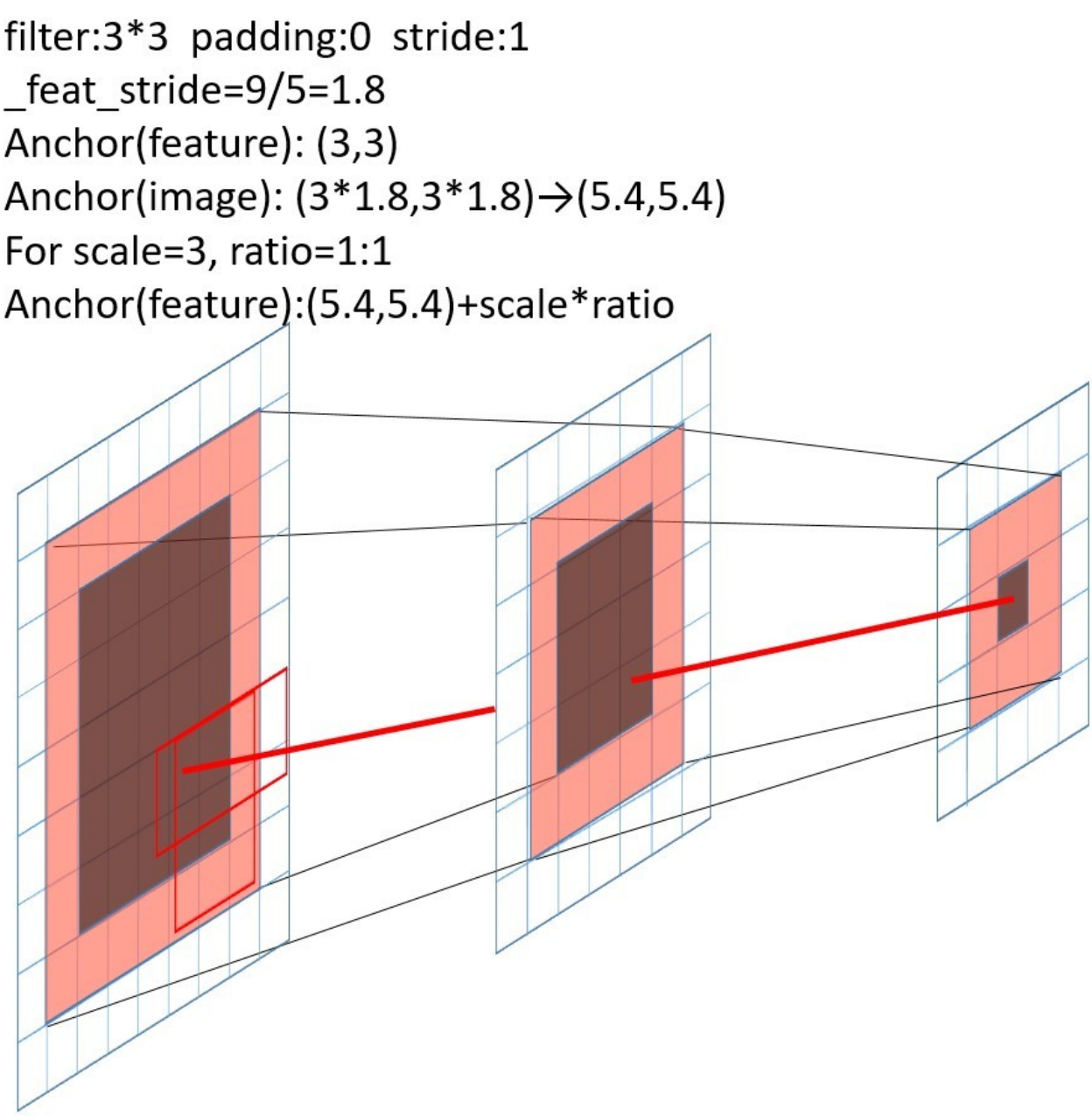


OCR之前Faster RCNN

右左瓜子
风里雨里一件冲锋衣

OCR应该算是深度学习领域的交叉学科，同时涉及图像处理和自然语言处理。先做图像识别进行粗分类，然后结合自然语言处理，把语言学上的特征纳入到分类考虑当中进一步提高准确率。目前为止比较前沿的是一种端到端的同时做文字检测和文字识别的网络，论文在这里：Towards End-to-end Text Spotting with Convolutional Recurrent Neural Networks 里边涉及到文字检测的部分采用了一种"TPN"的方法，是对经典的目标检测方法"RPN"做了一点变化。因为之前并没有处理过图像相关的内容，所以本节先介绍一下Faster RCNN("RPN"是他的一部分)

论文地址：Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks

因为我们的主要目的是做文
回顾了(实际上从它的名字可

Faster RCNN作为限定词是因为现在有些目标检测算法已经可以直接把物体轮廓描出来，而不是矩形框。

知道了它要做什么，现在来看看它是怎么做到的。Faster RCNN的主体分为两部分，一部分是深度全卷积网络来给出建议区域（proposes regions）（RPN），另一部分是使用Fast RCNN来探测上一步给出的建议区域是否是包含目标的。其中在给出建议区域的时候采用了注意力机制。通过RPN（Region Propersal Network）来告诉网络注意力的位置。注意，RPN出来了，这应该是整篇论文最核心的部分，也是最绕的部分，下面会重点介绍一下RPN。

- **Region Propersal Network**

> RPN是把一张图像作为输入，输出一系列可能包含有目标的矩形框，同时每个矩形框对应一个目标分数（衡量存在目标的可能性）。在这个模型中使用了全链接卷积，因为我们想和Fast RCNN网络共享参数，所以想让RPN和Fast RCNN共享同一个卷积层。在具体实验中使用了ZF模型和VGG-16模型来产生这个卷积层。

以上是我把第二篇论文的相关内容概括的翻译了一下。如果之前只做自然语言处理，那估计上面那段话会看的云里雾里。不过不要紧，抛开那些术语所包含的模型的细则，你只要知道整个流程是这样的，输入一张图，通过一个卷积模型得到一个卷积后的特征图（feature map），也就是上面说的卷积层。然后让RPN和Fast RCNN(注意这里是Fast不是Faster)在这个特征图上做计算。就好了，抛开VGG-16不讲（因为这又是一篇论文，但是实现方式比较简单，我们暂且认为它是一种可以很好的提取图像特征的卷积网络），接下来继续谈RPN，Fast RCNN后面会谈到不用着急。

前面我们提到RPN要给出建议区域，那建议区域怎么给出呢？是通过在特征图（在本文中特征图特指经过VGG-16或者ZF卷积产生的特征图）滑动一个nn的窗口，每个窗口映射成一个低维的向量（ZF对应256-d，VGG对应512-d）。好了，这里有疑问了，如何映射，因为文中没有细说，网上有人说"看作一个n*n的卷积"，但是"看作"两个字让人很不信服，后来在源码里面找到这么一段：

```
def _region_proposal(self, net_conv, is_training, initializer):
    rpn = slim.conv2d(net_conv, cfg.RPN_CHANNELS, [3, 3], trainable=is_training
                      scope="rpn_conv/3x3")
```

其中3*3是论文中取得窗口大小，通道数预先在配置文件中配置完毕。所以这一点有疑问的地方解决了，可以放心大胆的说这就是一个卷积（事实上在文章的后面确实提到了这就是一个卷积）。这里参考的是Faster RCNN的tensorflow实现版，地址：

https://github.com/endernewton/tf-faster-rcnn
github.com

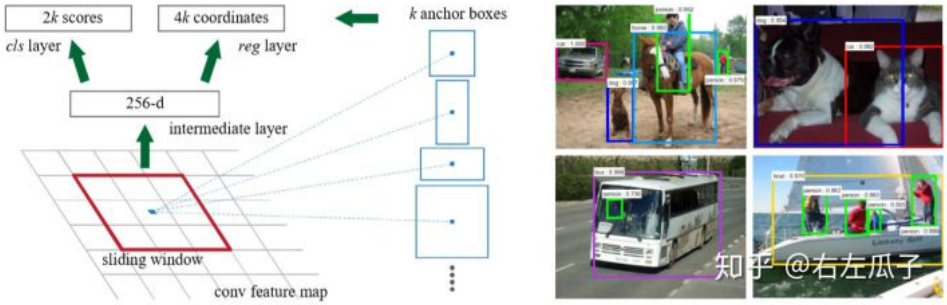
我在看这个的代码的时候当时我就想跪了，这么复杂的网络又梳理的这么好，而代码的作者又是论文的作者，第一次深切体会到什么叫高山仰止。代码做简单的配置就可以跑起来，简直是业界良心，好了，废话不多说了，继续谈RPN。

得到一维向量之后，把这个一维向量送给两个全链接层，一个是box-regression层，一个是box-classification层，都是1*1的全链接卷积层。整个RPN网络基本上就是这样，虽然我们做了一通计算，一顿操作猛如虎，可是好像什么都没得到，因为Anchor还没出来，RPN是Faster RCNN的核心，而Anchor是RPN的核心。

- **Anchors**

在每个滑动窗口中会同时预测k个可能的区域，所以box-regression会有4k个输出，而box-classification会有2k个输出。在这里我解释一下，如果没做过图像处理（像我这样的）可能对2k和4k很疑惑，为什么，后面会有详细的说明，在这里你只要知道box-regression衡量了目标位置的坐标，而box-classification衡量了目标存在的可能性。坐标需要4个值来确定，可能性2个值。anchor就位于滑动窗口的中心，注意，窗口是在特征图上滑动的，并不是在输入图像上滑动的，因为特征图是经过多层卷积得到的，VGG得到的特征图，一个3

于目标检测最大的区别就在于矩形框（anchor box）的选择，目标识别一般使用的1：1，1：2，2：1的矩形框，而文字识别用的是1：1，1：3甚至1：18这样的长方形矩形框。论文给出了一张图，可以很形象的展示anchor是什么。



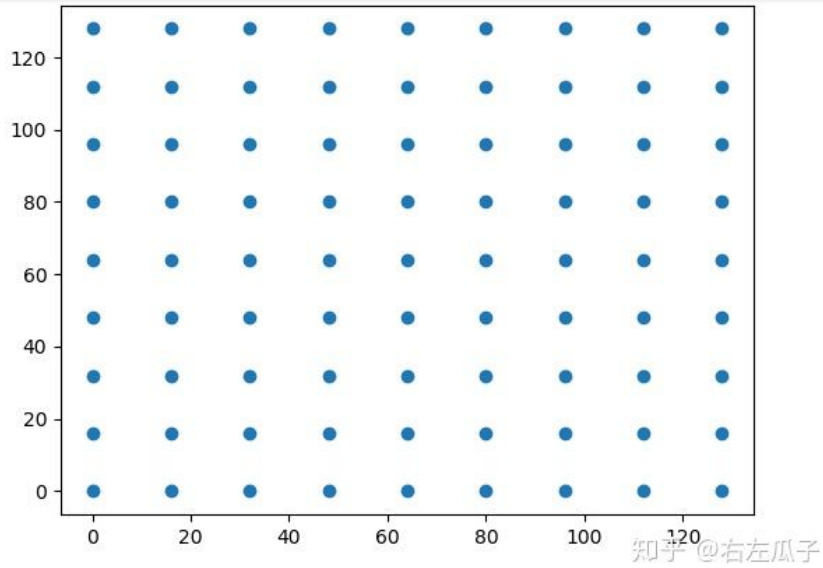
到这里，论文介绍Anchor的内容就没有，不知道你们懵不懵逼，反正我看完是懵逼的，说的是anchor和anchor box一一对应但是是怎么对应的？网上很多解释也是猜测居多，没有给出实锤，为了了解作者本意还是去代码中找答案。

```
def generate_anchors_pre(height, width, feat_stride, anchor_scales=(8,16,32), a
  """ A wrapper function to generate anchors given different scales
    Also return the number of anchors in variable 'length'
  """
  """生成anchor的预处理方法，generate_anchors方法就是直接产生各种大小的anchor box，gen
     是把每一个anchor box对应到原图上
      height = tf.to_int32(tf.ceil(self._im_info[0] / np.float32(self._feat_st
      width = tf.to_int32(tf.ceil(self._im_info[1] / np.float32(self._feat_str
      feat_stride: 经过VGG或者ZF后特征图相对于原图的在长或者宽上的缩放倍数，也就是说heigh
      anchor_scales：anchor尺寸
      anchor_ratios: anchor长宽比
  """
  anchors = generate_anchors(ratios=np.array(anchor_ratios), scales=np.array(ar
  A = anchors.shape[0] # anchor的种数
  shift_x = np.arange(0, width) * feat_stride # 特征图相对于原图的偏移
  shift_y = np.arange(0, height) * feat_stride # 特征图相对于原图的偏移
  shift_x, shift_y = np.meshgrid(shift_x, shift_y) # 返回坐标矩阵
  shifts = np.vstack((shift_x.ravel(), shift_y.ravel(), shift_x.ravel(), shift_
  K = shifts.shape[0]
  # width changes faster, so here it is H, W, C
  anchors = anchors.reshape((1, A, 4)) + shifts.reshape((1, K, 4)).transpose((1
  anchors = anchors.reshape((K * A, 4)).astype(np.float32, copy=False)
  length = np.int32(anchors.shape[0])
  return anchors, length
```

以上就是tensorflow实现版的anchor产生的部分代码，关键的地方我已经加了注释，对于shift_x和shift_y我们可以用代码段测试一下，更为直观。

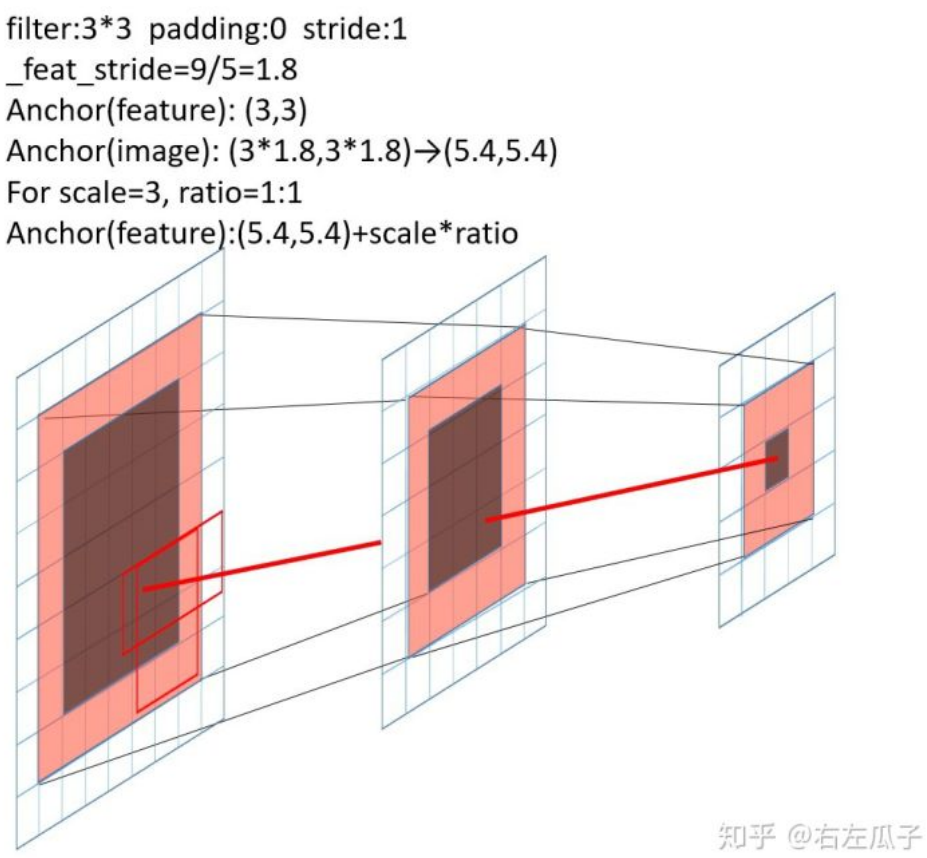
```
>>> shift_x = np.arange(0, 9) * 16 # 假设特征图为9*9， VGG的feat_stride为16
>>> shift_y = np.arange(0, 9) * 16
>>> shift_x, shift_y = np.meshgrid(shift_x, shift_y)
>>> fig = plt.figure()
>>> plt.scatter(shift_x.ravel(), shift_y.ravel())
>>> fig.show()
```

结果如下



特征图每个点对应于原图一个点，在原图对应的点上放anchor box

因为VGG的网络层数太多，所以下面用一个简单的卷积来展示一下他的计算方式，如果想要了解同样的方法一层一层推下去就好了



作图不易，使用注明出处

上面两张图基本上把anchor讲的很明白了，作者在代码中还把超出图片的anchor box删除了，有兴趣的话可以自己看源码。好了整个RPN除了损失函数外其他部分已经完全打通了，下面就讲讲损失函数（原文中还有对anchor的平移不变性以及高效性的论述在此不展开）。

- **Loss Function**

前面我们谈到RPN网络的reg层和cls层分别有4k和2k个输出，在探讨损失函数之前先来说明一下这两个层的输出到底代表了什
签，表明是否有目标。这个

$$IoU = \frac{TP}{FP+TP+FN}$$

这个公式对于做过自然语言处理的同学来说应该很熟悉了，两个区域重叠区域大小比上覆盖区域的大小就是了。有了这个衡量标准，我们通过两个条件获取正例：

1. 一个anchor对应的k个anchor box中IoU值最大的
2. IoU值大于0.7的

通常条件2就能获取大多数正例，条件一作为某些特殊情况的补充。反例的获取只有一个条件：IoU值小于0.3。其他的结果丢弃。通过这种方式得到cls层的标签，衡量了anchor box存在目标的可能性。对于reg层，我们给出这样的标签，训练时：

$$t_x^* = (x^* - x_a)/w_a \quad , t_y^* = (y^* - y_a)/h_a$$

$$t_w^* = \log(w^*/w_a) \quad , t_h^* = \log(h^*/h_a)$$

x, y, w, h 分别代表横纵坐标和宽高，x^* 用来代表ground-truth的值，x_a 代表anchor给的值的。预测时：

$$t_x = (x - x_a)/w_a \quad , t_y = (y - y_a)/h_a$$

$$t_w = \log(w/w_a) \quad , t_h = \log(h/h_a)$$

x 代表预测值。这样，训练时我们可以通过最小化 t 来训练网络，预测是可以通过 t 获取预测框的实际位置。

最终损失函数为：

$$L(\{p_i\},\{t_i\}) = \frac{1}{N_{cls}}\sum_i L_{cls}(p_i, p_i^*) + \lambda\frac{1}{N_{reg}}\sum_i p^* L_{reg}(t_i, t_i^*)$$

p_i 为anchor box是一个目标的概率，p_i^* 通过前面的IoU算法获取正反例后正例为1反例为0。$L_{reg}(t_i, t_i^*) = R(t_i, t_i^*)$，$R$ 是Fast RCNN中提出来的损失函数 $smooth\ L_1$.

$$smooth_{L_1}(x) = \begin{cases} 0.5x^2, & if: |t| \le 1 \\ |x| - 0.5 & otherwise, \end{cases}$$

最后：$t_i = \sum_{j\in\{x,y,w,h\}} t_j$

通过训练RPN提出建议区域，一般然后再用Fast RCNN在同样的特征图上对建议区域做目标探测获得最终结果，文中提到了三种RPN和Fast RCNN的训练方式。我们下一篇谈完Fast RCNN之后继续谈。（主要是这一篇有点长了，我自己画图找源码前前后后也写了三天，已经明显感觉到有些公式已经懒得解释了，继续怼下去怕影响质量，所以Fast RCNN单独开一篇）

编辑于 2018-09-19

OCR（光学字符识别） 深度学习（Deep Learning） 目标检测

文章被以下专栏收录

深度学习，谈点儿实在的
深度学习填坑历程