) # 偏移量的标准差
means = (0., 0., 0., 0.) # 偏移量的均值
锚框的编码
box_to_center = BBoxCornerToCenter() # 将 (xmin,ymin,xmax,ymax) 转换为 (x,y,w,h)
将锚框通过偏移量进行修正，并解码为 (xmin,ymin,xmax,ymax)
box_decoder = NormalizedBoxCenterDecoder(stds, means)
clipper = BBoxClipToImage() # 裁剪超出原图尺寸的边界
获取 COCO 的一张图片用来做实验
img, label = loader[0] # 获取一张图片
img = cv2.resize(img, (800, 800)) # resize 为 (800, 800)
imgs = nd.array(getX(img)) # 转换为 MXNet 的输入形式
提取最后一层卷积积的特征
net = vgg16(pretrained=True) # 载入基网络的权重
features = net.features[:29] # 卷积层特征提取器
fs = features(imgs) # 获取特征图张量
A = self.anchor_generator(fs) # 生成 (xmin,ymin,xmax,ymax) 形式的锚框
B = box_to_center(A) # 编码为 (x,y,w,h) 形式的锚框
x = self.conv1(fs) # conv1 卷积
sigmoid 激活之前 的 score
raw_rpn_scores = self.score(x).transpose(axes=(0, 2, 3, 1)).reshape((0, -1, 1))
rpn_scores = nd.sigmoid(nd.stop_gradient(raw_rpn_scores)) # 激活后的 score
loc 预测偏移量 (tx,ty,tw,th)
rpn_box_pred = self.loc(x).transpose(axes=(0, 2, 3, 1)).reshape((0, -1, 4))
修正正锚框的坐标
roi = box_decoder(rpn_box_pred, B) # 解码后的预测边界框 G (Rois)
print(roi.shape) # 裁剪剪之前
roi = clipper(roi, imgs) # 裁剪超出原图尺寸的边界

```

虽然，`roi` 已经裁剪掉超出原图尺寸的边界，但是还有一部分边界框实在有点儿小，不利于后续的训练，故而需要丢弃。丢弃的方法是将其边界框与得分均设置为-1：

```

def size_control(F, min_size, rois, scores):
 # 拆分坐标
 xmin, ymin, xmax, ymax = rois.split(axis=-1, num_outputs=4)
 width = xmax - xmin # 锚框宽度的集合
 height = ymax - ymin # 锚框高度的集合
 # 获取所有小于 min_size 的高宽
 invalid = (width < min_size) + (height < min_size) # 同时满足条件
 # 将不满足条件的锚框的坐标与得分均设置为 -1
 scores = F.where(invalid, F.ones_like(invalid) * -1, scores)
 invalid = F.repeat(invalid, axis=-1, repeats=4)

```

```

 rois = F.where(invalid, F.ones_like(invalid) * -1, rois)
 return
 min_size = 16 # 最小锚框的尺寸
 pre_nms = 12000 # nms 之前 的 bbox 的数目
 post_nms = 2000 # ms 之后 的 bbox 的数目
 rois, scores = size_control(nd, min_size, roi, rpn_scores)

```

为了可以让 Box 一次计算多个 bbox 的 IoU，下面需要重新改写 Box:

```

class BoxTransform(Box):
 """
 一组 bbox 的运算
 """

 def __init__(self, F, corners):
 """
 F 可以是 mxnet.nd, numpy, torch.tensor
 """

 super().__init__(corners)
 self.corner = corners.T
 self.F = F

 def __and__(self, other):
 """
 运算符 `&` 交集运算
 """

 xmin = self.F.maximum(self.corner[0].expand_dims(0), other.corner[0].expand_dims(1)) # xmin 中的大者
 xmax = self.F.minimum(self.corner[2].expand_dims(0), other.corner[2].expand_dims(1)) # xmax 中的小者
 ymin = self.F.maximum(self.corner[1].expand_dims(0), other.corner[1].expand_dims(1)) # ymin 中的大者
 ymax = self.F.minimum(self.corner[3].expand_dims(0), other.corner[3].expand_dims(1)) # ymax 中的小者

 w = xmax - xmin
 h = ymax - ymin
 cond = (w <= 0) + (h <= 0)
 I = self.F.where(cond, nd.zeros_like(cond), w * h)
 return I

 def __or__(self, other):
 """
 运算符 `|` 并集运算
 """

 I = self & other
 U = self.area.expand_dims(0) + other.area.expand_dims(1) - I
 return self.F.where(U < 0, self.F.zeros_like(I), U)

 def IoU(self, other):
 """
 交并比
 """

 I = self & other
 U = self | other
 return self.F.where(U == 0, self.F.zeros_like(I), I / U)

```

先测试一下：

```
a = BoxTransform(nd, nd.array([[0, 0, 15, 15]]))
b = BoxTransform(nd, 1+nd.array([[0, 0, 15, 15]]))
c = BoxTransform(nd, nd.array([[-1, -1, -1, -1]]))
```

创建了两个简单有效的 bbox (a, b) 和一个无效的 bbox (c)，接着看看它们的运算：

```
a & b, a | b, a.IoU(b)
```

输出结果：

```
([[[196.]]] <NDArray 1x1x1 @cpu(0)>, [[[316.]]] <NDArray 1x1x1 @cpu(0)>, [[0.62025315]]] <NDArray
1x1x1 @cpu(0)>)
```

而与无效的边界框的计算便是：

```
a & c, a | c, a.IoU(c)
```

输出结果：

```
([[0.]]
<NDArray 1x1x1 @cpu(0)>,
[[257.]]
<NDArray 1x1x1 @cpu(0)>,
[[0.]]
<NDArray 1x1x1 @cpu(0)>)
```

如果把无效的边界框看作是空集，则上面的运算结果便符合常识。下面讨论如何标记训练集？

COCO 的 bbox 是 (x,y,w,h) 格式的，但是这里的 (x,y) 不是中心坐标，而是左上角坐标。为了统一，需要将其转换为 (xmin, ymin, xmax, ymax) 格式：

```
labels = nd.array(label) # (x,y,w,h) , 其中 (x,y) 是左上角坐标
cwh = (labels[:,2:4]-1) * 0.5
labels[:,2:4] = labels[:,2:] + cwh # 转换为 (xmin, ymin, xmax, ymax)
```

下面计算真实边界框与预测边界框的 ious：

```
将 scores 按照从大到小进行排序，并返回其索引
orders = scores.reshape((0, -1)).argsort()[:, ::-1][:,pre_nms]
scores = scores[0][orders[0]] # 得分降序排列
rois = rois[0][orders[0]] # 按得分降序排列 rois
预测边界的框
G = BoxTransform(nd, rois.expand_dims(0))
真实边界的框
GT = BoxTransform(nd, labels[:,4].expand_dims(0))
ious = G.IoU(GT).T # 计算全部的 iou
ious.shape
```

输出结果：

```
(1, 12000, 6)
```

可以看出，总计一张图片，预测边界框 12000 个，真实边界框 6 个。

### 12.3.2 标注边界框

在训练阶段：NMS 直接使用 `nd.contrib.box_nms` 便可以实现，直接使用 `BoxTransform` 来构造 nms 的计算，暂时没有比较好的思路（还不如直接使用 for 循环来的简单，虽然它很低

效）。故而，暂时先将 BoxTransform 雪藏，待到想到比较不错的点子再来考虑。

编程暂且不提，这里主要讲讲如何标注训练集？上面利用 BoxTransform 已经计算出 矩阵 ious。下面就以此为基础进行 NMS 操作：

(1) 找出 ious 中的最大值，并定位其坐标（ious 中元素的索引），记作  $i_1, j_1$ ，即将  $i_1$  分配给  $j_1$ ；

(2) 删除  $i_1$  行所有元素，并删除  $j_1$  列所有元素，此时记 ious 为  $X_1$ ；

(3) 找出  $X_1$  中的最大值，并定位其坐标（ious 中元素的索引），记作  $i_2, j_2$ ，即将  $i_2$  分配给  $j_2$ ；

(4) 删除  $i_2$  行所有元素，并删除  $j_2$  列所有元素，此时记 ious 为  $X_2$ ；

(5) 不断重复上述操作，直至删除所有列为止；

(6) 对于剩余的行，通过阈值判断是否为锚框分配真实边界框。

标注训练集时不仅仅需要标注边界框的类别还要标注其相对于真实边界框的偏移量。

测试集的标注与训练集不同，因为，此时没有真实边界框，所以以当前边界框的得分最大者对应的类别标签来标注类别，接着计算剩余预测框与其的 IoU 并移除与其相似的边界框。不断的重复这种操作，直至无法移除边界框为止。（MXNet 中的 contrib.ndarray.MultiBoxDetection 实现该操作。）

NMS 那一节已经通过 RPN 提取出 2000 左右 Region Proposal（gluoncv 通过 RPNProposal 实现），这些 Region Proposal 作为 RoIs 当作 Fast R-CNN 的输入。

## 12.4 RoIHead

RPN 的整个流程如图 12.3 所示。

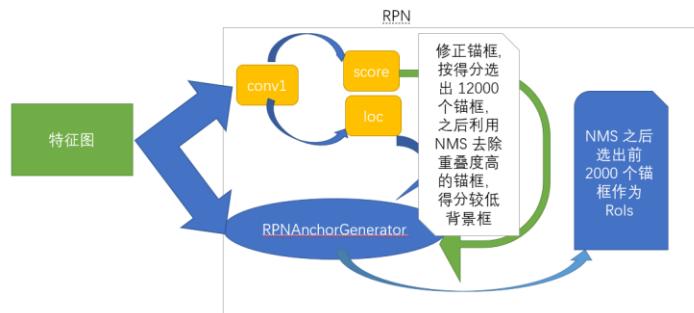


图 12.3 RPN 的流程图

RPN 去除了大量的重复锚框以及得分低的背景区域最后得到 2000 左右 RoIs 送入 RoIHead 再一次调整锚框的标注信息。虽然将 2000 个 RoIs 都作为 RoIHead 的输入是可以实现的，但是这样做会造成正负样本的不均衡问题，为此，论文提出均匀采用的策略：随机选择 256 个锚框，但要保证正负样本均衡（gluoncv 使用 ProposalTargetCreator 实现）。

## 12.5 本章小结

本章主要介绍了一些 Faster RCNN 的关键组件和思想，并借助 gluoncv 的源码来分析其设计的思路。

# 第 13 章 自制一个集数据与 API 于一身的项目

前面的章节我们已经知晓了如何使用 GitHub 创建属于自己的项目，也学会了如何创建属于自己的 API。本章将带您创建一个集数据与 API 于一身的项目，将前面所学习的知识做一个大一统。

## 13.1 构建项目的骨架

需要分别创建数据存储的仓库 datasome 与 API 的存储仓库 loader。下面详细介绍二者建立的细节。

### 13.1.1 datasome 的创建细节

由于 datasome 存储的数据，必存在很大文件，故此我们使用 Git LFS 进行管理。方法很简单，首先，需要先在 Github 上创建一个空的项目，接着，将其 git clone 到本地磁盘，然后进入到 datasome 根目录，运行如下命令安装 LFS：

```
$ git lfs install
```

需要追踪的数据可能是 HDF5, EXCEL, CSV 等，故而需要设置 LFS 的跟踪类型并提交到仓库。

```
$ git lfs track "*.h5"
$ git lfs track "*.xlsx"
$ git lfs track "*.xls"
$ git lfs track "*.csv"
$ git lfs track "*.zip"
$ git lfs track "*.rar"
$ git lfs track "*.json"
$ git lfs track "*.mat"
$ git add .gitattributes
$ git commit -m "添加数据的跟踪类型"
```

```
$ git push origin master
```

### 13.1.2 loader 的创建细节

本章重构前面章节的 mnist 与 cifar 代码放入 loader 之中。首先在项目根目录创建目录 custom 以及空文件 custom/\_\_init\_\_.py，然后创建文件 custom/mnist.py，并写入如下内容：

```
...
作者: xinetzone
时间: 2019/12/3
...
import struct
from pathlib import Path
import numpy as np
import gzip
class MNIST:
 def __init__(self, root, namespace):
 """MNIST 与 FASGION-MNIST 数据解码工具
 1. (MNIST handwritten digits dataset from http://yann.lecun.com/exdb/mnist) 下载后放置在 `mnist` 目录
 2. (A dataset of Zalando's article images consisting of fashion products,
 a drop-in replacement of the original MNIST dataset from
 https://github.com/zalandoresearch/fashion-mnist)
 数据下载放置在 `fashion-mnist` 目录
 Each sample is an image (in 2D NDArray) with shape (28, 28).
 参数
 =====
 root : 数据根目录, 如 'E:/Data/Zip/'
 namespace : 'mnist' or 'fashion-mnist'
 实例属性
 =====
 train_data: 训练数据集图片
 train_label: 训练数据集标签名称
 test_data: 测试数据集图片
 test_label: 测试数据集标签名称
 ...
 root = Path(root) / namespace
 self._name2array(root)
def _name2array(self, root):
...
官方网站是以 `[offset][type][value][description]` 的格式封装数据的,
因而我们使用 `struct.unpack`
```

```

...
_train_data = root / 'train-images-idx3-ubyte.gz' # 训练数据集文件名
_train_label = root / 'train-labels-idx1-ubyte.gz' # 训练数据集的标签文件名
_test_data = root / 't10k-images-idx3-ubyte.gz' # 测试数据集文件名
_test_label = root / 't10k-labels-idx1-ubyte.gz' # 测试数据集的标签文件名
self.train_data = self.get_data(_train_data) # 获得训练数据集图片
self.train_label = self.get_label(_train_label) # 获得训练数据集标签名称
self.test_data = self.get_data(_test_data) # 获得测试数据集图片
self.test_label = self.get_label(_test_label) # 获得测试数据集标签名称

def get_data(self, data):
 """获取图像信息"""
 with gzip.open(data, 'rb') as fin:
 shape = struct.unpack(">IIII", fin.read(16))[1:]
 data = np.frombuffer(fin.read(), dtype=np.uint8)
 data = data.reshape(shape)
 return data

def get_label(self, label):
 """获取标签信息"""
 with gzip.open(label, 'rb') as fin:
 struct.unpack(">ll", fin.read(8)) # 参考数据集的网站, 即 offset=8
 # 获得数据集的标签
 label = fin.read()
 label = np.frombuffer(label, dtype=np.uint8).astype(np.int32)
return label

```

该模块交代了 MNIST 与 FASHION-MNIST 数据集的解析方法，只需要将这两个数据集下载后分别创建目录 mnist, fashion-mnist，便可以轻松的解析这两个数据集。

同样，我们可以改写 cifar 的数据集处理工具：

```

...
作者: xinetzone
时间: 2019/12/3
...

import tarfile
from pathlib import Path
import pickle
import time
import numpy as np
class Cifar:

 def __init__(self, root, namespace):
 """CIFAR image classification dataset from https://www.cs.toronto.edu/~kriz/cifar.html
 Each sample is an image (in 3D NDArray) with shape (3, 32, 32).
 参数

```

```
=====
meta : 保存了类别信息
root : str, 数据根目录
namespace : 'cifar-10' 或 'cifar-100'
"""

#super().__init__(*args, **kwds)
#self.__dict__ = self
self.root = Path(root)
解压数据集到 root, 并将解析后的数据载入内存
self._load(namespace)

def _extractall(self, namespace):
 """解压 tar 文件并返回路径
 参数
 =====
 tar_name: tar 文件名称
 """

 tar_name = self.root / f'{namespace}-python.tar.gz'
 with tarfile.open(tar_name) as tar:
 tar.extractall(self.root) # 解压全部文件
 names = tar.getnames() # 获取解压后的文件所在目录
 return names

def _decode(self, path):
 """载入二进制流到内存"""
 with open(path, 'rb') as fp: # 打开文件
 # 载入数据到内存
 data = pickle.load(fp, encoding='bytes')
 return data

def _load_cifar10(self, names):
 """将解析后的 cifar10 数据载入内存"""
 # 获取数据根目录
 R = [self.root /
 name for name in names if (self.root / name).is_dir()][0]
 # 元数据信息
 meta = self._decode(list(R.glob('*.meta'))[0])
 # 训练集信息
 train = [self._decode(path) for path in R.glob('*_batch_*')]
 # 测试集信息
 test = [self._decode(path) for path in R.glob('*test*')][0]
 return meta, train, test

def _load_cifar100(self, names):
 """将解析后的 cifar100 数据载入内存"""
```

```

获取数据根目录
R = [self.root /
 name for name in names if (self.root / name).is_dir()][0]
元数据信息
meta = self._decode(list(R.glob('*meta*'))[0])
训练集信息
train = [self._decode(path) for path in R.glob('*train*')][0]
测试集信息
test = [self._decode(path) for path in R.glob('*test*')][0]
return meta, train, test
def _load(self, namespace):
 # 解压数据集到 root，并返回文件列表
 names = self._extractall(namespace)
 if namespace == 'cifar-10':
 self.meta, train, test = self._load_cifar10(names)
 self.trainX = np.concatenate(
 [x[b'data'] for x in train]).reshape(-1, 3, 32, 32)
 self.trainY = np.concatenate([x[b'labels'] for x in train])
 self.testX = np.array(test[b'data']).reshape(-1, 3, 32, 32)
 self.testY = np.array(test[b'labels'])
 elif namespace == 'cifar-100':
 self.meta, train, test = self._load_cifar100(names)
 self.trainX = np.array(train[b'data']).reshape(-1, 3, 32, 32)
 self.testX = np.array(test[b'data']).reshape(-1, 3, 32, 32)
 self.train_fine_labels = np.array(train[b'fine_labels'])
 self.train_coarse_labels = np.array(train[b'coarse_labels'])
 self.test_fine_labels = np.array(test[b'fine_labels'])
 self.test_coarse_labels = np.array(test[b'coarse_labels'])

```

并将其保持到 custom/cifar.py。接着将这些改动提交并推送到 Github 上。

## 13.2 创建一个数据与 API 的统一体

利用 datasome 与 loader 创建一个数据与 API 的统一体 datasetsome。该仓库把 datasome 与 loader 作为 Git 子模块进行管理，而该仓库本身记录数据的简介与 API 的使用说明。

### 13.2.1 datasetsome 项目初设

首先在 Github 创建一个空项目，取名为 datasetsome，然后添加 LFS 的 PDF 与 docx 文档的管理：

```
$ git lfs install
$ git lfs track "*.pdf"
$ git lfs track "*.docx"
$ git add .gitattributes
$ git commit -m "添加数据的跟踪类型"
$ git push origin master
```

接着，使用命令 `git submodule add <仓库地址> <本地路径>`添加 Git 子模块，其中`<本地路径>`是可选项，默认情况下，子模块会在当前目录下面，将子项目放到一个与仓库同名的目录中。如果指定了本地路径，则会把子项目放在指定的本地路径下。为此，在 `datasetsome` 项目根目录添加子模块：

```
$ git submodule add https://github.com/xinetzone/loader
$ git submodule add https://github.com/DataLoaderX/datasetsome
$ git add .gitmodules
$ git commit -m "添加两个子模块"
```

最后，需要修改自述文档 `README.md`，填入如下内容：

```
![GitHub issues](https://img.shields.io/github/issues/DataLoaderX/datasetsome)](https://github.com/DataLoaderX/datasetsome/issues) [![GitHub forks](https://img.shields.io/github/forks/DataLoaderX/datasetsome)](https://github.com/DataLoaderX/datasetsome/network) [![GitHub stars](https://img.shields.io/github/stars/DataLoaderX/datasetsome)](https://github.com/DataLoaderX/datasetsome/stargazers) [![GitHub license](https://img.shields.io/github/license/DataLoaderX/datasetsome)](https://github.com/DataLoaderX/datasetsome/blob/master/LICENSE)
[![HitCount](http://hits.dwyl.io/DataLoaderX/datasetsome.svg)](http://hits.dwyl.io/DataLoaderX/datasetsome) [![GitHub repo size](https://img.shields.io/github/repo-size/DataLoaderX/datasetsome)](https://github.com/DataLoaderX/datasetsome)
```

本项目主要用于记录与数据处理相关的资源，同时您可以进入 [mybinder:  
`datasetsome`](https://mybinder.org/v2/gh/DataLoaderX/datasetsome/master) 进行线编辑。为了将数据和代码以及文档分离，本项目将一些数据集存放在子模块  
[datasome](https://dataloaderx.github.io/datasome/) 之中，而将代码存放在子模块  
[loader](https://xinetzone.github.io/loader/) 之中。

## 数据集处理

- [COCO 数据集](coco/README.md)
- [CASIA 脱机和在线手写汉字库](casia/README.md)
- [omniglot 数据集](omniglot/README.md)
- [待办事项](TODOS/README.md)

该内容用于介绍项目的基本情况。为了让别人更方便的使用该仓库，还要介绍如何使用该仓库，即继续在 `README.md` 文件写下如下内容：

```
本项目的使用说明
因为本项目包含子模块，所以在克隆时，您可以选择使用命令：
```sh  
$ git clone --recursive https://github.com/DataLoaderX/datasetsome
```

```
```
直接下载项目并更新仓库中的每一个子模块。
如果您不想在克隆仓库时下载子模块，您只需要运行下面的命令即可：
```
$ git clone https://github.com/DataLoaderX/datasetsome
```
待您将项目下载到您的电脑后，输入命令：
```
$ git submodule init
$ git submodule update
```
拉取子模的更新。
```

### 13.2.2 在 datasetsome 中编写 loader 使用案例

先创建一个用于将 MNIST, Fashion MNIST, Cifar 10, Cifar 100 打包为 HDF5 的模块 loader/custom/genX.py, 具体内容如下：

```
...
作者: xinetzone
时间: 2019/12/3
...

import tables as tb
import numpy as np
from custom.cifar import Cifar
from custom.mnist import MNIST
class Bunch(dict):
 def __init__(self, root, *args, **kwargs):
 """将数据 MNIST, Fashion MNIST, Cifar 10, Cifar 100 打包
 为 HDF5

 参数
 =====
 root : 数据的根目录
 """
 super().__init__(*args, **kwargs)
 self.__dict__ = self
 self.mnist = MNIST(root, 'mnist')
 self.fashion_mnist = MNIST(root, 'fashion-mnist')
 self.cifar10 = Cifar(root, 'cifar-10')
 self.cifar100 = Cifar(root, 'cifar-100')
 def _change(self, img):
 """将数据由 (num, channel, h, w) 转换为 (num, h, w, channel)"""

```

```

 return np.transpose(img, (0, 2, 3, 1))

def toHDF5(self, save_path):
 """将数据打包为 HDF5 格式
 参数
 =====
 save_path: 数据保存的路径
 """
 filters = tb.Filters(complevel=7, shuffle=False)
 with tb.open_file(f'{save_path}/X.h5', 'w', filters=filters, title='XinetW's dataset') as h5:
 for name in self:
 h5.create_group('/', name, title=name)
 if name in ['mnist', 'fashion_mnist']:
 h5.create_array(
 h5.root[name], 'trainX', self[name].train_data)
 h5.create_array(
 h5.root[name], 'trainY', self[name].train_label)
 h5.create_array(
 h5.root[name], 'testX', self[name].test_data)
 h5.create_array(
 h5.root[name], 'testY', self[name].test_label)
 elif name == 'cifar10':
 h5.create_array(
 h5.root[name], 'trainX', self._change(self[name].trainX))
 h5.create_array(h5.root[name], 'trainY', self[name].trainY)
 h5.create_array(
 h5.root[name], 'testX', self._change(self[name].testX))
 h5.create_array(h5.root[name], 'testY', self[name].testY)
 h5.create_array(h5.root[name], 'label_names', np.array(
 self[name].meta[b'label_names']))
 elif name == 'cifar100':
 h5.create_array(
 h5.root[name], 'trainX', self._change(self[name].trainX))
 h5.create_array(
 h5.root[name], 'testX', self._change(self[name].testX))
 h5.create_array(
 h5.root[name], 'train_coarse_labels', self[name].train_coarse_labels)
 h5.create_array(
 h5.root[name], 'test_coarse_labels', self[name].test_coarse_labels)
 h5.create_array(
 h5.root[name], 'train_fine_labels', self[name].train_fine_labels)
 h5.create_array(

```

```
 h5.root[name], 'test_fine_labels', self[name].test_fine_labels)
 h5.create_array(h5.root[name], 'coarse_label_names', np.array(
 self[name].meta[b'coarse_label_names']))
 h5.create_array(h5.root[name], 'fine_label_names', np.array(
 self[name].meta[b'fine_label_names']))
```

然后，在 datasetsome 中将新添加的代码提交并推送到 loader 本地仓库：

```
$ cd loader
$ git add .
$ git commit -m "添加数据打包工具"
```

为了方便管理该子模块，为其创建一个分支 datasetsome，并切换到该分支：

```
$ git checkout -b datasetsome
```

接着，将该分支合并到远端的 master 上：

```
$ git switch master
$ git merge datasetsome
$ git push origin master
$ git switch datasetsome
```

这样，新增的内容被同步到了 loader 的远端 master 分支。接着，便可以编写一个使用 loader/custom/genX.py 模块的案例了。为了方便调试以及展示，使用 Jupyter Notebook 编写使用案例。只需写入如下内容便可打包那些数据：

```
import sys
为了让 loader 子模块可以使用，需要添加到系统路径
sys.path.append(r"D:\book\datasetsome\loader")
from custom.genX import Bunch
root = 'D:/datasets'
类的实例化
bunch = Bunch(root)
打包数据
bunch.toHDF5(root)
```

将打包后的数据移动到 datasetsome 的子模块 datasome 之中，并提交修改：

```
$ cd datasome
$ git add .
$ git commit -m "添加数据 X.h5"
```

下面跟 loader 一样操作，将修改推送到远端 master 分支。再次回到 datasetsome 根目录将修改推送到远端。

```
$ git add .
$ git commit -m "添加数据 X.h5"
$ git push origin master
```

### 13.3 编写 datasetsome 的使用案例

下面直接编写 datasetsome 的使用案例，利用子模块 datasome 的 X.h5 数据，写一个使用该数据的案例。

首先在项目 datasetsome 根目录创建目录 notebook 用于编写使用案例，接着创建文件 `X 使用说明.ipynb` 并写入如下内容：

```
%matplotlib inline
from matplotlib import pyplot as plt
import numpy as np
import tables as tb
def show_imgs(imgs, row):
 """
 展示 多张图片
 """
 n = imgs.shape[0] # 图片的个数
 h, w = row, int(n / row) # 图片队列的行数与列数
 figs = plt.subplots(h, w, figsize=(5, 5))
 K = np.arange(n).reshape((h, w))
 for i in range(h):
 for j in range(w):
 img = imgs[K[i, j]] # 取出一张图片并放入指定位置
 figs[i][j].imshow(img) # 显示图片
 figs[i][j].axes.get_xaxis().set_visible(False) # 隐藏坐标轴 x
 figs[i][j].axes.get_yaxis().set_visible(False) # 隐藏坐标轴 y
 plt.show()
```

接着编写 X.h5 的使用：

```
root = './datasome/X.h5'

h5 = tb.open_file(root)
show_imgs(h5.root.cifar100.trainX[99:114], 3)
h5.close()
```

输出如图 13.1 所示的图片。

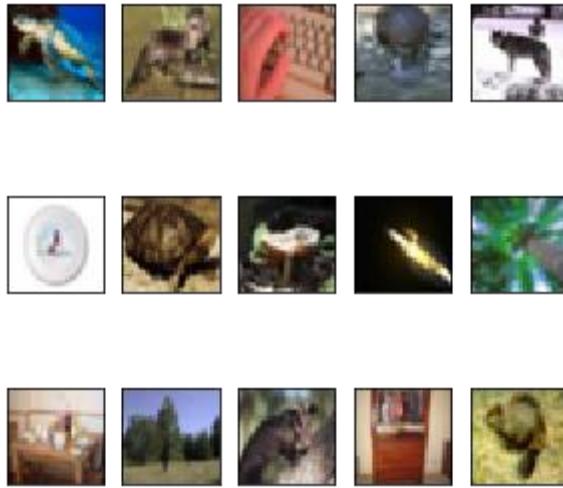


图 13.1 可视化多张 Cifar100 图片

为了更好的使用 X.h5 数据集我们需要编写一个模块:

```
class XDecode:
 def __init__(self, root='..../datasome/X.h5'):
 """解析数据 X
 实例属性
 =====
 members 即 ['mnist', 'fashion_mnist', 'cifar100', 'cifar10']
 ...
 self.h5 = tb.open_file(root)
 self.members = self.h5.root.__members__
 def summary(self):
 """打印 X 数据的摘要信息"""
 print(self.h5)
 def get_members(self, name):
 """获得给定 name 的成员名称列表
 参数
 =====
 name in ['mnist', 'fashion_mnist', 'cifar100', 'cifar10']
 ...
 return self.h5.root[name].__members__
 def get_mnist_members(self):
 return self.h5.root.mnist.__members__
 def get_trainX(self, name):
 """获得给定 name 的 trainX
 参数
 =====
 name in ['mnist', 'fashion_mnist', 'cifar100', 'cifar10']
 ...
```

```
 return self.h5.get_node(f'/{name}', 'trainX')

def get_testX(self, name):
 """获得给定 name 的 testX
 参数
 =====
 name in ['mnist', 'fashion_mnist', 'cifar100', 'cifar10']
 """
 return self.h5.get_node(f'/{name}', 'testX')

def get_trainY(self, name):
 """获得给定 name 的 trainY
 参数
 =====
 name in ['mnist', 'fashion_mnist', 'cifar10']
 """
 return self.h5.get_node(f'/{name}', 'trainY')

def get_testY(self, name):
 """获得给定 name 的 testY
 参数
 =====
 name in ['mnist', 'fashion_mnist', 'cifar10']
 """
 return self.h5.get_node(f'/{name}', 'testY')

def get_train_coarse_labels(self):
 """获得 Cifar100 训练集的粗标签"""
 return self.h5.get_node('/cifar100', 'train_coarse_labels')

def get_train_fine_labels(self):
 """获得 Cifar100 训练集的细标签"""
 return self.h5.get_node('/cifar100', 'train_fine_labels')

def get_test_coarse_labels(self):
 """获得 Cifar100 测试集的粗标签"""
 return self.h5.get_node('/cifar100', 'test_coarse_labels')

def get_test_fine_labels(self):
 """获得 Cifar100 的测试集细标签"""
 return self.h5.get_node('/cifar100', 'test_fine_labels')

def get_coarse_label_names(self):
 """获得 Cifar100 测试集的粗标签的名称"""
 label_names = self.h5.get_node('/cifar100', 'coarse_label_names')
 return np.asarray(label_names, "U")

def get_fine_label_names(self):
 """获得 Cifar100 的测试集细标签的名称"""
 label_names = self.h5.get_node('/cifar100', 'fine_label_names')
 return np.asarray(label_names, "U")
```

```

def get_label_names(self, name):
 """获得给定 name 的标签名称

参数
=====
name in ['mnist', 'fashion_mnist', 'cifar10']
...
if name == 'cifar10':
 label_names = self.h5.get_node('/cifar10', 'label_names')
 return np.asarray(label_names, "U")
elif name == 'fashion_mnist':
 return [
 'T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat',
 'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot'
]
elif name == 'mnist':
 return np.arange(10)

```

下面再编写 Xdecode.py 的使用说明。首先，使用 Jupyter Notebook 载入数据解析器，如图 13.2 所示。

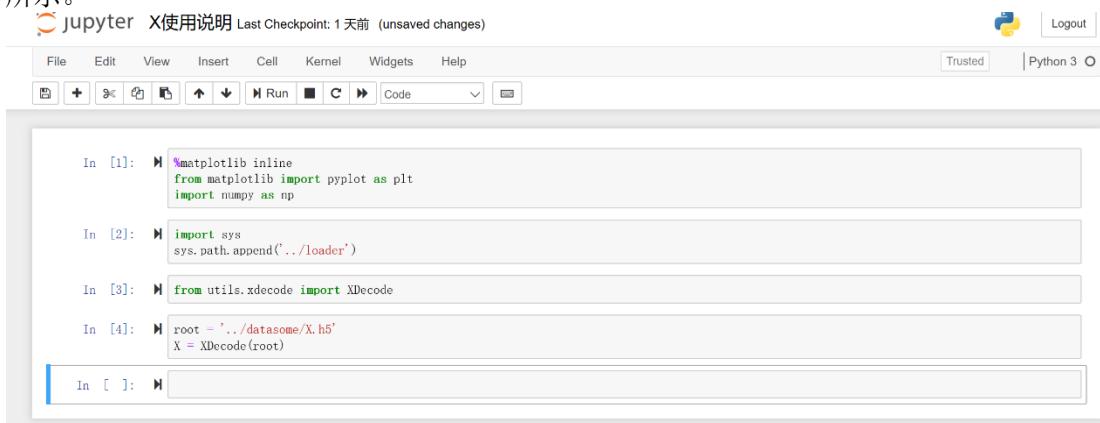


图 13.2 实例化 XDecode

接着，便可以使用 `X.get_label_names(name)` 获取'mnist', 'fashion\_mnist', 'cifar10' 的标签名称，具体见图 13.3。而对于 'cifar10' 分为粗标签与细标签，具体内容见图 13.4。

The screenshot shows a Jupyter Notebook session with several code cells. Cells In [6] and In [7] are shown with their outputs. Cell In [6] contains: `In [6]: X.get_label_names('mnist')` and the output `Out[6]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])`. Cell In [7] contains: `In [7]: X.get_label_names('fashion_mnist')` and the output `Out[7]: ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat', 'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']`. Cell In [8] contains: `In [8]: X.get_label_names('cifar10')` and the output `Out[8]: array(['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck'], dtype='|<U10')`.

图 13.3 获取 'mnist', 'fashion\_mnist', 'cifar10' 类别名称

```
In [9]: X.get_coarse_label_names() # 获取 cifar100 粗标签
Out[9]: array(['aquatic_mammals', 'fish', 'flowers', 'food_containers',
 'fruit_and_vegetables', 'household_electrical_devices',
 'household_furniture', 'insects', 'large_carnivores',
 'large_man-made_outdoor_things', 'large_natural_outdoor_scenes',
 'large_omnivores_and_herbivores', 'medium_mammals',
 'non_insect_invertebrates', 'people', 'reptiles', 'small_mammals',
 'trees', 'vehicles_1', 'vehicles_2'], dtype='<U30')

In [10]: X.get_fine_label_names() # 获取 cifar100 细标签
Out[10]: array(['apple', 'aquarium_fish', 'baby', 'bear', 'beaver', 'bed', 'bee',
 'beetle', 'bicycle', 'bottle', 'bowl', 'boy', 'bridge', 'bus',
 'butterfly', 'camel', 'can', 'castle', 'caterpillar', 'cattle',
 'chair', 'chimpanzee', 'clock', 'cloud', 'cockroach', 'couch',
 'crab', 'crocodile', 'cup', 'dinosaur', 'dolphin', 'elephant',
 'flatfish', 'forest', 'fox', 'girl', 'hamster', 'house',
 'kangaroo', 'keyboard', 'lamp', 'lawn_mower', 'leopard', 'lion',
 'lizard', 'lobster', 'man', 'maple_tree', 'motorcycle', 'mountain',
 'mouse', 'mushroom', 'oak_tree', 'orange', 'orchid', 'otter',
 'palm_tree', 'pear', 'pickup_truck', 'pine_tree', 'plain', 'plate',
 'poppy', 'porcupine', 'possum', 'rabbit', 'raccoon', 'ray', 'road',
 'rocket', 'rose', 'sea', 'seal', 'shark', 'shrew', 'skunk',
 'skyscraper', 'snail', 'snake', 'spider', 'squirrel', 'streetcar',
 'sunflower', 'sweet_pepper', 'table', 'tank', 'telephone',
 'television', 'tiger', 'tractor', 'train', 'trout', 'tulip',
 'turtle', 'wardrobe', 'whale', 'willow_tree', 'wolf', 'woman',
 'worm'], dtype='<U13')
```

图 13.4 获取 Cifar100 的标签列表

获取 'mnist', 'fashion\_mnist', 'cifar10' 的数值标签，可参考图 13.5。

下面介绍数值型数据，因为训练集与测试集是一致的，所以仅仅展示测试集。

### 获取 'mnist', 'fashion\_mnist', 'cifar10' 的数值标签

```
10]: X.get_testY('mnist')[:] # 'mnist' 的数值标签
Out[10]: array([7, 2, 1, ..., 4, 5, 6])

11]: X.get_testY('fashion_mnist')[:] # 'fashion_mnist' 的数值标签
Out[11]: array([9, 2, 1, ..., 8, 1, 5])

12]: X.get_testY('cifar10')[:] # 'cifar10' 的数值标签
Out[12]: array([3, 8, 8, ..., 5, 1, 7])
```

图 13.5 获取 'mnist', 'fashion\_mnist', 'cifar10' 的数值标签

同样可以获取 'cifar100' 的数值标签，具体见图 13.6。

### 获取 'cifar100' 的数值标签

```
: X.get_test_coarse_labels()[:] # 'cifar100' 的数值粗标签
:[13]: array([10, 10, 0, ..., 4, 8, 2])

: X.get_test_fine_labels()[:] # 'cifar100' 的数值细标签
:[14]: array([49, 33, 72, ..., 51, 42, 70])
```

图 13.6 获取 'cifar100' 的数值标签

可以使用切片或者索引的方式获取图片的数组信息，具体见图 13.7。

## 获取数据的图片数组

```
[15]: X.get_testX('mnist')[0].shape # 取一张'mnist'图片并返回 shape
Out[15]: (28, 28)

[16]: X.get_testX('fashion_mnist')
Out[16]: /fashion_mnist/testX (Array(10000, 28, 28))
atom := UInt8Atom(shape=(), dflt=0)
maindim := 0
flavor := 'numpy'
byteorder := 'irrelevant'
chunkshape := None

[17]: X.get_testX('cifar10')[0].shape # 取一张'cifar10'图片并返回 shape
Out[17]: (32, 32, 3)
```

图 13.7 用切片或者索引的方式获取图片的数组信息

为了综合标签名称与图片的信息，可以定义一个函数：

```
def show_imgs(imgs, labels, row_num):
 """展示 多张图片，并显示图片标签
 ...
 n = imgs.shape[0]
 assert n % row_num == 0, "请输入可以整除图片数量的行数"
 h, w = row_num, int(n / row_num)
 fig, ax = plt.subplots(h, w, figsize=(7, 7))
 K = np.arange(n).reshape((h, w))
 names = labels.reshape((h, w))
 for i in range(h):
 for j in range(w):
 img = imgs[K[i, j]]
 ax[i][j].imshow(img)
 ax[i][j].axes.get_yaxis().set_visible(False)
 ax[i][j].axes.set_xlabel(names[i][j])
 ax[i][j].set_xticks([])
 plt.show()
```

具体效果，见图 13.8。

最后，将上述的改变提交的本地仓库并上传到 Github 远端。

```
In [19]: name = 'cifar100'
select_image_id = slice(100, 112)
imgs = X.get_testX(name)[select_image_id]
label_names = X.get_fine_label_names()
labels = X.get_test_fine_labels()[select_image_id] # 获得标签的数值代表
labels = np.array([label_names[ind] for ind in labels])
row_num = 4
show_imgs(imgs, labels, row_num)
```



图 13.8 可视化多张图片

## 13.4 本章小结

本章主要介绍了如何将数据处理与代码设计集成于一体并上传到 GitHub 包装为一个项目。利用 Git LFS 管理大文件，Git 子模块处理多个子项目。

# 第 14 章 TensorFlow 基础教程

前面的章节已经介绍了 MXNet 的基础教程，本章主要讨论如何在数据集 X 上学习 TensorFlow2 的入门教程。本章大部分内容参考 TensorFlow 的官方教程。本书写作本章的初衷是由于 MXNet 与 Pytorch 的基础教程在《动手学习深度学习》中的教程已然很丰富，但是关于 TensorFlow2 的中文教程并不是那么的系统，所以本章借此简单的介绍 TensorFlow2 基础教程。

## 14.1 使用 tf.keras.Sequential 搭建模型

在 TensorFlow 中，借助子模块 tf.keras 的 Sequential 类很容易搭建一个神经网络模型：

```
%matplotlib inline
```

```
from matplotlib import pyplot as plt
import numpy as np

import sys
sys.path.append('../loader')
from utils.xdecode import XDecode

X = XDecode('../datasome/X.h5')
载入并准备好 MNIST 数据集。将样本从整数转换为浮点数：
name = 'mnist'

x_train, y_train = X.get_trainX(name)[:], X.get_trainY(name)[:]
x_test, y_test = X.get_testX(name)[:], X.get_testY(name)[:]
x_train, x_test = x_train / 255.0, x_test / 255.0
```

上面的例子我们将数据集 `X` 当作一个基础数据集，获取了数据集 MNIST 的训练集和测试集，并且将数据集的图片像素数组转换为浮点型，且对其作了数据归一化处理。为了搭建神经网络模型，需要借助 `tf.keras.Sequential` 类搭建网络“积木”将模型的各层堆叠起来，以搭建模型。

```
import tensorflow as tf
创建网络模型“积木”
model = tf.keras.models.Sequential([
 tf.keras.layers.Flatten(input_shape=(28, 28)),
 tf.keras.layers.Dense(128, activation='relu'),
 tf.keras.layers.Dropout(0.2),
 tf.keras.layers.Dense(10, activation='softmax')
])
```

为模型选择优化器和损失函数：

```
model.compile(optimizer='adam',
 loss='sparse_categorical_crossentropy', metrics=['accuracy'])
打印 tf 的版本信息
print(tf.__version__)
```

输出版本号为：

```
2.0.0
```

这样，便完成了模型的准备工作：数据准备、模型构建、定义损失函数、定义优化器。下面使用 `fit` 函数训练模型：

```
model.fit(x_train, y_train, epochs=5)
```

输出结果：

```
Train on 60000 samples
Epoch 1/5
60000/60000 [=====] - 3s 43us/sample - loss: 0.2996 - accuracy: 0.9129
Epoch 2/5
60000/60000 [=====] - 2s 38us/sample - loss: 0.1437 - accuracy: 0.9575
Epoch 3/5
60000/60000 [=====] - 2s 35us/sample - loss: 0.1085 - accuracy: 0.9674
```

```
Epoch 4/5
60000/60000 [=====] - 2s 35us/sample - loss: 0.0883 - accuracy: 0.9730
Epoch 5/5
60000/60000 [=====] - 2s 35us/sample - loss: 0.0760 - accuracy: 0.9761
```

该结果显示了模型的训练过程的损失函数以及准确度。最后，使用模型的 evaluate 函数获取测试集的准确度：

```
model.evaluate(x_test, y_test, verbose=2)
```

输出结果为：

```
10000/1 - 0s - loss: 0.0410 - accuracy: 0.9770
[0.07469123949403875, 0.977]
```

可以看出训练的泛化性在测试集上表现不错。

## 14.2 FASHION-MNIST 可视化实验结果

首先载入一些必备的包：

```
TensorFlow and tf.keras
import tensorflow as tf
from tensorflow import keras

Helper libraries
import numpy as np
from matplotlib import pyplot as plt
让 Jupyter Notebook 的画图可见
%matplotlib inline
```

载入自定义包：

```
import sys
sys.path.append('../loader')
from utils.xdecode import XDecode
```

下面定义一个可以处理 MNIST 与 FASHIO-MNIST 的通用类：

```
class Loader:
 def __init__(self, root, name):
 """针对 MNIST 与 FASHION-MNIST 的一个通用处理工具"""
 self.X = XDecode(root) # 载入数据集 X
 self.get_data(name) # 通过给的 name 获取训练集与测试集
 # 获取数据的标签名称
 self.class_names = self.X.get_label_names(name)
 def get_data(self, name):
 """通过给的 name 获取训练集与测试集"""
 x_train = self.X.get_trainX(name)[:]
 x_test = self.X.get_testX(name)[:]
 # 数据归一化
 self.x_train, self.x_test = x_train / 255.0, x_test / 255.0
 self.y_train = self.X.get_trainY(name)[:]
```

```

 self.y_test = self.X.get_testY(name)[:]
def get_model(self):
 """定义模型"""
 model = keras.Sequential([
 keras.layers.Flatten(input_shape=(28, 28)),
 keras.layers.Dense(128, activation='relu'),
 keras.layers.Dense(10, activation='softmax')
])
 return model
def fit(self, epochs=10):
 """训练模型"""
 model = self.get_model()
 model.compile(optimizer='adam',
 loss='sparse_categorical_crossentropy',
 metrics=['accuracy'])
 model.fit(self.x_train, self.y_train, epochs=epochs)
 return model
def predict(self, model):
 """通过训练好的模型预测测试集的标签"""
 predicted_array = model.predict(self.x_test)
 return predicted_array
def plot_image(self, img, predicted_array, true_label):
 plt.grid(False)
 plt.xticks([])
 plt.yticks([])
 # 由于图片是灰度图，所以采用 plt.cm.binary
 plt.imshow(img, cmap=plt.cm.binary)
 # 获取置信度最大的标签
 predicted_label = np.argmax(predicted_array)
 if predicted_label == true_label:
 color = 'blue'
 else:
 color = 'red'
 # 显示图片的标签信息
 plt.xlabel("{} {:.2f}% ({})".format(self.class_names[predicted_label],
 100*np.max(predicted_array),
 self.class_names[true_label]),
 color=color)
def plot_value_array(self, predictions_array, true_label):
 plt.grid(False) # 去除图片的网格
 plt.xticks(range(10))
 plt.yticks([])
 thisplot = plt.bar(range(10), predictions_array, color="#777777")
 plt.ylim([0, 1])
 predicted_label = np.argmax(predictions_array)
 # 设置条状的颜色
 thisplot[predicted_label].set_color('red')
 thisplot[true_label].set_color('blue')

```

这样直接调用该类的实例即可完成模型的训练与评估以及可视化：

```
root = './datasome/X.h5'
name = 'fashion_mnist'
loader = Loader(root, name) # 实例化模型的数据载入
定义模型参数
epochs = 10 # 定义训练的迭代次数
model = loader.fit(epochs) # 训练模型
predictions = loader.predict(model) # 预测模型
```

输出结果：

```
Train on 60000 samples
Epoch 1/10
60000/60000 [=====] - 8s 141us/sample - loss: 0.4989 - accuracy: 0.8243
Epoch 2/10
60000/60000 [=====] - 7s 123us/sample - loss: 0.3722 - accuracy: 0.8665
Epoch 3/10
60000/60000 [=====] - 7s 123us/sample - loss: 0.3366 - accuracy: 0.8761
Epoch 4/10
60000/60000 [=====] - 8s 128us/sample - loss: 0.3145 - accuracy: 0.8852
Epoch 5/10
60000/60000 [=====] - 8s 127us/sample - loss: 0.2979 - accuracy: 0.8891
Epoch 6/10
60000/60000 [=====] - 7s 122us/sample - loss: 0.2822 - accuracy: 0.8953
Epoch 7/10
60000/60000 [=====] - 8s 125us/sample - loss: 0.2708 - accuracy: 0.8996
Epoch 8/10
60000/60000 [=====] - 7s 123us/sample - loss: 0.2587 - accuracy: 0.9033
Epoch 9/10
60000/60000 [=====] - 7s 125us/sample - loss: 0.2483 - accuracy: 0.9064
Epoch 10/10
60000/60000 [=====] - 8s 131us/sample - loss: 0.2405 - accuracy: 0.9106
```

该结果显示了训练的速度以及损失函数和准确度。最后，需要将结果可视化出来：

```
Plot the first X test images, their predicted labels, and the true labels.
Color correct predictions in blue and incorrect predictions in red.

num_rows = 5
num_cols = 3
num_images = num_rows*num_cols
start_id = 100 # 图片的起始 ID
plt.figure(figsize=(2*2*num_cols, 2*num_rows))
for k in range(num_images):
 plt.subplot(num_rows, 2*num_cols, 2*k+1)
```

```

img = loader.x_test[start_id + k]
true_label = loader.y_test[start_id + k]
predicted_array = predictions[start_id + k]
loader.plot_image(img, predicted_array, true_label)
loader.plot_image(img, predicted_array, true_label)
plt.subplot(num_rows, 2*num_cols, 2*k+2)
loader.plot_value_array(predicted_array, true_label)
plt.tight_layout()
plt.show()

```

输出图片，如图 14.1 所示。

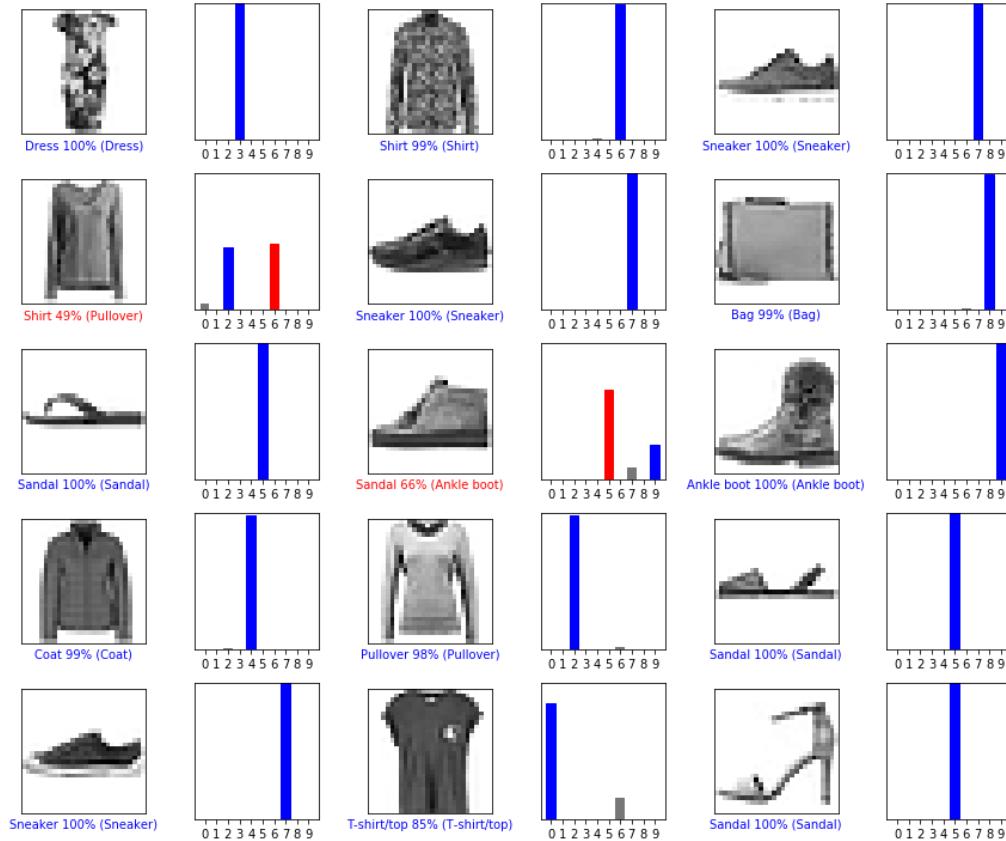


图 14.1 测试结果的可视化

### 14.3 PyTorch、TensorFlow、MXNet 的数据操作

PyTorch, TensorFlow, MXNet 操作数据的方法很相似。它们为了支持 GPU 等设备对数据进行运算均定义了“张量”的实体。PyTorch 使用 `torch.Tensor`, TensorFlow2 使用 `tensorflow.Tensor`, MXNet 使用 `mxnet.nd`。先载入相关的库

```

from mxnet import nd
import torch
import tensorflow as tf

```

这里 MXNet 的 `nd` 与 NumPy 最为接近。可以使用 `nd.arange(12)`, `nd.zeros((2, 3, 4))`, `nd.array([[2, 1, 4, 3], [1, 2, 3, 4], [4, 3, 2, 1]])` 这些形式来创建张量。对应于 PyTorch 的 `torch.arange(12)`, `torch.zeros((2, 3, 4))`, `torch.tensor([[2, 1, 4, 3], [1, 2, 3, 4], [4, 3, 2, 1]])`。而 TensorFlow 对应于 `tf.range(12)`, `tf.zeros((2, 3, 4))`, `tf.constant ([[2, 1, 4, 3], [1, 2, 3, 4], [4, 3, 2, 1]])`。

关于它们的张量的加法、减法、乘法、除法、矩阵运算等直接类比 NumPy 即可。

## 14.4 PyTorch, TensorFlow, MXNet 自动微分

对于微分运算, TensorFlow, MXNet, Pytorch 各自有不同的定义。其中 MXNet, Pytorch 极为相近。

我们讨论一个向量内积的微分, 即  $x \in \mathbb{R}^{n \times 1}$ , 计算导数:  $\frac{d}{dx}(x^T x) = 2x$ 。

下面以  $x = (0, 1, 2, 3, 4)$  为例使用不同的深度学习框架来计算其内积的导数。

先使用 NumPy 创建一个列向量:

```
import numpy as np
x = np.arange(5.).reshape(-1,1) # 设置数据类型为浮点型
```

先看看如何使用 Pytorch(1.2.0 版本)编写求微分的运算? 为了可以利用链式法则进行梯度传播需要将张量属性`requires_grad` 设置为 `True`, 以此开始追踪(track)在其上的所有操作:

```
import torch
x = torch.tensor(x, requires_grad=True)
```

下面定义内积运算:

```
y = torch.matmul(x.T, x) # 矩阵乘法
```

在 Pytorch 中 `grad` 在反向传播过程中是累加的(accumulated), 这意味着每一次运行反向传播, 梯度都会累加之前的梯度, 所以一般在反向传播之前需把梯度清零。不过, 这里仅仅只运行一次反向传播, 不需要清零:

```
x.grad.data.zero_() # 梯度清零
y.backward() # 反向传播
```

最后, 使用 `x.grad` 获取 `y` 关于 `x` 的梯度。

关于 MXNet 的求梯度的运算在前面的章节亦有所阐述, 下面仅仅提供一个代码实例:

```
x = nd.array(x)
x.attach_grad() # 添加追踪
with autograd.record(): # 记录需要计算梯度的变量
 y = 2 * nd.dot(x.T, x)
 y.backward() # 反向传播
x.grad # 获取梯度
```

最后, 再来看看 TensorFlow 如何求解微分?

```
x = np.arange(5.).reshape(-1,1) # 设置数据类型为浮点型
x = tf.constant(x)
with tf.GradientTape() as t: # 记录需要计算梯度的变量
 t.watch(x) # 追踪 x 的梯度
 y = tf.matmul(tf.transpose(x),x)
 # 计算 y 对 x 的梯度
dy_dx = t.gradient(y, x)
```

`tf.constant` 指代 TensorFlow 中的“常量”，而 `tf.Variable` 是 TensorFlow 中的“变量”，且不需要显式追踪 `x` 的梯度，即下面的代码也可达到上个代码的效果：

```
x = np.arange(5.).reshape(-1,1) # 设置数据类型为浮点型
x = tf.Variable(x)
with tf.GradientTape() as t: # 记录需要计算梯度的变量
 #t.watch(x) # 追踪 x 的梯度
 y = tf.matmul(tf.transpose(x),x)
计算 y 对 x 的梯度
dy_dx = t.gradient(y, x)
```

## 14.5 TensorFlow 实现手写汉字分类的实例

第 6 章我们已经讨论过 CASIA 手写汉字这一个数据集，为了让数据更加友好，需要进一步的简化。本教程讨论 HWDB1.0~1.1 以及 OLHWDB1.0~1.1，下载到个人电脑的同一目录下：

```
CASI 数据集所在根目录
root = 'D:/datasets/OCR/CASIA/data'
```

载入本教程需要使用的包：

```
import struct
from pathlib import Path
from zipfile import ZipFile
import numpy as np
import tables as tb
```

Path 更加友好的管理文件的路径：

```
root = Path(root)
查看 root 的全部文件
[name.parts[-1] for name in root.iterdir()]
```

下面显示了本章所使用的数据：

```
['HWDB1.0trn.zip',
'HWDB1.0trn_gnt.zip',
'HWDB1.0tst.zip',
'HWDB1.0tst_gnt.zip',
'HWDB1.1trn.zip',
'HWDB1.1trn_gnt.zip',
'HWDB1.1tst.zip',
'HWDB1.1tst_gnt.zip',
'OLHWDB1.0test_pot.zip',
'OLHWDB1.0train_pot.zip',
'OLHWDB1.0trn.zip',
'OLHWDB1.0tst.zip',
'OLHWDB1.1trn.zip',
'OLHWDB1.1trn_pot.zip',
'OLHWDB1.1tst.zip',
'OLHWDB1.1tst_pot.zip']
```

每个单字的特征均以`.mpf` 形式保存手工特征，可以看出上述文件均为压缩包，下面使用 `zipfile` 包去压缩文件进行解读：

```
z = ZipFile(list(root.glob('**/HWDB1.0trn.zip'))[0])
z.namelist()[1:5] # 查看前 4 个人写的 MPF
```

显示结果如下：

```
['HWDB1.0trn/001.mpf',
'HWDB1.0trn/002.mpf',
'HWDB1.0trn/003.mpf',
'HWDB1.0trn/004.mpf']
```

载入 MPF 的解码器：MPFDecoder，具体见我的 GitHub，即在 GitHub 搜索关键字：xinetzone/loader，下面载入需要的包：

```
import sys
sys.path.append("../loader") # loader 仓库路径
from casia.feature import MPFDecoder, zipfile2bunch
```

#### 14.5.1 loader 的制作原理

先简要了解 loader 的制作原理。

##### 1. 将 MPF 转换为 bunch。

```
zip_name = list(root.glob('**/HWDB1.0trn.zip'))[0]
mb = zipfile2bunch(zip_name)
```

##### 2. 将 bunch 转换为 JSON。

先载入一些工具：

```
from utils.dataset import bunch2json, json2bunch
```

测试 bunch 转换为 JSON 的时间：

```
%%time
json_path = 'data/features.json'
bunch2json(mb, json_path)
```

输出结果：

```
Wall time: 1.04 s
```

再载入看 JSON 花费的时间：

```
%%time
再次载入数据
mpf_bunch = json2bunch(json_path)
```

显示输出：

```
Wall time: 812 ms
```

可以看出 JSON 的载入时间很短。

##### 3. 将 bunch 转换为 HDF5

下面载入 bunch 转换为 HDF5 需要的包：

```
from casia.feature import bunch2hdf
```

测试 bunch 转换为 HDF5 的时间：

```
%%time
hdf_path = 'data/features.h5'
bunch2hdf(mpf_bunch, hdf_path)
```

显示输出：

```
Wall time: 2.4 s
```

再看看载入 HDF5 花费的时间：

```

%%time
h5 = tb.open_file(hdf_path)
显示输出:
Wall time: 1 ms
可以看出此时花费的时间更短, 可以以下面的方式获取 MPF 的特征矩阵及其 shape:
获取某个 mpf 的特征矩阵的 shape
h5.root.HWDB1_0trn__001_mpf.features[:,].shape
输出结果为 (样本数, 特征向量维数) :
(3728, 512)
也可以获取 MPF 的特征的简略介绍:
获取某个 mpf 的特征介绍
h5.root.HWDB1_0trn__001_mpf.text.read()
显示结果如下:
b'Character features extracted from grayscale images. #ftrtype=ncg, #norm=ldi, #aspect=4, #dirn=8,
#zone=8, #zstep=8, #fstep=8, $deslant=0, $smooth=0, $nmdir=0, $multisc=0'
仅仅知道 MPF 特征并不够, 我们还需要知道其对应的特征标签:
获取某个 mpf 的标签信息
labels = h5.root.HWDB1_0trn__001_mpf.labels.read().decode()
labels = np.array(labels.split(' '))
labels
显示结果为:
array(['扼', '遏', '鄂', ..., '娥', '恶', '厄'], dtype='<U1')

```

#### 4 测试 JSON 与 HDF5 的文件大小

为了比较 JSON 与 HDF5 的性能, 需要测试它们的占用大小:

```

import os
from sys import getsizeof
print(
 f"JSON Python 对象占用空间大小为 {getsizeof(mpf_bunch)/1e3} kB, 文件大小为
{os.path.getsize(json_path)/1e9} G")
print(
 f"HDF5 Python 对象占用空间大小为 {getsizeof(h5)} B, 文件大小为 {os.path.getsize(hdf_path)/1e9}
G")
h5.close() # 关闭

```

输出结果为:

```

JSON Python 对象占用空间大小为 9.32 kB, 文件大小为 0.65487944 G
HDF5 Python 对象占用空间大小为 80 B, 文件大小为 0.644292324 G

```

可以看出 JSON 与 HDF5 载入到内存的大小并不是很大, 但是 JSON 又比 HDF5 大 100 多倍, 所以 HDF5 的性能优于 JSON。

#### 14.5.2 打包多个 zip 文件

为了整合多个压缩文件, 接下来需要将其打包。首先需要将压缩文件进行划分:

```

zip_gnt_names = set(root.glob('*gnt*.zip')) # GNT 名称列表
zip_pot_names = set(root.glob('*pot*.zip')) # POT 名称列表
MPF 名称列表

```

```
zip_mpf_names = set(root.iterdir()) - zip_pot_names - zip_gnt_names
zip_mpf_names
```

显示结果：

```
{WindowsPath('D:/datasets/OCR/CASIA/data/HWDB1.0trn.zip'),
 WindowsPath('D:/datasets/OCR/CASIA/data/HWDB1.0tst.zip'),
 WindowsPath('D:/datasets/OCR/CASIA/data/HWDB1.1trn.zip'),
 WindowsPath('D:/datasets/OCR/CASIA/data/HWDB1.1tst.zip'),
 WindowsPath('D:/datasets/OCR/CASIA/data/OLHWDB1.0trn.zip'),
 WindowsPath('D:/datasets/OCR/CASIA/data/OLHWDB1.0tst.zip'),
 WindowsPath('D:/datasets/OCR/CASIA/data/OLHWDB1.1trn.zip'),
 WindowsPath('D:/datasets/OCR/CASIA/data/OLHWDB1.1tst.zip')}
```

下面的代码将压缩文件划分为 POT, GNT, MPF, 下面先研究 MPF:

```
mpf_bunch = {} # 打包全部 MPF 为 bunch
for mpf_name in zip_mpf_names:
 mpf_bunch.update(zipfile2bunch(mpf_name))
```

保存为 JSON 并测试花费时间：

```
%%time
json_path = 'data/features.json'
bunch2json(mpf_bunch, json_path)
```

显示结果为：

```
Wall time: 5.78 s
```

保存为 HDF5 并测试花费时间：

```
%%time
hdf_path = 'data/features.h5'
bunch2hdf(mpf_bunch, hdf_path)
```

显示结果为：

```
Wall time: 11 s
```

载入 JSON 并测试花费时间：

```
%%time
mpf_bunch = json2bunch(json_path)
```

显示结果为：

```
Wall time: 7.96 s
```

载入 HDF5 并测试花费时间：

```
%%time
h5 = tb.open_file(hdf_path)
```

显示结果为：

```
Wall time: 9 ms
```

再次测试文件大小：

```
from sys import getsizeof
print()
 f"JSON Python 对象占用空间大小为 {getsizeof(mpf_bunch)/1e3} kB, 文件大小为
{Path(json_path).stat().st_size/1e9} G"
print()
 f"HDF5 Python 对象占用空间大小为 {getsizeof(h5)} B, 文件大小为
{Path(hdf_path).stat().st_size/1e9} G"
```

显示结果为：

```
JSON Python 对象占用空间大小为 73.824 kB, 文件大小为 2.82091995 G
```

HDF5 Python 对象占用空间大小为 80 B, 文件大小为 2.775283309 G

从上述的展示可以看出 HDF5 优于 JSON 与 Zip 压缩文件, 所以下面仅仅考虑 HDF5 文件。

### 14.5.3 解析 features.h5

如图 14.2 所示, 展示了 features.h5 的几个使用技巧。

```
In [43]: h5.get_filesize() # 获取文件大小
Out[43]: 2775283309

In [52]: nodes = h5.list_nodes('/')
Out[52]: '/HWDB1_Otrn_001_mpf (Group)'
 children := ['features' (Array), 'labels' (Array), 'text' (Array)]

In [54]: nodes[0]
Out[54]: /HWDB1_Otrn_001_mpf (Group)
 children := ['features' (Array), 'labels' (Array), 'text' (Array)]

In [55]: len(nodes) # 统计 MPF 个数
Out[55]: 1440

In [57]: data_iter = h5.iter_nodes('/') # 所有 MPF 数据以迭代器的方式使用

In [60]: next(data_iter) # 取出一个 MPF
Out[60]: /HWDB1_Otrn_001_mpf (Group)
 children := ['features' (Array), 'labels' (Array), 'text' (Array)]
```

图 14.2 查看 features.h5 的几个简单特性

考虑到通用性, 图 14.3 定义了两个函数分别用于获取特征矩阵与标签。下面依据图 14.3 定义的两个函数创建一个 MPF 迭代器:

```
class CASIAFeature:
 def __init__(self, hdf_path):
 """casia 数据 MPF 特征处理工具"""
 self.h5 = tb.open_file(hdf_path)
 def _features(self, mpf):
 """获取 MPF 的特征矩阵"""
 return mpf.features[:]
 def _labels(self, mpf):
 """获取 MPF 的标签数组"""
 labels_str = mpf.labels.read().decode()
 return np.array(labels_str.split(' '))
 def __iter__(self):
 """返回 (features, labels)"""
 for mpf in self.h5.iter_nodes('/'):
 yield self._features(mpf), self._labels(mpf)
```

CASIAFeature 的使用方法如下:

```
mpf_iter = CASIAFeature(hdf_path)
以迭代器的方式获取数据
for features, labels in mpf_iter:
 print(features.shape, labels.shape)
break
```

```

In [48]: # mpf_name = 'HWDB1_0trn_007_mpf'
依据 MPF 的名称获取 MPF
mpf = h5.get_node('/', mpf_name)
mpf

Out[48]: /HWDB1_0trn_007_mpf (Group)
 children := ['features' (Array), 'labels' (Array), 'text' (Array)]

In [49]: def get_features(mpf):
 ''' 获取 MPF 的特征矩阵'''
 return mpf.features[:]

def get_labels(mpf):
 ''' 获取 MPF 的标签数组'''
 labels_str = mpf.labels.read().decode()
 return np.array(labels_str.split(' '))

In [50]: features = get_features(mpf) # 获取特征矩阵
labels = get_labels(mpf) # 获取标签

```

图 14.3 定义了两个函数分别用于获取特征矩阵与标签

#### 14.5.4 使用 TensorFlow 训练 MPF 分类器

一般需要将数据 划分为训练集和测试集，在 xinetzone/loader 模块中已经实现了数据的划分：

```

class CASIA:
 def __init__(self, root):
 self.root = Path(root)
 def names2bunch(self, names):
 """合并给定的 names 的 bunch"""
 bunch = {}
 for mpf_name in names:
 bunch.update(zipfile2bunch(mpf_name))
 return bunch
 def split(self, root):
 """划分 casia 的 bunch 为训练集与测试集"""
 train_names = set(root.glob('*trn.zip')) # 训练集名称列表
 test_names = set(root.glob('*tst.zip')) # 测试集名称列表
 # names 转换为 bunch
 train_bunch = self.names2bunch(train_names)
 test_bunch = self.names2bunch(test_names)
 bunch = {'train': train_bunch, 'test': test_bunch}
 return bunch
 def bunch2hdf(self, save_path):
 """将 bunch 转换为 HDF5"""
 bunch = self.split(self.root)
 # 过滤信息，用于压缩文件
 filters = tb.Filters(complevel=7, shuffle=False)
 with tb.open_file(save_path, 'w', filters=filters, title='Xinet\'s casia dataset') as h:
 for group_name, features in bunch.items():
 h.create_group('/', group_name)
 for name in features: # 生成数据集"头"
 name = name.replace('/', '')

```

```

 _name = _name.replace('.', '_')
 h.create_group(f'{group_name}',
 name=_name, filters=filters)
 h.create_array(f'{group_name}/{name}/text',
 bunch[group_name][name]['text'].encode())
 features = bunch[group_name][name]['dataset']
 h.create_array(f'{group_name}/{name}/labels',
 " ".join([l for l in features.index]).encode())
 h.create_array(f'{group_name}/{name}/",
 'features', features.values)

```

类 CASIA 实现了数据集的分开打包，下面还需重新 CASIA 的 HDF5 文件的解析类：

```

class CASIAFeature:
 def __init__(self, hdf_path):
 """casia 数据 MPF 特征处理工具"""
 self.h5 = tb.open_file(hdf_path)
 def _features(self, mpf):
 """获取 MPF 的特征矩阵"""
 return mpf.features[:]
 def _labels(self, mpf):
 """获取 MPF 的标签数组"""
 labels_str = mpf.labels.read().decode()
 return np.array(labels_str.split(' '))
 def train_iter(self):
 """返回 训练集的 (features, labels)"""
 for mpf in self.h5.iter_nodes('/train'):
 yield self._features(mpf), self._labels(mpf)
 def test_iter(self):
 """返回 测试集的 (features, labels)"""
 for mpf in self.h5.iter_nodes('/test'):
 yield self._features(mpf), self._labels(mpf)

```

这样便可以像下面的方式获取 CASIA 的 MPF 数据：

```

hdf_path = 'data/features.h5' # HDF5 文件所在路径
mpf_dataset = CASIAFeature(hdf_path)
以迭代器的方式获取数据
for features, labels in mpf_dataset.test_iter():
 # 打印特征向量与标签的 shape
 print(features.shape, labels.shape)
 break

```

显示结果为：

```
(3726, 512) (3726,
```

可以看出此 MPF 保存 3726 个样本，且特征向量的长度是 512。可以直接将全部的 MPF 数据转换为 NumPy 的 array 格式：

```

hdf_path = 'data/features.h5'
mpf_dataset = CASIAFeature(hdf_path)
划分数据集为训练集与测试集
train_features = np.concatenate([features for features, _ in mpf_dataset.train_iter()])
train_labels = np.concatenate([labels for _, labels in mpf_dataset.train_iter()])
test_features = np.concatenate([features for features, _ in mpf_dataset.test_iter()])

```

```
test_labels = np.concatenate([labels for _, labels in mpf_dataset.test_iter()])
```

统计训练集与测试集的标签名称并判断它们是否相同：

```
获取训练集的类别名称集
```

```
train_class_names = set(train_labels)
```

```
获取训练测试集的类别名称集
```

```
test_class_names = set(test_labels)
```

```
is_same_class = "相同" if train_class_names == test_class_names else "不相同"
```

```
print(f'训练集与测试集的类别标签是{is_same_class}的')
```

输出结果是：

```
训练集与测试集的类别标签是相同的
```

由于训练集与测试集的类别标签是相同的，所以下面可以将类别名称写作：

```
class_names = test_class_names
```

这样，便可以获取类别的个数：

```
获取类别个数
```

```
CLASS_NUM = len(class_names)
```

```
print('类别个数：', CLASS_NUM)
```

输出结果为：

```
类别个数： 3755
```

可以知道本章使用的 CASIA 的标签个数为 3755。接着，需要将名称标签转换为数值型数

组。首先，将类别名称转换为（编号 -> 类别名称）的映射关系：

```
将类别名称转换为 编号 -> 类别名称 的映射关系
```

```
cat_dict = dict(enumerate(class_names))
```

```
cat_dict[7]
```

显示结果为：`'凹'`（每次运行的结果可能一样）。我们可以通过索引获取类别名称，这样便可以将标签数组转换为数值型数组：

```
获取 index -> name 的 dict
```

```
name2index = {cat:cat_id for cat_id, cat in cat_dict.items()}
```

```
转换训练集的标签
```

```
_train_labels = np.array([name2index[cat_name] for cat_name in train_labels])
```

```
转换测试集的标签
```

```
_test_labels = np.array([name2index[cat_name] for cat_name in test_labels])
```

这里的标签类别个数有点多，下面仅仅是为了展示 TensorFlow 如何创建深度学习网络，没有必要使用那么多类别。下面仅仅选择 10 个字作为我们的学习目标。

```
class_names = ['登', '印', '枕', '孤', '孰', '熏', '琴', '驱', '吞',
```

```
'揪']
```

由于每一个 MPF 文件代表一个人写的汉字，所以需要在每一个 MPF 搜索存在 class\_names 的样本的类：

```
class SubCASIA(CASIAFeature):
```

```
 def __init__(self, class_names, hdf_path):
```

```
 """casia 子集数据 MPF 特征处理工具
```

```
 通过给定的 class_names 获取 CASIA 的子集数据
```

```
 ...
```

```
 self.h5 = tb.open_file(hdf_path)
```

```
 self.class_names = class_names
```

```
 def get_iter(self, super_iter):
```

```
 """从 super_iter 获取包含 self.class_names 的迭代器"""
```

```
 for features, labels in super_iter:
```

```
选择指定 class_names 的样本
frame = DataFrame(features, labels)
选择 frame.index 与 self.class_names 的交集
frame = frame.loc[frame.index & self.class_names]
features = frame.values
labels = frame.index.values
yield features, labels

def sub_train_iter(self):
 """从 self.train_iter() 获取包含 self.class_names 的迭代器"""
 return self.get_iter(self.train_iter())
def sub_test_iter(self):
 """从 self.test_iter() 获取包含 self.class_names 的迭代器"""
 return self.get_iter(self.test_iter())
```

先实例化：

```
class_names = ['登', '印', '枕', '孤', '美', '好', '琴', '驱', '吞',
'山']
mpf_dataset = SubCASIA(class_names, hdf_path)
```

我们可以看看训练集有多少个人写了汉字：

```
n_train = 0
for features, labels in mpf_dataset.sub_train_iter():
 n_train += 1
print(n_train)
```

显示结果为：1152。接着，统计有多少个样本：

```
n_train = 0
for features, labels in mpf_dataset.sub_train_iter():
 n_train += len(labels)
n_test = 0
for features, labels in mpf_dataset.sub_test_iter():
 n_test += len(labels)
print("训练集的样本数 {n_train}, 测试集的样本数 {n_test}")
```

显示结果：

```
训练集的样本数 11500, 测试集的样本数 2869
```

重置 cat\_dict：

```
cat_dict = dict(enumerate(class_names))
```

还需要重新划分数据集：

```
重新划分数据集为训练集与测试集
train_features = np.concatenate([features for features, _ in mpf_dataset.sub_train_iter()])
train_labels = np.concatenate([labels for _, labels in mpf_dataset.sub_train_iter()])
test_features = np.concatenate([features for features, _ in mpf_dataset.sub_test_iter()])
test_labels = np.concatenate([labels for _, labels in mpf_dataset.sub_test_iter()])
```

将标签数组转换为数值型数组：

```
获取 index -> name 的 dict
name2index = {cat:cat_id for cat_id, cat in cat_dict.items()}
转换训练集的标签
train_labels = np.array([name2index[cat_name] for cat_name in train_labels])
转换测试集的标签
test_labels = np.array([name2index[cat_name] for cat_name in test_labels])
```

接着，将 NumPy 数组转换为 TensorFlow 的迭代器：

```
train_dataset = tf.data.Dataset.from_tensor_slices((train_features, train_labels))
test_dataset = tf.data.Dataset.from_tensor_slices((test_features, test_labels))
```

接着，将数据分成批量，且打乱训练集数据：

```
BATCH_SIZE = 64
SHUFFLE_BUFFER_SIZE = 100
train_dataset = train_dataset.shuffle(SHUFFLE_BUFFER_SIZE).batch(BATCH_SIZE)
test_dataset = test_dataset.batch(BATCH_SIZE)
```

定义损失函数，评估函数，优化方法：

```
定义损失函数
loss_object = tf.keras.losses.SparseCategoricalCrossentropy()
定义模型训练的优化方法
optimizer = tf.keras.optimizers.Adam()
选择衡量指标来度量模型的损失值（loss）和准确率（accuracy）。这些指标在 epoch 上累积值，然后打印出整体结果
train_loss = tf.keras.metrics.Mean(name='train_loss')
train_accuracy = tf.keras.metrics.SparseCategoricalAccuracy(name='train_accuracy')
test_loss = tf.keras.metrics.Mean(name='test_loss')
test_accuracy = tf.keras.metrics.SparseCategoricalAccuracy(name='test_accuracy')
```

使用`tf.GradientTape` 定义如何训练模型以及评估模型性能：

```
定义训练策略
@tf.function
def train_step(images, labels):
 with tf.GradientTape() as tape:
 predictions = model(images)
 loss = loss_object(labels, predictions)
 gradients = tape.gradient(loss, model.trainable_variables)
 optimizer.apply_gradients(zip(gradients, model.trainable_variables))
 train_loss(loss)
 train_accuracy(labels, predictions)
定义评估方法
@tf.function
def test_step(images, labels):
 predictions = model(images)
 t_loss = loss_object(labels, predictions)
 test_loss(t_loss)
 test_accuracy(labels, predictions)
```

自定义模型：

```
class MyModel(Model):
 def __init__(self, CLASS_NUM):
 super().__init__()
 self.d1 = Dense(300, activation='relu')
 self.d2 = Dense(CLASS_NUM, activation='softmax')
 def call(self, x):
 x = self.d1(x)
 return self.d2(x)
实例化模型
CLASS_NUM = len(cat_dict)
model = MyModel(CLASS_NUM)
```

训练并评估模型：

```
EPOCHS = 20
模型训练的过程
for epoch in range(EPOCHS):
 for images, labels in train_dataset:
 train_step(images, labels)
 for test_images, test_labels in test_dataset:
 test_step(test_images, test_labels)
 template = 'Epoch {}, Loss: {}, Accuracy: {}, Test Loss: {}, Test Accuracy: {}'
 print(template.format(epoch+1,
 train_loss.result(),
 train_accuracy.result()*100,
 test_loss.result(),
 test_accuracy.result()*100))
重置 metrics
train_loss.reset_states()
train_accuracy.reset_states()
test_loss.reset_states()
test_accuracy.reset_states()
```

最终的结果可以查看图 14.4。从图 14.4 可以看出随着训练次数的增加训练集和测试集上的损失函数的值都在下降，同时训练和测试的精度也在不断的升高。由此可以看出，我这随意定义的模型的泛化性能还是不错的嘛。

```
Epoch 1, Loss: 1.7907116413116455, Accuracy: 92.7217309326172, Test Loss: 1.39178304536468, Test Accuracy: 73.1613846251593
Epoch 2, Loss: 0.180942964146203, Accuracy: 97.5781057421075, Test Loss: 0.302003520527011, Test Accuracy: 97.00244140625
Epoch 3, Loss: 0.02146561616965, Accuracy: 99.3043675485826, Test Loss: 0.102638015707008, Test Accuracy: 98.222539610234
Epoch 4, Loss: 0.0263854058949771, Accuracy: 99.8000033157581, Test Loss: 0.074927834956404, Test Accuracy: 97.46338510742
Epoch 5, Loss: 0.03617050447003035, Accuracy: 99.8695538227581, Test Loss: 0.221827099994041, Test Accuracy: 97.829837016291
Epoch 6, Loss: 0.028974921462035, Accuracy: 99.7836881622356, Test Loss: 0.177032495391098, Test Accuracy: 98.15723988136
Epoch 7, Loss: 0.010779192719075, Accuracy: 99.778910294994, Test Loss: 0.165304170199768, Test Accuracy: 98.251236691592
Epoch 8, Loss: 0.0396664504545885, Accuracy: 99.8726538936583, Test Loss: 0.04970291562228, Test Accuracy: 97.968837785547
Epoch 9, Loss: 0.01807235181334053, Accuracy: 99.7931291127927, Test Loss: 0.24186715480665466, Test Accuracy: 97.316139211914
Epoch 10, Loss: 0.0518227235198021, Accuracy: 99.4886016164746, Test Loss: 0.331700637703671, Test Accuracy: 96.897972949469
Epoch 11, Loss: 0.01567090311867567, Accuracy: 99.680065783694, Test Loss: 0.2276575207076684, Test Accuracy: 98.306652217069
Epoch 12, Loss: 0.00840747036369782, Accuracy: 99.8173904104053, Test Loss: 0.048173029980345, Test Accuracy: 98.36301236165894
Epoch 13, Loss: 0.0178329191386554, Accuracy: 99.659633217538, Test Loss: 0.166083524242616, Test Accuracy: 98.396652217699
Epoch 14, Loss: 0.019365761905244, Accuracy: 99.7394060032734, Test Loss: 0.176687916211554, Test Accuracy: 98.50122070125
Epoch 15, Loss: 0.03902457619325, Accuracy: 99.530131645505, Test Loss: 0.251023013130688, Test Accuracy: 98.865789195701
Epoch 16, Loss: 0.0242866221121910, Accuracy: 99.6566323945703, Test Loss: 0.399627764806476, Test Accuracy: 98.0491033251953
Epoch 17, Loss: 0.02954400202302046, Accuracy: 99.727479236565, Test Loss: 0.71496383222253, Test Accuracy: 95.533738720703
Epoch 18, Loss: 0.02096932117363894, Accuracy: 99.7391281272937, Test Loss: 0.2505144941463, Test Accuracy: 98.745367402438
Epoch 19, Loss: 0.01940937173166275, Accuracy: 99.7562312321422, Test Loss: 0.296591488752563, Test Accuracy: 98.1526641945701
Epoch 20, Loss: 0.02332359116568018, Accuracy: 99.7304930600734, Test Loss: 0.53125774974426, Test Accuracy: 96.8630123536594
```

图 14.4 模型的训练和评估结果

## 14.6 本章小结

本章主要介绍了 TensorFlow2 的使用教程。首先，利用 FASHION-MNIST 数据集展示了 TensorFlow 如何使用 `tf.keras.Sequential` 与 `tf.keras.layers` 搭建模型；接着，比较了 MXNet、Pytorch、TensorFlow 对于数据的操作以及自动微分的使用的不同；最后，介绍如何在数据集 CASIA 上利用 Model 自定义模型。