图像检索：layer选择与fine-tuning性能提升验证

2017年05月30日 Image Retrieval CBIR 字数:24512

> *这个世界上肯定有另一个我，做着我不敢做的事，过着我想过的生活。一个人逛街，一个人吃饭，一个人旅行，一个人做很多事。极致的幸福，存在于孤独的深海。在这样日复一日的生活里，我逐渐和自己达成和解。*

作为迁移学习的一种，finetune能够将general的特征转变为special的特征，从而使得转移后的特征能够更好的适应目标任务，而图像检索最根本的问题，仍在于如何在目标任务上获得更好的特征表达(共性与可区分性)。一种很自然的方式便是在特定的检索任务上，我们对imageNet学得的general的特征通过finetune的方式，使得表达的特征能够更好的适应我们的检索任务。在End-to-end Learning of Deep Visual Representations for Image Retrieval和 Collaborative Index Embedding for Image Retrieval中已经很清楚的指出，通过基本的classification loss的finetune的方式，能够较大幅度的提高检索的mAP。因此，在本篇博文中，小白菜针对检索，主要整理了下面四个方面的内容：

- CNN网络中哪一层最适合于做图像检索
- 基于pre-trained模型做图像检索几种典型的特征表示方法
- 抽取网络任意层的特征
- 数据增强(Data Augmentation)
- VGGNet16网络模型fine-tuning实践

在采用深度学习做检索的时候，上面四方面的问题和知识基本都回避不了，因此，小白菜以为，掌握这四方面的内容显得非常有必要。

特征表达layer选择

在AlexNet和VGGNet提出伊始，对于检索任务，小白菜相信，在使用pre-trained模型抽取特征的时候，我们最最自然想到的方式是抽取全连接层中的倒数第一层或者倒数第二层的特征，这里说的倒数第一层或者倒数第二层并没有具体指明是哪一层（fcx、fcx_relux、fcx_dropx），以VggNet16网络为例，全连接层包含两层，fc6和fc7，因此我们很自然想到的网络层有fc6、fc6_relu6、fc7、fc7_relu7甚至fc6_drop6和fc7_drop7（后面会说明fc6_drop6和fc6_relu6是一样的，以及fc7_drop7和fc7_relu7也是一样的），所以即便对于我们最最自然最最容易想到的方式，也面临layer的选择问题。为此，我们以VGGNet16网络为例，来分析CNN网络的语义层(全连接层)选择不同层作为特征做object retrieval的mAP的影响。

小白菜选取fc6、fc6_relu6、fc7、fc7_relu7这四层语义层的特征，在Oxford Building上进行实验，评价指标采用mAP，mAP的计算采用Oxford Building提供的计算mAP代码compute_ap.cpp，下表是fc6、fc6_relu6、fc7、fc7_relu7对应的mAP。

layer	mAP(128维)	mAP(4096维)	mAP(4096维, 未做PCA)
fc7_relu7	44.72%	1.11%	41.08%
fc7	45.03%	19.67%	41.18%
fc6_relu6	43.62%	23.0%	43.34%
fc6	45.9%	19.47%	44.78%

从上表可以看到，直接采用pre-trained模型抽取语义层的特征，在Oxford Building上取得的结果在45%左右，同时我们还可以看出，选取fc6、fc6_relu6、fc7、fc7_relu7对结果的影响并不大。这个结果只能说非常的一般，在基于pre-trained模

型做object retrieval的方法中，比如Cross-dimensional Weighting for Aggregated Deep Convolutional Features、Particular object retrieval with integral max-pooling of CNN activations以及What Is the Best Practice for CNNs Applied to Visual Instance Retrieval?指出，选用上层的语义层其实是不利于object retrieval，因为上层的语义层丢失了object的空间信息，并且从实验的角度说明了**选取中间层的特征**更利于object retrieval。

实际上，在选取中间层来表达特征的过程中，我们可以去掉全连接层，从而使得我们可以摆脱掉输入图像尺寸约束(比如224*224)的约束，而保持原图大小的输入。通常，图像分辨率越大，对于分类、检测等图像任务是越有利的。因而，从这一方面讲，**选取上层的全连接层作为特征，并不利于我们的object retrieval任务**。一种可能的猜想是，上层全连接层的语义特征，应该更适合做全局的相似。

虽然中间层更适合于做object retrieval，但是在选用中间层的feature map作为raw feature的时候，我们面临的一个主要问题是：如何将3d的tensor转成一个有效的向量特征表示？下面小白菜主要针对这一主要问题总结几种典型的特征表示方法，以及对中间层特征选择做一些探讨与实验。

基于pre-trained模型做Object Retrieval几种典型的特征表示

SUM pooling

基于SUM pooling的中层特征表示方法，指的是针对中间层的任意一个channel（比如VGGNet16, pool5有512个channel），将该channel的feature map的所有像素值求和，这样每一个channel得到一个实数值，N个channel最终会得到一个长度为N的向量，该向量即为SUM pooling的结果。

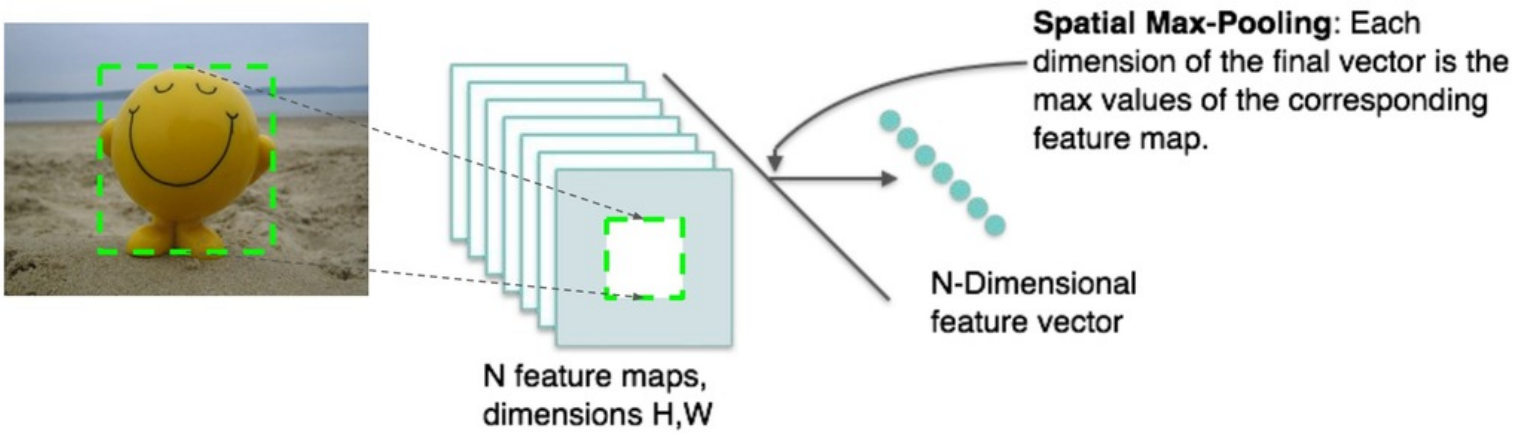
AVE pooling

AVE pooling就是average pooling，本质上它跟SUM pooling是一样的，只不过是将像素值求和后还除以了feature map的尺寸。小白菜以为，**AVE pooling可以带来一定意义上的平滑，可以减小图像尺寸变化的干扰**。设想一张*224224的图像，将其resize到448*448后，分别采用SUM pooling和AVE pooling对这两张图像提取特征，我们猜测的结果是，SUM

pooling计算出来的余弦相似度相比于AVE pooling算出来的应该更小，也就是AVE pooling应该稍微优于SUM pooling一些。

MAX pooling

MAX pooling指的是对于每一个channel（假设有N个channel），将该channel的feature map的像素值选取其中最大值作为该channel的代表，从而得到一个N维向量表示。小白菜在flask-keras-cnn-image-retrieval中采用的正是MAX pooling的方式。



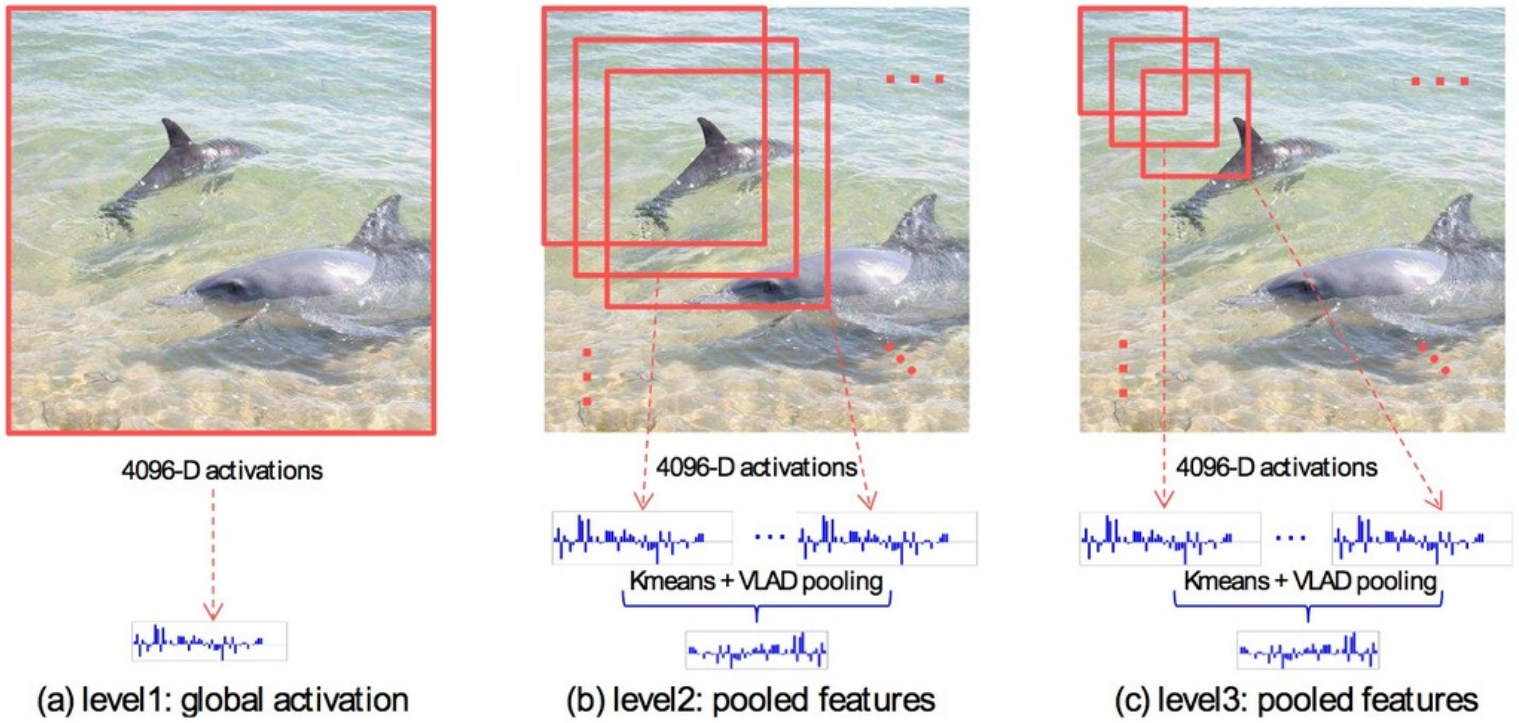
> *from Day 2 Lecture 6 Content-based Image Retrieval*

上面所总结的SUM pooling、AVE pooling以及MAX pooling，这三种pooling方式，在小白菜做过的实验中，MAX pooling要稍微优于SUM pooling、AVE pooling。不过这三种方式的pooling对于object retrieval的提升仍然有限。

MOP pooling

MOP Pooling源自Multi-scale Orderless Pooling of Deep Convolutional Activation Features这篇文章，一作是Yunchao Gong，此前在搞哈希的时候，读过他的一些论文，其中比较都代表性的论文是ITQ，小白菜还专门写过一篇笔记论文阅

读：Iterative Quantization迭代量化。MOP pooling的基本思想是多尺度与VLAD(VLAD原理可以参考小白菜之前写的博文图像检索：BoF、VLAD、FV三剑客)，其具体的pooling步骤如下：



> *from Multi-scale Orderless Pooling of Deep Convolutional Activation Features*
>
> *Overview of multi-scale orderless pooling for CNN activations (MOP-CNN). Our proposed feature is a concatenation of the feature vectors from three levels: (a)Level 1, corresponding to the 4096-dimensional CNN activation for the entire 256⊴256image; (b) Level 2, formed by extracting activations from 128⊴128 patches and VLADpooling them with a codebook of 100 centers; (c) Level 3, formed in the same way aslevel 2 but with 64⊴*64 patches.*

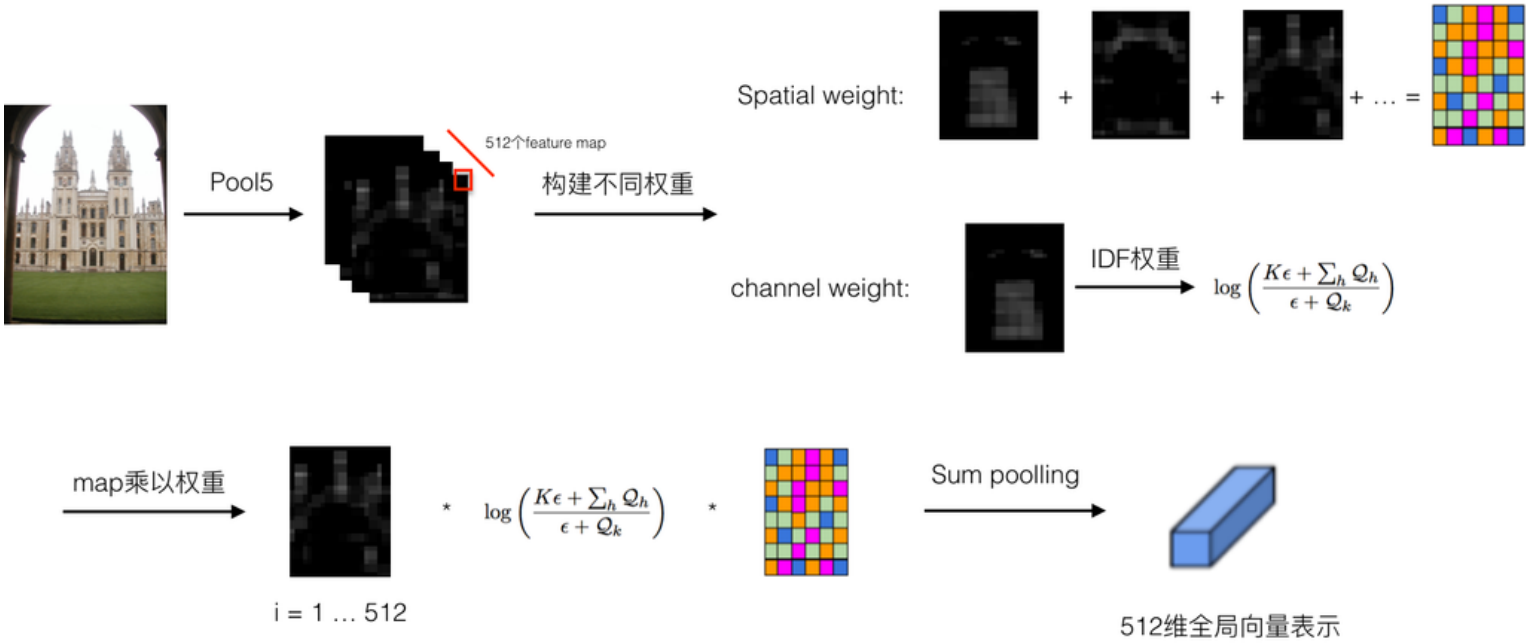
具体地，在L=1的尺度下，也就是全图，直接resize到256*256的大小，然后送进网络，得到第七层全连接层4096维的特征；在L=2时，使用128*128(步长为32)的窗口进行滑窗，由于网络的图像输入最小尺寸是256*256，所以作者将其上采样到256*256，这样可以得到很多的局部特征，然后对其进行VLAD编码，其中聚类中心设置为100，4096维的特征降到了

500维，这样便得到了50000维的特征，然后将这50000维的特征再降维得到4096维的特征；L=3的处理过程与L=2的处理过程一样，只不过窗口的大小编程了64*64的大小。

作者通过实验论证了MOP pooling这种方式得到的特征一定的不变性。基于这种MOP pooling小白菜并没有做过具体的实验，所以实验效果只能参考论文本身了。

CROW pooling

对于Object Retrieval，在使用CNN提取特征的时候，我们所希望的是在有物体的区域进行特征提取，就像提取局部特征比如SIFT特征构BoW、VLAD、FV向量的时候，可以采用MSER、Saliency等手段将SIFT特征限制在有物体的区域。同样基于这样一种思路，在采用CNN做Object Retrieval的时候，我们有两种方式来更细化Object Retrieval的特征：一种是先做物体检测然后在检测到的物体区域里面提取CNN特征；另一种方式是我们通过某种权重自适应的方式，加大有物体区域的权重，而减小非物体区域的权重。CROW pooling（Cross-dimensional Weighting for Aggregated Deep Convolutional Features）即是采用的后一种方法，通过构建Spatial权重和Channel权重，CROW pooling能够在**一定程度上**加大感兴趣区域的权重，降低非物体区域的权重。其具体的特征表示构建过程如下图所示：



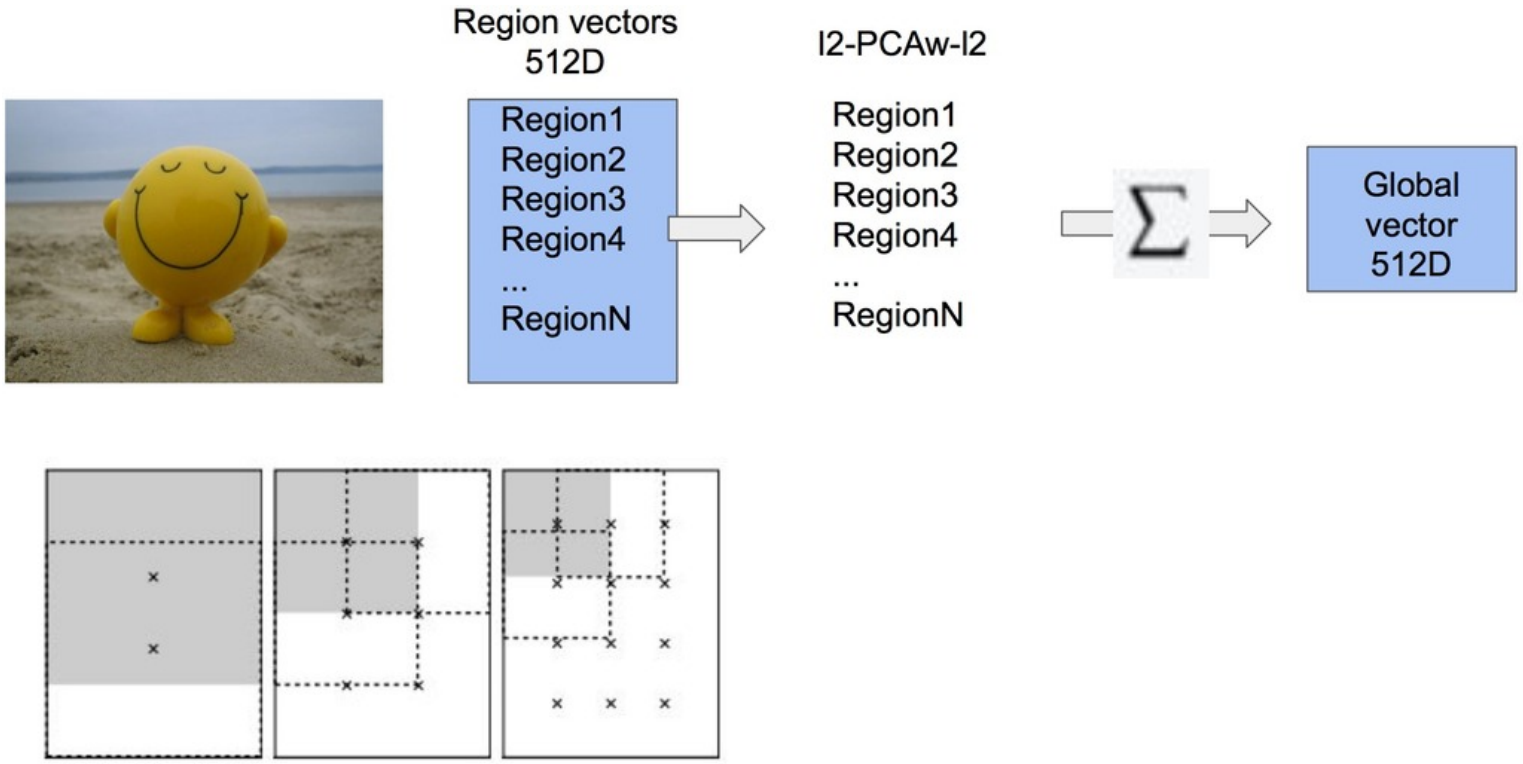
其核心的过程是Spatial Weight和Channel Weight两个权重。Spatial Weight具体在计算的时候，是直接对每个channel的feature map求和相加，这个Spatial Weight其实可以理解为saliency map。我们知道，通过卷积滤波，响应强的地方一般都是物体的边缘等，因而将多个通道相加求和后，那些非零且响应大的区域，也一般都是物体所在的区域，因而我们可以将它作为feature map的权重。Channel Weight借用了IDF权重的思想，即对于一些高频的单词，比如“the”，这类词出现的频率非常大，但是它对于信息的表达其实是没多大用处的，也就是它包含的信息量太少了，因此在BoW模型中，这类停用词需要降低它们的权重。借用到Channel Weight的计算过程中，我们可以想象这样一种情况，比如某一个channel，其feature map每个像素值都是非零的，且都比较大，从视觉上看上去，白色区域占据了整个feature map，我们可以想到，这个channel的feature map是不利于我们去定位物体的区域的，因此我们需要降低这个channel的权重，而对于白色区域占feature map面积很小的channel，我们认为它对于定位物体包含有很大的信息，因此应该加大这种channel的权重。而这一现象跟IDF的思想特别吻合，所以作者采用了IDF这一权重定义了Channel Weight。

总体来说，这个Spatial Weight和Channel Weight的设计还是非常巧妙的，不过这样一种pooling的方式只能在一定程度上契合感兴趣区域，我们可以看一下Spatial Weight*Channel Weight的热力图：

从上面可以看到，权重大的部分主要在塔尖部分，这一部分可以认为是discriminate区域，当然我们还可以看到，在图像的其他区域，还有一些比较大的权重分布，这些区域是我们不想要的。当然，从小白菜可视化了一些其他的图片来看，这种crow pooling方式并不总是成功的，也存在着一些图片，其权重大的区域并不是图像中物体的主体。不过，从千万级图库上跑出来的结果来看，crow pooling这种方式还是可以取得不错的效果。

RMAC pooling

RMAC pooling的池化方式源自于Particular object retrieval with integral max-pooling of CNN activations，三作是Hervé Jégou(和Matthijs Douze是好基友)。在这篇文章中，作者提出来了一种RMAC pooling的池化方式，其主要的思想还是跟上面讲过的MOP pooling类似，采用的是一种变窗口的方式进行滑窗，只不过在滑窗的时候，不是在图像上进行滑窗，而是在feature map上进行的(极大的加快了特征提取速度)，此外在合并local特征的时候，MOP pooling采用的是VLAD的方式进行合并的，而RMAC pooling则处理得更简单(简单并不代表效果不好)，直接将local特征相加得到最终的global特征。其具体的滑窗方式如下图所示:



> *from Day 2 Lecture 6 Content-based Image Retrieval*

图中示意的是三种窗口大小，图中‘x’代表的是窗口的中心，对于每一个窗口的feature map，论文中采用的是MAX pooling的方式，在L=3时，也就是采用图中所示的三种窗口大小，我们可以得到20个local特征，此外，我们对整个fature map做一次MAX pooling会得到一个global特征，这样对于一幅图像，我们可以得到21个local特征(如果把得到的global特征也视为local的话)，这21个local特征直接相加求和，即得到最终全局的global特征。论文中作者对比了滑动窗口数量对mAP的影响，从L=1到L=3，mAP是逐步提升的，但是在L=4时，mAP不再提升了。实际上RMAC pooling中设计的窗口的作用是定位物体位置的(CROW pooling通过权重图定位物体位置)。如上图所示，在窗口与窗口之间，都是一定的overlap，而最终在构成global特征的时候，是采用求和相加的方式，因此可以看到，那些重叠的区域我们可以认为是给予了较大的权重。

上面说到的20个local特征和1个global特征，采用的是直接合并相加的方式，当然我们还可以把这20个local特征相加后再跟剩下的那一个global特征串接起来。实际实验的时候，发现串接起来的方式比前一种方式有2%-3%的提升。在规模100万的图库上测试，RMAC pooling能够取得不错的效果，跟Crow pooling相比，两者差别不大。

上面总结了6中不同的pooling方式，当然还有很多的pooling方式没涵盖不到，在实际应用的时候，小白菜比较推荐采用RMAC pooling和CROW pooling的方式，主要是这两种pooling方式效果比较好，计算复杂度也比较低。

抽取网络任意层的特征

在上面一节中，我们频繁的对网络的不同层进行特征的抽取，并且我们还提到fc6_dropx和fc6_relux是一样的（比如fc7_drop7和fc7_relu7是一样的），这一节主要讲述使用Caffe抽取网络任意一层的特征，并从实验的角度验证fc6_dropx和fc6_relux是一样的这样一个结论。

为了掌握Caffe中网络任意一层的特征提取，不妨以一个小的题目来说明此问题。题目内容为：给定VGGNet16网络，抽取fc7、fc7_relu7以及fc7_drop7层的特征。
求解过程：VggNet16中deploy.txt中跟fc7相关的层如下：

```
layers {
  bottom: "fc6"
  top: "fc7"
  name: "fc7"
  type: INNER_PRODUCT
  inner_product_param {
    num_output: 4096
  }
}
layers {
  bottom: "fc7"
  top: "fc7"
  name: "relu7"
  type: RELU
}
layers {
  bottom: "fc7"
  top: "fc7"
  name: "drop7"
  type: DROPOUT
  dropout_param {
    dropout_ratio: 0.5
  }
}
```

如果使用`net.blobs['fc7'].data[0]`，我们抽取的特征是fc7层的特征，也就是上面：

```
layers {
  bottom: "fc6"
  top: "fc7"
  name: "fc7"
  type: INNER_PRODUCT
```

```
  inner_product_param {
    num_output: 4096
  }
}
```

这一层的特征，仿照抽取fc7特征抽取的代码，我们很自然的想到抽取relu7的特征为`net.blobs['relu7'].data[0]`和drop7的特征为`net.blobs['drop7'].data[0]`，但是在运行的时候提示不存在`relu7`层和`drop7`层，原因是：

> *To elaborate a bit further: The layers drop7 and relu7 have the same blobs as top and bottom, respectively, and as such the blobs' values are manipulated directly by the layers. The advantage is saving a bit of memory, with the drawback of not being able to read out the state the values had before being fed through these two layers. It is simply not saved anywhere. If you want to save it, you can just create another two blobs and re-wire the layers a bit.*

摘自Extracting 'relu' and 'drop' blobs with pycaffe，因而，为了能够提取relu7和drop7的特征，我们需要将上面的配置文件做些更改，主要是将layers里面的字段换下名字(在finetune模型的时候，我们也需要做类似的更改字段的操作)，这里小白菜改成了：

```
layers {
  bottom: "fc6"
  top: "fc7"
  name: "fc7"
  type: INNER_PRODUCT
  inner_product_param {
    num_output: 4096
  }
}
layers {
  bottom: "fc7"
  top: "fc7_relu7"
  name: "fc7_relu7"
```

```
  type: RELU
}
layers {
  bottom: "fc7"
  top: "fc7_drop7"
  name: "fc7_drop7"
  type: DROPOUT
  dropout_param {
    dropout_ratio: 0.5
  }
}
```

经过这样的修改后，我们使用`net.blobs['fc7_relu7'].data[0]`即可抽取到relu7的特征，使用`net.blobs['fc7_drop7'].data[0]`可抽取到drop7的特征。

数据增强

有了上面**CNN网络中哪一层最适合于做图像检索**、**基于pre-trained模型做图像检索几种典型的特征表示方法**以及**抽取网络任意层的特征**三方面的知识储备后，在具体fine-tuning网络进行图像检索实验前，还有一节很重要(虽然我们都很熟悉)内容，即数据增强(Data Augmentation)。数据增强作用有二：一是均衡样本，比如某些类别只有几张图片，而有的类别有上千张，如果不做均衡，分类的时候计算的分类准确率会向样本充足的类别漂移；二是提高网络对于样本旋转、缩放、模糊等的鲁棒性，提高分类精度。在实际工作中，我们拿到了图像数据样本对采用深度学习模型而言，经常是不充足且不均衡的，所以这一步数据的前置处理是非常重要的。

在正式开始数据增强之前，对**图片进行异常检测是非常重要的**，其具体的异常表现在图片内容缺失、图片不可读取或者可以读取但数据出现莫名的问题，举个例子，比如通过爬虫爬取的图片，可能上半部分是正常的，下半分缺失一片灰色。因此，如果你不能确保你训练的图片数据都是可正常读取的时候，最好对图片做异常检测。假设你的训练图片具有如下层级目录：

```
➜  imgs_diff tree -L 2
.
├── 0
│   ├── 1227150149_1.jpg
│   ├── 1612549977_1.jpg
│   ├── 1764084098_1.jpg
│   ├── 1764084288_1.jpg
│   └── 1764085346_1.jpg
└── 1
    ├── 1227150149_1.jpg
    ├── 1612549977_1.jpg
    ├── 1764084098_1.jpg
    ├── 1764084288_1.jpg
    └── 1764085346_1.jpg
```

下面是小白菜参考网上资料写的图片异常检测代码如下：

```
from PIL import Image
import glob
import cv2
import os
import numpy as np

from PIL import Image

def check_pic(path):
    try:
        Image.open(path).load()
    except:
        print 'ERROR: %s' % path
        return False
    else:
```

```
        return True


imgs_list = glob.glob('/raid/yuanyong/neuralcode/ncdata/*/*')

for img_path in imgs_list:
    #img = Image.open(img_path)
    #if img.verify() is not None or img is None:
    #    print img_path
    try:
        img = Image.open(img_path)
        im = img.load()
    except IOError:
        print 'ERROR: %s' % img_path
        continue
    try:
        img = np.array(img, dtype=np.float32)
    except SystemError:
        print 'ERROR: %s' % img_path
        continue
    if len(img.shape) != 3:
        print 'ERROR: %s' % img_path
```

通过上面的图片异常检测，我们可以找到那些不可读取或者读取有问题的图片找出来，这样在我们使用Caffe将图片转为LMDB数据存储的时候，不会出现图片读取有问题的异常。在图片异常检测完成后，便可以继续后面的数据增强了。

在小白菜调研的数据增强工具中，小白菜以为，最好用的还是Keras中的数据增强。Keras数据增强部分包含在image.py，通过类`ImageDataGenerator`可以看到Keras包含了对图片的不同处理，下面是小白菜基于Keras写的数据增强脚本，假设你的图像数据目录结构具有如下结构：

```
➜  imgs_dataset tree -L 2
.
├── 0
│   ├── 1227150149_1.jpg
│   ├── 1612549977_1.jpg
│   ├── 1764084098_1.jpg
│   ├── 1764084288_1.jpg
│   └── 1764085346_1.jpg
└── 1
    ├── 1227150149_1.jpg
    ├── 1612549977_1.jpg
    ├── 1764084098_1.jpg
    ├── 1764084288_1.jpg
    └── 1764085346_1.jpg
    ....
```

数据增强的脚本如下：

```
#!/usr/bin/env python
#-*- coding: utf-8 -*-
#Author: yuanyong.name

import os
import random
from keras.preprocessing.image import ImageDataGenerator, array_to_img, img_to_array, load_img

num_wanted = 800
target_path_dir = '/raid/yuanyong/neuralcode/ncdata'

datagen = ImageDataGenerator(
        rotation_range = 10,
        width_shift_range = 0.2,
```

```
        height_shift_range = 0.2,
        shear_range = 0.2,
        zoom_range = 0.2,
        horizontal_flip = True,
        fill_mode = 'nearest')

sub_dirs = os.walk(target_path_dir).next()[1]
for sub_dir in sub_dirs:
    sub_dir_full = os.path.join(target_path_dir, sub_dir)
    img_basenames = os.listdir(sub_dir_full)
    num_imgs = len(img_basenames)
    num_perAug = int(float(num_wanted)/float(num_imgs)) - 1
    if num_imgs >= num_wanted:
        continue
    num_total = 0
    for i, img_basename in enumerate(img_basenames):
        num_total = num_imgs + i*num_perAug
        if num_total >= num_wanted:
            break
        img_path = os.path.join(sub_dir_full, img_basename)
        #print "Aug: %s" % img_path
        img = load_img(img_path) # this is a PIL image, please replace to your own file path
        if img == None:
            continue
        try:
            x = img_to_array(img) # this is a Numpy array with shape (3, 150, 150)
            x = x.reshape((1,) + x.shape) # this is a Numpy array with shape (1, 3, 150, 150)
            i = 0
            for batch in datagen.flow(x, batch_size = 1, save_to_dir = sub_dir_full, save_prefi
x = 'aug', save_format = 'jpg'):
                i += 1
                if i >= num_perAug:
                    break # otherwise the generator would loop indefinitely
        except:
```

```
            print "%s" % img_path

# delete extra aug images
for sub_dir in sub_dirs:
    sub_dir_full = os.path.join(target_path_dir, sub_dir)
    img_basenames = os.listdir(sub_dir_full)
    num_imgs = len(img_basenames)
    if num_imgs <= num_wanted:
        continue
    aug_imgs = [img_basename for img_basename in img_basenames if img_basename.startswith('aug
_')]
    random.shuffle(aug_imgs, random.random)
    num_del = num_imgs - num_wanted
    aug_imgs_del = aug_imgs[-num_del:]

    for img_basename in aug_imgs_del:
        img_full = os.path.join(sub_dir_full, img_basename)
        os.remove(img_full)
```

`target_path_dir`是设置你训练数据集目录的，`num_wanted`是设置每一个类你想要多少个样本(包括原始样本在内)，比如在Neural Codes for Image Retrieval中的数据集landmark数据集，里面的类别少的样本只有9个，多的类别样本有上千个，样本部分极其不均匀，上面代码中，小白菜设置得是`num_wanted=800`，即让每一个类别的样本数据控制在800左右(多出来的类别样本不会删除，比如有的类别可能原始样本就有1000张，那么不会对该类做数据增强)。

在这一节，小白菜总结了为什么要数据增强，或者说数据增强有什么好处，然后提供了两个脚本：图片异常检测脚本check_image.py和数据增强脚本keras_imgAug.py。有了前面三部分的知识和本节数据前置处理的实践，我们终于可以进行fine-tuning了。

VGGNet16网络模型fine-tuning实践

在实际中，用CNN做分类任务的时候，一般我们总是用在ImageNet上预训练好的模型去初始化待训练模型的权重，也就是不是train from scratch，主要原因有二：一是在实际中很难获取到大量样本(即便是做了数据增强)；二是加快模型训练的速度。因而，针对检索这个任务，我们也采用fine-tuning的方式，让在ImageNet上预训练的模型迁移到我们自己的特定的数据集上，从而提升特征在检索任务上的表达能力。下面小白菜以fine-tuning Neural Codes提供的数据集为例，比较详实的总结一个完整的fine-tuning过程。

在fine-tuning之前，我们先追问一个简单的问题和介绍一下Neural Codes提供的landmark数据集。追问的这个问题很简单：为什么几乎所有的做检索的论文中，使用的都是AlexNet、VGGNet16（偶尔会见到一两篇使用ResNet101）网络模型？难道做研究的只是关注方法，使用AlexNet、VGGNet、ResNet或者Inception系列只是替换一下模型而已？小白菜曾也有过这样的疑问，但是对这些网络测试下来，发觉VGGNet在做基于预训练模型特征再表达里面效果是最好的，对于同一个方法，使用ResNet或Inception系列，其mAP反而没有VGGNet的高。至于为什么会这样，小白菜也没有想明白(如果有小伙伴知道，请告知)，我们就暂且把它当做一条经验。

我们再对Neural Codes论文里提供的landmark数据集做一个简单的介绍。该数据集共有680类，有的类别样本数据至于几个，多则上千，样本分布极其不均匀。不过这不是问题，通过第4节介绍的数据增强和提供的脚本，我们可以将每个类别的样本数目控制在800左右。同时，我们可以使用下面脚本将每个类别所在目录的文件夹名字命名为数字：

```
import os

path_ = '/raid/yuanyong/neuralcode/ncdata' # 该目录下有很多子文件夹，每个子文件夹是一个类别
dirs = os.walk(path_).next()[1]

dirs.sort()

for i, dir_ in enumerate(dirs):
    print '%d, %s' %(i, dir_)
    os.rename(os.path.join(path_, dir_), os.path.join(path_, str(i)))
```

得到清洗后的数据后，下面分步骤详解使用VGGNet16来fine-tuning的过程。

切分数据集为train和val

经过上面重命名后的数据集文件夹目录具有如下形式：

```
➜  imgs_dataset tree -L 2
.
├── 0
│   ├── 1227150149_1.jpg
│   ├── 1612549977_1.jpg
│   ├── 1764084098_1.jpg
│   ├── 1764084288_1.jpg
│   └── 1764085346_1.jpg
│   ....
└── 1
    ├── 1227150149_1.jpg
    ├── 1612549977_1.jpg
    ├── 1764084098_1.jpg
    ├── 1764084288_1.jpg
    └── 1764085346_1.jpg
    ....
```

我们需要将数据集划分为train数据集和val数据集，注意val数据集并不单纯只是在训练的时候测试一下分类的准确率。为了方便划分数据集，小白菜写了如下的脚本，可以很方便的将数据集划分为train数据集和val数据集：

```
import os
import glob
import argparse
import numpy as np
import random

classes_path  = glob.glob('/raid/yuanyong/neuralcode/ncdata/*') # 该目录下有很多子文件夹，每个子文件
```

```
夹train_samples = []
val_samples = []

imgs_total = 0

for i, class_path in enumerate(classes_path):
    class_ = class_path.split('/')[-1]
    imgs_path = glob.glob(class_path + '//*')
    num_imgs = len(imgs_path)
    num_train = int(num_imgs*0.6) # 训练集占60%
    num_val = num_imgs - num_train

    np.random.seed(1024)
    sample_idx = np.random.choice(range(num_imgs), num_imgs, replace=False)
    train_idx = sample_idx[0:num_train]
    val_idx = sample_idx[num_train:]

    for idx_ in train_idx:
        img_path = imgs_path[idx_]
        train_samples.append(img_path + ' ' + class_ + '\n')

    for idx_ in val_idx:
        img_path = imgs_path[idx_]
        val_samples.append(img_path + ' ' + class_ + '\n')

    imgs_total += num_imgs


random.shuffle(train_samples)
random.shuffle(val_samples)

with open('lmdb/train.txt', 'w') as f_train, open('lmdb/val.txt', 'w') as f_val:
    for sample in train_samples:
        f_train.write(sample)
```

运行上面脚本，会在lmdb目录下(事先需要建立lmdb目录)生成两个文本文件，分别为`train.txt`和`val.txt`，对于为训练数据集和验证数据集。

图片转成lmdb存储

为了提高图片的读取效率，Caffe将图片转成lmdb进行存储，在上面得到`train.txt`和`val.txt`后，我们需要借助caffe的`convert_imageset`工具将图片resize到某一固定的尺寸，同时转成为lmdb格式存储。下面是小白菜平时使用的完成该任务的一个简单脚本`crop.sh`

```
/home/yuanyong/caffe/build/tools/convert_imageset \
         --resize_height 256 \
         --resize_width 256 \
         / \
         lmdb/train.txt \
         lmdb/train_lmdb
```

运行两次，分别对应于`train.txt`和`val.txt`。运行完后，会在lmdb目录下生成`train_lmdb`和`val_lmdb`两目录，为了校验转成lmdb存储是否成功，我们最好分别进入这两个目录下看看文件的大小以做简单的验证。

生成均值文件

对于得到的`train_lmdb`，我们在其上计算均值。具体地，使用Caffe的`compute_image_mean`工具：

```
$CAFFE_ROOT/build/tools/compute_image_mean lmdb/train_lmdb lmdb/mean.binaryproto
```

在lmdb目录下即可得到均值文件`mean.binaryproto`。

修改train_val.prototxt和solver.prototxt

针对VGGNET16网络，在fine-tuning的时候，我们通常将最后的分类层的学习率设置得比前面网络层的要大，一般10倍左右。当然，我们可以结合自己的需要，可以将前面层的学习率都置为0，这样网络在fine-tuning的时候，只调整最后一层分类层的权重；在或者我们分两个阶段去做fine-tuning，第一阶段只fine-tuning最后的分类层，第二阶段正常的fine-tuning所有的层(包含最后的分类层)。同时，我们还需要对最后一层分类层重新换个名字，并且对应的分类输出类别也需要根据自己数据集的分类类别数目做调整，下面小白菜给出自己在fine-tuning Neural Codes的landmark数据集上train_val.prototxt的前面输入部分和后面分类的部分：

```
name: "VGG_ILSVRC_16_layers"
layers {
  name: "data"
  type: DATA
  include {
    phase: TRAIN
  }
 transform_param {
    crop_size: 224
    mean_file: "/raid/yuanyong/neuralcode/lmdb/mean.binaryproto"
    mirror: true
 }
 data_param {
    source: "/raid/yuanyong/neuralcode/lmdb/train_lmdb"
    batch_size: 64
    backend: LMDB
  }
  top: "data"
  top: "label"
}
layers {
  name: "data"
  type: DATA
  include {
```

```
    phase: TEST
  }
  transform_param {
    crop_size: 224
    mean_file: "/raid/yuanyong/neuralcode/lmdb/mean.binaryproto"
    mirror: false
  }
  data_param {
    source: "/raid/yuanyong/neuralcode/lmdb/val_lmdb"
    batch_size: 52
    backend: LMDB
  }
  top: "data"
  top: "label"
}

...
...
...

layers {
  name: "fc8_magic"     # 改名字
  type: INNER_PRODUCT
  bottom: "fc7"
  top: "fc8_magic"      # 改名字
  blobs_lr: 10          # 学习率是前面网络层是10倍
  blobs_lr: 20          # 学习率是前面网络层是10倍
  weight_decay: 1
  weight_decay: 0
  inner_product_param {
    num_output: 680     # 共680类
    weight_filler {
      type: "gaussian"
      std: 0.01
```

```
    }
    bias_filler {
      type: "constant"
      value: 0
    }
  }
}
layers {
  name: "loss"
  type: SOFTMAX_LOSS
  bottom: "fc8_magic"   # 改名字
  bottom: "label"
  top: "loss/loss"
}
layers {
  name: "accuracy"
  type: ACCURACY
  bottom: "fc8_magic"   # 改名字
  bottom: "label"
  top: "accuracy"
  include: { phase: TEST }
}
```

上面配置测试输入的时候，`batch_size`设置的是52，这个设置非常重要，我们一定要保证这个设置的`batch_size`跟solver.prototxt里面设置的`test_iter`乘起来等于测试样本数目。下面再看看solver.prototxt这个文件：

```
net: "train_val.prototxt"
test_iter: 4005
test_interval: 5000
base_lr: 0.001
lr_policy: "step"
gamma: 0.1
```

```
stepsize: 20000
display: 1000
max_iter: 50000
momentum: 0.9
weight_decay: 0.0005
snapshot: 5000
snapshot_prefix: "../models/snapshots_"
solver_mode: GPU
```

比较重要的5个需要调整参数分别是`test_iter`、`test_interval`、`base_lr`、`stepsize`，`momentum`。`test_iter`怎么设置上面已经介绍。`test_interval`表示迭代多少次进行一次验证及测试，`base_lr`表示基础学习率，一般要比正常训练时的学习率要小，`stepsize`跟步长相关，可以简单的理解为步长的分母，`momentum`按推荐设置为0.9就可以。

设置完上面的train_val.prototxt和solver.prototxt后，便可以开始正式fine-tuning了。

正式fine-tuning

借助Caffe工具集下的caffe，我们只需要简单的执行下面命令即可完成网络的fine-tuning:

```
$CAFFE_ROOT/build/tools/caffe train -solver  solver.prototxt -weights http://www.robots.ox.ac.u
k/~vgg/software/very_deep/caffe/VGG_ILSVRC_16_layers.caffemodel -gpu 0,1 | tee log.txt
```

其中`-gpu`后面接的数字表示GPU设备的编号，这里我们使用了0卡和1卡，同时我们将训练的日志输出到log.txt里面。

测试

完成了在Neural Codes的landmark数据集上的fine-tuning后，我们使用经过了fine-tuning后的模型在Oxford Building数据集上mAP提升了多少。为了方便对比，我们仍然提取fc6的特征，下面是不做ft(fine-tuning)和做ft的结果对比：

layer	mAP(128维)
fc6	45.9%
fc6+ft	60.2%

可以看到，经过fine-tuning，mAP有了较大幅度的提升。从而也从实验的角度验证了对于检索任务，在数据允许的条件，对预训练模型进行fine-tuning显得非常的有必要。

复现本文实验

如想复现本文实验，可以在这里fc_retrieval找到相应的代码。

总结

在本篇博文中，小白菜就5个方面的问题展开了总结和整理，分别是：

- CNN网络中哪一层最适合于做图像检索
- 基于pre-trained模型做图像检索几种典型的特征表示方法
- 抽取网络任意层的特征
- 数据增强(Data Augmentation)
- VGGNet16网络模型fine-tuning实践

整个文章的基本组织结构依照典型的工科思维方式进行串接，即从理论到实践。

←图像检索：再叙ANN Search 知行手记：毕业一周年→

comments powered by Disqus

Friend: Lichao Yihui Xuezhi 52ml cyq Rui Hu pyimagesearch bean ml dairy Resources