Pytorch-Faster-RCNN Code Analysis

faster-rcnn.pytorch:

|-- cfgs

res101.yml

|-- lib

|-- datasets

|-- tools

pascal_voc.py

imdb.py

|-- model

|-- faster_rcnn

faster_rcnn.py

resnet.py

|-- rpn

anchor_target_layer.py

bbox_transform.py

generate_anchors.py

proposal_layer.py

proposal_target_layer_cascade.py

rpn.py

|-- utils

blob.py

config.py

logger.py

net_utils.py

|-- nms

nms_warapper.py

|-- roi_align

|-- roi_crop

|-- roi_pooling

|-- roi_data_layer

minibatch.py

roibatchLoader.py

roidb.py

|-- pycocotools

trainval_net.py

test_net.py

demo.py

roibatchLoader.py

输入为roidb，ratio_list，ratio_index，batch_size，num_classes

roidb：原图加翻转的图片

ratio_list：宽高比从小到大排序的列表

ratio_index：ratio_list每个值的索引

batch_size：一次送入的图片数

num_classes：类别数

返回值为data，im_fo，gt_boxes，num_boxes

data：padding后的图片

im_fo：padding后的图片尺寸和目标尺寸和图片的比例

gt_boxes：padding后的gt_boxes

num_boxes：一张图的gt_boxes数量

```
Code1:
if self._roidb[index_ratio]['need_crop']:
    if ratio < 1:
        # this means that data_width << data_height, we need to crop the
        # data_height
        min_y = int(torch.min(gt_boxes[:,1]))
        max_y = int(torch.max(gt_boxes[:,3]))
        trim_size = int(np.floor(data_width / ratio))
        if trim_size > data_height:
            trim_size = data_height
        box_region = max_y - min_y + 1
        if min_y == 0:
            y_s = 0
        else:
            if (box_region-trim_size) < 0:
                y_s_min = max(max_y-trim_size, 0)
                y_s_max = min(min_y, data_height-trim_size)
                if y_s_min == y_s_max:
```

```
                    y_s = y_s_min
                else:
                    y_s = np.random.choice(range(y_s_min, y_s_max))
            else:
                y_s_add = int((box_region-trim_size)/2)
                if y_s_add == 0:
                    y_s = min_y
                else:
                    y_s = np.random.choice(range(min_y, min_y+y_s_add))
```

if self._roidb[index_ratio]['need_crop'] 判断是否剪切

if ratio < 1 和 if ratio > 1判断宽大于长还是长大于宽

if trim_size > data_height判断需要剪切的尺寸是否超出图片范围

if min_y == 0 判断一张图片中的所有目标的最小坐标是否等于0

if (box_region-trim_size) < 0 判断一张图片中的所有目标是否超出需要剪切的尺寸范围

if y_s_min == y_s_max 判断y_s可取范围是否为0(y_s表示的是gt_boxes能够移动的范围)

if y_s_add == 0 判断需要额外填充的像素值是否为0

```
Code2:
# crop the image
data = data[:, y_s:(y_s + trim_size), :, :]

# shift y coordiante of gt_boxes
gt_boxes[:, 1] = gt_boxes[:, 1] - float(y_s)
gt_boxes[:, 3] = gt_boxes[:, 3] - float(y_s)

# update gt bounding box according the trip
gt_boxes[:, 1].clamp_(0, trim_size - 1)
gt_boxes[:, 3].clamp_(0, trim_size - 1)
```

裁剪图片，移动gt_boxes，调整gt_boxes范围

Code1和Code2确保图片在ratio的范围内，且gt_boxes在图片内

```
# based on the ratio, padding the image.
if ratio < 1:
    # this means that data_width < data_height
    trim_size = int(np.floor(data_width / ratio))

    padding_data = torch.FloatTensor(int(np.ceil(data_width / ratio)), \
                                     data_width, 3).zero_()

    padding_data[:data_height, :, :] = data[0]
    # update im_info
    im_info[0, 0] = padding_data.size(0)
    # print("height %d %d \n" %(index, anchor_idx))
elif ratio > 1:
    # this means that data_width > data_height
```

```
    # if the image need to crop.
    padding_data = torch.FloatTensor(data_height, \
                                     int(np.ceil(data_height * ratio)), 3).zero_()
    padding_data[:, :data_width, :] = data[0]
    im_info[0, 1] = padding_data.size(1)
else:
    trim_size = min(data_height, data_width)
    padding_data = torch.FloatTensor(trim_size, trim_size, 3).zero_()
    padding_data = data[0][:trim_size, :trim_size, :]
    # gt_boxes.clamp_(0, trim_size)
    gt_boxes[:, :4].clamp_(0, trim_size)
    im_info[0, 0] = trim_size
    im_info[0, 1] = trim_size
```

Code3将调整好的图片和gt_boxes填充成满足ratio的图片，并更新im_info

```
Code4:
# check the bounding box:
a = gt_boxes[:, 0].numpy()
b = gt_boxes[:, 1].numpy()
#print(a.dtype, b.dtype)
for i in range(len(a)):
    if a[i] > 20000:
        a[i] = float(0)
gt_boxes[:, 0] = torch.from_numpy(a)
for i in range(len(b)):
    if b[i] > 20000:
        b[i] = float(0)
gt_boxes[:, 1] = torch.from_numpy(b)
not_keep = (gt_boxes[:,0] == gt_boxes[:,2]) | (gt_boxes[:,1] == gt_boxes[:,3])
```

Code4检查是否有边界问题，我直接跑源码时loss全是NAN，最后发现这一块0值变成了20000多，可能是边界判断出了问题。

```
Code5:
keep = torch.nonzero(not_keep == 0).view(-1)

gt_boxes_padding = torch.FloatTensor(self.max_num_box,gt_boxes.size(1)).zero_()
if keep.numel() != 0:
    gt_boxes = gt_boxes[keep]
    num_boxes = min(gt_boxes.size(0), self.max_num_box)
    gt_boxes_padding[:num_boxes,:] = gt_boxes[:num_boxes]
else:
    num_boxes = 0

    # permute trim_data to adapt to downstream processing
padding_data = padding_data.permute(2, 0, 1).contiguous()
im_info = im_info.view(3)
```

Code5保存没有异常的修改图片，并且对图片数据进行转置以适应后续代码的使用

resnet.py

```
self.RCNN_base = nn.Sequential(resnet.conv1, resnet.bn1,resnet.relu,
  resnet.maxpool,resnet.layer1,resnet.layer2,resnet.layer3)

self.RCNN_top = nn.Sequential(resnet.layer4)

self.RCNN_cls_score = nn.Linear(2048, self.n_classes)
if self.class_agnostic:
  self.RCNN_bbox_pred = nn.Linear(2048, 4)
else:
  self.RCNN_bbox_pred = nn.Linear(2048, 4 * self.n_classes)


def _head_to_tail(self, pool5):
    fc7 = self.RCNN_top(pool5).mean(3).mean(2)
    return fc7
```

构建RCNN_base，RCNN_top，RCNN_cls_score，RCNN_bbox_pred模块

```
# Fix blocks
for p in self.RCNN_base[0].parameters(): p.requires_grad=False
for p in self.RCNN_base[1].parameters(): p.requires_grad=False

assert (0 <= cfg.RESNET.FIXED_BLOCKS < 4)
if cfg.RESNET.FIXED_BLOCKS >= 3:
  for p in self.RCNN_base[6].parameters(): p.requires_grad=False
if cfg.RESNET.FIXED_BLOCKS >= 2:
  for p in self.RCNN_base[5].parameters(): p.requires_grad=False
if cfg.RESNET.FIXED_BLOCKS >= 1:
  for p in self.RCNN_base[4].parameters(): p.requires_grad=False

def set_bn_fix(m):
  classname = m.__class__.__name__
  if classname.find('BatchNorm') != -1:
    for p in m.parameters(): p.requires_grad=False

self.RCNN_base.apply(set_bn_fix)
self.RCNN_top.apply(set_bn_fix)
```

固定指定层的参数

faster_rcnn.py

输入为data，im_fo，gt_boxes，num_boxes（由roibatchLoader.py产生）

data：padding后的图片

im_fo：padding后的图片尺寸和目标尺寸和图片的比例(h,w,scale)

gt_boxes：padding后的gt_boxes

num_boxes：一张图的gt_boxes数量

返回值为rois，cls_prob，bbox_pred，rpn_loss_cls，rpn_loss_bbox，RCNN_loss_cls，RCNN_loss_bbox，rois_label

rois：train：sample的感兴趣区域 test：感兴趣区域

cls_prob：各类别的概率值

bbox_pred：bbox的预测值

rpn_loss_cls：rpn产生的类别loss

rpn_loss_bbox：rpn产生的bbox的loss

RCNN_loss_cls：RCNN产生的类别loss

RCNN_loss_bbox：RCNN产生的bbox的loss

rois_label：RPN产生的rois的分配label

```
def forward(self, im_data, im_info, gt_boxes, num_boxes):
    batch_size = im_data.size(0)

    im_info = im_info.data
    gt_boxes = gt_boxes.data
    num_boxes = num_boxes.data

    # feed image data to base model to obtain base feature map
    base_feat = self.RCNN_base(im_data)

    # feed base feature map tp RPN to obtain rois
    rois, rpn_loss_cls, rpn_loss_bbox = self.RCNN_rpn(base_feat, im_info, gt_boxes,
num_boxes)

    # if it is training phrase, then use ground trubut bboxes for refining
    if self.training:
        roi_data = self.RCNN_proposal_target(rois, gt_boxes, num_boxes)
        rois, rois_label, rois_target, rois_inside_ws, rois_outside_ws = roi_data

        rois_label = Variable(rois_label.view(-1).long())
        rois_target = Variable(rois_target.view(-1, rois_target.size(2)))
        rois_inside_ws = Variable(rois_inside_ws.view(-1, rois_inside_ws.size(2)))
        rois_outside_ws = Variable(rois_outside_ws.view(-1, rois_outside_ws.size(2)))
    else:
        rois_label = None
        rois_target = None
        rois_inside_ws = None
        rois_outside_ws = None
        rpn_loss_cls = 0
        rpn_loss_bbox = 0

    rois = Variable(rois)
    # do roi pooling based on predicted rois

    if cfg.POOLING_MODE == 'crop':
```

```
        # pdb.set_trace()
        # pooled_feat_anchor = _crop_pool_layer(base_feat, rois.view(-1, 5))
        grid_xy = _affine_grid_gen(rois.view(-1, 5), base_feat.size()[2:],
self.grid_size)
        grid_yx = torch.stack([grid_xy.data[:,:,:,1], grid_xy.data[:,:,:,0]],
3).contiguous()
        pooled_feat = self.RCNN_roi_crop(base_feat, Variable(grid_yx).detach())
        if cfg.CROP_RESIZE_WITH_MAX_POOL:
            pooled_feat = F.max_pool2d(pooled_feat, 2, 2)
    elif cfg.POOLING_MODE == 'align':
        pooled_feat = self.RCNN_roi_align(base_feat, rois.view(-1, 5))
    elif cfg.POOLING_MODE == 'pool':
        pooled_feat = self.RCNN_roi_pool(base_feat, rois.view(-1,5))

    # feed pooled features to top model
    pooled_feat = self._head_to_tail(pooled_feat)

    # compute bbox offset
    bbox_pred = self.RCNN_bbox_pred(pooled_feat)
    if self.training and not self.class_agnostic:
        # select the corresponding columns according to roi labels
        bbox_pred_view = bbox_pred.view(bbox_pred.size(0), int(bbox_pred.size(1) / 4),
4)
        bbox_pred_select = torch.gather(bbox_pred_view, 1,
rois_label.view(rois_label.size(0), 1, 1).expand(rois_label.size(0), 1, 4))
        bbox_pred = bbox_pred_select.squeeze(1)

    # compute object classification probability
    cls_score = self.RCNN_cls_score(pooled_feat)
    cls_prob = F.softmax(cls_score, 1)

    RCNN_loss_cls = 0
    RCNN_loss_bbox = 0

    if self.training:
        # classification loss
        RCNN_loss_cls = F.cross_entropy(cls_score, rois_label)

        # bounding box regression L1 loss
        RCNN_loss_bbox = _smooth_l1_loss(bbox_pred, rois_target, rois_inside_ws,
rois_outside_ws)


    cls_prob = cls_prob.view(batch_size, rois.size(1), -1)
    bbox_pred = bbox_pred.view(batch_size, rois.size(1), -1)

    return rois, cls_prob, bbox_pred, rpn_loss_cls, rpn_loss_bbox, RCNN_loss_cls,
RCNN_loss_bbox, rois_label
```

搭建前向传播过程

roibatchLoader-->RCNN_base-->RCNN_rpn-->RCNN_proposal_target-->RCNN_roi_crop-->RCNN_top-->(RCNN_cls_score,RCNN_bbox_pred)

base_feat = self.RCNN_base(im_data)：RCNN_base前传得到base_feature

rois, rpn_loss_cls, rpn_loss_bbox = self.RCNN_rpn(base_feat, im_info, gt_boxes, num_boxes)：RCNN_rpn前传得到感兴趣区域和对应的loss值

roi_data = self.RCNN_proposal_target(rois, gt_boxes, num_boxes)：RCNN_proposal_target前传从提议区域中筛选出提议目标

pooled_feat = self.RCNN_roi_crop(base_feat, Variable(grid_yx).detach())：RCNN_roi_crop作为RoIPooling，产生pooled_feature

pooled_feat = self._head_to_tail(pooled_feat)：RCNN_top前传产生最终的feature

bbox_pred = self.RCNN_bbox_pred(pooled_feat)：RCNN_bbox_pred产生预测bbox

cls_score = self.RCNN_cls_score(pooled_feat)：RCNN_cls_score产生分类分数

rpn.py

```
self.din = din  # get depth of input feature map, e.g., 1024
self.anchor_scales = cfg.ANCHOR_SCALES
self.anchor_ratios = cfg.ANCHOR_RATIOS
self.feat_stride = cfg.FEAT_STRIDE[0]

# define the convrelu layers processing input feature map
self.RPN_Conv = nn.Conv2d(self.din, 512, 3, 1, 1, bias=True)

# define bg/fg classifcation score layer
self.nc_score_out = len(self.anchor_scales) * len(self.anchor_ratios) * 2 # 2(bg/fg) *
9 (anchors)
self.RPN_cls_score = nn.Conv2d(512, self.nc_score_out, 1, 1, 0)

# define anchor box offset prediction layer
self.nc_bbox_out = len(self.anchor_scales) * len(self.anchor_ratios) * 4 # 4(coords) *
9 (anchors)
self.RPN_bbox_pred = nn.Conv2d(512, self.nc_bbox_out, 1, 1, 0)

# define proposal layer
self.RPN_proposal = _ProposalLayer(self.feat_stride, self.anchor_scales,
self.anchor_ratios)

# define anchor target layer
self.RPN_anchor_target = _AnchorTargetLayer(self.feat_stride, self.anchor_scales,
self.anchor_ratios)
```

RCNN_base产生的feature map的depth为1024，初始化构建卷积层

```
# return feature map after convrelu layer
rpn_conv1 = F.relu(self.RPN_Conv(base_feat), inplace=True)

# get rpn classification score
rpn_cls_score = self.RPN_cls_score(rpn_conv1)
rpn_cls_score_reshape = self.reshape(rpn_cls_score, 2)
rpn_cls_prob_reshape = F.softmax(rpn_cls_score_reshape, 1)
rpn_cls_prob = self.reshape(rpn_cls_prob_reshape, self.nc_score_out)

# get rpn offsets to the anchor boxes
rpn_bbox_pred = self.RPN_bbox_pred(rpn_conv1)
```

搭建RPN前向传播

```
rois = self.RPN_proposal((rpn_cls_prob.data, rpn_bbox_pred.data, im_info, cfg_key))
```

产生RPN提议区域

```
rpn_data = self.RPN_anchor_target((rpn_cls_score.data, gt_boxes, im_info, num_boxes))
```

产生anchor标签

```
self.rpn_loss_cls = F.cross_entropy(rpn_cls_score, rpn_label)
```

```
self.rpn_loss_box = _smooth_l1_loss(rpn_bbox_pred, rpn_bbox_targets,
rpn_bbox_inside_weights, rpn_bbox_outside_weights, sigma=3, dim=[1,2,3])
```

计算rpn的loss

proposal_layer.py（inference）

Outputs object detection proposals by applying estimated bounding-box transformations to a set of regular boxes (called "anchors").

```
feat_height, feat_width = scores.size(2), scores.size(3)
shift_x = np.arange(0, feat_width) * self._feat_stride
shift_y = np.arange(0, feat_height) * self._feat_stride
shift_x, shift_y = np.meshgrid(shift_x, shift_y)
shifts = torch.from_numpy(np.vstack((shift_x.ravel(), shift_y.ravel(),
                          shift_x.ravel(), shift_y.ravel())).transpose())
shifts = shifts.contiguous().type_as(scores).float()
```

将feature map的每个像素投影到原图上

```
A = self._num_anchors
K = shifts.size(0)

self._anchors = self._anchors.type_as(scores)
anchors = self._anchors.view(1, A, 4) + shifts.view(K, 1, 4)
anchors = anchors.view(1, K * A, 4).expand(batch_size, K * A, 4)
```

Tensor的shape是（tensor，1）和（1,tensor）这是可以相加的，会自动扩充。

per_anchor:torch.Size([1, 9, 4]) shifts:torch.Size([1900, 1, 4]) tot_anchor:torch.Size([1900, 9, 4])

构建每个投影到原图像素点的anchor

```
# Transpose and reshape predicted bbox transformations to get them
# into the same order as the anchors:

bbox_deltas = bbox_deltas.permute(0, 2, 3, 1).contiguous()
bbox_deltas = bbox_deltas.view(batch_size, -1, 4)
#torch.size(batch_size, 38*50*9, 4)

# Same story for the scores:
scores = scores.permute(0, 2, 3, 1).contiguous()
scores = scores.view(batch_size, -1)
#torch.size(batch_size, 38*50*9*2)
```

将预测bbox和scores储存成和anchors相同的形式

```
#apply predicted bbox deltas at cell i to each of the A anchors
# clip predicted boxes to image
```

1. Convert anchors into proposals via bbox transformations proposals = bbox_transform_inv(anchors, bbox_deltas, batch_size)

```
# 2. clip predicted boxes to image
proposals = clip_boxes(proposals, im_info, batch_size)

scores_keep = scores
proposals_keep = proposals
_, order = torch.sort(scores_keep, 1, True)

#定义输出形式
output = scores.new(batch_size, post_nms_topN, 5).zero_()
for i in range(batch_size):
    # # 3. remove predicted boxes with either height or width < threshold
    # # (NOTE: convert min_size to input image scale stored in im_info[2])
    proposals_single = proposals_keep[i]
    scores_single = scores_keep[i]

    # # 4. sort all (proposal, score) pairs by score from highest to lowest
    # # 5. take top pre_nms_topN (e.g. 6000)
```

```
    order_single = order[i]

    if pre_nms_topN > 0 and pre_nms_topN < scores_keep.numel():
        order_single = order_single[:pre_nms_topN]

    proposals_single = proposals_single[order_single, :]
    scores_single = scores_single[order_single].view(-1,1)

    # 6. apply nms (e.g. threshold = 0.7)
    # 7. take after_nms_topN (e.g. 300)
    # 8. return the top proposals (-> RoIs top)

    keep_idx_i = nms(torch.cat((proposals_single, scores_single), 1), nms_thresh,
force_cpu=not cfg.USE_GPU_NMS)
    keep_idx_i = keep_idx_i.long().view(-1)

    if post_nms_topN > 0:
        keep_idx_i = keep_idx_i[:post_nms_topN]
    proposals_single = proposals_single[keep_idx_i, :]
    scores_single = scores_single[keep_idx_i, :]

    # padding 0 at the end.
    num_proposal = proposals_single.size(0)
    output[i,:,0] = i #来自第i个batchsize
    output[i,:num_proposal,1:] = proposals_single #保存提议区域
```

generate_anchors.py

产生3x3种anchor，anchor的大小是相对于resize后的图片来说的

9个anchor对应于3种**scales**（**面积**分别为1282，2562，5122）和3种**aspect ratios**(**宽高比**分别为1:1, 1:2, 2:1)。

[[-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.]]

bbox_transform.py

```
def bbox_transform_inv(boxes, deltas, batch_size):
    #anchor{x,y,w,h}
    widths = boxes[:, :, 2] - boxes[:, :, 0] + 1.0
```

```
    heights = boxes[:, :, 3] - boxes[:, :, 1] + 1.0
    ctr_x = boxes[:, :, 0] + 0.5 * widths
    ctr_y = boxes[:, :, 1] + 0.5 * heights
    #pred offset
    dx = deltas[:, :, 0::4]
    dy = deltas[:, :, 1::4]
    dw = deltas[:, :, 2::4]
    dh = deltas[:, :, 3::4]
    #pred {x,y,w,h}
    pred_ctr_x = dx * widths.unsqueeze(2) + ctr_x.unsqueeze(2)
    pred_ctr_y = dy * heights.unsqueeze(2) + ctr_y.unsqueeze(2)
    pred_w = torch.exp(dw) * widths.unsqueeze(2)
    pred_h = torch.exp(dh) * heights.unsqueeze(2)

    pred_boxes = deltas.clone()
    # x1
    pred_boxes[:, :, 0::4] = pred_ctr_x - 0.5 * pred_w
    # y1
    pred_boxes[:, :, 1::4] = pred_ctr_y - 0.5 * pred_h
    # x2
    pred_boxes[:, :, 2::4] = pred_ctr_x + 0.5 * pred_w
    # y2
    pred_boxes[:, :, 3::4] = pred_ctr_y + 0.5 * pred_h

    return pred_boxes
```

$$
\begin{aligned}
t_{\mathrm{x}} &= (x - x_{\mathrm{a}})/w_{\mathrm{a}}, \quad t_{\mathrm{y}} = (y - y_{\mathrm{a}})/h_{\mathrm{a}}, \\
t_{\mathrm{w}} &= \log(w/w_{\mathrm{a}}), \quad t_{\mathrm{h}} = \log(h/h_{\mathrm{a}}), \\
t_{\mathrm{x}}^{*} &= (x^{*} - x_{\mathrm{a}})/w_{\mathrm{a}}, \quad t_{\mathrm{y}}^{*} = (y^{*} - y_{\mathrm{a}})/h_{\mathrm{a}}, \\
t_{\mathrm{w}}^{*} &= \log(w^{*}/w_{\mathrm{a}}), \quad t_{\mathrm{h}}^{*} = \log(h^{*}/h_{\mathrm{a}}),
\end{aligned}
$$

由于训练过程是将预测对anchor的偏移和gt对anchor的偏移进行回归的，所以预测值可以由anchor值转化而来

nms_cpu.py

```
def nms_cpu(dets, thresh):
    dets = dets.numpy()
    x1 = dets[:, 0]
    y1 = dets[:, 1]
    x2 = dets[:, 2]
    y2 = dets[:, 3]
    scores = dets[:, 4]
    #每一个检测框的面积
    areas = (x2 - x1 + 1) * (y2 - y1 + 1)
    #按照score置信度降序排序
    order = scores.argsort()[::-1]

    keep = [] #保留的结果框集合
    while order.size > 0:
```

```
        i = order.item(0)
        keep.append(i) #保留该类剩余box中得分最高的一个
        #得到相交区域,左上及右下
        xx1 = np.maximum(x1[i], x1[order[1:]])
        yy1 = np.maximum(y1[i], y1[order[1:]])
        xx2 = np.minimum(x2[i], x2[order[1:]])
        yy2 = np.minimum(y2[i], y2[order[1:]])
        #计算相交的面积,不重叠时面积为0
        w = np.maximum(0.0, xx2 - xx1 + 1)
        h = np.maximum(0.0, yy2 - yy1 + 1)
        inter = w * h
        #计算IoU：重叠面积 /（面积1+面积2-重叠面积）
        ovr = inter / (areas[i] + areas[order[1:]] - inter)
        #保留IoU小于阈值的box
        inds = np.where(ovr <= thresh)[0]
        order = order[inds + 1] #因为ovr数组的长度比order数组少一个,所以这里要将所有下标后移一位

    return torch.IntTensor(keep)
```

anchor_target_layer.py (train)

Assign anchors to ground-truth targets. Produces anchor classification labels and bounding-box regression targets.

输入rpn_cls_score，gt_boxes，im_info，num_boxes

rpn_cls_score：rpn得到的类别分数

```
# inds_inside表示anchor的4个点坐标都在图像内部的anchor的index
keep = ((all_anchors[:, 0] >= -self._allowed_border) &
        (all_anchors[:, 1] >= -self._allowed_border) &
        (all_anchors[:, 2] < long(im_info[0][1]) + self._allowed_border) &
        (all_anchors[:, 3] < long(im_info[0][0]) + self._allowed_border))

inds_inside = torch.nonzero(keep).view(-1)

# keep only inside anchors
#将不完全在图像内部（初始化的anchor的4个坐标点超出图像边界）的anchor都过滤掉，
# 一般过滤后只会有原来1/3左右的anchor。如果不将这部分anchor过滤，则会使训练过程难以收敛。
anchors = all_anchors[inds_inside, :]

# label: 1 is positive, 0 is negative, -1 is dont care
# 前面得到的只是anchor的4个坐标信息，接下来就要为每个anchor分配标签了，
# 初始化的时候标签都用-1来填充，-1表示无效，这类标签的数据不会对梯度更新起到帮助。
labels = gt_boxes.new(batch_size, inds_inside.size(0)).fill_(-1)
bbox_inside_weights = gt_boxes.new(batch_size, inds_inside.size(0)).zero_()
bbox_outside_weights = gt_boxes.new(batch_size, inds_inside.size(0)).zero_()

overlaps = bbox_overlaps_batch(anchors, gt_boxes)
# 每一行表示anchor，每一列表示object。 argmax_overlaps是计算每个anchor和哪个object的IOU最大，
# 维度是n*1，值是object的index。max_overlaps是具体的IOU值。
max_overlaps, argmax_overlaps = torch.max(overlaps, 2)
# gt_max_overlaps是具体的IOU值,是计算每个object和哪个anchor的IOU最大，维度是k*1
```

```
gt_max_overlaps, _ = torch.max(overlaps, 1)
# 这个条件语句默认是执行的，目的是将IOU小于某个阈值的anchor的标签都标为0，也就是背景类。
# 阈值config.TRAIN.RPN_NEGATIVE_OVERLAP默认是0.3。
# 如果某个anchor和所有object的IOU的最大值比这个阈值小，那么就是背景。
if not cfg.TRAIN.RPN_CLOBBER_POSITIVES:
    labels[max_overlaps < cfg.TRAIN.RPN_NEGATIVE_OVERLAP] = 0
#因为如果有多个anchor和某个object的IOU值都是最大且一样，
# 所以需要overlaps.eq(gt_max_overlaps.view(batch_size,1,-1)将IOU最大的那些anchor都捞出来。
gt_max_overlaps[gt_max_overlaps==0] = 1e-5
keep = torch.sum(overlaps.eq(gt_max_overlaps.view(batch_size,1,-1) \
.expand_as(overlaps)), 2)
# 有两种类型的anhor其标签是1，标签1表示foreground，也就是包含object。
# 第一种是和任意一个object有最大IOU的anchor，也就是前面得到的gt_argmax_overlaps。
if torch.sum(keep) > 0:
    labels[keep>0] = 1

# fg label: above threshold IOU
# 第二种是和所有object的IOU的最大值超过某个阈值的anchor，
# 其中阈值config.TRAIN.RPN_POSITIVE_OVERLAP默认是0.7。
labels[max_overlaps >= cfg.TRAIN.RPN_POSITIVE_OVERLAP] = 1
# 这一部分是和前面if not config.TRAIN.RPN_CLOBBER_POSITIVES条件语句互斥的，
# 区别在于背景类anchor的标签定义先后顺序不同，这主要涉及到标签1和标签0之间的覆盖。
if cfg.TRAIN.RPN_CLOBBER_POSITIVES:
    labels[max_overlaps < cfg.TRAIN.RPN_NEGATIVE_OVERLAP] = 0
```

if not cfg.TRAIN.RPN_CLOBBER_POSITIVES和if cfg.TRAIN.RPN_CLOBBER_POSITIVES决定了标签1和标签0覆盖的先后顺序

if not cfg.TRAIN.RPN_CLOBBER_POSITIVES执行时，先将与object重叠小的anchor标记为背景类，那么labels[keep>0] = 1就不会对IOU小的anchor标记为前景类

if cfg.TRAIN.RPN_CLOBBER_POSITIVES执行时，labels[keep>0] = 1将一些IOU小的anchor标记为前景类，然后将没有标记为前景类且IOU小的anchor标记为背景类

```
num_fg = int(cfg.TRAIN.RPN_FG_FRACTION * cfg.TRAIN.RPN_BATCHSIZE) #0.5*256

sum_fg = torch.sum((labels == 1).int(), 1)
sum_bg = torch.sum((labels == 0).int(), 1)

for i in range(batch_size):
    # subsample positive labels if we have too many
    if sum_fg[i] > num_fg:
        fg_inds = torch.nonzero(labels[i] == 1).view(-1)
        # torch.randperm seems has a bug on multi-gpu setting that cause the segfault.
        # See https://github.com/pytorch/pytorch/issues/1868 for more details.
        # use numpy instead.
        #rand_num = torch.randperm(fg_inds.size(0)).type_as(gt_boxes).long()
        rand_num =
torch.from_numpy(np.random.permutation(fg_inds.size(0))).type_as(gt_boxes).long()
        disable_inds = fg_inds[rand_num[:fg_inds.size(0)-num_fg]]
        labels[i][disable_inds] = -1

#           num_bg = cfg.TRAIN.RPN_BATCHSIZE - sum_fg[i]
```

```
    num_bg = cfg.TRAIN.RPN_BATCHSIZE - torch.sum((labels == 1).int(), 1)[i]

    # subsample negative labels if we have too many
    if sum_bg[i] > num_bg:
        bg_inds = torch.nonzero(labels[i] == 0).view(-1)
        #rand_num = torch.randperm(bg_inds.size(0)).type_as(gt_boxes).long()

        rand_num =
torch.from_numpy(np.random.permutation(bg_inds.size(0))).type_as(gt_boxes).long()
        disable_inds = bg_inds[rand_num[:bg_inds.size(0)-num_bg]]
        labels[i][disable_inds] = -1

offset = torch.arange(0, batch_size)*gt_boxes.size(1)
# bbox_target是每个bbox回归的ground truth，初始化为len(inds_inside)*4大小的numpy array,
# 所以包含了标签为1,0和-1三种类型的bbox。bbox_transform函数用来生成bbox_targets,
# 输入中gt_boxes原本是k*5的numpy array，k表示有几个object,
# 这里通过gt_boxes[argmax_overlaps, :4]扩增并取前4列值,
# 因为argmax_overlaps是和anchor的IOU最大的object的index，所以这种写法相当于复制
# 指定index的object的gt_boxes信息，因此某个anchor的坐标回归目标利用的就是和
# 该anchor的IOU最大的object的坐标通过一定公式转换后的信息
argmax_overlaps = argmax_overlaps + offset.view(batch_size, 1).type_as(argmax_overlaps)
bbox_targets = _compute_targets_batch(anchors, gt_boxes.view(-1,5)
[argmax_overlaps.view(-1), :].view(batch_size, -1, 5))

# use a single value instead of 4 values for easy index.
# bbox_weights变量是后续用来指定哪些anchor用于梯度更新的0,1矩阵，相当于一个mask,
# 只有标签是1的bbox的weight才有值，值是config.TRAIN.RPN_BBOX_WEIGHTS,
# 该变量默认4个值都是1。因此标签是0或-1的weight都是0。
#bbox_inside_weights 它实际上就是控制回归的对象的，只有真正是前景的对象才会被回归。
bbox_inside_weights[labels==1] = cfg.TRAIN.RPN_BBOX_INSIDE_WEIGHTS[0]

#bbox_outside_weights 也是用1/N1, 2/N0 初始化,对前景和背景控制权重，比起上面多了一个背景的权重，从
#第二步来看， positive_weights ，negative_weights有互补的意味。
if cfg.TRAIN.RPN_POSITIVE_WEIGHT < 0:
    num_examples = torch.sum(labels[i] >= 0)
    positive_weights = 1.0 / num_examples.item()
    negative_weights = 1.0 / num_examples.item()
else:
    assert ((cfg.TRAIN.RPN_POSITIVE_WEIGHT > 0) &
            (cfg.TRAIN.RPN_POSITIVE_WEIGHT < 1))

bbox_outside_weights[labels == 1] = positive_weights
bbox_outside_weights[labels == 0] = negative_weights

#接下来的4行代码则是将labels、bbox_targets和bbox_weights这4个变量重新映射回过滤之前的bbox。
# 所以假设RPN网络的输入feature map大小是38*50，那么最终这3个变量的第一个维度就是9*38*50=17100,
# 也就是最原始的anchor数量。当然，被过滤掉的bbox的label都是-1，bbox_targets都是0,
# bbox_weights都是0，因此对于RPN网络的训练而言没有帮助。
labels = _unmap(labels, total_anchors, inds_inside, batch_size, fill=-1)
bbox_targets = _unmap(bbox_targets, total_anchors, inds_inside, batch_size, fill=0)
bbox_inside_weights = _unmap(bbox_inside_weights, total_anchors, inds_inside,
batch_size, fill=0)
```

```
bbox_outside_weights = _unmap(bbox_outside_weights, total_anchors, inds_inside,
batch_size, fill=0)
#最后几个输出的情况：labels的维度是1*17100,
# bbox_targets的维度是1*36*38*50，bbox_inside_weights,bbox_outside_weights的维度是
1*36*38*50。
outputs = []

labels = labels.view(batch_size, height, width, A).permute(0,3,1,2).contiguous()
labels = labels.view(batch_size, 1, A * height, width)
outputs.append(labels)

bbox_targets = bbox_targets.view(batch_size, height, width,
A*4).permute(0,3,1,2).contiguous()
outputs.append(bbox_targets)

anchors_count = bbox_inside_weights.size(1)
bbox_inside_weights =
bbox_inside_weights.view(batch_size,anchors_count,1).expand(batch_size, anchors_count,
4)

bbox_inside_weights = bbox_inside_weights.contiguous().view(batch_size, height, width,
4*A)\
                    .permute(0,3,1,2).contiguous()

outputs.append(bbox_inside_weights)

bbox_outside_weights =
bbox_outside_weights.view(batch_size,anchors_count,1).expand(batch_size, anchors_count,
4)
bbox_outside_weights = bbox_outside_weights.contiguous().view(batch_size, height,
width, 4*A)\
                    .permute(0,3,1,2).contiguous()
outputs.append(bbox_outside_weights)

return outputs
```

RPN的过程就是筛选出与gt_boxes的IOU高的anchor，当作gt_boxes，训练RPN就是将推理得到的分类和bbox跟满足要求的anchor进行回归和分类，（训练得到的权重使得推理筛选出来的proposal更加能够概括gt）将推理筛选出的rois作为预先选定的区域，第二阶段进行微调回归框和分类。

proposal_target_layer_cascade.py（sample）

输入为rois，gt_boxes，num_boxes

rois：所有感兴趣的区域

输出为rois，labels，bbox_target，bbox_inside_weights，bbox_outside_weights

rois：sample后的感兴趣区域

labels：sample后的rois的label

bbox_target：sample后rois的回归目标t×

bbox_inside_weights，bbox_outside_weights：bbox的权重值

```
if fg_num_rois > 0 and bg_num_rois > 0:
    # sampling fg
    fg_rois_per_this_image = min(fg_rois_per_image, fg_num_rois)

    # torch.randperm seems has a bug on multi-gpu setting that cause the segfault.
    # See https://github.com/pytorch/pytorch/issues/1868 for more details.
    # use numpy instead.
    #rand_num = torch.randperm(fg_num_rois).long().cuda()
    rand_num =
torch.from_numpy(np.random.permutation(fg_num_rois)).type_as(gt_boxes).long()
    fg_inds = fg_inds[rand_num[:fg_rois_per_this_image]]

    # sampling bg
    bg_rois_per_this_image = rois_per_image - fg_rois_per_this_image

    # Seems torch.rand has a bug, it will generate very large number and make an error.
    # We use numpy rand instead.
    #rand_num = (torch.rand(bg_rois_per_this_image) * bg_num_rois).long().cuda()
    rand_num = np.floor(np.random.rand(bg_rois_per_this_image) * bg_num_rois)
    rand_num = torch.from_numpy(rand_num).type_as(gt_boxes).long()
    bg_inds = bg_inds[rand_num]

elif fg_num_rois > 0 and bg_num_rois == 0:
    # sampling fg
    #rand_num = torch.floor(torch.rand(rois_per_image) * fg_num_rois).long().cuda()
    rand_num = np.floor(np.random.rand(rois_per_image) * fg_num_rois)
    rand_num = torch.from_numpy(rand_num).type_as(gt_boxes).long()
    fg_inds = fg_inds[rand_num]
    fg_rois_per_this_image = rois_per_image
    bg_rois_per_this_image = 0
elif bg_num_rois > 0 and fg_num_rois == 0:
    # sampling bg
    #rand_num = torch.floor(torch.rand(rois_per_image) * bg_num_rois).long().cuda()
    rand_num = np.floor(np.random.rand(rois_per_image) * bg_num_rois)
    rand_num = torch.from_numpy(rand_num).type_as(gt_boxes).long()

    bg_inds = bg_inds[rand_num]
    bg_rois_per_this_image = rois_per_image
    fg_rois_per_this_image = 0
else:
    raise ValueError("bg_num_rois = 0 and fg_num_rois = 0, this should not happen!")
```

在rois中sample128个，用于训练

RoICrop

-src文件夹下是c和cuda版本的源码，其中roi_pooling的操作的foward是c和cuda版本都有的，而backward仅写了cuda版本的代码。 **-functions**文件夹下的roi_pool.py是继承了torch.autograd.Function类，实现RoI层的foward和backward函数。 **-modules**文件夹下的roi_pool.py是继承了torch.nn.Modules类，实现了对RoI层的封装，此时RoI层就跟ReLU层一样的使用了。 **-_ext**文件夹下还有个roi_pooling文件夹，这个文件夹是存储src中c，cuda编译过后的文件的，编译过后就可以被funcitons中的roi_pool.py调用了。

_smooth_l1_loss

```
def _smooth_l1_loss(bbox_pred, bbox_targets, bbox_inside_weights, bbox_outside_weights,
sigma=1.0, dim=[1]):
    sigma_2 = sigma ** 2
    box_diff = bbox_pred - bbox_targets
    in_box_diff = bbox_inside_weights * box_diff
    abs_in_box_diff = torch.abs(in_box_diff)
    smoothL1_sign = (abs_in_box_diff < 1. / sigma_2).detach().float()
    in_loss_box = torch.pow(in_box_diff, 2) * (sigma_2 / 2.) * smoothL1_sign \
                  + (abs_in_box_diff - (0.5 / sigma_2)) * (1. - smoothL1_sign)
    out_loss_box = bbox_outside_weights * in_loss_box
    loss_box = out_loss_box
    for i in sorted(dim, reverse=True):
      loss_box = loss_box.sum(i)
    loss_box = loss_box.mean()
    return loss_box
```

输入为预测框，目标框，win，wout

$$output = w_{out} * Smooth_{l1}(x_{new})$$
$$Smooth_{l1}(x) = 0.5 * (\sigma * x)^2 \text{ or } Smooth_{l1}(x) = |x| - 0.5/\sigma^2$$
$$x_{new} = x_{old} * w_{in}$$

xold = box_diff

xnew = in_box_diff

smoothl1 = in_loss_box

output = out_loss_box

可以看出，bbox_outside_weights 就是为了平衡 box_loss,cls_loss 的，因为二个loss差距过大，所以它被设置为 1/N 的权重。