Faster rcnn caffe 代码详解

符号说明：GT：ground truth

1.模型的流程：

- 单张图片进来先 resize
- resize 过后的图片进入基础网络得到最后一层的特征层：
  conv5([1,h,w,256])
- conv5 进入 RPN，通过 1*1 的卷积得到 anchors 的分类和回归的预测
  分类的输出 shape：[1,h,w,2*9]，2 表示前景和背景，9 表示 anchor 数目
  回归的输出 shape：[1,h,w,4*9],4 为坐标，9 为 anchor 数目
- 在 rpn.anchor_target_layer 层中，制作 RPN 层的标签。
  （训练时才有）
  1. 生成 anchors([K,A,4],K=w*h,A=9) → reshape 到[K*A,4]
     → 去除在图片之外的 anchors ，这时剩余的 anchor 为[N,4]
  2. 为 anchor 打分类标签(shape 为[N,1])，规则为：
     a.与 GT 的 IOU 最大的几个 anchor 标为 1(前景)。
     b.与 GT 的 IOU 最大值在 0.7~1 之间的标为 1。
     c.与 GT 的 IOU 最大值在 0~0.3 的标记为 0(背景)。
     d.没有被标记的 anchor 标记为-1(不关心)。
     e.如果前景数目超过 128,则将多余的 anchor 标记为-1
     f.如果背景数目超过 256-num(前景)，则将多余的标为-1
  3. 为 anchor 打回归标签（shape 为[N,4]）：
     计算在图像内的 anchor（[N,4]）到与之 IOU 最大的 GT 的偏差。
  4. 计算 inside_weight(shape 为[N,4])：
     只有前景的 anchor 才会计算 anchor 回归的 loss，所以这个 weight 就是为了标记前景样本。
  5. 计算 outside_weight(shape 为[N,4]):
     outside_weight 是前景和背景的一个加权。
  6. 将打好的标签和 weight 映射到[K*A,~]空间。
  7. 把[K*A,~]reshape 到对应的输出 shape。
- 计算 RPN 层的 loss
- 生成 proposal
  1. 将之前的 rpn_bbox_pred（shape 为[1,h,w,4*9]）reshape 到
     [h*w*9,4]→ 用其调整 anchors 得到的结果称为 proposal
     (shape 为[h*w*9,4])

2. 将 proposal 裁剪到图像大小之内
3. 根据之前的 rpn_cls_score(shape 为[1,h,w,2*9])，取前景得分 score（shape 为[1,h,w,9]）→ reshape 到[h*w*9,1]。
4. 去除尺寸（长，宽）小于阈值的 proposals，剩下的 shape 为[N1,4], 和其前景得分的 shape[N1,1]。
5. 根据 proposal 的前景得分排序，选取出前 6000 个 proposal，这时的 proposal.shape=[6000,4],score.shape=[6000,1]
6. 对剩下的 6000 个框进行 NMS
7. 剩下的框，根据得分选取前 2000 个框作为 RPN 层的输出。输出 2000 个框的位置信息，即输出的 shape 为[2000,4]

- rpn.proposal_target_layer 层。根据 RPN 层的产生的 2000 个候选框选出 128 个 ROI。制作其分类和回归的标签，还有两个回归的权重（inside_weight,outside_weight）
  1. 计算 proposals 与 GT 的 IOU，IOU 大于 0.5 的为前景，小于 0.5 大于 0.1 的为背景。前景不超过 32 个，背景不超过 128-#前景。
  2. 根据选出的前景背景的 index，为其打上分类标签，背景打为 0,前景为与其最大 IOU 的 GT 的标签。labels.shape = [128,1]
  3. 根据选出的前景背景的 index,索引出其坐标，再与其最接近的 GT 算出 bbox_target，bbox_target.shape=[128,4]。
  4. 将 bbox_target 做成网络预测的 shape，并生成 inside_weight。输入为 labels 和 bbox_target。输出的 shape[128,4*num_class], inside_weight 的 shape 也为[128,4*num_class]，是前景的一个 mask。
  5. outside_weight 与 inside_weight 相同。
- ROIPooling: 用上一层产生的 rois，和基础网络出来的 conv5 特征，计算 rois 的特征。即使 roi 的大小会不一样，但是经过这层之后的特征都是一样的了
- 两个全连接层
- 全连接之后，接一个全连接产生分类预测，接另外一个全连接层产生回归预测。
- 用之前做的 roi_bbox_target 和 roi_labels,inside_weight, outside_weight,产生分类和回归的 loss

2.模型细节

给出 anchor 的分类（前景和背景）和位置预测的标签，还有两个权重，这一层主要是为计算 RPN 的 loss 做标签

```
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"
  }
}
```

1.生成 anchor：
a.生成左上角的 9 个 anchor（3 个 scale，3 个 ratio）

anchor = np.array([[-83., -39., 100.,   56.],
    [-175., -87., 192., 104.],
    [-359., -183., 376., 200.],
    [-55., -55.,   72.,   72.],
    [-119., -119., 136., 136.],
    [-247., -247., 264., 264.],
    [-35., -79.,   52.,   96.],
    [-79., -167.,   96., 184.],
    [-167., -343., 184., 360.]])
三个连在一起的框的 ratio 是相同的，每隔两个框的 scale 是相同的。
anchor 的大小和 ratio 由下列三个参数控制：

```
generate_anchors(base_size=16, ratios=[0.5, 1, 2],
                 scales=2**np.arange(3, 6)):
"""
```

b.生成偏移量：
偏移量是基于特征图的大小和 resize 过后的图到特征图的缩放倍数来等距离生成的。

```
# 1. Generate proposals from bbox deltas and shifted anchors
shift_x = np.arange(0, width) * self._feat_stride
shift_y = np.arange(0, height) * self._feat_stride
shift_x, shift_y = np.meshgrid(shift_x, shift_y)
shifts = np.vstack((shift_x.ravel(), shift_y.ravel(),
                    shift_x.ravel(), shift_y.ravel())).transpose()
```

_feat_stride 为图片经过基础网络之后生成特征图的缩放倍数，

width 和 height 是特征图的宽和高。
所以 anchor 是原图 resize 之后的图中的固定框。

c.由 9 个 anchor 和生成的偏移量来生成全图的 anchor：

```
# add A anchors (1, A, 4) to
# cell K shifts (K, 1, 4) to get
# shift anchors (K, A, 4)
# reshape to (K*A, 4) shifted anchors
A = self._num_anchors
K = shifts.shape[0]
all_anchors = (self._anchors.reshape((1, A, 4)) +
               shifts.reshape((1, K, 4)).transpose((1, 0, 2)))
all_anchors = all_anchors.reshape((K * A, 4))
total_anchors = int(K * A)
```

其中 K = feature_h*feature_w,A=9

2.对 anchor 的一些简单处理：
只保留在图片之内的一些框：

```
# only keep anchors inside the image
inds_inside = np.where(
    (all_anchors[:, 0] >= -self._allowed_border) &
    (all_anchors[:, 1] >= -self._allowed_border) &
    (all_anchors[:, 2] < im_info[1] + self._allowed_border) &  # width
    (all_anchors[:, 3] < im_info[0] + self._allowed_border)    # height
)[0]
```

其中 im_info 为图片 resize 之后的大小，这个保存在 LMBD 之中。
之后打标签的 anchor 都在这里面选

3.分类标签：
对上一步产生的 anchor 分配 target label，前景 or 背景，以便训练 rpn

1. 对于每个 gt box，找到与他 iou 最大的 anchor 然后设为正样本
2. 对于每个 anchor 只要它与任意一个 gt box iou>0.7 即设为正样本
3. 对于每个 anchor 它与任意一个 gt box iou 都<0.3 即设为负样本
4. 不是正也不是负的 anchor 被忽略

a.初始化标签：

```
# label: 1 is positive, 0 is negative, -1 is dont care
labels = np.empty((len(inds_inside), ), dtype=np.float32)
labels.fill(-1)
```

1 为正样本，0 为负样本，-1 为不关心的样本

b.计算候选的 anchor 与真实框 IOU，并计算所有 anchor 与真实框的最大值及其位置，所有真实框与 anchor 的最大值及其位置。

```
# overlaps between the anchors and the gt boxes
# overlaps (ex, gt)
overlaps = bbox_overlaps(
    np.ascontiguousarray(anchors, dtype=np.float),
    np.ascontiguousarray(gt_boxes, dtype=np.float))
argmax_overlaps = overlaps.argmax(axis=1)
max_overlaps = overlaps[np.arange(len(inds_inside)), argmax_overlaps]
gt_argmax_overlaps = overlaps.argmax(axis=0)
gt_max_overlaps = overlaps[gt_argmax_overlaps,
                           np.arange(overlaps.shape[1])]
gt_argmax_overlaps = np.where(overlaps == gt_max_overlaps)[0]
```

c.根据上面算得的两个指标来为 anchor 打分类标签：

```
if not cfg.TRAIN.RPN_CLOBBER_POSITIVES:
    # assign bg labels first so that positive labels can clobber them
    labels[max_overlaps < cfg.TRAIN.RPN_NEGATIVE_OVERLAP] = 0

# fg label: for each gt, anchor with highest overlap
labels[gt_argmax_overlaps] = 1

# fg label: above threshold IOU
labels[max_overlaps >= cfg.TRAIN.RPN_POSITIVE_OVERLAP] = 1

if cfg.TRAIN.RPN_CLOBBER_POSITIVES:
    # assign bg labels last so that negative labels can clobber positives
    labels[max_overlaps < cfg.TRAIN.RPN_NEGATIVE_OVERLAP] = 0
```

所有 anchor 与真实框的最大值及其位置（gt_max_overlaps,gt_argmax_overlaps）：表示几个与真实框最接近的 anchor。这里得到的几个 anchor 标为正样本
所有真实框与 anchor 的最大值及其位置(max_overlaps,argmax_overlaps)：表示 anchor 与最接近的 GT 是所少。这里最大值大于阈值（0.7）的 anchor 标为正样本，小于阈值（0.3）的标为负样本，位于之间的标为不关心样本。

d.正负样本之和为 256。

如果正样本超过 128,则将多余的正样本标记为不关心样本。
其中 num_fg = int(0.5*256)

```
# subsample positive labels if we have too many
num_fg = int(cfg.TRAIN.RPN_FG_FRACTION * cfg.TRAIN.RPN_BATCHSIZE)
fg_inds = np.where(labels == 1)[0]
if len(fg_inds) > num_fg:
    disable_inds = npr.choice(
        fg_inds, size=(len(fg_inds) - num_fg), replace=False)
    labels[disable_inds] = -1
```

如果负样本超过 256-num(正样本)，则将多余的负样本标记为不关心样本。

4.回归标签：

```
bbox_targets = np.zeros((len(inds_inside), 4), dtype=np.float32)
bbox_targets = _compute_targets(anchors, gt_boxes[argmax_overlaps, :])
```

其中：argmax_overlaps 为 anchor 到最近的 ground truth 的标号。gt_boxes[argmax_overlaps,:]为 anchors 到最近的 ground truth 的坐标（左上角坐标，右下角坐标），所以 anchors 和 gt_boxes[argmax_overlaps,:]的维度是一样的。anchors[0,:]这个框对应的最近的 GT 为 gt_boxes[argmax_overlaps,:][0,:]

计算 anchor 和 GT 偏差的函数

```
def bbox_transform(ex_rois, gt_rois):
    ex_widths = ex_rois[:, 2] - ex_rois[:, 0] + 1.0
    ex_heights = ex_rois[:, 3] - ex_rois[:, 1] + 1.0
    ex_ctr_x = ex_rois[:, 0] + 0.5 * ex_widths
    ex_ctr_y = ex_rois[:, 1] + 0.5 * ex_heights

    gt_widths = gt_rois[:, 2] - gt_rois[:, 0] + 1.0
    gt_heights = gt_rois[:, 3] - gt_rois[:, 1] + 1.0
    gt_ctr_x = gt_rois[:, 0] + 0.5 * gt_widths
    gt_ctr_y = gt_rois[:, 1] + 0.5 * gt_heights

    targets_dx = (gt_ctr_x - ex_ctr_x) / ex_widths
    targets_dy = (gt_ctr_y - ex_ctr_y) / ex_heights
    targets_dw = np.log(gt_widths / ex_widths)
    targets_dh = np.log(gt_heights / ex_heights)

    targets = np.vstack(
        (targets_dx, targets_dy, targets_dw, targets_dh)).transpose()
    return targets
```

5.计算 inside_weights，outside_weights

这两个 weights 在计算 RPNloss 的时候会用到
a.计算 inside_weights
只有前景（正样本）的 anchor 才会计算回归，inside_weights 就是一个正样本的 mask。

```
bbox_inside_weights = np.zeros((len(inds_inside), 4), dtype=np.float32)
bbox_inside_weights[labels == 1, :] = np.array(cfg.TRAIN.RPN_BBOX_INSIDE_WEIGHTS)
```

```
__C.TRAIN.RPN_BBOX_INSIDE_WEIGHTS = (1.0, 1.0, 1.0, 1.0)
```

b.计算 outside_weights
outside_weights 为正负样本的一个加权。

```
bbox_outside_weights = np.zeros((len(inds_inside), 4), dtype=np.float32)
if cfg.TRAIN.RPN_POSITIVE_WEIGHT < 0:
    # uniform weighting of examples (given non-uniform sampling)
    num_examples = np.sum(labels >= 0)
    positive_weights = np.ones((1, 4)) * 1.0 / num_examples
    negative_weights = np.ones((1, 4)) * 1.0 / num_examples
else:
    assert ((cfg.TRAIN.RPN_POSITIVE_WEIGHT > 0) &
            (cfg.TRAIN.RPN_POSITIVE_WEIGHT < 1))
    positive_weights = (cfg.TRAIN.RPN_POSITIVE_WEIGHT /
                        np.sum(labels == 1))
    negative_weights = ((1.0 - cfg.TRAIN.RPN_POSITIVE_WEIGHT) /
                        np.sum(labels == 0))
bbox_outside_weights[labels == 1, :] = positive_weights
bbox_outside_weights[labels == 0, :] = negative_weights
```

6.将打好标签的 label，bbox，weights 装进盒子中

a.映射到原来 anchors 的集合中（包含图片外的 anchor 的集合中）

```
# map up to original set of anchors
labels = _unmap(labels, total_anchors, inds_inside, fill=-1)
bbox_targets = _unmap(bbox_targets, total_anchors, inds_inside, fill=0)
bbox_inside_weights = _unmap(bbox_inside_weights, total_anchors, inds_inside, fill=0)
bbox_outside_weights = _unmap(bbox_outside_weights, total_anchors, inds_inside, fill=0)
```

其中 total_anchors = int(K*A)=h*w*9
inds_inside 为在图像中的 anchors 的 索引。

```
def _unmap(data, count, inds, fill=0):
    """ Unmap a subset of item (data) back to the original set of items (of
    size count) """
    if len(data.shape) == 1:
        ret = np.empty((count, ), dtype=np.float32)
        ret.fill(fill)
        ret[inds] = data
    else:
        ret = np.empty((count, ) + data.shape[1:], dtype=np.float32)
        ret.fill(fill)
        ret[inds, :] = data
    return ret
```

b. reshape 到网络输出的格式

```
# labels
labels = labels.reshape((1, height, width, A)).transpose(0, 3, 1, 2)
labels = labels.reshape((1, 1, A * height, width))
top[0].reshape(*labels.shape)
top[0].data[...] = labels

# bbox_targets
bbox_targets = bbox_targets \
    .reshape((1, height, width, A * 4)).transpose(0, 3, 1, 2)
top[1].reshape(*bbox_targets.shape)
top[1].data[...] = bbox_targets

# bbox_inside_weights
bbox_inside_weights = bbox_inside_weights \
    .reshape((1, height, width, A * 4)).transpose(0, 3, 1, 2)
assert bbox_inside_weights.shape[2] == height
assert bbox_inside_weights.shape[3] == width
top[2].reshape(*bbox_inside_weights.shape)
top[2].data[...] = bbox_inside_weights

# bbox_outside_weights
bbox_outside_weights = bbox_outside_weights \
    .reshape((1, height, width, A * 4)).transpose(0, 3, 1, 2)
assert bbox_outside_weights.shape[2] == height
assert bbox_outside_weights.shape[3] == width
top[3].reshape(*bbox_outside_weights.shape)
top[3].data[...] = bbox_outside_weights
```

生成 proposal，输入：rpn 输出的 bbox_pred,cls_prob,im_info。

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

1.初始化一些参数：

```
cfg_key = str(self.phase) # either 'TRAIN' or 'TEST'
pre_nms_topN  = cfg[cfg_key].RPN_PRE_NMS_TOP_N
post_nms_topN = cfg[cfg_key].RPN_POST_NMS_TOP_N
nms_thresh    = cfg[cfg_key].RPN_NMS_THRESH
min_size      = cfg[cfg_key].RPN_MIN_SIZE
```

pre_nms_topN =12000(先选取前景框分数前 12000 的 anchor，在测试时这个数值是 6000)
post_nms_topN = 2000(在 NMS 之后，再选取前景框分数前 2000 的 anchor，测试时数值为 300)
nms_thresh = 0.7(NMS 时，IOU 的阈值)
min_size = 16（最小的框的阈值）

2.利用 bbox_deltas 和 anchors 产生 proposals
a.产生 anchors
方法与之前的一样

```
# Enumerate all shifts
shift_x = np.arange(0, width) * self._feat_stride
shift_y = np.arange(0, height) * self._feat_stride
shift_x, shift_y = np.meshgrid(shift_x, shift_y)
shifts = np.vstack((shift_x.ravel(), shift_y.ravel(),
                    shift_x.ravel(), shift_y.ravel())).transpose()

# Enumerate all shifted anchors:
#
# add A anchors (1, A, 4) to
# cell K shifts (K, 1, 4) to get
# shift anchors (K, A, 4)
# reshape to (K*A, 4) shifted anchors
A = self._num_anchors
K = shifts.shape[0]
anchors = self._anchors.reshape((1, A, 4)) + \
          shifts.reshape((1, K, 4)).transpose((1, 0, 2))
anchors = anchors.reshape((K * A, 4))
```

b. reshape bbox_deltas 和 scores

```
# Transpose and reshape predicted bbox transformations to get them
# into the same order as the anchors:
#
# bbox deltas will be (1, 4 * A, H, W) format
# transpose to (1, H, W, 4 * A)
# reshape to (1 * H * W * A, 4) where rows are ordered by (h, w, a)
# in slowest to fastest order
bbox_deltas = bbox_deltas.transpose((0, 2, 3, 1)).reshape((-1, 4))
```

```
# Same story for the scores:
#
# scores are (1, A, H, W) format
# transpose to (1, H, W, A)
# reshape to (1 * H * W * A, 1) where rows are ordered by (h, w, a)
scores = scores.transpose((0, 2, 3, 1)).reshape((-1, 1))
```

c. 用 bbox_deltas 对 anchors 进行微调

```
# Convert anchors into proposals via bbox transformations
proposals = bbox_transform_inv(anchors, bbox_deltas)
```

```
def bbox_transform_inv(boxes, deltas):
    if boxes.shape[0] == 0:
        return np.zeros((0, deltas.shape[1]), dtype=deltas.dtype)

    boxes = boxes.astype(deltas.dtype, copy=False)

    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

    dx = deltas[:, 0::4]
    dy = deltas[:, 1::4]
    dw = deltas[:, 2::4]
    dh = deltas[:, 3::4]

    pred_ctr_x = dx * widths[:, np.newaxis] + ctr_x[:, np.newaxis]
    pred_ctr_y = dy * heights[:, np.newaxis] + ctr_y[:, np.newaxis]
    pred_w = np.exp(dw) * widths[:, np.newaxis]
    pred_h = np.exp(dh) * heights[:, np.newaxis]

    pred_boxes = np.zeros(deltas.shape, dtype=deltas.dtype)
    # 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
```

3. 将 proposals 剪裁到图片之中

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

```
def clip_boxes(boxes, im_shape):
    """
    Clip boxes to image boundaries.
    """

    # x1 >= 0
    boxes[:, 0::4] = np.maximum(np.minimum(boxes[:, 0::4], im_shape[1] - 1), 0)
    # y1 >= 0
    boxes[:, 1::4] = np.maximum(np.minimum(boxes[:, 1::4], im_shape[0] - 1), 0)
    # x2 < im_shape[1]
    boxes[:, 2::4] = np.maximum(np.minimum(boxes[:, 2::4], im_shape[1] - 1), 0)
    # y2 < im_shape[0]
    boxes[:, 3::4] = np.maximum(np.minimum(boxes[:, 3::4], im_shape[0] - 1), 0)
    return boxes
```

4.去除 proposals 中尺寸小于阈值的

```
# 3. remove predicted boxes with either height or width < threshold
# (NOTE: convert min_size to input image scale stored in im_info[2])
keep = _filter_boxes(proposals, min_size * im_info[2])
proposals = proposals[keep, :]
scores = scores[keep]
```

```
def _filter_boxes(boxes, min_size):
    """Remove all boxes with any side smaller than min_size."""
    ws = boxes[:, 2] - boxes[:, 0] + 1
    hs = boxes[:, 3] - boxes[:, 1] + 1
    keep = np.where((ws >= min_size) & (hs >= min_size))[0]
    return keep
```

5.取 scores 前 6000 的 proposals

```
# 4. sort all (proposal, score) pairs by score from highest to lowest
# 5. take top pre_nms_topN (e.g. 6000)
order = scores.ravel().argsort()[::-1]
if pre_nms_topN > 0:
    order = order[:pre_nms_topN]
proposals = proposals[order, :]
scores = scores[order]
```

6. NMS → 取 scores 前 2000(测试时为 300)的 proposals

```
# 6. apply nms (e.g. threshold = 0.7)
# 7. take after_nms_topN (e.g. 300)
# 8. return the top proposals (-> RoIs top)
keep = nms(np.hstack((proposals, scores)), nms_thresh)
if post_nms_topN > 0:
    keep = keep[:post_nms_topN]
proposals = proposals[keep, :]
scores = scores[keep]
```

7.返回 2000 个 proposals

```
# Output rois blob
# Our RPN implementation only supports a single input image, so all
# batch inds are 0
batch_inds = np.zeros((proposals.shape[0], 1), dtype=np.float32)
blob = np.hstack((batch_inds, proposals.astype(np.float32, copy=False)))
top[0].reshape(*(blob.shape))
top[0].data[...] = blob
```

根据 GT 制作 roi 的 labels、bbox_targets、和两个 weight，这个和之前制作 anchors 的标签类似。

Rois：[128,5] 第一列为 0,后面四列为坐标。
labels：[128,1] 分类标签

bbox_targets：[128,4*number_cls],位置标签。
bbox_inside_weights：[128,4*number_cls],分类标签不为背景的 rois 的 mask
bbox_outside_weights:与 inside_weights 一样。

```
layer {
  name: 'roi-data'
  type: 'Python'
  bottom: 'rpn_rois'
  bottom: 'gt_boxes'
  top: 'rois'
  top: 'labels'
  top: 'bbox_targets'
  top: 'bbox_inside_weights'
  top: 'bbox_outside_weights'
  python_param {
    module: 'rpn.proposal_target_layer'
    layer: 'ProposalTargetLayer'
    param_str: "'num_classes': 7"
  }
}
```

1.一些初始化的设置

```
def forward(self, bottom, top):
    # Proposal ROIs (0, x1, y1, x2, y2) coming from RPN
    # (i.e., rpn.proposal_layer.ProposalLayer), or any other source
    all_rois = bottom[0].data
    # GT boxes (x1, y1, x2, y2, label)
    # TODO(rbg): it's annoying that sometimes I have extra info before
    # and other times after box coordinates -- normalize to one format
    gt_boxes = bottom[1].data

    # Include ground-truth boxes in the set of candidate rois
    zeros = np.zeros((gt_boxes.shape[0], 1), dtype=gt_boxes.dtype)
    all_rois = np.vstack(
        (all_rois, np.hstack((zeros, gt_boxes[:, :-1])))
    )

    # Sanity check: single batch only
    assert np.all(all_rois[:, 0] == 0), \
            'Only single item batches are supported'

    num_images = 1
    rois_per_image = cfg.TRAIN.BATCH_SIZE / num_images
    fg_rois_per_image = np.round(cfg.TRAIN.FG_FRACTION * rois_per_image)
```

这里把 GT 框也加入到 roi 中了
其中：fg_rois_per_image = 0.25*128=32

2.对 rois 进行采样，并制作标签和 weight

```
 # Sample rois with classification labels and bounding box regression
 # targets
 labels, rois, bbox_targets, bbox_inside_weights = _sample_rois(
     all_rois, gt_boxes, fg_rois_per_image,
     rois_per_image, self._num_classes)
```

```
def _sample_rois(all_rois, gt_boxes, fg_rois_per_image, rois_per_image, num_classes):
    """Generate a random sample of RoIs comprising foreground and background
    examples.
    """
```

a.计算 GT 与 rois 之间的 overlaps，并将每个 roi 根据 overlap 分配到最接近的 GT 上。

```
# overlaps: (rois x gt_boxes)
overlaps = bbox_overlaps(
    np.ascontiguousarray(all_rois[:, 1:5], dtype=np.float),
    np.ascontiguousarray(gt_boxes[:, :4], dtype=np.float))
gt_assignment = overlaps.argmax(axis=1)
max_overlaps = overlaps.max(axis=1)
labels = gt_boxes[gt_assignment, 4]
```

其中：
gt_assignment 的 shape 为[rois,1],值为最近的 GT 的 index。
max_overlaps 的 shape 为[rois,1],值为最大的 IOU。
labels 的 shape 为[rois,1],值为最接近的 GT 的分类标签。

b.卡 IOU 阈值选取前景，并对所有前景进行采样。

```
# Select foreground RoIs as those with >= FG_THRESH overlap
fg_inds = np.where(max_overlaps >= cfg.TRAIN.FG_THRESH)[0]
# Guard against the case when an image has fewer than fg_rois_per_image
# foreground RoIs
fg_rois_per_this_image = int(min(fg_rois_per_image, fg_inds.size))
# Sample foreground regions without replacement
if fg_inds.size > 0:
    fg_inds = npr.choice(fg_inds, size=fg_rois_per_this_image, replace=False)
```

其中：
TRAIN.FG_THRESH = 0.5， 表示 rois 与 GT 最大的 IOU 超过 0.5,则设置为前景。
fg_rois_per_this_image 最大为 32。所以前景最多采样 32 个。

c.卡 IOU 阈值选取背景，并对所有的背景进行采样。

```
# Select background RoIs as those within [BG_THRESH_LO, BG_THRESH_HI)
bg_inds = np.where((max_overlaps < cfg.TRAIN.BG_THRESH_HI) &
                   (max_overlaps >= cfg.TRAIN.BG_THRESH_LO))[0]
# Compute number of background RoIs to take from this image (guarding
# against there being fewer than desired)
bg_rois_per_this_image = rois_per_image - fg_rois_per_this_image
bg_rois_per_this_image = min(bg_rois_per_this_image, bg_inds.size)
# Sample background regions without replacement
if bg_inds.size > 0:
    bg_inds = npr.choice(bg_inds, size=bg_rois_per_this_image, replace=False)
```

其中：
TRAIN.BG_THRESH_HI=0.5
TRAIN.BG_THRESH_LO=0.1
所以 rois 与 GT 最大的 IOU 在 0.1~0.5 范围之内的标记为背景。
背景 rois 最多为 128-32 = 96 个。

d. 根据 b,c 两步选出的前后背景的 inds（索引）来选出其 labels，和坐标

```
# The indices that we're selecting (both fg and bg)
keep_inds = np.append(fg_inds, bg_inds)
# Select sampled values from various arrays:
labels = labels[keep_inds]
# Clamp labels for the background RoIs to 0
labels[fg_rois_per_this_image:] = 0
rois = all_rois[keep_inds]
```

注意：
需要将背景的分类标签置为 0,
这里 labels 为前面采样出来的 roi 的分类标签。
rois 为前面采样出来的 roi 的位置坐标。

e. 计算出采样出的 rois 与其最近的 GT 之间的偏差（即为 bbox_targets）

```
bbox_target_data = _compute_targets(
    rois[:, 1:5], gt_boxes[gt_assignment[keep_inds], :4], labels)
```

```
def _compute_targets(ex_rois, gt_rois, labels):
    """Compute bounding-box regression targets for an image."""

    assert ex_rois.shape[0] == gt_rois.shape[0]
    assert ex_rois.shape[1] == 4
    assert gt_rois.shape[1] == 4

    targets = bbox_transform(ex_rois, gt_rois)
    if cfg.TRAIN.BBOX_NORMALIZE_TARGETS_PRECOMPUTED:
        # Optionally normalize targets by a precomputed mean and stdev
        targets = ((targets - np.array(cfg.TRAIN.BBOX_NORMALIZE_MEANS))
                   / np.array(cfg.TRAIN.BBOX_NORMALIZE_STDS))
    return np.hstack(
            (labels[:, np.newaxis], targets)).astype(np.float32, copy=False)
```

注意：
这里返回的为分类标签和位置标签，这是为了后面制作与输出对应的回归标签，因为需要分类标签。
即 shape 为[128,5]第一列为分类标签，后四列为位置标签。

f.制作与输出对应的回归标签，和权重。

```
bbox_targets, bbox_inside_weights = \
    _get_bbox_regression_labels(bbox_target_data, num_classes)
```

```
def _get_bbox_regression_labels(bbox_target_data, num_classes):
    """Bounding-box regression targets (bbox_target_data) are stored in a
    compact form N x (class, tx, ty, tw, th)

    This function expands those targets into the 4-of-4*K representation used
    by the network (i.e. only one class has non-zero targets).

    Returns:
        bbox_target (ndarray): N x 4K blob of regression targets
        bbox_inside_weights (ndarray): N x 4K blob of loss weights
    """

    clss = bbox_target_data[:, 0]
    bbox_targets = np.zeros((clss.size, 4 * num_classes), dtype=np.float32)
    bbox_inside_weights = np.zeros(bbox_targets.shape, dtype=np.float32)
    inds = np.where(clss > 0)[0]
    for ind in inds:
        cls = clss[ind]
        start = int(4 * cls)
        end = start + 4
        bbox_targets[ind, start:end] = bbox_target_data[ind, 1:]
        bbox_inside_weights[ind, start:end] = cfg.TRAIN.BBOX_INSIDE_WEIGHTS
    return bbox_targets, bbox_inside_weights
```

```
layer {
  name: "bbox_pred"
  type: "InnerProduct"
  bottom: "fc7"
  top: "bbox_pred"
  param { lr_mult: 1.0 }
  param { lr_mult: 2.0 }
  inner_product_param {
    num_output: 28
    weight_filler {
      type: "gaussian"
      std: 0.001
    }
```

可以看到，全链接输出的为[N,28](这里类别数是 7)
所以，我们要将之前得到的[N,4]的位置标签，
结合[N,1]的分类标签来生成与输出匹配的[N,28]的标签.

其中：
TRAIN.BBOX_INSIDE_WEIGHTS = (1.0, 1.0, 1.0, 1.0)

可以看到，这里是将位置标签复制到对应的类处。
inside_weights 的制作相同。
g. outside_weight

```
# bbox_outside_weights
top[4].reshape(*bbox_inside_weights.shape)
top[4].data[...] = np.array(bbox_inside_weights > 0).astype(np.float32)
```

outside_weight 与 inside_weight 一样。

Roipooling：
根据 roi 的位置坐标来计算其对应区域的特征。
输入：rois，位置坐标
　　conv5, 基础网络特征
输出：[128,6,6]

```
layer {
  name: "roi_pool_conv5"
  type: "ROIPooling"
  bottom: "conv5"
  bottom: "rois"
  top: "roi_pool_conv5"
  roi_pooling_param {
    pooled_w: 6
    pooled_h: 6
    spatial_scale: 0.0625 # 1/16
  }
}
```

roipooling 具体源码解读可以看这篇博客：
http://blog.csdn.net/lanran2/article/details/60143861

两个全连接层：

```
layer {
  name: "fc6"
  type: "InnerProduct"
  bottom: "roi_pool_conv5"
  top: "fc6"
  param { lr_mult: 1.0 }
  param { lr_mult: 2.0 }
  inner_product_param {
    num_output: 4096
  }
}
layer {
  name: "relu6"
  type: "ReLU"
  bottom: "fc6"
  top: "fc6"
}
layer {
  name: "drop6"
  type: "Dropout"
  bottom: "fc6"
  top: "fc6"
  dropout_param {
    dropout_ratio: 0.5
    scale_train: false
  }
}
```

```
layer {
  name: "fc7"
  type: "InnerProduct"
  bottom: "fc6"
  top: "fc7"
  param { lr_mult: 1.0 }
  param { lr_mult: 2.0 }
  inner_product_param {
    num_output: 4096
  }
}
layer {
  name: "relu7"
  type: "ReLU"
  bottom: "fc7"
  top: "fc7"
}
layer {
  name: "drop7"
  type: "Dropout"
  bottom: "fc7"
  top: "fc7"
  dropout_param {
    dropout_ratio: 0.5
    scale_train: false
  }
}
```

由全连接层得到位置预测：

```
layer {
  name: "bbox_pred"
  type: "InnerProduct"
  bottom: "fc7"
  top: "bbox_pred"
  param { lr_mult: 1.0 }
  param { lr_mult: 2.0 }
  inner_product_param {
    num_output: 28
    weight_filler {
      type: "gaussian"
      std: 0.001
    }
    bias_filler {
      type: "constant"
      value: 0
    }
  }
}
```

由全连接层得到类别预测：

```
layer {
  name: "cls_score"
  type: "InnerProduct"
  bottom: "fc7"
  top: "cls_score"
  param { lr_mult: 1.0 }
  param { lr_mult: 2.0 }
  inner_product_param {
    num_output: 7
    weight_filler {
      type: "gaussian"
      std: 0.01
    }
    bias_filler {
      type: "constant"
      value: 0
    }
  }
}
```

由之前的 roi-data 层产生的 labels 标签生成分类 loss：

```
layer {
  name: "loss_cls"
  type: "SoftmaxWithLoss"
  bottom: "cls_score"
  bottom: "labels"
  propagate_down: 1
  propagate_down: 0
  top: "cls_loss"
  loss_weight: 1
  loss_param {
    ignore_label: -1
    normalize: true
  }
}
```

由之前的 roi-data 层产生的 rpn_bbox_targets 标签，inside_weight，outside_weight 生成回归的 loss

```
layer {
  name: "loss_bbox"
  type: "SmoothL1Loss"
  bottom: "bbox_pred"
  bottom: "bbox_targets"
  bottom: 'bbox_inside_weights'
  bottom: 'bbox_outside_weights'
  top: "bbox_loss"
  loss_weight: 1
}
```