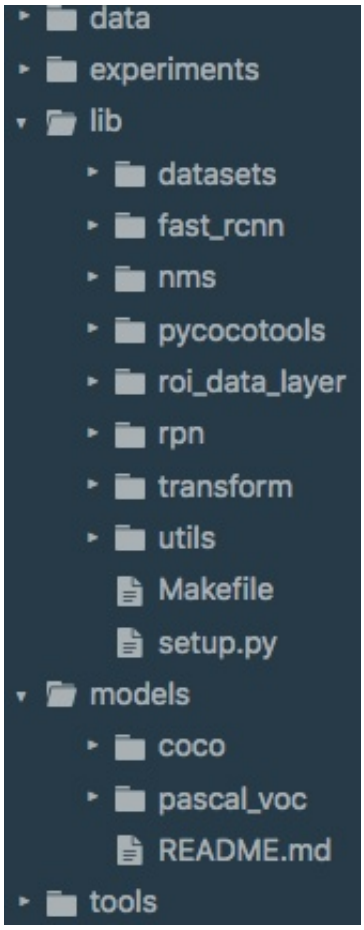
简介

此文承接《Faster R-CNN 论文阅读记录（一）：概览》。Faster R-CNN 论文是我目前阅读论文里面花费的时间最长的，前前后后差不多花费了一周的时间。被里面的训练细节、anchor 的挑选等困惑，在网上找了很多资料还不不明白到底是怎么回事。看这些资料也是云里雾里，不知道他们在说些什么，而且还有多资料说的内容存在矛盾的地方，看这些资料真心累。后来索性去看了下 Faster R-CNN 的源码，通过可视化网络结构才算是了解到了它究竟是什么样。再进一步阅读源码内容和论文，才算是感觉有点通透了。下面详细介绍下我通过阅读源码和论文的一些收获。

Faster R-CNN 训练方式在源码中给出了两种，即交替训练 RPN 和 Fast R-CNN、end2end 方式一次训练两个网络。在查阅资料的时候，会看到很多讲解 Faster R-CNN 训练的文章，但大部分文章都只是讲解了 Faster R-CNN 的种训练方式，所以就看到有的文章这么说，另一些文章这么讲，不知道究竟发生了什么。这里就把两种训练方式都做介绍，这里不涉及到源码细节，只在于把训练过程讲述明白，方便去阅读其他讲解文章。

caffe 模型结构的可视化使用了 Netscope 在线工具，将 caffe 定义的模型内容粘贴进去即可。

源码结构

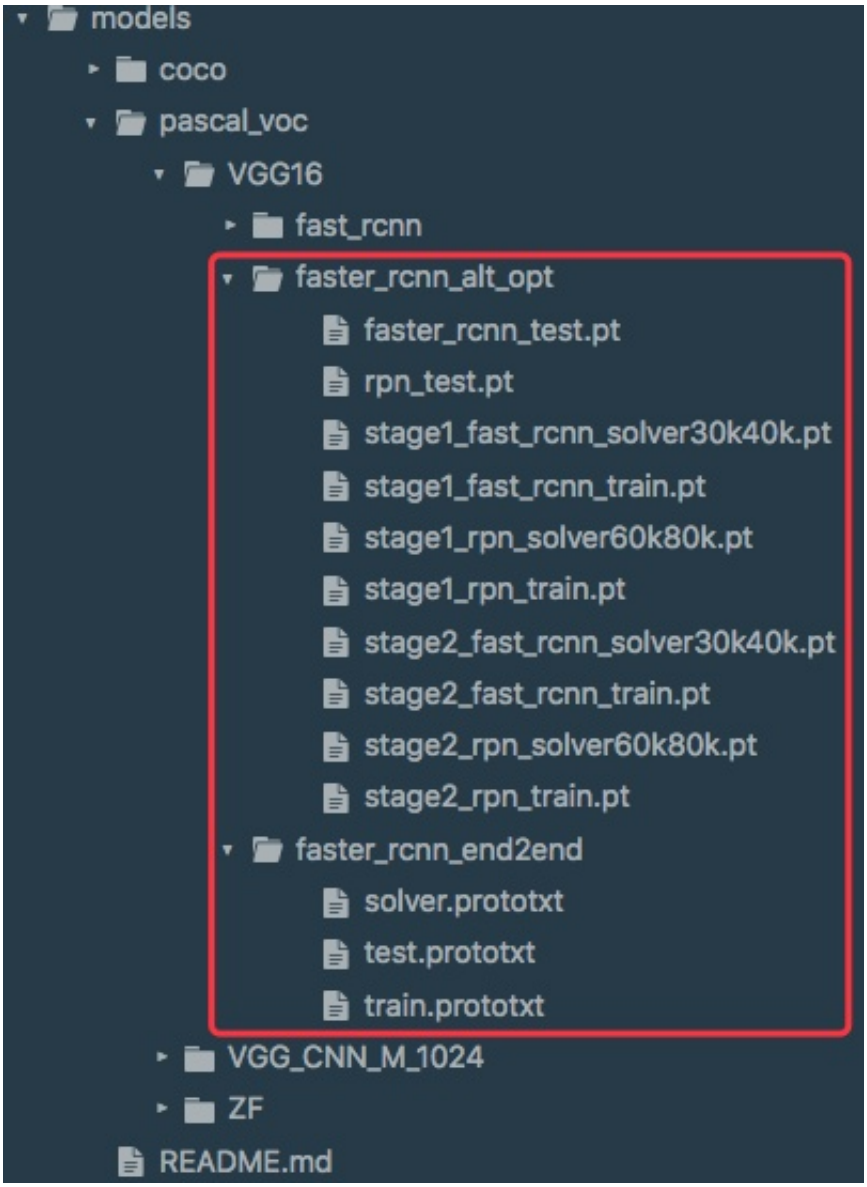


源码分为以下几个内容：

- data：存放 caffe 预训练模型、Faster R-CNN模型、数据集的符号链接
- experiments：里面的脚本用于做实验

- lib：Faster R-CNN 实现的主要代码
- models：在不同的数据集上训练使用的模型，文中结构图均参考此做成
- tools：训练和测试使用的工具

文中结构图使用 models 中 PASCAL VOC 下的 VGG16 做成：



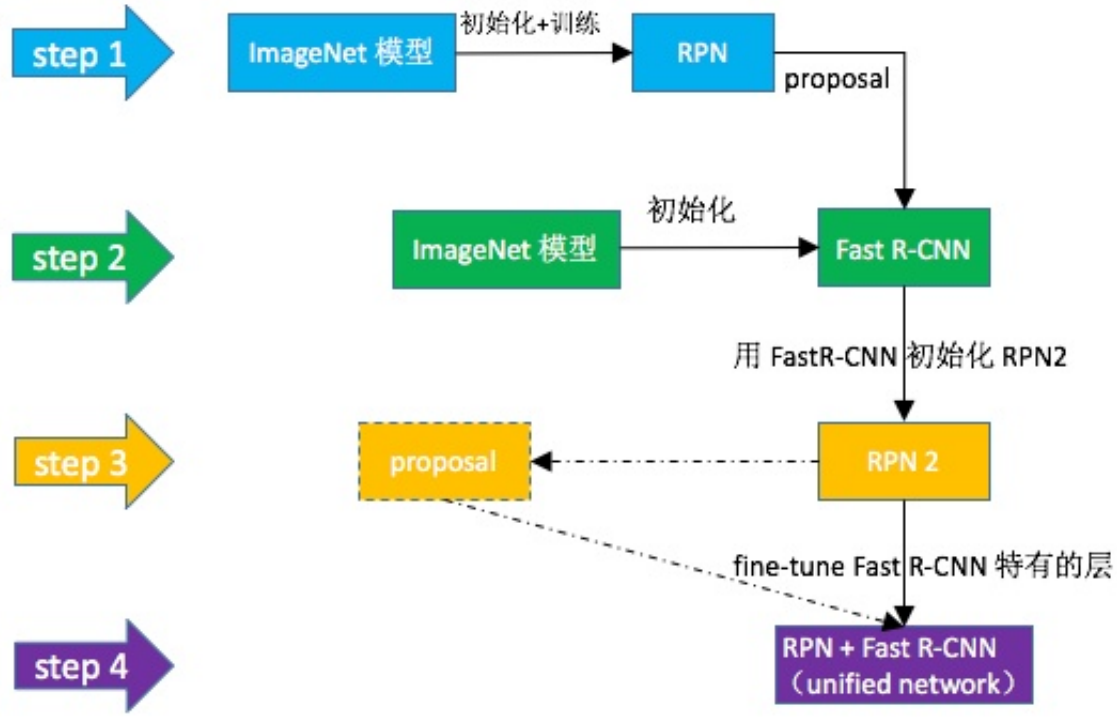
faster_rcnn_alt_opt 中的 alt_opt 值得就是 alternating optimization ，即交替优化，交替训练 RPN 和 fast rcnn，主要实现过程在 /tools/train_fast_rcnn_alt_opt.py 文件中。

end2end 就是端对端一次训练 RPN 和 fast rcnn。

Faster R-CNN 四步交替训练（alternating optimization）

1. 训练 RPN，通过在 ImageNet 上预训练的 VGG16 来初始化 RPN 网络，对网络进行微调。通过训练，使得 RPN 可以产生好的 region proposal。最后把初步训练好的 RPN 网络模型保存供接下来训练 Fast R-CNN 使用。
2. 使用 ImageNet 上预训练的 VGG16 来初始化 Fast R-CNN ，使用上一步训练好的 RPN 网络模型来生成 region proposal 作为输入数据。此时训练的模型还没有共享卷积层。

3. 使用上一步训练的 fast r-cnn 来初始化 RPN 网络，再次训练 RPN 模型，这次有了共享的卷积层，并且这次只微调 RPN 模型特有的层。训练好之后将模型保存。
4. 使用上一步训练的 RPN 来初始化 fast r-cnn 网络，并使用上一步保存的 RPN 模型来生成 region proposal 作为输入数据。保持共享的卷积层不变，微调 fast r-cnn 独有的层。这样两个网络就共享了卷积层，构成了一个统一的模型。

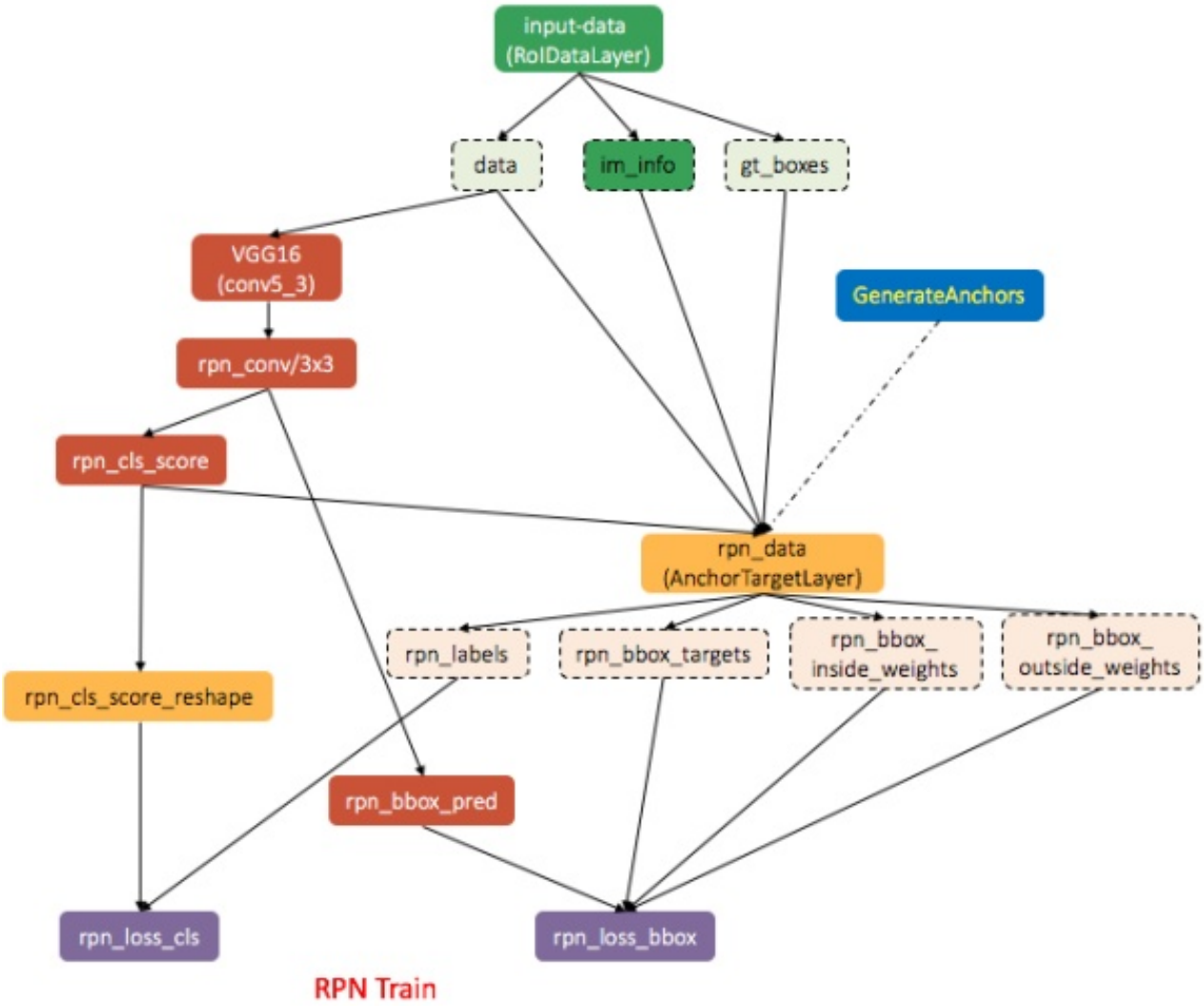


训练过程的实现在 train_faster_rcnn_alt_opt.py 中。此四步训练划分为两个阶段，前两个步骤划分到第一阶段，后两个步骤划分到第二阶段。

第一阶段

第一步：训练 RPN

训练 RPN 的结构图如下，由 stage1_rpn_train.pt 定义的结构绘制：



此时训练使用的数据是有标签的，即 PASCAL VOC 提供的训练数据，通过 ROIDataLayer 来完成数据的提取。在训练时只需要图片数据 data、图片信息数据（宽高缩放） im_info、物体定位数据 gt_boxes，训练 RPN 模型用不到被检测物体的 labels 数据，因此 RoIDataLayer并没有输出 labels 数据。但是在训练 fast r-cnn 时是需要的。

在上图的左侧部分，输出了 rpn_cls_score，每个位置有 9 个 anchor，需要判断这 9 anchor 是前景还是背景，这样输出就有 2×9 = 18 个分数。rpn_bbox_pred 则会输出预测的 box 的位置，每个 anchor 有 4 个参数确定，中心点 x、y 和w、h 宽高，这样每个位置就会输出 4×9=36 个值，此时输出的是 Δx、Δy、Δw、Δh 并不是 box 的实际位置。

AnchorTargetLayer

右侧部分，也就是 rpn_data ，用来生成用于计算 RPN 损失的部分，也是关键部分。此部分主要是由 AnchorTargetLayer 来完成，主要代码部分在./lib/rpn/anchor_target_layer.py 中。该层的定义如下：

```
layer {
  name: 'rpn-data'
  type: 'Python'
  bottom: 'rpn_cls_score'
  bottom: 'gt_boxes'
  bottom: 'im_info'
  bottom: 'data'
```

```
    top: 'rpn_labels'
    top: 'rpn_bbox_targets'
    top: 'rpn_bbox_inside_weights'
    top: 'rpn_bbox_outside_weights'
    python_param {
      module: 'rpn.anchor_target_layer'
      layer: 'AnchorTargetLayer'
      param_str: "'feat_stride': 16"
    }
  }
```

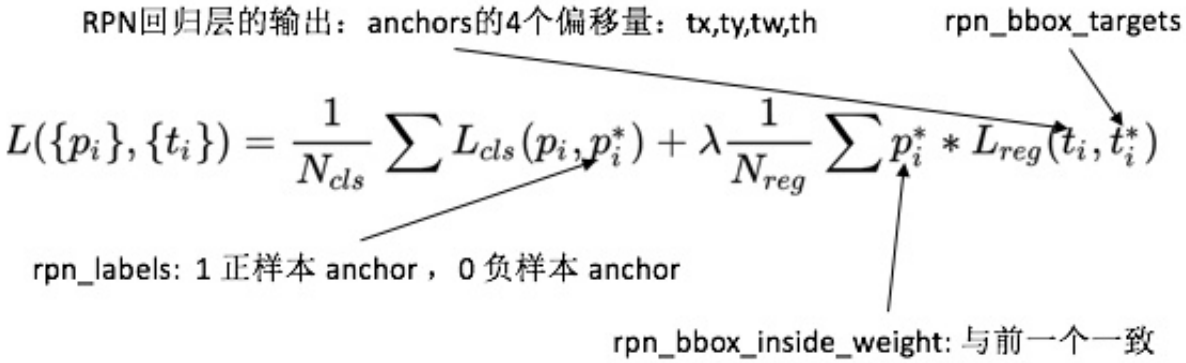
在代码中按照 bottom 的顺序就可以获得相应的输入，如 bottom[0] 指的就是 rpn_cls_score。在 AnchorTargetLayer 中首先由 generate anchors 生成 anchor，主要是 9 种类型的 anchor。之后再由 AnchorTargetLayer 依据这 9 中比例，依据 VGG16 最终生成 feature map 的大小和累积 stride，生成在原始图片上的尺寸。对于尺寸为 1000 × 600 的图片大约生成 21 k 个 anchor。但这 21k 个 anchor 有一部分是超出了原图片，需要把超出边界的 anchor 去除掉，这样剩下大约 6000 个 anchor。

接下来就需要为生成的这些 anchor 打上标签，用于区分前景（fg）和背景（bg），这样就可以用于计算 rpn_loss_cls 。标签为 1 表示正样本，标签为 0 表示负样本，标签为 -1 表示忽略不关心。 生成标签需要gt_boxes 的信息，然后使用 IoU 来判断是否符合标准。生成标签的规则如下：

- 与 gt_boxes 的 IoU 最大的 anchor 标记为正样本，即 label 为 1，前景。这样做的目的是为了让每一个 gt_boxes 都分配到了一个 anchor，避免了使用下面👇的策略时，有的 gt_boxes 无法分配到 anchor；
- 与 gt_boxes 的 IoU 大于 0.7 的 anchor 标记为正样本，即 label 为 1，前景。这样做的目地是为了生成足够多的正样本；
- 与 gt_boxes 的 IoU 小于 0.3 的 anchor 标记为负样本，即 label 为 0，背景；
- 与 gt_boxes 的 IoU 在[0.3 , 0.7] 之间的 anchor 分配标签为 -1，这部分 anchor 既含有前景又含有背景，不适合做训练。

但并不是所有标记为正样本和负样本的 anchor 都会用于损失的计算，而是从中抽取 256 个样本，正负样本的比例是 1:1，即正负样本各 128 个。如果正样本的数量不够，就用负样本来补充。上面筛选的过程并不是把其他的都删除掉了，而是把需要的256个 anchor 筛选出来，其他的 anchor 都标记为 -1。这样 6000 个 anchor 仍在，只是大部分 anchor 都被标记为了 - 1，除了计算损失需要的这 256 个 anchor。

通过 AnchorTargetLayer 处理之后就生成了 rpn_labels、rpn_bbox_targets、rpn_bbox_inside_weights、rpn_bbox_outside_weights 四个输出，接下来就是计算 RPN 的损失。损失函数如下：

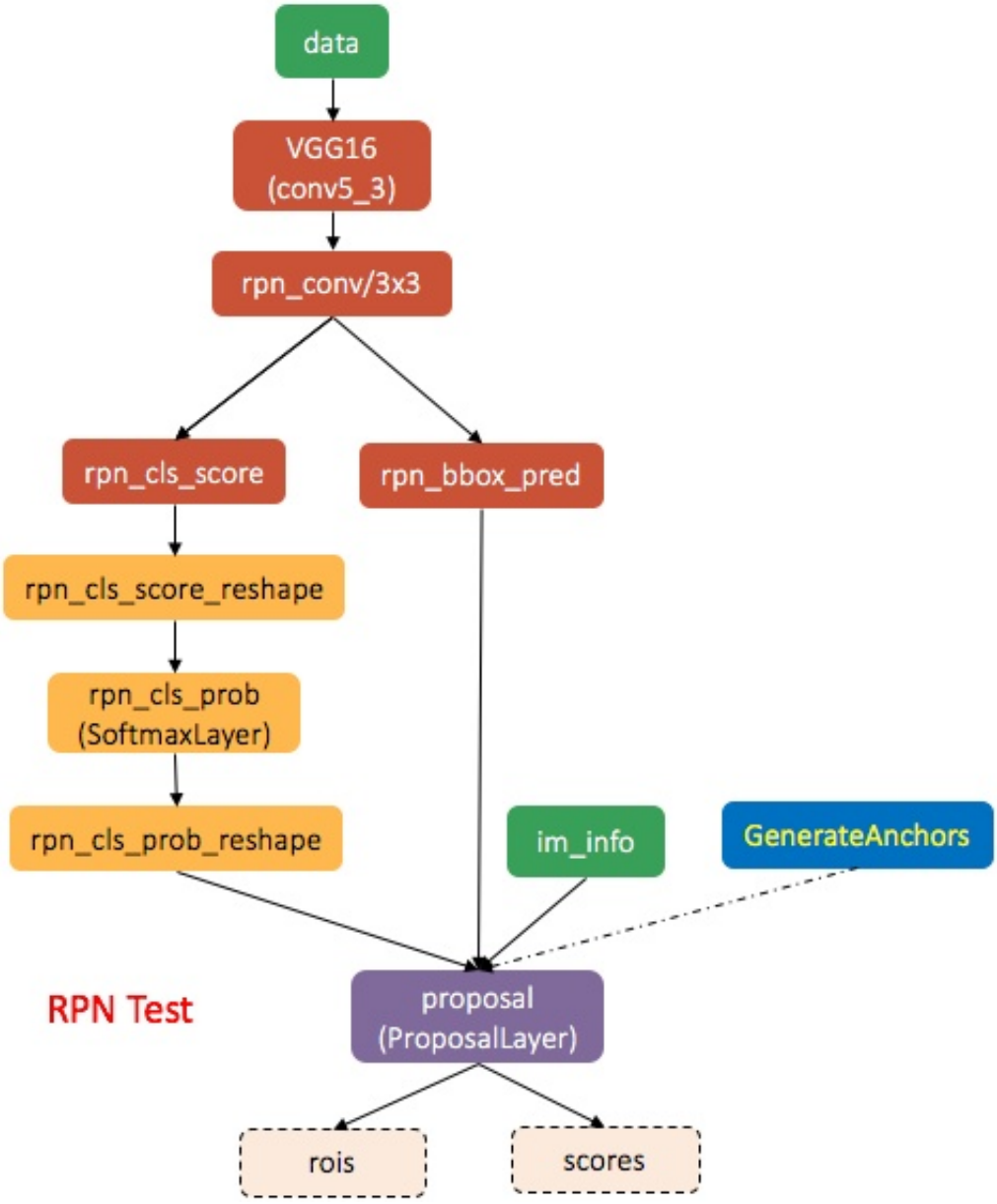


通过不停的迭代直到 RPN 网络收敛，这样就训练好了 RPN 模型。之后将训练好的模型保存供后来使用。

第二阶段的 RPN 训练与此相似，初始化时使用的是第二步训练的 Fast R-CNN。

使用 RPN 生成 rois

在训练 Fast R-CNN 之前，需要使用上面训练好的 RPN 网络来生成 rois。生成 rois 时的网络结构如下：



生成 rois 的结构也是测试 RPN 网络时的结构，使用的都是 rpn_test_prototxt 配置文件。与 RPN 训练是的结构有不少的区别，首先少了 AnchorTargetLayer，新增了 ProposalLayer，用来生成 rois 和 每个 rois 的得分。此时的输入是无监督的数据，输入data 只有图像数据，需要由 RPN 来生成相应的 rois。

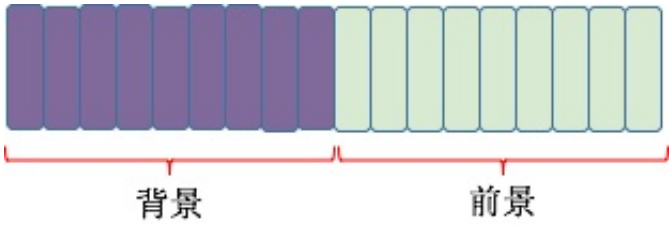
这里的关键就是 ProposalLayer 层，该层的代码在 lib/rpn/proposal_layer.py 文件中，下面就看看此层都做了哪些事情。

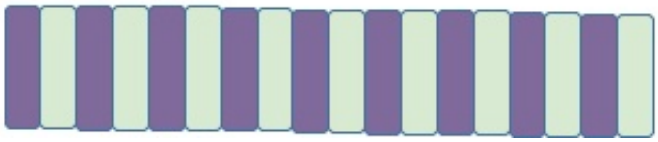
该层的输入包括 rpn_cls_prob_reshape、rpn_bbox_pred、im_info，还有不在层定义中的generate anchor 层，定义如下：

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

首先依然是使用 generate anchor 生成 9 中尺寸的 anchor，然后通过 im_info 中的信息，枚举出全部 21k 个 anchor 的信息。

之后从 rpn_cls_prob_reshape 取出前景框的概率，这里需要说明下，在 rpn_cls_prob_reshape 中，前九个是背景框的概率，后九个才是前景的概率，因此需要取出后面九个的概率得分：





通过生成的 21k 个 anchor 与 rpn_bbox_pred 生成的 bbox_deltas $(\Delta x、\Delta y、\Delta w、\Delta h)$数据，完成对 anchor 的转换，生成实际预测的 proposals。

在 RPN 训练的阶段是移除了超出边界的 anchor，这里不移除超出边界的 proposals（需要注意这里的称呼 anchor 与 proposals 的不同），而是将超出的部分裁剪掉。

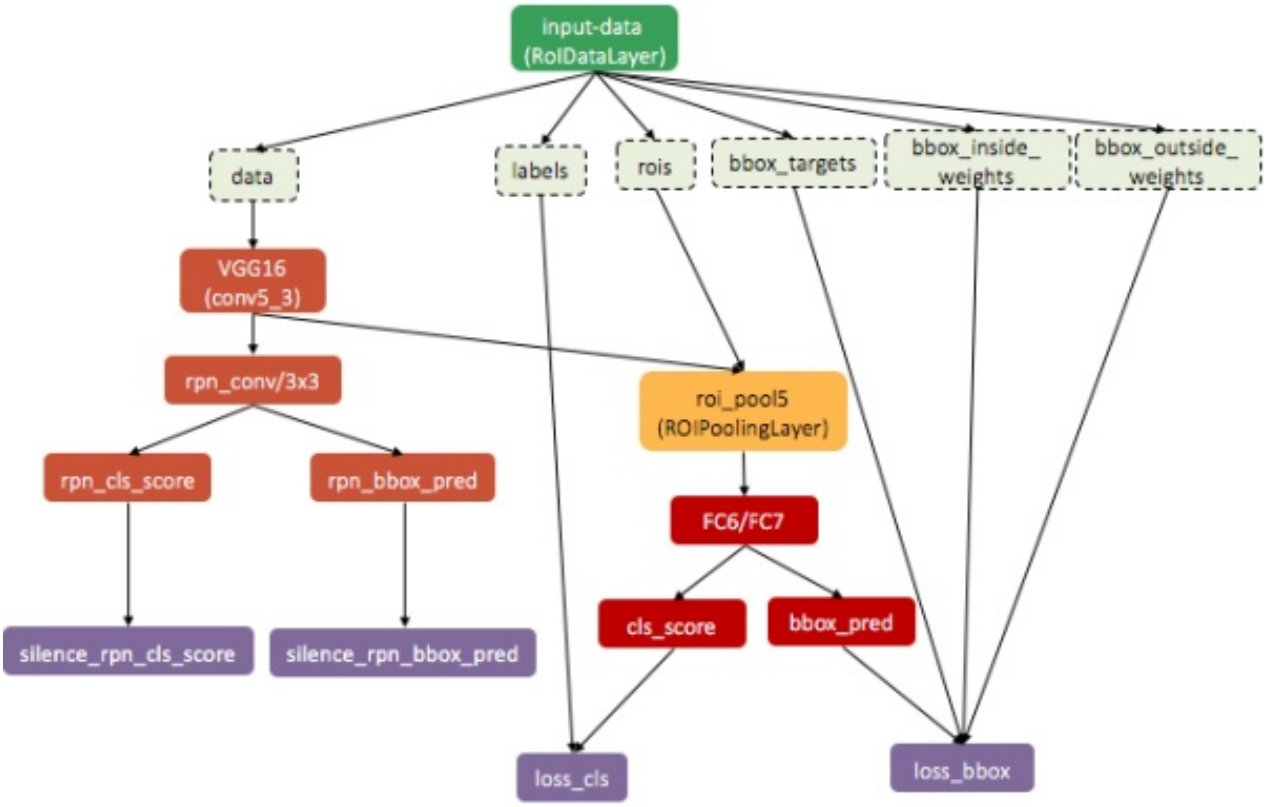
有了裁剪就会造成有的 proposals 宽、高过小，因此需要把宽高过小的 proposal 过滤掉，这里的宽高指的是在原图上的尺寸，设定的阈值为 16 ，即如果宽或者高小于 16 的过滤掉不要。

按照 proposals 的得分（前景的得分），从大到小排序，取前 6000 （pre_nms_topN）个。

最后使用对着 6000 个 proposals 应用 NMS，NMS 设置阈值为 0.7，然后从剩下的 proposals 中按分值排序选择 300 （after_nms_topN）个，将这 300 个 proposals 返回，就是最终需要的 rois，将这些 rois 存储到磁盘上供 fast r-cnn 训练使用。

第二步：训练 Fast R-CNN

有了通过 RPN 生成的 rois 就可以训练 Fast R-CNN，使用 ImageNet 上预训练的模型 VGG16 来初始化 Fast R-CNN ，训练时的结构如下：



Fast R-CNN Train

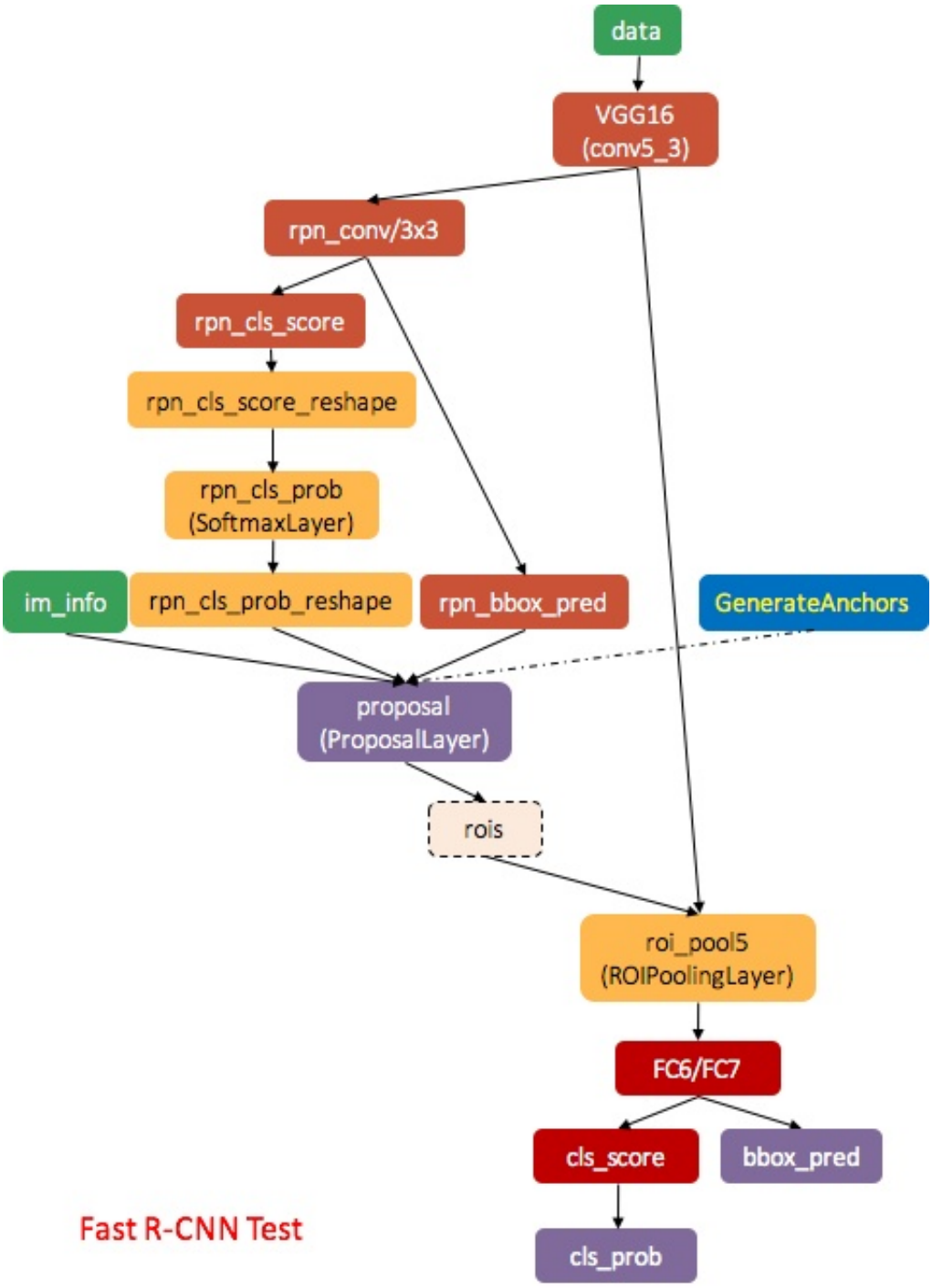
上图的 rois 就是 RPN 生成的 proposals 数据。训练 fast r-cnn 的 batch_size 大小为 128 ，设置的前景参数为 0.25 ，也就是如果使用一张图，那么每张图需要提供 32 个前景 roi。如果是两张图，那么每张图提供 16 个情景 roi。

通过 RoIDataLayer 来生成所需要的训练数据，这里数据的生成涉及到的模块比较多，而且也相对比较复杂，暂时没有深入的了解，以后了解深入了再做补充。

将需要的数据准备完成之后就可以对 fast r-cnn 进行训练，此训练的方式与之前 fast r-cnn 的训练方式一样，并没有太大区别。也是使用了 ROIPooling 让大小不同的图片生成大小一致的 feature map，供以后的分类和 bbox reg 使用。损失的计算也与 fast r-cnn 一致，不再说明。

测试Fast R-CNN

训练完成之后，就可以对 fast r-cnn 进行测试，测试的结构如下：



左侧的 RPN 生成 rois 的过程与训练 Fast R-CNN 是使用 RPN 生成 rois 一致，可以参考上面的讲解。但在此的数据并不是有标签的数据，需要模型自己来做推断。至此第一阶段的训练就就完成了，接下来就是第二阶段的训练。

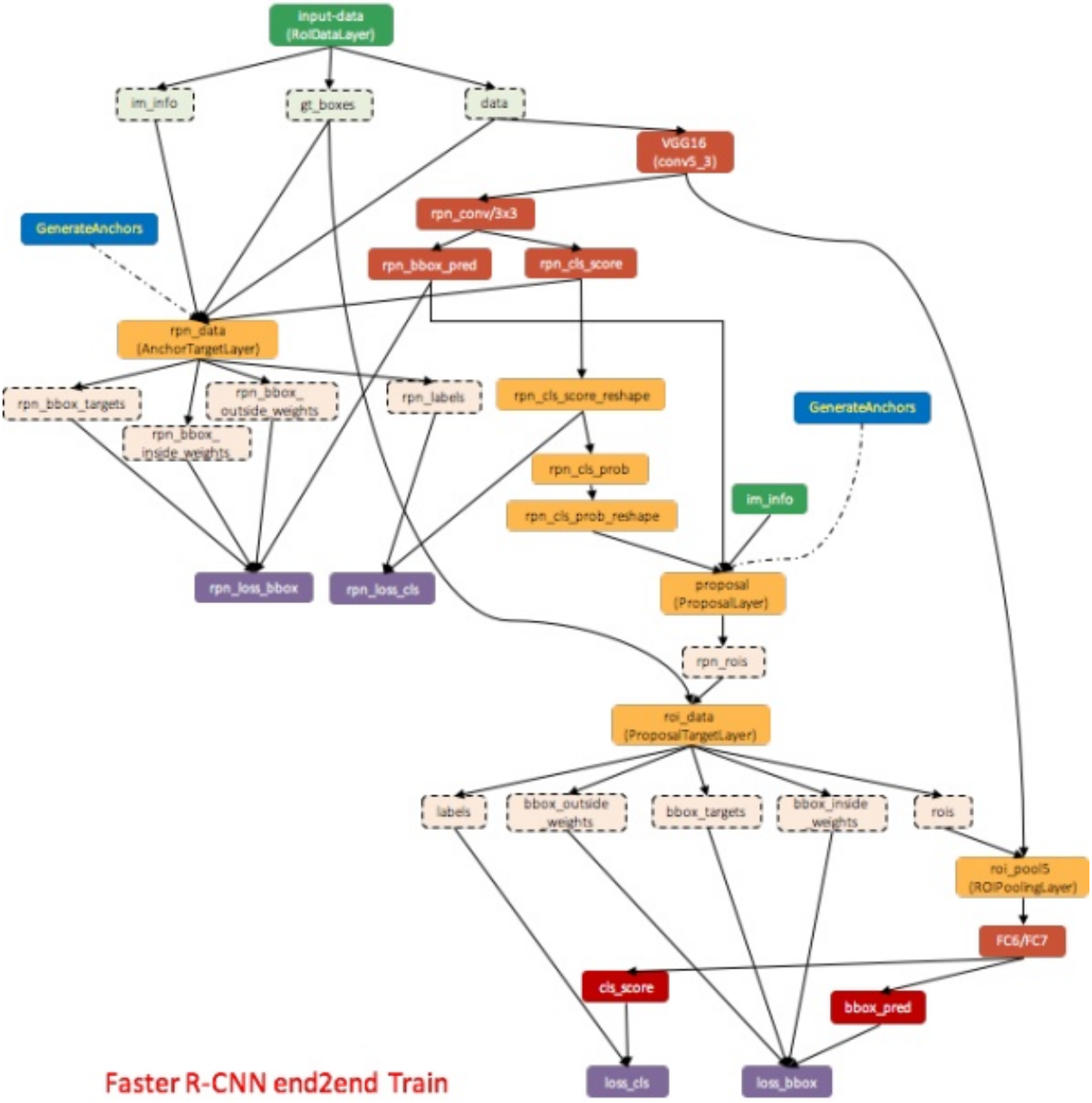
第二阶段

完成上面第一阶段训练之后，第二阶段的训练与第一阶段类似。只是第三步训练 RPN 初始化时使用的是训练的 Fast R-CNN 的参数，第四步训练 Fast R-CNN 初始化使用的是第三步训练的 RPN 模型的参数。其他操作训练方式均相同。

至此，四步交替训练就算成此了，最终使用的模型与上图一致。

融合训练

论文中也提到了融合训练，此训练方式比四步交替训练快很多，但精度却没有损失。end2end 训练的结构图如下：



可以看到此方式融合了图4 RPN 训练 和图7 Fast R-CNN 训练，但 Fast R-CNN 的 input_data 换成了 roi_data 由 ProposalTargetLayer 来完成对 rois、labels、bbox_targets、bbox_inside_weights、bbox_outside_weights 数据的生成。

融合的训练过程需要计算四个损失函数，其他的与四步交替训练的过程类似。测试的过程与测试 Faster R-CNN 一样。

融合训练的方式更新细的资料可以参考《Object Detection and Classification using R-CNNs》这篇文章，作者全面讲述了训练过程和一些细节。

【参考】

- 个站 - “Fast R-CNN and Faster R-CNN”
- CSDN - 深度学习: RPN (区域候选网络)
- 个站 - Notes on Faster RCNN
- CSDN - 详细的Faster R-CNN源码解析之RPN源码解析

- CSDN - 深度学习: one-stage/two-stage/multi-stage 目标检测算法
- 知乎 - 检测任务专题2: two-stage检测
- 简书 - 在Caffe中加Python Layer的方法
- 简书 - 深度学习caffe框架(2): layer定义
- 个站 - Training R-CNNs of various velocities Slow, fast, and faster rbg 大神的Faster R-CNN 讲座
- 知乎 - Faster-RCNN四步交替法源码阅读笔记
- 知乎 - 从编程实现角度学习Faster R-CNN（附极简实现） 其中的模型架构图可以清楚的看到反向传播是怎么发生的。
- CSDN - 详细的Faster R-CNN源码解析之proposal_layer和proposal_target_layer源码解析
- CSDn - faster rcnn源码解读（五）之layer（网络里的input-data） RoIDataLayer
- CSDN - faster-rcnn 之 基于roidb get_minibatch（数据准备操作）