

Fast(er) rcnn的损失函数总结

参考文献:

1. Fast(er) rcnn的损失函数总结: https://blog.csdn.net/qq_17846375/article/details/100687504
2. 【Faster RCNN】损失函数理解: https://blog.csdn.net/Mr_health/article/details/84970776
3. Mask-RCNN中的损失函数: <https://blog.csdn.net/lanyuxuan100/article/details/71126778>

Fast RCNN 将分类与回归做到了一个网络里面，因此损失函数必定是多任务的：

$$L(p, u, t^u, v) = L_{\text{cls}}(p, u) + \lambda[u \geq 1]L_{\text{loc}}(t^u, v)$$

分类任务部分

分类任务还是我们常用的**对数损失**。即**对数似然损失**(Log-likelihood Loss)，或**交叉熵损失**(cross-entropy Loss)，是在概率估计上定义的。

对数损失通过惩罚错误的分类，实现对分类器的准确度(Accuracy)的量化。

最小化对数损失基本等价于最大化分类器的准确度。为了计算对数损失，分类器必须提供对输入的所属的每个类别的概率值，不只是最可能的类别。对数损失函数的计算公式如下：

$$L(Y, P(Y|X)) = -\log P(Y|X) = -\frac{1}{N} \sum_{i=1}^N \sum_{j=1}^M y_{ij} \log(p_{ij})$$

其中，Y 为输出变量，X 为输入变量，L 为损失函数。N 为输入样本量，M 为可能的类别数， y_{ij} 是一个二值指标，表示类别 j 是否是输入实例 x_i 的真实类别。 p_{ij} 为模型或分类器预测输入实例 x_i 属于类别 j 的概率。

如果只有两类 {0, 1}，则对数损失函数的公式简化为：

$$-\frac{1}{N} \sum_{i=1}^N (y_i \log p_i + (1-y_i) \log (1-p_i))$$

这时， y_i 为输入实例 x_i 的真实类别， p_i 为预测输入实例 x_i 属于类别 1 的概率。对所有样本的对数损失表示对每个样本的对数损失的平均值，对于完美的分类器，对数损失为 0。

意义：

对数损失是用于最大似然估计的（以最大概率为标准来判断结果，即叫做极大似然估计）。一组参数在一堆数据下的似然值，等于每一条数据在这组参数下的条件概率之积。而损失函数一般是每条数据的损失之和，为了把积变为和，就取了对数。再加个负号是为了让**最大似然值**和**最小损失**对应起来。

用来判断实际的输出与期望输出的接近程度，刻画除去实际输出概率与期望输出概率的距离，即**交叉熵越小，两个概率分布越接近。**

总结：

回归任务部分

t^u 的定义方式与 RCNN 中一致，为中心区域坐标，以及区域宽度及高度。但是使用的损失函数不同，Fast RCNN 使用的损失函数为鲁棒性更佳的 L1 损失函数，而不是 RCNN 中使用的 L2 损失函数，而且训练的过程也简单很多，需要注意的每个区域候选对于每个类都有区域回归训练。

$$L_{\text{loc}}(t^u, v) = \sum_{i \in \{x, y, w, h\}} \text{smooth}_{L_1}(t_i^u - v_i), \quad (2)$$

in which

$$\text{smooth}_{L_1}(x) = \begin{cases} 0.5x^2 & \text{if } |x| < 1 \\ |x| - 0.5 & \text{otherwise,} \end{cases} \quad (3)$$

L1 范数是最小绝对值偏差，是鲁棒的，是因为它能处理数据中的异常值。如果需要考虑任一或全部的异常值，那么最小绝对值偏差是更好的选择。

L2 范数将误差平方化（如果误差大于1，则误差会放大很多），模型的误差会比 L1 范数来得大，因此模型会对这个样本更加敏感，这就需要调整模型来最小化误差。如果这个样本是一个异常值，模型就需要调整以适应单个的异常值，这会牺牲许多其它正常的样本，因为这些正常样本的误差比这单个的异常值的误差小。

L2是平方差，L1是绝对差，如果有异常点，前者相当于放大了这种误差，而绝对差没有放大。

<https://www.cnblogs.com/pacino12134/p/11104446.html>

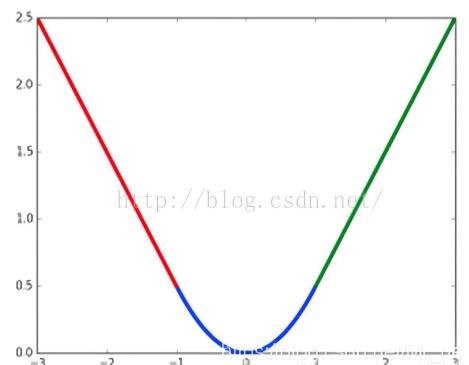
使用Smooth L1 Loss的原因 (摘抄自：https://blog.csdn.net/Mr_health/article/details/84970776)

对于边框的预测是一个回归问题。通常可以选择平方损失函数（L2损失） $f(x)=x^2$ 。但这个损失对于比较大的误差的惩罚很高。

我们可以采用稍微缓和一点绝对损失函数（L1损失） $f(x)=|x|$ ，它是随着误差线性增长，而不是平方增长。但这个函数在0点处导数不存在，因此可能会影响收敛。

一个通常的解决办法是，分段函数，在0点附近使用平方函数使得它更加平滑。它被称之为平滑L1损失函数。它通过一个参数 σ 来控制平滑的区域。一般情况下 $\sigma = 1$ ，在 faster rcnn 函数中 $\sigma = 3$ 。

$$\text{smooth}_{L_1}(x) = \begin{cases} 0.5x^2 \times 1/\sigma^2 & \text{if } |x| < 1/\sigma^2 \\ |x| - 0.5 & \text{otherwise,} \end{cases}$$



总结：

Faster RCNN

遵循 multi-task loss 定义，最小化目标函数，Faster R-CNN中对一个图像的函数定义为：

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

p_i 为 anchor 预测为目标的概率；

GT标签: $p_i^* = \begin{cases} 0 & \text{negative label} \\ 1 & \text{positive label} \end{cases}$;

$t_i = \{t_x, t_y, t_w, t_h\}$ 是一个向量，表示预测的 bounding box 包围盒的 4 个参数化坐标；

t_i^* 是与 positive anchor 对应的 ground truth 包围盒的坐标向量；

RPN网络的产生的anchor只分为前景和背景，
前景的标签为1，背景的标签为0。

$L_{cls}(p_i, p_i^*)$ 是两个类别（目标 vs. 非目标）的对数损失：

$$L_{cls}(p_i, p_i^*) = -\log [p_i^* p_i + (1 - p_i^*)(1 - p_i)]$$

$L_{reg}(t_i, t_i^*)$ 是回归损失，用 $L_{reg}(t_i, t_i^*) = R(t_i - t_i^*)$ 来计算，R 是 smooth L1 函数。

$p_i^* L_{reg}$ 这一项意味着只有前景 anchor ($p_i^* = 1$) 才有回归损失，其他情况就没有 ($p_i^* = 0$)。cls 层和 reg 层的输出分别由 $\{p_i\}$ 和 $\{u_i\}$ 组成，这两项分别由 N_{cls} 和 N_{reg} 以及一个平衡权重 λ 归一化（早期实现及公开的代码中， $\lambda = 10$ ，cls 项的归一化值为 mini-batch 的大小，即 $N_{cls} = 256$ ，reg 项的归一化值为 anchor 位置的数量，即 $N_{reg} \sim 2400 (40 * 60)$ ，这样 cls 和 reg 项差不多是等权重的。）

一些感悟

论文中把 N_{cls} , N_{reg} 和 λ 都看做是平衡分类损失和回归损失的归一化权重，但是我在看 tensorflow 代码实现 faster rcnn 的损失时发现（这里以 fast rcnn 部分的分类损失和 box 回归损失为例，如下），可以看到在计算分类损失时，并没有输入 N_{cls} 这个参数，只是在计算 box 回归损失的时候输入了 outside_weights 这个参数。这时候我才意识到分类损失是交叉熵函数，求和后会除以总数量，除以 N_{cls} 已经包含到交叉熵函数本身。

为了平衡两种损失的权重，outside_weights 的取值取决于 N_{cls} ，而 N_{cls} 的取值取决于 batch_size。因此才会有

$$\text{outside_weights} = \frac{\lambda}{N_{reg}} = \frac{1}{batch\ size} = \begin{cases} \frac{1}{256}, & \text{RPN 训练阶段} \\ \frac{1}{128}, & \text{fast rcnn 阶段} \end{cases}$$

截自：https://blog.csdn.net/Mr_health/article/details/84970776

总结：

Faster RCNN

参考文献:

1. 一文读懂Faster RCNN: <https://zhuanlan.zhihu.com/p/31426458>
2. 挣一挣pytorch官方FasterRCNN代码: <https://zhuanlan.zhihu.com/p/145842317>

经过 R-CNN 和 Fast RCNN 的积淀，Ross B. Girshick 在2016年提出了新的 Faster RCNN，在结构上，Faster RCNN 已经将 **特征抽取(feature extraction)**, **proposal提取**, **bounding box regression(rect refine)**, **classification** 都整合在了一个网络中，使得综合性能有较大提高，在检测速度方面尤为明显。

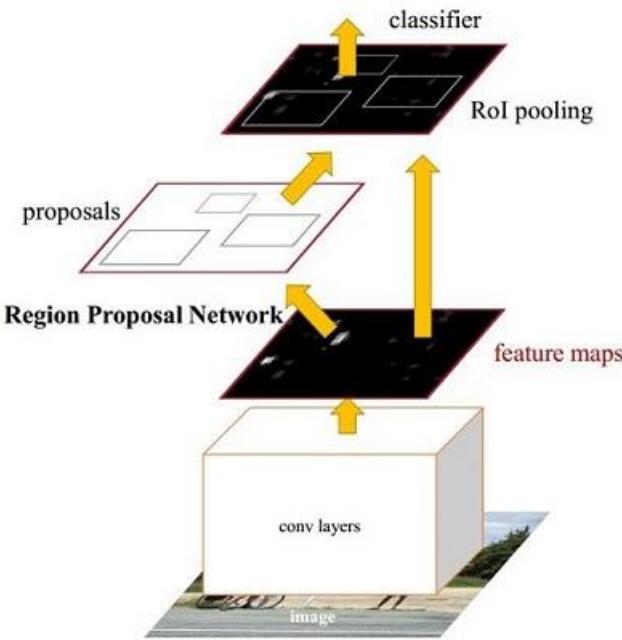


图1 Faster RCNN基本结构（来自原论文）

Faster RCNN其实可以分为4个主要内容：

- Conv layers。作为一种CNN网络目标检测方法，Faster RCNN首先使用一组基础的 conv+relu+pooling层 提取 image 的 feature maps。该feature maps被共享用于后续 RPN层 和 全连接层。
- Region Proposal Networks。RPN 网络用于生成 region proposals。该层通过 softmax 判断 anchors 属于 positive 或者 negative，再利用 bounding box regression 修正 anchors 获得精确的 proposals。
- Roi Pooling。该层收集输入的 feature maps 和 proposals，综合这些信息后提取 proposal feature maps，送入后续全连接层判定目标类别。
- Classification。利用 proposal feature maps 计算 proposal 的类别，同时再次 bounding box regression 获得检测框最终的精确位置。

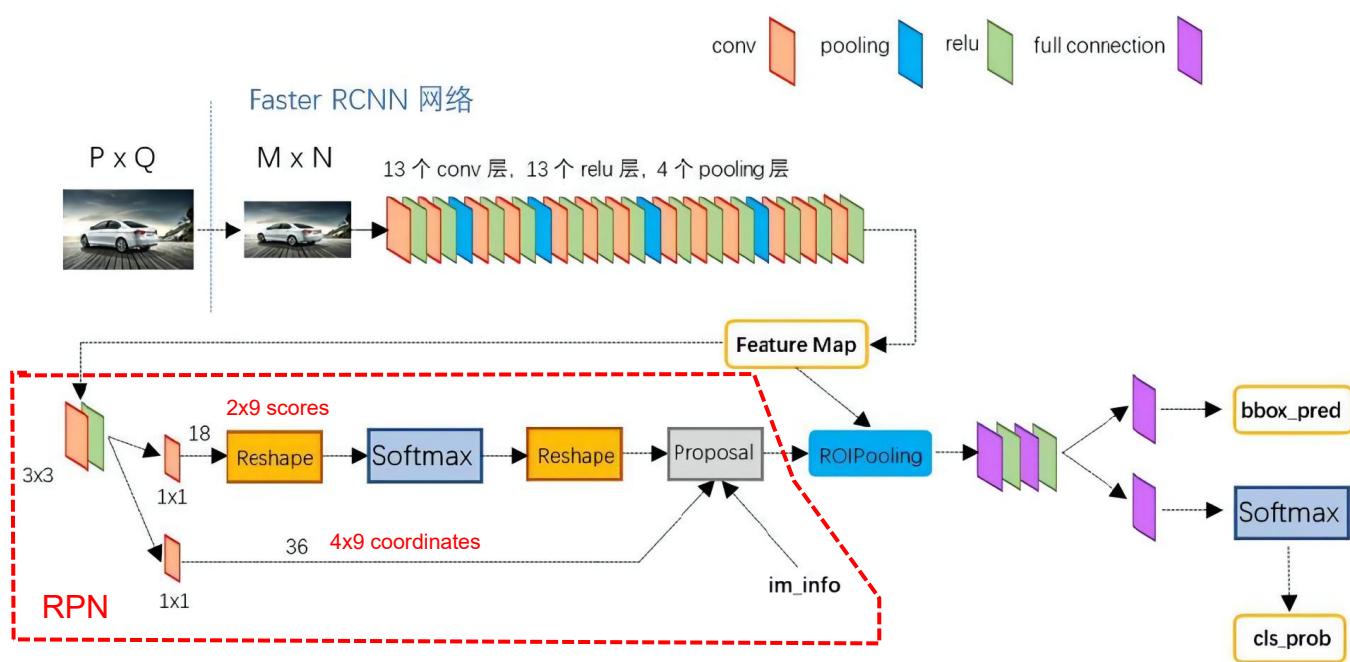


图2 faster_rcnn_test.pt 网络结构 (pascal_voc/VGG16/faster_rcnn_alt_opt/faster_rcnn_test.pt)

上图展示了python版本中的 VGG16 模型中的 faster_rcnn_test.pt 的网络结构，可以清晰的看到该网络对于一副任意大小 PxQ 的图像：

- 首先缩放至固定大小 $M \times N$ ，然后将 $M \times N$ 图像送入网络；
- 而 Conv layers 中包含了 13个conv层 + 13个relu层 + 4个pooling层；
- RPN网络首先经过 3×3 卷积，再分别生成 positive anchors 和对应 bounding box regression偏移量，然后计算出 proposals；
- 而 Roi Pooling 层则利用 proposals 从 feature maps 中提取 proposal feature 送入后续 全连接和softmax网络 作 classification (即分类proposal到底是什么object) 。

新出炉的pytorch官方Faster RCNN代码导读：<https://zhuanlan.zhihu.com/p/145842317>

1 Conv layers

这里有一个非常容易被忽略但是又无比重要的信息，在 Conv layers 中：

- 所有的 conv 层都是： kernel_size=3, pad=1, stride=1
- 所有的 pooling 层都是： kernel_size=2, pad=0, stride=2

为何重要？ 在 Faster RCNN 的 Conv layers 中对所有的卷积都做了扩边处理 (pad=1, 即填充一圈0)，导致原图变为 $(M+2) \times (N+2)$ 大小，再做 3×3 卷积 后输出 $M \times N$ 。正是这种设置，使得 Conv layers 中的 conv 层 不改变输入和输出矩阵大小。如下图：

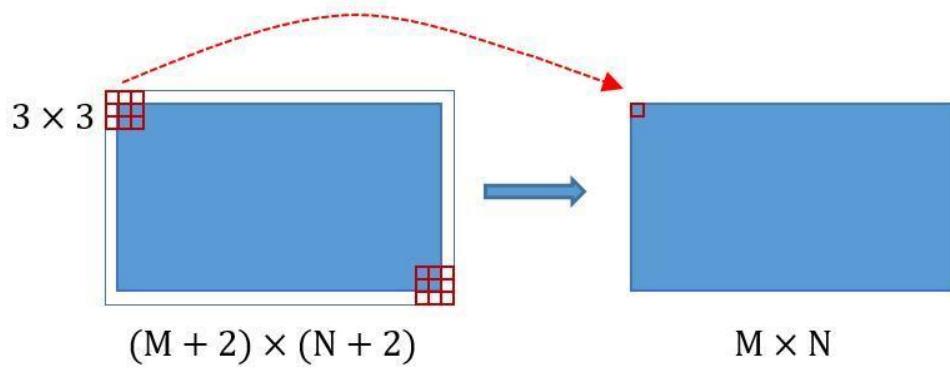


图3 卷积示意图

类似的是，Conv layers 中的 pooling 层 kernel_size=2, stride=2。这样每个经过 pooling 层的 $M \times N$ 矩阵，都会变为 $(M/2) \times (N/2)$ 大小。

综上所述，在整个 Conv layers 中，conv 和 relu 层 不改变输入输出大小，只有 pooling 层 使输出长宽都变为输入的 $1/2$ 。

那么，一个 $M \times N$ 大小的矩阵经过 Conv layers 固定变为 $(M/16) \times (N/16)$ ！这样 Conv layers 生成的 feature map 中都可以和原图对应起来。

2 Region Proposal Networks (RPN)

经典的检测方法生成检测框都非常耗时，如 OpenCV adaboost 使用滑动窗口+图像金字塔生成检测框；或如 R-CNN 使用 SS(Selective Search)方法生成检测框。而 Faster RCNN 则抛弃了传统的滑动窗口和 SS 方法，直接使用 RPN 生成检测框，这也是 Faster R-CNN 的巨大优势，能极大提升检测框的生成速度。

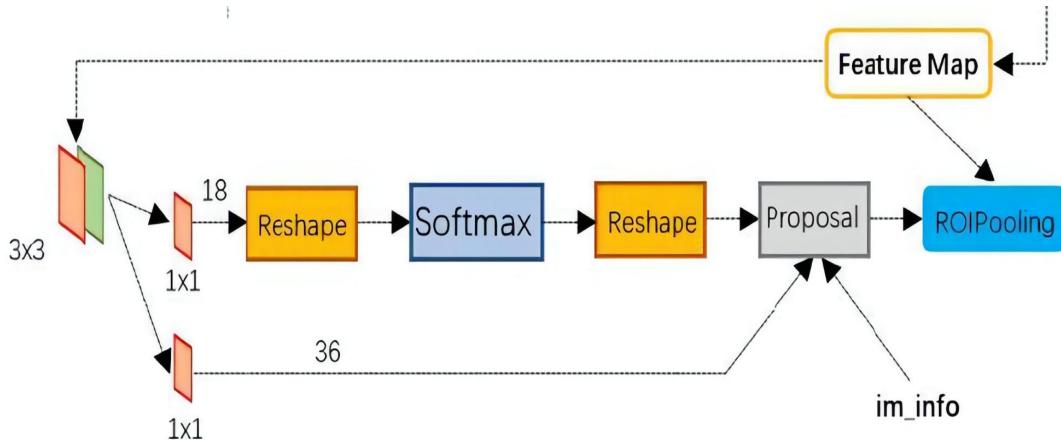


图4 RPN网络结构

上图展示了 RPN 网络的具体结构。可以看到 RPN 网络 实际分为2条线：

- 上面一条通过 softmax 分类 anchors 获得 positive 和 negative 分类；
- 下面一条用于计算对于 anchors 的 bounding box regression 偏移量，以获得精确的 proposal。

而最后的 Proposal 层则负责综合 positive anchors 和对应 bounding box regression 偏移量 获取 proposals，同时剔除太小和超出边界的 proposals。其实整个网络到了 Proposal Layer 这里，就完成了相当于目标定位的功能。

2.1 多通道图像卷积基础知识介绍

在介绍RPN前，还要多解释一些基础知识：

- 对于 单通道图像+单卷积核 做卷积，图3已经展示了；
- 对于 多通道图像+多卷积核 做卷积，计算方式如下：

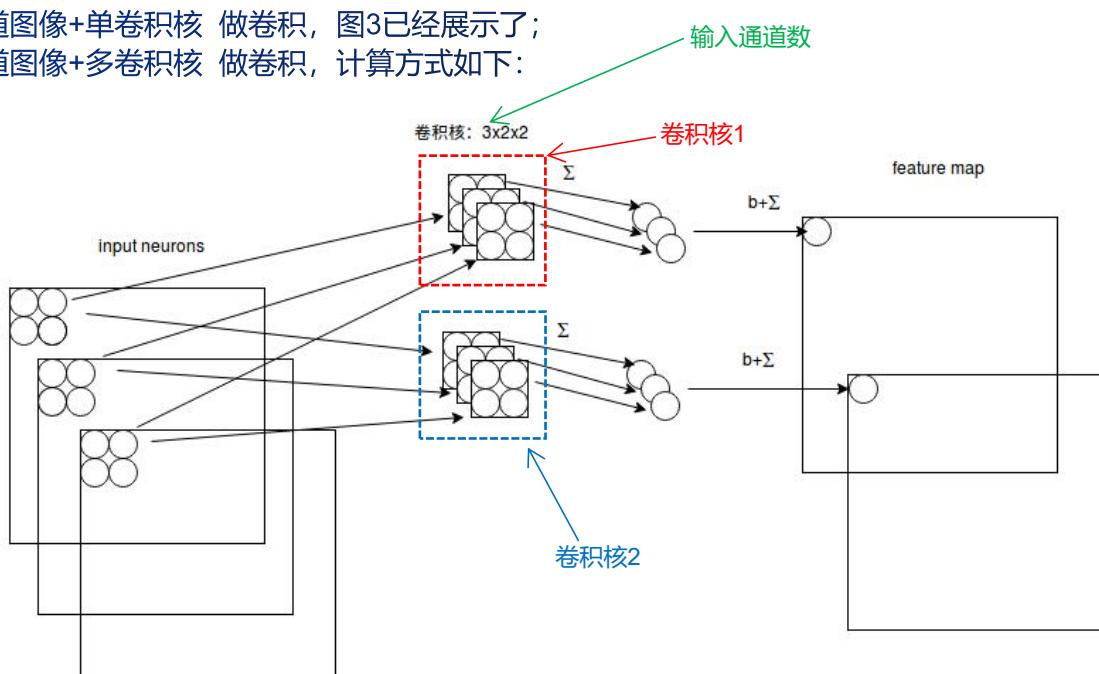


图5 多通道+多卷积核计算

如图5，输入有3个通道，同时有2个卷积核。

对于每个卷积核，先在输入3个通道分别作卷积，再将3个通道结果加起来得到卷积输出。

所以对于某个卷积层，无论输入图像有多少个通道，**输出图像通道数总是等于卷积核数量！**

对多通道图像做 1×1 卷积，其实就是将输入图像于每个通道乘以卷积系数后加在一起，即相当于把原图像中本来各个独立的通道“联通”在了一起。

2.2 anchors

提到 RPN 网络，就不能不说 anchors。所谓 anchors，实际上就是一组由 `rpn/generate_anchors.py` 生成的矩形。直接运行作者 demo 中的 `generate_anchors.py` 可以得到以下输出：

```
[[ -84. -40. 99. 55.]
 [-176. -88. 191. 103.]
 [-360. -184. 375. 199.]
 [-56. -56. 71. 71.]
 [-120. -120. 135. 135.]
 [-248. -248. 263. 263.]
 [-36. -80. 51. 95.]
 [-80. -168. 95. 183.]
 [-168. -344. 183. 359.]]
```

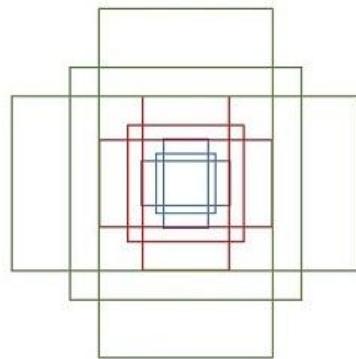


图6 anchors 示意图

其中每行的4个值 (x_1, y_1, x_2, y_2) 表矩形左上和右下角点坐标。9个矩形共有3种形状，长宽比为大约为 $width : height \in \{1:1, 1:2, 2:1\}$ 三种，如图6。

实际上通过 anchors 就引入了检测中常用到的**多尺度方法**。

注：关于上面的 anchors size，其实是根据检测图像设置的。在 python demo 中，会把任意大小的输入图像 reshape 成 800x600（即图2中的 $M=800, N=600$ ）。再回头来看 anchors 的大小，anchors 中长宽 1:2 中最大为 352×704 ，长宽 2:1 中最大 736×384 ，基本是 cover 了 800×600 的各个尺度和形状。

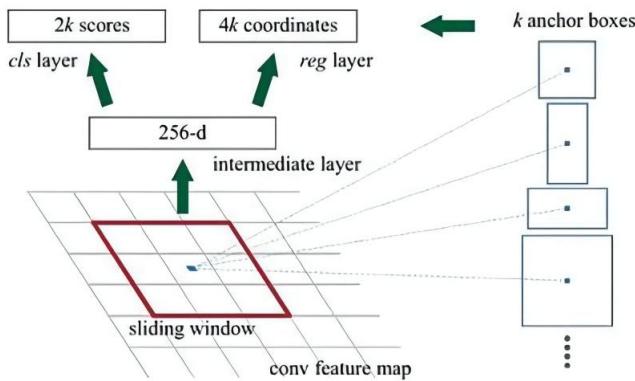


图7

那么这 9 个 anchors 是做什么的呢？

借用 Faster RCNN 论文中的原图，如图7，遍历 Conv layers 计算获得的 feature maps，为每一个点都配备这 9 种 anchors 作为初始的检测框。这样做获得检测框很不准确，不用担心，后面还有 2 次 bounding box regression 可以修正检测框位置。

解释一下左图中的数字：

- 在原文中使用的是 ZF model 中，其 Conv Layers 中最后的 conv5 层 `num_output=256`，对应生成 256 张特征图，所以相当于 feature map 每个点都是 256-dimensions。

- 在 conv5 之后，做了 `rpn_conv/3x3` 卷积且 `num_output=256`，相当于每个点又融合了周围 3×3 的空间信息，同时 256-d 不变（如图4和图7中的红框）
- 假设在 conv5 feature map 中每个点上有 k 个 anchor (默认 $k=9$)，而每个 anchor 要分 positive 和 negative，故每个点由 256d feature 转化为 $cls=2 \cdot k$ scores；而每个 anchor 都有 (x, y, w, h) 对应 4 个偏移量，故 $reg=4 \cdot k$ coordinates。

总结：

- 补充一点，全部 anchors 拿去训练太多了，训练程序会在合适的 anchors 中随机选取 128 个 positive anchors + 128 个 negative anchors 进行训练（什么是合适的 anchors 下文5.1有解释）。

注意，在本文讲解中使用的 VGG conv5 num_output=512，所以是512d，其它类似。

其实 RPN 最终就是在原图尺度上，设置了密密麻麻的候选 anchor。然后用 cnn 去判断哪些 anchor 是里面有目标的 positive anchor，哪些是没目标的 negative anchor。所以，仅仅是个二分类而已！

那么 anchor 一共有多少个？原图 800x600，VGG下采样16倍，feature map每个点设置 9 个 anchor，所以：

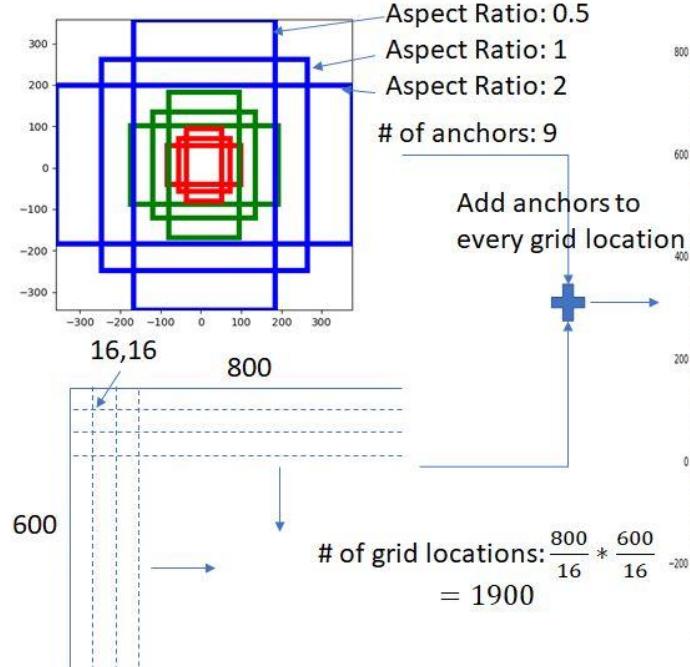
$$\text{ceil}(800/16) \times \text{ceil}(600/16) \times 9 = 50 \times 38 \times 9 = 17100 \quad (1)$$

其中 ceil() 表示向上取整，是因为VGG输出的 feature map size= 50*38。

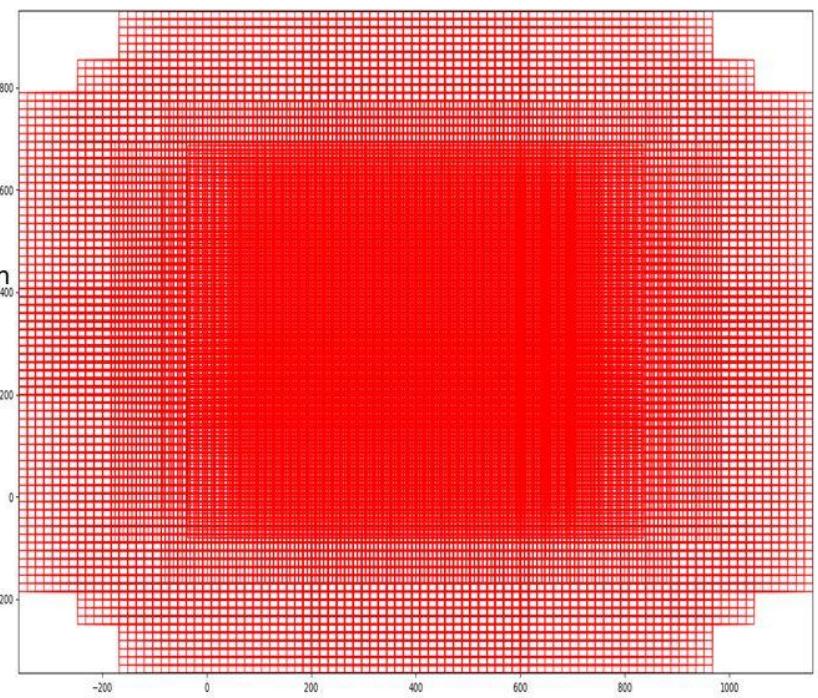
Generate Anchors

Given:

- Set of aspect ratios (0.5, 1, 2)
- Stride length (downscaling performed by resnet head: 16)
- Anchor Scales (8, 16, 32)



Total number of anchors: $1900 * 9 = 17100$
Some boxes lie outside the image boundary



Create uniformly spaced grid with
spacing = stride length

图8 Gernerate Anchors

2.3 softmax 判定 positive 与 negative

一副 $M \times N$ 大小的矩阵送入 Faster RCNN 网络后，到 RPN 网络变为 $(M/16) \times (N/16)$ ，不妨设 $W=M/16$, $H=N/16$ 。在进入 reshape 与 softmax 之前，先做了 1×1 卷积，如图9：



图9 RPN中判定 positive/negative 网络结构

该 1×1 卷积的 caffe prototxt 定义如右：

可以看到其 num_output=18，也就是经过该卷积的输出图像为 $W \times H \times 18$ 大小（注意第二章开头提到的卷积计算方式）。这也就刚好对应了 feature maps 每一个点都有 9 个 anchors，同时每个 anchors 又有可能是 positive 和 negative，所有这些信息都保存 $W \times H \times (9 \times 2)$ 大小的矩阵。

为何这样做？

```
layer {
    name: "rpn_cls_score"
    type: "Convolution"
    bottom: "rpn/output"
    top: "rpn_cls_score"
    convolution_param {
        num_output: 18    # 2(positive/negative) * 9(anchors)
        kernel_size: 1 pad: 0 stride: 1
    }
}
```

后面接 softmax 分类获得 positive anchors，也就相当于初步提取了检测目标候选区域 box（一般认为目标在 positive anchors 中）。

那么为何要在 softmax 前后都接一个 reshape layer？

其实只是为了便于 softmax 分类，至于具体原因这就要从 caffe 的实现形式说起了。在 caffe 基本数据结构 blob 中以如下形式保存数据：

```
blob=[batch_size, channel, height, width]
```

对应至上面的保存 positive/negative anchors 的矩阵，其在 caffe blob 中的存储形式为 $[1, 2 \times 9, H, W]$ 。而在 softmax 分类时需要进行 positive/negative 二分类，所以 reshape layer 会将其变为 $[1, 2, 9 \times H, W]$ 大小，即单独“腾空”出来一个维度以便 softmax 分类，之后再 reshape 回复原状。

贴一段 caffe softmax_loss_layer.cpp 的 reshape 函数的解释，非常精辟：

```
"Number of labels must match number of predictions; "
"e.g., if softmax axis == 1 and prediction shape is (N, C, H, W), "
"label count (number of labels) must be N*H*W, "
"with integer values in {0, 1, ..., C-1}.";
```

综上所述，RPN 网络中利用 anchors 和 softmax 初步提取出 positive anchors 作为候选区域（另外也有实现用 sigmoid 替代 softmax，输出 $[1, 1, 9 \times H, W]$ 后接 sigmoid 进行 positive/negative 二分类，原理一样）。

2.4 bounding box regression 原理

如图 10 所示绿色框为飞机的 Ground Truth(GT)，红色为提取的 positive anchors，即便红色的框被分类器识别为飞机，但是由于红色的框定位不准，这张图相当于没有正确的检测出飞机。所以我们希望采用一种方法对红色的框进行微调，使得 positive anchors 和 GT 更加接近。

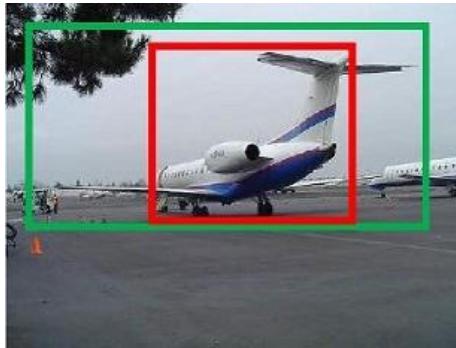


图10

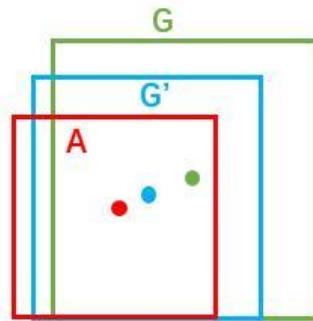


图11

对于窗口一般使用四维向量 (x, y, w, h) 表示，分别表示窗口的中心点坐标和宽高。

对于图 11，红色的框 A 代表原始的 positive Anchors，绿色的框 G 代表目标的 GT，**我们的目标是寻找一种关系，使得输入原始的 anchor A 经过映射得到一个跟真实窗口 G 更接近的回归窗口 G'**，即：

- 给定 anchor $A = (A_x, A_y, A_w, A_h)$ 和 $GT = (G_x, G_y, G_w, G_h)$
- 寻找一种变换 F ，使得： $F(A_x, A_y, A_w, A_h) = (G'_x, G'_y, G'_w, G'_h)$ ，其中 $(G'_x, G'_y, G'_w, G'_h) \approx (G_x, G_y, G_w, G_h)$ 。

那么经过何种变换 F 才能从图 11 中的 anchor A 变为 G' 呢？比较简单的思路就是：

- 先做平移

$$G'_x = A_w \cdot d_x(A) + A_x \quad (2)$$

$$G'_y = A_h \cdot d_y(A) + A_y \quad (3)$$

- 再做缩放

$$G'_w = A_w \cdot \exp(d_w(A)) \quad (4)$$

$$G'_h = A_h \cdot \exp(d_h(A)) \quad (5)$$

观察上面 4 个公式发现，需要学习的是 $d_x(A), d_y(A), d_w(A), d_h(A)$ 这四个变换。当输入的 anchor A 与 GT 相差较小时，可以认为这种变换是一种线性变换，那么就可以用线性回归来建模对窗口进行微调（注意，只有当 anchors A 和 GT 比较接近时，才能使用线性回归模型，否则就是复杂的非线性问题了）。

接下来的问题就是如何通过线性回归获得 $d_x(A), d_y(A), d_w(A), d_h(A)$ 了。线性回归就是给定输入的特征向量 X，学习一组参数 W，使得经过线性回归后的值跟真实值 Y 非常接近，即 $Y = WX$ 。对于该问题，输入 X 是 CNN feature map，定义为 Φ ；同时还有训练传入 A 与 GT 之间的变换量，即 (t_x, t_y, t_w, t_h) 。输出是 $d_x(A), d_y(A), d_w(A), d_h(A)$ 四个变换。那么目标函数可以表示为：

$$d_*(A) = W_*^T \cdot \phi(A) \quad (6)$$

其中 $\phi(A)$ 是对应 anchor 的 feature map 组成的特征向量， W_* 是需要学习的参数， $d_*(A)$ 是得到的预测值 (* 表示 x, y, w, h，也就是每一个变换对应一个上述目标函数）。

总结：

为了让预测值 $d_*(A)$ 与真实值 t_* 差距最小，设计 L1 损失函数：

$$\text{Loss} = \sum_i^N |t_*^i - W_*^T \cdot \phi(A^i)| \quad (7)$$

函数优化目标为：

$$\hat{W}_* = \operatorname{argmin}_{W_*} \sum_i^n |t_*^i - W_*^T \cdot \phi(A^i)| + \lambda ||W_*|| \quad (8)$$

为了方便描述，这里以 L1 损失为例介绍，而真实情况中一般使用 **smooth-L1 损失**。

需要说明，只有在 GT 与需要回归框位置比较接近时，才可近似认为上述线性变换成立。

说完原理，对应于 Faster RCNN 原文，positive anchor 与 ground truth 之间的平移量 (t_x, t_y) 与尺度因子 (t_w, t_h) 如下：

$$t_x = (x - x_a)/w_a \quad t_y = (y - y_a)/h_a \quad (9)$$

$$t_w = \log(w/w_a) \quad t_h = \log(h/h_a) \quad (10)$$

对于 bouding box regression 网络回归分支，输入是 cnn feature Φ ，监督信号是 anchor 与 GT 的差距 (t_x, t_y, t_w, t_h) ，即训练目标是：输入 Φ 的情况下使网络输出与监督信号尽可能接近。那么当 bouding box regression 工作时，再输入 Φ 时，回归网络分支的输出就是每个 anchor 的平移量和变换尺度，显然即可用来修正 anchor 位置了。

2.5 对 proposals 进行 bounding box regression

在了解 bounding box regression 后，再回头来看 RPN 网络第二条线路，如图12。



图12 RPN 中的 bbox reg

先来看一看上图12中 1x1卷积 的 caffe prototxt 定义：

可以看到其 num_output=36，即经过该卷积输出图像为 WxHx36，在 caffe blob 存储为 [1, 4x9, H, W]，这里相当于 feature maps 每个点都有 9 个 anchors，每个 anchors 又都有4个用于回归的 $[d_x(A), d_y(A), d_w(A), d_h(A)]$ 变换量。

在图8，VGG输出 $50*38*512$ 的特征，对应设置 $50*38*k$ 个 anchors，而RPN输出：

- 大小为 $50*38*2k$ 的 positive/negative softmax 分类特征矩阵。
- 大小为 $50*38*4k$ 的 regression 坐标回归特征矩阵。

恰好满足 RPN 完成 positive/negative 分类 + bounding box regression 坐标回归。

```
layer {
    name: "rpn_bbox_pred"
    type: "Convolution"
    bottom: "rpn/output"
    top: "rpn_bbox_pred"
    convolution_param {
        num_output: 36      # 4 * 9(anchors)
        kernel_size: 1 pad: 0 stride: 1
    }
}
```

2.6 Proposal Layer

Proposal Layer 负责综合所有 $[d_x(A), d_y(A), d_w(A), d_h(A)]$ 变换量和 positive anchors，计算出精准的 proposal，送入后续 RoI Pooling Layer。

先来看看 Proposal Layer 的 caffe prototxt 定义：(见右边)

Proposal Layer有3个输入：

- positive vs negative anchors 分类器结果 rpn_cls_prob_reshape;
- bbox reg 的 $[d_x(A), d_y(A), d_w(A), d_h(A)]$ 变换量 rpn_bbox_pred;
- 以及 im_info。

另外还有参数 feat_stride=16，这和图4是对应的。

首先解释 im_info。对于一副任意大小 PxQ 图像，传入 Faster RCNN 前首先 reshape 到固定MxN (见图13)，im_info=[M, N, scale_factor] 则保存了此次缩放的所有信息。然后经过 Conv Layers，经过4次 pooling 变为 $W \times H = (M/16) \times (N/16)$ 大小，其中 feature_stride=16 则保存了该信息，用于计算 anchor 偏移量。

```
layer {
    name: 'proposal'
    type: 'Python'
    bottom: 'rpn_cls_prob_reshape'
    bottom: 'rpn_bbox_pred'
    bottom: 'im_info'
    top: 'rois'
    python_param {
        module: 'rpn.proposal_layer'
        layer: 'ProposalLayer'
        param_str: "'feat_stride': 16"
    }
}
```

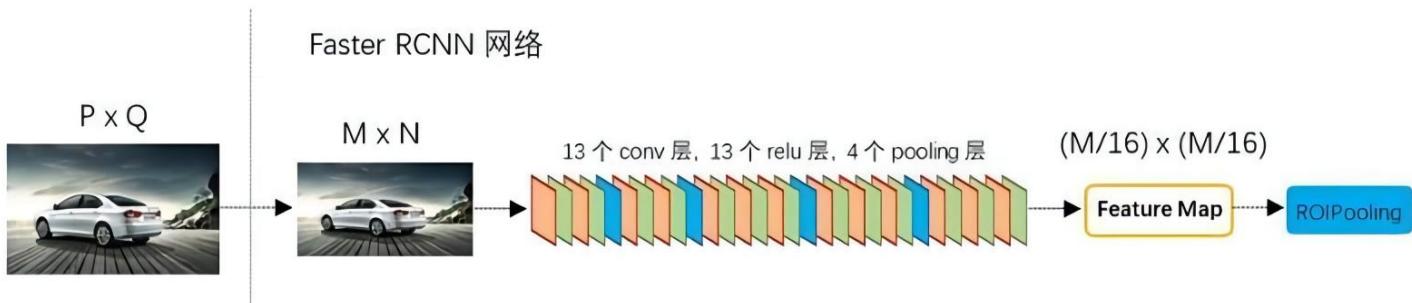


图13

Proposal Layer forward (caffe layer的前传函数) 按照以下顺序依次处理:

- 生成 anchors, 利用对所有的 anchors 做 bbox regression 回归 (这里的anchors生成和训练时完全一致)。
- 按照输入的 positive softmax scores 由大到小排序 anchors, 提取前 pre_nms_topN(e.g. 6000) 个 anchors, 即提取修正位置后的 positive anchors。
- 限定超出图像边界的 positive anchors 为图像边界, 防止后续 roi pooling 时 proposal 超出图像边界 (见文章底部QA部分图21)。
- 剔除尺寸非常小的 positive anchors。
- 对剩余的 positive anchors 进行NMS (nonmaximum suppression)。
- Proposal Layer有3个输入。

之后输出 $\text{proposal}=[x_1, y_1, x_2, y_2]$, 注意, 由于在第三步中将 anchors 映射回原图判断是否超出边界, 所以这里输出的 proposal 是对应 $M \times N$ 输入图像尺度的, 这点在后续网络中有用。另外我认为, 严格意义上的检测应该到此就结束了, 后续部分应该属于识别了。

RPN网络结构就介绍到这里, 总结起来就是:

- 生成 anchors →
- softmax分类器提取 positvie anchors →
- bbox reg 回归 positive anchors →
- Proposal Layer 生成proposals

3 ROI pooling

ROI Pooling层则负责收集 proposal，并计算出 proposal feature maps，送入后续网络。从图2中可以看到 ROI pooling 层有 2 个输入：

- 原始的feature maps
- RPN输出的proposal boxes（大小各不相同）

3.1 为何需要 ROI Pooling

先来看一个问题：对于传统的CNN（如AlexNet和VGG），当网络训练好后，输入的图像尺寸必须是固定值，同时网络输出也是固定大小的 vector or matrix。如果输入图像大小不定，这个问题就变得比较麻烦。**有2种解决办法：**

- 从图像中 crop 一部分传入网络。
- 将图像 warp 成需要的大小后传入网络。

两种办法的示意图如图14，可以看到无论采取那种办法都不好，要么 crop 后破坏了图像的完整结构，要么 warp 破坏了图像原始形状信息。



图14 crop 与 warp 破坏图像原有结构信息

回忆 RPN 网络生成的 proposals 的方法：

对 positive anchors 进行 bounding box regression，那么这样获得的 proposals 也是大小形状各不相同，即也存在上述问题。所以 Faster R-CNN 中提出了 ROI Pooling 解决这个问题。**不过 ROI Pooling 确实是从 Spatial Pyramid Pooling 发展而来**，但是限于篇幅这里略去不讲，有兴趣的读者可以自行查阅相关论文。

<https://link.zhihu.com/?target=https%3A//arxiv.org/abs/1406.4729>

3.2 ROI Pooling原理

分析之前先来看看 ROI Pooling Layer 的 caffe prototxt 的定义：

其中有新参数 pooled_w 和 pooled_h，另外一个参数 spatial_scale 认真阅读的读者肯定已经知道知道用途。

ROI Pooling layer forward 过程：

```
layer {
    name: "roi_pool5"
    type: "ROIPooling"
    bottom: "conv5_3"
    bottom: "rois"
    top: "pool5"
    roi_pooling_param {
        pooled_w: 7
        pooled_h: 7
        spatial_scale: 0.0625 # 1/16
    }
}
```

总结：

- 由于 proposal 是对应 $M \times N$ 尺度的，所以首先使用 spatial_scale 参数将其映射回 $(M/16) \times (N/16)$ 大小的 feature map 尺度；
- 再将每个 proposal 对应的 feature map 区域水平分为 $pooled_w \times pooled_h$ 的网格；
- 对网格的每一份都进行 max pooling 处理。

这样处理后，即使大小不同的 proposal 输出结果都是 $pooled_w \times pooled_h$ 固定大小，实现了固定长度输出。

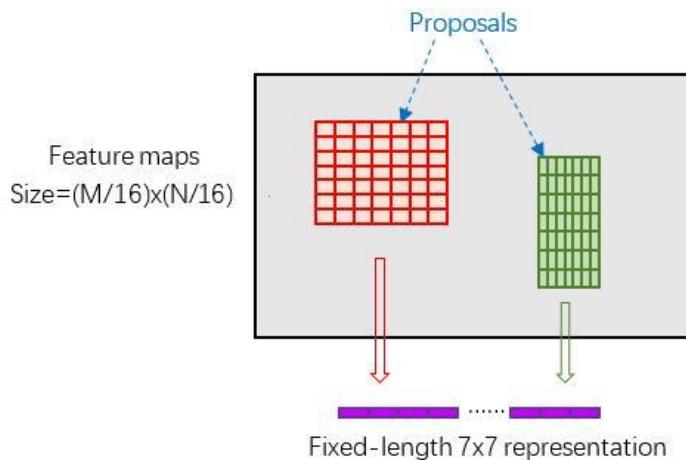


图15 proposal示意图

4 Classification

Classification 部分利用已经获得的 proposal feature maps，通过 full connect 层与 softmax 计算每个 proposal 具体属于那个类别（如人，车，电视等），输出 cls_prob 概率向量；同时再次利用 bounding box regression 获得每个 proposal 的位置偏移量 bbox_pred，用于回归更加精确的目标检测框。

Classification部分网络结构如图16。

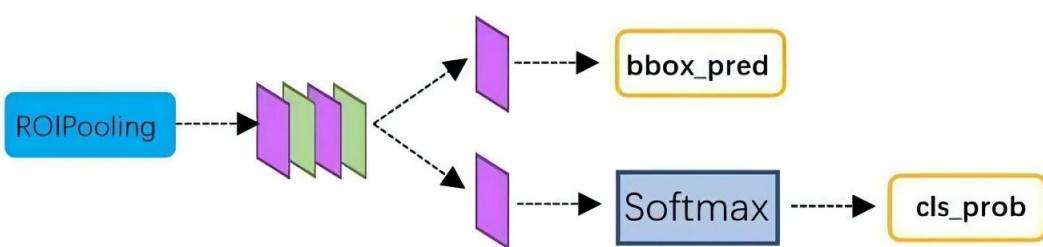


图16 Classification 部分网络结构图

从 RoI Pooling 获取到 $7 \times 7 = 49$ 大小的 proposal feature maps 后，送入后续网络，可以看到做了如下2件事：

- 通过全连接和 softmax 对 proposals 进行分类，这实际上已经是识别的范畴了。
- 再次对 proposals 进行 bounding box regression，获取更高精度的 rect box。

5 Faster RCNN训练

Faster R-CNN 的训练，是在已经训练好的 ImageNet-pre-trained model (如VGG_CNN_M_1024, VGG, ZF) 的基础上继续进行训练。实际中训练过程分为6个步骤：(【平】建议详见 Faster RCNN 原论文 3.2 节的 4-Step Alternating Training)

1. 在已经训练好的 ImageNet-pre-trained model 上，训练 RPN 网络，对应 stage1_rpn_train.pt。
2. 利用步骤1中训练好的 RPN 网络，收集 proposals，对应 rpn_test.pt。
3. 第一次训练 Fast RCNN 网络，对应 stage1_fast_rcnn_train.pt。
4. 第二次训练 RPN 网络，对应 stage2_rpn_train.pt
5. 再次利用步骤4中训练好的 RPN 网络，收集 proposals，对应 rpn_test.pt
6. 第二次训练 Fast RCNN 网络，对应 stage2_fast_rcnn_train.pt

可以看到训练过程类似于一种“迭代”的过程，不过只循环了2次。至于只循环了2次的原因是应为作者提到：“A similar alternating training can be run for more iterations, but we have observed negligible improvements”，即循环更多次没有提升了。接下来本章以上述6个步骤讲解训练过程。

下面是一张训练过程流程图，应该更加清晰：

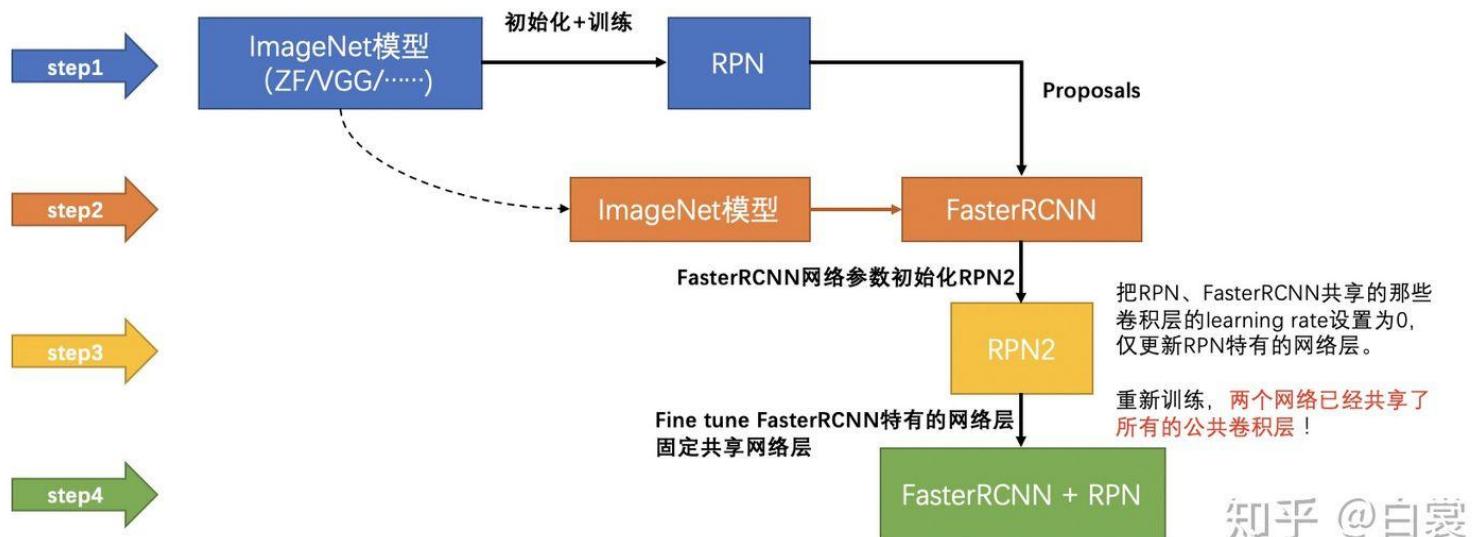


图18 (图17没写入此笔记)

5.1 训练RPN网络

在该步骤中，首先读取 RBG 提供的预训练好的 model (本文使用VGG)，开始迭代训练。来看看 stage1_rpn_train.pt 网络结构，如图19。

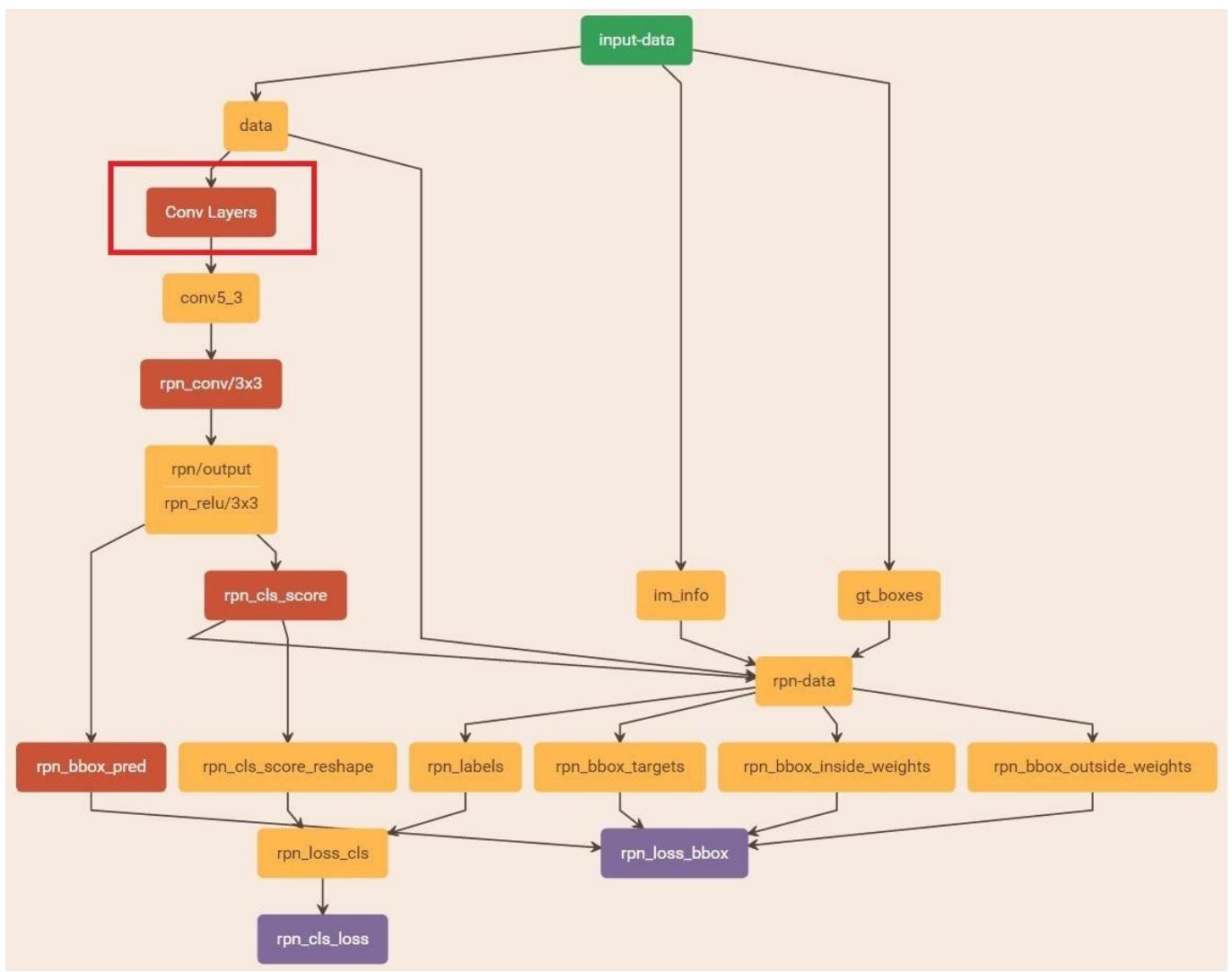


图19 stage1_rpn_train.pt (考虑图片大小, Conv Layers中所有的层都画在一起了, 如红圈所示, 后续图都如此处理)

与检测网络类似的是, 依然使用 Conv Layers 提取 feature maps。整个网络使用的Loss如下:

$$L(\{p_i\}, \{t_i\}) = \frac{1}{N_{cls}} \sum_i L_{cls}(p_i, p_i^*) + \lambda \frac{1}{N_{reg}} \sum_i p_i^* L_{reg}(t_i, t_i^*) \quad (12)$$

上述公式中 i 表示 anchors index, p_i 表示 positive softmax probability, p_i^* 代表对应的 GT predict 概率 (即当第 i 个 anchor 与 GT 间 $IoU > 0.7$, 认为是该 anchor 是 positive, $p_i^* = 1$; 反之 $IoU < 0.3$ 时, 认为是该 anchor 是 negative, $p_i^* = 0$; 至于那些 $0.3 < IoU < 0.7$ 的 anchor 则不参与训练) ; t 代表 predict bounding box, t^* 代表对应 positive anchor 对应的 GT box。可以看到, 整个 Loss 分为 2 部分:

由于在实际过程中, N_{cls} 和 N_{reg} 差距过大, 用参数 λ 平衡二者 (如 $N_{cls} = 256$, $N_{reg} = 2400$ 时设置 $\lambda = \frac{N_{reg}}{N_{cls}} \approx 10$) , 使总的网络 Loss 计算过程中能够均匀考虑 2 种 Loss。这里比较重要是 L_{reg} 使用的 smooth L1 loss, 计算公式如下:

了解数学原理后, 反过来看图18: (略, 详见原文)

特别需要注意的是, 在训练和检测阶段生成和存储 anchors 的顺序完全一样, 这样训练结果才能被用于检测!

5.2 通过训练好的 RPN 网络收集 proposals

在该步骤中，利用之前的 RPN 网络，获取 proposal rois，同时获取 positive softmax probability，如图20，然后将获取的信息保存在python pickle文件中。该网络本质上和检测中的RPN网络一样，没有什么区别。

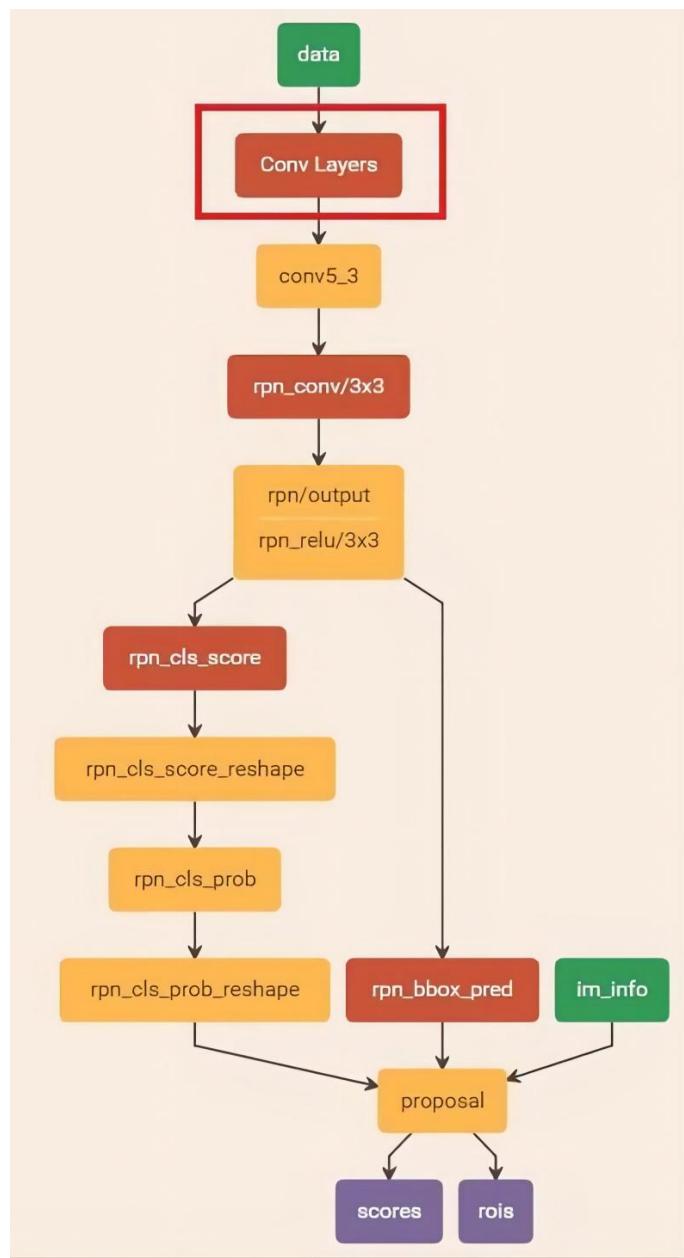


图20 rpn_test.pt

5.3 训练 Faster RCNN 网络

读取之前保存的 pickle 文件，获取 proposals 与 positive probability。从 data 层输入网络。然后：

- 将提取的 proposals 作为 rois 传入网络，如图21蓝框。
- 计算 bbox_inside_weights+bbox_outside_weights，作用与 RPN 一样，传入 smooth_L1_loss layer，如图21绿框。

这样就可以训练最后的识别 softmax 与最终的 bounding box regression了。

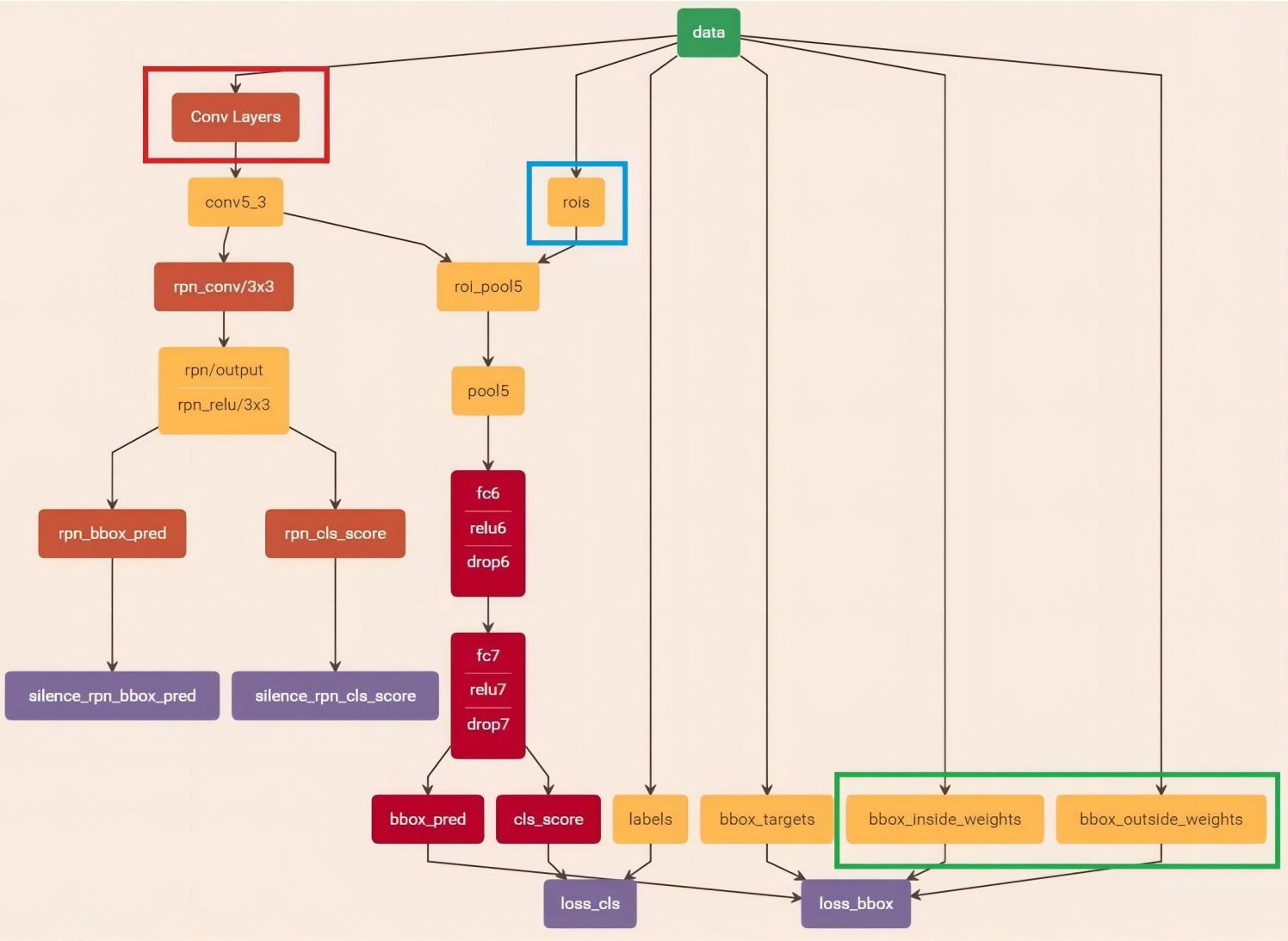


图21 stage1_fast_rcnn_train.pt

之后的 stage2 训练都是大同小异，不再赘述了。

Faster R-CNN 还有一种 end-to-end 的训练方式，可以一次完成 train，有兴趣请自己看作者 GitHub 吧。
<https://link.zhihu.com/?target=https%3A//github.com/rbgirshick/py-faster-rcnn>

QA

为什么 anchor 坐标中有负数?

回顾 anchor 生成步骤：首先生成 9 个 base anchor，然后通过坐标偏移在 50×38 大小的 $1/16$ 下采样 FeatureMap 每个点都放上这 9 个 base anchor，就形成了 $50 \times 38 \times k$ 个 anchors。至于这 9 个 base anchor 坐标是什么其实并不重要，不同代码实现也许不同。

显然这里面有一部分边缘 anchors 会超出图像边界，而真实中不会有超出图像的目标，所以会有 clip anchor 步骤。

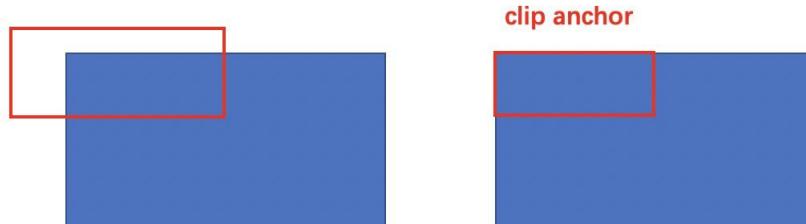


图22 clip anchor

Anchor 到底与网络输出如何对应？

VGG 输出 $50 \times 38 \times 512$ 的特征，对应设置 $50 \times 38 \times k$ 个 anchors，而 RPN 输出 $50 \times 38 \times 2k$ 的分类特征矩阵和 $50 \times 38 \times 4k$ 的坐标回归特征矩阵。其实在实现过程中，每个点的 $2k$ 个分类特征与 $4k$ 回归特征，与 k 个 anchor 逐个对应即可，这实际是一种“人为设置的逻辑映射”。当然，也可以不这样设置，但是无论如何都需要保证在训练和测试过程中映射方式必须一致。

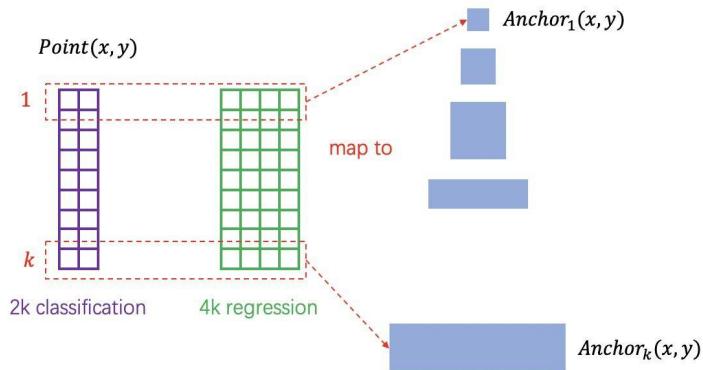


图23 anchor 与网络输出对应方式

为何有 ROI Pooling 还要把输入图片 resize 到固定大小的 MxN?

由于引入 ROI Pooling，从原理上说 Faster R-CNN 确实能够检测任意大小的图片。但是由于在训练的时候需要使用大 batch 训练网络，而不同大小输入拼 batch 在实现的时候代码较为复杂，而且当时以 Caffe 为代表的第一代深度学习框架也不如 Tensorflow 和 PyTorch 灵活，所以作者选择了把输入图片 resize 到固定大小的 800×600 。这应该算是历史遗留问题。另外很多问题，都是属于具体实现问题，真诚的建议读者阅读代码自行理解。

拓展

- 关于torchvision中的FasterRCNN代码：<https://zhuanlan.zhihu.com/p/145842317>
- Faster RCNN在文字检测中的应用：CTPN <https://zhuanlan.zhihu.com/p/34757009>
- Faster RCNN在高德导航中的应用：
https://link.zhihu.com/?target=https%3A//mp.weixin.qq.com/s/IJUMCOBhgXHv7VC1YT4q_g