mmdetection 解析

Sisyphes,yehao

2019 年 8 月 1 日

目录

第一节　结构设计

- BackBone: 特征提取骨架网络,ResNet,ResneXt 等.
- Neck: 连接骨架和头部. 多层级特征融合,FPN,BFP 等.
- DenseHead: 处理特征图上的密集框部分, 主要分 AnchorHead, AnchorFreeHead 两大类，分别有 RPNHead, SSDHead,RetinaHead 和 FCOSHead 等.
- RoIExtractor: 汇集不同层级的特征框, 将其大小统一, 为二步定位, 类别优化服务.
- RoIHead (BBoxHead/MaskHead): 类别分类或位置回归等.
- OneStage: BackBone + Neck + DenseHead
- TwoStage: BackBone + Neck + (DenseHead) + RoIExtractor + RoIHead

代码结构:
configs 网络组件结构等配置信息
tools: 训练和测试的最终包装
mmdet:

apis: 分布式环境设定, 推断和训练基类代码
core: anchor,bbox,mask 等在训练前和训练中的各种变换函数
datasets:coco 和 voc 格式的数据类以及一些增强代码
models: 模型组件，采用注册和组合构建的形式完成模型搭建
ops: 优化加速代码, 包括 nms,roialign,dcn,gcb,mask，focal_loss 等

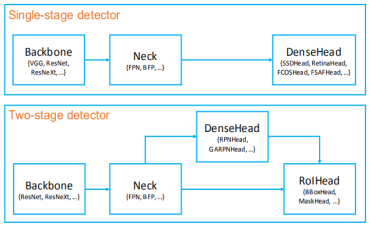


图 1: Framework

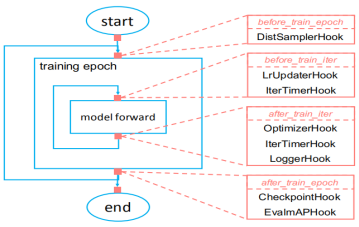


图 2: Trainning pipeline

1.1　总体逻辑

在最外层的 train.py 中能看到:

1. mmcv.Config.fromfile 从配置文件解析配置信息, 并做适当更新, 包括预加载模型文件, 分布式相关等
2. mmdet.models.builder 中的 build_detector 根据配置信息构造模型

   2.5 build 函数调用 _build_module(新版为 build_from_cfg) 函数, 按 type 关键字从注册表中获取相应的模型对象, 并根据配置参数实例化对象 (配置文件的模型参数只占了各模型构造参数的一小部分, 模型结构并非可以随意更改).

   2.6 registr.py 实现了模型的注册装饰器, 其主要功能就是将各模型组件类对象保存到 registry.module_dict 中, 从而可以实现 2.5 所示功能.

   2.7 目前包含 BACKBONES,NECKS,ROI_EXTRACTORS,SHARED_HEADS,HEADS,LOSSES,DETECTORS 七个 (容器). 注册器可按 @NAME.register_module 方式装饰, 新增. 所有被注册的对象都是一个完整的 pytorch 构图

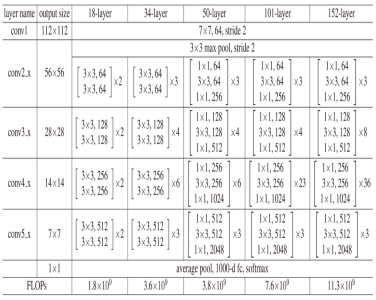
   2.9 @DETECTORS.register_module 装饰了完整的检测算法 (OneStage，TwoStage), 各个部件在其 init() 函数中实例化, 实现 2.5 的依次调用.
3. 最后是数据迭代器和训练 pipeline四.

1.2　Configs

配置方式支持 python/json/yaml, 从 mmcv 的 Config 解析, 其功能同 maskrcnn-benchmark 的 yacs 类似, 将字典的取值方式属性化.

配置文件模型部分包含模型组件及其可改动模型结构的参数, 比如 backbone 的层数, 冻结的 stage;bbox_head 的 in_channel，类别，损失函数等; 训练部分主要包括 anchor 采样相关系数; 测试包括非极大抑制等相关参数; 剩下数据, 优化器, 模型管理, 日志等相关信息, 一看即明.

1.3 Backbone



layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
		3×3 max pool, stride 2				
conv2_x	56×56	[3×3, 64; 3×3, 64] ×2	[3×3, 64; 3×3, 64] ×3	[1×1, 64; 3×3, 64; 1×1, 256] ×3	[1×1, 64; 3×3, 64; 1×1, 256] ×3	[1×1, 64; 3×3, 64; 1×1, 256] ×3
conv3_x	28×28	[3×3, 128; 3×3, 128] ×2	[3×3, 128; 3×3, 128] ×4	[1×1, 128; 3×3, 128; 1×1, 512] ×4	[1×1, 128; 3×3, 128; 1×1, 512] ×4	[1×1, 128; 3×3, 128; 1×1, 512] ×8
conv4_x	14×14	[3×3, 256; 3×3, 256] ×2	[3×3, 256; 3×3, 256] ×6	[1×1, 256; 3×3, 256; 1×1, 1024] ×6	[1×1, 256; 3×3, 256; 1×1, 1024] ×23	[1×1, 256; 3×3, 256; 1×1, 1024] ×36
conv5_x	7×7	[3×3, 512; 3×3, 512] ×2	[3×3, 512; 3×3, 512] ×3	[1×1, 512; 3×3, 512; 1×1, 2048] ×3	[1×1, 512; 3×3, 512; 1×1, 2048] ×3	[1×1, 512; 3×3, 512; 1×1, 2048] ×3
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

图 3: resnet

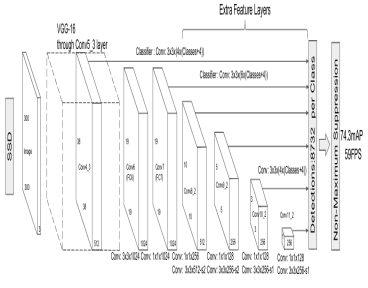
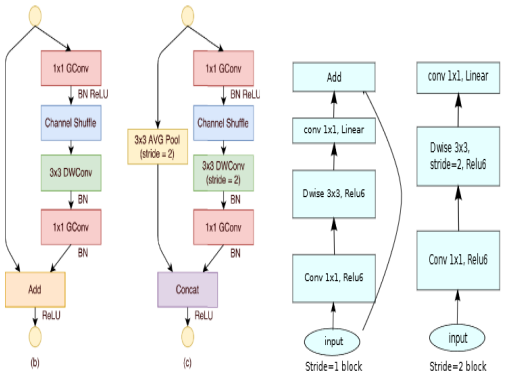


图 4: ssd



(c) ShuffleNet [20]　　(d) Mobilenet V2

图 5: mobilenetv2

1.3.1 backbone 及改进

Resnt 的 backbone 由 BasicBlock，Bottleneck，make_res_layer 构成，前两个均是基本的残差结构，以卷积核大小分别为 1,3,1 的顺序构件图，最后或接一下采样层，组成一个基础的残差 block。make_res_layer 中重复构造残差模块，增加感受野和特征的复杂度。

backbone 的改进主要有模型轻量和功能扩展。

这里主要说说 resnet.py 中引进的 dcn，gcb 和 gen_attention 等功能模块。从 Bottleneck 的 _inner_forward 可以看到，gen_attention_block

加到 kerner_size=1, kerner_size=3 组 (组: 卷积，归一化，激活) 之后, kerner_size=1,planes 扩展组之前。gcb 加到 planes 扩展组之后，然后才是下采样。因 dcn 是替换卷积算子，其参数和基本卷积相同，因此你可以任意替换 conv2d，只是试验表明，替换此处的 kerner_size=3 组效果更好。

代码参数说明，主要和 fpn 对齐部分，在 config 中按照如下对齐方式更改即可。

```

@BACKBONES.register_module
class ResNet(nn.Module):
  arch_settings = {
    18: (BasicBlock, (2, 2, 2, 2)),    # (2, 2, 2, 2)各层残差块的重复数目
    34: (BasicBlock, (3, 4, 6, 3)),
    50: (Bottleneck, (3, 4, 6, 3)),
    101: (Bottleneck, (3, 4, 23, 3)),
    152: (Bottleneck, (3, 8, 36, 3))
  }
  def __init__(self,
          depth,
          in_channels=3,
          num_stages=4,
          strides=(1, 2, 2, 2),
          dilations=(1, 1, 1, 1),
          out_indices=(0, 1, 2, 3),
          style='pytorch',
          frozen_stages=-1,
          conv_cfg=None,
          norm_cfg=dict(type='BN', requires_grad=True),
          norm_eval=True,
          dcn=None,
          stage_with_dcn=(False, False, False, False),
          gcb=None,
          stage_with_gcb=(False, False, False, False),
          gen_attention=None,
          stage_with_gen_attention=((), (), (), ()),
          with_cp=False,
          zero_init_residual=True):
    super(ResNet, self).__init__()
  # num_stages: 下采样特征层数目
  # strides: 不同残差block的stride数
  # out_indices:  需要的特征层索引，对齐fpn
  # frozen_stages: 冻结的残差层
  # stage_with_dcn，stage_with_gcb，stage_with_gen_attention均需要和fpn层对齐

```

```
38  def forward(self, x):
39    # 首先是两个下采样:kernel_size=7和maxpool，此时尺度为原始1/4
40    x = self.conv1(x)
41    x = self.norm1(x)
42    x = self.relu(x)
43    x = self.maxpool(x)
44    # 然后是几个尺度减半的特征层 make_res_layer，对应fpn层
45    outs = []
46    for i, layer_name in enumerate(self.res_layers):
47      res_layer = getattr(self, layer_name)
48      x = res_layer(x)
49      if i in self.out_indices:
50        # out_indices 选取需要的层对应fpn层(通常index是连续的,除非数据特殊,
      刚好都只有一大一小物体)
51          outs.append(x)
52    return tuple(outs)
```

1.4 Necks

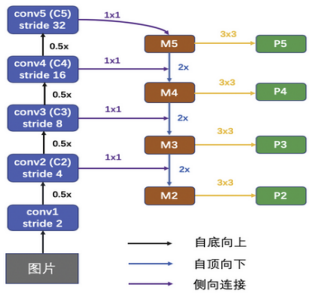


图 6: fpn

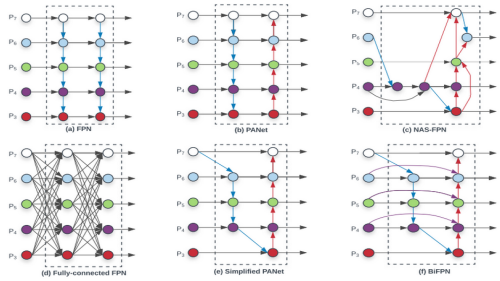


图 7: bifpn

1.4.1　多尺度结构

FPN,BFPN BiFPN, HRFPN 等。

1.5　Heads

Heads 主要包含了是三部分,anchor_heads, bbox_heads, mask_heads, anchor 是 two-stage 的 rpn 部分或者 one-stage 的头部. 最终的损失函数均在这里实现.

1.5.1　AnchorHead

anchor head 主要包括 AnchorGenerator, anchor_target, 后者包含了 assign,sample. 因为不同尺度操作雷同, 于是作者设计了一个公共函数 multi_apply. 这一小节主要说明以上函数, 损失函数见1.6.

AnchorGenerator 类为不同特征层生成 anchor, 输入参数 base_size, scales, ratios 分别表示:anchor 在特征层上的基础大小 (特征层相对于原图的 stride),anchor 在特征层上的尺度大小 (可以多个，增加感受野),anchor 在保持基础大小不变的情况下的长宽比. len(scales)*len(ratios) 即为一格的 anchor 个数.

比如输入图像大小 (640*640), 选择 (p2, p3) 作为其特征层, 则 p2 大小为 (160*160),base_ size=4, 若设定 ratios=[0.5,1.0,2.0], scales=[8, 16], 则在 p2 上一格对应的 base_anchor 的 (w,h) 为 [(45.25,22.63), (90.51, 45.25), (32.00, 32.00), (64.00, 64.00), (22.63, 45.25),(45.25, 90.51)]. 其中 $64 = 4 * 16 * 1, 90.51 = 4 * 16 * \sqrt{2}, 22.63 = 4 * 8/\sqrt{2}$. 那么每一格所对应的 6 个 base_anchor 相对于中心点的偏移量即为:

```
[[-21.,  -9.,  24.,  12.],
[-43., -21.,  46.,  24.],
[-14., -14.,  17.,  17.],
[-30., -30.,  33.,  33.],
[ -9., -21.,  12.,  24.],
[-21., -43.,  24.,  46.]]
```

因为这里都是相对于自身一格的偏移量, 所有当要算在原图上的中心偏移量时, 直接根据 torch 或者 numpy 的广播机制, 首先得到特征图在原图的网格坐标, 然后和此相加, 即得此特征层的所有 all_anchors. 见 grid_anchors 函数. 这里面涉及到一些操作技巧, 比如:

```
ws = (w * w_ratios[:, None] * self.scales[None, :]).view(-1)
hs = (h * h_ratios[:, None] * self.scales[None, :]).view(-1)
```

包括 nms,assign 等部分都有很多细节操作, 这些实现检测算法的基础函数, 需要多加练, 才能复现新的算法. 可见八. 如上不难看到, 输入图像的所有 anchor 数量巨大.

模块 anchor 中还有一个 anchor_target 文件, 其主要是为了给设计的 anchor 标定 label 以及筛选 anchor(便于优化等).

这里利用的 anchor_target_single 函数来解释 multi_apply 函数:

```
from functools import partial
from six.moves import map, zip
def multi_apply(func, *args, **kwargs):
    pfunc = partial(func, **kwargs) if kwargs else func
    map_results = map(pfunc, *args)
  return tuple(map(list, zip(*map_results)))
```

partial 函数的功能就是：把一个函数的某些参数给固定住，返回一个新的函数 (这里就是将 **kwargs 中的参数固定住). 当函数参数太多，需要固定某些参数时，可以使用 functools.partial 创建一个新的函数 map(function, sequence) , 对 sequence 中的 item 依次执行 function(item)，并将结果组成一个迭代器返回, 最后 zip 并行循环.

在 anchor_target 中

```
result = multi_apply(
         anchor_target_single,
         anchor_list,
         valid_flag_list,
         gt_bboxes_list,
         gt_bboxes_ignore_list,
         gt_labels_list,
         img_metas,
         target_means=target_means,
         target_stds=target_stds,
         cfg=cfg,
         label_channels=label_channels,
         sampling=sampling,
     unmap_outputs=unmap_outputs)
```

是先将配置文件, 采样与否的 flag，anchor 于 gt_box 在映射空间 (平移归一, 放缩 log) 中各偏差量的滑动平均处理 (见 core/bbox/transforms/bbox2delta 函数最后一行) 固定住. 然后 anchor_target_single 对 *args 参数：anchor_list,gt_bboxes_list, img_metas 等并行处理. 得到最终的 la-

bel,bbox, 正负样本 index. 这里 label_weights 用来记录筛选后的信息位置.

*args 参数的说明:

- anchor_list:见 anhor_head 中的 get_anchor 函数, 是一个 list[list[Tensors]] 结构，最外层是图片个数，再内一层是尺度个数，里面的 Tensors 的 shape 是 [H*W*4, 4]，其中 H 和 W 代表对应尺度特征图的高和宽.
- img_metas 有五个字段:ori/img/pad_shape, scale_factor, flip

在调用 anchor_target_single 函数之前:

```
num_imgs = len(img_metas)
assert len(anchor_list) == len(valid_flag_list) == num\_imgs
# anchor number of multi levels
# 选出第一张图的各个特征层的anchor数量,比如上面给的p2即为160*160*6,一格点对应的anhor shape为(6,4)
# 因为有多层,比如p2,p3，则[(160*160*4,4), (80*80*6,4)]
# 为images_to_levels函数,切片用:将所有以图片为第一维度的结果,转换成以特征图尺度个数为第一维度的结果(算loss)
num_level_anchors = [anchors.size(0) for anchors in anchor_list[0]]
# concat all level anchors and flags to a single tensor
#在调用multi\_apply之前需要将anchor\_list的不同层级的anchor cat在一起,最终变为list[Tensor]结构.
#最外层为图像个数.这样就能利用multi\_apply中的map在每张图上做做同样的操作了.
#将所有图的anchor合在一起
for i in range(num_imgs):
  assert len(anchor_list[i]) == len(valid_flag_list[i])
  anchor_list[i] = torch.cat(anchor_list[i])
  valid_flag_list[i] = torch.cat(valid_flag_list[i])
  # valid_flag_list结合meta信息,进一步筛选无效anhor(其实筛掉的很少)
```

anchor_target_single 函数：主要涉及 assign_and_sample,bbox2delta, unmap 四个函数.

assign_and_sample 根据 cfg 从配置信息拿到对应的 assign,sample 类对象, one-stage 模型没有 sample(不代表损失函数利用所有的 anchor),sample 主要为 two-stage 服务.

采样主要包含 RandomSampler，OHEMSampler，InstanceBalancedPosSampler，IoUBalancedNegSampler, 均继承自 BaseSampler 为一抽象类, 继承自它的的类必须完成 _sample_pos,_sample_neg 两函数. 按照 python 的语法，子类可以使用 raise NotImplementedError 来避免不能实例化的问题.PseudoSampler 即是如此, 它重写了 sample 函数, 该函数并没有做任何

筛选.RandomSampler 的意思自明, 其他采样后补. 标定 MaxIoUAssigner 见1.7.1.

bbox2delta 主要注意一点的是相对位置是相对于 anchor 的, 所以除的是 anchor 的 w, h

$$d_x = (g_x - p_x)/p_w, d_y = (g_y - p_y)/p_h, d_w = \log(g_w/p_w), d_h = \log(g_h/p_h).$$

其中 x, y 是中心坐标. 这里涉及到一些优化目标函数的变量的" 变换" 问题, 比如这里若 $g_w - p_w = 1 + \epsilon$, 则 $d_w = \epsilon$. 如此似乎可以将其替换为 $d_w = g_w/p_w - 1$, 实际上这样的效果可能还是不及 log, 因为一开始它就将变化幅度压缩了. 另外, 相对位置的归一化差异是关键, 比如人脸识别中,landmarks 的优化, 同常损失函数为欧氏距离, 如何对变量进行处理? 若是先预测 bbox, 再预测 landmarks, 可以相对于 bbox 的中心坐标做变换, 若直接预测呢?

1.5.2 SSDHead

ssd 结构的检测网络，目前已有 ssd300,ssd512, 结构细节参考1.3. 从配置文件中可有看到, 它没有 neck, 因层级结构在 backbone 实现.

ssdhead 继承自 anchorhead, 主要功能为处理多层级特征上的 anchor 构造和 target 标定与筛选, 基本的 feauturemap 上的 acnhor 生成由 mmdet.core.anchor 中的 AnchorGenerator 完成, 优化目标 anchor 由 anhor_target 完成. ssdhead 中 forward 前向返回各层级对应的类别分数和坐标信息,loss 函数则得到对应的损失函数, 以字典的形式返回, 最终求导时, 汇总成一个值, 同时也能计算各个部分损失函数的均值, 方差，方便优化,debug.

此处的难点在于 anchor 的设定和 target 的标定, 筛选. 现就 anchor 这一块细说如下:

anchor 基本介绍: anchor 设计和 caffe ssd anchor 设计一致, 假设 min_size 为 a, max_size 为 b, 则先生成 ratio 为 1, 宽度和高度为 (a, a), $(\sqrt{ab}, \sqrt{ab})$ 的两个 anchor, ratio 为 $2, 1/2, 3, 1/3$ 则分别生成宽度和高度为 $(a * \sqrt{ratio}, a/\sqrt{ratio})$ 的 anchor, mmdetection 中必须设定每一层的 min_size, max_size, 因此 ratios 为 [2] 则对应 4 个 anchor, ratios 为 $[2, 3]$ 则对应 6 个 anchor.

在 init() 函数中, 先生成 min_size, max_size, 注意它这里是必须要指定 max_size(和 caffe SSD 不同, 无法生成奇数个 anchor), 确保 len(min_size)=

len(max_size), 调用 AnchorGenerator() 类生成了 base_anchors, 数量是 6 或者 10, 使用 indices 操作从 6 个 anchor 里选择 $(0, 3, 1, 2)$ 或者从 10 个 anchor 里选择 $(0, 5, 1, 2, 3, 4)\rightarrow$ 最终生成 4 个或者 6 个 anchor. 于在多个 feature map 上生成 anchor, 因此使用了一个 for 循环操作, 将 anchor_generator 放入到 anchor_generatos[] 中.

AnchorGenerator 类, init() 函数需要如下参数:

- base_size: 即设置的 min_size
- scales: 是 $(1, \sqrt{max\_size/min\_size})$, 用来生成 ratio 为 1 的两个 anchor
- ratios: 是 $(1, 2, 1/2)$ 或者 $(1, 2, 1/2, 3, 1/3)$
- ctr: ctr 由 stride 生成, 是 anchor 的中心坐标, $(\frac{stride-1}{2}, \frac{stride-1}{2})$ 在 gen_base_anchor() 函数里, 使用上面的参数来计算 base_anchor, 计算流程如下:
  - 根据 ratios 来计算 h_ratios 和 w_ratios, 即上面所述的 $(1/\sqrt{ratios}, \sqrt{ratios})$.
  - 根据 scales 来计算 base_size, 一共有 2 个分别是

    $$(min\_size, \sqrt{min\_size * max\_size}) = min\_size * scales$$

  - 计算 anchors 的宽度和高度, 只以宽度举例: $w = base\_size * w\_ratios$, 以 ratios 是 $(1, 2, 1/2)$ 举例, base_size shape 为 $(2, 1)$, w_ratios shape 为 $(1, 3)$, 计算出的 w 是 $(2, 3)$ 一共生成了 6 个 anchor, 如果 ratios 是 $(1, 2, 1/2, 3, 1/3)$, 则生成 10 个 anchor (此处 anchor 数量和标准 ssd anchor 数量不一致 $\rightarrow$ 再筛选 (即 ssd_head.py 中使用 indices 操作进行筛选))

1.5.3 RetinaHead

后补。

1.5.4 GuidedHead

guided_anchor_head.py, ga_retina_head.py

1.5.5 FCOSHead

若其他人补。

1.6 Losses

1.6.1 基本认识

Fcoalloss. 想像一下特征层上的锚框, 远离 gt 的必然占绝大多数, 围绕在 gt 周围的 bbox 有模棱两可的 (正负), 故有正负难易之分.FcoalLoss 是为解决难易样本的不平衡问题. 公式如下:

$$FL(p_t) = -\alpha_t(1-p_t)^{\gamma}log(p_t), 其中 p_t = \begin{cases} p & if y = 1 \\ 1-p & else. \end{cases} \tag{1}$$

α_t 缓解样本不平衡现象,$(1-p_t)^{\gamma}$ 为降低易分样本的损失值 (考虑到易分样本比例高). 公式可以这样理解: 对于正样本分对了, 则 $p_t -> 1$,$(1-p_t)^{\gamma}$ 有减缓效果, 当分错了, 即 $p_t << 0.5$, 则 $1-p_t >> 0.5$, 效果和 CrossEntropy 没差, 对于负样本也是如此. 因此起到压缩图 6 中的左图右端, 右图左端的效果.

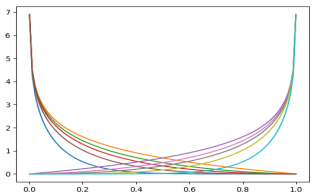


图 8: FcoalLoss

GHM. GHM 算是对 Fcoal 改进, 作者统计了样本的梯度信息, 提出梯度均衡机制, 让各种难度类型的样本有均衡的累计贡献. 具体细节看论文即可.

IoULoss. BoundedIoULoss.

1.6.2 实现解析

mmdet.models.losses 里面实现了所有损失函数。

losses.utils 有两个基本函数，weight_reduce_loss 和 weighted_loss，前者将返回的损失向量点乘一权重向量，再分别求和或者算平均值。后者为

一装饰器，装饰你自己定义的损失函数，实现带权重效果。具体为先算自定义 loss_func，然后再 weight_reduce_loss 一下。其中 weighted_loss 中的 @functools.wraps 装饰的好处是返回的函数保持被装饰函数名字。接下来逐一说明各个损失函数。

CrossEntropyLoss，其理论来源有两点，信息论或极大似然估计，可参考花书。pytorch 官方文档给的公式，明显看出是极大似然的写法。注意实际运算是张 (向) 量的形式。$\text{loss}(x,\ \text{class}\ ) = -\log\left(\frac{\exp(x[\ \text{class}\ ])}{\sum_j \exp(x[j])}\right) = -x[\ \text{class}\ ] + \log\left(\sum_j \exp(x[j])\right)$，增加权重因子为

$$\text{loss}(x,\ \text{class}\ ) = weight[\ \text{class}\ ]\left(-x[\ \text{class}\ ] + \log\left(\sum_j \exp(x[j])\right)\right).$$

在 cross_entropy_loss.py 中实现了三种交叉熵损失函数，binary_cross_entropy，mask_cross_entrop 以及 cross_entropy，前两者相同点在于将标签扩展成 one hot 形式然后调用 F.binary_cross_entropy_with_logits 函数，不同点为 mask 没有乘权重向量，cross_entropy 调用 F.cross_entropy，并被 weight_reduce_loss 了。需要注意的点是，最终的 CrossEntropyLoss 中的 loss_weight 参数是多任务中此 Loss 的权重。而 weight_reduce_loss 中的 weight 是各类别的权重。可以说前者和样本平衡有关，后者和任务平衡有关。

SmoothL1Loss 来源于 FastRCNN，用于解决方框回归的不稳定问题 (也许是优化速度)，由 L2，L1 演变而来。

$$L_1 = \begin{array}{ll} 0.5x^2, & |x| < 1 \\ |x| - 0.5, & x < -1 \text{ or } x > 1 \end{array}$$

三者的关系，画个图其意自明：L2 两边变化太快，对不好优化的 anchor 不友好 (离群点)，L1 0 点不可导，Smooth 两边较为平缓。在 smooth_l1_loss.py 中，smooth_l1_loss 被上述 weighted_loss 装饰，实现代码比较简单:

```
diff = torch.abs(pred - target)
loss = torch.where(diff < beta, 0.5 * diff * diff / beta,
          diff - 0.5 * beta)
```

注意这里的 where 实现了分段函数，所以其含义自明。

BalancedL1Loss 来自 Libra R-CNN，为 SmoothL1 的改进。改进函数为

$$L_b(x) = \begin{cases} \frac{\alpha}{b}(b|x|+1)\ln(b|x|+1) - \alpha|x| & \text{if } |x| < 1 \\ \gamma|x| + C & \text{otherwise} \end{cases}$$

其梯度为：

$$\frac{\partial L_b}{\partial x} = \begin{cases} \alpha \ln(b|x|+1) & \text{if } |x|<1 \\ \gamma & \text{otherwise} \end{cases}$$

其中 $b = e^{\frac{\gamma}{\alpha}}$, 使得函数连续。容易得到在 $|x|<1$ 附近的梯度比 SmoothL1 要大。具体分析可参考BalancedL1Loss，代码和 SmoothL1 一致，故析略。

Fcoalloss 原理见上节,其扩展有CenterNet,GHM 系列。因 _sigmoid_focal_loss 由 cpp,cu 实现，故略 (可看看 debug 版本)。

GHMLoss 有 GHMC, GHMR，分别作用于分类，回归。原理查看论文或者参考GHM，其主要思想是简单和特难的样本均进行抑制，也算是 FocalLoss 的改进，析码略。

剩下的 IoULoss,BoundedIoULoss,GIoULoss 等以 iou 的方式进行 Box 回归，含义自现。三者的细致分析 (各自优缺点及关系) 看原始论文。代码上根据如下公式，即可容易写出。Giou:

$$\begin{aligned} IoU &= \frac{|A\cap B|}{|A\cup B|} \\ GIoU &= IoU - \frac{|C\backslash(A\cup B)|}{|C|} \end{aligned}$$

其中 C 为 A, B 的闭包。在 iou_loss.py 中，最终的 iou_loss 为 $-log(ious)$, giou_loss 为 $1-gious$, boundedloss 和 SmoothL1Loss 差不多，只是此时被求的变量 (dx, dy, dw, dh) 的编码方式变了。具体关系如下：

论文中的分段函数为 Huber loss $L_\tau(z) = \begin{cases} \frac{1}{2}z^2 & |z|<\tau \\ \tau|z| - \frac{1}{2}\tau^2 & \text{otherwise} \end{cases}$

和 SmoothL1 类似，但代码用的是带了一个 β 因子的 SmoothL1。

分量损失为 $\text{cost}_{\text{i}} = 2L_1\left(1-\text{IoU}_{\text{B}}\left(i, b_t\right)\right), i \in \{x, y, w, h\}$. 原始的 $L1$(见 FasterRCNN) 定义为

$$\begin{aligned} \text{cost}_x &= L_1\left(\frac{\Delta x}{w_s}\right) \\ \text{cost}_{\text{w}} &= L_1\left(\ln\left(\frac{w}{w_t}\right)\right) \end{aligned}$$

这里使用的 IoU_{B} 定义为:

$$\begin{aligned} \text{IoU}_{\text{B}}\left(x, b_t\right) &= \max\left(0, \frac{w_t - 2|\Delta x|}{w_t + 2|\Delta x|}\right) \\ \text{IoU}_{\text{B}}\left(w, b_t\right) &= \min\left(\frac{w}{w_t}, \frac{w_t}{w}\right) \end{aligned} .$$

最终分别算出 x, y, w, h 的各自 IoU_B" 损失"，作为最终损失的分量，带入 SmoothL1 分段函数中即可。

以上的代码中出现了 new_full,nonzero,numel,expand,where 等常用函数。

1.7 Detectors

这节主要分析 maskrcnn 和 reppoints,retinaFace 三个算法.

1.7.1 maskrcnn

以配置文件 mask_rcnn_r50_fpn_1x.py 为例说说 twao_stage 的实现过程. 配合 two_stage 的 forward_train() 函数和配置文件, 即可.

首先 backbone 为 resnet50，(resnet 系列结构参见3), 其以 tuple 形式返回 4 个 stage 的特征图, 片段代码如下:

```
1
2 outs = []
3 for i, layer_name in enumerate(self.res_layers):
4   res_layer = getattr(self, layer_name)
5   x = res_layer(x)
6   if i in self.out_indices:
7     outs.append(x)
```

然后 neck 为 fpn, 结构参见6, fpn 根据 config 中的 out_indices 取出以 resnet50 输出的对应 stage, 分别构造输出 channel 维度统一的卷积算子, 然后按照6所示融合方式进行不同尺度的特征融合, 以元组形式输出结果. 在配置信息里有一条 num_outs=5, 是为 mask-rcnn 在最顶层特征增加的最大池化特征输出. 以上两块为提取特征, 被 extract_feat 整合在一块,

紧接着 forward_train 中包含了剩下的所有流程.

$$rpn\_head \rightarrow rpn\_head.loss \rightarrow rpn\_head.get\_bboxes \rightarrow assign \rightarrow sample \rightarrow bbox\_roi\_extractor \rightarrow bbox\_head \rightarrow bbox\_head.get\_target. \rightarrow bbox\_head.loss \rightarrow mask\_roi\_extractor \rightarrow mask\_head \rightarrow mask\_head.get\_target$$

这里梳理一下部分函数.

候选框层 RPN,RPNHead 继承 AnchorHead, 它的几个核心操作都在 anchor_head.py 中实现, 主要包括 get_anchors, anchor_target 见1.5.1, 函数 get_bboxes 结合配置参数从 rpn 前向得到的 2 分类和位置预测结果中筛选出最终的 proposals.

get_bboxes 中先通过 self.anchor_generators[i].grid_anchors() 这个函数取到所有的 anchor_boxs, 再通过 self.get_bboxes_single() 根据 rpn 前向的结果选出候选框, 在 self.get_bboxes_single() 中，先在每个尺度上取 2000(配置) 个 anchor 出来，concat 到一起作为该图像的 anchor，对这些 anchor boxs 作 nms(thr=0.7) 就得到了所需的候选框. 需注意预测的 bbox 是对数化了的，在做 iou 计算之前需用 delta2bbox() 函数进行逆变换.bbox_head 中的 bbox2roi 类似.

得到的候选框最终由配置中 train_cfg 的 rcnn.assigner, rcnn.sampler 进行标定和筛选, 保持正负样本平衡和框的质量, 方便优化.

MaxIoUAssigner:

1. 所有候选框置-1
2. 将与所有 gtbbox 的 iou 小于 neg_iou_thr 置 0
3. iou 大于 pos_iou_thr 的将其匹配
4. 为了避免标定框无训练目标, 将 gtbbox 匹配于与它 iou 最近的 bbox(会导致部分正样本的匹配 iou 值很小).

```
# 交并比矩阵(n,m), gt=n, bboxes=m
overlaps = bbox_overlaps(gt_bboxes, bboxes)
# 每个bbox和所有gt的最大交并比,(m,)
max_overlaps, argmax_overlaps = overlaps.max(dim=0)
# 每个gt和所有bbox的最大交并比
gt_max_overlaps, gt_argmax_overlaps = overlaps.max(dim=1)
# 1将所有bbox赋值为-1,注意new_full操作
assigned_gt_inds = overlaps.new_full(
    (num_bboxes, ), -1, dtype=torch.long)
# 2交并比大于0同时小于负阈值的赋值为0
assigned_gt_inds[(max_overlaps >= 0)
            & (max_overlaps < self.neg_iou_thr)] = 0
# 将与gt交并比大于正阈值的赋值为1(可能没有)
pos_inds = max_overlaps >= self.pos_iou_thr
assigned_gt_inds[pos_inds] = argmax_overlaps[pos_inds] + 1
# 保证每个gt至少对应一个bbox
# 遍历gt,将与gt最近(max(iou))的bbox,将gt的label赋值给此bbox

for i in range(num_gts):
    if gt_max_overlaps[i] >= self.min_pos_iou:
      # 此判断较迷
      max_iou_inds = overlaps[i, :] == gt_max_overlaps[i]
      assigned_gt_inds[max_iou_inds] = i + 1
      # 与gt最大iou的bbox 赋值为i+1
```

RandomSampler, 保持设定的平衡比例, 随机采样.

然后通过 SingleRoIExtractor(roi_extractors/single_level.py) 统一 RoI Align 四个尺度且大小不同的的 proposals, 使其大小为 7*7(bbox) 或 14*14 (mask). 配置信息 rpn_head 中的 anchor_strides 为 5 个尺度, 包含了 fpn 额外加入的最大池化层, 而 bbox_roi_extractor 的 featmap_strides 却只包含四个尺度, 表明只需对前四层进行 align. 最终送入 bbox head 和 mask head 做第二次优化 (two stage).

RoIAlign 在 ops 中, 经 cuda 加速, 详解待后. 其中 roi_extractors 中的特征层级映射函数如下:

```
def map_roi_levels(self, rois, num_levels):
"""Map rois to corresponding feature levels by scales.
 self.finest_scale = 56, 映射到0级的阈值
$(0, 56, 56*2, 56*4, \infty) \rightarrow (0, 1, 2, 3)$
bbox2roi变换后的rois

Returns:
  Tensor: Level index (0-based) of each RoI, shape (k, )
  因不同层级对应不同的ROIAlign
"""
scale = torch.sqrt(
  (rois[:, 3] - rois[:, 1] + 1) * (rois[:, 4] - rois[:, 2] + 1))
target_lvls = torch.floor(torch.log2(scale / self.finest_scale + 1e-6))
target_lvls = target_lvls.clamp(min=0, max=num_levels - 1).long()
# 这个变换在原始论文中有.
return target_lvls
```

1.7.2 RepPoints

利用 DCN 特性, 实现了检测的结构点表示方式的优化.

1.7.3 RetinaFace

再说，和 ssd 重复较大。

第二节　数据处理

2.1　检测分割数据

看看配置文件，数据相关的有 data dict，里面包含了 train,val,test 的路径信息，用于数据类初始化。然后就是 pipeline，将各个函数及对应参数以字典形式放到列表里面，是对 pytorch 原装的 transforms+compose，在检测，分割相关数据上的一次封装，使得形式更加统一。

从 builder.py 中 build_dataset 函数能清晰的看到，构建数据有三种方式，ConcatDataset，RepeatDataset 和从注册器中提取。

其中 dataset_wrappers.py 中 ConcatDataset 和 RepeatDataset 意义自明。前者继承自 pytorch 原始的 ConcatDataset，将多个数据集整合到一起，将不同序列 (可参考容器的抽象基类) 的长度相加，___getitem___ 函数对应 index 替换一下。后者就是单个数据类 (序列) 的多次重复。就功能来说，前者提高数据丰富度，后者可解决数据太少使得 loading 时间长的问题。而与注册相关的，被注册的数据类在 datasets 下大家熟知的数据类型的 py 文件中。基类为 custom.py 中的 CustomDataset，coco 继承自它，cityscapes 继承自 coco，xml_style 的 XMLDataset 继承 CustomDataset，然后 wider_face，voc 均继承自 XMLDataset。因此这里先分析一下 CustomDataset。

CustomDataset 记录数据路径等信息，解析标注文件，将每一张图的所有信息以字典作为数据结构存在 results 中。然后进入 pipeline，也即数据增强相关操作。从下面代码可以清晰的看到。这里数据结构的选取需要注意一下，字典结构，在数据增强库 albu 中也是如此处理，因此可以快速替换 albu 中的算法。另外每个数据类增加了各自的 evaluate 函数。evaluate 基础函数在 mmdet.core.evaluation 中，后做补充。

```
  self.pipeline = Compose(pipeline)
  # Compose是实现了___call___方法的类，其作用是使实例能够像函数一样被调用，同
    时不影响实例本身的生命周期
def pre_pipeline(self, results):
  # 扩展字典信息
  results['img_prefix'] = self.img_prefix
  results['seg_prefix'] = self.seg_prefix
  results['proposal_file'] = self.proposal_file
  results['bbox_fields'] = []
  results['mask_fields'] = []
  results['seg_fields'] = []

```

```
def prepare_train_img(self, idx):
  img_info = self.img_infos[idx]
  ann_info = self.get_ann_info(idx)
  # 基本信息，初始化字典
  results = dict(img_info=img_info, ann_info=ann_info)
  if self.proposals is not None:
    results['proposals'] = self.proposals[idx]
  self.pre_pipeline(results)
  return self.pipeline(results)    # 数据增强

def __getitem__(self, idx):
  if self.test_mode:
    return self.prepare_test_img(idx)
  while True:
    data = self.prepare_train_img(idx)
    if data is None:
      idx = self._rand_another(idx)
      continue
    return data
```

mmdet 的数据处理，字典结构，pipeline，evaluate 是三个关键部分。其他所有类的文件解析部分，数据筛选等，看看即可。因为我们知道，pytorch 读取数据，是将序列转化为迭代器后进行 io 操作的。所以在 dataset 下除了 pipelines 外还有 loader 文件夹，里面实现了分组，分布式分组采样方法，以及调用了 mmcv 中的 collate 函数，且 build_dataloader 封装的 DataLoader 最后在 train_detector 中被调用，这部分将在后面补充，这里说说 pipelines。所有 pipelines 的注册函数在 __init__.py 中可见。

返回 maskrcnn 的配置文件，可以看到，训练和测试的不同，LoadAnnotations，MultiScaleFlipAug，DefaultFormatBundle 和 Collect。虽然测试没有 LoadAnnotations，实际上根据前面的 CustomDataset 可知道，它仍然需要标注文件。这和 inference 的 pipeline 是不同的。

```
# 序列中的dict可以随意删减，增加，所谓数据增强调参
train_pipeline = [
    dict(type='LoadImageFromFile'),
    dict(type='LoadAnnotations', with_bbox=True, with_mask=True),
    dict(type='Resize', img_scale=(1333, 800), keep_ratio=True),
    dict(type='RandomFlip', flip_ratio=0.5),
    dict(type='Normalize', **img_norm_cfg),
    dict(type='Pad', size_divisor=32),
    dict(type='DefaultFormatBundle'),
    dict(type='Collect', keys=['img', 'gt_bboxes', 'gt_labels', 'gt_masks'])
    ,
```

```
]

test_pipeline = [
    dict(type='LoadImageFromFile'),
    dict(
        type='MultiScaleFlipAug',
        img_scale=(1333, 800),
        flip=False,
        transforms=[
            dict(type='Resize', keep_ratio=True),
            dict(type='RandomFlip'),
            dict(type='Normalize', **img_norm_cfg),
            dict(type='Pad', size_divisor=32),
            dict(type='ImageToTensor', keys=['img']),
            dict(type='Collect', keys=['img']),
        ])
]
```

这些所有操作被 Compose 串联起来 (列表是序列结构)。

```
@PIPELINES.register_module
class Compose(object):

  def __init__(self, transforms):
    assert isinstance(transforms, collections.abc.Sequence)
    self.transforms = []
    for transform in transforms:
      if isinstance(transform, dict):
        transform = build_from_cfg(transform, PIPELINES)
        self.transforms.append(transform)
      elif callable(transform):
        self.transforms.append(transform)
      else:
        raise TypeError('transform must be callable or a dict')

  def __call__(self, data):
    for t in self.transforms:
      data = t(data)
      if data is None:
        return None
    return data
```

上面代码能清晰的看到，配置文件中 pipeline 中的字典传入 build_from_cfg 函数，逐一实现了各个增强类。扩展的增强类均需实现 __call__ 方法，这和 pytorch 原始方法是一致的。

```
def build_from_cfg(cfg, registry, default_args=None):
```

```
  assert isinstance(cfg, dict) and 'type' in cfg
  assert isinstance(default_args, dict) or default_args is None
  args = cfg.copy()
  obj_type = args.pop('type')
  if mmcv.is_str(obj_type):
    # 从注册类中拿出obj_type类
    obj_cls = registry.get(obj_type)
    if obj_cls is None:
      raise KeyError('{} is not in the {} registry'.format(
        obj_type, registry.name))
  elif inspect.isclass(obj_type):
    obj_cls = obj_type
  else:
    raise TypeError('type must be a str or valid type, but got {}'.format(
      type(obj_type)))
  if default_args is not None:
    # 增加一些新的参数
    for name, value in default_args.items():
      args.setdefault(name, value)
  return obj_cls(**args)    # **args是将字典解析成位置参数(k=v)。
```

有了以上认识，重新梳理一下 pipelines 的逻辑，由三部分组成，load，transforms，和 format。load 相关的 LoadImageFromFile，LoadAnnotations 都是字典 results 进去，字典 results 出来。具体代码看下便知，LoadImageFromFile 增加了'filename'，'img'，'img_shape'，'ori_shape','pad_shape','scale_factor','img_norm_cfg' 字段。其中 img 是 numpy 格式。LoadAnnotations 从 results['ann_info'] 中解析出 bboxs,masks,labels 等信息。注意 coco 格式的原始解析来自 pycocotools，包括其评估方法，这里关键是字典结构。transforms 中的类作用于字典的 values，也即数据增强。format 中的 DefaultFormatBundle 是将数据转成 mmcv 扩展的容器类格式 DataContainer。另外 Collect 会根据不同任务的不同配置，从 results 中选取只含 keys 的信息生成新的字典，具体看下该类帮助文档。这里看一下从 numpy 转成 tensor 的代码:

```
def to_tensor(data):
    """Convert objects of various python types to :obj:`torch.Tensor`.

    Supported types are: :class:`numpy.ndarray`, :class:`torch.Tensor`,
    :class:`Sequence`, :class:`int` and :class:`float`.
    """
    if isinstance(data, torch.Tensor):
        return data
    elif isinstance(data, np.ndarray):
```

```
        return torch.from_numpy(data)
    elif isinstance(data, Sequence) and not mmcv.is_str(data):
        return torch.tensor(data)
    elif isinstance(data, int):
        return torch.LongTensor([data])
    elif isinstance(data, float):
        return torch.FloatTensor([data])
    else:
        raise TypeError('type {} cannot be converted to tensor.'.format(
      type(data)))
  以上代码告诉我们，基本数据类型，需掌握。
```

那么 DataContainer 是什么呢？它是对 tensor 的封装，将 results 中的 tensor 转成 DataContainer 格式，实际上只是增加了几个 property 函数，cpu_only，stack，padding_value，pad_dims，其含义自明，以及 size，dim 用来获取数据的维度，形状信息。考虑到序列数据在进入 DataLoader 时，需要以 batch 方式进入模型，那么通常的 collate_fn 会要求 tensor 数据的形状一致。但是这样不是很方便，于是有了 DataContainer。它可以做到载入 GPU 的数据可以保持统一 shape，并被 stack，也可以不 stack，也可以保持原样，或者在非 batch 维度上做 pad。当然这个也要对 default_collate 进行改造，mmcv 在 parallel.collate 中实现了这个。

也许可以看看通常的 collate_fn，它是 DataLoader 中将序列 dataset 组织成 batch 大小的函数。

```
def collate_fn_1(batch):
  # 这是默认的，明显batch中包含相同形状的img\_tensor和label
  return tuple(zip(*batch))

def coco_collate_2(batch):
  # 传入的batch数据是被albu增强后的(字典结构)
    imgs = [s['image'] for s in batch]    # tensor, h, w, c->c, h, w ,
     handle at transform in ___getitem___
    annots = [s['bboxes'] for s in batch]
    labels = [s['category_id'] for s in batch]

  # 以当前batch中图片annot数量的最大值作为标记数据的第二维度值，空出的就补-1
     。
    max_num_annots = max(len(annot) for annot in annots)
    annot_padded = np.ones((len(annots), max_num_annots, 5))*-1

    if max_num_annots > 0:
        for idx, (annot, lab) in enumerate(zip(annots, labels)):
            if len(annot) > 0:
```

```
                annot_padded[idx, :len(annot), :4] = annot
        # 不同模型，损失值计算可能不同，这里ssd结构需要改为xyxy格式并且要做尺度归一化
        # 这一步完全可以放到\_\_getitem\_\_中去，只是albu的格式需求问题。
                annot_padded[idx, :len(annot), 2] += annot_padded[idx, :len(annot), 0]    # xywh-->x1,y1,x2,y2 for general box,ssd target assigner
                annot_padded[idx, :len(annot), 3] += annot_padded[idx, :len(annot), 1]    # contains padded -1 label
                annot_padded[idx, :len(annot), :] /= 640    # priorbox for ssd primary target assinger
                annot_padded[idx, :len(annot), 4] = lab
    return torch.stack(imgs, 0), torch.FloatTensor(annot_padded)

def detection_collate_3(batch):
    targets = []
    imgs = []
    for _, sample in enumerate(batch):
        for _, img_anno in enumerate(sample):
            if torch.is_tensor(img_anno):
                imgs.append(img_anno)
            elif isinstance(img_anno, np.ndarray):
                annos = torch.from_numpy(img_anno).float()
                targets.append(annos)
    return torch.stack(imgs, 0), targets    # 做了stack， DataContainer可以不做stack
```

以上就是数据处理的相关内容。最后再用 DataLoader 封装拆成迭代器，其相关细节，sampler 等就略去了。

```
data_loader = DataLoader(
  dataset,
  batch_size=batch_size,
  sampler=sampler,
  num_workers=num_workers,
  collate_fn=partial(collate, samples_per_gpu=imgs_per_gpu),
  pin_memory=False,
  worker_init_fn=init_fn,
  **kwargs)
```

第三节 FP16

模型预测加速等.

第四节　训练 pipeline

4.1　训练逻辑

图见2注意它的四个层级. 主要查看 api/train.py, mmcv 中的 runner 相关文件. 主要两个类:Runner 和 Hook Runner 将模型, 批处理函数 batch_processor, 优化器作为基本属性, 是为训练过程中记录相关节点信息, 这些信息均被记录在 mode,_hooks,_epoch,_iter,_inner_iter,_max_epochs,_max_iters 中. 从而实现训练过程中插入不同的操作, 也即各种 hook. 理清训练流程只需看 Runner 的成员函数 run. 在 run 里会根据 mode 按配置 (workflow)epoch 循环调用 train 和 val 函数, 跑完所有的 epoch. 其中 train 代码如下:

```
def train(self, data_loader, **kwargs):
  self.model.train()
  self.mode = 'train'    # 改变模式
  self.data_loader = data_loader
  self._max_iters = self._max_epochs * len(data_loader)    # 最大batch循环次数
  self.call_hook('before_train_epoch')    # 根据名字获取hook对象函数
  for i, data_batch in enumerate(data_loader):
    self._inner_iter = i    # 记录训练迭代轮数
    self.call_hook('before_train_iter')    # 一个batch前向开始
    outputs = self.batch_processor(
      self.model, data_batch, train_mode=True, **kwargs)
    self.outputs = outputs
    self.call_hook('after_train_iter')    # 一个batch前向结束
    self._iter += 1    # 方便resume时,知道从哪一轮开始优化

  self.call_hook('after_train_epoch')    # 一个epoch结束
  self._epoch += 1    # 记录训练epoch状态,方便resume
```

上面让人困惑的是 hook 函数, hook 函数继承自 mmcv 的 Hook 类, 其默认了 6+8+4 个函数, 也即2所示的 6 个层级节点, 外加 2*4 个区分 train 和 val 的节点记录函数, 以及 4 个边界检查函数. 从 train.py 中容易看出, 在训练之前, 已经将需要的 hook 函数注册到 Runner 的 self._hook 中了, 包括从配置文件解析的优化器, 学习率调整函数, 模型保存, 一个 batch 的时间记录等 (注册 hook 算子在 self._hook 中按优先级升序排列). 于是只需理解 call_hook 函数即可.

```
def call_hook(self, fn_name):
  for hook in self._hooks:
```

```
    getattr(hook, fn_name)(self)
```

如上看出, 在训练的不同节点, 将从注册列表中调用实现了该节点函数的类成员函数. 比如

```
class OptimizerHook(Hook):

    def __init__(self, grad_clip=None):
        self.grad_clip = grad_clip

    def clip_grads(self, params):
        clip_grad.clip_grad_norm_(
            filter(lambda p: p.requires_grad, params), **self.grad_clip)

    def after_train_iter(self, runner):
        runner.optimizer.zero_grad()
        runner.outputs['loss'].backward()
        if self.grad_clip is not None:
            self.clip_grads(runner.model.parameters())
        runner.optimizer.step()
```

将在每个 train_iter 后实现反向传播和参数更新.

学习率优化相对复杂一点, 其基类 LrUpdaterHook, 实现了 before_run, before_train_epoch, before_train_iter 三个 hook 函数, 意义自明. 这里选一个余弦式变化, 稍作说明:

```
class CosineLrUpdaterHook(LrUpdaterHook):

    def __init__(self, target_lr=0, **kwargs):
        self.target_lr = target_lr
        super(CosineLrUpdaterHook, self).__init__(**kwargs)

    def get_lr(self, runner, base_lr):
        if self.by_epoch:
            progress = runner.epoch
            max_progress = runner.max_epochs
        else:
            progress = runner.iter
            max_progress = runner.max_iters
        return self.target_lr + 0.5 * (base_lr - self.target_lr) * \
      (1 + cos(pi * (progress / max_progress)))
```

从 get_lr 可以看到, 学习率变换周期有两种,epoch->max_epoch, 或者更大的 iter->max_iter, 后者表明一个 epoch 内不同 batch 的学习率可以不同, 因为没有什么理论, 所有这两种方式都行. 其中 base_lr 为初始学习率,target_lr 为学习率衰减的上界, 而当前学习率正如函数的返回表达式.

第五节　更改模型

5.1　增加模块

本身就包含了很多更改配置文件。

5.2　模型瘦身

替换 backbone。

第六节　抽离模型

6.1　retinanet_resnet18

主要注意点。

第七节　新增模型

7.1　centernet

直接抽离一个就行了。

第八节　numpy,torch 某些基础函数

8.1　损失函数部分

torch 相关:new_full,nonzero,numel,expand,where 见损失函数一节。

8.2　模型实现部分

待补。

8.3　数据类部分

待补。

第九节　计划

0. datasets
1. 更改模型
2. 抽离模型
3. 新增模型 centernet
6. ops 中的 ROIAlign,DCN
7. 检测模型的简略综述

第十节　检测模型的简略综述

检测算法是由一些基本的组件组合而成. 这也是 mmdection 出现的原因. 这章节会在不同算法系列中总结一些基础组件, 最后再做个提取.

10.1　通用物体检测

按照物体编码方式可分为三类. 矩形编码 (anchor 机制), 结构点编码 (RepPoints) 和所有点编码 (mask).

10.1.1　Yolo 系列

yolo 相对孤立,v1 就不说了,v2,v3 基本思想还是来源于 resnet,fpn,ssd 等. 代码可参考 yolov3.

与 SSD 的不同之处:

- 根据数据聚类 9 个先验框, 分成 3 个尺度, 分别作为三个检测层的 base anchor.
- 因为网络结构只有卷积和池化, 所以可以做多尺度输入 320-608,steps=32.
- 基础网络 DarkNet19,53 较 resnet 更为轻量, 主要是 $1*1$ 卷积的大量使用.
- 多尺度的处理方式是在每个尺度上均计算一次检测 (yolo layer), 这和 ssd 合并起来做统一处理不同.

- 框回归编码不同 (相对于网格中心的偏移量, 物体由一个中心网格预测, 当物体重合度高时, 此假设不成立).
- 网络结构实现方式大多采用配置文件解析 (DarkNet).

10.1.2 SSD 系列

ssd 是第一个包含了几乎所有的检测组件的算法 (各种检测算法的所有组件集), 故能在后续的发展中多次被更改, 用于其他任务中, 比如 ctpn,textboxes++,faceboxes 等. 代码可参考:SSD-Tutorial手把手教你实现 ssd, 以及相关原理讲解. ssd.pytorch 最先看的 ssd 源码, 简洁完完整. maskrcnn-benchmark 风格版 ssd.

原始 SSD:

- 多尺度
  - 不同尺度的 feature map 上生成 anchor(比如:$300**2 -> 38**2*4 + 19**2*6 + 10**2*6 + 5**2*6 + 3**2*4 + 1**2*4$), 进行位置回归和类别判断.
  - 一个 $m*n$ 大小的 feature map, 若每个 cell(可以理解成物体的离散表示点) 分配 k 个 anchor, 则每个 cell 输出 $(c+4)*k*m*n$ 个预测值 (class, box relative offset).
  - 注意每个 cell 其实是一个向量, 长度为 channel of feature map, 可以理解成一个物体的某一部分 (或全部) 的向量表示或者整体表示的一部分.
  - 事实上这里完成了两个任务, 分类和位置回归, 所以 cell 向量可能具有分段表示功效 (这和权重共享是不矛盾的, 共享的权重可能就具有可分离性).
  - anchor 计算损失函数前的有效编码: 首先中心坐标在特征层上归一化 (等同于相对于原图的归一化), 尺度根据原图尺寸以及当前特征图相对原图的尺寸进行设计, 比如第一特征层相对于原图的 0.1, 然后计算相对偏移量以及坐标和尺度的" 等效" 处理，尺度求对数.
- 数据增强

  - DistortImage: 修改图像本身的 brightness, contrast, saturation, hue, reordering channels.
  - ExpandImage: 将 DistortImage 的图片用像素 0 进行扩展, 同时以黑边的左上角为原点计算 $[0,1]$ 的 bbox 的左上角和右下角两个点坐标.
  - BatchSampler: sampled_bboxes 的值是随机在 $[0,1]$ 上生成的 bbox，并且和某个 gt_bboxes 的 IOU 在 $[min,max]$ 之间
  - resize 到固定大小 $300*300$，label 也同时线性缩放.
  - 以 0.5 概率随机水平翻转, 或者 crop 等

- 样本平衡
  - 难例挖掘, 正负样本 1:3 等. 我的理解是制造更有效的优化空间 (anchor 是让优化空间变得合理).

- 损失函数
  - 利用 GT box 给个生成的 8732(可变)anchor 打标签, 筛选出有效优化对象, 计算分类和回归值.

- 后处理
  - NMS, Soft-NMS, OHEM

以上五部分均是以后论文的改进点. 比如特征提取基础结构, 采用其他有效的分类模型 resnet, 或者轻量级的 mobilenet 等, 或者替换在新的结构上替换一些结构组成算子, 卷积, 激活,BN 等操作. 比如多尺度的 FPN 类似思想, 融合不同特征层，这在一定程度上解决了重复框, 小物体问题 (将同一物体的不同尺度表达进行融合, 当然能减缓底层表达能力不足的现象). 这些有 FSSD,RSSD(没有必要都要去看, 检测类的文章, 理清基本组件, 花时间分析组件功能, 做实验验证想法, 就 ok 了) 等. 数据增强的方式各不相同, 主要是提高数据的丰富性, 增加模型的泛化能力, 这个属于工程问题, 基本方法都来源于传统的图像处理. 样本平衡的扩展可参考 mmdetection, 用制造更有效的优化空间来理解, 就可以随意发挥了.

具体案例:

FaceBoxes:

textboxes++:

10.1.3　Fast RCNN 系列

Mask R-CNN:

maskrcnn-benchmark能学习的东西都在这里面了 (仔细研读 3 遍). Mask R-CNN = Faster R-CNN with FCN on ROIs. 其主要流程参见1.7.1.ROI Align 原理: 去掉了图像下采样到特征图的坐标量化, 保持分数坐标, 同时 ROI 分格池化时, 对格子的坐标也取消量化, 从而减少了坐标的二度漂移. 若两次量化, 最坏的情况, 下采样 5 次, 会有近 64 的位置偏移, 这样会漏掉小目标. 最后的池化采用双线性插值, 原理就是每个点的像素值是其临近 4 个像素点的距离权重平均.

Cascade RCNN 发现只有 proposal 自身的阈值和训练器的训练阈值较为接近时, 训练器的性能才最好.

10.1.4　Anchor Free 系列

FCOS

10.2　总结