编　　者

- 主编：严晓东
- 编辑秘书：陈熙；崔文海；贾锦然；吉晓婷；连蓓蓓；刘亚萍；任静；宋佳珊

目　录

第 1 章　python 与爬虫

1.1　python 简介

python 是一种开源的面向对象的解释型语言，语法简洁，使用空白符作为缩进，一般为四个空白符作为一个缩进。python 中具有各种各样的库函数，能够方便的调用其它计算机语言制作的功能模块。

1.1.1　python 解释器

python 解释器的主要作用是执行 py 类型的代码，可以在官网上进行下载。

python 官网地址：*https*：*//www.python.org/*

打开 python 官网后点击上方菜单栏 Downloads，便可以进入 python 解释器的下载。如图 1.1。

目前 python 解释器的最新版本为 3.9.2，也可以选择下载旧版本，可以根据自己的需要选择下载相应的 python 版本。python 版本选择如图 1.2 所示，点击相应的 python 解释器版本后的 Download，我们就可以选择和电脑相应系统的解释器进行下载。

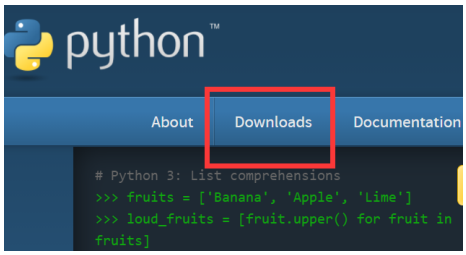
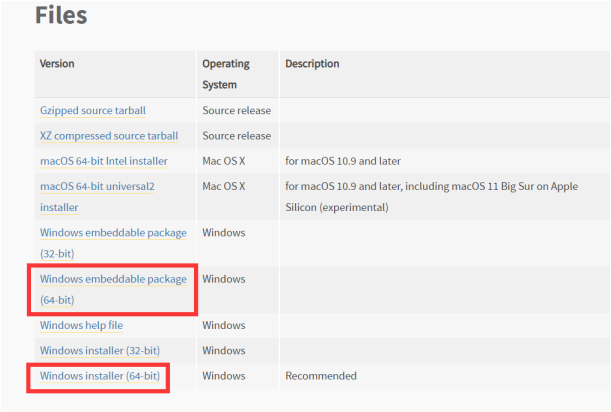


图 1.1: python 官网下载

图 1.2: 不同版本 python 选择

例如下载 python3.9.2 的解释器，若电脑是 64 位的 Windows 系统，这时候可以选择 Windows embeddable package(64-bit) 或者 Windows install(64-bit)。如图 1.3。



Files

Version	Operating System	Description
Gzipped source tarball	Source release	
XZ compressed source tarball	Source release	
macOS 64-bit Intel installer	Mac OS X	for macOS 10.9 and later
macOS 64-bit universal2 installer	Mac OS X	for macOS 10.9 and later, including macOS 11 Big Sur on Apple Silicon (experimental)
Windows embeddable package (32-bit)	Windows	
Windows embeddable package (64-bit)	Windows	
Windows help file	Windows	
Windows installer (32-bit)	Windows	
Windows installer (64-bit)	Windows	Recommended

图 1.3: python 下载

这两个的区别在于：前者下载后是一个 zip 类型的压缩包，解压缩后可以直接完成 python 解释器的安装；后者下载后是一个 exe 类型的文件，下载好后双击然后一直选择下一步便可以安装好 python 解释器。

python 解释器安装好后一般默认在 C 盘，如果不希望下载到默认路径，也可以自定义路径。如果在安装过程中遗忘 python 解释器的安装路径，可以通过在命令提示符输入 *where python* 查看。如图 1.4。

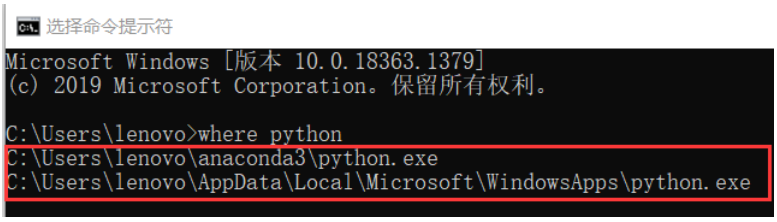


图 1.4: python 安装位置

python 安装成功后，输入 python，便可以进入 python 环境做各种操作。例如图 1.5，使用 python 进行各种计算，python 作为计算器非常好用，不会溢出。

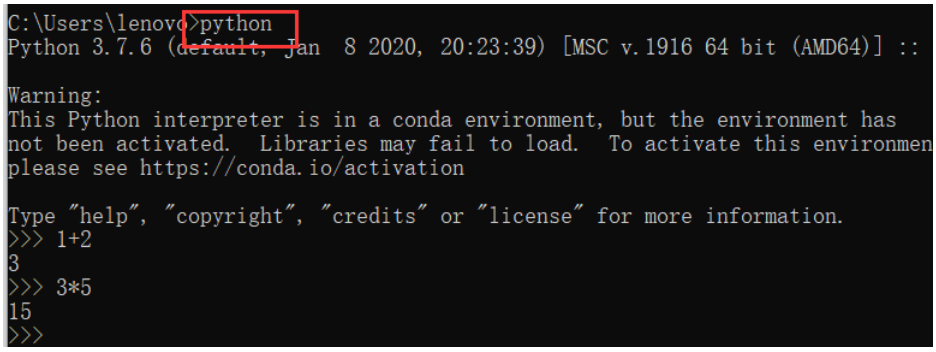


图 1.5: python 命令提示符调用

命令提示符中输入 python 进入的环境与高级系统设置中环境变量 path 有关，例如图 1.6

所示的 python 解释器的安装路径和环境变量 path 中的设置是一致的。

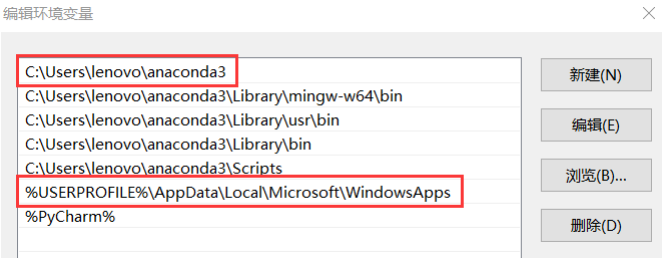


图 1.6: python 环境变量设置

1.1.2 python Package

python 的 Package 实际上是用 C/C++ 写成的各种算法。由于 python 是面向对象编程，本身不利于海量数据为基础的数据挖掘。相比之下，C/C++ 更适合做大量数据运算。但是 python 本身的特点是代码简洁方便，所以可以将其与 C/C++ 在数据运算方面的特长相结合，便形成了各种各样的 python 包。例如 *numpy* 就是以 C、C++ 和 Fortran 为核心内部实现语言的 python Package，而 *numpy* 的外部实现语言是 python。

python 包的调用看起来像是用 python 去写代码，但实际上是 C/C++ 在实现数据运算，所以运行速度相比 python 本身要更快。python 的重要价值就在于 Package 的使用上。python 中各种包可以从官网上去下载，在图 1.7 所示的搜索框中输入需要下载的 Package 就可以直接下载。

Package 的下载地址：*https* : *//pypi.org/*。

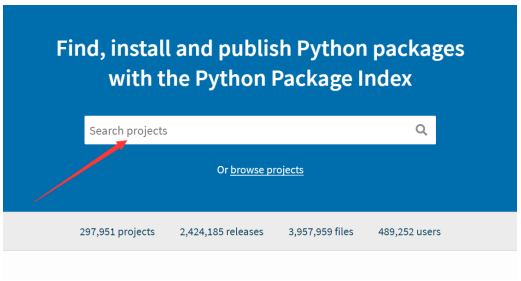


图 1.7: python Package 下载

但是一般更经常使用的方式是通过 pip 进行包的安装，在 python 解释器安装时 pip 也会相应的安装，但是 pip 同样需要加入到环境变量中。例如图 1.8 所示的 Anaconda 中的 python 解释器对应的 pip 配置在环境变量中路径是在 Scripts 文件夹下。

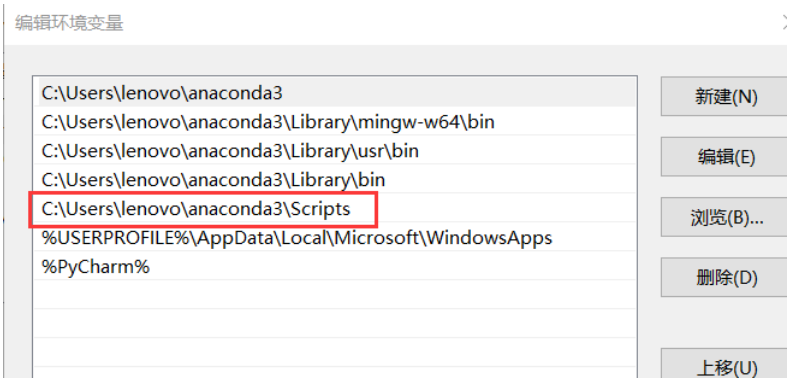


图 1.8: python Pip 配置

包的安装常用的方式是在终端输入 *pip install* (*package name*)。但是由于有些包比较复杂，在安装过程中速度会比较慢，这时候可以借助镜像。常用的是清华大学的镜像网站，也就是在安装时更换为国内源：

pip install $-i$ *https* : *//pypi.tuna.tsinghua.edu.cn/simple/* (*package name*)

python 的 Package 在使用过程中需要注意包与包之间的依赖关系，例如进行 *tensorflow* 的安装就依赖于 *numpy*、*pandas* 等包，所以在 *tensorflow* 的安装过程中会同时把 *numpy*、*pandas* 等包安装上。

python 的各种包还需要注意版本问题，有些包并不一定支持到 python 的最新版本。如图 1.9。以 *tensorflow* 为例，*tensorflow*2.4 能够支持 python3.6、python3.7、python3.8，但并不支持 python3.9，所以在安装 python 的时候并不是 python 的版本越新越好。

图 1.9: Package 版本选择

1.1.3 python IDE

1、Anaconda

Windows 下的 Anaconda 安装比较方便，直接去 Anaconda 的官方网站下载即可。

Anaconda 下载地址：*https*：*//www.anaconda.com/products/individual*。

打开网址后，点击 Download，页面会直接跳转到如图 1.10 所示的下载界面。如图 1.11，在 Windows 下选择下载相应的 Anaconda 应用程序即可。

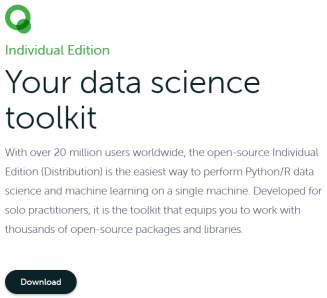


图 1.10: Anaconda 下载界面

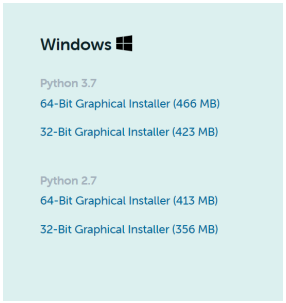


图 1.11: 选择 Anaconda 应用程序

在 Anaconda 应用程序下载完成后，当安装到图 1.12 这一步时，上面那个方框要打勾，表示安装成功后会自动配置环境变量，否则在 Anaconda 安装后需要自行配置环境变量，环境变量如果不配置就无法正常使用 Anaconda。

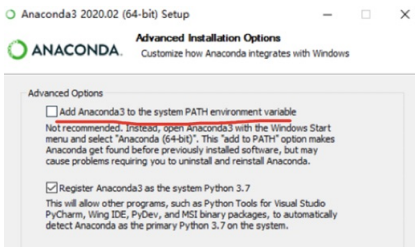


图 1.12: Anaconda 勾选配置环境变量

判断 Anaconda 是否安装成功，可以在命令提示符中进行，选择 Win+R，在弹出的对话框中输入 *cmd*，即打开了命令提示符，在命令提示符中输入 *conda* $--$*version*，其中 conda 是 Anaconda 中 python 包的管理工具，如果下方出现 conda 相应的版本，即表示 Anaconda 安装成功，可以正常使用，如图 1.13。

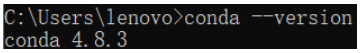


图 1.13: 查看 Anaconda 是否安装成功

Anaconda 安装成功后会得到图 1.14 中的几个 exe，我们在写 python 程序时用到比较多的是 Spyder，有时也会用到 Jupyter Notebook，后面我们会介绍下相关使用。

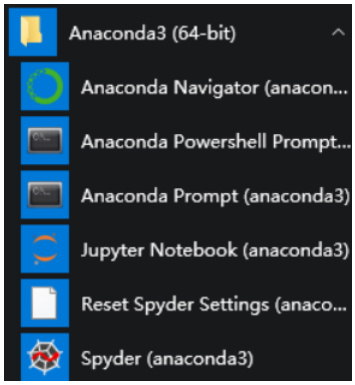


图 1.14: Anaconda 的 exe

Anaconda 安装成功后，我们需要了解如何查看、安装和卸载一些 python 包，Anaconda 相比 python 解释器速度要更快，所以在安装 Package 时更加方便。具体操作为：

(1) 查看已安装的 python 包。在命令提示符下输入 *conda list*，如图 1.15。

```
C:\Users\lenovo>conda list
# packages in environment at C:\Users\lenovo\anaconda3:
#
# Name                    Version                   Build  Channel
_ipyw_jlab_nb_ext_conf    0.1.0                    py37_0
_pytorch_select           1.1.0                      cpu
_tflow_select             2.2.0                    eigen
absl-py                   0.9.0                    py37_0
alabaster                 0.7.12                   py37_0
anaconda                  2020.02                  py37_0
anaconda-client           1.7.2                    py37_0
anaconda-navigator        1.9.12                   py37_0
```

图 1.15: 查看已安装 python 包

(2) 安装新的 python 包。命令提示符下输入“*conda install* (*package name*)”或者“*pip install* (*package name*)”，其中 pip 是 python 包的管理工具。例如如果我们要下载 *numpy*，我们可以输入 *conda install numpy* 或者 *pip install numpy*，如图 1.16。

```
C:\Users\lenovo>conda install numpy
```

图 1.16: *numpy* 包的安装

有些 python 包下载可能比较慢，这时候可以使用清华大学的镜像网站来下载提高速度，具体使用方法如图 1.17，需要在安装时输入：
pip install -i https : *//pypi.tuna.tsinghua.edu.cn/simple* (package name)

```
C:\Users\lenovo>pip install -i https://pypi.tuna.tsinghua.edu.cn/simple numpy
```

图 1.17: 使用镜像网站安装 python 包

(3) 卸载 python 包。在命令提示符下输入 *conda uninstall* (*package name*) 或者 *pip uninstall* (*package name*)，例如如果要卸载 *numpy*，就输入 *conda uninstall numpy* 或者 *pip uninstall numpy*，如图 1.18。

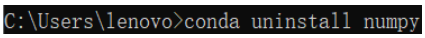


图 1.18: *numpy* 包的卸载

(4) 虚拟环境的创建。输入 $conda\ info\ --envs$，表示查看现在已经安装的环境，右边“*”表示当前使用的环境，如图 1.19。

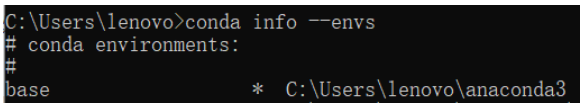


图 1.19: 虚拟环境的创建

创建 $Tensorflow$ 环境，输入 $conda\ create\ -n\ tensorflow\ python = 3.7$，表示创建一个名字为 $tensorflow$ 的环境，这个环境用的 python 版本是 3.7 版本的，如图 1.20。创建环境时你也可以用你自己喜欢名字去命名。

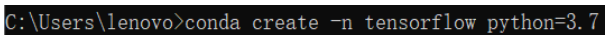


图 1.20: $Tensorflow$ 环境的创建

创建成功后再输入 $conda\ info\ --envs$，就会发现下方多了名为一个 $tensorflow$ 的环境，我们就在这个环境下配置 $tensorflow$，如图 1.21。

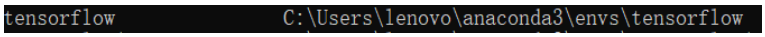


图 1.21: 虚拟环境的查看

至此，我们就完成了环境的建立，接下来我们就要在这个环境下配置 $Tensorflow$，首先进入这个环境，直接输入 $activate\ tensorflow$，我们就可以进入这个环境，进入之后如图 1.22，之后我们继续安装 python 库函数，都是安装在了名为 $tensorflow$ 这个环境下。

图 1.22: 虚拟环境的调用

2、PyCharm

官网可以直接下载，下载好之后正常安装就可以了，选 Next。到选择安装目录的时候，可以根据自己的需求进行修改。如图 1.23。选择好安装目录继续选择 Next，如图 1.24，下一个对话框我这里全都勾选的（仅供参考）。

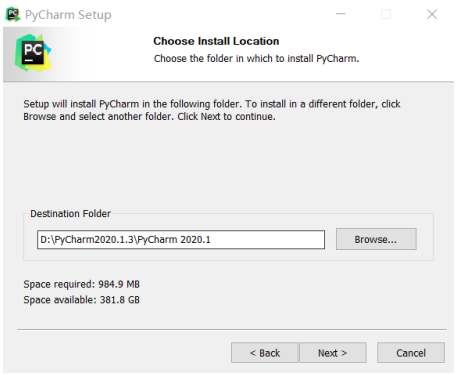


图 1.23: 选择安装目录

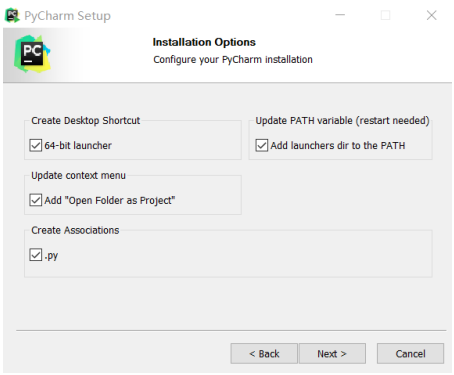


图 1.24: 安装选项

继续 Next，如图 1.25 的对话框出现之后选择 Install 就可以了。图 1.25 表示，当 PyCharm 安装成功开始运行后，选择要在其中创建程序快捷方式的开始菜单文件夹，也可以输入名称来创建新文件夹。

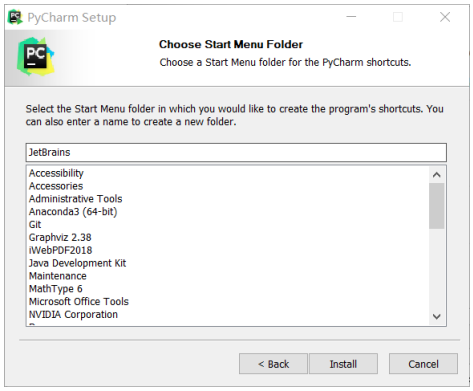


图 1.25: 选择开始菜单

PyCharm 是个平台，跟 Anaconda 不一样，里面没有内置的 python 编辑器之类的，所以需要事先设置一个 python 环境，可以把 Anaconda 中的 python 环境放到 PyCharm 中去用。

打开 Pycharm 之后首先创建一个 New Project，比如 test（随便取名），然后在菜单栏选择 File，下拉以后选择 Settings。

打开 Settings 之后选择 Project:test 这一项，然后点击 python Interpreter，如图 1.26。

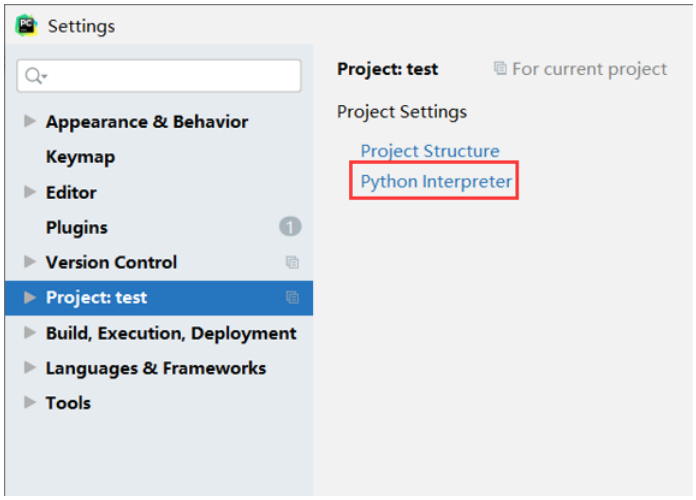


图 1.26: python 解释器的添加

找到上方右边设置的图标，点击之后选择 Add，如图 1.27。

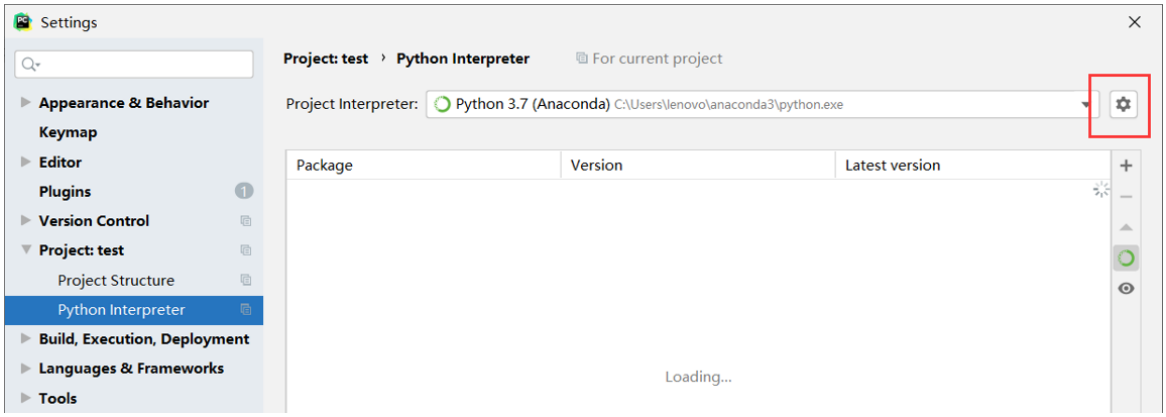


图 1.27: python 解释器的添加

选择 Add 之后会出现如图 1.28 的对话框，选择 Existing environment，表示把电脑上已经有的 python 编译器导入到 PyCharm 里面，然后点击图 1.28 箭头指的那个图标，就可以选择电脑上已经有的 python 编译器：

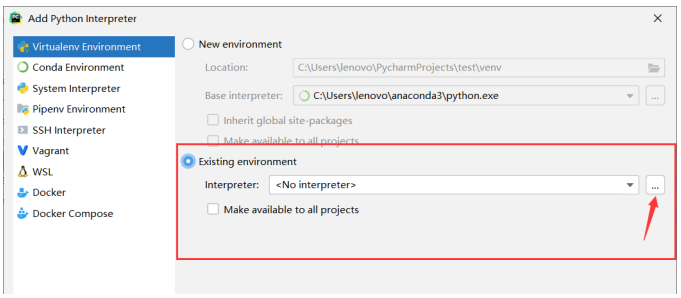


图 1.28: python 解释器的添加

下一步找到 Anaconda3 安装的位置，我安装的时候是默认位置，所以安装位置是在用户那个目录下面，然后找到 python.exe，点击之后选择 OK。如图 1.29 所示。之后又会回到之前的对话框，注意要勾选 Make available to all projects，它表示之后创建任何 python 项目，都不需要再重新导入 python 编辑器了，勾选之后选 OK 就可以了。如图 1.30 所示。

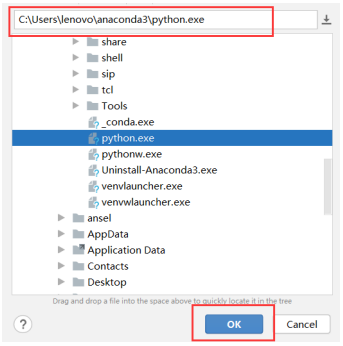


图 1.29: python 解释器的添加

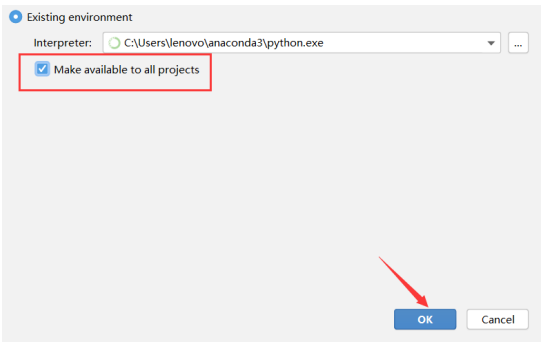


图 1.30: python 解释器的添加

如图 1.31 所示，可以发现这里有了 python 编译器，然后正常使用就可以了。

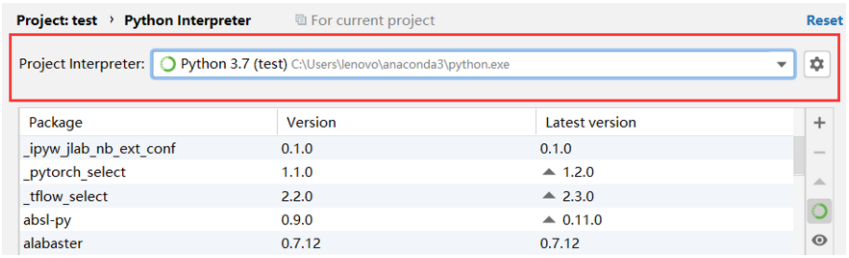


图 1.31: python 解释器的添加

在 PyCharm 里用 Anaconda 的 python 编译器的时候，*numpy* 之类的包还是可以在 *cmd* 里通过 *conda install numpy* 去安装。PyCharm 实际上就是提供了一个写代码的平台，这里你可以导入各种 python 编译器。

1.2 python 简单案例

1.2.1 图片转换为字符

每张图片都是由不同大小的像素组成的，图片上任一点的像素取值在 0-255 之间。通过读取图片上的每一点的像素值，并将该像素值用对应的字符替换，能够实现图片向字符的转化。

例如下面的例子将图 1.32 的 Lena 图片转化为字符。使用字符 “$MNHQ\$OC?7>!:-;.\$” 来替换像素值，其中像素取值 0-15 时将图片上对应点用字符 “M” 替换，像素值为 16-30 将对应像素点用 “N” 替换……以此类推，因此图像上像素值为 240-255 将由 “\” 来替换。转化后的字符用 word 打开后的效果如图 1.33。具体代码实现如下。

图 1.32: Lena 原始图片

图 1.33: Lena 字符图片

```
import numpy as np
from PIL import Image #读入图像
import os
import subprocess #滚动输出
if __name__ == '__main__':
    image_file = 'lena.png'
    height = 100 #输出 100 行的字符
    img = Image.open(image_file) #图像打开函数
    print(img)
    img_width, img_height = img.size
    #设高度是宽度的 2 倍，令高度为 100
    width = int(2*height*img_width//img_height)
    img = img.resize((width, height), Image.ANTIALIAS) #图像缩放
    pixels = np.array(img.convert('L')) #转化成灰度图
    print('type(pixels)=', type(pixels))
    print(pixels.shape)
    print(pixels)
    chars = "MNHQ$OC?7>!:-;. " # 0-255 像素的值对应 16 个字符
    N = len(chars)
    step = 256 // N
    result = ''
    for i in range(height):
        for j in range(width):
            result += chars[pixels[i][j]//step]
        result += '\n'
    #写入 txt 文件中
    image_file_name = os.path.splitext(image_file)[0] + '.txt'
    with open(image_file_name, mode='w') as f:
        f.write(result)
    print(image_file_name + '文件生成完成。')
    subprocess.call(['notepad', './lena.txt'], shell=False)
```

1.2.2 集成算法的准确率

假如样本只有正例（用 1 表示）与负例（用 0 表示）两个类别，来做两分类模型，模型能够进行预测，但模型的准确率并不是 100%。假如存在相互独立的几个模型，每个模型都具有 60% 的准确率，这几个模型可能构成一个新模型。现在需要求预测准确率为 60% 的几个相互独立的模型，当它们构成一个集成模型时，预测的准确率有多大。

计算方法为：如果有五个模型，五个模型中有三个预测正确，则集成模型准确率为 $C_5^3(0.6)^3\times 0.4^2$；如果有四个预测准确，则集成模型准确率为 $C_5^4(0.6)^4\times 0.4^1\cdots\cdots$因此该集成模型的准确率是能够进行计算的。因此可以通过构造函数来实现，该函数能够计算不同个数的模型构成的集成模型的准确率。假设有 100 个独立模型，计算不同个数的模型构造的集成模型的预测准确率的具体实现代码如下。

首先导入所需的计算和绘图模块，求包括 n 个独立模型的集成模型的正确率。

```
import operator
import numpy as np
import matplotlib.pyplot as plt
import matplotlib as mpl          # 绘图包
from functools import reduce
from scipy.special import comb
def c(n, k):
    return reduce(operator.mul,
        list(range(n-k+1, n+1))) /
        reduce(operator.mul, list(range(1, k+1)))
# 求包括 n 个独立模型的集成模型的正确率
def bagging(n, p):
    s = 0
    for i in range(n//2+1, n+1):
        s += comb(n, i)*p**i*(1-p)**(n-i)
    return s
if __name__ == "__main__":
    n = 200 # 模型个数
    x = np.arange(1, n, 4) # 设步长为 4
    y = np.empty_like(x, dtype=np.float) # 初始化y，与x维度相同
    # 遍历 x ，计算正确率 y
    for i, t in enumerate(x):
        y[i] = bagging(t, 0.6)
        if t % 10 == 9:
            print(t, '每个分类器的正确率：', y[i])
```

进行可视化处理。设置坐标图的横纵坐标名称、刻度范围，绘出坐标点和准确率曲线，输出图形。

```
    # 进行可视化
    mpl.rcParams['font.sans-serif'] = 'simHei'
    mpl.rcParams['axes.unicode_minus'] = False
    plt.figure(facecolor='w')
```

```
    plt.plot(x, y, 'ro-', lw=2, mec='k')
    plt.xlim(0, n)
    plt.ylim(0.6, 1)
    plt.xlabel('分类器个数', fontsize=16)
    plt.ylabel('正确率', fontsize=16)
    plt.title('随机森林正确率', fontsize=20)
    plt.grid(b=True, ls=':', color='#606060')
    plt.show()
```

代码中对集成模型随着独立模型个数增加时准确率的变化进行了可视化，如图 1.34。

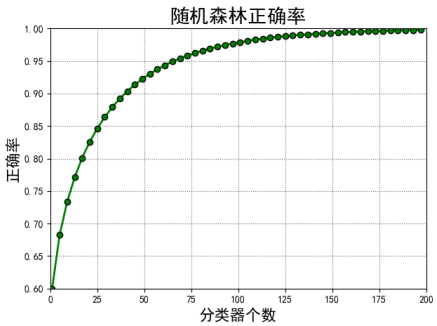


图 1.34: 随机森林准确率可视化

1.2.3 赢得一局比赛的可能性

假如 A 和 B 在比赛中，A 得 1 分的概率为 60%。现在如果只打一局比赛（一局比赛 11 分），那么 A 有多大概率获胜?

第一种方法可以使用 python 进行暴力模拟，假设模拟足够多局比赛（这里模拟了 1000000 局），在这些比赛中统计 A 赢得比赛的局数，最后便可以计算 A 赢得一局比赛的概率。具体实现代码为:

```
if method == 'simulation':
    p = 0.6
    # a , b 分别表示两位选手， c 表示 A 的局数
    a, b, c = 0, 0, 0
    # 假定统计足够多次的模拟（T=1000000），t表示当前的计数情况
    t, T = 0, 1000000
    while t < T:
        a = b = 0
        # 只要双方都没有获得 11 分，就继续比赛
        while (a<=11) and (b<=11):
            if np.random.uniform() < p:
                a += 1
            else:
                b += 1
        if a > b:
```

```
            c += 1
        t += 1
    # 用 A 赢得的局数与总局数相除来求 A 赢得一局比赛的概率
    print(float(c)/float(T))
```

其次可以通过直接计算的方式。在 11 分中，只要 A 与 B 的比分为 11:0、11:1…11:9，A 都可以赢得比赛。具体实现代码为：

```
elif method == 'simple':
    answer = 0
    # 每分的胜率
    p = 0.8364351998415863
    # 每局多少分
    N = 3
    # x 为对手得分
    for x in np.arange(N):
        answer += special.comb(N+x-1, x)*((1-p)**x)*(p**N)
    print(answer)
```

最后如果严格考虑 10:10 情况，如果 A 能够获得两分，则能够赢得比赛。具体实现代码为：

```
else:
    answer = 0
    # 每分的胜率
    p = 0.6
    # 每局多少分
    N = 11
    # x 为对手得分：11:9  11:8  11:7  11:6...
    for x in np.arange(N-1):
        answer += special.comb(N+x-1, x)*((1-p)**x)*(p**N)
    # 10:10 的概率
    p10 = special.comb(2*(N-1), N-1)*((1-p)*p)**(N-1)
    t = 0
    for n in np.arange(100):
        t += (2*p*(1-p))**n*p*p
    answer += p10*t
    print(answer)
```

最后结果输出为 0.8256 ，这便是 A 赢得一局比赛的概率。

1.3 python 爬虫

1.3.1 url 的获取

爬虫获取数据的过程为，先获取网页上的全部内容并返回初始结果，分析网页中的内容，将网页中想要获取的内容保存。爬虫是根据被爬取的网页来编写相应的代码，一旦网页改版，原来的爬虫也会相应失效。

对于爬虫来说，需要一部分网页制作的基础，大体了解网页代码的结构，包括 table、div、class、tr、td 等基本标签的含义。

python 爬虫在爬取网页上的数据时，找到对应数据的 url 是比较关键的步骤。例如如果我们想爬五粮液官网的 logo 图片，首先我们需要获得该图片的 url。获取步骤为：

(1) 通过 F12 或者在网页空白处右击选择“检查”打开网页代码，代码一般会显示在网页右侧；

(2) 找到 logo 图片相应的 div 模块，在五粮液官网上，logo 对应的 div 命名为“logo”。如图 1.35 所示，当鼠标滑到这一模块时，在左侧网页对应对部分会相应的变成蓝色。

图 1.35: 获取五粮液 logo 的 url

(3) 如图 1.36 所示，打开 logo 的 div 模块，找到 img 标签，img 标签后的 src 即为该 logo 的 url。

图 1.36: 获取五粮液 logo 的 url

更简单的方法为，直接点击右侧快速查找，然后点击左侧页面对应内容，网页代码对应标签行会闪烁。例如点击左侧菜单栏中的“服务支持”时，会直接找到右侧网页代码中对应模块，如图 1.37：

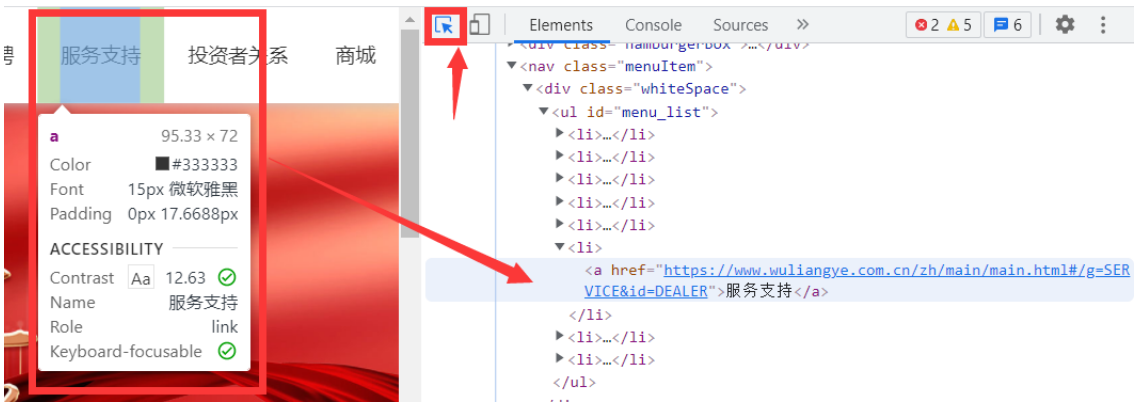


图 1.37: 查找对应网页代码

1.3.2 简单图片爬取案例

简单了解网页中 url 的获取后便可以编写爬虫获取网页中的图片。一个简单的例子是获取贵州茅台官网上的 logo 图片。具体步骤为：首先获取图片的 url，其次使用 *requests* 请求 url 响应对图片进行获取，最后将图片保存。获取到的五粮液 logo 如图 1.38。

图 1.38: 获得的五粮液 logo

具体实现代码如下：

```
import requests
if __name__ == '__main__':
    url = 'https://www.wuliangye.com.cn/zh/main/images/logo.svg'
    req = requests.get(url)
    f = open('wuliangye.svg', 'wb')
    f.write(req.content)
    f.close()
```

1.3.3 直接通过 url 内容获取已知数据——获取《青春有你 3》选手照片

爬取《青春有你 3》选手图片的步骤和 1.3.2 的爬取图片的过程类似：首先爬取《青春有你 3》百度百科网页中的内容，返回结果；其次对网页中的内容进行分析，获取《青春有你 3》选手图片；最后对图片进行保存，保存到代码指定文件夹中。具体操作过程及代码如下：

(1)*crawl_data*() 函数：爬取百度百科中《青春有你 3》中参赛选手信息，返回 html。在这里代码需要网页的 UserAgent。网页 UserAgent 的获取步骤为：右击网网页空白处 → 选择“检查”打开网页代码 → 选择“Network”选项 → 在“Name”下点击 → 右侧出现“Headers”下拉找到 User-Agent。如图 1.39。

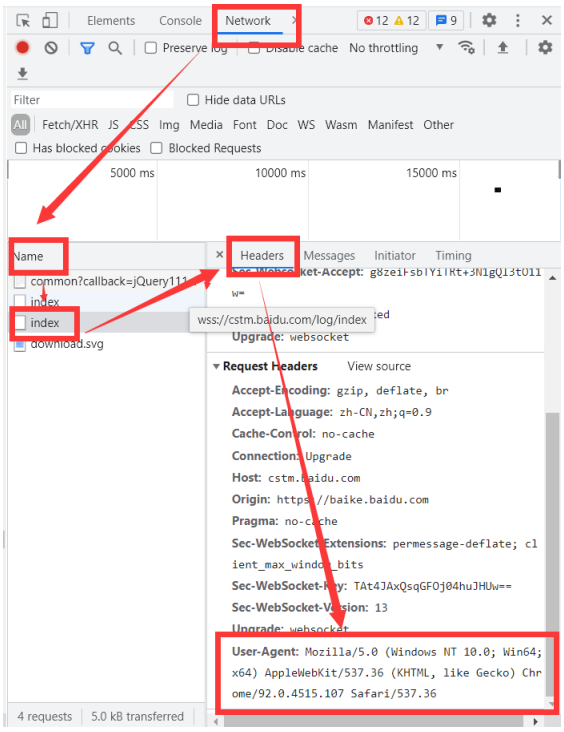


图 1.39: 查找 User-Agent

在这一步中通过 url 和设置 Headers 的 User-Agent 可以得到一个返回的响应，将网页的内容保存。如果是文本将会返回相应的文本的内容，同时用 *BeautifulSoup* 包对文本进行解析。

通过检查网页代码可以发现参赛学员一栏实际上是一个 table 模块,因此可以通过 *BeautifulSoup* 包找到所有的“table”，并选择 class 属性为“参赛选手”的 table。通过 *for* 循环找到 div 模块为“h3”的地方（在爬取网页信息过程中需要对应网页“检查”的代码），最终 *crawl_data*() 函数返回的 table 就是我们希望爬取的《青春有你 3》参赛选手对应的 url 所在的位置。

具体实现代码如下：

```
def crawl_data():
    # 网页入口
    url = 'https://baike.baidu.com/item/青春有你第三季'
    headers = {
        'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit
        /537.36 (KHTML, like Gecko) Chrome
        /92.0.4515.107 Safari/537.36'
    }
    try:
        response = requests.get(url, headers=headers)
        # status_code 返回值状态， 404 ， 200 ，正常等等
        print('code = {}'.format(response.status_code))
        # 表示用 lxml 解析
        soup = BeautifulSoup(response.text, 'lxml')
        # 通过检查网页找到参赛选手的 table，爬取内容
```

```
        tables = soup.find_all('table')
        tables =
        [table for table in tables if not table.has_attr('class')]
        crawl_table_title = "参赛选手"
        # 对多个表格进行遍历
        for table in tables:
            # 对当前节点前面的标签和字符串进行查找
            table_titles = table.find_previous('div').find_all('h3')
            for title in table_titles:
                if crawl_table_title in title:
                    return table
    except Exception as e:
        print(e)
    return None
```

(2)*parse_data*() 函数：解析 html，将其解析成我们想要存储的 json 格式。json 格式的文件用记事本打开后如图 1.40。

图 1.40: JSON 格式文件的获取

这一步同样需要用到 *BeautifulSoup* 包，根据行、列找到每个学员的姓名、百度百科链接、籍贯、身高、体重和所属公司信息。将这些信息存储为 json 格式，保存为以今天的日期命名的、后缀为 json 的文件，使用 json 格式的文件好处是可以直接变成结构化的数据。但是 json 格式的文件可以借助第三方工具去查看，例如 *www.sojson.com*，能够更直观地查看 json 文件的内容。如图 1.41。

图 1.41: JSON 格式文件的查看

parse_data() 函数具体实现代码如下：

```
def parse_data(table_html):
    bs = BeautifulSoup(str(table_html), 'lxml')
    all_trs = bs.find_all('tr')
    error_list = ['\'', '\"']
    stars = []
    for tr in all_trs[1:]:
        all_tds = tr.find_all('td')
        star = {}

        star["name"] = all_tds[0].text                  # 姓名
        star["link"] = 'https://baike.baidu.com'
            + all_tds[0].find('a').get('href')          # 个人百度百科链接
        star["zone"] = all_tds[1].text                  # 籍贯
        star["height"] = all_tds[2].text                # 身高
        star["weight"] = all_tds[3].text                # 体重
        if not all_tds[4].find('a') is None:            # 公司
            star["company"] = all_tds[4].find('a').text
        else:
            star["company"] = all_tds[4].text
        stars.append(star)
    json_data = json.loads(str(stars).replace("\'", "\""))
    print('json_data=', json_data)
    if not os.path.exists(output_path):
        os.makedirs(output_path)
    with open(os.path.join(output_path, today+'.json'),
    'w', encoding='utf-8') as f:
        json.dump(json_data, f, ensure_ascii=False)
```

(3)*crawl_pic_urls*() 函数：爬取图片的 url 并保存。在对 json 格式的文件进行解析后，对

文件中的内容进行筛选，判断是否是图片格式并且是否是链接形式，如果是，加上链接前缀并保存在图片列表 pic_urls 中。具体实现代码为：

```
def crawl_pic_urls():
    with open(os.path.join(output_path, today+'.json'),
    'r', encoding='UTF-8') as file:
        json_array = json.loads(file.read())
    headers = {
        'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64)
        AppleWebKit/537.36 (KHTML, like Gecko) Chrome
        /92.0.4515.107 Safari/537.36'
    }
    for star in json_array:
        name = star['name']
        link = star['link']
        # 将所有图片 url 存储在一个列表 pic_urls 中
        response = requests.get(link, headers=headers)
        bs = BeautifulSoup(response.text, 'lxml')
        try:
            # 查找是否是图片格式，并且查看是否是链接
            pic_list_url = bs.select('.summary-pic a')[0].get('href')
            # 取该图片的局部地址并加上前缀地址形成 url
            pic_list_url = 'https://baike.baidu.com' + pic_list_url
        except Exception as e:
            print('出现异常:{}'.format(str(e)))
            continue
        pic_list_response = requests.get(pic_list_url, headers=headers)
        bs = BeautifulSoup(pic_list_response.text, 'lxml')
        pic_list_html = bs.select('.pic-list img')
        pic_urls = []
        for pic_html in pic_list_html:
            pic_url = pic_html.get('src')
            pic_urls.append(pic_url)
        # 根据图片链接列表 pic_urls ，下载所有图片，保存在以 name 命名的文件夹中
        down_pic(name, pic_urls)
```

(4)*down_pic*() 函数：根据图片链接列表 pic_urls，下载所有图片，保存在命名为 name 的文件夹中。最终我们就可以获得所有参赛学员的图片。具体实现代码为：

```
def down_pic(name, pic_urls):
    path = os.path.join(output_path, 'pictures', name)
    if not os.path.exists(path):
        os.makedirs(path)
    for i, pic_url in enumerate(pic_urls):
        try:
            pic = requests.get(pic_url, timeout=15)
            with open(os.path.join(path, str(i+1)+'.jpg'), 'wb')
```

```
            as f:
                f.write(pic.content)
                print(name+'第%s 张图片: %s' %(str(i+1), str(pic_url)))
        except Exception as e:
            print('下载第%s 张图片时失败: %s' %(str(i+1), str(pic_url)))
            print(e)
            continue
```

最后经过主函数调用以上函数就可以完成《青春有你 3》参赛选手的图片，但是此次爬取的图片仅限于选手百度百科中存在的，因此爬取的结果有的选手图片多有的选手图片少。

1.3.4 给定 url 获取数据——获取股票数据

爬取《青春有你 3》选手图片时由于我们需要先根据网页获取选手对应的 url，因此代码冗长。但是如果在给定 url 的前提下，爬虫的代码要简短很多。例如如果我们想要爬取搜狐财经上的五粮液（股票代码：000858）的股票数据，可以直接在五粮液历史行情的网页上，找到历史股票数据对应的 url，然后根据这个 url 编写爬虫代码。

找到被爬取数据的 url 是比较关键的步骤，但是被爬取数据的 url 需要在网页源代码中查找。通过 F12 或者右击网页空白处选择“检查”，在右侧弹出的网页代码上方菜单栏选择 Network，然后查看 Name 一栏是否有和历史行情相关。如果没有，可以通过 Ctrl+R 对其进行刷新，刷新之后，我们发现在 Name 一栏增加了许多内容，如图 1.42。在 Name 一栏往下查找可以发现名为 hisHq 的一栏，因此我们猜测它所对应的 url 和五粮液历史股票数据有关，如图 1.43。

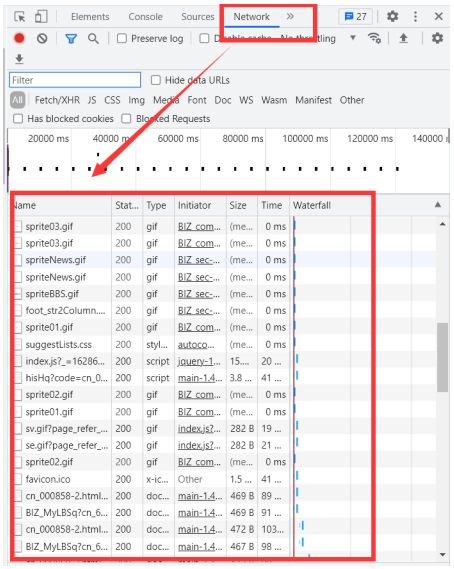


图 1.42: 选择 Network

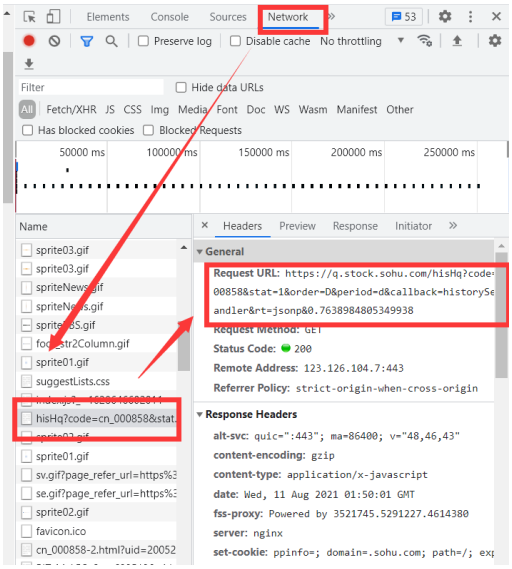


图 1.43: url 获取

我们可以尝试将这个 url 输入到地址栏中，然后就可以发现这个 url 能够返回股票数据的列表，而且它是 json 格式。如图 1.44。

图 1.44: 五粮液股票数据 url 的直接获取

因此后续我们就可以直接根据这个 url 来编写五粮液的爬虫。由于它返回的是 json 格式的结果，所以通过解析这个结果，就可以爬取五粮液的历史股票数据。

具体实现代码如下：

```
if __name__ == '__main__':
    # 根据历史股票数据对应的url获得json格式的结果，保存在data中
    url_WLY = 'https://q.stock.sohu.com/hisHq?
    code=cn_000858&start=20200101&end=20210811
    &stat=1&order=D&period=d&callback=historySearchHandler
    &rt=jsonp&0.7638984805349938'
    response = requests.get(url_WLY)
    data = response.text
    # 对 data 进行解析
    start = data.find('{')
    end = data.find('}')
    data = data[start:end+1]
    print(data)
    # 将数据变成 key-value 的字典格式
    data = json.loads(data)
    print(data)
    # 将字典中的值转成 ndarray
    data = data['hq']
    data = np.array(data)
    data = data[:, [0,1,2,5,6,7,8]]
    # 将 ndarray 转换成 DataFrame 格式
    data = pd.DataFrame(data, columns=['日期', '开盘', '收盘', '最低','最高', '
        成交量(手)', '成交金额(万)'])
    data.set_index(keys=['日期'], inplace=True)
    data.sort_index(inplace=True)
    print(data)
    ata.to_excel('WLY_20210811.xls')  # 保存为 excel 文件
```

这种根据 url 爬取数据的方式比解析的方式要方便，只要找到数据对应的 url，便可以直接进行操作。因此如果知道 url 可以采用这一节的方式，但是如果不知道 url 就需要根据 1.3.3 的方式来操作。

1.4 python 数据分布

python 数据分布实际上是作为逻辑回归、聚类等内容的预备知识。这一部分内容主要包括以下几部分：验证中心极限定理、3D 绘图、统计量的计算、高斯分布的绘制、阶乘和 Gamma 函数、验证 Pearson 相关系数、奇异值分解。

1.4.1 验证中心极限定理

如果我们随机取 0—1 之间的 10000 个浮点数，这 10000 个数值满足均匀分布，通过 python 的 *matplotlib* 包对其进行可视化，例如我们采用直方图来显示，如图 1.45。

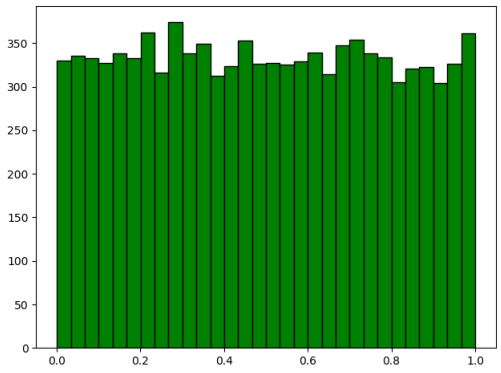


图 1.45: 均匀分布的直方图

其中横轴表示每个数值范围在 0—1 之间，纵轴表示这 10000 个数值中落在这个区间的个数。具体实现代码如下：

```
# 强制使用 numpy 的浮点数输出显示
np.set_printoptions(suppress=True)
# 随机取 0—1 之间的 100 个数
d = np.random.uniform(0, 1, 10000)
print(d)
# # 对数据进行可视化，横轴0—1 ，纵轴0—100
# plt.plot(d,'go',ms=4)
# 也可以采用直方图来显示，横轴为0—1，纵轴表示该值落在区间的个数
# 如果数据的数量越大，分布越接近均匀分布
plt.hist(d, bins=30, color='g', edgecolor='k')
plt.show()
```

如果我们随机生成 100 个均匀分布，每个分布都包括 10000 个 0—1 之间的数值，将这 100 个分布相加后，就可以得到高斯分布，如图 1.46。具体实现代码如下：

```
np.set_printoptions(suppress=True)
N = 10000
z = np.zeros(N)
for i in range(100):
```

```
    z += np.random.uniform(0, 1, 10000)
z /= 100
plt.hist(z, bins=30, color='g', edgecolor='k')
plt.show()
```

如果我们随机生成 100 个泊松分布，其中 λ 取 10，每个分布包括 10000 个数值，将这 100 个泊松分布相加后，仍然可以得到高斯分布，如图 1.47。实现代码与均匀分布时类似，只需要修改以下代码即可。

```
z += np.random.poisson(10, 10000)
```

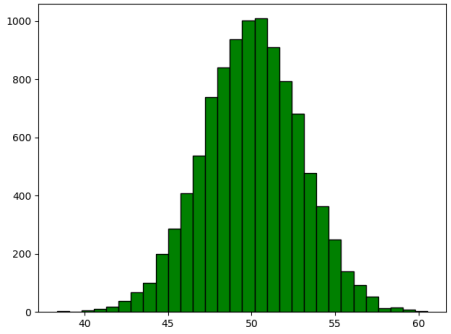


图 1.46: 均匀分布相加后的直方图

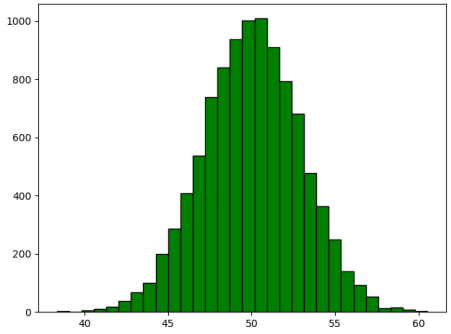


图 1.47: 泊松分布相加后的直方图

1.4.2 3D 绘图

假如绘制 $z = x^2/3 + y^2/3$ 的 3D 图形，x 和 y 分别取-50—50 之间的 1000 个任意的数。

在 *matplotlib* 包中包含可以设置 $3D$ 坐标轴的函数，在 *numpy* 包中包括网格点坐标矩阵函数。将 x 和 y 输入到网格点坐标矩阵函数中，x 和 y 就变成了二维数据。但是对于 x 来说，x 的每一行都是相同的 1000 个数值，y 的每一列都是相同的 1000 个数值。因此 x 和 y 就构成了 1000*1000 的格点数据。然后计算 z 的值，将 x、y、z 放在一个 3D 坐标系中便能够绘制三维曲面图。

具体实现代码为：

```
ax = plt.axes(projection='3d')
# x, y 都是 -50 到 50 的随机值, 各 1000 个
x = np.linspace(-50, 50, 1000)
y = np.linspace(-50, 50, 1000)
print('x=', x)
print('y=', y)
# 网格点坐标矩阵, 1000 * 1000 的网格
x, y = np.meshgrid(x, y)
# z 的计算
z = x**2/3 + y**2/3
# 画出三维曲面, cmap颜色变化, rstride和cstride表示横纵方向步长
ax.plot_surface(x, y, z, rstride=100, cstride=100,
```

```
    cmap='Greens', edgecolor='k', lw=0.1)
plt.show()
```

三维曲面图如图 1.48 所示。

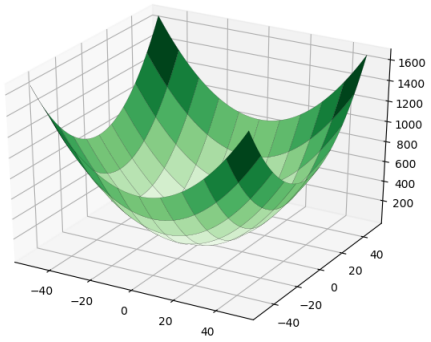


图 1.48: 三维曲面图

1.4.3 统计量计算

python 中的 *scipy* 包中包括计算像均值、标准差、偏度、峰度等统计量的函数，例如我们可以通过公式对 *scipy* 包中计算统计量的函数进行验证。

对于正态分布来说，偏度的公式为：

$$\text{skew} = E[((X-\mu)/\sigma)^3] = (E[X^3] - 3\mu\sigma^2 - \mu^3)/\sigma^3 \tag{1.1}$$

峰度的计算公式为：

$$\text{kurtosis} = \mu^4/\sigma^4 - 3 = (1/n\sum_{i=1}^{n}(x_i - \overline{x}))^4/(1/n\sum_{i=1}^{n}(x_i - \overline{x})^2)^2 - 3 \tag{1.2}$$

验证统计量的具体实现代码为：

```
import numpy as np
from scipy import stats
if __name__ == '__main__':
    x = np.random.randn(10000)
    print(x)
    print(x.shape)
    n = x.shape[0]    # 样本个数
    # 手动计算
    m = 0
    m2 = 0
    m3 = 0
    m4 = 0
    for t in x:
        m += t
```

```
        m2 += t * t
        m3 += t ** 3
        m4 += t ** 4
    m /= n
    m2 /= n
    m3 /= n
    m4 /= n
    mu = m
    sigma = np.sqrt(m2-mu*mu)
    skew = (m3-3*mu*m2+2*mu**3)/sigma**3
    kurtosis = (m4-4*mu*m3+6*mu*mu*m2-4*mu**3*mu+mu**4)/sigma**4-3
```

将手动计算出的统计量值输出，然后输出系统函数计算出的值，进行对比。

```
    print('手动计算均值、标准差、偏度、峰度：', mu, sigma, skew, kurtosis)
    # 使用系统函数验证
    mu = np.mean(x, axis=0)
    sigma = np.std(x, axis=0)
    skew = stats.skew(x)
    kurtosis = stats.kurtosis(x)
    print('函数库计算均值、标准差、偏度、峰度：', mu, sigma, skew, kurtosis)
```

图 1.49 的运行结果表明了函数库计算的统计量与公式计算的是一致的。

```
手动计算均值、标准差、偏度、峰度： -0.005133089152609225 1.0002285041313592 0.013795630464537566 0.0037675656912838917
函数库计算均值、标准差、偏度、峰度： -0.005133089152609214 1.0002285041313619 0.013795630464537342 0.0037675656912519173
```

图 1.49: 计算结果

1.4.4 高斯分布的绘制

这一小节我们通过比较不同方差下的差异，来了解高斯分布的 3D 绘图。

首先，给定四个不同的方差矩阵如下：

$$\begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix} \quad \begin{bmatrix}2 & 0\\ 0 & 2\end{bmatrix} \quad \begin{bmatrix}3 & 0\\ 0 & 5\end{bmatrix} \quad \begin{bmatrix}3 & 2\\ 2 & 6\end{bmatrix}$$

我们的任务是绘制这四种不同方差矩阵下的高斯分布。其中二维分布的每一维数据均值都是 0，即 $\mu 1$=0、μ 2=0。在这四个方差矩阵中，第一个矩阵表示是标准正态分布；第二个表示比标准正态分布“胖”一点；第三个表示“椭圆”形式的正态分布、第四个表示“斜”椭圆正态分布。

绘制出的高斯分布图形如图 1.50。

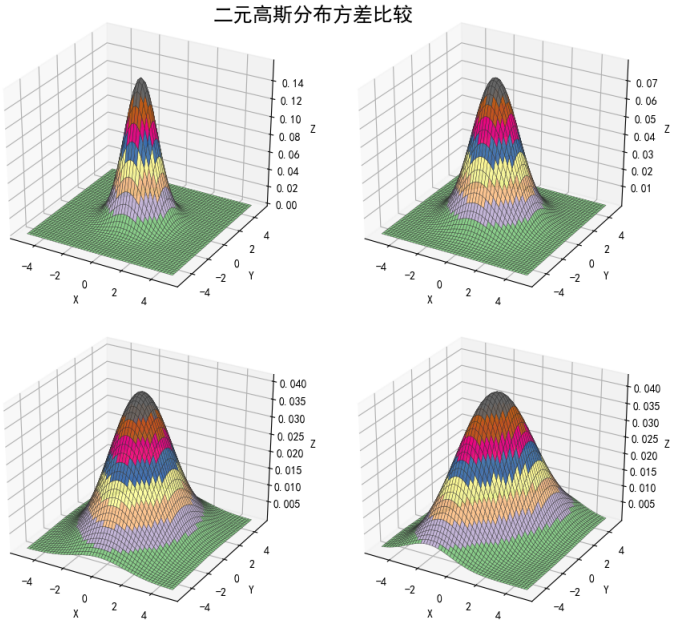


图 1.50: 二元高斯分布图比较

具体实现代码如下：

首先导入画图和计算所需要的模块，对 x1 和 x2 赋值后进行堆叠，对 sigma 进行赋值。

```
import numpy as np
from scipy import stats
import matplotlib as mpl
import matplotlib.pyplot as plt
from matplotlib import cm
from mpl_toolkits.mplot3d import Axes3D
if __name__ == '__main__':
    # -5到5之间的51个数
    x1, x2 = np.mgrid[-5:5:51j, -5:5:51j]
```

```
    # 将 x1 ， x2 堆叠
    x = np.stack((x1, x2), axis=2)
    print('x1=\n', x1)
    print('x2=\n', x2)
    print('x=\n', x)
    mpl.rcParams['axes.unicode_minus'] = False
    mpl.rcParams['font.sans-serif'] = 'SimHei'
    plt.figure(figsize=(9, 8), facecolor='w')
    # 四个 sigma 的取值分别为：
    sigma = (np.identity(2), np.diag((2,2)),
        np.diag((3,5)), np.array(((3,2), (2,6))))
```

用 for 循环依次遍历四个方差矩阵，构建多元正态分布函数，并计算概率密度函数，最后将密度函数放在三维坐标图中，将不同方差对应的四张子图以图形的方式进行输出，以便比较不同方差的高斯分布。

```
    for i in np.arange(4):
        ax = plt.subplot(2, 2, i+1, projection='3d')
        # 多元正态分布函数， mu1 mu2 取值都为 0 ，sigma 取值不同
        norm = stats.multivariate_normal((0, 0), sigma[i])
        # 将 x 放入某个正态分布中，计算这个分布的概率密度函数
        y = norm.pdf(x)
        ax.plot_surface(x1, x2, y, cmap=cm.Accent,
            rstride=1, cstride=1, alpha=0.9,
            lw=0.3, edgecolor='#303030')
        ax.set_xlabel('X')
        ax.set_ylabel('Y')
        ax.set_zlabel('Z')
    plt.suptitle('二元高斯分布方差比较', fontsize=18)
    plt.tight_layout(1.5)
    plt.show()
```

1.4.5 阶乘和 Gamma 函数

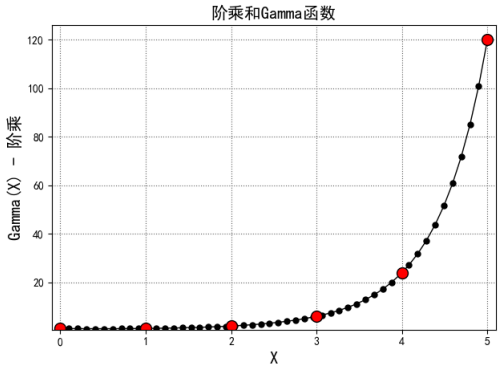


图 1.51: 阶乘和 *Gamma* 函数

在 *scipy* 中存在 *Gamma* 函数，它实际上是阶乘的推广。*Gamma* 函数常用在主题模型的 LDA 算法，在这个算法的迪利克雷分布中，系数有 Gamma 分布。通过 Gamma 计算出来的阶乘与使用公式计算出来的实际上是一致的。如图 1.51，红色点是使用公式计算出来整数的阶乘，黑色点是使用 *Gamma* 函数计算出来的，它们是吻合的。

具体实现代码如下：

```
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
from scipy.special import gamma
from scipy.special import factorial
mpl.rcParams['axes.unicode_minus'] = False
mpl.rcParams['font.sans-serif'] = 'SimHei'
if __name__ == '__main__':
    N = 5
    x = np.linspace(0, N, 50)
    y = gamma(x+1)
    plt.figure(facecolor='w')
    plt.plot(x, y, 'k-', x, y, 'ko', linewidth=1,
        markersize=5, mec='k')
    z = np.arange(0, N+1)
    f = factorial(z, exact=True) #阶乘
    plt.plot(z, f, 'ro', markersize=10, markeredgecolor='k')
    plt.grid(b=True, ls=':', color='#606060')
    plt.xlim(-0.1,N+0.1)
    plt.ylim(0.5, np.max(y)*1.05)
    plt.xlabel('X', fontsize=15)
    plt.ylabel('Gamma(X) - 阶乘', fontsize=15)
    plt.title('阶乘和Gamma函数', fontsize=15)
    plt.show()
```

1.4.6 验证 Pearson 相关系数

Pearson 相关系数时经常用来做数据分析的工具，一般来说进行数据分析前首先需要对数据做一个基本的统计，然后对数据做进一步挖掘。

Pearson 相关系数公式为：

$$\rho_{xy} = (Cov(X,Y))/\sqrt{(Var(X)Var(Y))} \tag{1.3}$$

从公式可以看出，$|\rho_{XY}| \leq 1$。当且仅当 X 和 Y 有线性关系时，等号成立。

python 的 *scipy* 包中含有计算 Pearson 相关系数的函数，在这里我们也可以通过公式手动计算相关系数，并与 *scipy* 中的函数计算的相关系数进行对比，具体实现代码如下：

```
# 公式计算 Pearson 相关系数
def calc_pearson(x, y):
    std1 = np.std(x)
    std2 = np.std(y)
    cov = np.cov(x, y, bias=True)[0,1]
    return cov/(std1* td2)
# 函数计算和公式计算对比
def intro():
    N = 10
    x = np.random.rand(N)
    y = 2*x+np.random.randn(N)*0.1
    print(x)
    print(y)
    print('系统计算：', stats.pearsonr(x, y)[0])
    print('手动计算：', calc_pearson(x, y))
```

其输出结果如图 1.52，我们可以发现计算出的相关系数基本是一致的。

```
系统计算： 0.9838743911427421
手动计算： 0.9838743911427422
```

图 1.52: 相关系数计算结果

这里可以通过 python 来验证 X 与 Y 的相关性。首先当 X 与 Y 是一次函数关系时，也就是使 X 随机取值，Y 取零，但是对 Y 加入轻微的噪声，同时使 X 与 Y 进行旋转，每次旋转角度为 45，并通过可视化可以观察 X 与 Y 相关系数的变化。结果可以发现对数据旋转并没有改变数据和它们之间的几何关系，相对位置是不变的，但是对于数据整体来说是发生了变化。当数据整体发生变化时，统计量就会变化。例如 PCA，如果对比原数据的 K 均值聚类和对数据进行主成分分析后的 K 均值聚类，最终得到的结果是不同的。具体实现代码如下：

首先，对 X 与 Y 进行随机取值，计算 Pearson 相关系数，然后对 X 和 Y 进行旋转处理。

```
if __name__ == '__main__':
    np.random.seed(0)
    N = 10000
    tip = '一次函数关系'
```

```
    # x 取随机数， y 全部取 0
    x = np.random.rand(N)
    # 加入噪声 np.random.randn(N)*0.001
    y = np.zeros(N) + np.random.randn(N)*0.001
    pearson(x, y, tip)
# 对 x、y 做旋转
def rotate(x, y, theta=45):
    data = np.vstack((x, y))
    mu = np.mean(data, axis=1)
    mu = mu.reshape((-1, 1))
    data -= mu
    theta *= (np.pi/180)
    c = np.cos(theta)
    s = np.sin(theta)
    m = np.array(((c, -s), (s, c)))
    return m.dot(data)+mu
```

求旋转后的 Pearson 相关系数，并将数据分布和相关系数进行可视化的输出，以便对比。

```
# 求 x、y 的相关系数
def pearson(x, y, tip):
    clrs = list('rgbmyc')
    plt.figure(figsize=(8, 6), facecolor='w')
    for i, theta in enumerate(np.linspace(0, 90, 6)):
        xr, yr = rotate(x, y, theta)
        p = stats.pearsonr(xr, yr)[0]
        print('旋转角度：', theta, 'Pearson相关系数：', p)
        str = '相关系数：%.3f' % p
        plt.scatter(xr, yr, s=30, alpha=0.8,
            linewidths=0.4, c=clrs[i], marker='o',
            label=str, edgecolors='k')
    plt.legend(loc='upper left', shadow=True)
    plt.xlabel('X')
    plt.ylabel('Y')
    plt.title('Pearson相关系数与数据分布：%s' % tip, fontsize=16)
    plt.grid(b=True, ls=':', color='#606060')
    plt.show()
```

如果对数据进行改变，使 X 与 Y 是二次函数关系。这时候取 X 为（-1，1）区间上的 101 个数字，Y 取 X 的平方，此时对 Y 不加入噪声。

因此相应的代码实现中只需要改变主函数中的 X 与 Y 的设置即可。

具体的修改代码为：

```
tip = u'二次函数关系'
x = np.linspace(-1, 1, 101)
y = x**2
```

X 与 Y 是一次函数关系时，数据分布和相关系数如图 1.53。我们可以观察到水平线和竖直线表示 X 与 Y 不相关，而中间倾斜的线 X 与 Y 的相关系数为 1，此时 X 与 Y 是完全正相关的。X 与 Y 是二次函数关系时，数据分布和相关系数如图 1.54。首先观察图 1.54 中红色的线，这条线表示了 X 与 Y 的关系是：$Y = X^2$。在 (-1，0) 区间上，随着 X 增大，Y 减小；在 (0，1) 区间上，随着 X 增大，Y 也增大。虽然在两个区间中我们分别可以看到 X 与 Y 有很好的相关关系，但是当代入公式中时，相关系数为零。因此当相关系数为零时，X 与 Y 可能没有相关性。

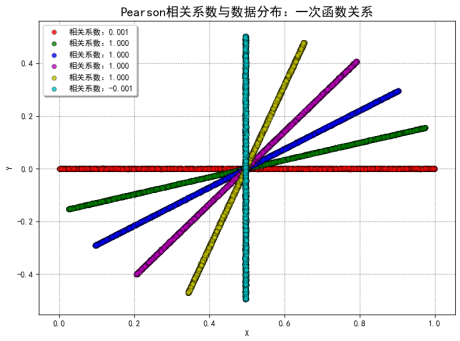


图 1.53: 一次函数关系的相关系数

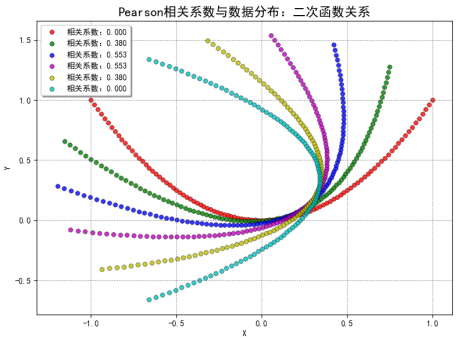


图 1.54: 二次函数关系的相关系数

我们也可以通过经典的鸢尾花数据集（iris.data）来计算一下该数据集的 Pearson 相关系数。

鸢尾花数据集中一共包含 150 个样本，前四列数据分别表示花萼的长宽和花瓣的长款，最后一列数据表示鸢尾花的三种类别：山鸢尾 (Setosa)、杂色鸢尾 (Versicolour)、维吉尼亚鸢尾 (Virginica)。

在对鸢尾花数据集求 Pearson 相关系数前需要先将最后一列数据进行重新编码，因为最后一列是字符串格式的数据。变换后最后一列就变成了采用 0、1 等进行分类的形式。最终求出来的 Pearson 相关系数是一个五维的相关系数矩阵，这个矩阵是对称的，并且对角线相关系数为 1。如图 1.55。

	0	1	2	3	4
0	1.000000	-0.109369	0.871754	0.817954	0.782561
1	-0.109369	1.000000	-0.420516	-0.356544	-0.419446
2	0.871754	-0.420516	1.000000	0.962757	0.949043
3	0.817954	-0.356544	0.962757	1.000000	0.956464
4	0.782561	-0.419446	0.949043	0.956464	1.000000

图 1.55: 鸢尾花数据集相关系数

具体实现代码如下：

```
data = pd.read_csv('iris.data', header=None)
data[4] = pd.Categorical(data[4]).codes
print(data.corr())
```

通过观察相关系数矩阵可以发现，鸢尾花的类别和花萼的长宽相关性不大，但是和花瓣的长宽相关性比较大。虽然我们并不知道花萼的长宽对于分类的有效性，但是由于花瓣的长宽与类别

的相关性相对较高，因此有理由认为花瓣的长宽对于分类起到了重要作用。

1.4.7 奇异值分解 SVD

奇异值分解可以看作对称方阵在任意矩阵上的推广。假设 A 是一个 $m \times n$ 的实矩阵，则存在一个分解使得：

$$A_{m\times n} = U_{m\times m}\epsilon_{m\times n}V^T_{n\times n} \tag{1.4}$$

其中 ϵ 对角线上的元素便是 A 的奇异值，通常将奇异值由大到小排列。在 python 的 *numpy* 包里面提供了奇异值分解的函数。

一般来说，如果对图像做奇异值分解后可以通过线性相乘相加进行恢复，随着奇异值个数的增加，恢复的图像越接近原始图像。在后面卷积神经网络中，一般来说随着权重个数的增加，得到的结果与原始图像就非常接近。这里的权重实际上相当于奇异值。并且在卷积神经网络中对图像的压缩处理也类似奇异值分解。但是奇异值分解做的是线性运算，而在卷积神经网络中做的是非线性的运算。因此 SVD 的其中一个目的是近似图像。

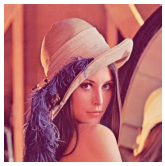


图 1.56: Lena 原始图像

图 1.57: 前 12 个奇异值的 Lena 图像

例如我们可以对 Lena 的图像进行奇异值分解再恢复，原始图像为图 1.56。恢复取前 12 个奇异值的结果如图 1.57。观察图像结果，可以发现奇异值个数越多，图像越接近原始图像。

具体实现代码如下：

首先导入计算、绘图等模块，然后计算奇异值并提取左右特征向量。

```
import numpy as np
import os
from PIL import Image
import matplotlib.pyplot as plt
```

```
import matplotlib as mpl
# 奇异值、左特征向量、右特征向量
def restore1(sigma, u, v, K):
    m = len(u)
    n = len(v[0])
    a = np.zeros((m, n))
    for k in range(K):
        uk = u[:, k].reshape(m, 1)
        vk = v[k].reshape(1, n)
        # 求 A 的近似值
        a += sigma[k]*np.dot(uk, vk)
    # 对 A 的值中小于 0 取 0 ，大于 255 取 255
    a = a.clip(0, 255)
    # 把 A 的值变成无符号的 8 位的整数
    return np.rint(a).astype('uint8')
```

将图像转换成 *np.array* 格式，提取三个通道并进行奇异值分解，取前 k 个奇异值进行通道恢复并堆叠成图像，显示取前 12 个值的结果。

```
if __name__ == "__main__":
    A = Image.open(".\lena.png", 'r')
    print(A)
    output_path = r'.\SVD_Output'
    if not os.path.exists(output_path):
        os.mkdir(output_path)
    # 将图像转换成 np.array 格式
    a = np.array(A)
    print('type(a)=', type(a))
    print('原始图片大小：', a.shape)
    # 将图像的红色、绿色和蓝色通道分别提取出来进行奇异值分解
    # 其中 u 和 v 都是正交矩阵，但是 sigma 可以选择前若干个奇异值
    u_r, sigma_r, v_r = np.linalg.svd(a[:, :, 0])
    u_g, sigma_g, v_g = np.linalg.svd(a[:, :, 1])
    u_b, sigma_b, v_b = np.linalg.svd(a[:, :, 2])
    plt.figure(figsize=(8, 8), facecolor='w')
    mpl.rcParams['font.sans-serif'] = ['simHei']
    mpl.rcParams['axes.unicode_minus'] = False
    K = 50
    for k in range(1, K+1):
        print(k)
        # 取前k个值恢复成R、G、B三个通道，堆叠恢复成图像并保存
        R = restore1(sigma_r, u_r, v_r, k)
        G = restore1(sigma_g, u_g, v_g, k)
        B = restore1(sigma_b, u_b, v_b, k)
        I = np.stack((R, G, B), axis=2)
        Image.fromarray(I).save('%s\\svd_%d.png' % (output_path, k))
        # 显示取前 12 个奇异值输出结果
```

```
        if k <= 12:
            plt.subplot(3, 4, k)
            plt.imshow(I)
            plt.axis('off')
            plt.title('奇异值个数：%d' % k)
    plt.suptitle('SVD与图像分解', fontsize=20)
    plt.tight_layout(0.3, rect=(0, 0, 1, 0.92))
    plt.show()
```

第 2 章　机器学习

2.1　引言

2.1.1　研究内容

机器学习研究的主要是借助计算机从数据中产生“模型”(model) 从而改善系统自身的性能的算法，即研究关于“学习算法”(learning algorithm) 的学问。故而基于规则的、类似专家系统、利用大量的 if-else 逻辑结构的算法不是机器学习。

2.1.2　基本术语

收集到的全部记录称为“数据集”，每条记录称为一个“样本”。每个样本的表现或性质称为“属性或特征”，属性个数即样本“维数”。属性上的取值称为“属性值”，属性张成的空间称为“特征空间”，特征提取是机器学习模型好坏的关键。

从数据中学得模型的过程称为“学习”或“训练”，训练过程通过对“训练集”执行某个学习算法来完成，其适用于新样本的能力称为“泛化”能力。学得模型反映出数据的某种潜在的规律或样本空间的特性，亦称“假设”。训练样本结果的信息称为“标记”，其集合称为“输出空间”。空间过程即样本空间（输入空间）—假设空间（属性空间）—版本空间（输出空间）。模型评估与选择中用于评估测试的数据集常称为“验证集”。学得模型后使用模型进行预测的过程称为“测试”，对应样本为“测试集”。

2.1.3　目的分类

研究目的：找到满足其归纳偏好的最优的“学习算法”。若追求极大的训练集准确率，比如 MLP，若追求模型的泛化能力可考虑 SVM。

任务分类：当训练数据有标记则为监督学习，即输入空间到输出空间的映射。其中预测的是离散值为分类，预测的是连续值为回归。当无标记信息时为无监督学习，而聚类则是其代表。

2.1.4　模型选择

尽可能得到泛化误差小的模型，而训练误差存在不可避免的过拟合现象，故而常用实验评估方法、性能度量来比较检验和进行选择。

评估方法

评估方法：即测试集选择，包括留出法、交叉验证、自助法。

1. 留出法（Hold-out）：直接将数据集 D 划分为两个互斥的集合, 一个集合作为训练集 S, 另一个作为测试集 T。训练和测试集的划分要尽可能保持数据分布的一致性，避免因数据划分过程引入额外的偏差而对最终结果产生影响，类似 “分层抽样”。单次使用留出法得到的估计结果往往不够稳定可靠，在使用留出法时，一般要采用若干次随机划分、重复进行实验评估后取平均值作为留出法的评估结果。对于 S 和 T 样本量导致的训练和测试保真性的平衡没有完美解决方案，常见方法是将大概 2/3∼4/5 的样本用于训练。

2. 交叉验证法（Cross-validation）:将数据集 D 分为 k 个大小相似的互斥子集 $\mathrm{D}_1,\cdots,\mathrm{D}_k$, 每个子集 D_i 尽可保持数据分布的一致性，即从 D 中通过分层采样得到。每次用 $k-1$ 个子集的并集作为训练集, 余下的那 1 个子集作测试集；从而可进行 k 次训练和测试，返回的是这 k 个测试结果的均值。为缓解样本划分不同引入的差别，k 折交叉验证通常要随机使用不同的划分重复 p 次, 最终的评估结果是这 p 次 k 折交叉验证结果的均值。交叉验证法结果的稳定性取决于 k 的取值，k 最常见的取值为 10。其中一个特例为留一法（Leave-One-Out），m 个样本只有唯一的方式划分为 m 个子集，每个子集包含一个样本，评估结果较准确，但数据量较大时计算有负担。

3. 自助法 (Bootstrap)：从数据集 D 中有放回的随机抽 m 个样本得到 D'，样本在 m 次采样中始终没被抽中的概率取极限为 $1/e$, 即大约 1/3 用于测试。自助法在数据集较小、难以有效划分训练和测试集时很有用；但自助法产生的数据集改变了初始数据集的分布，引入估计偏差，在初始数据量足够时留出法和交叉验证法更常用。

性能度量

性能度量：评估模型学习的误差或损失。

1. 回归任务：均方误差 MSE、平均绝对误差 MAE、确定系数 R^2 等。

$$\mathrm{MSE}=\frac{1}{n}\sum_{i=1}^{n}\left(\hat{y}_i-y_i\right)^2 \tag{2.1}$$

$$\mathrm{MAE}=\frac{1}{n}\sum_{i=1}^{n}\mid\hat{y}_i-y_i\mid \tag{2.2}$$

其中：$\hat{y}_i=\hat{f}(\boldsymbol{x}_i)$.

$$\mathrm{R}^2=\frac{\mathrm{SSR}}{\mathrm{SST}}=\frac{\mathrm{SST}-\mathrm{SSE}}{\mathrm{SST}}=1-\frac{\mathrm{SSE}}{\mathrm{SST}} \tag{2.3}$$

其中：SST 为离差平方和, 反应数据波动性的大小, 即 $\mathrm{SST}=\sum_{i=1}^{n}\left(y_i-\overline{y}\right)^2, \overline{y}=\sum_{i=1}^{n}y_i/n$; SSE 为拟合数据和原始数据对应点的误差平方和，即 $\mathrm{SSE}=\sum_{i=1}^{n}\left(y_i-\hat{y}_i\right)^2$；SSR 为回归平方和，即 $\mathrm{SSR}=\sum_{i=1}^{n}\left(\hat{y}_i-\overline{y}\right)^2$.

2. 分类任务: 错误率与精度、混淆矩阵、PC 曲线平衡点、ROC 曲线等。

（1）错误率与精度

错误率是分类错误的样本数占样本总数的比例，精度则是分类正确的样本数占样本总数的比例：

$$\mathrm{E}(\hat{f};\mathrm{D}) = \frac{1}{n}\sum_{i=1}^{n}\mathrm{I}\left(\hat{f}(\boldsymbol{x}_i) \neq y_i\right) \tag{2.4}$$

（2）混淆矩阵

混淆矩阵特别用于监督学习，在无监督学习一般叫做匹配矩阵。在图像精度评价中，主要用于比较分类结果和实际测得值，可以把分类结果的精度显示在一个混淆矩阵里面。混淆矩阵是通过将每个实测像元的位置和分类与分类图像中的相应位置和分类相比较计算的。对于二分类问题，可将其真实类别与学习器预测类别对应的样例数划分为 TP 真正例 (true positive) 、FP 假正例 (false positive) 、TN 真反倒 (true negative)、FN 假反例 (false negative) 四种情形，显然有 TP+FP+TN+FN= 样例总数。

表 2.1: 混淆矩阵

真实情况	预测结果（正）	预测结果（反）
正例	TP（真正例）	FN（假反例）
反例	FP（假正例）	TN（真反例）

（3）P-R 曲线

查准率 P 为正例中有多少为真正例，查全率 R 为正例被正确划分的比例。查准率和查全率是一对矛盾的度量。一般来说，查准率高时，查全率往往偏低。查准率-查全率 “P-R 曲线” 交叉，则可以通过平衡点两者相等时进行比较，取值大则优，应用中 F1 度量更常用。

$$\mathrm{P} = \frac{\mathrm{TP}}{\mathrm{TP}+\mathrm{FP}}\ \mathrm{R} = \frac{\mathrm{TP}}{\mathrm{TP}+\mathrm{FN}} \tag{2.5}$$

$$\mathrm{F1} = \frac{2\times\mathrm{P}\times\mathrm{R}}{\mathrm{P}+\mathrm{R}} = \frac{2\times\mathrm{TP}}{\text{样例总数}+\mathrm{TP}-\mathrm{TN}}\ . \tag{2.6}$$

（4）ROC 曲线

ROC 曲线：主要分析工具是一个画在二维平面上的曲线。平面的横坐标是 false positive rate(FPR)，纵坐标是 true positive rate(TPR)。

$$\mathrm{FPR} = \frac{\mathrm{FP}}{\mathrm{TN}+\mathrm{FP}} \tag{2.7}$$

$$\mathrm{TPR} = \frac{\mathrm{TP}}{\mathrm{TP}+\mathrm{FN}} \tag{2.8}$$

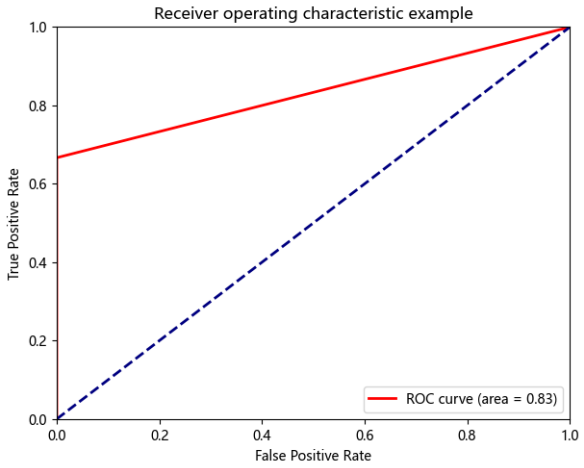


图 2.1: ROC 曲线图

(0,0) 和 (1,1) 连线形成的 ROC 曲线实际上代表的是一个随机分类器。ROC 曲线越靠近左上角，模型查全率越高，最靠近左上角的 ROC 曲线上的点是分类错误最少的最好阈值，其假正例和假反例总数最少。将各个学习器的 ROC 曲线绘制在同一坐标中，直观比较，越靠近左上角的 ROC 曲线代表的学习器准确性越高。AUC 的值就是处于 ROC 曲线下方的那部分面积的大小。通常，AUC 的值介于 0.5 到 1.0 之间，较大的 AUC 代表了较好的模型表现。AUC 同时考虑了学习器对于正例和负例的分类能力，在样本不平衡的情况下，依然能对分类器做出合理评价。

2.2 梯度算法

2.2.1 梯度下降 GD

1. 梯度下降（Gradient descent）

梯度下降是一个用来求函数最小值的算法。有一个可微分的函数（山）目标是找到这个函数的最小值（山底）。最快的下山的方式就是就是找到给定点的梯度（当前位置最陡峭的方向），然后朝着梯度相反的方向，就能到达局部的最小值。不论是在线性回归还是 Logistic 回归中，它的主要目的是通过迭代找到目标函数的最小值，或者收敛到最小值。

在单变量的函数中，梯度其实就是函数的微分，代表着函数在某个给定点的切线的斜率。在多变量函数中，梯度是一个向量，向量有方向，梯度的方向就指出了函数在给定点的上升最快的方向。

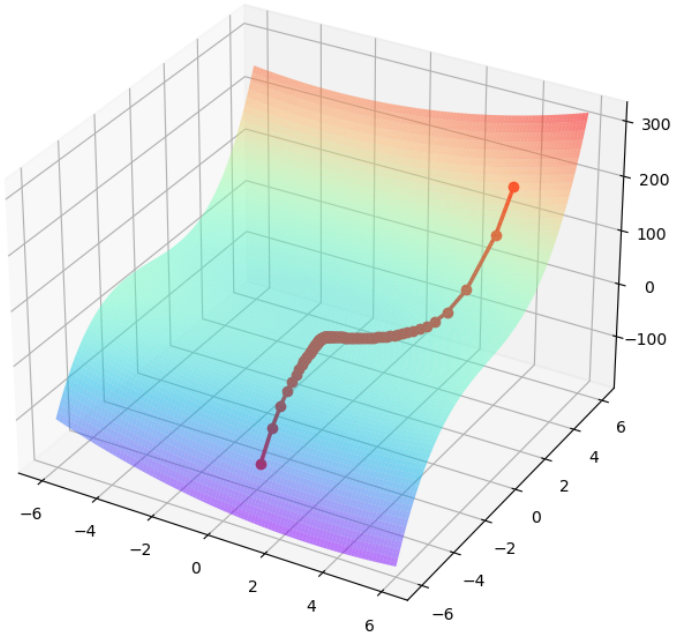


图 2.2: 梯度下降图

2. 数学公式:

$$\theta_j = \theta_{j-1} - \alpha \nabla J(\theta_{j-1}) \tag{2.9}$$

上式中：$J(\theta)$ 是关于 θ 的一个函数，θ_{j-1} 为当前的函数值（所处的位置），要从这个点走到 $J(\theta)$ 的最小值点（山底）。首先我们先确定前进的方向 $-\nabla J(\theta_{j-1})$(梯度的反向)，然后走一段距离的步长 α，走完这个段步长就到达了 θ_j 更新后的函数值点。

α 在梯度下降算法中被称作为学习率或者步长，意味着我们可以通过 α 来控制每一步走的距离，以保证不要步子跨的太大错过了最低点，同时也要保证不能太小可能导致迟迟走不到最低点，故而 α 的选择在梯度下降法中往往是很重要的。梯度的方向实际就是函数在此点上升最快的方向，而我们需要朝着下降最快的方向走，自然就是负的梯度的方向；那么如果是上坡就不需要，即梯度上升算法。初始位置：线性回归初值都为 0，卷积神经网络截断的高斯分布则不一定（不同模型不同，可随机但卷积不能全为 0）。

上图为计算函数 $z = x^2 + y^2 + y^3 + xy$ 从初始位置 x=4，y=5（右上角），迭代次数为 122，学习率为 0.01 的梯度下降 3D 曲线图，并用红点记录了每次梯度下降更新后点的坐标。

2.2.2 批量梯度下降 BGD

批量梯度下降法（Batch gradient descent）是指在每一次迭代时使用所有样本来进行梯度的更新。在梯度下降的每一步中，用所有的训练样本数据进行求和运算，在梯度下降计算微积分时，每一个样本都计算会导致运算速度比较慢。例如：我们使用只含有一个特征的线性回归来展开，此时线性回归的假设函数为：$h_{\boldsymbol{\theta}}(\boldsymbol{x}_i) = \boldsymbol{\theta}^T \boldsymbol{x}_i$，其中 i=1,2,...,n 表示样本数。

代价函数（目标函数）：

$$J(\boldsymbol{\theta}) = \frac{1}{2n}\sum_{i=1}^{n}\left(h_{\boldsymbol{\theta}}(\boldsymbol{x}_i) - y_i\right)^2 \tag{2.10}$$

梯度（对目标函数求偏导）：

$$\frac{\Delta J(\boldsymbol{\theta})}{\Delta \theta_j} = \frac{1}{n}\sum_{i=1}^{n}\left(h_{\boldsymbol{\theta}}(\boldsymbol{x}_i) - y_i\right)x_{ij} \tag{2.11}$$

其中 i=1,2,...,n 表示样本数，j=0,1 表示特征数，这使用了偏置项 x_{i0}=1。

每次迭代对参数进行更新：

$$\theta_j = \theta_j - \alpha\frac{1}{n}\sum_{i=1}^{n}\left(h_{\boldsymbol{\theta}}(\boldsymbol{x}_i) - y_i\right)x_{ij} \tag{2.12}$$

优点：（1）一次迭代是对所有样本进行计算，此时利用矩阵进行操作，实现了并行。（2）由全数据集确定的方向能够更好地代表样本总体，从而更准确地朝向极值所在的方向。当目标函数为凸函数时，BGD 一定能够得到全局最优。

缺点：当样本数目 n 很大时，每迭代一步都需要对所有样本计算，训练过程会很慢。

2.2.3 随机梯度下降 SGD

随机梯度下降法（Stochastic gradient descent）不同于批量梯度下降，随机梯度下降是每次迭代使用一个样本来对参数进行更新，使得训练速度加快。以下 i=1,2,...,n 表示样本数，j=0,1 表示特征数，偏置项 x_{i0}=1。

对于一个样本的代价函数：

$$J_i(\boldsymbol{\theta}) = \frac{1}{2}\left(h_{\boldsymbol{\theta}}(\boldsymbol{x}_i) - y_i\right)^2 \tag{2.13}$$

梯度：

$$\frac{\Delta J_i(\boldsymbol{\theta})}{\Delta \theta_j} = \left(h_{\boldsymbol{\theta}}(\boldsymbol{x}_i) - y_i\right)x_{ij} \tag{2.14}$$

参数更新：

$$\theta_j = \theta_j - \alpha\left(h_{\boldsymbol{\theta}}(\boldsymbol{x}_i) - y_i\right)x_{ij} \tag{2.15}$$

优点：由于不是在全部训练数据上的损失函数，而是在每轮迭代中，随机优化某一条训练数据上的损失函数，这样每一轮参数的更新速度大大加快。

缺点：（1）准确度下降。由于即使在目标函数为强凸函数的情况下，SGD 仍旧无法做到线性收敛。（2）可能会收敛到局部最优，由于单个样本并不能代表全体样本的趋势。（3）不易于并行实现。

2.2.4 小批量梯度下降 MBGD

小批量梯度下降（Mini-batch gradient descent）是对批量梯度下降以及随机梯度下降的一个折中办法。其思想是每次迭代使用 “batch_ size” 个样本来对参数进行更新。这里设 batch_size=t，其中 k=1,2,...,n 表示样本数，j=0,1 表示特征数，偏置项 x_{i0}=1。例如：n=1000，t=10，i=1,11,21,...,991。

参数更新：

$$\theta_j = \theta_j - \alpha \frac{1}{t} \sum_{k=i}^{i+t-1} \left(h_{\boldsymbol{\theta}} \left(\boldsymbol{x}_k \right) - y_k \right) x_{kj} \tag{2.16}$$

优点：（1）通过矩阵运算，每次在一个 batch 上优化神经网络参数并不会比单个数据慢太多。（2）每次使用一个 batch 可以大大减小收敛所需要的迭代次数，同时可以使收敛到的结果更加接近梯度下降的效果（比如样本数为 30 万，设置 batch_ size=100 时，需要迭代 3000 次，远小于 SGD 的 30 万次)。（3）可实现并行化。

缺点：batch_size 的不当选择可能会带来很多其他问题。

2.2.5 Adam 算法

Adam 算法是对梯度下降算法的改进，其思想是，通过引入动量因子，运用滑动平均方法对随机梯度中的随机性以及后续可能会出现的模型不稳定性进行修正。

参数更新：

$$\theta_j = \theta_{j-1} - \alpha \nabla J(\theta_{j-1}) \tag{2.17}$$

引入动量因子，并进行参数修正。

$$v_j = \beta_1 v_{j-1} + (1 - \beta_1) \cdot \nabla J(\theta_{j-1}) \tag{2.18}$$

$$\hat{v}_j = \frac{v_j}{1 - {\beta_1}^j} \tag{2.19}$$

进行均方根传递过程，并进行参数修正。

$$m_j = \beta_2 m_{j-1} + (1 - \beta_2) \nabla J(\theta_{j-1})^2 \tag{2.20}$$

$$\hat{m}_j = \frac{m_j}{1 - {\beta_2}^j} \tag{2.21}$$

最终形成 adam 算法的参数更新过程

$$\theta_j = \theta_{j-1} - \alpha \frac{\hat{v}_j}{\sqrt{\hat{m}_j}} \tag{2.22}$$

2.2.6 求解参数最优解

解析解：令偏导为 0，去掉极大值和鞍点。

数值解：运用梯度，即目标函数一阶导存在，若个别不存在可人为规定（如：令连续不可导的点斜率为 1)。

Smooth 函数：该类函数可导，用梯度下降法或者 Adam 求解最优值。

NON-Smooth 函数：该类函数不可导；Lasso 正则是各个参数绝对值之和，导致含有 Lasso 正则的目标函数是 NON-Smooth 函数，不平滑；梯度下降法等优化算法对它统统失效了，考虑求极值解法——坐标轴下降法。

优化器：如果输入数据集比较稀疏，SGD 等方法可能效果不好。因此对于稀疏数据集，应该使用某种自适应学习率的方法，且另一好处为不需要人为调整学习率，使用默认参数就可能获得最优值。

如果想使训练深层网络模型快速收敛或所构建的神经网络较为复杂，则应该使用 Adam 或其他自适应学习速率的方法，因为这些方法的实际效果更优。

2.2.7 案例

本例进行三种梯度下降算法的代码运行以及它们之间的特点比较。

首先生成伪随机数据，并用线性模型进行初步拟合和预测。

```
# 导入计算、回归和画图的模块
import numpy as np
from sklearn.linear_model import LinearRegression
import matplotlib.pyplot as plt
np.random.seed(46)  # 随机种子
# 创建伪数据
x = 2*np.random.rand(100,1)
y = 4 + 3 * x + np.random.randn(100,1)
# 最小二乘法得到的回归模型参数
X= np.c_[np.ones((100,1)),x] # 添加常数项特征
xnew=np.array([[0],[2]])     # 创建测试数据
Xnew=np.c_[np.ones((2,1)),xnew]
lin_reg=LinearRegression()   # 创建线性回归对象
lin_reg.fit(x,y)             # 拟合训练数据
print('最小二乘法模型参数：',lin_reg.intercept_,lin_reg.coef_)
                             # 输出拟合截距,斜率
lin_reg.predict(xnew)        # 对测试集进行预测
print('测试数据预测结果：',lin_reg.predict(xnew))  # 输出预测结果
```

我们可以得到最小二乘法估计回归方程参数的结果为 [4.08，2.84]，和实际情况相符，预测数据的结果为 [4.08，9.76]。

使用批量梯度下降算法来更新拟合得到参数值。首先要给定步长、迭代次数，并随机给出一个初始位置，然后利用更新方程进行迭代，得到优化的参数值。

```
# 批量梯度下降算法
alpha = 0.1              # 步长$\alpha$
n_iterations = 1000      # 迭代次数
m = 100                  # 样本数
theta = np.random.randn(2,1)   #初始位置
for iteration in range(n_iterations):
    gradients = 1/m * X.T.dot(X.dot(theta) - y)     # 梯度
    theta = theta - alpha * gradients               # 更新$\theta$
print('批量梯度下降算法模型参数：',theta)
```

我们自定义一个 BGD 函数，用来记录批量梯度下降法下参数更新的过程。运用一系列可视化调整图形格式的代码，得到的是前十次参数更新的回归模型。

```
theta_path_bgd= []                  # 自定义BGD的函数
def plot_gradient_descent(theta,alpha,theta_path = None):
    m = len(X)                      # 样本数
    plt.plot(X, y, "b.")            # 数据分布
    n_iterations = 1000             # 迭代次数
    for iteration in range(n_iterations):
        if iteration < 10:
           y_predict = Xnew.dot(theta)
           style = "r-"
           plt.plot(xnew, y_predict, style)      # 前10次迭代结果
        gradients = 2/m * X.T.dot(X.dot(theta)-y)# 梯度
        theta = theta - alpha * gradients        # 更新theta
        if theta_path is not None:
            theta_path.append(theta)             # 记录每次位置
        plt.xlabel("$x_1$",fontsize = 12)
        plt.axis([0,2,0,15])                 # 设置x轴和y轴的范围
        plt.title(r"$\alpha = {}$".format(alpha), fontsize = 16)
                                                  # 格式化标题
```

再次随机给出一个参数的初始值，预先设置整个图形的宽和高，然后分别作出步长为 0.02、0.1 和 0.5 的三个子图，然后输出图形。

```
theta = np.random.randn(2,1)
plt.figure(figsize=(10,6))  # 设置图形的宽为10，高为6
plt.ylabel("$y$", rotation=0, fontsize=18)
# 三个不同步长前10次迭代的子图
plt.subplot(131);plot_gradient_descent(theta, alpha=0.02)
plt.subplot(132);plot_gradient_descent(theta,alpha=0.1, theta_path=theta_path_
    bgd)
plt.subplot(133);plot_gradient_descent(theta,alpha=0.5)
plt.show()
```

我们得到使用批量梯度下降法计算的回归方程参数估计的结果为 [4.08，2.84]，同时根据如下不同的步长前 10 次迭代参数的结果可得：子图一 α=0.02 步长过小会导致一步步迭代到最优值的过程会比较长，不过结果会较为精确，需要注意的是，倘若迭代次数不够，就会导致最终结果往往不是最优解；子图三 α=0.5 步长过长虽然可以使得迭代过程缩短，但是对于精度丢失会比较大，并且最终解很有可能会在最有点两侧来回跳动，导致始终接近不了最优解。

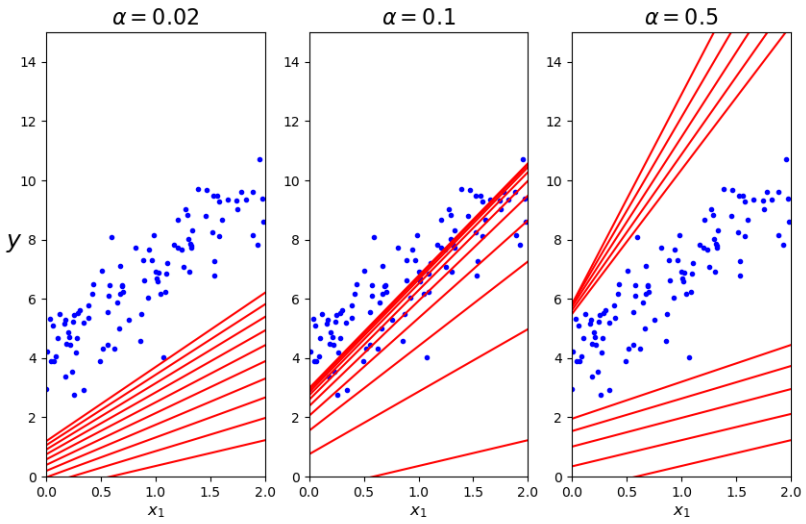


图 2.3: 批量梯度下降算法图

在随机梯度下降算法中，我们要额外指定循环次数的取值，说明整个样本要遍历循环多少次，因此在下面的更新循环语句中会嵌套两个循环，外层循环用于遍历整个样本集合，内层循环用于遍历样本集合内部的单个样本。并且只记录第一个 *epoch* 的前 20 次更新情况。

```
# 随机梯度下降算法
theta_path_sgd = [] # 每次更新参数值
m = len(X)          # 样本数
n_epochs = 50       # 循环次数
theta = np.random.rand(2,1)   # 初始位置
for epoch in range(n_epochs):
    for i in range(m):        # 每次对单个样本进行
        if epoch == 0 and i <20:  #记录前20个样本的参数更新情况
            y_predict = Xnew.dot(theta)
            style = "r-"
            plt.plot(xnew, y_predict, style)
        random_index = np.random.randint(m)   # 生成随机索引
        xi = X[random_index:random_index+1]   # 对应的随机伪数据
        yi = y[random_index:random_index+1]
        gradients = 2 * xi.T.dot(xi.dot(theta) - yi)  # 计算梯度
        alpha = 0.1         # 步长
        theta = theta - alpha * gradients   # 更新参数
        theta_path_sgd.append(theta)   # 记录每次更新的参数值
```

同批量梯度算法一样，设置坐标图的坐标轴名称和显示范围，然后把样本点和前 20 次的更新模型画在坐标图中，用图形把结果输出，可以直观地看到随机梯度下降法的更新过程。

```
 # 同上作图
plt.plot(x, y, "b.")
plt.xlabel("$x_1$",fontsize = 18)
```

```
plt.ylabel("$y$", rotation = 0, fontsize = 18)
plt.axis([0,2,0,15])
plt.show()
print('随机梯度下降算法模型参数:  ',theta)
```

我们得到使用随机梯度下降法计算的回归方程参数估计的结果为 [4.42，2.38]，样本点和更新过程如下图。可以看到相较于批量梯度下降的结果和真实值的差距较大，估计的准确度大大下降，这是由于随机梯度下降每次更新参数仅使用一个样本，自然远不如批量梯度下降法使用所有样本来的准确。

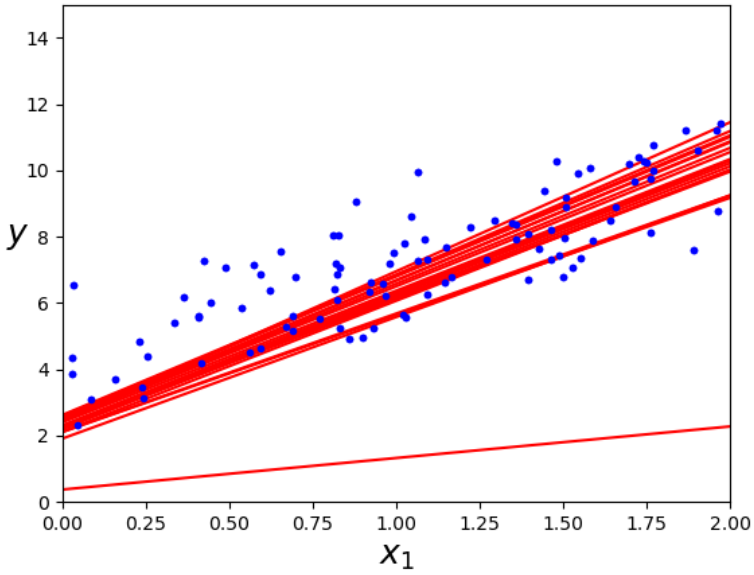


图 2.4: 随机梯度下降算法图

在小批量梯度下降算法中，要指定每次更新迭代所需的样本数，在每次迭代的循环中能够随机选取指定数量的样本。同样指定步长和随机初始位置后，记录更新结果。

```
# 小批量梯度下降算法
theta_path_mgd = []  # 每次更新参数值
n_iterations = 50    # 迭代次数
minibatch_size = 20  # 每次进行迭代的样本数
theta = np.random.randn(2,1)  # 初始位置
for epoch in range(n_iterations):
    shuffled_indices = np.random.permutation(m)# 打乱原数据顺序
    X_shuffled = X[shuffled_indices]   # 打乱后的伪数据
    y_shuffled = y[shuffled_indices]
    for i in range(0,m,minibatch_size):
        xi = X_shuffled[i:i+minibatch_size]   # 选择每次迭代的数据
        yi = y_shuffled[i:i+minibatch_size]
        gradients = 2/minibatch_size * xi.T.dot(xi.dot(theta)-yi)
```

```
                                                # 计算梯度
        alpha = 0.1    #步长
        theta = theta - alpha * gradients    # 更新参数
        theta_path_mgd.append(theta)         # 记录参数值
```

我们得到使用小批量梯度下降法计算的回归方程参数估计的结果为 [4.07，2.85]，准确性比随机梯度下降要好得多，是兼顾批量梯度下降法的训练速度慢，以及随机梯度下降法的准确性的方法。

接下来，将以上三种梯度下降算法更新参数的路径提取并记录下来，放在横坐标标为常数项、纵坐标为截距项的坐标图中，输出并对比。

```
theta_path_bgd = np.array(theta_path_bgd) # 提取三种算法的更新路径
theta_path_sgd = np.array(theta_path_sgd)
theta_path_mgd = np.array(theta_path_mgd)
plt.figure(figsize=(8,6))      # 分别作三种梯度下降参数更新的路径图
plt.plot(theta_path_sgd[:,0], theta_path_sgd[:,1], "r-o", linewidth=1, label="
    Stochastic")
plt.plot(theta_path_mgd[:,0], theta_path_mgd[:,1], "g-s", linewidth=2, label="
    Mini-batch")
plt.plot(theta_path_bgd[:,0], theta_path_bgd[:,1], "b-+", linewidth=3, label="
    Batch")
plt.legend(loc="upper left", fontsize=16)     # 图例位置左顶图
plt.xlabel(r"$\theta_0$", fontsize=20)        # 横坐标为常数项
plt.ylabel(r"$\theta_1$", fontsize=20,rotation=0)# 纵坐标为截距项
plt.axis([2.5, 4.5, 2.3, 3.9])
plt.show()
```

下图中为三种梯度下降算法每次迭代回归模型常数项和截距项的路径，通过直观地对比，可以很明确发现随机梯度下降算法的准确性误差较大，而小批量梯度下降算法是随机梯度下降算法和批量梯度下降算法的这种，是比较综合地考虑了准确性和计算速度的首选。

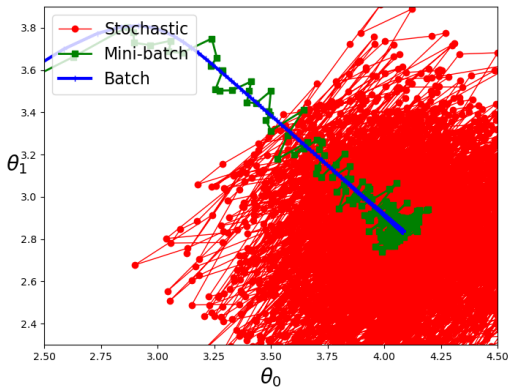


图 2.5: 小批量梯度下降算法图

2.3 回归算法

2.3.1 线性回归

回归分析是一种预测建模技术的方法，能估计两个或者多个变量之间的关系，揭示了因变量和自变量之间的显著关系和影响程度大小。可被用于预测、构建时间序列模型和寻找变量之间相关关系等方面。回归分析的类别主要由三个方面度量：独立变量的数量、独立变量的类型和回归线的形状驱动，可以对上面的参数进行组合甚至创造出新的回归。常见回归类型有：线性回归（Linear Regression）、岭回归（Ridge Regression）、套索回归（Lasso Regression）、弹性回归（ElasticNet Regression）、多项式回归（Polynomial Regression）和局部加权线性回归（Locally Weighted Linear Regression）。

线性回归目的是学得一个线性模型来准确地预测实值输出标记。线性模型形式简单，是一个通过对属性的线性组合来进行预测的函数，即 $f(\boldsymbol{x}) = h_{\boldsymbol{\theta}}(\boldsymbol{x}) = w_1x_1 + w_2x_2 + \ldots + w_dx_d + b$。给定 d 个属性的示例 $\boldsymbol{x} = (x_1, x_2, \ldots, x_d)$，其中 x_j 是 $\boldsymbol{x}$ 在第 j 个属性上的取值. 令 $\boldsymbol{w} = (w_1, w_2, \ldots, w_d)$，注意在本章中出现的 $(\boldsymbol{w}, b)$ 都等价于 $\boldsymbol{\theta}$。此处的向量形式可以表示为如下形式：

$$h_{\boldsymbol{\theta}}(\boldsymbol{x}) = \boldsymbol{w}^{\mathrm{T}}\boldsymbol{x} + b \tag{2.23}$$

损失函数是误差值之和，其求解可以借助正规方程、梯度下降等。对于一维属性，可采用 “最小二乘法” 基于均方误差最小化来进行模型求解。当输入属性只有一个时，给定数据集 D=$\{(\boldsymbol{x}_1, y_1), (\boldsymbol{x}_2, y_2), \ldots, (\boldsymbol{x}_n, y_n)\}$，其中 $\boldsymbol{x}_i = (x_{i1}, x_{i2}, \ldots, x_{id})$，$\boldsymbol{w} = (w_1, w_2, \ldots, w_d)$，其中 $i = 1, 2, \ldots, n$。求解 $\boldsymbol{w}$ 和 b 使得 $\mathrm{E}_{(\boldsymbol{w},b)} = \sum_{i=1}^{n} \left(y_i - \boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}_i - b\right)^2$ 最小化的过程：

$$\begin{aligned} \left(\hat{\boldsymbol{w}}, \hat{b}\right) &= \arg\min_{(\boldsymbol{w},b)} \sum_{i=1}^{n} \left(h_{\boldsymbol{\theta}}(\boldsymbol{x}_i) - y_i\right)^2 \\ &= \arg\min_{(\boldsymbol{w},b)} \sum_{i=1}^{n} \left(y_i - \boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}_i - b\right)^2 \end{aligned} \tag{2.24}$$

若 $d = 1$, 分别对 w 和 b 求导可得最优价的闭式解：

$$\hat{w} = \frac{\sum_{i=1}^{n} y_i \left(x_i - \bar{x}\right)}{\sum_{i=1}^{n} x_i^2 - \left(\sum_{i=1}^{n} x_i\right)^2} \tag{2.25}$$

$$\hat{b} = \frac{1}{n} \sum_{i=1}^{n} \left(y_i - \hat{w}x_i\right) \tag{2.26}$$

对于多元线性回归，我们把估计出来的 $\boldsymbol{w}$ 和 b 收入向量形式 $\boldsymbol{\theta} = (\boldsymbol{w}; b)$，把 $\boldsymbol{X}$ 表示为一个 n 行 $d+1$ 列的矩阵，其中每行对应于一个示例，该行前 d 个元素对应于示例的属性值，最后一个元素恒置为 1，$\boldsymbol{Y}$ 是一个 n 行 1 列的矩阵，即：

$$\boldsymbol{X} = \begin{pmatrix} x_{11} & x_{12} & \ldots & x_{1d} & 1 \\ x_{21} & x_{22} & \ldots & x_{2d} & 1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ x_{n1} & x_{n2} & \ldots & x_{nd} & 1 \end{pmatrix} = \begin{pmatrix} \boldsymbol{x}_1^{\mathrm{T}} & 1 \\ \boldsymbol{x}_2^{\mathrm{T}} & 1 \\ \vdots & \vdots \\ \boldsymbol{x}_n^{\mathrm{T}} & 1 \end{pmatrix}; \boldsymbol{Y} = \begin{pmatrix} y_1 \\ y_2 \\ \ldots \\ y_n \end{pmatrix} \tag{2.27}$$

最小二乘法优化目标函数为：

$$\hat{\boldsymbol{\theta}} = \arg\min_{\tilde{\theta}} (\boldsymbol{Y} - \boldsymbol{X}\boldsymbol{\theta})^{\mathrm{T}} (\boldsymbol{Y} - \boldsymbol{X}\boldsymbol{\theta}) \tag{2.28}$$

令 $E_{\boldsymbol{\theta}} = (\boldsymbol{Y} - \boldsymbol{X}\boldsymbol{\theta})^{\mathrm{T}} (\boldsymbol{Y} - \boldsymbol{X}\boldsymbol{\theta})$，对 $\boldsymbol{\theta}$ 求导得到

$$\frac{\partial E_{\boldsymbol{\theta}}}{\partial \boldsymbol{\theta}} = 2\boldsymbol{X}^{\mathrm{T}} (\boldsymbol{X}\boldsymbol{\theta} - \boldsymbol{Y}) \tag{2.29}$$

令上式为 0 即得 $\boldsymbol{\theta}$ 最优解的闭式解。当 $\boldsymbol{X}^{\mathrm{T}}\boldsymbol{X}$ 为满秩矩阵或正定矩阵时，可得：

$$\hat{\boldsymbol{\theta}} = \left(\boldsymbol{X}^{\mathrm{T}}\boldsymbol{X}\right)^{-1} \boldsymbol{X}^{\mathrm{T}}\boldsymbol{Y} \tag{2.30}$$

最终学得的线性回归模型为：

$$\hat{f}\left(\boldsymbol{x}_i\right) = h_{\hat{\boldsymbol{\theta}}}(\boldsymbol{x}_i) = \boldsymbol{x}_i^{\mathrm{T}} \left(\boldsymbol{X}^{\mathrm{T}}\boldsymbol{X}\right)^{-1} \boldsymbol{X}^{\mathrm{T}}\boldsymbol{Y} \tag{2.31}$$

注意: 线性回归有一些要求和特点。一，线性回归要求自变量和因变量之间必须满足线性关系，当然进行对数等变换最终为线性关系也可以。二是模型对异常值非常敏感，异常值会严重影响回归线和最终的预测值。三是多元回归存在多重共线性，自相关性和异方差性。多重共线性会增加系数估计的方差，使得系数估计不稳定，估计对模型中的微小变化非常敏感。四是在多个自变量的情况下，我们可以采用向前选择、向后消除的逐步选择方法来选择自变量。

2.3.2 Ridge 回归

现实任务中（$\boldsymbol{X}^{\mathrm{T}}\boldsymbol{X}$）往往不是满秩矩阵，常见的解决方法是引入正则化项。正则化是对最小化经验误差函数上加约束或先验信息，缩小求解的范围。正则化处理有两方面的作用：一是约束有引导作用，在优化误差函数的时候倾向于选择满足约束的梯度减少的方向，使最终的解倾向于符合先验知识，一般对应于稀疏参数的平滑解。二是解决了逆问题的不适定性，产生的解是唯一存在且依赖于数据的，噪声对不适定的影响就弱，解就不会过拟合，而且如果先验或正则化合适，则解就倾向于更符合真解，更不会过拟合了，即使训练集中彼此间不相关的样本数很少。

岭回归是当数据受变量高度相关引发多重共线性的影响时使用的一种技术，通过牺牲偏差，减少方差来防止过拟合。在多重共线性中，即使最小二乘估计 OLS 是无偏差的，但方差很大使得观察值远离真实值。

增加 L_2 正则项的目标函数：

$$J(\boldsymbol{\theta}) = \frac{1}{2}\sum_{i=1}^{n} \left(h_{\boldsymbol{\theta}}\left(\boldsymbol{x}_i\right) - y_i\right)^2 + \lambda \sum_{j=1}^{d} w_j^2, \quad \lambda > 0 \tag{2.32}$$

参数最优解：

$$\hat{\boldsymbol{w}}(\lambda) = \left(\boldsymbol{X}^{\mathrm{T}}\boldsymbol{X} + \lambda \boldsymbol{I}\right)^{-1} \boldsymbol{X}^{\mathrm{T}}\boldsymbol{Y} \tag{2.33}$$

通过确定的值可使得在方差和偏差之间达到平衡：随着 λ 的增大，模型方差减小而偏差增大。不假定正态性，岭回归与最小二乘回归的所有假设一样的。岭回归缩小系数的值，但没有达到零，这表明它没有进行特征选择。

2.3.3 Lasso 回归

惩罚函数使用的是系数的绝对值之和，导致惩罚项约束估计的绝对值之和，使得一些回归系数估计恰好为零。施加的惩罚越大，估计就越接近零。实现从 d 个变量中进行选择。

增加 L_1 正则项：

$$J(\boldsymbol{\theta}) = \frac{1}{2}\sum_{i=1}^{n}\left(h_{\boldsymbol{\theta}}\left(\boldsymbol{x}_i\right) - y_i\right)^2 + \lambda\sum_{j=1}^{d}|w_j|, \quad \lambda > 0 \tag{2.34}$$

参数最优解如下，其中 $i = 1, 2, ..., n$ 表示样本数，$j = 1, 2, ..., d$ 表示特征数，偏置项 b：

$$\hat{w}_j = \begin{cases} \frac{-c_j - \lambda}{a_j}, c_j < -\lambda \\ 0, -\lambda \leq c_j \leq \lambda \\ \frac{\lambda - c_j}{a_j}, \lambda > c_j \end{cases} \tag{2.35}$$

$$c_j = 2\sum_{i=1}^{d}\left(\sum_{i\neq j} w_j x_{ij} + b - y_i\right) x_{ij}, a_j = 2\sum_{i=1}^{d} x_{ij}^2 \tag{2.36}$$

稀疏：消除数据中一些特征，用来使模型泛化，减小过拟合的几率。Lasso 回归将系数收缩到零，有助于特征选择。如果一组自变量高度相关，则 Lasso 回归只会选择其中一个，而将其余的缩小为零。能够减少变异性和提高线性回归模型的准确性。不假定正态性，Lasso 回归与最小二乘回归的所有假设是一样的。

稀疏解：以二维为例，阴影部分是 L_1、L_2 正则项约束后的解空间，红色的等高线是凸优化问题中的目标函数（未加入正则项的）的等高线，如图所示，L_2 正则项约束后的解空间是圆形，而 L_1 正则项约束后的解空间是菱形，显然，菱形的解空间更容易在尖角处与等高线碰撞出稀疏解。

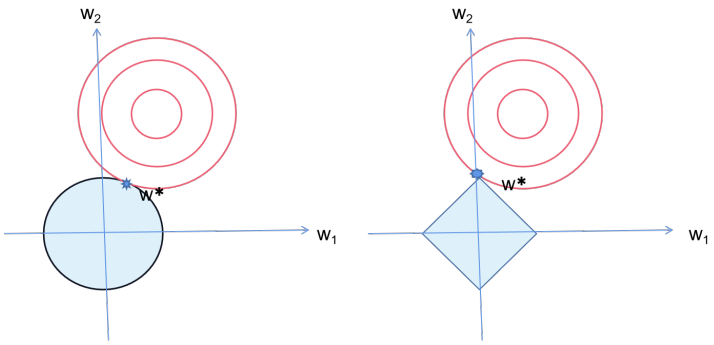


图 2.6: 几何空间的稀疏解图

2.3.4 弹性回归

弹性回归是岭回归和 Lasso 回归的混合技术，它同时使用 L_2 和 L_1 正则化。当有多个相关的特征时，弹性网络是有用的。Lasso 回归很可能随机选择其中一个，而弹性回归很可能都会选择。

同时增加 L_2 和 L_1 惩罚:

$$J(\boldsymbol{\theta}) = \frac{1}{2}\sum_{i=1}^{n}\left(h_{\boldsymbol{\theta}}\left(\boldsymbol{x}_i\right) - y_i\right)^2 + \lambda\left(\rho\cdot\sum_{j=1}^{d}|w_j| + (1-\rho)\cdot\sum_{j=1}^{d}w_j^2\right), \quad \rho\in[0,1] \tag{2.37}$$

在高度相关变量的情况下，弹性回归支持群体效应。对所选变量的数目没有限制，具有两个收缩因子。

2.3.5 多项式回归

研究人员假设的某些关系是曲线的。例如：组织生长速度、疾病流行病的进展、湖泊沉积物中碳同位素的分布检查残差等。

如一元 m 次多项式的回归方程为:

$$h_{\boldsymbol{\theta}}(x) = b_0 + b_1 x + b_2 x^2 + \ldots + b_m x^m \tag{2.38}$$

如果我们尝试用线性模型来拟合呈曲线分布的数据，则预测变量（X 轴）上的残差（Y 轴）的散点图将在中间具有许多正残差的斑块。因此，在这种情况下是不合适的，我们需要将每一项即 x 的某次幂看作一个新的变量。通常的多元线性回归分析的假设是所有自变量都是独立的。在多项式回归模型中，不满足该假设。

缺点：对异常值过于敏感。一两个异常值会严重影响非线性分析的结果，且用于检测非线性回归中的异常值的模型验证工具少于线性回归。

2.3.6 局部加权线性回归

对于有周期性、波动性的数据，并不能简单以线性的方式拟合，否则模型会偏差较大，而局部加权回归 (Lowess) 能较好的处理这种问题。可以拟合出一条符合整体趋势的线，进而做预测。也能较好的解决平滑问题。

思想：以某一个 x 为中心，向前后截取一段长度为 frac 的数据，对于该段数据用权值函数 π 做一个加权的线性回归，记 $(x,\hat{y})$ 为该回归线的中心值，其中 $\hat{y}$ 为拟合后曲线对应值。对于所有的 n 个数据点则可以做出 n 条加权回归线，每条回归线的中心值的连线则为这段数据的 Lowess 曲线。

代价函数:

$$J(\boldsymbol{\theta}) = \frac{1}{2}\sum_{i=1}^{n}\pi_i(x)\left(h_{\boldsymbol{\theta}}\left(x_i\right) - y_i\right)^2$$

权重函数:

$$\pi_i(x) = \exp\left(-\frac{\left(x_i - x\right)^2}{2k^2}\right) \tag{2.39}$$

其中 k 为带宽 (bandwidth) 常量，距离输入越远，权重越小，反之越大。权重函数使预测点的临近点在最小化目标函数中贡献大。w_i 是权重，它是通过要预测的点 i 与数据集中点的距离来确定的，距离越近权值越大，相应的误差值影响就越大，可以达到增加多项式的效果。

局部多项式回归通过加入偏差，降低预测的均方误差。当样本点 x_i 接近预测点 x 时，权值较大，接近于 1。当样本点 x_i 非常远离预测点 x 时，权值就会非常小，接近于 0。权值系数

$\pi_i(x)$ 指数衰减，其中参数 k 为衰减因子，k 越小即权重衰减的速率越快。权重常采用“核”函数的方式进行加权，例如 SVM 核函数的选择方案是通过两个样本的相似程度；而局部加权回归在每一次预测新样本时都会重新确定参数，以达到更好的预测效果。当数据规模比较大的时候计算量很大，学习效率很低。并且局部加权回归也不是一定能够完全避免欠拟合，因为那些波动的样本可能是异常值或者数据噪声。

2.3.7 案例

本案例中我们用线性回归（最小二乘、梯度下降）、Ridge 回归、Lasso 回归、弹性回归来预测美国波士顿地区房价中位数，并对这几种方法进行比较。

首先，需要导入该案例所需的模块函数和数据集，模块包括数据处理、构建模型和对比结果可视化的函数。

```
# 导入第三方库，用来生成美观的ASCII格式的表格
from prettytable import PrettyTable
# 从sklearn导入数据集
from sklearn.datasets import load_boston
# 导入交叉验证实现线性回归、岭回归、lasso回归、弹性回归
from sklearn.linear_model import LinearRegression, SGDRegressor, RidgeCV,
    LassoCV, ElasticNetCV
# 导入计算均方误差、平均绝对误差、$R^2$统计量
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
# 导入分割训练集和测试集的函数
from sklearn.model_selection import train_test_split
# 导入去均值和方差归一化的方法
from sklearn.preprocessing import StandardScaler
```

接下来对波士顿数据集进行预处理。划分训练集和测试集后，对训练集和测试集的特征值和目标值分别进行标准化处理。

```
# 波士顿数据集处理
def linearmodel():        # 自定义函数
    ld = load_boston()    # 加载数据集
    x_train, x_test, y_train, y_test = train_test_split(ld.data, ld.target, test
        _size=0.25) # 分割训练集和测试集
    std_x = StandardScaler()  # 特征值标准化处理
    x_train = std_x.fit_transform(x_train)
    x_test = std_x.transform(x_test)
    std_y = StandardScaler()      # 目标值处理
    y_train = std_y.fit_transform(y_train.reshape(-1, 1))
    y_test = std_y.transform(y_test.reshape(-1, 1))
```

数据预处理后，就可以运用模型对数据进行拟合和训练。首先用简单线性回归和梯度下降算法进行拟合训练如下：

```
    # 简单线性回归、梯度下降估计、岭估计对房价进行预测
    lr = LinearRegression()
```

```
    lr.fit(x_train, y_train.ravel())   # 简单线性回归拟合训练集
    y_lr_predict = lr.predict(x_test)  # 预测测试集
    sgd = SGDRegressor()
    sgd.fit(x_train, y_train.ravel())  # 梯度下降估计训练集
    y_sgd_predict = sgd.predict(x_test) # 预测测试集
```

用内置交叉验证的岭回归、lasso 回归和弹性回归对数据进行拟合和训练如下：

```
    # 交叉验证得到试验的惩罚因子，自动选择测试最优参数
    # GridSearchCV(ridge, param_grid={'alpha': np.logspace(-4, 4, 20)}, cv=3)
        专门调参的函数
    rd = RidgeCV(alphas=[0.01, 0.05, 0.1, 0.5, 1.0, 10.0])
    # 内置交叉验证的岭回归，一种有效的留一交叉验证的形式
    rd.fit(x_train, y_train.ravel())       # 岭回归拟合训练集
    y_rd_predict = rd.predict(x_test)      # 预测测试集
    lasso = LassoCV(alphas=[0.01, 0.05, 0.1, 0.5, 1.0, 10.0])
    # 内置交叉验证的lasso回归
    lasso.fit(x_train, y_train.ravel())     # lasso回归拟合训练集
    y_lasso_predict = lasso.predict(x_test) # 预测测试集
    en = ElasticNetCV(alphas=[0.01, 0.05, 0.1, 0.5, 1.0, 10.0],
         l1_ratio=[0.1, 0.3, 0.5, 0.7, 0.9]) # 选择弹性回归参数
    en.fit(x_train, y_train.ravel())       # 弹性回归拟合训练集
    y_en_predict = en.predict(x_test)      # 预测测试集
```

最后，分别计算这五种回归模型的模型得分（Score）、R^2、MSE 和 MAE 的值并用表格形式输出，比较这些模型的优劣。

```
    # 不同评价机制对五种模型的回归性能作出评价
    table = PrettyTable(['模型', 'score', 'R方', 'MSE', 'MAE'])
    table.add_row(['线性回归', lr.score(x_train, y_train.ravel()),
                   r2_score(y_test, y_lr_predict),
                   mean_squared_error(std_y.inverse_transform(y_test),
                   std_y.inverse_transform(y_lr_predict)),
                   mean_absolute_error(std_y.inverse_transform(y_test),
                   std_y.inverse_transform(y_lr_predict))])
                  # 将标准化后的数据转换为原始数据
    #机梯度下降回归、岭回归、Lasso回归、ElasticNet回归同上输出
    print(table)
if __name__ == '__main__':
linearmodel()   # 作为脚本直接执行
```

表 2.2: 不同回归算法的性能

模型	Score	R^2	MSE	MAE
线性回归	0.7216	0.7909	16.0844	2.8616
随机梯度下降回归	0.7172	0.7901	16.1414	2.8780

岭回归	0.7215	0.7912	16.0584	2.8587
Lasso 回归	0.7166	0.7847	16.5619	2.8721
ElasticNet 回归	0.7211	0.7920	15.9990	2.8517

通过这一比较发现，ElasticNet 回归的 R^2 最大，MAE 和 MSE 最小，效果较好。MSE 均方误差是误差的平方的期望值，MAE 为平均绝对误差，均是越小越好；R^2 取值范围为 [0,1]，越大表示模型拟合效果越好。如果面对训练数据规模十分庞大的任务，随机梯度法不论是在分类还是回归问题上都表现的十分高效，可以在不损失过多性能的前提下，节省大量计算时间，但是精度有待考量。所有这些回归正则化方法（Lasso，Ridge 和 ElasticNet）在数据集中变量之间的高维度和多重共线性的情况下都能很好地工作。

2.4 分类算法

我们要明确，不推荐用回归解决分类问题，回归问题最好用回归算法，分类问题就用分类算法。因为线性回归用来解决分类问题时，稳定性差。当样本分布比较复杂或有异常值时，线性回归无法做到准确的分类。

2.4.1 Logistic 回归

Logistic 分布是连续型的概率分布，其分布函数如下：

$$F(x) = P(X \leq x) = \frac{1}{1 + e^{-(x-\mu)/\gamma}}, -\infty < \mu < \infty, \sigma > 0, \quad \gamma > 0 \tag{2.40}$$

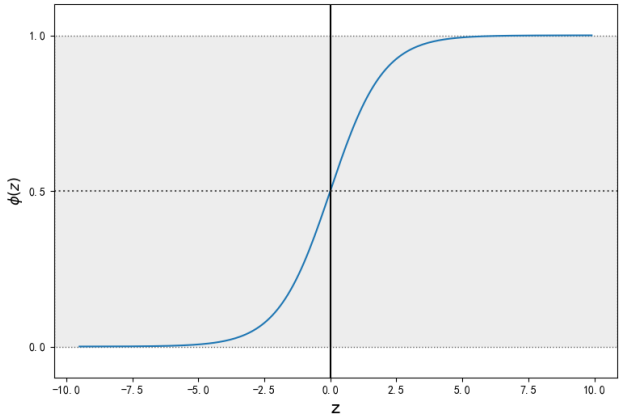


图 2.7: Sigmoid 函数图

Sigmoid 函数: 用于隐层神经元输出，取值范围为 (0,1)，它可以将一个实数映射到 (0,1) 的区间，可以结合阈值用来做二分类。选择 0.5 作为阈值是一个一般的做法，实际应用时特定的情况可以选择不同阈值，如果对正例的判别准确性要求高，可以选择阈值大一些，若对正例的召回要求高，则可以选择阈值小一些。公式形式为 $\phi(z) = \frac{1}{1+e^{-z}}$。

模型

样本为 1 的概率

$$p(x) = \frac{1}{1+e^{-\theta^T x}} = \frac{e^{\theta^T x}}{1+e^{\theta^T x}} \tag{2.41}$$

事件几率

$$logit(p) = \log \frac{p}{1-p} = \log \left(\frac{\frac{1}{1+e^{-\theta^T x}}}{\frac{e^{-\theta^T x}}{1+e^{-\theta^T x}}} \right) = \theta^T x \tag{2.42}$$

二分类可以推出多分类, 通过 one-hot 编码归一化。一对多的划分可能存在数据不均衡，有一定的倾向性；一对一的划分直观，模型个数是 $k(k-1)/2$，其中 k 是要划分的类别数。若某连续随机分布的对数能写成随机变量一次项和对数项的和，则该分布是 Gamma 分布；若某对数分布能够写成随即变量二次形式，则该分布必然是正态分布。

思路：先拟合决策边界，回归方程不局限于线性还是多项式，再建立这个边界与分类的概率联系，从而得到了二分类情况下的概率。

损失函数：交叉熵（log 对数损失函数）；激活函数的梯度在两端为 0。

求解

可以考虑极大似然估计、梯度下降等，交叉熵侧重分类正确的损失，而 MSE 侧重全部。在求根过程中，二分法迭代次数较多速率慢，梯度下降法是针对平面，迭代也较慢。牛顿法是针对二次曲面，是极小值点在二阶的泰勒展开，极小值点的估计迭代较少，优化函数二次曲线一步到位，二阶导数保证除了当前之外下一步还是最陡的。

考虑二分类的条件概率求解似然函数：

$$L(w) = \prod_{i=1}^{n} \left[p\left(x_i\right) \right]^{y_i} \left[1 - p\left(x_i\right) \right]^{1-y_i} \tag{2.43}$$

通常采用梯度下降法或拟牛顿法求解参数 w。得到的损失函数越大, 证明我们得到的 w 就越好。因为在函数最优化的时候习惯让一个函数越小越好, 所以我们在前边加一个负号。就得到我们逻辑回归的损失函数，我们叫它交叉熵损失函数。在逻辑回归模型中，我们最大化似然函数和最小化损失函数实际上是等价的。

优点：(1) 直接对分类的概率建模，无需假设数据分布，避免了假设分布不准确带来的问题（区别于生成式模型）；(2) 不仅可预测出类别，还能得到该预测的概率，对一些利用概率辅助决策的任务很有用；(3) 对数几率函数是任意阶可导的凸函数，有许多数值优化算法都可以求出最优解。

总结

逻辑回归是一种用于解决二分类（0 或 1）问题的机器学习方法，用于估计某种事物的可能性。比如某用户购买某商品的可能性，某病人患有某种疾病的可能性，以及某广告被用户点击的可能性等。注意，这里用的是 “可能性”，而非数学上的 “概率”，logisitc 回归的结果并非数学定义中的概率值，不可以直接当做概率值来用。该结果往往用于和其他特征值加权求和，而非直接相乘。

逻辑回归与线性回归都是一种广义线性模型。逻辑回归假设因变量 y 服从伯努利分布，而线性回归假设因变量 y 服从高斯分布。因此与线性回归有很多相同之处，去除 Sigmoid 映射函数的话，逻辑回归算法就是一个线性回归。可以说，逻辑回归是以线性回归为理论支持的，但是逻辑回归通过 Sigmoid 函数引入了非线性因素，因此可以轻松处理 0/1 分类问题。

Softmax 回归

Softmax 回归是 logistic 回归的一般形式，logistic 回归用于二分类，而 softmax 回归用于多分类。思想是对于输入数据的 k 个类别，Softmax 回归主要估算其归属于每一类的概率，即

$$h_{\boldsymbol{\theta}}\left(x_i\right)=\begin{bmatrix} p\left(y_i=1 \mid x_i;\theta\right) \\ p\left(y_i=2 \mid x_i;\theta\right) \\ \vdots \\ p\left(y_i=k \mid x_i;\theta\right) \end{bmatrix}=\frac{1}{\sum_{j=1}^{k} e^{\theta_j^T x_i}}\begin{bmatrix} e^{\theta_1^T x_i} \\ e^{\theta_2^T x_i} \\ \vdots \\ e^{\theta_k^T x_i} \end{bmatrix} \tag{2.44}$$

概率取值属于 [0,1] 且和为 1，即 softmax 回归将输入数据 x_i 归属于类别 j 的概率为：

$$p\left(y_i=j \mid x_i;\theta\right)=\frac{e^{\theta_i^T x_i}}{\sum_{j=1}^{k} e^{\theta_j^T x_i}} \tag{2.45}$$

代价函数：

$$L(\theta)=-\frac{1}{n}\left[\sum_{i=1}^{n}\sum_{j=1}^{k} I\left\{y_i=j\right\}\log\frac{e^{\theta_i^T x_i}}{\sum_{j=1}^{k} e^{\theta_j^T x_i}}\right] \tag{2.46}$$

其中 I{ · } 是示性函数，即 I(值为真的表达式)=1，I(值为假的表达式)=0。

案例

泰坦尼克号乘客存活率预测。

首先，导入所需的模块函数并分别读取泰坦尼克数据集的训练集和测试集，将这两个数据集进行合并以便后续统一进行数据处理。

```
# 导入数据分析工具
import pandas as pd
# 导入分割训练集和测试集的函数
from sklearn.model_selection import train_test_split
# 导入逻辑回归算法
from sklearn.linear_model import LogisticRegression
# 读入数据
train = pd.read_csv("F:/pytest/train.csv") # 训练数据集: (891, 12)
test = pd.read_csv("F:/pytest/test.csv")   # 测试数据集: (418, 11)
rowNum_train = train.shape[0]        # 测试和训练集样例数
rowNum_test = test.shape[0]
full = train.append(test, ignore_index=True) # 合并后的数据集: (1309, 12)
```

接下来进行数据处理。首先对合并后的数据集中的缺失值进行填补，本例中对“Age”、“Fare”、“Embarked” 和 “Cabin” 属性列进行填补。

```
# 缺失值处理
full.info()  # 获取处理前合并数据集的信息
full['Age'] = full['Age'].fillna(full['Age'].mean())
# 用均值填补数值型数据的缺失值
full['Fare'] = full['Fare'].fillna(full['Fare'].mean())
full['Embarked'].head()     # 查看处理后Embarked上船的地方前5行
full['Embarked'] = full['Embarked'].fillna('S')
# 用众数填补属性数据的缺失值
full['Cabin'] = full['Cabin'].fillna('U')
full.info()                 # 查看缺失值处理后的数据信息
```

接下来对数据中的属性值进行处理。对具有属性特征的列进行赋值，可以直接设置哑变量，也可以用 One-Hot 编码进行分类赋值。

```
# 属性数据处理
sex_mapDict = {'male': 1, 'female': 0}      # 设置哑变量
full['Sex'] = full['Sex'].map(sex_mapDict)
# 根据提供的函数对指定序列做映射
embarkedDf = pd.DataFrame()                 # 定义空数据框
# 对类别型特征做One-Hot编码
embarkedDf = pd.get_dummies(full['Embarked'], prefix='Embarked')
full = pd.concat([full, embarkedDf], axis=1) # 数据按列作简单融合
full.drop('Embarked', axis=1, inplace=True)  # 删除表中Embarked列
pclassDf = pd.DataFrame()
pclassDf = pd.get_dummies(full['Pclass'], prefix='Pclass')  # 编码
full = pd.concat([full, pclassDf], axis=1)  # 分类数据合并设计阵
full.drop('Pclass', axis=1, inplace=True)
# 姓名Name变量替换为头衔变量
def getTitle(name):
    str1 = name.split(',')[1]  # 通过指定分隔符对字符串进行切片
    str2 = str1.split('.')[0]
    str3 = str2.strip()        # 移除字符串头尾指定的字符
    return str3                # 从姓名中获取头衔
titleDf = pd.DataFrame()       # 存放提取后的特征
titleDf['Title'] = full['Name'].map(getTitle)  # 根据头衔作映射
title_mapDict = {"Capt": "Officer","Col": "Officer",
     "Major": "Officer","Jonkheer": "Royalty",
     "Don": "Royalty","Sir": "Royalty",
         "Dr": "Officer","Rev": "Officer",
     "the Countess": "Royalty",
         "Dona": "Royalty","Mme": "Mrs",
     "Mlle": "Miss","Ms": "Mrs",
         "Mr": "Mr","Mrs": "Mrs","Miss": "Miss",
     "Master": "Master","Lady": "Royalty"} # 对应关系
titleDf['Title'] = titleDf['Title'].map(title_mapDict)
titleDf['Title'].value_counts()  # 分类计数
```

```
titleDf = pd.get_dummies(titleDf['Title']) # 对特征做One-Hot编码
full = pd.concat([full, titleDf], axis=1)  # 将头衔数据合并到full
full.drop('Name', axis=1, inplace=True)    # 删除表中的Name列
```

通过相关系数对特征进行初步选择，合并特征数据后划分训练集和测试集，运用逻辑回归模型进行拟合预测，并得到模型预测的准确率结果。

```
# 特征选择
corrDf = full.corr()  # 相关系数矩阵
corrDf['Survived'].sort_values(ascending=False)  #查看相关系数
full_X = pd.concat([titleDf, pclassDf, familyDf, full['Fare'], cabinDf,
    embarkedDf, full['Sex']], axis=1) # 特征数据合并
# 创建训练数据和测试集
source_X = full_X.iloc[:891, :]         # 原始数据集特征
source_y = full.loc[:890, 'Survived']  # 标签
pred_X = full_X.iloc[891:, :]           # 预测数据集特征
print('预测数据集行列数:', pred_X.shape)
train_X, test_X, train_y, test_y = train_test_split(source_X, source_y, train_
    size=.8)          # 数据分割
# 逻辑回归模型
model = LogisticRegression()
model.fit(train_X, train_y)   # 逻辑回归分类
LogisticRegression(C=1.0, class_weight=None, dual=False,
             fit_intercept=True,intercept_scaling=1,
             max_iter=100, multi_class='ovr',n_jobs=1,
             penalty='l2', random_state=None,
             solver='liblinear', tol=0.0001,
             verbose=0, warm_start=False)
model.score(test_X, test_y)  # 评估模型准确率
```

可得用逻辑回归分类得到的最终的模型准确率为 0.85。其中参数 penalty 表示惩罚项选择 L_2，即向量中各个元素平方之和再开根号，作用是选择较多的特征，使他们都趋近于 0；C 值越小，则正则化强度越大。

2.4.2 决策树与随机森林

树形与线性模型的区别在于之前线性模型是所有特征给予权重相加得到一个新的值，而树形模型是一个一个地对特征进行处理。

决策树与逻辑回归的分类区别也在于此，逻辑回归是将所有特征变换为概率后，通过大于某一概率阈值的划分为一类，小于某一概率阈值的为另一类；而决策树是对每一个特征做一个划分。另外逻辑回归只能找到线性分割，输入特征 x 与 $logit$ 之间是线性的，除非对 x 进行多维映射，而决策树可以找到非线性分割。

表 2.3: 逻辑回归和决策树的比较

	逻辑回归	决策树桩
得分函数	Sigmoid 函数（连续）	阶跃函数（非连续）
得分 y	概率	分类众数；回归平均数
自变量 x	连续	连续或离散
标签	0，1	多标签也可
损失函数	交叉熵	分类平均基尼系数；回归平均平方差
求解方法	梯度下降	离散穷举
得到结果	权重与截距	阈值或分类子集

决策树

决策树是一种树形结构，采用自顶向下的递归的方法，其中每个内部节点表示在一个属性上的测试，每个分支代表一个测试输出，每个叶节点代表一种类别。

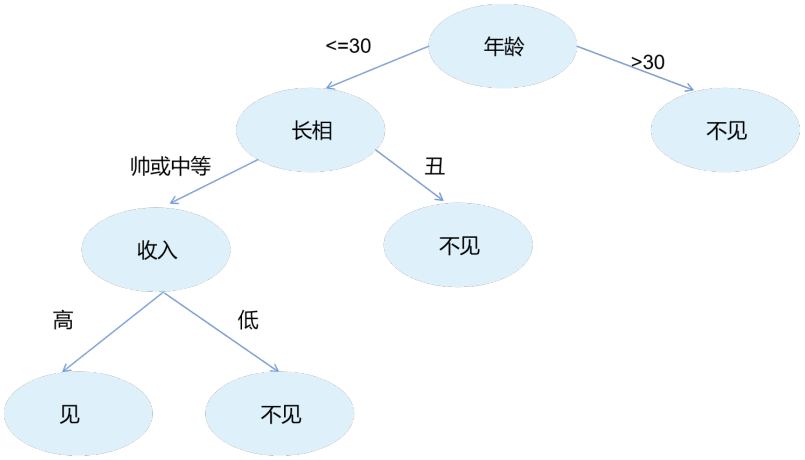


图 2.8: 决策树图

（1）工作原理：决策树以迭代地方式将数据分解成不同的子集来工作。对于回归树，选择它们来最小化所有子集中的 MSE（均方误差）或 MAE（平均绝对误差）是不错的选择；对于分类树，选择分解数据以便最小化所得到的子集中的熵或基尼杂质，构造一棵熵值下降最快的树，所得到的分类器将特征空间分成不同的子集，直到所有的样本都属于一个类别，即一个叶子节点，叶子节点处熵值尽可能为 0。

（2）学习步骤：特征选择、决策树的生成和决策树的剪枝。

由图可得，高度是 H 的“二叉树”最多有 H^2 个叶子节点。要想做随机森林，只要会决策树；要想会决策树，只要会一层的划分递归就可以。那么划分依据是什么，如何进行特征选择呢?

（3）特征选择原理：通过建树和剪枝寻找最纯净的划分，建立决策树的关键是当前状态下选择哪个属性作为分类依据，使得不确定性越来越小，根据不同的目标函数主要有三种算法。

（4）算法

ID3 算法（信息增益）是一种贪心算法，用来构造决策树。ID3 算法起源于概念学习系统（CLS），以信息熵的下降速度为选取测试属性的标准，即在每个节点选取还尚未被用来划分的具

有最高信息增益的属性作为划分标准，然后继续这个过程，直到生成的决策树能完美分类训练样例。

C4.5 算法（信息增益率）采用信息增益率来选择最优划分属性。不是直接选择信息增益率最大的候选划分属性，而是先从候选划分属性中找出信息增益高于平均水平的属性，再从中选择信息增益率最高的。采用了信息增益比来选择特征，以减少信息增益容易选择特征值多的特征的问题。

CART 算法（分类：基尼系数；回归：均方差）CART 分类树算法使用基尼系数来代替信息增益比，基尼系数代表了模型的不纯度，基尼系数越小，则不纯度越低，特征越好。这和信息增益 (比) 是相反的。CART 回归树的度量目标是，对于任意划分特征 A，对应的任意划分点 s 两边划分成的数据集 D1 和 D2，求出使 D1 和 D2 各自集合的均方差最小，同时 D1 和 D2 的均方差之和最小所对应的特征和特征值划分点。

（5）损失函数

评价：假定样本的总类别为 k 个。对于决策树的某叶结点，假定该叶结点含有样本数目为 n, 其中第 k 类的样本点数目为 n_k，$k = 1, 2.., k$。若某类样本 $n_j = n$ 而其他均为 0，称该结点为纯结点; 若各类样本数目 $n_1 = n_2 = n_k = n/k$，称该样本为均结点。纯结点的熵 H_p=0 最小。均结点的熵 $\mathrm{H}_u = lnk$ 最大。

对所有叶结点的熵求和，该值越小说明对样本的分类越精确。各叶结点包含的样本数目不同，可使用样本数加权求熵和。评价函数为 $\mathrm{C(T)} = \sum_{\mathrm{t}\in\mathrm{leaf}} \mathrm{N_t H(t)}$，其中 N_t 是每个叶子节点中样本的个数，在这个公式中相当于权重；$\mathrm{H}(t)$ 是每个叶子节点的熵值，评价函数越小越好。

过拟合

决策树对训练数据有很好的分类能力，但对未知的测试数据未必有好的分类能力, 泛化能力弱，即可能发生过拟合现象。解决方案是对决策树进行剪枝或构建随机森林。在此处我们主要介绍剪枝，随机森林在后续进行介绍。

（1）预剪枝

预先设置每一个结点所包含的最小样本数目，例如 10，若该结点总样本数小于 10 时，则不再分；或者预先指定树的高度或者深度，例如树的最大深度为 4；或者指定结点的熵小于某个值时就不再继续划分。

（2）后剪枝算法

对于给定的完全决策树 T_0，计算所有内部节点的剪枝系数，查找最小剪枝系数的结点，剪枝得到决策树 T_1，再次剪枝部分结点得到 T_2，...，重复以上步骤，直到仅剩树根的决策树 T_k 只有一个结点。在验证数据集上对这 k 个树分别评价，选择损失函数最小的树 T_α(结点 r 的剪枝系数)。

（3）剪枝系数

叶节点越多，决策树越复杂，泛化损失越大，进行模型复杂度修正 $\mathrm{C}_\alpha(T)$=$\mathrm{C}(T) + \alpha \bullet |T_{\mathrm{leaf}}|$，$\alpha$=0 未剪枝的决策树损失最小，$\alpha$=$\infty$ 单根节点的损失越小，其中 T 表示树，$\mathrm{C}(T)$ 表示当前损失，T_{leaf} 表示分裂后的叶子节点数。因为是从全树到树桩的剪枝过程，所以，α 是从小开始，逐渐增大的。假设当前对以 t 为根节点的子树 T_t 剪枝，剪枝后只保留 t 本身，而删除掉所有的子节点，剪枝前后损失函数一致，剪枝后的损失函数为 $\mathrm{C}_\alpha(t)$=$\mathrm{C}(t) + \alpha$，剪枝前的损失函数为

$C_\alpha(T_t)=C(T_t)+\alpha \bullet |T_{\text{leaf}}|$。可得结点 r 的剪枝系数 $\alpha=\frac{C(t)-C(T)}{T_{\text{leaf}}-1}$。因为损失相同，那么就取复杂度小的，所以就可以剪枝。

决策树是弱分类器，层数不多时分类效果一般，是一种非线性模型，适用面广，容易过拟合。不加约束时一般用 CART 算法，数据数量差别大可先考虑 C4.5 算法。最大深度设定后，若叶子节点仍有多类别未训练好，可按照概率判别。

随机变量不确定性的度量

（1）信息量

随机变量 x 的概率分布为 $p(x)$，则 $h(x)=-\log_2 p(x)$，即某事件发生的概率越小，该事件的信息量越大。

（2）信息熵

随机事件信息量的期望，不确定性的度量。$H(x)=-\sum_{x\in X} p(x)\ln p(x)$。特殊地，两点分布的信息熵，$H(x)=-\sum_{x\in X} p(x)\ln p(x)=-p\ln p-(1-p)\ln(1-p)$。熵是随机变量不确定性的度量，不确定性越大，熵值越大，若退化为定值则熵最小为 0，若随机分布为均匀分布则熵最大，所以信息熵的取值范围为 $0\le H(x)\le \log|x|$。

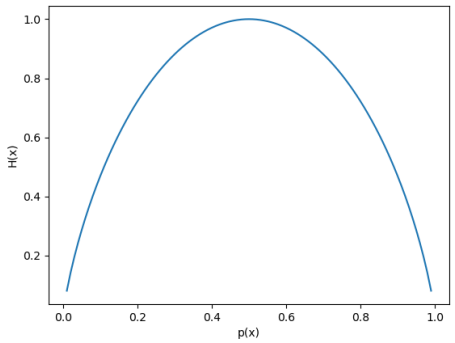


图 2.9: 两点分布的信息熵图

（3）条件熵

在已知随机变量 X 的条件下随机变量 Y 的不确定性可表示为：$H(Y\mid X)=\sum_{i=1}^{n} p_i H(Y\mid X=x_i)$，当熵和条件熵中的概率由数据估计（特别是极大似然估计）得到时，所对应的熵与条件熵分别称为经验熵和经验条件熵。

（4）信息增益

信息增益表示得知特征 X 的信息而使得类 Y 的信息的不确定性减少的程度。特征 A 对训练数据集 D 的信息增益 g(D,A) 为集合 D 的经验熵 H(D) 与特征 A 给定条件下 D 的经验条件熵 H(D|A) 之差，即：$g(D,A)=H(D)-H(D\mid A)$，表示训练数据与特征的互信息，越高越好，表示分割性越强。这种表示方法下得到的决策树为 ID3 算法。

（5）信息增益率

信息增益率在信息增益的基础上增加了惩罚项，惩罚项是特征的固有值，是避免上述情况而设计的。定义为信息增益除以特征的固有值，即 $\frac{g(Y,X)}{H(X)}$，起到惩罚的作用，类别越多这个量越

大，得到的决策树为 C4.5 算法。

（6）基尼系数

从数据集中随机抽取两个样本类别标记不一致的概率，得到的决策树为 CART 算法（分类与回归树）。Gini $(p) = \sum_{k=1}^{K} p_k (1 - p_k) = 1 - \sum_{k=1}^{K} p_k^2$，不确定性越大，基尼系数越大，越不好划分。（贫富差距也用基尼系数，但越大越利于划分，与此处概念不同）。

随机森林

尽管有剪枝等方法，一棵树的生成肯定还是不如多棵树，因此就有了随机森林来解决决策树泛化能力弱的缺点。由此便引出集成学习的基本问题：有一些预测效果一般的“弱学习器”(基学习器)，以某种方式将它们组合构成一个预测效果优良的“强学习器”。

Bootstrap：有放回的采样，每个采样出来的样本集都和原始数据集一样大。从原始数据中抽取子集，分别求各个子集的统计特征，最终将统计特征合并，即以学习器之间串行的方式去学习，每个学习器带有权重。

Bagging：首先从数据集中采样出 T 个数据集，然后基于这 T 个数据集，每个训练出一个基分类器，将这些基分类器进行组合做出预测，即学习器之间并行的方式去学习。Bagging 在做预测时，对于分类任务使用简单的投票法。

随机森林：与 Bagging 决策树相似，从样本集中用 Bootstrap 随机选取 n 个样本，只不过 Bagging 在分割时考虑全部的 M 个特征，随机森林只考虑 $m < M$ 个特征。随机森林中有许多的分类树，我们要将一个输入样本进行分类，需要将输入样本输入到每棵树中进行分类。

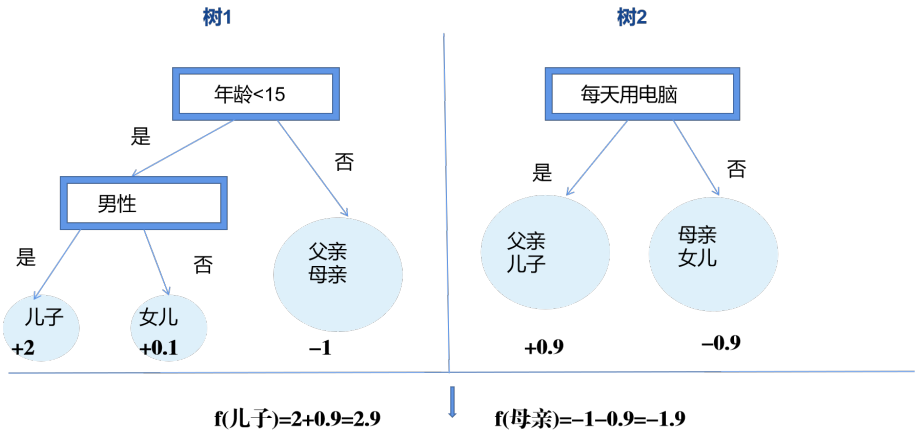


图 2.10: 随机森林图

（1）思想：

样本作随机（从样本集中用 Bootstrap 随机选取 n 个样本）；特征作随机，所有属性中随机选择 k 个属性（决策树当前最优——梯度下降局部最优）（不是完全最优，而是概率最优）；重复 m 次，建立 m 棵 CART 决策树形成随机森林，投票表决属于哪一类（阈值、加权、众数概率、贝叶斯）。

（2）原理：

由于随机决策树生成过程采用的 Boostrap，所以在一棵树的生成过程并不会使用所有的样

本，未使用的样本就叫 oob(out of bag) 袋外样本。通过袋外样本，可以评估这个树的准确度；最后，其他子树取平均值即是随机森林算法的性能。

在损失函数方面，分类随机森林对应的 CART 分类树默认是基尼系数 (gini)，另一个可选择的标准是信息增益。回归随机森林对应的 CART 回归树默认是均方误差 (MSE)，另一个可以选择的标准是绝对值误差 (MAE)。

(3) 应用:

特征选择——因为袋外样本的存在，因此不需要进行十字交叉测试（节省时间），通过依次对每个特征赋予一个随机数，观察算法性能的变化，倘若变化大，则说明该特征重要，sklearn 中会对每个特征赋予一个分数，分数越大，特征越重要，因此可以根据特征重要性排序，然后选择最佳特征组合。

计算特征重要度——计算正例经过的结点，使用经过结点的数目、经过结点的 gini 系数和等指标。或者随机替换一列数据，重新建立决策树，计算新模型的正确率变化，从而考虑这一列特征的重要性。

异常值检测——统计样本间位于相同决策树的叶结点的个数，形成样本相似度矩阵。

(4) 总结:

随机森林是一个包含多个决策树的分类器，并且其输出的类别是由个别树输出的类别的众数而定。Leo Breiman 和 Adele Cutler 研究推出了随机森林的算法。随机森林在过去几年一直是新兴的机器学习技术。它是基于非线性的决策树模型，通常能够提供准确的结果。然而，随机森林大多是黑盒子，难以解读和充分理解。

当然可以使用决策树作为基本分类器，也可以使用 SVM、Logistic 回归等其他分类器，习惯上，这些分类器组成的“总分类”器仍然叫做随机森林。

随机森林算法特征维度不同于主成分分析 (PCA)，随机森林算法能够考虑到特征对类别的影响，而 PCA 是单纯的数据方差；随机森林的缺点是需要迭代计算，如果在大数据下进行选择，就难免有点捉襟见肘；与线性判别分析 (LDA) 区别在于 LDA 根据标签通过变换将同标签数据距离缩小，将累间距离方法；LDA 是一种有监督方法，PCA 属于无监督方法。

案例

决策树/随机森林对鸢尾花数据两特征组合的分类。多个决策树：采用特征组合的方式，将鸢尾花四个特征两两组合，分别建立决策树模型，并对其进行验证。首先导入绘图和数据处理的相关函数，并导入划分数据集、判断准确率和构建决策树随机森林的模块函数。

```
# 导入图像
import matplotlib as mpl
import matplotlib.pyplot as plt
# 导入数据处理
import numpy as np
import pandas as pd
# 导入准确率函数
from sklearn.metrics import accuracy_score
# 分割测试集和训练集
from sklearn.model_selection import train_test_split
```

```
# 导入决策树和随机森林算法
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
```

然后设置图像中文标签和坐标轴显示负号，将鸢尾花的四个特征用字符显示，读取鸢尾花的数据文件后提取特征值并将目标值数字化，分割数据集并将特征两两组合，并设置返回元素及索引。

```
if __name__ == "__main__":
    mpl.rcParams['font.sans-serif'] = ['SimHei'] # 中文标签字体
    mpl.rcParams['axes.unicode_minus'] = False   # 显示负号
    #字符显示(解决matplotlib不能读取中文的问题)
    iris_feature = u'花萼长度', u'花萼宽度', u'花瓣长度',
                   u'花瓣宽度'  # 特征元组
    path = 'iris.data'  # 数据文件路径
    data = pd.read_csv(path, header=None)
    x_prime = data[list(range(4))]  # 特征值
    y = pd.Categorical(data[4]).codes  # 目标值数字化
    # 分割数据集
    x_prime_train, x_prime_test, y_train, y_test =
    train_test_split(x_prime, y, train_size=0.7, random_state=0)
    # 特征两两组合，共6对
    feature_pairs = [[0,1], [0,2], [0,3], [1,2], [1,3], [2,3]]
    # 图像宽高和背景白色
    plt.figure(figsize=(8, 6), facecolor='#FFFFFF')
    # 枚举：返回的是元素以及对应的索引
    for i, pair in enumerate(feature_pairs):
    x_train = x_prime_train[pair]
    x_test = x_prime_test[pair]
```

接下来进行决策树或随机森林模型的学习，设置可视化绘图的一些参数，并输出训练集和测试集预测结果。

```
    # 决策树/随机森林学习
    model = DecisionTreeClassifier(criterion='entropy',min_samples_leaf=3) # 损
        失函数：gini(基尼系数)或entropy(信息熵)
    # model = RandomForestClassifier(n_estimators=100,criterion='entropy',max_
        depth=5, oob_score=True)
    # 画图与预测
    N, M = 500, 500  # 横纵各采样个数
    x1_min, x2_min = x_train.min()
    x1_max, x2_max = x_train.max()
    t1 = np.linspace(x1_min, x1_max, N)  # 坐标轴范围
    t2 = np.linspace(x2_min, x2_max, M)
    x1, x2 = np.meshgrid(t1, t2)          # 生成网格采样点
    x_show = np.stack((x1.flat, x2.flat), axis=1)  # 测试集
    # 训练集与测试集的预测结果
```

```
    y_train_pred = model.predict(x_train)
    acc_train = accuracy_score(y_train, y_train_pred)
    y_test_pred = model.predict(x_test)
    acc_test = accuracy_score(y_test, y_test_pred)
    print('特征：', iris_feature[pair[0]], ' + ', iris_feature[pair[1]])
    # print('OOB Score:', model.oob_score_)  #最佳特征选择
    print('\t训练集准确率：%.4f%%' % (100 * acc_train))
    print('\t测试集准确率：%.4f%%\n' % (100 * acc_test))
```

继续进行可视化操作，将预测的结果以不同颜色的样本点显示出来，并直观绘出分类边界，对图形的横纵坐标刻度和名称等各个参数进行设置，最后以图形的形式输出两两特征组合最终的分类结果。

```
    cm_light = mpl.colors.ListedColormap(['#A0FFA0', '#FFA0A0', '#A0A0FF'])
    cm_dark = mpl.colors.ListedColormap(['g', 'r', 'b'])
    y_hat = model.predict(x_show)  # 预测集的预测值
    y_hat = y_hat.reshape(x1.shape)
    plt.subplot(2, 3, i + 1)
    plt.contour(x1, x2, y_hat, colors='k', levels=[0, 1],
                antialiased=True, linewidths=1)
    plt.pcolormesh(x1, x2, y_hat, shading='auto', cmap=cm_light)
    # 预测值（直观表现出分类边界）
    plt.scatter(x_train[pair[0]],x_train[pair[1]],c=y_train,s=20,edgecolors='k',
        cmap=cm_dark,label=u'训练集')
    plt.scatter(x_test[pair[0]],x_test[pair[1]],c=y_test,s=80,marker='*',
        edgecolors='k',cmap=cm_dark,label=u'测试集')
    plt.xlabel(iris_feature[pair[0]], fontsize=12) # 横纵坐标标签
    plt.ylabel(iris_feature[pair[1]], fontsize=12)
    # plt.legend(loc='upper right', fancybox=True, framealpha=0.3)
    plt.xlim(x1_min, x1_max)  # 横纵坐标刻度
    plt.ylim(x2_min, x2_max)
    plt.grid(b=True, ls=':', color='#606060')
    plt.suptitle(u'决策树对鸢尾花数据两特征组合的分类结果', fontsize=15)
    plt.tight_layout(rect=(0, 0, 1, 1))  # 自动调整子图参数
    plt.show()
```

输出的结果如下：

```
特征： 花萼宽度 + 花瓣长度
    训练集准确率：97.1429%
    测试集准确率：95.5556%

特征： 花萼宽度 + 花瓣宽度
    训练集准确率：96.1905%
    测试集准确率：95.5556%

特征： 花瓣长度 + 花瓣宽度
    训练集准确率：98.0952%
    测试集准确率：97.7778%
```

图 2.11: 决策树对鸢尾花数据两特征组合的分类准确率

可见，花瓣长度和花瓣宽度这两个属性对于分类的准确率更高，分类效果更好。决策树中 max_depth(default=None) 设置了决策树和随机森林中树的最大深度，深度越大越容易过拟合，推荐树的深度为 5 到 20。min_samples_leaf 这个值限制了叶子节点最少的样本数，如果某叶子节点数目小于样本数，则会和兄弟节点一起被剪枝。经验上这个值必须大于 100，如果一个节点都没有 100 个样本支持这个决策，一般都被认为是过拟合。从下图中可以更直观的看到分类结果。

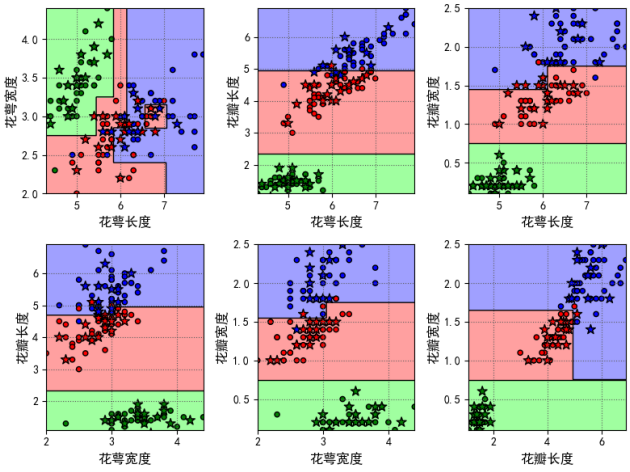


图 2.12: 决策树对鸢尾花数据两特征组合的分类

随机森林中 n_estimators 是随机森林中决策树个数，整数 (默认值为 10)。feature _ importances _ 是特征的重要性，值越高特征越重要。oob_score(default=False) 是否使用袋外样本来估计泛化精度，如果设置为使用时能获得训练数据集的得分。

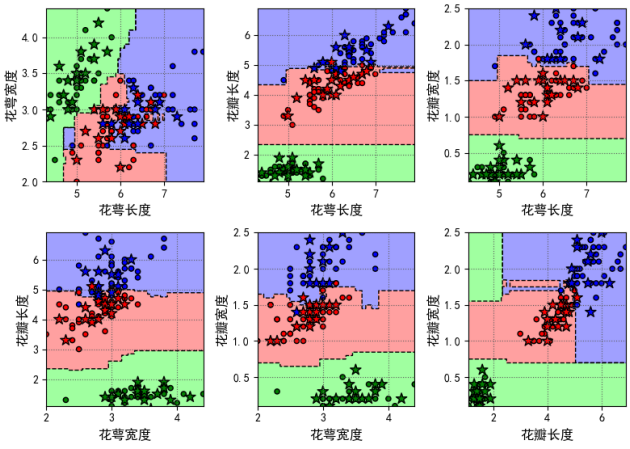


图 2.13: 随机森林对鸢尾花数据两特征组合的分类

2.4.3 支持向量机

概念

(1) 线性可分：C 和 D 是 n 维欧氏空间中的两个点集。如果存在 n 维向量 $\boldsymbol{w}$ 和实数 b，使得所有属于 C 的点都有 $\boldsymbol{wx}+b>0$，而对于所有属 D 的点则 $\boldsymbol{wx}+b<0$，则我们称 C 和 D 线性可分。在二维空间上，两类点被一条直线完全分开叫做线性可分。

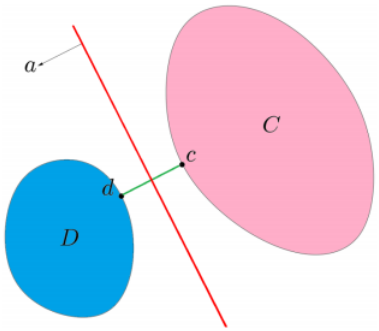


图 2.14: 线性可分图

(2) 最大间隔超平面：从二维扩展到多维空间中时（线—面—超平面），将 C 和 D 完全正确地划分开的 $\boldsymbol{wx}+b=0$ 就成了一个超平面。为了使这个超平面更具鲁棒性，我们会去找最佳超平面，即以最大间隔把两类样本分开的超平面，也称之为最大间隔超平面。两类样本分别分割在该超平面的两侧；两侧距超平面最近的样本点到超平面的距离被最大化。

上图中 C 和 D 是二维欧氏空间的两个点集，直线 a 可将这两个集合完全分开，两侧距直线 a 最近的样本点分别为 d 和 c，是线性可分的情况。

(3) 支持向量：样本中距离超平面最近的一些点，这些点叫做支持向量。所以我们的目标函数是使样本到直线的最小距离取最大（最大化 margin）；而线性回归、逻辑回归的目标函数是使似然函数取最大，决策树的目标函数是使叶子结点的信息熵的和。

(4) SVM：SVM 是一种二分类模型，它的目的是寻找一个超平面来对样本进行分割，即最优决策边界，分割的原则是间隔最大化（距支持向量最远），最终转化为一个凸二次规划问题来求解。可分类为线性可分支持向量机（训练样本线性可分时，硬间隔最大化）、线性支持向量机（训练样本近似线性可分时，集合部分相交，软间隔最大化）、非线性支持向量机（训练样本线性不可分时，核技巧和软间隔最大化）。

线性可分支持向量机

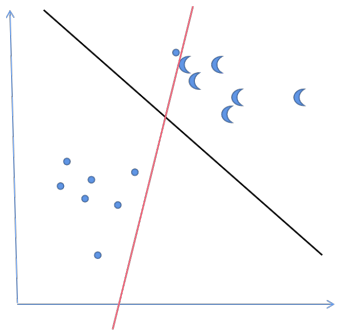


图 2.15: 线性可分支持向量机

上图中两种不同形状的点代表了正负样本，从图中这个二分类我们可知：最内侧的交点就是支持向量，它们确定出了两条虚线，我们称之为“支撑超平面”，两条虚线之间的线条都是可以把训练样本集完全分开的，它们之间的距离称为“间隔”(margin)，而它们“正中间”的位置就是我们要找的最优划分超平面的位置。

分离超平面：$y(\boldsymbol{x}) = \boldsymbol{w}^T\phi(\boldsymbol{x}) + b$，其中 $\phi(\boldsymbol{x})$ 是某个确定的特征空间转换函数，它的作用是将 $\boldsymbol{x}$ 映射到更高的维度，最简单 $\phi(\boldsymbol{x}) = \boldsymbol{x}$，已知的某种特征变换。

分类决策函数为：$f(\boldsymbol{x}) = \text{sign}\left(\boldsymbol{w}^T\phi(\boldsymbol{x}) + b\right)$

目标函数：

$$\arg\max_{\boldsymbol{x},b}\left\{\frac{1}{\|\boldsymbol{w}\|}\min_i\left[y_i\cdot\left(\boldsymbol{w}^T\phi(\boldsymbol{x})+b\right)\right]\right\}$$

即：

$$\arg\max_{\boldsymbol{x},b}\frac{1}{\|\boldsymbol{w}\|}$$

约束条件：

$$y_i\cdot\left(\boldsymbol{w}^T\phi(\boldsymbol{x})+b\right)\geq 1$$

估计：

$$\hat{\boldsymbol{w}} = \sum_{i=1}^{N}\alpha_i^* y_i\left(\phi\left(\boldsymbol{x}_i\right)\phi\left(\boldsymbol{x}_j\right)\right)$$

$$b^* = y_i - \sum_{i=1}^{N}\alpha_i^* y_i\left(\phi\left(\boldsymbol{x}_i\right)\phi\left(\boldsymbol{x}_j\right)\right)$$

推导过程：二维空间点到直线的距离（支持向量到超平面距离）

$$d = \frac{\left|\boldsymbol{w}^T\boldsymbol{x}+b\right|}{\|\boldsymbol{w}\|}, \|\boldsymbol{w}\| = \sqrt{{w_1}^2+\ldots+{w_n}^2}$$

$$\begin{cases}\frac{\left|\boldsymbol{w}^T\boldsymbol{x}+b\right|}{\|\boldsymbol{w}\|}\geq d, y=1\\ \frac{\left|\boldsymbol{w}^T\boldsymbol{x}+b\right|}{\|\boldsymbol{w}\|}< d, y=-1\end{cases}$$

其中分母为正，固定令其为 1，将两个方程合并。方便推导和优化，且这样做对目标函数的优化没有影响。也就是 $y\left(\boldsymbol{w}^T\boldsymbol{x}+b\right)\geq 1$，令等式左边 $y\left(\boldsymbol{w}^T\boldsymbol{x}+b\right) = \left|\boldsymbol{w}^T\boldsymbol{x}+b\right|$。

就可以得到最大间隔超平面的上下两个超平面，则上述的距离 d 可以写成 $d=\frac{y\left(\boldsymbol{w}^T\boldsymbol{x}+b\right)}{\|\boldsymbol{w}\|}$（最大化这个距离），即：$\min(f(\boldsymbol{w}))=\min\frac{1}{2}\|\boldsymbol{w}\|^2$ s.t. $g_i(\boldsymbol{w})=1-y_i\left(\boldsymbol{w}^T\boldsymbol{x}_i+b\right)\leq 0$

不等式的约束优化问题通过引入松弛变量使得 $h_i\left(\boldsymbol{w},a_i\right)=g_i(\boldsymbol{w})+a_i^2=0$ 转换为等式约束。满足 KKT 条件，是强对偶性的充要条件（原始问题与对偶问题的解完全等价），下式中的 L 是拉格朗日函数，λ 为等式约束的拉格朗日乘子。

$$\begin{cases}\frac{\partial L}{\partial w_i}=\frac{\partial f}{\partial w_i}+\sum_{j=1}^{n}\lambda_j\frac{\partial g_i}{\partial w_i}=0\\ \lambda_i g_i(\boldsymbol{w})=0\\ g_i(\boldsymbol{w})\leq 0\\ \lambda_i\geq 0\end{cases}$$

强对偶性转化：若凸优化问题强对偶性成立，则 $\min\max f=\max\min f$。且对偶问题将原始问题中的约束转为了对偶问题中的等式约束，对偶问题往往更加容易求解。无论原始问题是否是凸的，对偶问题都是凸优化问题。

拉格朗日函数为:

$$\min_{\boldsymbol{w},b}\max_{\lambda}L(\boldsymbol{w},b,\lambda)=\frac{1}{2}\|\boldsymbol{w}\|^2+\sum_{i=1}^{n}\lambda_i\left[1-y_i\left(\boldsymbol{w}^T\boldsymbol{x}_i+b\right)\right],s.t.\lambda_i\geq 0$$

利用强对偶性性质，可得：

$$\max_{\lambda}\min_{\boldsymbol{w},b}L(\boldsymbol{w},b,\lambda)$$

继而对参数 $\boldsymbol{w}$ 和 b 求偏导数：

$$\begin{array}{ll}\frac{\partial L}{\partial \boldsymbol{w}}=\boldsymbol{w}-\sum_{i=1}^{n}\lambda_i\boldsymbol{x}_i y_i=0 & \sum_{i=1}^{n}\lambda_i\boldsymbol{x}_i y_i=\boldsymbol{w}\\ \frac{\partial L}{\partial b}=\sum_{i=1}^{n}\lambda_i y_i=0 & \sum_{i=1}^{n}\lambda_i y_i=0\end{array}$$

将这个结果带回到函数中可得：

$$L(\boldsymbol{w},b,\lambda)=\frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{n}\lambda_i\lambda_j y_i y_j\left(\boldsymbol{x}_i\cdot\boldsymbol{x}_j\right)+\sum_{i=1}^{n}\lambda_i-\sum_{i=1}^{n}\lambda_i y_i\left(\sum_{j=1}^{n}\lambda_j y_j\left(\boldsymbol{x}_i\cdot\boldsymbol{x}_j\right)+b\right)$$

$=\frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{n}\lambda_i\lambda_j y_i y_j\left(\boldsymbol{x}_i\cdot\boldsymbol{x}_j\right)+\sum_{i=1}^{n}\lambda_i-\sum_{i=1}^{m}\sum_{j=1}^{n}\lambda_i\lambda_j y_i y_j\left(\boldsymbol{x}_i\cdot\boldsymbol{x}_j\right)-\sum_{i=1}^{n}\lambda_i y_i b$
$=\sum_{j=1}^{n}\lambda_j-\frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{n}\lambda_i\lambda_j y_i y_j\left(\boldsymbol{x}_i\cdot\boldsymbol{x}_j\right)$
即：
$\min_{\boldsymbol{w},b}L(\boldsymbol{w},b,\lambda)=\sum_{j=1}^{n}\lambda_j-\frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{n}\lambda_i\lambda_j y_i y_j\left(\boldsymbol{x}_i\cdot\boldsymbol{x}_j\right)$

$\max_{\lambda}\left[\sum_{j=1}^{n}\lambda_j-\frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{n}\lambda_i\lambda_j y_i y_j\left(\boldsymbol{x}_i\cdot\boldsymbol{x}_j\right)\right]\quad s.t.\sum_{i=1}^{n}\lambda_i y_i=0,\lambda_i\geq 0$

二次规划问题，问题规模正比于训练样本数，我们常用 SMO 算法求解。SMO 算法是序列最小优化算法，其核心思想是每次只优化一个参数，其他参数先固定住，仅求当前这个优化参数的极值。将 N 个解问题，转换成两个变量的求解问题，并且目标函数是凸的。

SMO 算法每次只优化一个参数，但我们的优化目标有约束条件，无法一次只变动一个参数。所以我们选择了一次变动两个参数，其他变量为定值，在满足约束条件下考虑同号或异号是否会

使得目标函数变小，不断随机选两个参数，两两求值，最后回代到上面可求得 $\boldsymbol{w}$，进而可得到 b（大多数的样本不会影响到分界面的确定，支持向量确定，稀疏模型）。

线性支持向量机

样本数据本身线性不可分，不一定分类完全正确的超平面就是最好的。牺牲个别样本使得模型泛化能力较好也是可行的。

软间隔：允许个别样本点出现在间隔带里面，即部分样本点不满足约束条件 $1-y_i\left(\boldsymbol{w}^T\boldsymbol{x}_i+b\right)\leq 0$。为了度量这个间隔软到何种程度，增加了松弛因子 ξ 这个变量。

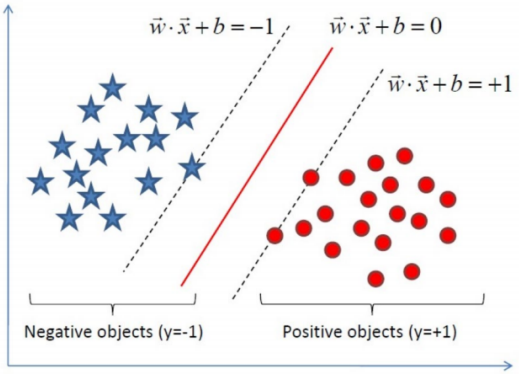


图 2.16: 线性支持向量机

增加松弛因子，约束条件：

$$y_i\left(\boldsymbol{w}^T\boldsymbol{x}_i+b\right)\geq 1-\xi_i$$

目标函数：

$$\min_{\boldsymbol{w},b,\zeta}\frac{1}{2}\|\boldsymbol{w}\|^2+C\sum_{i=1}^{n}\xi_i$$

s.t. $y_i\left(\boldsymbol{w}^T\boldsymbol{x}_i+b\right)\geq 1-\xi_i;\ i=1,2,\ldots,n;\ \xi_i\geq 0, i=1,2,\ldots,n$

其中 C 是一个大于 0 的常数，可以理解为错误样本的惩罚程度，若 C 为无穷大，松弛因子必然无穷小，如此一来线性 SVM 就又变成了线性可分 SVM；当 C 为有限值的时候，才会允许部分样本不遵循约束条件。

拉格朗日函数：

$$L(\boldsymbol{w},b,\xi,\alpha,\mu)=\frac{1}{2}\|\boldsymbol{w}\|^2+C\sum_{i=1}^{n}\xi_i-\sum_{i=1}^{n}\alpha_i\left(y_i\left(\boldsymbol{w}^T\cdot\boldsymbol{x}_i+b\right)-1+\xi_i\right)-\sum_{i=1}^{n}\mu_i\xi_i$$

乘子均大于等于 0，根据强对偶性，分别对主问题参数其求偏导并令其等于 0，回代消去 $\boldsymbol{w}$ 和 ξ_i，用 SMO 算法可得最终目标函数:

$$\min_{a}\frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n}\alpha_i\alpha_j y_i y_j\left(\boldsymbol{x}_i\cdot\boldsymbol{x}_j\right)-\sum_{i=1}^{n}\alpha_i$$

$$s.t.\sum_{i=1}^{n}\alpha_i y_i=0, 0\leq\alpha_i\leq C, i=1,2\ldots,n$$

计算:

$\boldsymbol{w}^* = \sum_{i=1}^{n} \alpha_i^* y_i \boldsymbol{x}_i$

$b^* = \frac{\max_{i:y_i=-1} \boldsymbol{w}^{*T}\boldsymbol{x}_i + \min_{i:y_i=1} \boldsymbol{w}^{*T}\boldsymbol{x}_i}{2}$

在间隔内的那部分样本点是支持向量。由求参数 $\boldsymbol{w}$ 的那个式子可看出，只要 $\lambda > 0$ 的点都能够影响我们的超平面，因此都是支持向量。

非线性支持向量机

硬间隔和软间隔都是在说样本的完全线性可分或者大部分样本点的线性可分。但我们可能会碰到的一种情况是样本点完全不是线性可分的 (如下)。这种情况的解决方法就是：将二维线性不可分样本映射到高维空间中，让样本点在高维空间线性可分。再通过间隔最大化的方式，学习得到支持向量机，就是非线性 SVM。

分割超平面:

$$f(\boldsymbol{x}) = \boldsymbol{w}^T \phi(\boldsymbol{x}) + b$$

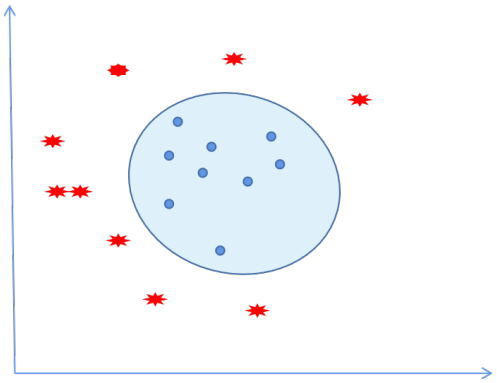


图 2.17: 非线性支持向量机

非线性 SVM 的对偶问题就变成了:

$$\min_{\lambda} \left[\frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{n} \lambda_i \lambda_j y_i y_j \left(\phi\left(\boldsymbol{x}_i\right) \cdot \phi\left(\boldsymbol{x}_j\right) \right) - \sum_{j=1}^{n} \lambda_i \right]$$

$$s.t. \quad \sum_{i=1}^{n} \lambda_i y_i = 0, \lambda_i \geq 0, C - \lambda_i - \mu_i = 0$$

可以看到与线性 SVM 唯一的不同就是：$\boldsymbol{x}$ 变成 $\phi(\boldsymbol{x})$，$\phi(\boldsymbol{x})$ 是特征空间转换函数。

核函数: 因为低维空间映射到高维空间后维度可能会很大，如果将全部样本的点乘全部计算好，这样的计算量太大了。如果在低维输入空间存在某个函数 $K(\boldsymbol{x}, \boldsymbol{x}^\intercal)$，它恰好等于在高维空间中的内积，即 $K(\boldsymbol{x}, \boldsymbol{x}^\intercal) = \langle \phi(\boldsymbol{x}), \phi(\boldsymbol{x}^\intercal) \rangle$，那么支持向量机就不用计算复杂的非线性变换，而由这个函数 $K(\boldsymbol{x}, \boldsymbol{x}^\intercal)$ 直接得到非线性变换的内积，使大大简化了计算。这样的函数称为核函数。

表 2.4: 常见的核函数

类型	形式
线性核函数	$K(\boldsymbol{x}_i, \boldsymbol{x}_j) = \boldsymbol{x}_i^T \boldsymbol{x}_j$
多项式核函数	$K(\boldsymbol{x}_i, \boldsymbol{x}_j) = \left(\boldsymbol{x}_i^T \boldsymbol{x}_j\right)^d$
高斯核函数	$K(\boldsymbol{x}_i, \boldsymbol{x}_j) = \exp\left(-\frac{\|\boldsymbol{x}_i - \boldsymbol{x}_j\|}{2\delta^2}\right)$

优点：(1) 有严格的数学理论支持，可解释性强，不依靠统计方法，从而简化了通常的分类和回归问题；(2) 能找出对任务至关重要的关键样本（即支持向量）；(3) 采用核技巧之后，可以处理非线性分类/回归任务；(4) 最终决策函数只由少数的支持向量所确定，计算的复杂性取决于支持向量的数目，而不是样本空间的维数，这在某种意义上避免了“维数灾难”。

缺点：训练时间长。当采用 SMO 算法时，由于每次都需要挑选一对参数，因此时间复杂度为 O(N^2)，其中 N 为训练样本的数量；当采用核技巧时，如果需要存储核矩阵，则空间复杂度为 O(N^2)；模型预测时，预测时间与支持向量的个数成正比。当支持向量的数量较大时，预测计算复杂度较高。

因此支持向量机目前只适合小批量样本的任务，无法适应百万甚至上亿样本的任务。SVM 三层的稀疏的神经网络，很像 CNN，工程实践上调参很麻烦，很少用。

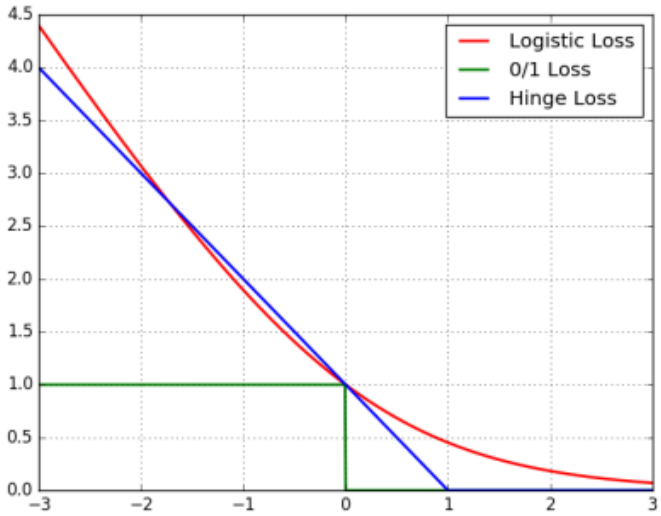


图 2.18: 损失函数比较

损失函数分析：绿色是 0-1 损失函数，以正样本为例，样本到直线距离为正说明分类正确没有损失，样本到直线距离为负需要惩罚 +1。蓝色是 SVM Hinge 损失函数，直线距离为负但距离远近程度不同，根据远近选择惩罚大小（60 分以下惩罚，做对的缓冲区内部惩罚小）深度学习经常自定义损失函数，在损失函数基础上加 1。红色是 Logistic 损失函数，对数似然函数取负号。

案例

利用不同核函数的非线性 SVM 进行分类，本案例是根据 bipartition 数据两门考试预测某一学生能否被录取。首先导入绘图和数据分析模块，以及 SVM 的模块函数。

```
# 导入作图
import matplotlib as mpl
import matplotlib.colors
import matplotlib.pyplot as plt
# 导入数据分析
import numpy as np
import pandas as pd
# 导入准确率和支持向量机算法
from sklearn import svm
from sklearn.metrics import accuracy_score
```

读入数据并分离特征和响应变量，构造参数不同的线性和高斯核分类器，生成网格，设置样本点颜色。

```
if __name__ == "__main__":
    data = pd.read_csv('bipartition.txt', sep='\t', header=None)
    x, y = data[[0, 1]], data[2]    # 读入数据，分离特征和响应变量
    # 构造参数不同的线性和高斯核分类器
    clf_param = (('linear', 0.1), ('linear', 0.5),
                 ('linear', 1), ('linear', 2),
                 ('rbf', 1, 0.1), ('rbf', 1, 1),
                 ('rbf', 1, 10), ('rbf', 1, 100),
                 ('rbf', 5, 0.1), ('rbf', 5, 1),
                 ('rbf', 5, 10), ('rbf', 5, 100))
    # 生成网格大小和间隔
    x1_min, x2_min = np.min(x, axis=0)
    x1_max, x2_max = np.max(x, axis=0) # 返回每列最小值和最大值
    x1, x2 = np.mgrid[x1_min:x1_max:200j, x2_min:x2_max:200j]
    # 增加列维度
    grid_test = np.stack((x1.flat, x2.flat), axis=1)
    cm_light = mpl.colors.ListedColormap(['#77E0A0',
    '#FFA0A0'])  # 颜色根据色板列表循环
    cm_dark = mpl.colors.ListedColormap(['g', 'r'])
    mpl.rcParams['font.sans-serif'] = ['SimHei']  # 字体
    mpl.rcParams['axes.unicode_minus'] = False
    # 字符显示(解决matplotlib不能读取中文的问题)
```

根据不同核函数进行建模，拟合并预测数据，输出模型结果。

```
    plt.figure(figsize=(13, 9), facecolor='w')
    # 设置子图大小和颜色,默认惩罚项C为1，考虑线性和高斯核
    for i, param in enumerate(clf_param):
        clf = svm.SVC(C=param[1], kernel=param[0])
        # 不同惩罚参数和核函数的非线性SVM
        if param[0] == 'rbf':
            clf.gamma = param[2]
            # 选取核函数参数并格式化标题
```

```
            title = '高斯核, C=%.1f, $\gamma$ =%.1f' % (param[1], param[2])
        else:
            title = '线性核, C=%.1f' % param[1]
        clf.fit(x, y)
        y_hat = clf.predict(x)      # 预测数据
        print('准确率: ', accuracy_score(y, y_hat))
        print('支撑向量的数目: ', clf.n_support_)
        print('支撑向量的系数: ', clf.dual_coef_)
        print('支撑向量: ', clf.support_)
```

最后进行绘图，对图形参数进行设置，将分类结果显示在图中，包括样本点、支撑向量、间隔平面等等。

```
        # 画图
        plt.subplot(3, 4, i + 1)
        grid_hat = clf.predict(grid_test)      # 预测分类值
        grid_hat = grid_hat.reshape(x1.shape) # 使之与输入形状相同
        plt.pcolormesh(x1, x2, grid_hat, cmap=cm_light, alpha=0.8)
        plt.scatter(x[0], x[1], c=y, edgecolors='k', s=40, cmap=cm_dark)
        plt.scatter(x.loc[clf.support_, 0], x.loc[clf.support_, 1], edgecolors='
            k', facecolors='none', s=100,marker='o') # 支撑向量
        z = clf.decision_function(grid_test)
        # 计算样本点到分割超平面的函数距离
        print('clf.decision_function(x) = ', clf.decision_function(x))
        print('clf.predict(x) = ', clf.predict(x))
        z = z.reshape(x1.shape)
        plt.contour(x1, x2, z, colors=list('kgrgk'),
            linestyles=['--', ':', '-', ':', '--'],
        linewidths=1, levels=[-1, -0.5, 0, 0.5, 1])
        plt.xlim(x1_min, x1_max)       # 设置坐标刻度
        plt.ylim(x2_min, x2_max)
        plt.title(title, fontsize=12)  # 设置标题和大小
    plt.suptitle('SVM不同参数的分类', fontsize=16)
    plt.tight_layout(1.4)              # 自动调整子图参数
    plt.subplots_adjust(top=0.92)      # 子图所在区域边界
    plt.show()
```

可得高斯核，C=5，γ=100 时的准确率为 0.99，支持向量的数目两边分别为 35 和 46。支持向量的系数等结果如下：

```
准确率： 0.99
高斯核，C=5.0，$\gamma$ =100.0
支撑向量的数目： [35 46]
支撑向量的系数： [[-0.37996631 -0.63758776 -0.97821702 -0.54787985 -1.10309485 -1.16619903
  -0.77535815 -0.44584005 -1.18762759 -0.88981698 -0.4826347  -5.
  -1.53376835 -1.09321594 -0.91839327 -0.75396033 -1.22397785 -1.14717885
  -0.77722647 -1.05849146 -0.59380046 -1.5506366  -0.30339944 -1.5025815
  -0.31356106 -0.48455412 -0.97433445 -0.45832485 -0.58657481 -1.16690615
  -0.84367518 -0.34625116 -0.6590926  -1.18824998 -0.6573527   0.58799858
   0.72657002  0.57884937  0.45552366  0.61337031  1.37935807  0.80157168
   0.77350957  0.5810975   0.40395773  0.72050683  0.40130284  0.27107101
   0.59360366  0.41414292  0.71050237  0.80721885  1.26219598  0.56180412
   0.6412647   0.71579661  0.55690843  0.70083421  0.56419538  0.69469257
   0.26287415  0.79671942  0.7116272   0.80202831  0.48117176  0.95406245
   0.81174731  0.78410765  0.59371553  5.          0.63332535  0.62574972
   0.43642453  0.08554003  0.7075916   0.71665342  0.40642796  0.33809572
   0.53659067  0.62968282  0.8977553 ]]
支撑向量： [ 4  5  6  8  9 11 12 13 14 17 19 20 22 23 25 26 27 29 30 31 32 33 34 35
 36 39 40 41 42 43 45 46 47 48 49 50 52 53 54 55 56 57 58 59 60 61 62 63
 64 65 67 68 69 70 71 72 74 75 76 77 78 79 80 81 82 83 84 85 87 88 89 90
```

图 2.19: SVM 不同参数的分类准确率类

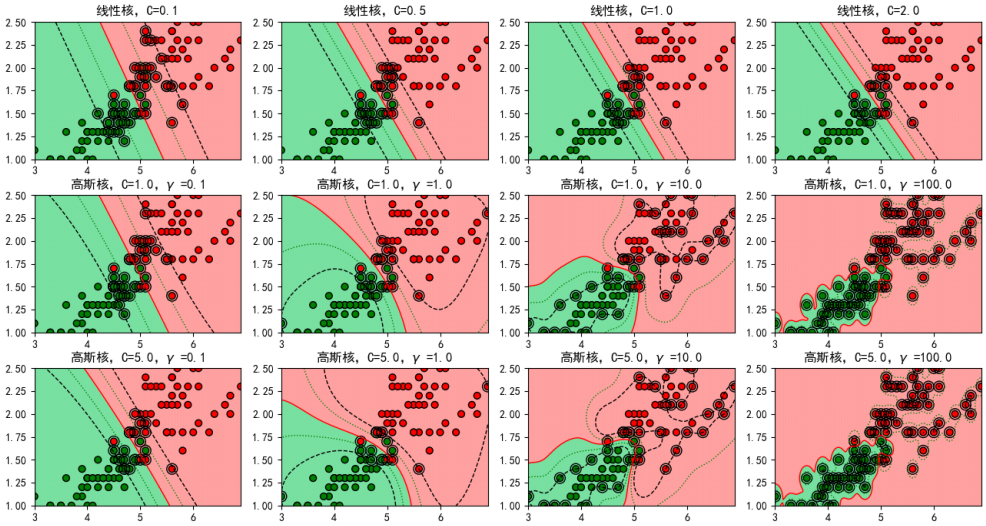


图 2.20: SVM 不同参数的分类

上述结果可得 C=5，γ=100 的高斯核与 C=1，γ=100 的高斯核的准确率较高为 0.99。故而最佳的参数范围是 C 较大 gamma 较小；或 C 较小 gamma 较大。如果希望得到的边界复杂多样，应该运用高斯核函数，如果想计算快，强调效率就用线性核函数。此外，支持向量 svm.svc 中 C 为目标函数的惩罚系数，用来平衡分类间隔 margin 和错分样本的，默认为 1.0；kernel 是参数，可供选择的有 RBF, Linear, Poly, Sigmoid，默认的是 “RBF”; 正则化参数 C 表示模型对误差的惩罚系数，gamma 反映了数据映射到高维特征空间后的分布；C 越大，模型越容易过拟合；C 越小，模型越容易欠拟合。核函数参数 gamma 越大支持向量越多，理论上 SVM 可以拟合任何非线性数据；gamma 值越小支持向量越少，模型的泛化性变好，但过小模型实际上会退化为线性模型。

第 3 章　卷积神经网络

3.1　神经网络

3.1.1　什么是神经网络

人工神经网络（Artificial Neural Network, ANN）是模仿生物神经网络结构和功能的数学模型，它通过大量的人工神经元联结进行计算，通常简称为神经网络。

神经网络一般的结构如图3.1所示，包含输入层、隐藏层和输出层等。

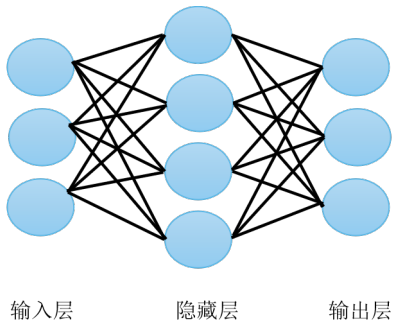


图 3.1: 神经网络一般结构

为便于后续分析，记输入层为 $\boldsymbol{x} = (x_1, x_2, x_3)^T$，隐藏层为 $\boldsymbol{z} = (z_1, z_2, z_3, z_4)^T$，输出层为 $\boldsymbol{y} = (y_1, y_2, y_3)^T$，此时有

$$\boldsymbol{z} = \boldsymbol{w}_1 * \boldsymbol{x} + \boldsymbol{b}_1$$

$$\boldsymbol{y} = \boldsymbol{w}_2 * \boldsymbol{z} + \boldsymbol{b}_2$$

其中，$\boldsymbol{w}_1$ 为四行三列的矩阵，$\boldsymbol{w}_2$ 为三行四列的矩阵，$\boldsymbol{b}_1$ 为四行一列的列向量、$\boldsymbol{b}_2$ 为三行一列的列向量，它们都是神经网络中的参数。

一系列线性方程的运算最终都可以用一个线性方程表示。也就是说，上述两个式子联立后可以用一个线性方程表达。即：$\boldsymbol{y} = \boldsymbol{w}_2 * (\boldsymbol{w}_1 * \boldsymbol{x} + \boldsymbol{b}_1) + \boldsymbol{b}_2 = \boldsymbol{w}_2 * \boldsymbol{w}_1 \boldsymbol{x} + \boldsymbol{w}_2 * \boldsymbol{b}_1 + \boldsymbol{b}_2$。两层神经网络如此，增加神经网络的隐藏层层数依旧如此，此时神经网络所发挥的作用相当有限，仅仅实现了数据的线性变换。

3.1.2 激活函数

激活函数是在神经网络的神经元上运行的函数，负责将神经元的输入映射到输出。激活函数的引入将增强神经网络模型的非线性、稳定性和泛化能力。常见的激活函数有 **Sigmoid** 函数，**Tanh** 函数，**ReLU** 函数和 **Leaky ReLU** 函数等。

1.**Sigmoid** 激活函数

Sigmoid 函数的数学表达式为：

$$y = \frac{1}{1 + e^{-x}}$$

其几何图像如下所示：

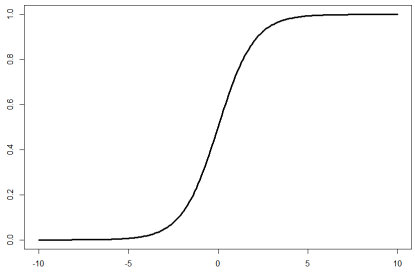


图 3.2: Sigmoid 函数

Sigmoid 函数能够将输入的连续实值变换为 0 和 1 之间的输出，这样的结果很容易充当下一层的输入，常用于二分类问题中。不过也存在以下缺点：(1) **Sigmoid** 函数的输出不是零均值，导致后续进行参数更新时只能朝一个方向更新，影响梯度下降的动态性；(2) 在应用梯度下降法过程中有可能出现梯度消失现象；(3) 从 **Sigmoid** 函数的表达式可以看出，在运算过程中含有幂次运算，计算复杂度高。

2.**Tanh** 激活函数

Tanh 函数的数学表达式为：

$$y = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

其几何图像如下所示：

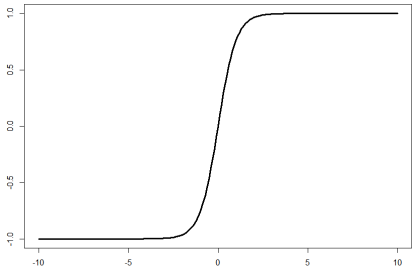


图 3.3: Tanh 函数

Tanh 函数解决了 **Sigmoid** 函数的输出不是零均值输出的问题。同时，**Sigmoid** 函数与有 **Tanh** 函数密切联系，若记 $f(x) = \frac{1}{1+e^{-x}}$，此时有 Tanh(x)=$2f(2x) - 1$。

3.**ReLU** 激活函数

ReLU 函数的数学表达式为：$y = \max(0, x)$

其几何图像如下所示：

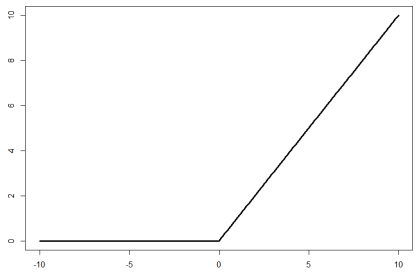


图 3.4: ReLU 函数

ReLU 函数求导方便，计算速度快，优化算法的收敛速度也远快于 **Sigmoid** 函数和 **Tanh** 函数，同时在正区间上解决了梯度消失的问题。然而 **ReLU** 函数的输出不是 0 均值，同时可能出现 "$Dead\ ReLU\ Problem$"（神经元坏死现象），即 **ReLU** 函数在训练的时很 "脆弱"。在 $x < 0$ 时，梯度为 0，这个神经元及之后的神经元梯度永远为 0，不再对任何数据有所响应，导致相应参数永远不会被更新。但这并不影响 **ReLU** 函数是使用最为广泛的激活函数。

4.**Leaky ReLU** 激活函数

Leaky ReLU 函数的数学表达式为：$y = \max(\alpha x, x)$，这里 α 为常数，一般设置 0.01。

其几何图像如下所示：

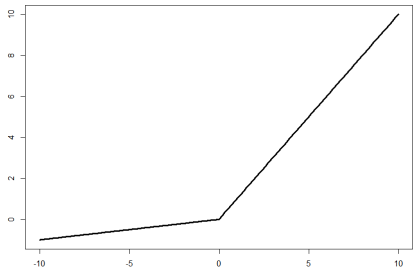


图 3.5: Leaky ReLU 函数

为解决 **ReLU** 函数中某些神经元可能永远不会被激活的问题，考虑将 **ReLU** 函数的前半段设为 αx 而非 0，使得 **ReLU** 函数在负数区域更偏向于激活而不是死掉。

近些年来又陆续出现了更多的激活函数。如谷歌在 2017 年提出的 **Swish** 激活函数，它几乎是 **Sigmoid** 函数和 **ReLU** 函数的拼凑，具备无上界有下界、平滑、非单调的特性，性能在总体上优于 **ReLU** 函数。还有结合 **ReLU** 函数和 **Tanh** 函数得到的 **Mish** 函数等等。

整体来看，不同的激活函数，特点和应用大相径庭，在操作中需要根据所解决的实际问题恰当选择合适的激活函数。

3.1.3 损失函数

神经网络在加入激活函数后，从输入到输出的运算可以顺利完成。然而神经网络的好坏如何衡量？接下来介绍损失函数的相关内容。

通过神经网络将目标 (实际) 值与预测值进行匹配，再经过定义好的损失函数就可以计算出损失。常见的损失函数有均方误差损失函数和交叉熵损失函数等。

（一）均方误差损失函数

$$J = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

其中，n 是样本个数，y_i 为真实值，$\hat{y}_i$ 为预测值，J 通过计算均方误差损失来衡量模型的好坏。均方误差损失函数通常用于回归问题的分析中。

（二）交叉熵损失函数

交叉熵是信息论中的一个重要概念，能够衡量同一个随机变量中的两个不同概率分布的差异程度，在神经网络中就表示为真实概率分布与预测概率分布之间的差异。交叉熵的值越小，模型预测效果就越好。

信息论奠基人香农认为“信息是用来消除随机不确定性的东西”，也就是说信息量的大小取决于该信息消除不确定性的程度。通常来说，信息量的大小与信息发生的概率成反比。发生的概率越大，信息量越小，反之越大。

假设某一时间发生的概率为 $p(x)$，其信息量为

$$I(x) = -\log(p(x))$$

其中 $I(x)$ 表示信息量。

所有信息量的期望 $H(x)$ 称为信息熵，定义如下：

$$H(x) = -\sum_{i=1}^{n} p(x_i)\log(p(x_i))$$

如果随机变量 X 有两个单独的概率分布 $p(x)$ 和 $q(x)$，我们使用 KL 散度来衡量两个概率分布之间的差异。

KL 散度定义如下：

$$D_{KL} = \sum_{i=1}^{n} p(x_i)\log(\frac{p(x_i)}{q(x_i)})$$

其中 n 为样本个数。

上式可写成如下形式：

$$\begin{aligned} D_{KL} &= \sum_{i=1}^{n} p(x_i)\log(p(x_i)) - \sum_{i=1}^{n} p(x_i)\log(q(x_i)) \\ &= H(p(x)) - \sum_{i=1}^{n} p(x_i)\log(q_(x_i)) \end{aligned}$$

不难看出，第一项为信息熵，第二项定义为交叉熵，其形式如下：

$$J(p,q) = -\sum_{i=1}^{n} p(x_i)\log(q(x_i))$$

其中 n 为样本个数，$p(x)$ 和 $q(x)$ 为随机变量 X 两个单独的概率分布。

3.1.4 梯度下降法

损失函数的建立有助于优化神经网络的参数，我们的目标是通过优化神经网络的参数来最大程度地减少神经网络的损失，接下来需要解决的问题就是如何得到损失函数的最小值。神经网络中常用到的是梯度下降法，主要包括批量梯度下降法（Batch Gradient Descent，GD）、随机梯度下降法（Stochastic Gradient Descent，SGD）和小批量梯度下降法（Mini-batch Gradient Descent）。具体信息可参照第二章第 2 节的相关内容。

epoch、batch 和 iteration 是梯度下降法中常用到的参数：epoch 指的循环次数，epoch=10 指的是把整个数据集训练 10 次；batch size 指的是数据的个数，batch size=10 指的是每次训练 10 个数据；iteration 指的是次数，iteration=10 指的是把整个数据集分成 10 次代入神经网络中进行训练。

例如，有 100 个训练数据，epoch = 10, batch size = 5, iteration = 100/5 = 20 ，即要把 100 个数据集扔进神经网络训练 10 次，每次（每个 epoch）把 100 个数据集分成 20 份，每份数据为 5 个（batch size=5），所以需要投入 20 次（iteration）来完成一个 epoch，一个 epoch 中模型参数更新次数为 20。

需要注意的是，在网络训练次数得到 epoch 设定的数值后，损失函数的值不一定是最小的，所以在测试代码时需要更改 epoch 的数值，以确定损失函数在 epoch 处于何值时损失函数值基本不变。通常来说，刚开始损失函数值会下降很快，如果搭建的网络合适的话，损失函数值会下降然后趋于不变，此时 epoch 对应的数值不需要再变动。

3.1.5 Back Propagation（BP）算法：前向和反向传播

BP 算法由输入信息的前向传播和误差的反向传播两个过程组成。前向传播的主要要素包括加权和、激活函数、输出函数、误差；反向传播的主要要素包括误差偏导（链式法则）、输出函数到激活函数的偏导、激活层到加权和的偏导。

下面结合二分类问题简要说明 BP 算法的应用过程。这里激活函数选择 **Sigmoid** 函数，损失函数选择交叉熵损失函数。

1.Logistic 回归（隐藏层为一层的神经网络）

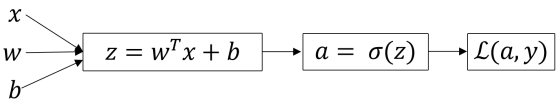


图 3.6: Logistic 回归

其中，x 是输入变量，w 和 b 是模型参数，$a = \sigma(z)$ 是激活函数，且 $\sigma'(z) = \sigma(z)(1-\sigma(z))$。损失函数 $\mathcal{L}(a, y) = -y\log(a) - (1-y)\log(1-a)$。计算得到损失函数后，前向传播结束，下面

考虑反向传播，主要是求解各变量偏导数，具体如下：

$$
\begin{aligned}
da &= -\frac{y}{a} + \frac{1-y}{1-a} \\
dz &= da \cdot \sigma'(z) \\
&= da \cdot \sigma(z) \cdot (1 - \sigma(z)) \\
&= a - y \\
dw &= dz \cdot x \\
db &= dz
\end{aligned}
$$

这里涉及到的均为单变量，因此偏导数即为导数，求解过程的主要工具为链式法则。计算结束后即可采用梯度下降法进行参数的更新，如此反复，前向反向传播不断进行。

2. 隐藏层为两层的神经网络

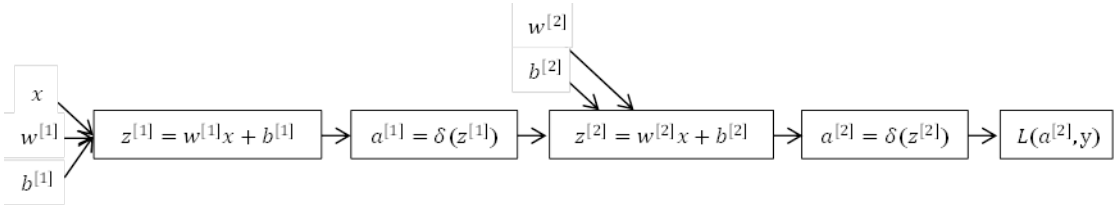


图 3.7: 两层隐藏层的神经网络

x，W 和 b 等含义与设置与（1）中 Logistic 回归相同，右上标中 [1，2] 表明该参数位于神经网络隐藏层中的第 1 层或第 2 层。接下来分别考虑隐藏层为两层的神经网络中的前向传播和反向传播。

（一）前向传播

$$
\begin{aligned}
z^{[1]} &= w^{[1]} * a^{[0]} + b^{[1]} \\
a^{[1]} &= \sigma(z^{[1]}) \\
z^{[2]} &= w^{[2]} * a^{[1]} + b^{[2]} \\
a^{[2]} &= \sigma(z^{[2]}) \\
\mathcal{L}(a^{[2]}, y) &= -y \log(a^{[2]}) - (1-y) \log(1 - a^{[2]})
\end{aligned}
$$

其中，$a^{[0]}$ 是神经网络的输入特征 x，这里为统一形式写为 $a^{[0]}$。

（二）反向传播

计算偏导数是反向传播过程中至为重要的一步，使用到的工具主要是链式法则。涉及到的变量主要包括 z，w，b 和 a。下面依次计算 $dz^{[2]}$，$dw^{[2]}$，$db^{[2]}$，$dz^{[1]}$，$dw^{[1]}$，$db^{[1]}$。由损失函数的定义不难得到：

$$
da^{[2]} = \frac{d}{da}\mathcal{L}(a, y) = -\frac{y}{a^{[2]}} + \frac{1-y}{1-a^{[2]}}
$$

根据前面激活函数的设置，$a = \sigma(z) = \frac{1}{1+e^{-z}}$，因此 $\sigma'(z) = \sigma(z)(1-\sigma(z))$。因此有

$$
\begin{aligned}
dz^{[2]} &= a^{[2]} - y \\
dw^{[2]} &= dz^{[2]} a^{[1]T} \\
db^{[2]} &= dz^{[2]} \\
da^{[1]} &= dz^{[2]} w^{[2]}
\end{aligned}
$$

同样地，我们可以得到如下三个等式：

$$
\begin{aligned}
dz^{[1]} &= dz^{[2]} w^{[2]T} * \sigma'(zz^{[1]}) \\
dw^{[1]} &= dz^{[1]} x^T \\
db^{[1]} &= dz^{[1]}
\end{aligned}
$$

确定相应的数值后，代入梯度下降法的公式，即可完成两层神经网络正向和反向的一次传播。

注意到，在计算梯度的过程中出现了转置运算。假设输入 x 维度为 $n^{[0]}$，隐藏层维度为 $n^{[1]}$，输出层维度为 $n^{[2]}$。因此 $w^{[2]}$ 维度为 $(n^{[2]}, n^{[1]})$，$W^{[1]}$ 维度为 $(n^{[1]}, n^{[0]})$，$z^{[2]}$ 维度为 $(n^{[2]}, 1)$，$z^{[1]}$ 维度为 $(n^{[1]}, 1)$。因此需要注意，在计算时需要用到某些向量的转置。

3.1.6 案例：鸢尾花分类

鸢尾花数据集是在统计学习和机器学习领域中经常被用作示例的经典数据集。该数据集内包含 3 类共 150 条记录，每类各 50 个数据，每条记录都有 4 项特征：花萼长度、花萼宽度、花瓣长度、花瓣宽度，可以通过这 4 个特征预测鸢尾花卉属于（Setosa，Versicolour，Virginica）中的哪一品种。

导入 numpy、tensorflow 库和鸢尾花数据集，数据包含 4 个特征变量：花萼长度、花萼宽度、花瓣长度、花瓣宽度，以及 3 个标签：Setosa，Versicolour，Virginica。

```
# 利用鸢尾花数据集，实现前向传播、反向传播，可视化loss曲线、acc曲线
import numpy as np
import tensorflow
from sklearn import datasets
from matplotlib import pyplot as plt
x_data = datasets.load_iris().data
y_data = datasets.load_iris().target
```

使用相同的 seed，保证输入特征和标签一一对应。shuffle 函数将序列的所有元素随机排序，避免原数据排序影响分类结果。将打乱后的数据集分割为训练集和测试集，训练集为前 120 行，测试集为后 30 行。

```
np.random.seed(415)
np.random.shuffle(x_data)
np.random.seed(415)
np.random.shuffle(y_data)
x_train = x_data[:-30]
y_train = y_data[:-30]
```

```
x_test = x_data[-30:]
y_test = y_data[-30:]
```

tensorflow.cast 是类型转换函数，否则后面矩阵相乘时会因数据类型不一致报错。并将输入特征和标签值一一对应，另外利用 batch() 把数据集分批次，每个批次 batch 组数据。

```
x_train = tensorflow.cast(x_train, tensorflow.float32)
x_test = tensorflow.cast(x_test, tensorflow.float32)
train_db = tensorflow.data.Dataset.from_tensor_slices((x_train, y_train)).batch
    (30)
test_db = tensorflow.data.Dataset.from_tensor_slices((x_test, y_test)).batch(30)
```

生成神经网络的参数，这里考虑隐藏层数只有一层的神经网络。设定学习率为 0.1，并将每轮的 loss 记录列表中，为后续画 loss 曲线提供数据。将每轮的 acc 记录在列表中，为后续画 acc 曲线提供数据。循环 300 轮，每轮分 4 个 step，并记录四个 step 生成的 4 个 loss 的和。

```
w1 = tensorflow.Variable(tensorflow.random.truncated_normal([4, 3], stddev=0.1,
    seed=1))
b1 = tensorflow.Variable(tensorflow.random.truncated_normal([3], stddev=0.1,
    seed=1))
alhpa = 0.1
train_loss_results = []
test_acc = []
epoch = 300
loss_all = 0
```

进入到训练部分。首先，先进行数据集级别的循环，每个 epoch 循环一次数据集，再进行 batch 级别的循环。接下来使用 with 结构记录梯度信息，在 with 结构中实现神经网络乘加运算，并使输出 y 符合概率分布，之后将标签值转换为独热码格式，方便计算 loss 和 accuracy。损失函数采用均方误差损失函数。将每个 step 计算出的 loss 累加，为后续求 loss 平均值提供数据，这样计算的 loss 更准确。计算 loss 对各个参数的梯度并进行梯度更新。每个 epoch，打印 loss 信息。接下来进行参数 w1 和参数 b 自更新。将 4 个 step 的 loss 求平均记录在变量中。总误差归零，为记录下一个 epoch 的 loss 做准备。

```
# 训练部分
for epoch in range(epoch):
    for step, (x_train, y_train) in enumerate(train_db):
        with tensorflow.GradientTape() as tape:
            y = tensorflow.matmul(x_train, w1) + b1
            y = tensorflow.nn.softmax(y)
            y_ = tensorflow.one_hot(y_train, depth=3)
            loss = tensorflow.reduce_mean(tensorflow.square(y_ - y))
            loss_all += loss.numpy()
        grads = tape.gradient(loss, [w1, b1])
        w1.assign_sub(alhpa * grads[0])
        b1.assign_sub(alhpa * grads[1])
    train_loss_results.append(loss_all / 4)
```

```
    loss_all = 0
```

进入到测试部分。将预测对的样本个数和测试的总样本数初始化为 0。使用更新后的参数进行预测，pred 返回 y 中最大值的索引，即预测的分类，并将 pred 转换为测试集的数据类型。若分类正确，则 correct=1，否则为 0，将 bool 型的结果转换为 int 型。并将每个 batch 的 correct 数加起来，将所有 batch 中的 correct 数加起来。总的准确率等于预测正确的个数除以总个数。

```
    #测试部分
    total_correct, total_number = 0, 0
    for x_test, y_test in test_db:
        y = tensorflow.matmul(x_test, w1) + b1
        y = tensorflow.nn.softmax(y)
        pred = tensorflow.argmax(y, axis=1)
        pred = tensorflow.cast(pred, dtype=y_test.dtype)
        correct = tensorflow.cast(tensorflow.equal(pred, y_test), dtype=
            tensorflow.int32)
        correct = tensorflow.reduce_sum(correct)
        total_correct += int(correct)
        total_number += x_test.shape[0]
    acc = total_correct / total_number
    test_acc.append(acc)
```

绘制 loss 曲线。

```
plt.title('Loss Function Curve')  # 图片标题
plt.xlabel('Epoch')  # x轴变量名称
plt.ylabel('Loss')  # y轴变量名称
plt.plot(train_loss_results, label="$Loss$")  #逐点画出trian_loss_results值并连
    线，连线图标是Loss
plt.legend()  # 画出曲线图标
plt.savefig('Loss Function Curve')
plt.show()  # 画出图像
```

绘制 Accuracy 曲线。

```
plt.title('Acc Curve')  # 图片标题
plt.xlabel('Epoch')  # x轴变量名称
plt.ylabel('Acc')  # y轴变量名称
plt.plot(test_acc, label="$Accuracy$")  # 逐点画出test_acc值并连线，连线图标是
    Accuracy
plt.legend()  # 画出曲线图标
plt.savefig('Acc Curve')
plt.show()  # 画出图像
```

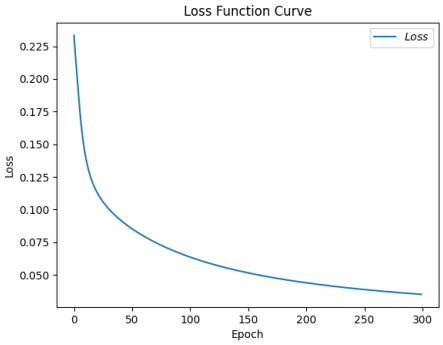


图 3.8: 损失函数

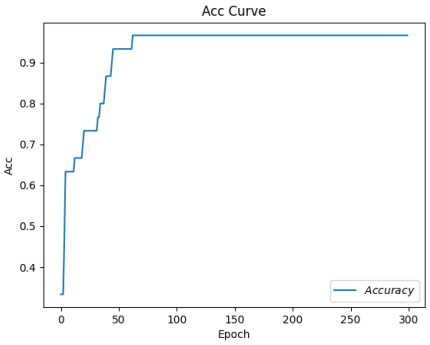


图 3.9: 正确率

在进行鸢尾花分类时，采用含有一层隐藏层的神经网络，主要通过 lstlisting 中 Tensorflow 相关的函数搭建网络实现鸢尾花的分类。从输出结果不难看出，分类的正确率非常不错，这印证了神经网络在分类领域的优越性能。在后续学习中，将继续介绍神经网络在图片分类等方面的广泛应用，主要介绍卷积神经网络。

3.2 卷积神经网络

3.2.1 卷积神经网络的建立

神经网络在得到广泛应用的同时，参数过多、容易发生过拟合和训练时间长等缺点也暴露出来。能否减少神经网络中参数的数目，并进一步提升神经网络的性能？卷积神经网络（Convolutional Neural Networks, CNN）应运而生。

CNN 最早可以追溯到 1986 年 BP（Back Propagation）算法的提出，1989 年 LeCun 将 BP 算法用到多层神经网络中，1998 年 LeCun 提出 LeNet-5 模型，卷积神经网络的雏形完成。在接下来近十年的时间里，卷积神经网络的相关研究趋于停滞，原因有两个：一是研究人员意识到多层神经网络在进行 BP 训练时的计算量极其之大，当时的硬件计算能力完全不可能实现；二是包括 SVM 在内的浅层机器学习算法也渐渐开始暂露头脚。直到 2006 年，Hinton 在 Science 发文，指出“多隐层神经网络具有更为优异的特征学习能力，并且其在训练上的复杂度可以通过逐层初始化来有效缓解”，DeepLearning（深度学习）开始觉醒，逐渐走入人们的视线。2009 年，李飞飞牵头建立了 ImageNet 数据集，该数据集包含图片的种类和数量远远超过以往所有的数据集。自 2010 年以来，每年都举行 ImageNet 大规模视觉识别挑战赛（ILSVRC），研究团队在给定的数据集上评估其算法，并在几项视觉识别任务中争夺更高的准确性。在刚开始的两年，冠军被 SVM（Support Vector Machines，支持向量机）取得，而在 2012 年，AlexNet 取得了比赛的冠军，识别的正确率得到明显提升，这之后占据高位的一直是卷积神经网络，包括我们现在所熟知的 VGGNet，GooLeNet，ResNet 等。2016 年，AlphaGo 战胜了围棋世界冠军、职业九段棋手李在石，更是将深度学习和卷积神经网络推向了高潮。

下图展示了近些年来 CNN 的发展历程。

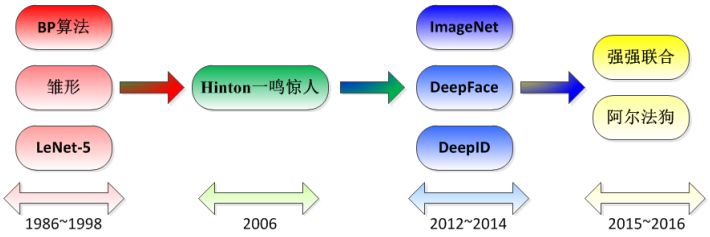


图 3.10: CNN 大事件

3.2.2 卷积和卷积神经网络

卷积 (Convolution) 运算有连续和离散两种定义，一维情形下有如下所示：

$$(f * g)(n) = \int_{-\infty}^{\infty} f(t)g(n-t)\,dt$$

$$(f * g)(n) = \sum_{t=-\infty}^{\infty} f(t)g(n-t)$$

从中可以发现，卷积运算先对 g 函数进行翻转，相当于在数轴上把 g 函数从右边折到左边去，也就是卷积中的“卷”。然后再把 g 函数平移到 n，在这个位置对两个函数的对应点相乘，然后相加，这个过程是卷积中的“积”。因此，所谓“卷积”，可直观理解为“卷”和“积”两个过程。

在卷积神经网络中“卷”和“积”如何体现呢？下面通过图（3.11）说明卷积神经网络的计算过程。

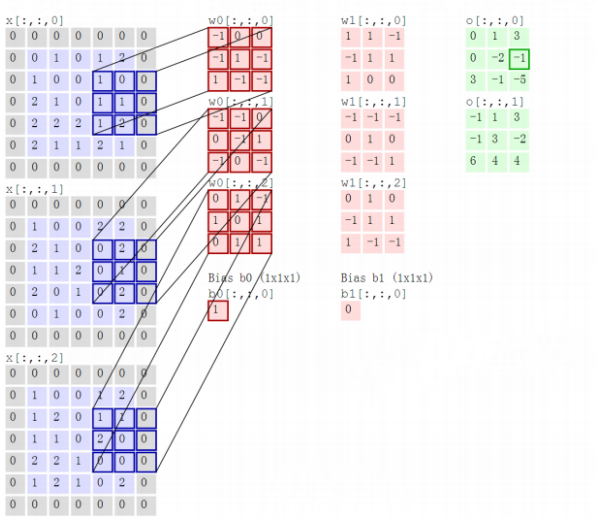


图 3.11: 卷积运算

最左侧为输入，中间红色部分表示卷积核，最右侧为输出。卷积核移动到输入左上角开始计算，以此向右、向下滑动。例如，第一个卷积核移动到第一个输入的左上角时，此时计算过程为：$0*(-1)+0*0+0*0+0*(-1)+0*1+1*(-1)+0*1+1*(-1)+0*(-1)=-2$。类似地，可以继续后面的运算。

通过计算的过程可以发现，“卷”并没有发挥作用，卷积神经网络仅仅应用了“积”。卷积神经网络具备两大特点：局部连接和权值共享。局部连接指的是卷积层的节点仅仅和其前一层的部分节点相连接；权值共享指的是同一张图使用相同的卷积核。局部连接和权值共享减少了参数数

量，加快了神经网络的学习速率，同时也在一定程度上减少了过拟合的可能。下面通过一个简单的例子来理解卷积神经网络如何节省参数。

假设输入为 100*100 的矩阵，最终的输出为 200*200 的矩阵。一般的神经网络若想达到此目的，分为如下两步完成。

1. 得到 200*100 的矩阵 Image（100,100）* w1（100,100）=y1

Image（100,100）* w2（100,100）=y2

...

Image（100,100）* w200（100,100）=y200

2. 得到 200*200 的矩阵

Image（100,100）* w1（200，100, 100）=y1（200）

Image（100,100）* w2（200，100, 100）=y2（200）

...

Image（100,100）* w200（200，100, 100）=y200（200）

→

Image（100,100）* w（200，200，100,100）=y（200,200）

在此过程中，参数数量为 200 * 200 * 100 * 100=400000000。而使用卷积神经网络，Image（100,100）* w（200，200，3，,3）=y（200,200），参数数量为 200 * 200 * 3 * 3=480000，充分利用卷积神经网络局部连接和权值共享的特点，极大地节省了参数。

3.2.3 基本参数

了解了卷积神经网络的基本过程后，接下来学习卷积神经网络中的几个基本参数。

（一）Padding

在进行卷积操作的时候，6*6 的图像经过 3*3 的 filter 卷积得到 4*4 的卷积结果，如图3.12所示。用更加一般的形式表达：如果我们有一个 n×n 的图像，用 f×f 的过滤器做卷积，那么输出的维度就是 (n-f+1)×(n-f+1)。在这个例子里是 6-3+1=4，因此得到了一个 4×4 的输出。

但是这样做存在明显缺点：每次卷积操作后图像尺寸都会缩小，同时角落边缘的像素点，在卷积计算的时候只被一个输出所触碰或者使用，但是中间的像素点会被多次卷积计算，所以那些在角落的像素点参与卷积计算较少，意味着丢掉一些图像边缘位置的信息。

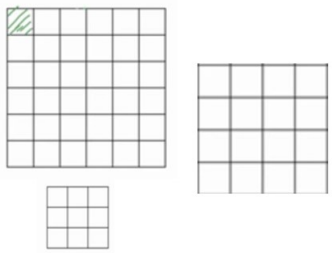


图 3.12: 图像由 6*6 到 3*3

卷积操作中的 padding，可以自行指定 p 的值。一般有如下两种 Padding 方式。（1）Valid：代表不填充，即 $P = 0$；（2）Same：代表填充后输出矩阵的大小与原矩阵保持一致，即 $P = \frac{f-1}{2}$

, 其中卷积核的大小为 fXf。通常将 f 设置为奇数，这样我们方便我们对称填充。

（二）Stride

滑动卷积核时，我们会先从输入的左上角开始，每次往左滑动一列或者往下滑动一行逐一计算输出，我们将每次滑动的行数和列数称为 Stride。

（三）Pooling

池化是实现下采样的一种运算，能在提取图像关键特征的基础上减小图片尺寸，以减少训练中的参数数量，达到减少计算量、增大感受野并防止过拟合的目的。

池化主要有最大值池化 (Max-Pooling) 和平均值池化 (Average-Pooling) 两种。最大值池化从所选区域的矩阵元素中取最大值，而平均值池化是将所选区域的矩阵元素求和后取平均值。

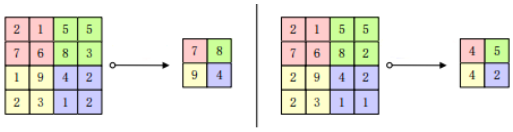


图 3.13: 最大值池化和平均值池化示意图

（四）欠拟合和过拟合

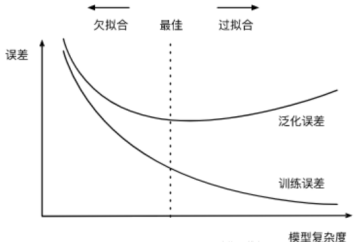


图 3.14: 欠拟合和过拟合

1. 欠拟合

欠拟合是指模型拟合程度不高，数据距离拟合曲线较远，或指模型没有很好地捕捉到数据特征，不能够很好地拟合数据。解决办法是提高模型的复杂度，如增加神经网络的层数、宽度；减少正则化的参数等。

2. 过拟合

一个假设在训练数据上能够获得比其他假设更好的拟合，但是在训练数据外的数据集上却不能很好地拟合数据，表现为泛化能力差。导致过拟合的原因有很多，主要包括以下几点：模型过于复杂、参数过多；数据太少；训练集和测试集的数据分布不同；样本里的噪音数据干扰过大，大到模型过分记住了噪音特征，反而忽略了真实的输入输出间的关系。解决办法包括 L_1、L_2 正则化；Dropout；扩增数据；Early Stopping。

（1）L_1、L_2 正则化

在损失函数上添加正则化项，其中 L_1 正则化为参数 w 的绝对值（岭回归）、L_2 正则化为参数 w 的平方值（Lasso 回归）。通过对 w 值的修正，使其偏离不会太大，从而减少过拟合的产生。

需要注意的是，这里仅考虑参数 w，不考虑偏移量 b。因为在实际操作中不难发现，模型的过拟合是由 w 引起的，偏移量 b 维度低，影响有限。

（2）扩增数据

更多的数据集，能够让搭建的网络在更多的数据中不断修正调整，进而训练出更好的模型。然而数据量都是有限的，所以可以从已有数据出发，对其进行调整以得到更多的数据集。例如可以针对已有图像应用随机图像转换，如旋转、对称和放大等，来增加图像数量。

图 3.15: 原图

图 3.16: 翻转、镜面、缩放变换

（3）Dropout

不同于 L_1、L_2 正则化通过修改代价函数防止过拟合，Dropout 修改的对象是神经网络。其思想主要是随机使得神经网络隐藏层中的一些神经元失活，在简化网络结构的同时防止了过拟合，而且随机失活迫使每一个神经元学习到有效的特征，增强了搭建网络的泛化能力。

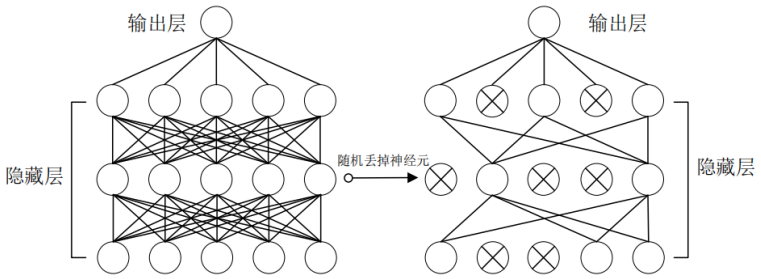


图 3.17: Dropout

在神经网络中使用 Dropout 时，一般以概率 0.5 选择神经元是否有效，即每层仅留下半数的神经元。因此使用 Dropout 相当于训练了多个只有半数隐层神经元的神经网络，每一个这样的网络，都可以给出一个分类结果，这些结果有对有错。随着训练的进行，大部分网络可以给出正确的分类结果，此时少数的错误分类结果不会对最终结果造成大的影响。

（4）Early Stopping

当训练有足够的表示能力甚至会过拟合的大模型时，我们经常观察到训练误差会随着时间的推移逐渐降低但验证集的误差会再次上升。这意味着如果我们返回使验证集误差最低的参数设置，就可以获得更好的模型（有希望获得更好的测试误差）。在每次验证集误差有所改善后，我们存储模型参数的副本。当训练算法终止时，我们返回这些参数而不是最新的参数。当验证集上的误差在事先指定的循环次数内没有进一步改善时，算法就会终止。需要注意的是，通过提前终止自动选择超参数的显著代价是训练期间要定期评估验证集。

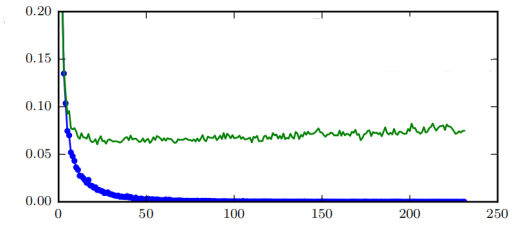


图 3.18: Early Stopping

图中横轴为 epoch 次数，纵轴为损失。蓝线为训练集上的损失，随着 epoch 次数的不断增加明显减少，绿线为验证集上的误差。上述策略称为提前终止（Early Stopping）。这是深度学习中最常用的正则化形式，它的流行主要是因为有效性和简单性。

3.2.4 卷积神经网络的一般结构

在卷积神经网络中，输入图像通过多个卷积层和池化层进行特征提取，逐步由低层特征变为高层特征；高层特征再经过全连接层和输出层进行特征分类，产生一维向量，表示当前输入图像的类别。因此，根据每层的功能，卷积神经网络可以划分为两个部分：由输入层、卷积层和池化层构成特征提取器，以及由全连接层和输出层构成分类器。

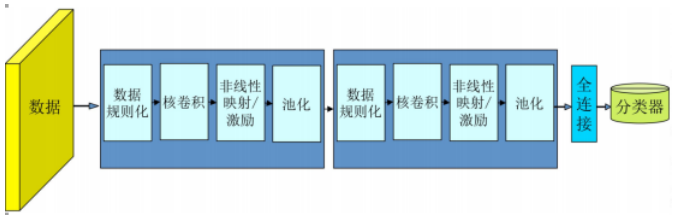


图 3.19: 卷积神经网络架构

1. 数据规则化：Batch Normalization

随着网络的深度增加，每层特征值分布会逐渐地向激活函数的输出区间的上下两端（激活函数饱和区间）靠近，这样下去会导致梯度消失。BN 通过将该层特征值分布重新拉回标准正态分布，特征值将落在激活函数对于输入较为敏感的区间，输入的小变化可导致损失函数较大的变化，使得梯度变大，避免梯度消失，同时也可加快收敛。

Batch Normalization 在实际工程中被证明了能够缓解神经网络难以训练的问题。主要来说有如下优点：

1.BN 使得神经网络中每层输入数据的分布相对稳定，加速了模型学习速度；2.BN 使得模型对网络中的参数不那么敏感，简化调参过程，使得网络学习更加稳定；3.BN 允许网络使用饱和性激活函数（例如 sigmoid，tanh 等），缓解梯度消失问题；4.BN 具有一定的正则化效果。

2. 卷积池化层：特征提取

给定图片，如何确定图中包含哪些物体？可以检测图片中的边缘来达到识别的目的。比如说，图片中的栏杆和行人的轮廓线都可以看作是垂线，同样，栏杆就是很明显的水平线，它们也能被检测到。图片中的大部分物体都能用垂直和水平边缘线来刻画，那么如何在图像中检测这些边缘呢？下面通过一个简单例子来说明。

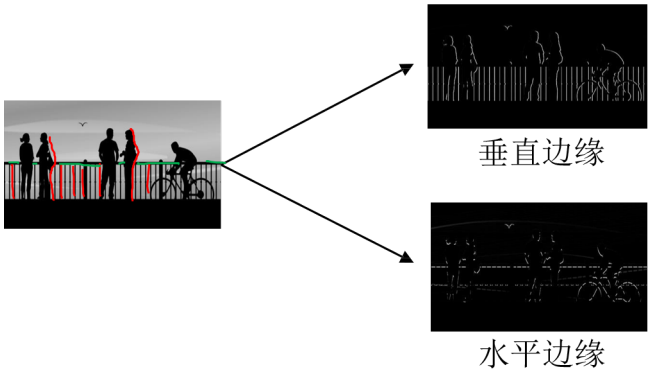


图 3.20: 边缘检测

一个 6×6 的灰度图，可以表示为左亮右暗的形式，如图左所示。很明显，中间存在明显的分界线，因此我们试图将该垂直边缘识别出来。选择卷积核为 3×3，相应的输出为 4×4。我们仍旧通过颜色的亮暗来表示卷积核和输出矩阵，如图中和图右所示。不难看出，输出矩阵对应的图中间有段发亮的区域，对应原图中的垂直边缘。

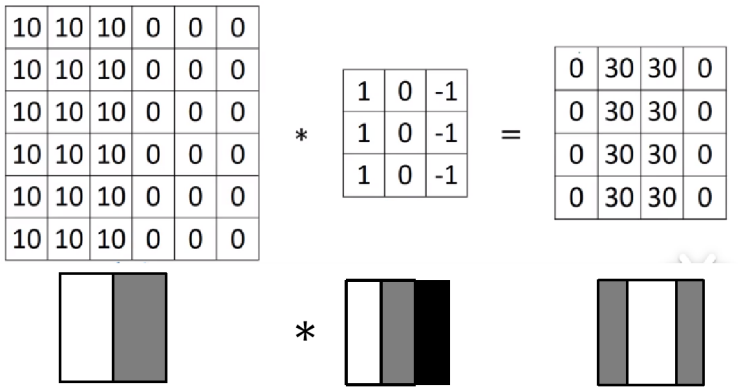


图 3.21: 卷积提取特征示意图

显然这里检测到的边缘宽度远大于原图中的边缘宽度，这和原图像素大小有关，如果改用像素值为 500×500 的图像或者更大，检测效果将会有显著提升。垂直边缘通过这样的方式可以很容易检测出，水平边缘的检测类似。推广到一般情况，卷积核就是通过这样的方式，提取到输入图像的特征。

3. 全连接层：把分布式特征（representation）映射到样本标记空间，即将特征（representation）整合到一起输出为一个值，减少特征位置对分类带来的影响。

例如：假设你是一只小蚂蚁，你的任务是找小面包。你的视野比较窄，因此只能看到很小的一片区域。当你找到一片小面包之后，你不确定你找到的是不是全部的小面包，所以你和其余的蚂蚁一块儿开了个会，把所有的小面包都拿出来以确定是否已经找到所有的小面包。某种程度上可以认为，全连接层就是这个蚂蚁大会。需要注意的是，在数据输入到全连接层前还需要进行拉直操作，将所有数据拉成一维向量。或者可以认为是某种卷积操作。由于全连接层参数过多，正逐渐被 GAP（Global Average Pooling，全局平均池化）的方法代替。

3.2.5 常见的卷积神经网络

(一) LeNet

LeNet 由 LeCun 在 1998 年提出，用于解决手写数字识别的视觉任务。

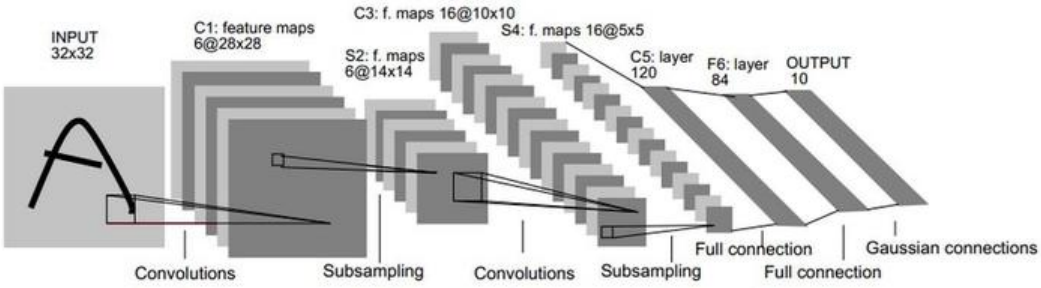


图 3.22: LeNet

图中，Convolutions 表示卷积，Subsampling 表示下采样，即池化，Full Connection 表示全连接，Gaussian Connection 表示高斯连接。具体来看，LeNet 共有五层，包括 2 层卷积层和 3 层全连接层。

LeNet 中每个卷积层均使用尺寸为 5×5 的卷积核，步长为 1，并使用 Sigmoid 激活函数。两个最池化层中池化核均为 2×2，且步长为 2，这里选择的是平均池化。由于池化核尺寸与步长相同，因此池化后在输出上每次滑动所覆盖的区域互不重叠，池化不改变通道数量。当卷积层块的输出传入全连接层块时，全连接层块会将小批量中每个样本变平。全连接层块含 3 个全连接层。它们的输出个数分别是 120、84 和 10，其中 10 为输出类别的个数。

(二) AlexNet

AlexNet 是 2012 年 ImageNet 竞赛的冠军，自此越来越多越来越复杂的神经网络被提出并得到广泛应用。

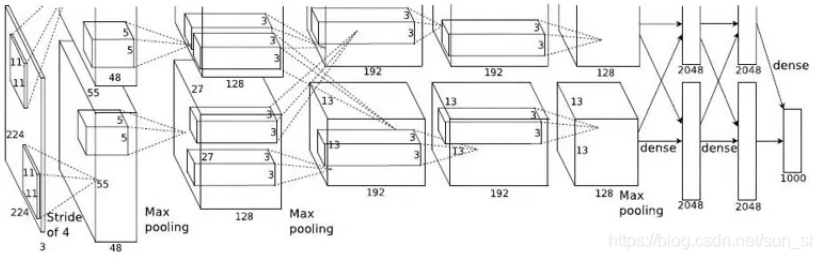


图 3.23: AlexNet

限于单个 GPU 的计算能力，AlexNet 在两个 GPU 上实现卷积神经网络的搭建和计算。图中 Stride 表示步长，Max pooling 表示池化选择最大池化，dense 为全连接。搭建和计算的大致如下：

AlexNet 共有八层，包括 5 层卷积层和 3 层全连接层。输入尺寸为 227× 227× 3 的彩色图片，使用 11× 11 的卷积核，步长选择为 4，输出图像的尺寸为 55× 55× 96，其中输出图像长（宽）度 $=\frac{\text{输入图像长（宽）度+2*padding−卷积核长（宽）度}}{\text{步长}}+1$，因此 $\frac{227+2*0-11}{4}+1=55$。

Pooling 选择步长为 2,3× 3 的池化核，输出图像的尺寸变为 27× 27× 96。再次使用 5× 5 的卷积核，步长选择为 1，并且使用 padding=2，输出得到 $\frac{27+2*2-5}{1}+1=27$，通道数为 256。

Pooling 选择步长为 2,3× 3 的池化核，输出图像的尺寸变为 13× 13× 256。第三、四、五层均选择 3× 3 的卷积核，且步长均为 1。Pooling 仍旧选择步长为 2,3×3 的池化核，输出图像的尺寸变为 6× 6× 256。

AlexNet 具有如下四个特点：

1. 使用 ReLU 作为激活函数，很好地增强了网络的非线性表达能力；

2. 引入局部响应归一化（Local Response Normalization, LRN）。通过 ReLU 函数得到的值域没有固定区间，因此要对得到的结果进行归一化。具体来说放大对分类贡献较大的神经元，抑制对分类贡献较小的神经元。

3. 使用重叠的最大池化。即池化层中卷积核的尺寸大于步长，这样池化层的输出间出现重叠和覆盖，提升了特征的丰富性。

4. 通过 Dropout 和数据扩增等方式来防止神经网络出现过拟合现象。

（三）ZFNet

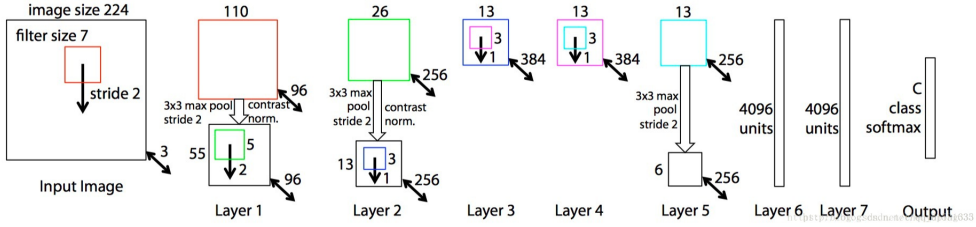


图 3.24: ZFNet

图中 image size 为输出图片的尺寸，224*224，filter size 为卷积核的大小，stride 为步长，max pool 表示池化选择最大池化，softmax 为最终用于分类的函数，Layer 标明了 ZFNet 的每一层。ZFNet 是 AlexNet 的改进版本，共有八层，仍旧是 5 层卷积层和 3 层全连接层。改进主要有以下两点：

（1）使用一块 GPU 搭建稠密连接结构；（2）第一个卷积层中卷积核的尺寸从 11× 11 变为 7× 7，同时步长从 4 减小为 2；为了让后续输出图像的尺寸与 AlexNet 中的输出图像尺寸保持一致，第 2 个卷积层的步长从 1 变为 2；

整体来看，ZFNet 的改进似乎并不明显。那么为什么要做这样的修改？修改的动机又是什么呢？这是基于卷积神经网络深层特征的可视化提出的。可视化操作，针对的是已经训练好的网络，或者训练过程中的网络快照。可视化操作不会改变网络的权重，只是用于分析和理解在给定输入图像时网络观察到了什么样的特征，以及训练过程中特征发生了什么变化。

可视化主要有如下三个操作：

（1）Rectification：因为使用的是 ReLU 激活函数，前向传播时只将正值原封不动输出，负值置 0，“反激活”过程与激活过程没什么分别，直接将来自上层的输出再次输入到 ReLU 激活

函数中即可；

（2）Unpooling：在前向传播时，记录相应 max pooling 层每个最大值来自的位置，在 unpooling 时，根据来自上层的 map 直接填在相应位置上，其余位置为 0；

（3）transposed convolution：卷积操作输出图像的尺寸一般小于等于输入图像的尺寸，transposed convolution 可以将尺寸恢复到与输入相同，相当于上采样过程，该操作的做法是，与 convolution 共享同样的卷积核，但需要将其左右上下翻转（即中心对称），然后作用在来自上层的输出图像进行卷积，结果继续向下传递。

不断进行上述操作，可以将特征映射回输入所在的像素空间，进而可以呈现出人眼可以理解的特征。给定不同的输入图像，看看每一层关注到最显著的特征是什么。

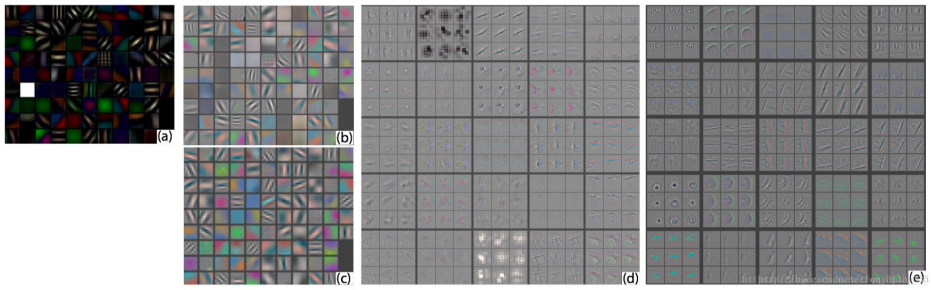


图 3.25: 可视化特征图

上图（a）为没有经过裁剪的图片经过第一个卷积层后的特征可视化图，注意到有一个特征全白，（b）为 AlexNet 中第一个卷积层特征可视化图，（c）为 ZFNet 中第一个卷积层可视化图，可以看到相比前面有更多的独特的特征以及更少的无意义的特征，如第 3 列的第 3 到 6 行，（d）为 AlexNet 中第二个卷积层特征可视化图，（e）为 ZFNet 中的第二个卷积层特征可视化图，可以看到（e）中的特征更加干净，清晰，保留了更多的第一层和第二层中的信息。

通过对 AlexNet 的特征进行可视化，Zeiler 等人发现 AlexNet 第一层中有大量的高频和低频信息的混合，却几乎没有覆盖到中间的频率信息；且第二层中由于第一层卷积用的步长为 4 太大了，导致了有非常多的混叠情况。为了解决这个问题，Zeiler 等人提高采样频率，将步长从 4 调整为 2，与之相应的将卷积核尺寸也缩小（可以认为步长变小了，卷积核没有必要看那么大范围了），修改后第一层呈现了更多更具区分力的特征，第二层的特征也更加清晰。

（四）VGGNet

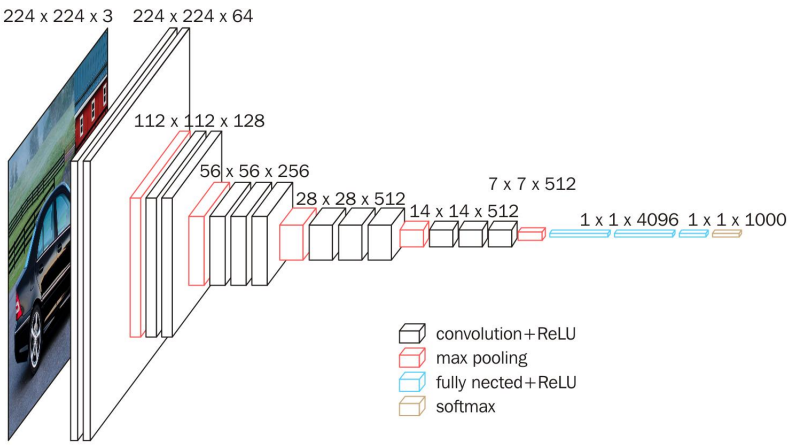


图 3.26: VGGNet

VGGNet（Visual Geometry Group Net）是由牛津大学计算机视觉组合和 Google DeepMind 公司研究员一起研发的深度卷积神经网络。它探索了卷积神经网络的深度和其性能之间的关系，通过反复的堆叠 3×3 的卷积核和 2×2 的最大池化层，成功地构建了 16 到 19 层的卷积神经网络。因此 VGGNet 是指一系列网络，下面以 VGG-16 为例进行介绍。

图中 convolution+Relu 表示卷积后选择 Relu 激活函数，max pooling 表示池化层选择最大池化，full nected+Relu 表示全连接层后选择 Relu 激活函数，softmax 激活函数用于最后的分类。

VGGNet 是一种专注于构建卷积层的简单网络，相比 AlexNet 和 ZFNet 参数量大大减少，一个很重要的原因是用到了卷积核的堆叠。例如通过 2 个 3×3 的卷积核代替 1 个 5×5 的卷积核，3 个 3×3 的卷积核代替 1 个 7×7 的卷积核。通过这样的方式不仅节省了大量参数，还使得网络获得了更大的感受野，同时增强了卷积神经网络的非线性能力。

（五）GoogLeNet

在通过卷积神经网络识别处理图像时，经常遇到图像突出部分的大小差别很大的情况。例如，猫的图像可以是以下任意情况。每张图像中猫所占区域是不同的。

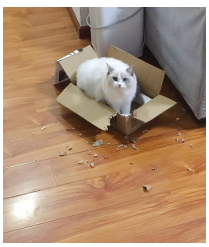


图 3.27: 不同位置的猫 1

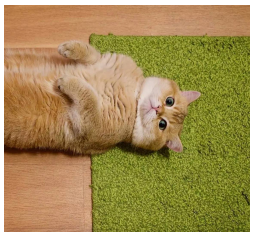


图 3.28: 不同位置的猫 2

由于信息位置的巨大差异，为卷积操作选择合适的卷积核大小就比较困难。信息分布更全局性的图像偏好较大的卷积核，信息分布比较局部的图像偏好较小的卷积核。同时，非常深的网络

更容易过拟合。将梯度更新传输到整个网络是很困难的。再加上简单地堆叠较大的卷积层非常消耗计算资源，因此考虑在同一层级上运行具备多个尺寸的卷积核呢？网络本质上会变得稍微宽一些，而不是更深。Inception 模块应运而生。

GoogLeNet 是 google 推出的基于 Inception 模块的深度神经网络模型，在 2014 年的 ImageNet 竞赛中夺得了冠军，在随后的两年中一直在改进，形成了 Inception V2、Inception V3、Inception V4 等版本。

Inception 就是把多个卷积或池化操作，放在一起组装成一个网络模块，设计神经网络时以模块为单位去组装整个网络结构。在未使用这种方式的网络里，我们一层往往只使用一种操作，比如卷积或者池化，而且卷积操作的卷积核尺寸也是固定大小的。但是，在实际情况下，在不同尺度的图片里，需要不同大小的卷积核，这样才能使性能最好，或者或，对于同一张图片，不同尺寸的卷积核的表现效果是不一样的，因为他们的感受野不同。所以，我们希望让网络自己去选择，Inception 便能够满足这样的需求，一个 Inception 模块中并列提供多种卷积核的操作，网络在训练的过程中通过调节参数自己去选择使用，同时，由于网络中都需要池化操作，所以此处也把池化层并列加入网络中。

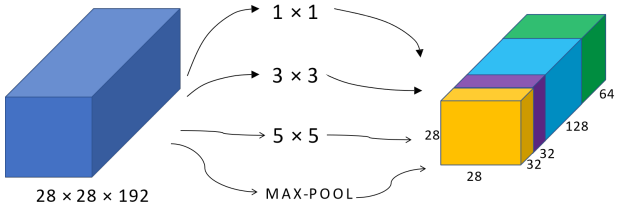


图 3.29: Inception

然而在此结构下计算仍较为复杂，单就通过 5×5 卷积得到 28×28×32 的输出来看，共进行了 28×28×32×5×5×192 次乘法运算（暂不考虑加法运算），大约在 1.2 亿次。

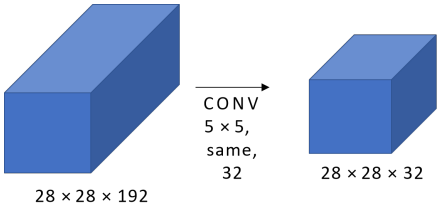


图 3.30: 1×1 卷积

能否有效减少运算次数提高计算效率呢？考虑 1×1 卷积核，如下图所示：

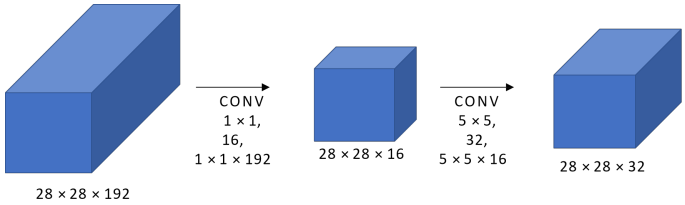


图 3.31: 1×1 卷积

此时计算分两次完成，分别是 28×28×16×1×1×192+28×28×32×5×5×16，大约是 1200 万左右，即乘法运算次数变为原来的十分之一，计算效率得到明显提升。

参数数量的变化：

原来：5×5×192×32=153600

现在：1×1×192×16+5×5×16×32=15872

如何理解 1× 1 卷积核

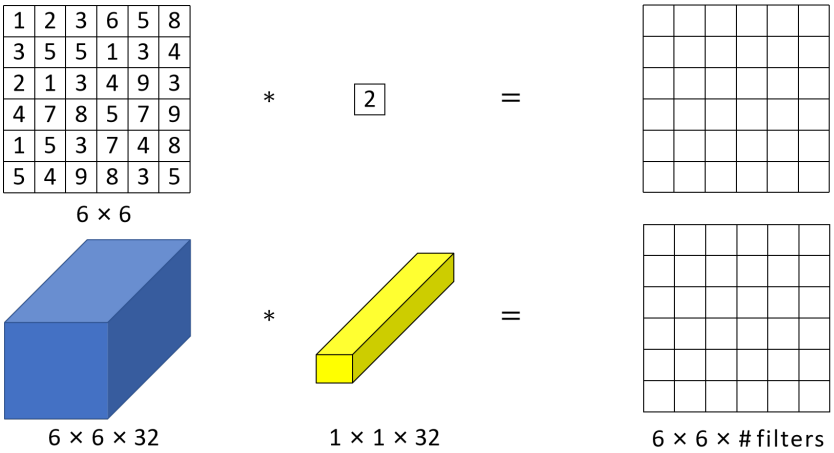


图 3.32: 1×1 卷积核的作用

当输入为 6×6×1 的单通道图片时，进行 1×1 卷积的结果是每个位置上的数字变为原来的 2 倍，似乎并没有看出 1×1 卷积核发挥了不同寻常的效果。

而当输入尺寸仍为 6×6，但通道数变为 32 时，再使用 ×1 的卷积核，此时卷积核中的 32 个元素和输入中不同平面上的 32 个元素相乘，然后应用 ReLU 激活函数。1×1×32 的卷积核中的 32 个数字可以看做是一个神经元的 32 个输入，乘以相同高度和宽度上某个切片的 32 个数字，这 32 个数字通道不同。因此，1×1 卷积从根本上可以认为这 32 个数字应用了一个全连接网络。

总的来说，1×1 卷积核减少了网络的参数，而且能够很方便地实现通道数的变化，并且增加网络的非线性特性。

Inception V1 结构如下所示：

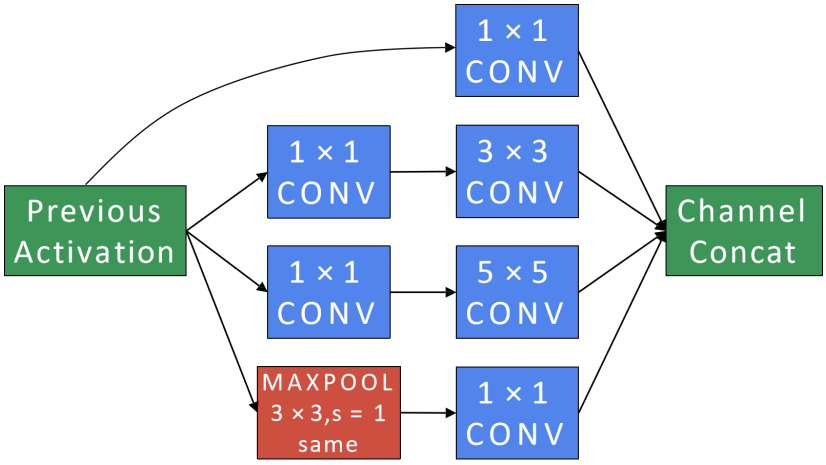


图 3.33: InceptionModule

完整的 GoogleNet 如下图所示：

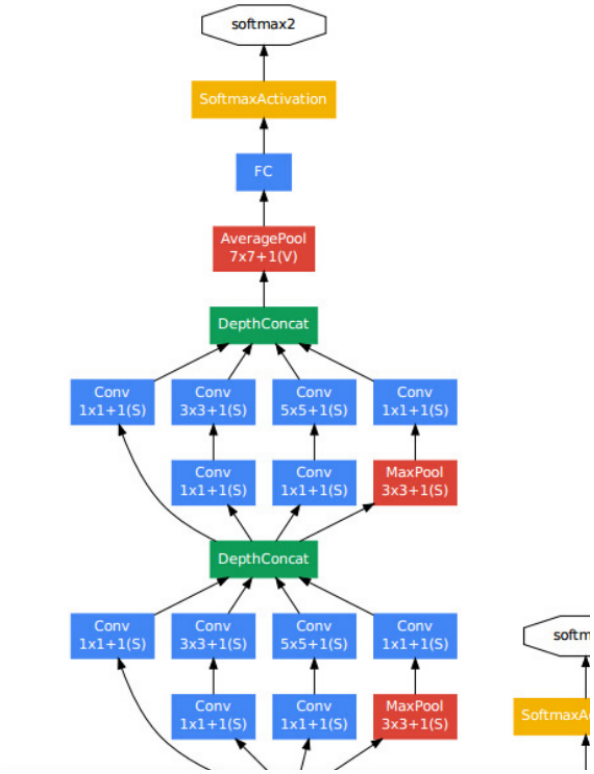


图 3.34: GoogleNet1

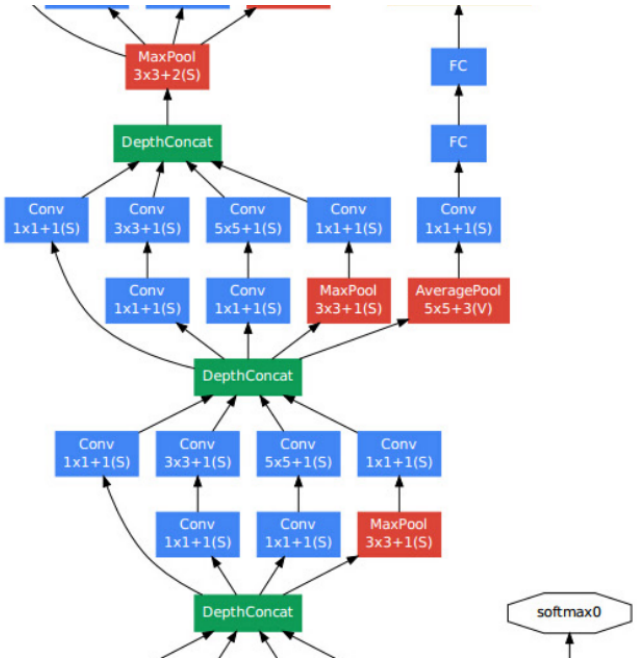


图 3.35: GoogleNet2

图 3.36: GoogleNet3

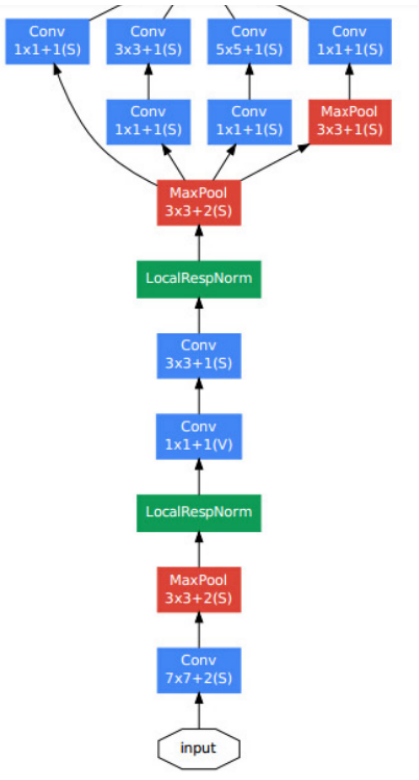


图 3.37: GoogleNet4

从中可以看出，GoogleNet 是许多 Inception 块的集合。其中 Conv 是卷积，MaxPool 是进行最大池化，DerthConcat 是进行 Inception 块的拼接，FC 是全连接层，Softmax 函数用于最终的分类。从 GoogleNet1 到 GoogleNet4，自上而下组成完整的 GoogleNet，这里为方便查看网络细节，将其拆分成四部分来分析研究。GoogleNet 的具体参数细节参见下图。

type	patch size/ stride	output size	depth	#1×1	#3×3 reduce	#3×3	#5×5 reduce	#5×5	pool proj	params	ops
convolution	7×7/2	112×112×64	1							2.7K	34M
max pool	3×3/2	56×56×64	0								
convolution	3×3/1	56×56×192	2		64	192				112K	360M
max pool	3×3/2	28×28×192	0								
inception (3a)		28×28×256	2	64	96	128	16	32	32	159K	128M
inception (3b)		28×28×480	2	128	128	192	32	96	64	380K	304M
max pool	3×3/2	14×14×480	0								
inception (4a)		14×14×512	2	192	96	208	16	48	64	364K	73M
inception (4b)		14×14×512	2	160	112	224	24	64	64	437K	88M
inception (4c)		14×14×512	2	128	128	256	24	64	64	463K	100M
inception (4d)		14×14×528	2	112	144	288	32	64	64	580K	119M
inception (4e)		14×14×832	2	256	160	320	32	128	128	840K	170M
max pool	3×3/2	7×7×832	0								
inception (5a)		7×7×832	2	256	160	320	32	128	128	1072K	54M
inception (5b)		7×7×1024	2	384	192	384	48	128	128	1388K	71M
avg pool	7×7/1	1×1×1024	0								
dropout (40%)		1×1×1024	0								
linear		1×1×1000	1							1000K	1M
softmax		1×1×1000	0								

图 3.38: GoogLeNet 参数表

关于 GoogLeNet 的几点说明：

（1）GoogLeNet 采用了模块化的结构（Inception 结构），方便增添和修改；（2）网络最后采用了 average pooling（平均池化）来代替全连接层，事实证明这样可以将准确率提高 0.6 个百分点；（3）虽然移除了全连接，但是网络中依然使用了 Dropout ；（4）为了避免梯度消失，网络额外增加了 2 个辅助的 softmax 用于向前传导梯度（辅助分类器）。这里的辅助分类器只是在训练时使用，在正常预测时会被去掉。辅助分类器促进了更稳定的学习和更好的收敛，往往在接近训练结束时，辅助分支网络开始超越没有任何分支的网络的准确性，达到了更高的水平。

随着研究的深入，陆续出现了 InceptionV2、InceptionV3、InceptionV4、Inception-Resnet V1 和 Inception-Resnet V2 等结构，并逐渐取代了 InceptionV1。例如 InceptionV2 将 InceptionV1 中的 5×5 的卷积分解为两个 3×3 的卷积运算以提升计算速度。一个 5×5 的卷积在计算成本上是一个 3×3 卷积的 2.78 倍，所以叠加两个 3×3 卷积实际上在性能上会有所提升。InceptionV2 如下图所示：

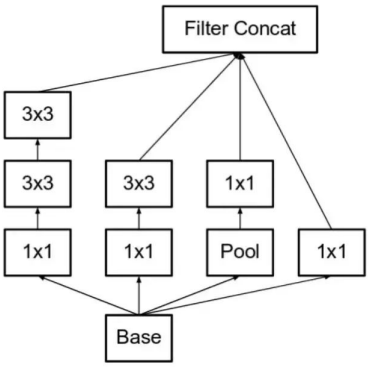


图 3.39: InceptionV2-1

此外，n×n 的卷积核尺寸分解为 1×n 和 n×1 两个卷积。例如，一个 3×3 的卷积等价于首先执行一个 1×3 的卷积再执行一个 3×1 的卷积。此时的 InceptionV2 如下图所示：

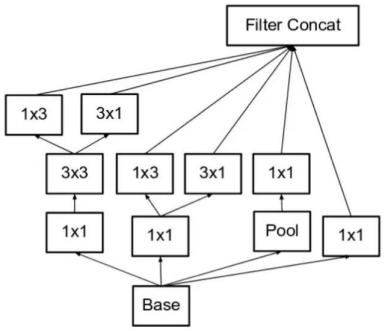


图 3.40: InceptionV2-2

3.2.6 案例：MNIST 图片分类

MNIST 数据集是机器学习领域中非常经典的一个数据集，由 60000 个训练样本和 10000 个测试样本组成，每个样本都是一张 28 * 28 像素的灰度手写数字图片。

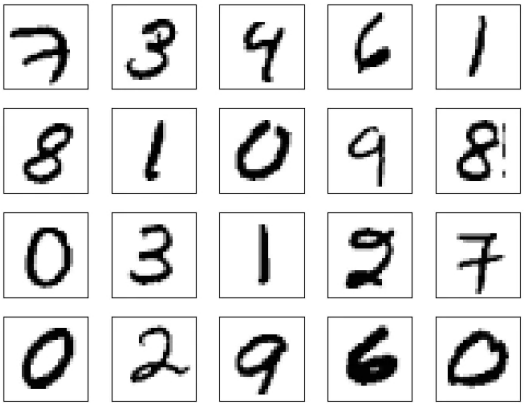


图 3.41: mnist 数据集示例

导入所需要的 keras、tensorflow 和 numpy。

```
from keras import Sequential
from keras.layers import Conv2D , MaxPooling2D , Dense , Flatten
from keras.datasets import mnist
```

```
from keras.utils import np_utils
from keras.losses import categorical_crossentropy
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
```

载入图片，并将数据尺寸调整为 28*28。

```
#载入数据
(x_train , y_train), (x_test , y_test) = mnist.load_data()
x_train = x_train.reshape((-1, 28, 28, 1))
x_test = x_test.reshape((-1, 28, 28, 1))
y_train = np_utils.to_categorical(y_train)
y_test = np_utils.to_categorical(y_test)
```

在卷积神经网络中添加卷积层和池化层。

```
model = Sequential(name='LeNet')
model.add(Conv2D(6, 5, activation='sigmoid', padding='valid',
input_shape=(28, 28, 1)))
model.add(MaxPooling2D((2, 2), 2, padding='valid'))
model.add(Conv2D(16, 5, activation='sigmoid',
padding='valid'))
model.add(MaxPooling2D((2, 2), 2, padding='valid'))
model.add(Flatten()) # 拉直运算
```

添加三层全连接层。第一层全连接，激活函数为 sigmoid 函数，输出为 120 维向量，第二层全连接，激活函数为 sigmoid 函数，输出为 84 维向量，第三层全连接，通过 softmax 函数输出 10 维向量，即分类结果。

```
model.add(Dense(120, activation='sigmoid'))
model.add(Dense(84, activation='sigmoid'))
model.add(Dense(10, activation='softmax'))
model.summary()
```

训练模型，优化方法选择 adam 方法，损失函数选择交叉熵损失函数，并设定相关参数。带入测试集测试。

```
model.compile(optimizer='adam', loss=categorical_crossentropy ,metrics=['acc'])
model.fit(x_train , y_train , batch_size=100, epochs=7,
validation_split=0.1)
result = model.evaluate(x_test , y_test)
print('测试结果:  ', result)
```

使用模型，模型可视化。

```
#使用模型
plt.figure()
for i in range(10):
```

```
    num = np.random.randint(1,10000)
    plt.subplot(2,5,i+1)
    plt.axis('off')
    plt.imshow(x_test[num],cmap='gray')
    demo = tf.reshape(x_test[num],(1,28,28))
    y_pred = np.argmax(model.predict(demo))
    plt.title('\n预测值: '+str(y_pred))
plt.show()
```

测试结果：[0.050293486565351486, 0.9824000000953674]

表 3.1: 测试结果

最终的测试结果较符合预期，在测试集上的 loss 值约为 0.05，准确率约为 0.98，在测试集中选取的 10 张手写灰度图片也成功预测。

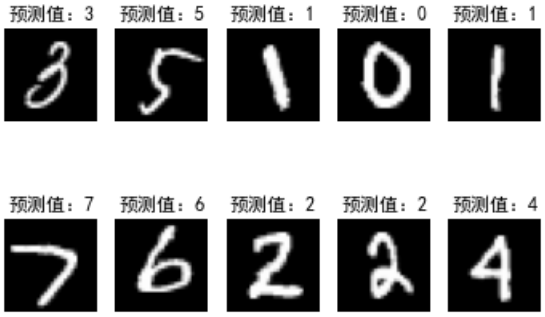


图 3.42: 分类结果

第 4 章　目标检测

4.1　ResNet

我们知道，对浅层网络逐渐叠加层，模型在训练集和测试集上的性能会变好，因为模型复杂度增加，可以对潜在的映射关系拟合得更好。但与此相悖，退化指的是给网络叠加更多的层后，性能却快速下降的情况。训练集上的性能下降，可以排除过拟合，并且 BN（Batch Normalization）层的引入也基本解决了平原网络的梯度消失和梯度爆炸问题。如果不是过拟合或者梯度消失导致的，那退化出现的原因是什么？Resnet 的提出很好地解释了这一问题。如今 Resnet 已经代替 VGG 成为一般计算机视觉领域问题中的基础特征提取网络，并且可有效生成多尺度特征表达的 FPN 网络也可通过将 Resnet 作为其基础网络从而得到一张图片最优的 CNN 特征组合。

4.1.1　ResNet 提出背景

我们首先要考虑两个问题。第一个问题是网络层数的意义，CNN 能够提取各层的特征，网络的层数越多，意味着能够提取到的特征越丰富。第二个问题是仅仅简单地增加网络层数会带来的副作用，对于原来的网络，如果简单地增加深度，会导致梯度消失或梯度爆炸。何恺明等人对于第二个问题给出了解决方法——正则初始化和中间正则化层，这样的话可以训练几十层的网络。

但是这会导致另一个问题，也就是退化问题。随着网络层数的增加，在训练集上的准确率却饱和甚至下降了。这个和过拟合不一样，因为过拟合在训练集上的表现会更加出色。按照人们一般的想法，深层的网络结构能够得到更优的解，性能会比浅层网络更佳。

但是实际上并非如此，图4.1是基于 CIFAR-10 数据集建立的 20 层和 56 层平原网络训练误差（左侧）和测试误差（右侧）变化图，深层网络无论从训练误差还是测试误差来看，都有可能比浅层误差更差，这也证明了并非是过拟合的原因。

这个问题可能是因为随机梯度下降的策略往往得到的并不是全局最优解，而是局部最优解。由于深层网络的结构更加复杂，所以梯度下降算法得到局部最优解的可能性就会更大。另外，退化问题也说明了深层的网络结构难于优化，也就是说对深层网络训练以达到相当理想的目标状态是没有实际意义的。

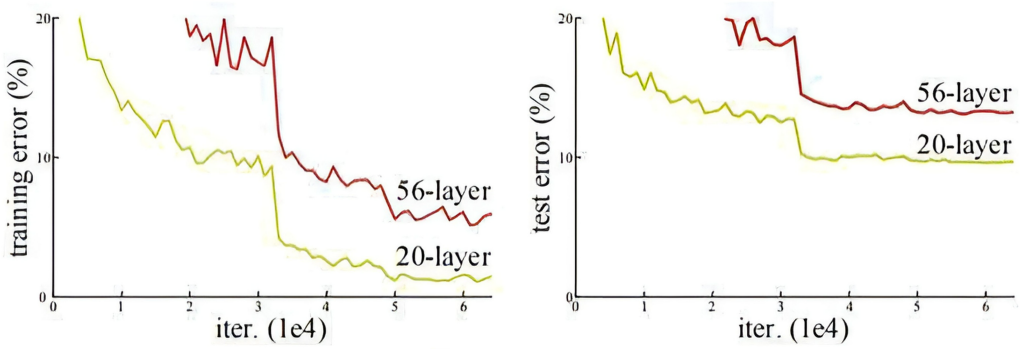


图 4.1: 20 层和 56 层平原网络误差对照图

4.1.2 退化问题的解决

既然深层网络相比于浅层网络具有退化问题，那么是否可以保留深层网络的深度，又可以有浅层网络的优势而避免退化问题呢？如果将深层网络的后面若干层学习成恒等映射 H(X)=X，那么模型就退化成浅层网络。但是直接去学习这个恒等映射是很困难的，那么就换一种方式，把网络设计成：H(X)=F(X)+X，如图4.2所示。

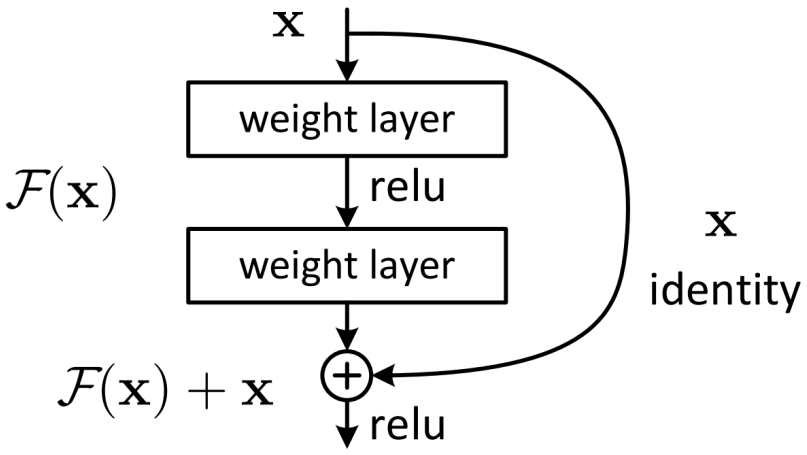


图 4.2: 残差块结构

对于退化问题，Resnet 提出了两种映射——恒等映射和残差映射。如果网络已经到达最优，继续加深网络，残差映射将被压缩为 0，只剩下恒等映射，这样理论上网络将一直处于最优状态，也就避免了网络的性能随着深度增加而降低。

4.1.3 Resnet 的结构

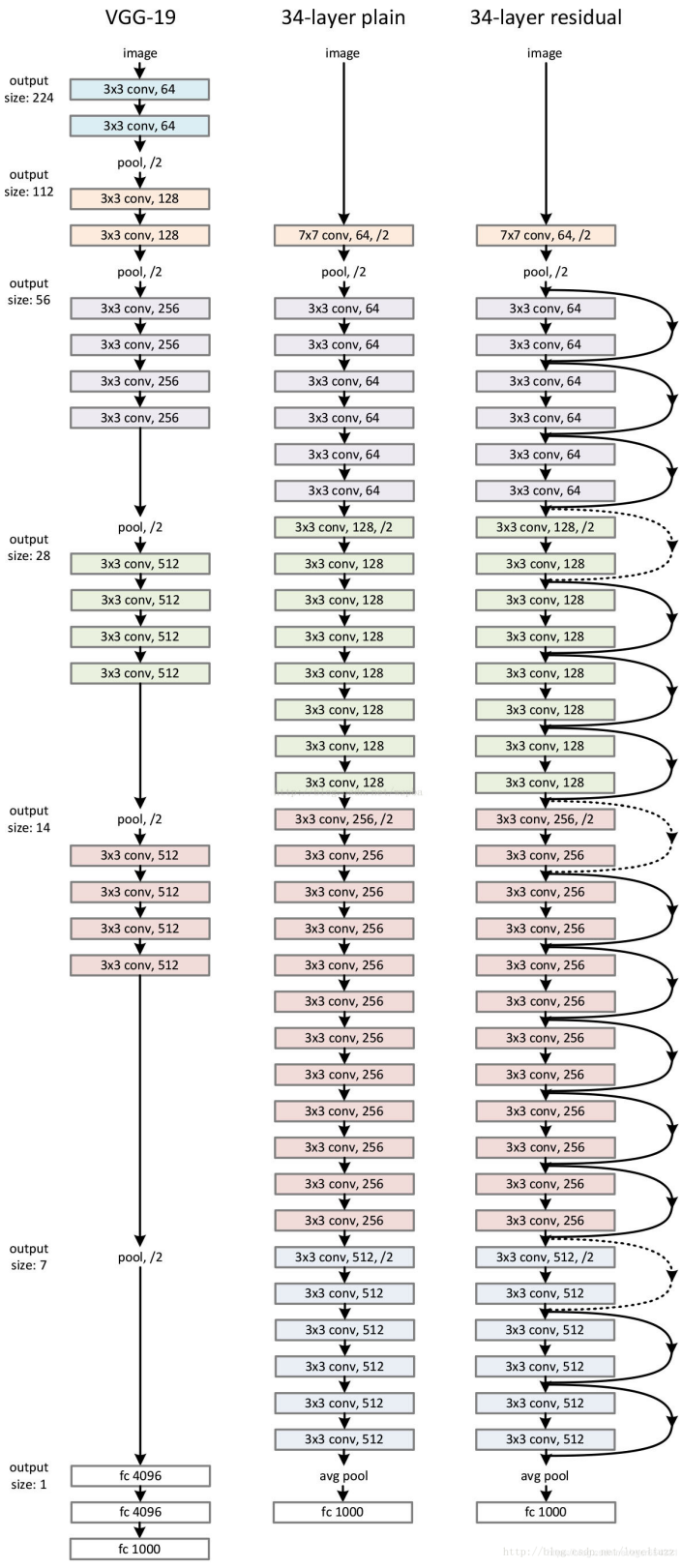


图 4.3: 不同网络结构对照

Resnet 使用了一种连接方式，叫做 “shortcut connection”，如图4.2所示。顾名思义，shortcut 就是 “抄近道” 的意思，即增加一个恒等映射，将原始需要学的函数 H(X) 转换成 F(X)+X。这两种表达的效果相同，但是 F(X) 的优化会比 H(X) 简单得多。

如图4.3所示，普通的平原网络与深度残差网络的最大区别在于，深度残差网络有很多旁路的支线将输入直接连到后面的层，使得后面的层可以直接学习残差，这些支路就叫做 shortcut。

传统的卷积层或全连接层在信息传递时，或多或少会存在信息丢失、损耗等问题。ResNet 在某种程度上解决了这个问题，通过直接将输入信息绕道传到输出，保护信息的完整性，整个网络则只需要学习输入、输出差别的那一部分，简化学习目标和难度。表4.1给出了不同层数的 Resnet 结构。

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

表 4.1: 不同层数的 Resnet 结构

我们知道残差单元通过恒等映射的引入在输入 X 和输出 Y 之间建立了一条直接的关联通道，从而使得有参层集中精力学习输入和输出之间的残差。一般我们用 $f(X,\omega)$ 来表示残差映射，ω 是残差映射对应权值，那么输出即为 Y= $f(X,\omega)$+X。

当输入和输出的通道数相同时，我们自然可以直接使用 X 进行相加。而当它们的通道数不同时，我们就需要考虑建立一种有效的恒等映射，从而使得处理后的输入 X 与输出 Y 具有相同的通道数目。

当 X 与 Y 通道数目不同时，有两种恒等映射的方式。一种即简单地将 X 相对 Y 缺失的通道直接补零从而使其能够对齐，另一种则是通过使用 $1*1$ 的卷积 W_s 从而使得最终恒等映射得到的 W_s*X 与输出的通道达到一致的方式，即 Y= $f(X,\omega)$+W_s*X。

4.1.4 案例：基于 Keras ResNet50 训练数据集进行图像分类

Keras 的应用模块（keras.applications）提供了带有预训练权值的深度学习模型，这些模型可以用来进行预测、特征提取和微调（fine-tuning）。基于 Imagenet 数据集预训练的权值，我们可以对类别包含在其中的物体进行分类。Imagenet 数据集中包含 1000 个类别，像我们常见的一些动物或者物品是可以直接加载预训练权值进行分类的。

直接使用 ImageNet 预训练的 ResNet50 进行预测

```
#导入预训练模型
from tensorflow.keras.applications.resnet50 import ResNet50
from keras.preprocessing import image
from tensorflow.keras.applications.resnet50 import preprocess_input, decode_
    predictions
import numpy as np

model = ResNet50(weights='imagenet')
```

初始化一个预训练模型时，会自动下载权重到 /.keras/models/目录下，若自动下载失败，需要去网站手动下载，并放在“ /.keras/models/”中。

```
#导入待检测图片，并进行预处理，使其满足模型所需格式
img_path = 'elephant.jpg'
img = image.load_img(img_path, target_size=(224, 224))
x = image.img_to_array(img)
x = np.expand_dims(x, axis=0)
x = preprocess_input(x)

preds = model.predict(x)
print('Predicted:', decode_predictions(preds, top=3)[0])
#输出最可能的3个预测值，格式为(class, description, probability)
```

应用猫狗大战数据集训练并保存 ResNet50 模型

如果我们要进行分类的物体超出 Image 数据集的 1000 个类别，或不包含在 Image 数据集中，我们就需要自己去训练模型权值。这里以 Kaggle 比赛猫狗大战数据集为例，介绍应用已有数据集基于 resnet50 训练一个对猫狗分类的模型。实验用到了 1000 张猫和 1000 张狗的图像，按照 4:1 的比例划分训练集与测试集。

```
import numpy as np
from keras.optimizers import adam_v2
import cv2, os
from keras.preprocessing.image import img_to_array
from sklearn.model_selection import train_test_split
from keras.callbacks import ModelCheckpoint, ReduceLROnPlateau
from keras.applications.resnet import ResNet50
from keras.models import load_model
from keras.preprocessing.image import ImageDataGenerator
norm_size = 100     #设置输入图像的大小
datapath = 'data1/train'
EPOCHS = 70
batch_size = 16     #根据硬件情况和数据集大小设置，一般为2的次方
INIT_LR = 0.001
classnum = 2
dicClass = {'cat': 0, 'dog': 1}
```

模型导入并设置好参数后，接下来需要对待检测图片数据进行预处理。

```
labelList = []
def loadImageData():
    imageList = []
    listImage = os.listdir(datapath)
    for img in listImage:
        labelName = dicClass[img.split('.')[0]]
        print(labelName)
        labelList.append(labelName)
        dataImgPath = os.path.join(datapath, img)
        print(dataImgPath)
        image = cv2.imdecode(np.fromfile(dataImgPath, dtype=np.uint8), -1)
        image = cv2.resize(image, (norm_size, norm_size), interpolation=cv2.
            INTER_LANCZOS4)
        image = img_to_array(image)
        imageList.append(image)
    imageList = np.array(imageList, dtype="int")/255.0
    return imageList
```

数据预处理之后，我们需要划分训练集和测试集，一般按照 4:1 的比例来划分。

```
print("开始加载数据")
imageArr = loadImageData()
labelList = np.array(labelList)
print("加载数据完成")
print(labelList)
trainX, valX, trainY, valY = train_test_split(imageArr, labelList, test_size
    =0.2, random_state=1)
#random_state设置随机数的种子保证每次结果相同
```

接下来需要通过 ImageDataGenerator() 进行图像增强操作。

```
train_datagen = ImageDataGenerator(featurewise_center=True,featurewise_std_
    normalization=True,rotation_range=20,width_shift_range=0.2,height_shift_range
    =0.2,horizontal_flip=True)
val_datagen = ImageDataGenerator()
train_generator = train_datagen.flow(trainX, trainY, batch_size=batch_size,
    shuffle=True)
val_generator = val_datagen.flow(valX, valY, batch_size=batch_size, shuffle=True
    )
```

最后是模型的模型训练与保存，然后就可以调用该模型进行分类。

```
checkpointer = ModelCheckpoint(filepath='weights_best_Reset50_model.hdf5',
    monitor='val_accuracy',verbose=1,save_best_only=True,mode='max')
reduce = ReduceLROnPlateau(monitor='val_accuracy', patience=10,verbose=1,factor
    =0.5,min_lr=1e-6)
model = ResNet50(weights=None, classes=classnum)
```

```
optimizer = Adam(lr=INIT_LR)
model.compile (loss='sparse_categorical_crossentropy',optimizer=optimizer,
    metrics=['accuracy'])
history = model.fit_generator(train_generator,steps_per_epoch=trainX.shape[0] /
    batch_size,validation_data=val_generator,epochs=EPOCHS,validation_steps=valX.
    shape[0] / batch_size,callbacks=[checkpointer, reduce],verbose=1, shuffle=
    True)
model.save('my_model_resnet.h5')
```

4.2 目标检测

目标检测是计算机视觉和数字图像处理的一个热门方向，广泛应用于机器人导航、智能视频监控、工业检测、航空航天等诸多领域，通过计算机视觉减少对人力资本的消耗，具有重要的现实意义。因此，目标检测也就成为了近年来理论和应用的研究热点，它是图像处理和计算机视觉学科的重要分支，也是智能监控系统的核心部分，同时目标检测也是泛身份识别领域的一个基础性的算法，对后续的人脸识别、步态识别、人群计数、实例分割等任务起着至关重要的作用。由于深度学习的广泛运用，目标检测算法得到了较为快速的发展。接下来将主要介绍基于深度学习的两种目标检测算法，分别为 One-Stage 目标检测算法 RCNN 系列算法以及 Two-Stage 目标检测算法 YOLO 算法。

4.2.1 相关介绍

什么是目标检测

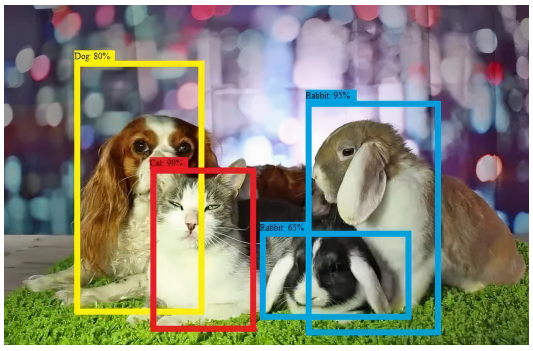


图 4.4: 多物体目标检测

目标检测问题就是找出图像中所有感兴趣的物体，包含物体定位和物体分类两个子任务，同时确定物体的类别和位置。另外，如图4.4所示，目标检测除了对单个物体进行检测，还支持对多个物体进行检测。

基于深度学习的目标检测算法

目标检测任务可分为两个关键的子任务：目标分类和目标定位。目标分类任务负责判断输入图像或所选择图像区域（Proposals）中是否有感兴趣类别的物体出现，输出一系列带分数的标

签表明感兴趣类别的物体出现在输入图像或所选择图像区域中的可能性。目标定位任务负责确定输入图像或所选择图像区域中感兴趣类别的物体的位置范围，输出物体的包围盒、物体中心、或物体的闭合边界等，通常使用方形包围盒，即 Bounding Box 来表示物体的位置信息。

目前主流的目标检测算法主要可以分成两大类别：

(1) 两阶段目标检测算法

这类检测算法将检测问题划分为两个阶段，第一个阶段首先产生候选区域（Region Proposals），包含目标大概的位置信息，然后第二个阶段对候选区域进行分类和位置精修，这类算法的典型代表有 RCNN，Fast RCNN，Faster RCNN 等；

(2) 一阶段目标检测算法

这类检测算法不需要先产生候选框，可以通过一步直接输出物体的类别概率和位置坐标，比较典型的算法有 YOLO、SSD 和 CornerNet。

目标检测模型的主要性能指标是检测准确度和速度，其中准确度主要考虑物体的定位以及分类准确度。一般情况下，两阶段算法在准确度上有优势，而一阶段算法在速度上有优势。不过，随着研究的深入，两类算法都在一定程度做了改进，均能在准确度以及速度上取得较好的结果。

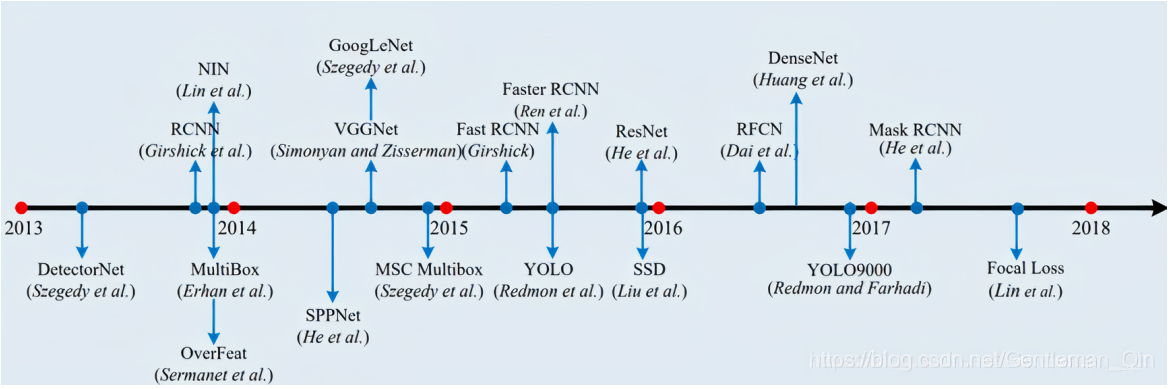


图 4.5: 目标检测算法发展历史

如图4.5所示，在 2014 年左右已经出现一些算法，比如 RCNN 算法，当然在 RCNN 出现之前，一些传统的图像处理方法也可以处理目标检测问题，能够识别图像中的物体并且确定图像中物体的位置，但是这些算法的速度和准确度有一定的限制，所以还没有达到实用性的要求。虽然 RCNN 的出现使得准确度有了很大提升，但是检测速度极慢。后面 SPPNet 的提出，使得在没有牺牲任何检测精度的前提下，提升了算法速度。这之后还有 Fast RCNN 以及 Faster RCNN。2015 年之后开始出现 YOLO 系列的算法以及 SSD 算法。综合考虑准确度以及速度，现在目标检测用的比较多的算法是 YOLO 系列算法。

候选区域的产生

很多目标检测技术都会涉及候选框（Bounding Boxes）的生成，对于物体候选框的获取当前主要使用图像分割与区域生长技术。区域生长（合并）依据的主要是图像中物体的局部区域相似性（颜色、纹理等）。

(1) 滑动窗口

如图4.6, 首先对输入图像应用不同窗口大小的滑窗, 进行从左往右、从上到下的滑动。每次滑动时候对当前窗口执行分类器 (分类器是事先训练好的)。如果当前窗口得到较高的分类概率, 则认为检测到了物体。对每个不同大小的滑窗都进行检测后, 会得到不同的物体标记。但这些窗口会存在重叠部分, 最后采用非极大值抑制 (Non-Maximum Suppression, NMS) 的方法进行筛选。最终, 经过 NMS 筛选后获得检测到的物体。

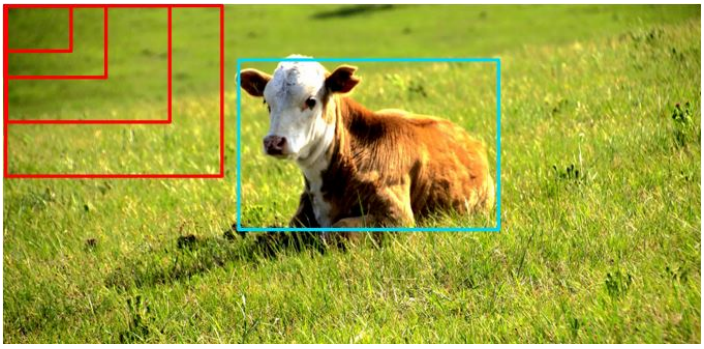


图 4.6: 滑动窗口目标检测

滑窗法思路简单, 但是不同大小的滑窗进行图像全局搜索会导致效率低下, 而且设计窗口大小时还需要考虑物体的长宽比。所以, 对于实时性要求较高的分类器, 不推荐使用滑窗法。

(2) 选择性搜索

滑窗法类似穷举进行图像子区域搜索, 但是一般情况下图像中大部分子区域是没有物体的, 那么只对图像中最有可能包含物体的区域进行搜索, 有可能提高计算效率, 这是选择性搜索最初的一种想法。选择搜索 (Selective Search, SS) 方法是当下最为熟知的图像候选框提取算法, 由 Koen E.A 于 2011 年提出。

图像中物体可能存在的区域应该具有某些相似性或连续性特点, 选择性搜索基于上面这一想法采用子区域合并的方法提取 Bounding Boxes。给定一张图像, 先对输入图像进行分割, 产生许多小的子区域, 再根据区域的颜色、纹理、大小等相似性, 不断的进行区域迭代合并。每次迭代对新合并的子区域做外切矩形, 这些子区域的外切矩形就是通常所说的候选框 (Bounding Boxes)。图4.7第一行展示了区域的合并过程, 另外, 第二行图像中的蓝色矩形框表示的就是所有可能的候选框。

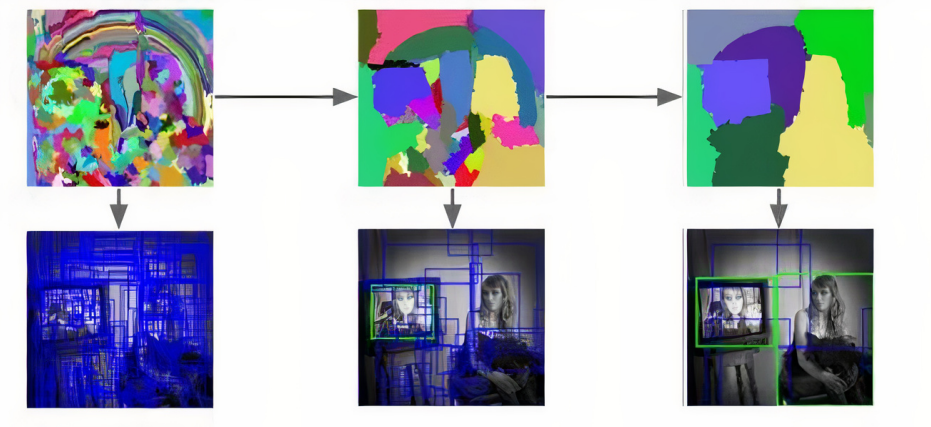


图 4.7: 候选框的产生

选择性搜索计算效率优于滑窗法，采用子区域合并策略，可以得到包含各种大小的疑似物体框；并且，判定相似区域时的指标多样性，也提高了物体检测精度。

常用图像标注工具 LabelImg

LabelImg 是一款开源的图像标注工具，标签可用于分类和目标检测，它是用 lstlisting 编写的，并使用 Qt 作为其图形界面，简单好用。注释以 PASCAL VOC 格式保存为 XML 文件，这是 ImageNet 使用的格式。此外，它还支持 COCO 数据集。

LabelImg 安装方法如下：

(1) 安装依赖包

pip install sip

pip install PyQt5

pip install pyqt5-tools

pip install lxml

(2) 生成 resources.py 文件

先切换路径到 LabelImg 文件夹所在目录，然后在命令行窗口输入命令：pyrcc5 -o libs/resources.py resources.qrc

(3) 在命令行运行 LabelImg lstlisting labelImg.py

LabelImg 安装过程中常见错误处理：

(1) 报错：no module named ‘xxx’

处理方式：大概率是要安装对应的包‘xxx’

(2) 报错：file do not exist ‘resources.qrc’

处理方式：将 lstlisting 下 scripts 添加到环境变量 path 中，并把 libs 下的 resources.qrc 或者 resources.py 拷贝到 libs 的同级目录下

数据表示

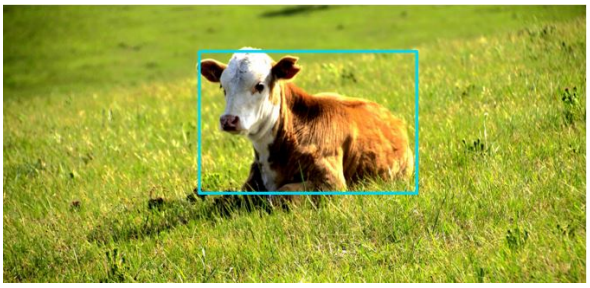


图 4.8: GT

我们一般会用如图4.8所示的人工正确标注的矩形框来表示图像中目标物体的真实位置(Ground Truth, GT)。比如对于由三个类别的图像数据训练集得到的目标检测模型，模型输出形式可以表示为:$y = (b_x, b_y, b_w, b_h, C_1, C_2, C_3)^T$, 其中，$b_x, b_y, b_w, b_h$ 表示预测边框坐标；C_1, C_2, C_3 表示属于某个类别的概率。

IoU 值

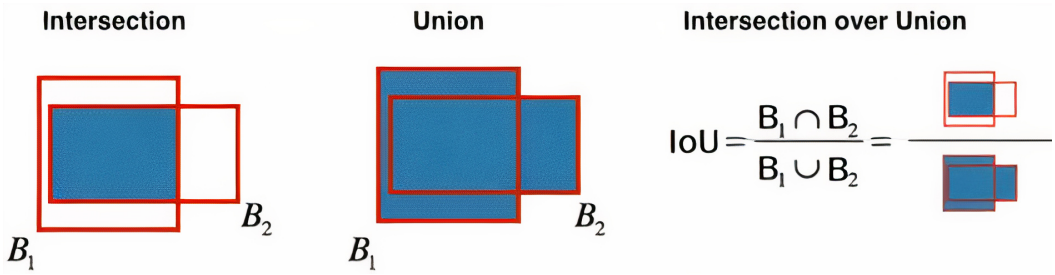


图 4.9: IoU 值的计算

如图4.9，IoU（Intersection over Union）就是两个框的交集面积除以并集面积，其中交集面积是图4.9中左侧图的阴影面积，并集面积是图4.9中右侧图阴影部分的面积。需要注意 IoU 值并非只限于衡量模型预测边框与实际边框的差异，实际上任何两个矩形框之间都可以计算 IoU 值，用于衡量两个矩形框的接近程度。

非极大值抑制

在目标检测的过程中，对目标具体位置定位不管是使用滑动窗口还是选择性搜索算法，都会产生很多的候选区域。导致不同窗口之间存在大量重合情况。也就是目标检测过程中在同一目标的位置上产生大量的候选框，这些候选框相互之间可能会有重叠。这就导致最后输出的边界框数量往往远大于实际数量，而这些模型的输出边界框往往是堆叠在一起的。因此，我们需要通过非极大值抑制（NMS）从堆叠的边框中挑出最好的那个。

如图4.10所示，一幅图像输入目标检测模型，首先产生许多候选框，第二步通过模型的分类器确定这些候选框对应类别，并且通过回归对位置进行微调，第三步通过 NMS 算法剔除冗余框，最后输出模型预测结果。

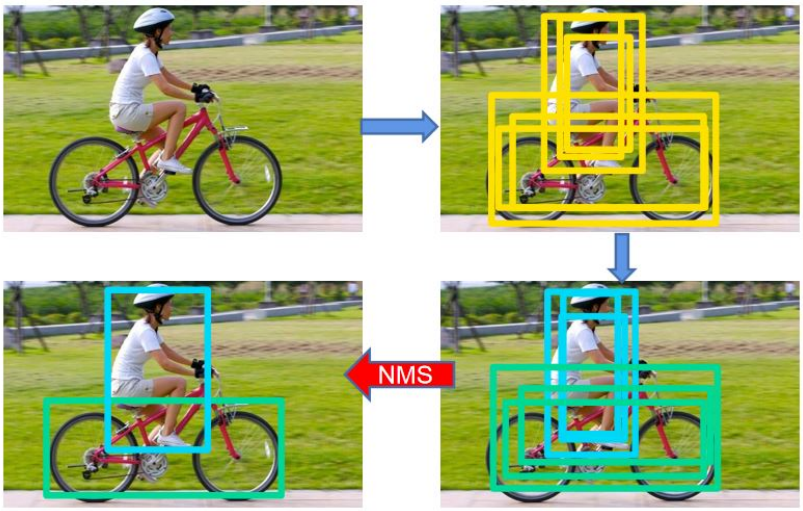


图 4.10: 一幅图像的目标检测过程

(1) 算法输入

每次输入的不是一张图所有的边框，而是一张图中属于某个类的所有边框（因此极端情况下，若所有框的都被判断为背景类，则 NMS 不执行；反之若存在物体类边框，那么有多少类物体则分别执行多少次 NMS）。

注：NMS 使用在模型的预测结果已经由置信度阈值初步筛选之后。比如在 RCNN 模型中，当所有边界框已经被分类且精修了位置，并且所有预测结果已经由置信度阈值初步筛选之后，会应用 NMS 进行冗余边界框的剔除。

(2) 算法流程

1) 边界框列表中的所有框根据置信度得分进行排序，输出得分最高的框，将其从边界框列表中删除；

2) 计算置信度最高的框与其余框的 IoU 值；

3) 删除 IoU 值大于阈值的边界框；

4) 重复上述过程，直至边界框列表为空。

(3) 算法实现

```
import numpy as np
def nms(bounding_boxes, confidence_score, threshold):
    if len(bounding_boxes) == 0:
        return [], []     #如果输入为空，则返回空列表
    boxes = np.array(bounding_boxes)      #候选框列表
    start_x = boxes[:, 0]
    start_y = boxes[:, 1]
    end_x = boxes[:, 2]
    end_y = boxes[:, 3]      #候选框位置坐标
```

```
    score = np.array(confidence_score)      #候选框对应置信度
    picked_boxes = []
    picked_score = []
    areas = (end_x - start_x ) * (end_y - start_y ) #候选框面积
    order = np.argsort(score)      #按照候选框置信度得分排序
```

接下来就开始 NMS 算法的迭代过程：

```
    while order.size > 0:
        index = order[-1]      #输出置信度得分最高的候选框
        picked_boxes.append(bounding_boxes[index])
        picked_score.append(confidence_score[index])
        x1 = np.maximum(start_x[index], start_x[order[:-1]])
        x2 = np.minimum(end_x[index], end_x[order[:-1]])
        y1 = np.maximum(start_y[index], start_y[order[:-1]])
        y2 = np.minimum(end_y[index], end_y[order[:-1]])
        w = np.maximum(0.0, x2 - x1 )
        h = np.maximum(0.0, y2 - y1 )
        intersection = w * h
        ratio = intersection / (areas[index] + areas[order[:-1]] - intersection)
        #计算IoU值，大于阈值的剔除，小于阈值的保留，进入循环
        left = np.where(ratio < threshold)
        order = order[left]
    return picked_boxes, picked_score
```

下面来看一个具体的演示例子（框的位置可根据图像自行更改）：

```
image = cv2.imread(image_name)      #导入图像
bounding_boxes = [(140, 50, 300, 275), (103, 81, 327, 226), (146, 20, 275, 252)]
confidence_score = [0.9, 0.75, 0.8]
org = image.copy()      #复制一个原始图像
font = cv2.FONT_HERSHEY_SIMPLEX
font_scale = 1
thickness = 2      #设置候选框参数
threshold = 0.5      #IoU阈值
for (start_x, start_y, end_x, end_y), confidence in zip(bounding_boxes,
    confidence_score):
    (w, h), baseline = cv2.getTextSize(str(confidence), font, font_scale,
        thickness)
    cv2.rectangle(org, (start_x, start_y - (2 * baseline + 5)), (start_x + w,
        start_y), (0, 255, 255), -1)
    cv2.rectangle(org, (start_x, start_y), (end_x, end_y), (0, 255, 255), 2)
    cv2.putText(org, str(confidence), (start_x, start_y), font, font_scale, (0,
        0, 0), thickness)      #绘制矩形框
```

生成候选框后接下来就开始用 NMS 进一步处理：

```
picked_boxes, picked_score = nms(bounding_boxes, confidence_score, threshold)
```

```
for (start_x, start_y, end_x, end_y), confidence in zip(picked_boxes, picked_
    score):
    (w, h), baseline = cv2.getTextSize(str(confidence), font, font_scale,
        thickness)
    cv2.rectangle(image, (start_x, start_y - (2 * baseline + 5)), (start_x + w,
        start_y), (0, 255, 255), -1)
    cv2.rectangle(image, (start_x, start_y), (end_x, end_y), (0, 255, 255), 2)
    cv2.putText(image, str(confidence), (start_x, start_y), font, font_scale,
        (0, 0, 0), thickness)      #绘制处理后的图像
cv2.imshow('Original', org)
cv2.imshow('NMS', image)      #对比NMS处理前后效果
```

图 4.11: NMS 剔除冗余框

代码结果：如图4.11所示，在目标检测过程中，如果不进行非极大值抑制，模型输出结果会像图4.11左侧输出多个预测框，明显有很多冗余，不符合目标检测的预期。图4.11右侧是通过设定 IoU 阈值进行非极大值抑制之后的结果，剔除了冗余候选框。

另外，关于代码部分有一点说明，示例主要目的是对 NMS 算法进行应用，对于代码后半部分调用 nms 函数用到的图像，为了简化，图像候选框在本例中是人为设置的。但是 NMS 算法应用在目标检测过程的最后一步（参考图4.10），处理的是经过目标检测前两个步骤之后得到的候选框。

4.2.2 RCNN

RCNN(Regions with CNN features)，是 RCNN 系列的第一代算法，其实没有过多地使用深度学习思想，而是将深度学习和传统的计算机视觉相结合。RCNN 可以说是利用深度学习进行目标检测的开山之作，模型提出者 Ross Girshick 多次在 PASCAL VOC 的目标检测竞赛中折桂，2010 年更是带领团队获得了终身成就奖。

算法简述

数据集采用 PASCAL VOC，这个数据集的 object 一共有 20 个类别。如图4.12，RCNN 算法主要分三步：

(1) 划分候选框

(2) 利用 CNN 提取特征

(3) 得到类别信息

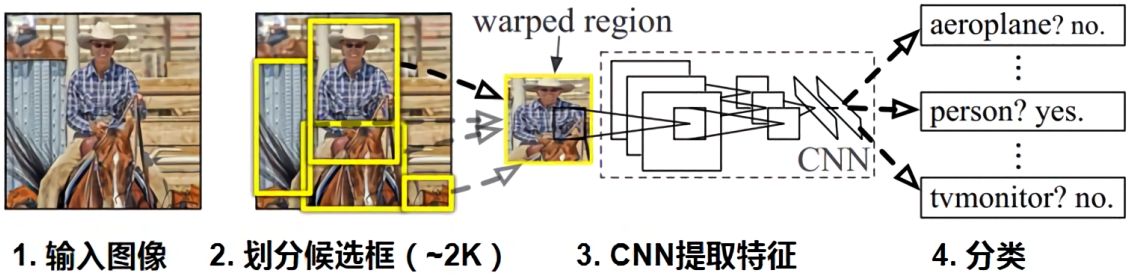


图 4.12: RCNN 算法主要流程

首先用选择性搜索算法在每张图像上选取约 2000 个候选区域，候选区域就是目标物体有可能出现的位置。然后根据这些候选区域构造训练和测试样本，注意这些候选区域的大小不一，另外样本的类别是 21 个（包括了背景）。然后是预训练，即在 ImageNet 数据集下，对 AlexNet 进行训练，并在 PASCAL VOC 数据集上 fine-tuning。之后将 AlexNet 抽取的 2000*4096 维特征向量送入 SVM 分类器中，预测出候选区域中所含物体的属于每个类的概率值。这里每个类别会训练一个 SVM 分类器，所以最后会输出 2000*20 维的得分。接下来需要对这个得分输出进行非极大值抑制（NMS）处理，去掉冗余框。最后，为了提升定位的准确性，还需要训练一个边界框回归模型，通过边框回归模型对框的准确位置进行修正。

RCNN 训练过程

(1) 特征提取网络的训练

1) 模型：TensorFlow 深度学习模型

2) 训练方式：在预训练（pre-trained）神经网络（如 AlexNet、VGG）的基础上进行微调（fine-tuning）

3) 训练数据：

正样本：将所有 Ground Truth（GT）以及和 GT 的 IoU 值在 0.5 以上的候选框作为正样本

负样本：将和 GT 的 IoU 的值在 0.5 以下的候选框作为负样本

(2)SVM 模型训练

1) 模型：sklearn 的 SVM 模型，（每个类别一个 SVM 模型，每个 SVM 模型仅区分是属于当前类别，还是属于背景）

2) 训练方式：基于 AlexNet 微调后输出的特征向量来训练 SVM 模型

3) 训练数据：

正样本: 所有 GT 实际边框作为正样本

负样本：所有和 GT 的 IoU 值在 0.3 以下的边框作为负样本

NOTE: 由于正负样本的比例是比较悬殊的，所以最好在模型训练的时候选择一些比较难区分的负样本，推荐采用 hard negative mining 方法来训练模型。

(3) 回归模型训练

1) 模型：TensorFlow 深度学习模型，每个类别（$i = 1, \cdots, N$）分别训练 4 个回归模型

2) 训练方式：基于 AlexNet 微调后输出的特征向量来训练回归模型

3) 训练数据：所有的 GT 以及和 GT 的 IoU 值在 0.6 以上的候选框全部作为训练数据

4) 损失函数：

$$\widehat{\mathbf{w}}_* = \underset{\mathrm{w}_*}{\operatorname{argmin}} \sum_{\mathrm{i}=1}^{\mathrm{N}} \left(\mathrm{t}_*^{\mathrm{i}} - \mathbf{w}_*^{\top} \phi_5\left(\mathrm{P}^{\mathrm{i}}\right)\right)^2 + \lambda \left\|\mathbf{w}_*\right\|^2$$

其中 $\mathrm{P}^{\mathrm{i}} = \left(\mathrm{P}_{\mathrm{x}}^{\mathrm{i}}, \mathrm{P}_{\mathrm{y}}^{\mathrm{i}}, \mathrm{P}_{\mathrm{w}}^{\mathrm{i}}, \mathrm{P}_{\mathrm{h}}^{\mathrm{i}}\right)$，$\mathrm{G} = (\mathrm{G_x}, \mathrm{G_y}, \mathrm{G_w}, \mathrm{G_h})$ 分别表示作为训练数据的 Bounding box 和 GT，$\phi_5(\mathrm{P})$ 为 CNN 输出的特征向量，$\mathrm{t_x} = (\mathrm{G_x} - \mathrm{P_x})/\mathrm{P_w}$，$\mathrm{t_y} = (\mathrm{G_y} - \mathrm{P_y})/\mathrm{P_h}$，$\mathrm{t_w} = \log(\mathrm{G_w}/\mathrm{P_w})$，$\mathrm{t_h} = \log(\mathrm{G_h}/\mathrm{P_h})$。

RCNN 的测试阶段

(1)Selective Search 产生候选框

(2) 使用训练好的特征提取网络提取候选框对应的特征向量

(3) 使用训练好的 SVM 模型，对提取的特征向量进行分类，得到该候选框是否属于当前类别以及属于当前类别的置信度

(4) 对于确定为物体的候选框，对其进行微调，并使用非极大值抑制来确定最终的边框。

(5) 输出结果: 候选框属于哪个类别、属于这个类别的概率值以及候选框位置。

模型评估—mAP（mean Average Precision）

我们前面介绍了关于 RCNN 模型的建立过程，对于建立的模型我们还需要一个指标来评估模型的性能。我们一般用平均准确率的均值（mAP）来度量模型准确度。假设我们有多张待检测的图像，并且这些图像里面可能会有多个类别的物体，我们对每个类别分别计算一个 AP 值(P-R 曲线的下面积)，然后将所有类别的 AP 值取平均数就可以得到 mAP 值。对于多分类目标检测的任务，会分别计算每个类别的 TP、FP、FN，进一步计算每个类别的 Precision、Recall，然后对各类别分别计算各自的 AP 值。

在目标检测背景下，对于某个类别模型在一个给定的置信度阈值下输出的所有预测框就是预测的正样本。

TP: 检测的 IoU≥IoU 阈值（一般取 0.5）

FP: 检测的 IoU<IoU 阈值

FN: 没有被检测到的 GT

通过计算可得到给定置信度阈值下的 Precision 以及 Recall 值。通过更改置信度阈值，可以改变模型输出的预测框，进而得到 P-R 曲线以及该类别的 AP 值。

RCNN 模型评价

存在的问题：

(1) 训练分为多个步骤，比较繁琐。需要微调 CNN 网络提取特征，需要训练 SVM 进行分类，还需要训练边框回归器得到最终的预测位置；

(2) 占用磁盘空间大。中间要存储候选区域的特征向量，5000 张的图片会生成几百 G 的特征文件；

(3) 预测速度很慢。一般检测一张图像大约需要 40-50 秒。

这些是 RCNN 存在的问题，之后出现的目标检测算法也基本会从这几方面做出改变，从而提升算法性能。

4.2.3 RCNN 系列模型

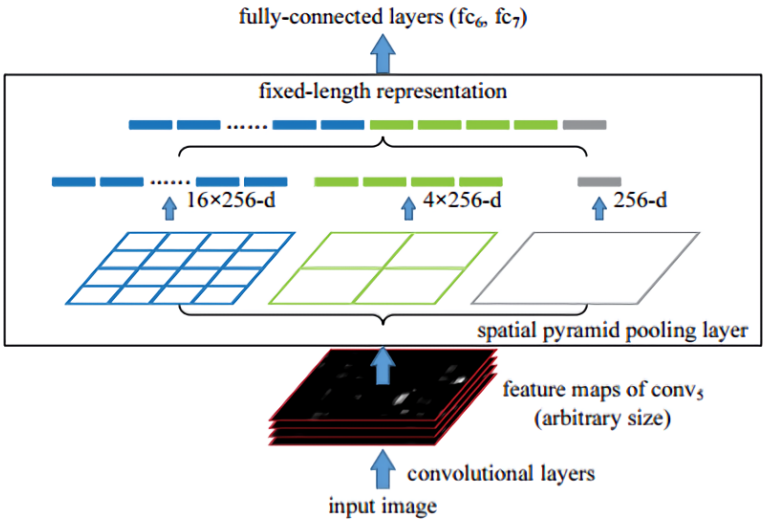


图 4.13: SPP 层

SPPNet（Spatial Pyramid Pooling Convolution NetWorks）相比原来的 RCNN 改进之处在于特征提取方面。SPPNet 用 SPP 层（Spatial Pyramid Pooling Layer）（见图4.13）代替了原来的池化层，从而使得不同大小的输入图像在经过 SPP 层后可以得到长度相同的特征向量。

RCNN 是先划分 2000 个候选框，然后将这 2000 个子图分别输入卷积网络（CNN）去提取特征，也就是说仅仅对于 1 张图像，RCNN 就进行了 2000 次的特征提取。而对于 SPPNet，虽然也有用 Selective Search 方法划分候选框这个步骤，但是 SPPNet 模型是将整图输入到网络中（见图4.14），然后输出一张特征图。

区别于 RCNN，新方法不需要先对不同尺寸的图片进行裁剪或放缩，而是直接根据候选区域在图中的相对位置计算得到在整张特征图中的映射结果。这样，对于 2000 个候选框，我们实际上只做了一次卷积操作，然后进行 2000 次特征的集合映射，再通过 SPP 层和全连接层，就可以生成最终 2000 个特征向量。接下来的步骤就和 RCNN 类似，这里不再重述。

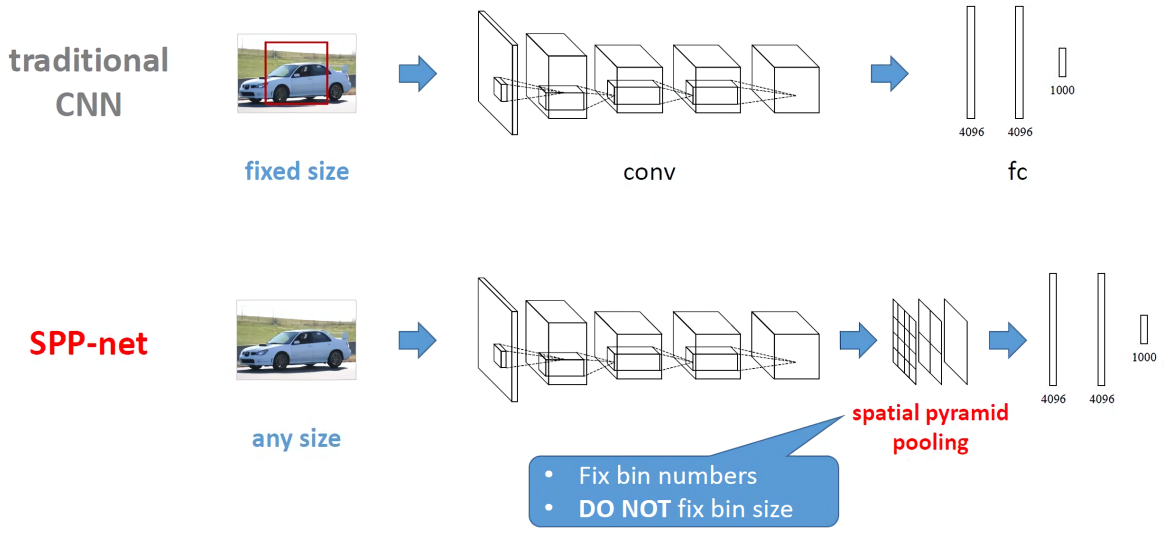


图 4.14: 传统 CNN 和 SPP-Net 流程对比

传统 CNN 和 SPPNet 流程对比如图4.14所示，传统 CNN 网络中，卷积层对输入图像大小不作特别要求，但全连接层要求输入图像具有统一尺寸大小。因此在 RCNN 中，对于选择性搜索算法（Selective Search）提取到的不同大小的候选框需要先通过裁剪、放缩来统一大小，然后用 CNN 提取特征。相比之下，SPPNet 在最后一个卷积层与其后的全连接层之间添加了一个 SPP 层，从而避免对候选区域进行裁剪或放缩操作。

总而言之，SPP 层适用于不同尺寸的输入图像，通过 SPP 层后可产生固定长度的特征向量，进而匹配后续的全连接层。

SPPNet 存在的问题：

(1) 训练仍然分为多个阶段，需要 Selective Search 方法提取候选框，需要 SVM 以及回归模型；

(2)RCNN 和 SPPNet 用于训练 SVM 分类器的特征需要提前保存在磁盘，考虑到 2000 个候选框的 CNN 特征总量还是比较大，因此造成空间代价较高；

(3) 特征提取 CNN 的训练和 SVM 分类器的训练在时间上是先后顺序，两者的训练方式独立，因此 SVM 的训练无法更新 SPP 层之前的卷积层参数，因此即使采用更深的 CNN 网络进行特征提取，也无法保证 SVM 分类器的准确率一定能够提升，并且 SPP 层之前的所有卷积层不能进行 fine-tuning。

Fast RCNN

Fast RCNN 的架构如图4.15所示，输入一幅图像，通过选择性搜索算法 Selective Search 生成一系列候选框，经过卷积操作得到特征图，然后用 RoI（Region of Interest）层处理最后一个卷积层得到的特征图，为每一个候选框生成一个固定长度的特征向量。接着经过全连接层产生最终用于多任务学习的特征并计算损失 Loss。

全连接输出包括两个分支：

(1)SoftMax Loss: 计算 K+1 类的分类损失（1 表示背景）

(2)Regression Loss: 计算最后输出的候选框坐标值的损失

图 4.15: Fast RCNN 流程图

Fast RCNN 的改进效果:

(1)Fast RCNN 检测效果优于 RCNN 和 SPPNet;

(2) 训练方式简单，基于多任务 Loss, 不需要训练 SVM 分类器;

(3)Fast RCNN 可以更新所有层的网络参数，从而更接近端对端的模式;

(4) 不需要将特征缓存到磁盘。

Faster RCNN

在之前介绍的 Fast RCNN 中，第一步需要先使用选择性搜索算法 Selective Search 提取候选框。基于 CPU 实现的 Selective Search 提取一幅图像的所有候选框大约需要 2s。在不考虑候选框提取的情况下，Fast RCNN 基本可以实现实时目标检测。但是，如果从端到端的角度考虑，显然候选框提取会成为影响算法性能的瓶颈。因此，Ren Shaoqing 提出新的 Faster RCNN 算法，该算法引入了 RPN 网络（Region Proposal Network）提取候选框。RPN 网络是一个全卷积神经网络，通过共享卷积层特征可以实现候选框的提取。RPN 提取一幅图像的候选框只需要 10ms，并且提取的候选框数量减少到 300 左右。

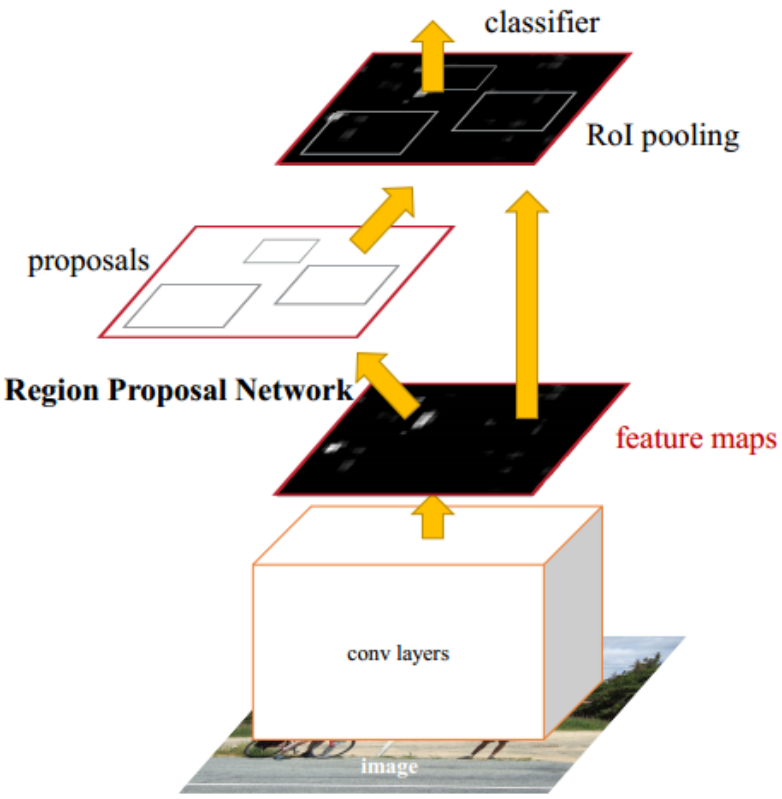


图 4.16: Faster RCNN 模块组成

Faster RCNN 算法由两大模块组成：

(1)PRN 候选框提取模块

(2)Fast RCNN 检测模块

其中，RPN 是全卷积神经网络，用于提取候选框；Fast RCNN 基于 RPN 提取的候选框检测并识别其中的目标。

关于 Region Proposal Network(RPN) 的介绍

RPN 网络的输入图片可以是任意大小（但仍有最小分辨率要求，例如 VGG 是 228*228）。如图4.17所示，RPN 的实现方式为在卷积特征图上用一个 $n*n$ 的滑窗（模型提出者选用了 n=3，即 3*3 的滑窗）生成一个长度为 256（对应于 ZF 网络）或 512（对应于 VGG 网络）维的全连接特征。然后在这个 256 维或 512 维的特征后产生两个全连接层：

(1)reg-layer：用于预测候选框的中心锚点对应的坐标 x，y 和宽高 w，h；

(2)cls-layer：用于判定该候选框是前景还是背景。

滑窗处理方式保证 reg-layer 和 cls-layer 关联了特征图的全部空间。另外对于 Anchors，字面上可以理解为锚点，位于之前提到的 n*n 滑窗的中心处，所有的锚点都具有尺度不变性。

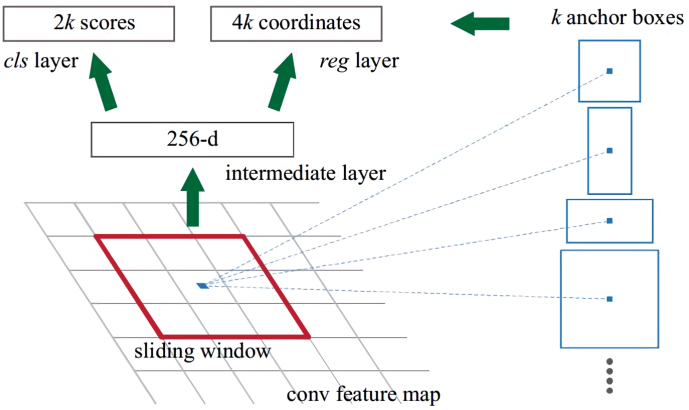


图 4.17: RPN 原理

RPN 与 Faster RCNN 特征共享

RPN 在提取得到候选区域后，使用 Fast RCNN 实现最终的目标检测和识别。RPN 和 Fast RCNN 共用了 13 个 VGG 的卷积层，因此将这两个网络完全孤立进行训练显然不是明智的选择。

对于 Faster RCNN，我们采用 RPN 和 Fast R-CNN 交替训练的方式，这样可以保持它们共享卷基层的参数不变。训练主要有 3 个部分：

(1) 训练 RPN；

(2) 用 RPN 提取得到的候选区域训练 Fast RCNN；

(3) 用 Faster RCNN 初始化 RPN 网络中共用的卷积层。

迭代执行上述三步，直到训练结束为止。其中，第一次迭代时，用 ImageNet 得到的模型初始化 RPN 和 Fast RCNN 中卷积层的参数；从第二次迭代开始，训练 RPN 时，用 Fast RCNN 的共享卷积层参数初始化 RPN 中的共享卷积层参数，然后只 Fine-tuning 不共享的卷积层和其他层的相应参数。训练 Fast-RCNN 时，保持其与 RPN 共享的卷积层参数不变，只 Fine-tuning 不共享的层对应的参数。这样就可以实现两个网络卷积层特征共享训练。

Faster RCNN 的优势

通过特征共享方式同时训练 RPN 和 Fast RCNN 能够让 Faster RCNN 实现极佳的检测效果。特征共享训练实现了买一送一，RPN 在提取候选框时不仅没有时间成本，而且候选框的质量也有所提高。因此这种交替训练的方式比原来的 “Selective Search+Fast RCNN” 更上一层楼。

总结

前面对 RCNN 的系列算法进行了一个大致的梳理，从最初 RCNN 将 CNN 应用于目标检测，到 Fast RCNN 进一步提升检测速度，再到 Faster RCNN 越来越接近端到端检测，表4.2给出了 RCNN、Fast RCNN 以及 Faster RCNN 的一个总结。

4.2.4 YOLO 相关介绍

YOLO（You Only Look Once）是一个可以一次性预测多个 Box 位置和类别的卷积神经网络，能够实现端到端的目标检测和识别，其最大的优势就是速度快。事实上，目标检测的本质就

	方法步骤	缺点	改进
RCNN	1.SS 提取 RP； 2.CNN 提取特征； 3.SVM 分类； 4. 边框回归。	1. 训练步骤繁琐(微调网络 + 训练 SVM+ 训练 BBox)； 2. 训练、测试速度都很慢； 3. 训练占内存。	1. 准确度有所提升； 2. 引入 CNN。
Fast RCNN	1.SS 提取 RP； 2.CNN 提取特征； 3.softmax 分类； 4. 多任务损失函数的边框回归。	1. 依旧使用 SS 提取 RP； 2. 无法满足实时检测，没有真正实现端到端检测； 3. 虽然应用了 GPU，但是候选区域划分在 CPU 上进行。	1. 准确度提升； 2. 处理一张图像约 3s。
Faster RCNN	1.RPN 提取 RP； 2.CNN 提取特征； 3.softmax 分类； 4. 多任务损失函数的边框回归。	1. 还是无法达到实时检测； 2. 计算量仍然很大。	1. 检测准确度和速度有所提升； 2. 离端到端更进一步。

表 4.2: 目标检测算法比较

是回归，因此一个实现回归功能的 CNN 并不需要复杂的设计过程。YOLO 没有选择滑窗或提取区域的方式训练网络，而是直接选用整图训练模型。这样做的好处在于可以更好地区分目标和背景区域。相比之下，Fast RCNN 采用的训练方式常常把背景区域误检为特定目标。

当然，YOLO 在提升检测速度的同时牺牲了一些精度。图4.18是 YOLO 检测的流程：

(1) 将图像 Resize 到 448∗448

(2) 运行 CNN；

(3) 通过非极大值抑制（NMS）优化检测结果。

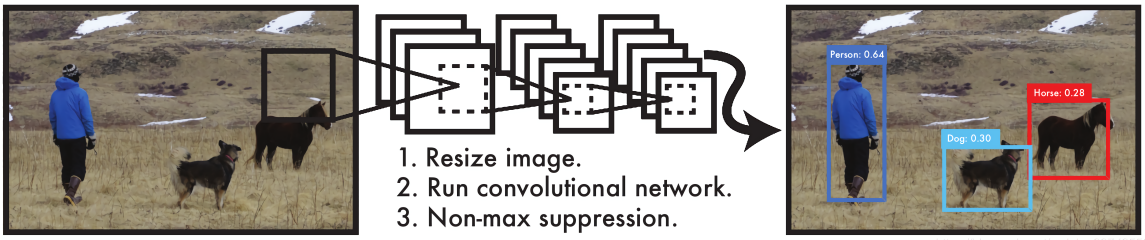


图 4.18: YOLO 算法主要步骤

端对端的检测方案

YOLO 将输入图像划分为 S*S 个网格，如果一个物体的中心落在某网格（cell）内，则相应网格负责检测该物体。在训练和测试时，每个网络预测 B 个候选框，每个候选框对应 5 个预测参数，即候选框的中心点坐标（x,y），宽高（w,h），和置信度得分。

这里的置信度得分为 Pr(Object)*IoU(pred|truth)，综合反映基于当前模型候选框内存在目标的可能性 Pr(Object) 以及候选框预测目标位置的准确性 IoU(pred|truth)。如果候选框内不存在物体，则 Pr(Object)=0。如果存在物体，则根据预测的候选框和真实的 GT 计算 IoU，同时会预测该物体属于某一类的后验概率 $\Pr(\text{Class}_i|\text{Object})$。

假定一共有 C 类物体，那么每一个网格只预测一次 C 类物体的条件概率并且每一个网格预测 B 个边界框的位置。即这 B 个边界框共享一套条件类概率 $\Pr(\text{Class}_i|\text{Object}), i = 1, 2, ..., C$。另外，测试时某个边界框类别的相关置信度计算如下：

$$\begin{aligned} &\Pr\left(\text{Class}_i \mid \text{Object}\right) * \Pr(\text{Object}) * \text{IoU}(\text{pred} \mid \text{truth}) \\ =&\Pr\left(\text{Class }_i\right) * \text{IoU}(\text{pred} \mid \text{truth}). \end{aligned}$$

如图4.19所示若将输入图像划分为 7*7 网格（S=7），每个网格预测 2 个边界框（B=2），有 20 类待检测的目标（C=20），则相当于最终预测一个长度为 S*S*(B*5+C)=7*7*30 的张量（tensor），从而完成检测以及识别任务。

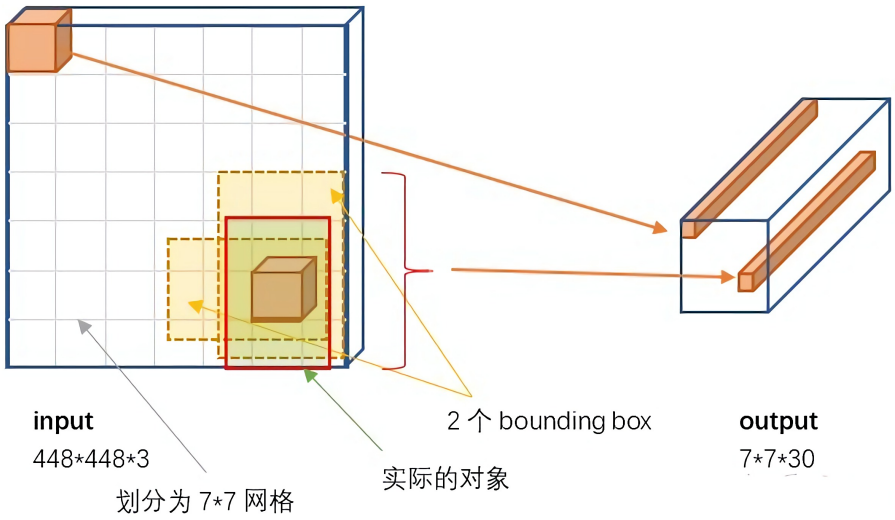


图 4.19: YOLO 模型输入和输出

YOLOv1 网络设计

YOLO 网络设计遵循了 GoogleNet 的思想，但与之有所区别。YOLO 使用了 24 个级联的卷积层（conv）和 2 个全连接层（fc），其中 conv 层包括 3*3 和 1*1 两种 Kernel，最后一个 fc 层即 YOLO 网络的输出，长度为 $S*S*(B*5+C)=7*7*30$。

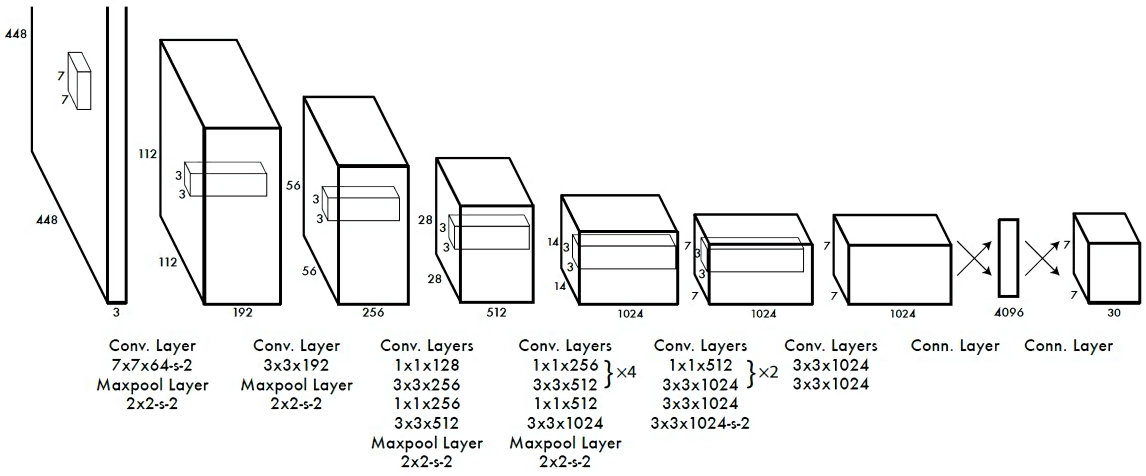


图 4.20: YOLOv1 网络结构

YOLO 模型训练

YOLO 网络的训练是分步骤进行的：首先，从图4.20网络中取出前 20 个 conv 层，然后添加了一个 average pooling 层和一个 fc 层，用 1000 个类别的 ImageNet 数据集训练。在 ImageNet2012 上用 224*224 的图像训练后得到的 top5 准确率是 0.88。然后，在 20 个预训练好的 conv 层后添加了 4 个新的 conv 层和 2 个 fc 层，并用随机参数初始化这些新添加的层，在 fine-tuning 新

层时，选用 448*448 图像训练。最后一个 fc 层可以预测物体属于不同类的概率和候选框中心点坐标 x,y 以及宽高 w,h。另外，候选框的宽高是相对于图像宽高归一化后得到的。

在设计损失函数时，有两个主要的问题：(1) 对于最后一层长度为 7*7*30 长度预测结果，计算预测损失通常会选用平方和误差。然而这种损失函数的位置误差和分类误差是 1:1 的关系。(2) 整个图有 7*7 个网格，大多数网格实际不包含物体（当物体的中心位于网格内才算包含物体），如果只计算 $\Pr(\text{Class}_i)$，很多网格的分类概率为 0，网格损失呈现出稀疏矩阵的特性，使得损失收敛效果变差，模型不稳定。

为了解决上述问题，另外采用了一系列方案：

(1) 增加候选框坐标预测的损失权重，降低候选框分类的损失权重。坐标预测和分类预测的权重都是 0.5。

(2) 平方和误差对于大小不一的候选框的权重是相同的，为了降低不同大小候选框的宽高预测的方差，采用了平方根形式计算宽高预测损失。

训练过程损失函数组成形式较为复杂，这里不作列举，如有兴趣可以参考模型提出的原文理解体会。

YOLO 模型评价

(1)YOLO 检测物体速度非常快，在增强版 GPU 中能跑 45fps（frame per second），简化版 155fps；

(2) 在训练和测试时都能看到一整张图的信息（而不像其它算法只能看到局部图片信息），因此 YOLO 在检测物体时能很好地利用上下文信息，从而不容易在背景上预测出错误的物体信息，但是预测准确度低于 Faster RCNN；

(3) 对小物体检测效果不好，尤其是密集的小物体，由于损失函数的问题，定位误差是影响检测效果的主要原因，尤其是大小物体处理上还有待加强。