实验报告

第 28 组

目录

摘要

本实验实现的是一个景点搜索网站。实验有谢知晖、徐子涵、隆嘉轩、蓝雨霆完成，其中徐子涵负责前端设计，蓝雨霆负责数据库搭建和爬虫，隆嘉轩负责数据增强部分的实现，谢知晖负责深度学习网络的搭建和训练，而整合工作主要由徐子涵和谢知晖完成。

本网站亮点在于利用了深度学习进行特征的提取，在图像的检索精确度上有良好的表现。

实验基于 Ubuntu18 的 Python3.6 平台。

1 前后端搭建

1.1 前端框架选择

当下比较热门的前端框架有 AngularJs，React，Vue.js，Flask，django 等，但是为了更好地契合本课程的要求，我们选择了 Web.py 这一轻量级的 web 工具作为我们的前端框架。

Web.py 框架最大的优势就是轻便，"Small is beautiful"；并且使用 web.py 开发的前后端可以很好地与 python 融合，这对于我们这样的一个小项目来说可以说是再适合不过了。

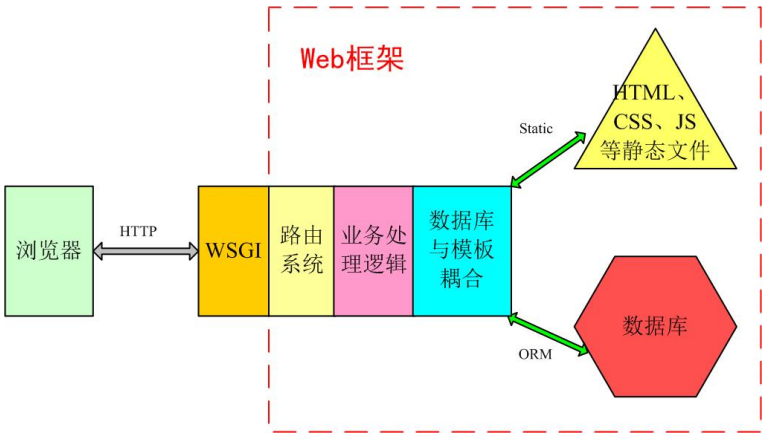


Fig1: Web 框架

1.2 后端结构设计

MVC 是众所周知的 Web 框架设计模式，即将应用程序分解成 model、view 和 controller 三个组成部分。用户输入 URL，客户端发送请求，Controller 首先会拿到请求，然后用 Model 从数据库取出所有需要的数据进行必要的处理，将处理后的结果发送给 View，视图利用获取到的数据进行渲染生成 Html 返回给客户端。MVC 设计模式将业务逻辑、数据、界面显示分离，业务逻辑聚集到一个模块中，使得在更改界面时无需重新编写业务逻辑，提高网站的维护性。

以下是后端的基本代码：

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import web
from web import form

urls = (
    '/', 'index',
```

```
)
#匹配/


render = web.template.render('templates')

login = form.Form(
    form.Textbox('keyword',  id = "kw"),
    form.Button('text', value = "text"),
    form.Button('img', value = "img")
)

class index:
    def GET(self):
        ......
        return render.index(keyword,mode,data,siteInfo)


if __name__ == "__main__":
    app = web.application(urls, globals(), False)
    app.run()
```

对应地在前端代码加上:

```
$def with (keyword, mode, data, siteInfo)
```

就可以在前端中接受和展示数据了。

1.3 前端设计

1.3.1 搜索框响应

搜索功能采用 GET 请求，同时由于我们的系统有图片搜索、文字搜索等多种功能，GET 请求中还要写入搜索模式。如下图：

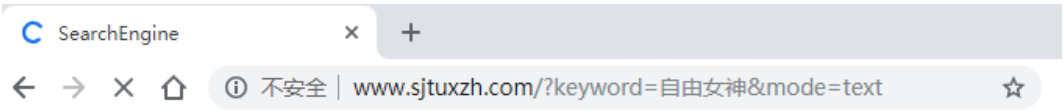


Fig2

为了实现搜索模式选择功能，我加入了一个选择模式的菜单组件，得到的效果是这样的:

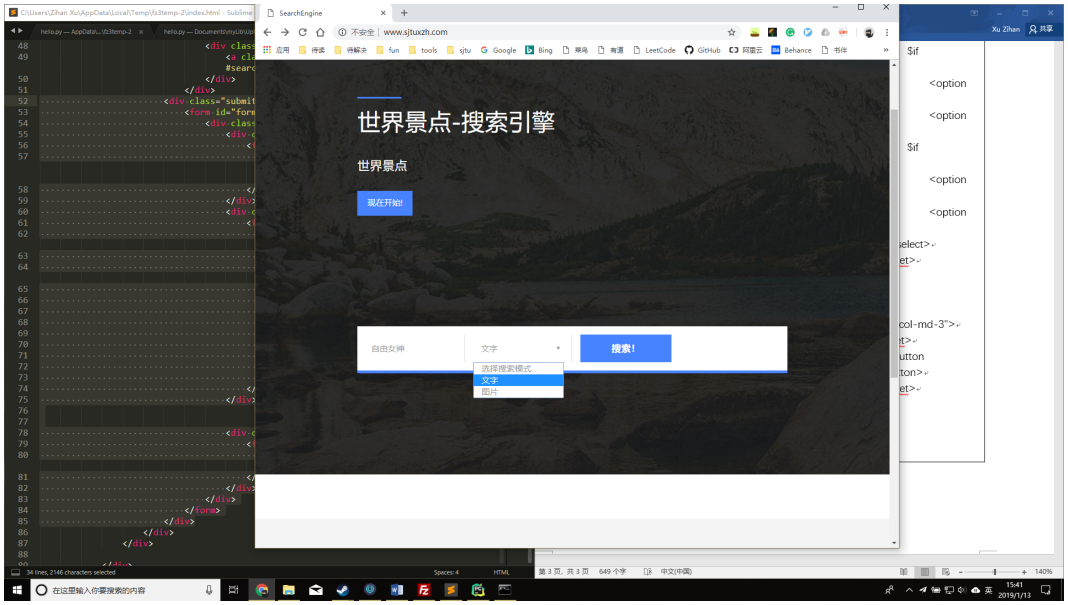


Fig3

后端解析 GET 请求也是十分简便的：

```
def GET(self):
    user_data = web.input()
    keyword = user_data.keyword
    mode = user_data.mode
    if(mode == 'text'):
        siteInfo = getNum(keyword)
        num = getNumFromName(siteInfo['name'])
        data, a = get_text_data(num)
    elif(mode == 'img'):
        data, siteInfo = getData(keyword)
    return render.index(keyword,mode,data,siteInfo)
```

1.3.2 地图组件

为了显示景点的地理坐标，我在页面中嵌入了高德提供的 API。查阅官方文档后，我找到了一个 Circle Marker 类可以用于标注景点。

这个类有 map，center，radius，bubble 等丰富的参数可供设置，生成的地图可以在网页中自由缩放移动，效果很清晰。调用的代码为：

效果图如下:

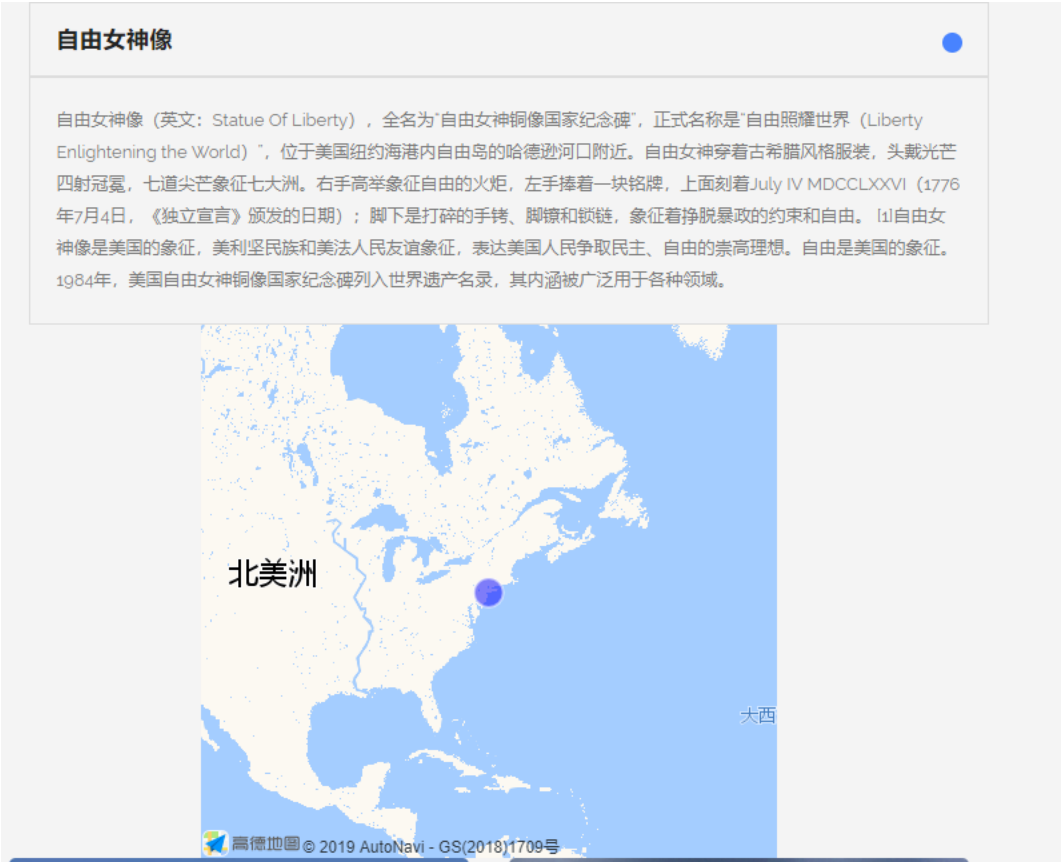


Fig4

1.3.3 模块化设计

在前端的设计中，由于采用了 web.py 框架，很多代码可以通过模块化而很简便地实现重复使用。

```
$def with (keyword, mode, data, siteInfo)

<!DOCTYPE html>
<html>
    <head>
        ...
    </head>

<body>
    $if keyword == "":
        ...
    <section class="featured-places" id="blog">
        <div class="container">
            ...
            $if siteInfo!=" ":
                ...
            $if siteInfo!=" ":
                <div class="row">
                $if mode==mode:
                    $for item in data:
                        ...
                </div>
        </div>
    </section>
...
</body>
</html>
```

Fig5

这样，图片搜索中展示的景点介绍可以直接使用文字搜索的模块完成；文字搜索中展示的图片墙可以借助图片搜索的模块完成。

模块化设计有很多优势。首先，我们的网页框架是高度可定制化的，它可以灵活地应对各种组件上的变化；其次，模块化的网页设计能够带给用户流畅的访问体验，而不用不停地重复刷新页面。

1.4 数据库 & 爬虫

1.4.1 实验思路

数据库 我们获取了景点的相应相应信息过后需要建立数据库，以便于文本搜索，和图片搜索。

在文本搜索方面，我们首先使用 lucene 建立数据库，并对相应景点名称使用 jieba 分词处理建立索引，针对景点进行搜索，然后将景点的相关信息建立数据库，最终发现，这样的效果并不好。

一方面，我们针对景点的搜索，一般是以景点名称的全名为主，辅以景点的部分名称，比如北京故宫（称呼故宫），美国自由女神像（输入自由女神像，女神像）等，我们使用结巴分

```
doc = Document()
doc.add(Field("longtitude",longitude,t2))
doc.add(Field("latitude",latitude,t2))
doc.add(Field("name",name, t2))
doc.add(Field("description",description,t2))
doc.add(Field("country",country,t2))
doc.add(Field("indexoffile",indexoffile,t2))
```

Figure 1:

词的搜索引擎分词的时候，分词得过多，导致匹配的时候出现混乱（类似于某小组的同学搜索关岛，出现皮皮岛等各种岛屿）。

另外一个方面是有的同学的代码是用 python 3 写的，Lucene 并不能很好的在 python3 的版本上使用。

于是我们最后选择使用 MySQL 作为数据库。MySQL 具有体积小，速度快，成本低，免费开源等优点，具体的是 mysql 把我们存入数据保存在一张表中，（本次实验没有使用）如果使用更多张表，还可以添加不同的 index 来加快搜索速度，增加了灵活性。并且使用 mysql

name	description	country	longitude	latitude	indexoftext
自由女神像	自由女神像（英文	美国	-74.004915	40.68954	1
乌鲁鲁巨石	艾尔斯岩石（Ayer:	澳大利亚	121.987088	-24.7076136	2
凯旋门	巴黎凯旋门，即雄	法国	2.29495	48.87384	3
伦敦塔桥	伦敦塔桥（Tower	英国	51.50551	0.0748	4
比萨斜塔	比萨斜塔：意大利	意大利	43.723	10.396	5
埃菲尔铁塔	埃菲尔铁塔（法语	法国	45.859	2.293	6
帕特农神庙	帕特农神庙位于卷	卷腊	37.9715	23.7269	7

Figure 2:

自带的查询语句 select 我们可以使用 like 的方式，使用 mysql 官网上的查询方式，类似于正则表达式的匹配形式，匹配完成对景点的模糊搜索，避免了进行分词。模糊搜索的具体实现方法是使用%{}% 方法，匹配景点名称的中间字节，这样兼顾了在绝大部分完整搜索的情况下的准确度，也兼顾了部分字节搜索的情况。

使用 mysql 建立数据库的时候值得注意的一点是我们储存的是使用 utf8 编码的中文数据，同时也是数据库设置的也是 utf8 的中文编码，然而实际过程中报了编码错误的 bug，因为 mysql 内部对 uft8 的使用并不是太好，我们需要使用 utf8mb4, 这样由 utf8 是 3 个字节，转化成了 utf8mb4 的四个字节，而且 utf8mb4 是兼容 utf8 的。

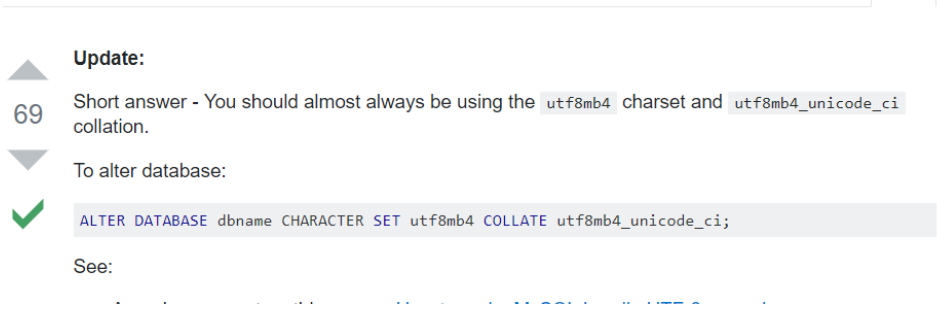


Figure 3:

爬虫 本次爬虫，我们需要对景点的地理坐标，描述信息，图片进行三个方面的爬虫。

首先是对景点的描述信息进行爬虫，为了便于景点信息的精确匹配，而不是出现搜索故宫，出现了北京故宫，台湾故宫等匹配程度不好的信息，于是我们选择了针对百度百科进行爬

虫，百度百科没有设置反爬虫机制，网页也属于静态网页，我们利用上课知识即可。

Figure 4:

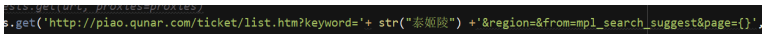


Figure 5:

然后是针对景点的坐标进行爬虫，首先我们想的爬取高德地图，百度地图等相应的地图系统，后来发现这些系统因为种种原因只能获取国内地标的信息，比如我们搜索埃菲尔铁塔，它反馈的会是广州的小世界埃菲尔铁塔的坐标，于是我们选择去哪儿网。去哪儿网包含了全世界各个地方景点的信息，而且顺序优先级比较符合期望。

在爬取去哪儿网的时候,我们尝试使用另外的一种爬虫库,可以参考相关的 tutorial https://lxml.de/tutorial.lxml etree 一种树形结构，它的功能同样非常强大。

在爬去哪儿网的时候，我遇到了验证码反爬虫，和封锁 IP 反爬虫。

第一种解决办法是使用 IP 池，是出于节约的想法，我尝试使用网上的免费 ip 网站，使用爬虫，爬取上面的 ip，当然，这种免费的 ip 池，必然绝大部分都被封掉了，我们需要使用相应的 IP 去测试能不能够访问去哪儿网，把能够访问的保存下来（详见 ip_pool.py)。

第二种方法是设置时间间隔，使用 time.sleep() 的方法，从而减少被视作爬虫程序的风险，并且在大量爬取信息的时候保护自己电脑的硬盘。

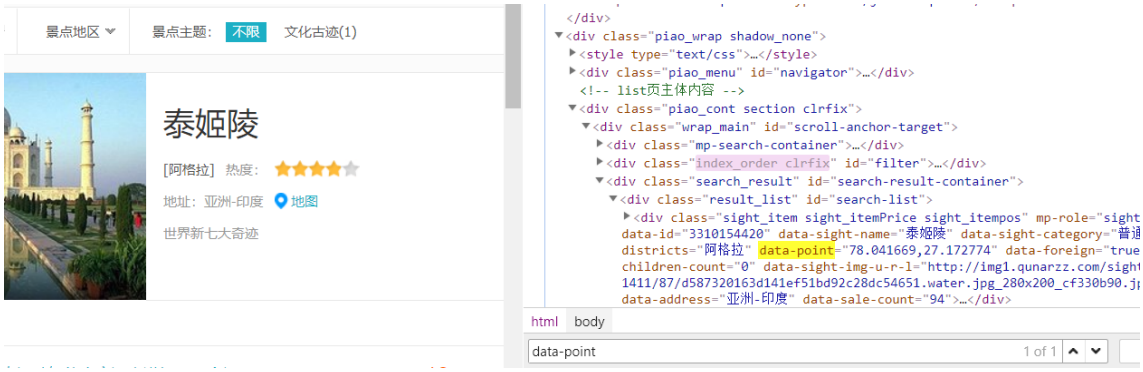


Figure 6:

最后爬取大量和某一景点的图片，一般的爬虫思想是效率很低而且错误率很高的，但是一种比较快速的方法是通过百度图片来获取大量不同的图片。在这里我们需要使用 json 库来获取图片地址。

1.4.2 实验步骤

数据库

```
connection = pymysql.connect(
  host="localhost",
  user="root",
  password="",
  db='ee208',
  charset="utf8",
  port=3306,
  cursorclass=pymysql.cursors.Cursor
)
```

在这里我们安装了 pymysql 建立数据库，我们设置主机，用户，密码，数据库名称，编码，端口等建立数据库。

```
#cursor.execute("CREATE DATABASE ee208")
cursor.execute("DROP TABLE IF EXISTS information_of_spots")
cursor.execute("""create table information_of_spots(
                            name varchar(15),
                            description varchar(1000),
```

```
                    country varchar(10),
                    longitude varchar(25),
                    latitude   varchar(25),
                    indexoftext varchar(10)
                    )"""")
```

这里建立相应的表，和列，包括景点的名称，描述，国家，经纬度等。

```
cursor.execute("ALTER DATABASE ee208 CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci")
cursor.execute("""INSERT INTO information_of_spots VALUES ('{}','{}','{}','{}','{}','{}')""".
                                                    format (name,description,country,longitude,
                                                    latitude,str(count)))
```

然后我们解决实验思路里面的编码问题，并且插入相应数据。

在文本搜索查询阶段

```
sql1="""SELECT name,description,country,longitude,latitude from information_of_spots
        where name like'%{}%' """.format(name)
```

查询语句只需要一条 sql 语句即可，这条语句针对的是名称查询，匹配景点名称的中间字节，这样同时兼顾了在绝大部分完整搜索的情况下的准确度，也兼顾了部分字节搜索的情况。

爬虫 首先，我们对百度百科进行了爬虫。

```
 cj = cookielib.CookieJar()
    opener = urllib2.build_opener(urllib2.HTTPCookieProcessor(cj))
    urllib2.install_opener(opener)

    postdata1="Taijiling"
    req = urllib2.Request(url = 'https://baike.baidu.com/item/\% s' \% postdata1)

    for i in soup.findAll(meta='',attrs={"name":"description"}):
```

在对百度百科进行爬虫的时候，我们需要对传入景点的名称进去，于是使用了 postdata 的方法用 post 进行搜索。然后寻找 name=description 的即可。

对去哪儿网我们也进行了爬虫。

```
url = 'http://piao.qunar.com/ticket/list.htm?keyword='+ str(place) +'\&region=\&from=
                                                    mpl_search_suggest\&page={}'
    print url
    page = getPage(url.format(1))
    selector = etree.HTML(page)
    print "get" +place + 'mark'
    informations = selector.xpath('//div[@class="result_list"]/div')
    inf=informations[0] #get information
    sight_point = inf.xpath('./@data-point')[0]
    print sight_point
```

代码已经在前面分析过，我们首先使用 etree 解析网页，然后找出 data-point 对应的坐标即可。

关于 IP 池部分，在 IP_Pool.py 里面, 主要是使用了一个 Ip_spider 爬虫相关网站上的 ip 地址，然后使用 Ip_pool 测试从这个网站上爬取的 ip 地址能不能使用。

而对于百度图片，我们使用同样的方法。

Figure 7:

1.4.3 爬取百度图片

当搜索不同关键词时，只有 word 参数会发生改变，值等于关键词的 utf-8 格式数据；还要注意的是 pn 参数表示页数，可以控制爬虫访问的页数。所以我们就可以通过控制这两个参数来模拟浏览器的访问操作

```
url = 'http://image.baidu.com/search/avatarjson?tn=resultjsonavatarnew\&ie=utf-8\&word=' +
                                                search + '\&cg=girl\&pn=' + str(pn) + '\&rn=60\
                                                &itg=0\&z=0\&fr=\&width=\&height=\&lm=-1\&ic=0\
                                                &s=0\&st=-1\&gsm=1e0000001e'
```

```
req = urllib.request.Request(url=url, headers=self.headers)
page = urllib.request.urlopen(req)
rsp = page.read().decode('unicode_escape')
```

就可以获取网页响应中的 json 数据：其中可以发现图片的 url 储存在 objURL 字段中 所以只

Figure 8:

需要在 python 中分解处 objURL 这个字段的数据即可，这里同时可以指定 UA 和 referrer，减少 403 错误

```
suffix = self.get_suffix(image_info['objURL'])
refer = self.get_referrer(image_info['objURL'])
```

保存图片：首先通过图片 url 拼接出访问头，这一步可以通过 urllib 来完成

```
            opener = urllib.request.build_opener()
            opener.addheaders = [
                ('User-agent', 'Mozilla/5.0 (Windows NT 10.0; WOW64; rv:55.0) Gecko/
                                                        20100101 Firefox/55.0'),
                ('Referer', refer)
```

```
]
urllib.request.install_opener(opener)
```

得到请求头之后就可以通过简单的文件操作来将图片下载并保存到本地

```
    myfile=open('./pic/'+str(count1)+'/index.txt','a')
myfile.write(str(self.__counter)+'\t'+str(image_info['objURL'])+'\n')
urllib.request.urlretrieve(image_info['objURL'], './pic/' + str(count1) + '/'
                                                + str(self.__counter) + str(
                                                suffix))
```

实际操作中，我们设定每个景点爬取 800 张图片，共 200 个景点，爬取的图片数据达到了 4GB。

1.4.4 爬虫补充

采用了动态网页加载（ajax），这是一种用于创建快速动态网页的技术。这种技术可以通过在后台与服务器进行少量数据交换，从而使网页实现异步更新。这意味着可以在不重新加载整个网页的情况下，浏览器可以对网页的某部分进行更新。所以需要通过一些反 ajax 手段来获取真实的网页内容。获取动态数据有两种方法：1. 分析页面请求 2. 利用 selenium 模拟浏览器行为或其他抓包工具直接获取（看到其他小组也有使用类似技术）为了提高爬取效率，我们决定使用第一种方法。

2 图像检索

2.1 数据增强

2.1.1 简单介绍

数据增强 (data augmentation)，又称数据扩充，是深度学习前关键也是必要的一步。在普通的图像处理方法中，大多数程序员也许都遇到过类似这样的问题：无法提供足够的内存以支持庞大的数据集；而在现实的应用之中，简单的特征提取早已无法满足我们的需求，机器学习、深度学习的重要性体现了出来，而数据增强的意义也在于此。因此，在实践中数据增强便成为深度模型训练的第一步。有效的数据增强不仅能扩充训练样本容量，还能增加训练样本的多样性 (由于其提取特征的随机性)，一方面可避免过拟合，另一方面又会带来模型性能的提升。

2.1.2 数据增强方式

在如今庞大的 python 库中，有如下几种数据增强库供我们使用:

1. Imgaug: 一个封装好的用来进行图像 augmentation 的 python 库，支持关键点 (keypoint) 和 bounding box 一起变换。
2. PyTorch:PyTorch 是使用 GPU 和 CPU 优化的深度学习张量库。
3. Keras: 一个高层神经网络 API,Keras 由纯 python 编写而成并基 Tensorflow,Thaeno 以及 CNTK 后端。
4. PIL: 即 python imaging library, 是 python 平台上标准的图像处理标准库。

5. Tensorflow:TensorFlow 是一个基于数据流编程的符号数学系统，被广泛应用于各类机器学习算法的编程实现，其前身是谷歌的神经网络算法库 DistBelief。

而最终我们选择了 Imgaug 进行数据增强。

2.1.3 数据增强的实现

一些思考 我们在开始进行对数据增强的实现时，针对实验环境、过程、结果做了一些思考:

1. 有多种数据增强库进行选择，哪一种最契合我们实验需要的内容，而又较为简洁；
2. 当库选择完毕后，哪些数据增强方法是我们需要的，哪些我们不需要；
3. 由于环境的限制，数据增强过程中速度显得尤为重要；
4. 由于爬虫速度以及数量的需要，对所有带有图像后缀名的文件均将被爬出来，而其图像格式各式各样，故可能需要一个统一的处理。

由于环境配置以及版本兼容问题 (python2 & python3)，PIL,keras 以及 PyTorch 被我们排除了，仅剩 Imgaug 以及 Tensorflow 两个选择。[1]

到此为止，基本的思路已经理清，只剩下最后一点内容难以实现: 分类数据库中的相似图像信息。这终于将我引向这次实验的主题:LSH 算法

数据增强前的处理 就像上面所说的，经爬虫得到的图像数据无法全部满足数据增强需要的图像格式 (即 3 维张量格式)，我们得到的许多图片的后缀名可能为.bmp .PNG .jpeg 等不符合要求的格式，故数据增强前此步骤至关重要。

开始时我们尝试了一些暴力更改法，发现其会导致被更改的图像失效，经多次探索后终于发现 os 库中的 rename 即可帮助我们实现我们所需要的功能。代码如下：

```
import os
import string
# 将图像统一转换为.jpg格式
dirName = 'C:/kejian/diangongdaoC/big_hw/test_jpg/'
li = os.listdir(dirName)
# print(len(li))
for filename in li:
    # print(filename)
    newname = filename
    newname = newname.split('.')
    # print(newname[-1])
    if newname[-1]=='png' or newname[-1]=='jpeg' or newname[-1] == 'gif' or newname[-1] == '
                                                    JPEG' or newname[-1]=='bmp':
        newname[-1] = 'jpg'
        newname = str.join('.', newname)
        filename = dirName + filename
        newname = dirName + newname
        # 将其后缀名更改
        os.rename(filename, newname)
```

[1]关于它们之间的其选择与权衡将放在后面的部分进行讨论。

```
        print(newname, "updated successfully")
```

这样就能帮助我们从图像数据库中将不符合要求的后缀名直接更改为.jpg 格式，满足数据增强图片格式的需要。

数据增强的实现 经过缜密的思考以及数据增强前的处理后，我们手上现在已经有了数据增强需要的所有材料：正确格式的图像、库函数。在使用 Imgaug 库实现数据增强之前，我们先对其功能进行一些了解：使用该库需要形成一个处理图片的 Sequential，在这个 Sequential 中定义多种图像处理方式对原图进行相应的处理。举例来说，其图像处理方法有：翻转、随机截取、添加噪音、卷积、加像素、乘像素等。在进行所有的操作之前，由于我们需要实现的是对应景点的搜索，翻转等对于训练便显得尤为重要；而局部扭曲、随机移动像素等操作反而可能减少模型的精确度。故我们对每种处理方法选择的频率将不同，我们便使用 lambda 定义了三种频率：

```
#定义三种概率处理方式
often = lambda aug: iaa.Sometimes(0.75, aug)
sometimes = lambda aug: iaa.Sometimes(0.5, aug)
rarely = lambda aug: iaa.Sometimes(0.25, aug)
```

这样，我么就可以选择性的将各种处理方法的概率确定下来，从而形成我们的侧重点。对数据增强方法的选择思路过于细乱，在此不予赘述。在定义了 3 中选择频率之后，我们便可以形成我们图像处理的 Sequential:

```
seq = iaa.Sequential([
    iaa.Fliplr(0.5),
    iaa.Flipud(0.5),
    often(
        iaa.Affine(
            scale=(0.9,1.1),
            translate_percent=(0.05,0.1),
            rotate=(-10,10),
            shear=(-5,5),
            order=1,
            cval=0,
        )
    ),

    iaa.SomeOf((0,5),
    [
        rarely(
            iaa.Superpixels(
                p_replace=(0,1.0),
                n_segments=(20,200)
            )
        ),

        iaa.OneOf([
            iaa.GaussianBlur((0, 3.0)),
            iaa.AverageBlur(k=(2,4)),
            iaa.MedianBlur(k=(3,5))
        ]),
```

```
        iaa.Sharpen(alpha=(0,1.0), lightness=(0.8,1.2)),

        iaa.Emboss(alpha=(0,1.0), strength=(0, 2.0)),
    rarely(iaa.OneOf([
        iaa.EdgeDetect(alpha=(0,0.3)),
        iaa.DirectedEdgeDetect(alpha=(0,0.7), direction=(0.0,1.0)),]
    )),
        iaa.AdditiveGaussianNoise(
            loc=0, scale=(0.0, 0.05*255)
        ),
        rarely(
            iaa.OneOf([
                iaa.Dropout((0.0,0.05),per_channel=0.5),
                iaa.CoarseDropout(
                    (0.03,0.05),size_percent=(0.01,0.05),per_channel=0.2
                )
            ])
        ),
        rarely(iaa.Invert(0.05, per_channel=True)),

        iaa.OneOf([
            iaa.Add((-10,10),per_channel=0.5),
            iaa.AddElementwise((-40,40))
        ]),

        iaa.OneOf([
            iaa.Multiply(mul=0.9),
            iaa.Multiply(mul=1.1),
        ]),

        iaa.Grayscale(alpha=(0.0,1.0)),

        iaa.ContrastNormalization((0.5,2.0)),
    ],
    random_order=True
),
    iaa.AddToHueAndSaturation(value=(-10,10), per_channel=True)

], random_order=True)
```

其中，有一些用法值得考究：

1. iaa.SomeOf(a, b) 此命令意思为在 (a,b] 范围中随机选择一个数 k，对该图进行 k 种其包含在内的操作
2. iaa.OneOf 顾名思义，即选择一种进行操作

这样便完成了我们程序的主体部分 (即数据增强部分)，之后的操作便是文件的添加和储存与次数选择操作。

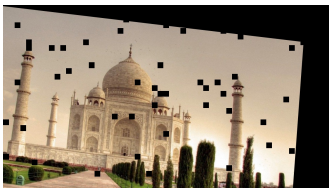
```
for item in filelist:
    # print(item)
    # print(path + item)
```

```
    img = cv2.imread(path+item)
    # print(item, type(img))
    if str(type(img)) == "<class 'NoneType'>":  # 排除无法读入的情况
        con+=1
        continue
    # img = cv2.resize(img, (224, 224)).astype(np.int8)
    imglist.append(img)

    for count in range(100): # count即一张图需要生成的数据增强图的数量
    images_aug = seq.augment_images(imglist)
    for index in range(len(images_aug)):
        filename = str(count)+str(index) + '.jpg'
        # print(savepath+filename)
        cv2.imwrite(savepath+filename, images_aug[index])
        print('image of count\%s index\%s has been writen' \% (count, index))
```

在程序中我们也可以注意到，为了增加程序的健壮性，我们将极小概率出现的 NoneType(即图片无法使用 opencv 读入的情况) 排除，保证程序能够准确无误的运行，从而实现我们需要的功能。下面是利用泰姬陵进行数据增强后的图片：

(a) 数据增强对象



(b) 第一个增强结果

(c) 第二个增强结果

以泰姬陵为例的数据增强结果

2.1.4 一些讨论

关于 Tensorflow 与 Imgaug 的选择问题 众所周知，Tensorflow 基于 Annaconda 环境，而在 linux 系统中，由于使用的是虚拟机，其数据增强过于缓慢；而在 windows 系统中，配置 Annaconda 环境十分麻烦；对于 Imgaug 而言，其仅需 python 环境，兼容性较好，图像处理较快，符合我们的要求。举例来说，使用 Tensorflow 实现如下基本图像处理操作，需要的时间为 5.0s：

```
img = cv2.imread("1.jpg")
    # 将图片缩减大小
    crop_img = tf.random_crop(img, [120,160,3])
```

```
# 随机设置图片的亮度
random_brightness = tf.image.random_brightness(img, max_delta=30)
# 随机设置图片的对比度
random_contrast = tf.image.random_contrast(img, lower=0.2, upper=1.8)
# 随机设置图片的色度
random_hue = tf.image.random_hue(img, max_delta=0.3)
# 随机设置图片的饱和度
random_satu = tf.image.random_saturation(img, lower=0.2, upper=1.8)
# 水平翻转
h_flip_img = tf.image.random_flip_left_right(img)
# 垂直翻转
v_flip_img = tf.image.random_flip_up_down(img)
sess = tf.InteractiveSession()
```

而对于 Imgaug 而言，实现 7 个随机操作需要的时间不到 1s，速度显著快于 Tensorflow。

2.1.5 一些评价

数据增强虽然难度不大，但其值得考究的部分实际上有很多，我们考虑的也仅是皮毛，还有许多可以利用、需要改进的部分。

1. Imgaug 中的 bounding box. 从某种意义上来说，bounding box 才是 Imgaug 数据增强的大头，其是一种变换方式，可以根据我们的不同的需求得到不同的处理方式 (如热度图)，而囿于时间我们没有去探讨这个部分，以后的时间中我们会尝试利用它来解决更高深的问题；
2. 其他库的使用。前面我们虽然解释了为什么我们放弃了一些库，没有去使用它，但归根结底来说，各库对于数据增强的方式是截然不同的，在时间充裕的情况下，为了得到更高的准确率，各库之间的对比显得尤为重要。我们也会去尝试一些其他的库来进行数据增强。

2.2 特征提取

2.2.1 实验思路

根据实验内容的需要，我在图像检索部分考虑了以下内容。

我们的用户希望的是通过他拥有的图片来查找到景点的信息，而这是建立在他并不知道照片中的景点是什么的前提下的。因此，我们需要给他景点的具体分类判断。

同时，用户可能已经知道这张图片中的景点是埃菲尔铁塔，但他希望在网站上找到更多相似的照片，或许他觉得这个视角下的埃菲尔铁塔十分有魅力。这又涉及到相似度的问题。

因此，在图像检索这一部分，我需要解决的是两个方面的问题: 特征的提取以及分类的确定。而摆在我面前的有两种途径。

传统特征提取技术 在之前的实验中，我们曾通过传统的特征提取技术 SIFT 提取图像特征，这无疑是很好的尝试。但同时，传统特征提取技术也暴露出许多弊端。这里我主要讨论在实验课上接触到的 SIFT 算法。

传统特征提取技术的实现步骤通常是较为繁琐的。SIFT 算法的实现需要依次对图像进行角点提取、主方向计算、描述子生成三大步骤，而仅在角点提取中就需要通过计算 DOG(差分金字塔) 并取得极值点，再排除边缘处的某些点。虽然实际上这些步骤的计算量并不大，但紧

密联系的环节让我们在尝试优化算法时不得不考虑每个环节之间的联系，以至于感到无从下手的苦涩。在"SIFT" 那次实验中，我的实验结果并不是十分理想，在对匹配进行测试时，出现了两张相似图片间关键点无法成功的情况，而这有可能是因为主方向选取时邻域的大小设定不正确，或者是角点的选取不合适。

总而言之，传统的特征提取算法虽然具有较强的可解释性，但其由于步骤复杂、优化起来比较困难，因此并没有作为我们的选择。

深度学习 在选择放弃传统特征提取算法之后，我开始关注深度学习在图像检索方面的应用。

鉴于深度学习在近年来的热度，我在此不再详细讨论深度学习的发展以及其技术特点，而直接介绍我在搭建模型时的思考过程。

由于在之前曾经接触过一些相关知识，我对于 CNN(卷积神经网络) 有一些粗浅的理解，但对于如何将网络输出应用到图像检索领域并没有十分明确的流程思路。选择什么模型呢？怎样让我们的模型能够提取出不同景点的特征呢？一些更为细节的问题有，我应该利用分类结果吗还是直接对提取出的特征向量做诸如 LSH 的后续实现？

直到我偶然看到 15 年的一篇 CVPR 会议论文 <Deep Learning of Binary Hash Codes for Fast Image Retrieval>[2]。由于这是一篇比较早的文章，其结果在现在看来可能并不算优秀，但也十分具有启发性。在这篇文章中，作者在 AlexNet 的基础上对全连接层进行了改动，在 FC7 与 FC8 之间添加了一层隐藏层，用以得到一个 48/128 位的 Hash 值输出。通过这种方式，我们可以直接学习得到一个 Hash 码，用于图像检索，在保证了特征完整性的同时大幅降低了检索时的数据量。

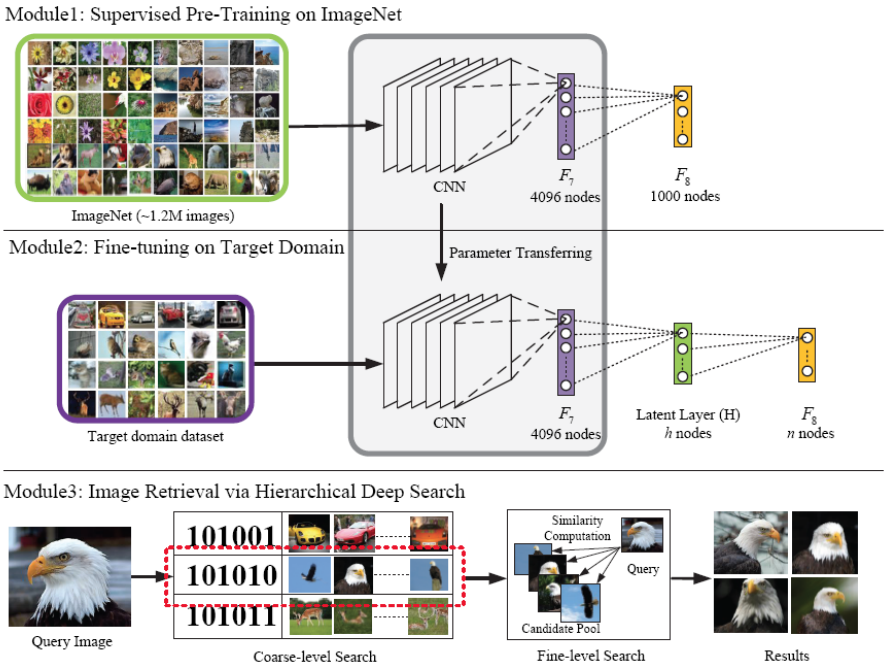


Fig1: 算法框架流程

网络以“点对”的方式学习哈希编码，利用了 CNN 的增量学习性质，引入了一种简单有效的监督学习框架适用于图像检索，使得结果在超过了 baseline，检索正确率优良。

[2]原文：https://www.iis.sinica.edu.tw/~kevinlin311.tw/cvprw15.pdf

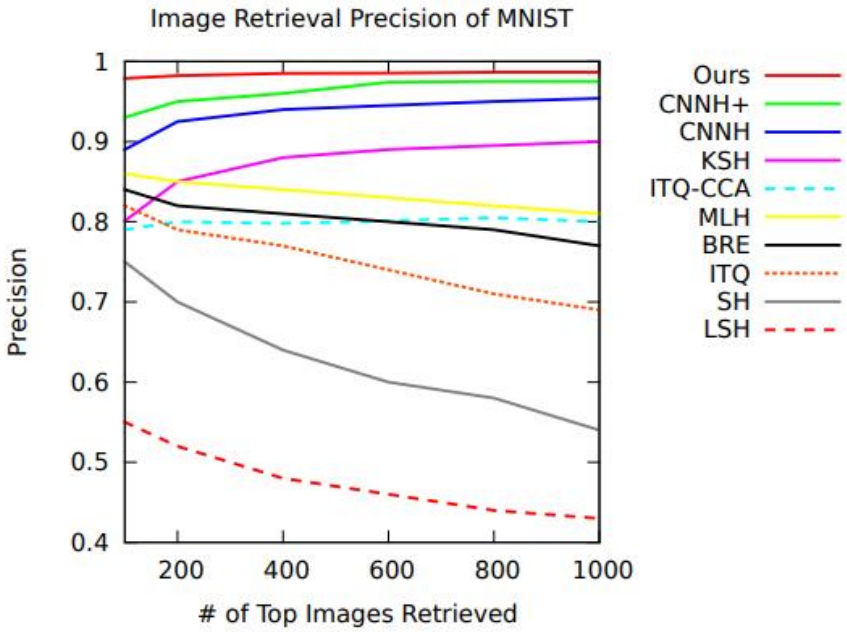


Fig2: 在 MNIST 数据集上测试的 48bits 检索正确率

受这篇文章启发，我尝试着复现文章中的方法，但在训练时遇到不少问题。在我们的自制数据集上，加入隐藏层后训练得到的结果并不十分理想，在验证集上的正确率往往只能达到 0.4 的水平。这应该是我对论文的细节理解不到位造成的，而由于时间受限，我只好放弃仿照文章实现我们的图像检索。

于是，我选择较为简单的一种方法，即不添加隐藏层，而仅 fine-tune 预训练的网络，利用训练好的网络提取 bottleneck 特征作为特征向量，再将特征向量用 LSH 的方式组织成数据库。

Fine-tune Fine-tune(微调) 是迁移学习的一种实现方式。迁移学习 (Transfer learning) 顾名思义就是把已训练好的模型参数迁移到新的模型来帮助新模型训练。考虑到大部分数据或任务都是存在相关性的，所以通过迁移学习我们可以将已经学到的模型参数 (也可理解为模型学到的知识) 通过某种方式来分享给新模型从而加快并优化模型的学习效率不用像大多数网络那样从零学习。

由于现在有许多大型的开源数据集，不少框架内都包含预训练好的各种网络，这为我们进行训练提供极大的便利。

2.2.2 环境介绍

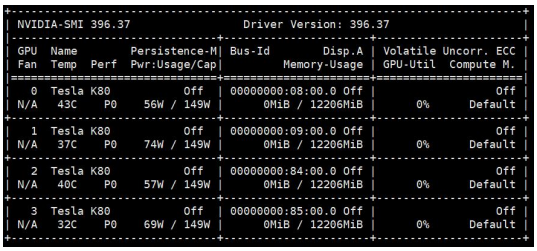
框架选择 为了便捷地搭建网络，我选择了基于 Python 的深度学习库 Keras。这是一个十分易于学习和使用的 API，它将常见用例所需的用户操作数量降至最低，并且在用户错误时提供清晰和可操作的反馈。并且，通过与底层深度学习语言 (特别是 TensorFlow) 集成在一起，Keras 在保证易用性的同时也不失灵活性，可以与其他深度学习 API 无缝对接。

当然，因为实验是在短时间内完成的，我更看中的自然是它的简单易用。官方文档上介绍了不少适用于小数据集的训练思路，并且框架本身对网络结构的搭建也十分简单，这使得我能够快速上手调整网络。

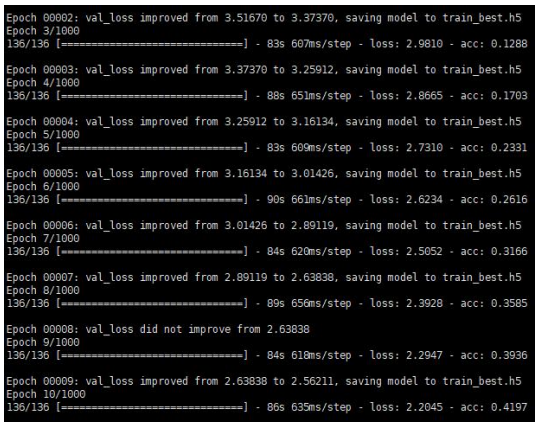
硬件支持 由于我不需要从头训练，而是使用了预训练的模型进行 fine-tune，实际上对于硬件的需求并不是非常高。但为了更好地调整网络，达到更优秀的结果，我还是借到了运算中心

的一台服务器，用 Nvidia 的计算卡进行训练。

与笔记本的 CPU 相比，专业计算卡在性能上无疑有着绝对的优势。在训练中，每个 Epoch 只需要 85s 左右，这比在笔记本上长达 1h 的训练时间快了非常多，因而让我能够反复训练调整模型。



(a) 硬件支持



(b) 训练速度

Fig3: 硬件支持/训练速度

数据集结构 在训练之前，我首先整理了数据集结构。

根据我们最终筛选出的景点，我将各个景点的图片组织成以下形式 (与 Keras 文档实例中的组织结构相同)。

```
data/
    train/
        dogs/
            dog001.jpg
            dog002.jpg
            ...
        cats/
            cat001/jpg
            cat002.jpg
            ...
    validation/
        dogs/
            dog001.jpg
            dog002.jpg
            ...
        cats/
            cat001/jpg
            cat002.jpg
            ...
```

Fig4: 数据集结构

2.2.3 训练过程

在最初我使用的是 VGG16 来训练，但鉴于 VGG16 较为庞大，在之后我改用 RESNET-50 来训练。得益于 Keras 的便利，我可以直接通过网络下载好预训练的权重。此外，Keras 封装好的 API 诸如"flow_from_directory" 使图片数据的读取和处理变得十分简单，因此我就不详细地展开讨论，而直接介绍 fine-tune 的具体细节。

Fine-tune 在最初的尝试中，我冻结了整个全连接层之上的所有层，而仅训练自己的全连接层，最终的结果不是十分理想。在验证集上，最终的准确率只能达到 0.7，这与我们期望的有着不小差距。

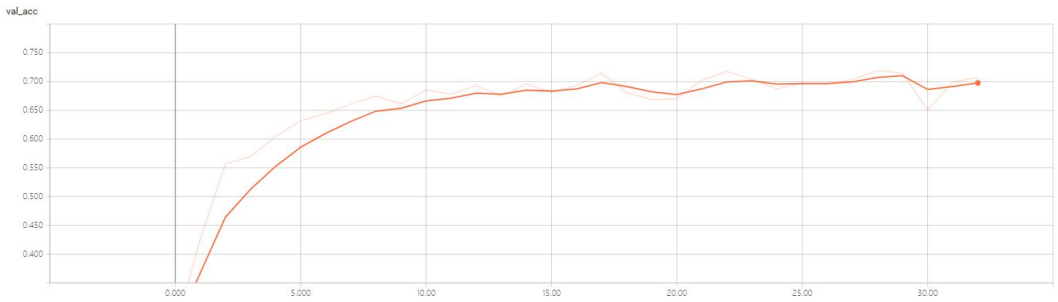


Fig5: 失败的尝试

于是，我尝试更新后面的网络权重，而冻结前大部分的网络。这样做的依据在于，网络的前几层学习到的是图像的基础特征，如角、边缘等，而后几层则是对更加抽象特征的学习。由于预训练的模型是在 ImageNet 上训练的，而 ImageNet 数据集的种类很多，理论上预训练的网络应该有很好的基础特征提取能力，但 ImageNet 的数据中并没有很多景点类型的图像，这使得网络对于图像抽象特征的抓取能力较逊。

为此，我将网络后 1/4 层冻结，并根据我们数据集的分类数目设置了新的 Softmax 层进行训练。训练时配置的优化器为 Adam 优化器，并将 learning rate 设的很小，以满足微调原来已经收敛的模型的需求。

```
    self.__model.layers.pop()
    for i, layer in enumerate(self.__model.layers):
        if i > 160:
            break
        layer.trainable = False

    last_layer = self.__model.layers[-1].output

    classes = list(iter(batches.class_indices))
    x = Dense(len(classes), activation="softmax")(last_layer)

    self.__model_finetuned = Model(self.__model.input, x)
    self.__model_finetuned.compile(optimizer=Adam(lr=0.000001), loss='categorical_crossentropy
                                                    ', metrics=['accuracy'])
    for c in batches.class_indices:
        classes[batches.class_indices[c]] = c
    self.__model_finetuned.classes = classes
```

接着，我设置了一些 callbacks，这也是 Keras 的便利之处。在训练时，模型可以设置提前终止训练，当验证集上的 loss 连续 10 个 Epoch 没有降低时便结束训练；模型也能够设置 checkpoint，在训练时显示必要的信息，并设置在得到更好结果时保存最佳结果；此外，模型自动地将训练过程记录利用 Tensorboard 的 API 保存起来。这些方法的实现也都只需简单的调用。

```
    tensorboard = keras.callbacks.TensorBoard(log_dir)
    early_stopping = EarlyStopping(patience=10)
    checkpointer = ModelCheckpoint('train_best.h5', verbose=1, save_best_only=True)
```

```
    self.__model_finetuned.fit_generator(batches,
        steps_per_epoch=num_train_steps, epochs=1000,
        callbacks=[early_stopping, checkpointer, tensorboard],
        validation_data=val_batches,
```

```
        validation_steps=num_valid_steps)
```

最后，我训练好的模型权重保存在 hdf5 文件当中。Keras 对于 hdf5 文件有着很好的支持，可以直接读取 hdf5 中储存的网络。

2.2.4 建立数据库

将我们的图像整理为易于操作的数据无疑是十分重要的一环。但好在之前已经有过 LSH 的实验课程，在这里我能够直接使用 LSH 的思想实现数据库的建立。

提取特征向量 首先，对于我们数据集中的图片，我对每一个图片都进行了特征向量的提取。该步骤是通过函数"extra_feats" 实现的。

我将网络的 bottleneck 输出 (全连接层的输入) 作为图像特征，提取出 2048 维的特征向量。同一个景点分类下的图像特征信息被保存在同一个 hdf5 文件中，每条信息包含图像文件名以及特征向量，以便于后续的处理。

量化 但显然，如此高的维度是不适合直接进行检索的，2048 维浮点数的运算代价无疑是巨大的。为此，我需要对特征向量进行量化。量化方式为直接将浮点数映射到 0、1、2 三值上，并用"check" 方法检测量化合理性。

但是在这里，我并没有再对量化后的向量进行分桶，因为我们的数据集较小，为了保证图片质量，每个景点中的图片并没有很多，因此我感觉不是很有必要再进行分桶，而直接使用了暴力检索。而在后续的测试中[3]，检索的速度也是保持了较高的水准。

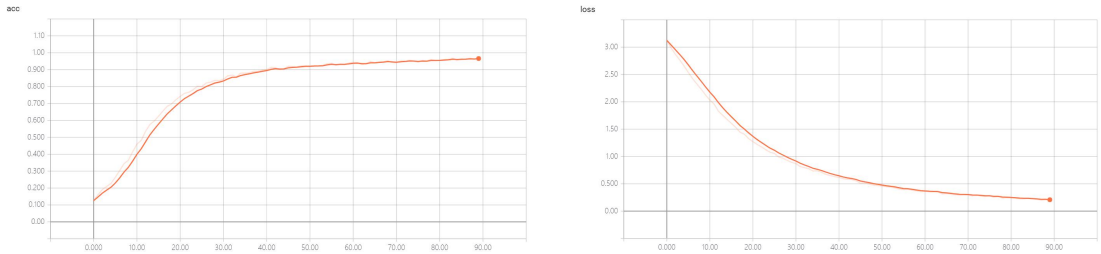
检索过程 有了训练好的网络，检索的实现并不是什么难事。

每张图像可以直接通过输入到网络中得到 bottleneck 特征向量以及 softmax 输出的分类。在检索时，图像将首先根据分类结果被判别为一个景点，这便完成了图像分类的任务。接着，通过访问该类下的图像特征数据进行相似度计算，我们可以返回相似度排名较高的图像文件名，作为后端的输出对接前端。

2.2.5 实验结果

下面展示实验的训练、预测结果。

训练 通过查看 Tensorboard 日志[4]，我们可以直观地观察训练的情况。

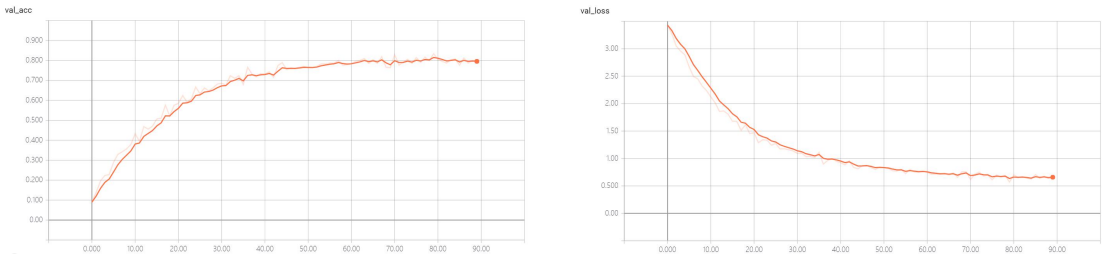


(a) acc (b) loss

Fig6: 训练集

[3]在之后会详细讨论

[4]在文件中附有多次训练日志信息



(a) acc (b) loss

Fig7: 验证集

可以发现，训练集和验证集的结果还是有着不小的差距，说明在我们较小的数据集下出现了过拟合的现象，导致验证集上准确率只能达到 0.84，而训练集上已经收敛。但这个准确率对于我们的实验还是足够的。

预测　经过实践测试，预测的准确度是喜人的。对于给定的待检测图像，模型能够在 3s 左右给出判断的分类，并返回相似度在前列的图片文件名，虽然在速度上可能不算出色[5]，判断的正确率和召回图像的相似性都十分良好。

```
The image is identified as Type 19
top 10 images in order are: [b'407.jpg', b'392.jpg', b'391.jpg', b'50.jpg', b'165.jpg', b'4.jpg', b'91.jpg', b'2.jpg', b'40.jpg', b'327.jpg']
(19, [b'407.jpg', b'392.jpg', b'391.jpg', b'50.jpg', b'165.jpg', b'4.jpg', b'91.jpg', b'2.jpg', b'40.jpg', b'327.jpg'])
3.066809593733965
```

Fig8: 本地测试情况

2.2.6 结果思考

在实验中我遇到了不少问题。经过对结果的分析，我得出了以下一些反思。

特征的进一步简化　实验中我对提取出的特征进行了量化操作，减少了一定的运算量，但向量的维数依然较高，这使得检索的速度相比不进行量化并没有很大的改观。而显然，如果我们的数据集更为庞大，高维的特征向量势必会拖累我们的检索性能。

显然，在前面介绍到的论文中提供了一种较好的解决方案。论文仅保留 48 维/128 维向量，从而大大简化相似度计算，这也确实是文章的精髓之处。

训练分类器　针对检索速度，另一个改进的方法更为直接。

在实验中，我们的每一次预测都是在整个 RESNET-50 网络上进行的。虽然 RESNET-50 相比 VGG16 等模型已经算得上是轻量的，但其参数量依然是巨大的。有没有办法在网络的基础上简化计算呢？

事实上，我们可以选择用网络提取出的特征训练一个传统的分类器 (如 SVM)，从而大幅降低检索代价。但这只停留在分类问题上，对于特征的提取问题，如果需要通过深度学习实现，似乎只能通过使用更小的网络来实现。

[5]在结果思考中由相关的讨论