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

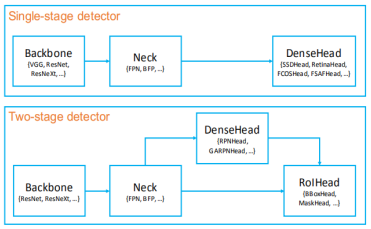


图 1: Framework

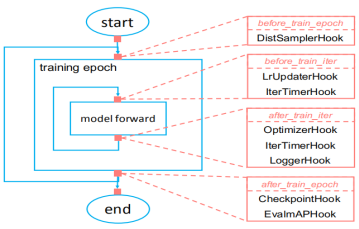


图 2: Trainning pipeline

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

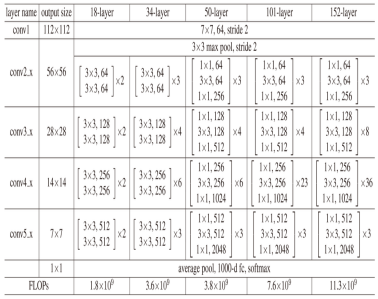
   2.9 @DETECTORS.register_module 装饰了完整的检测算法 (OneStage, TwoStage), 各个部件在其 init() 函数中实例化, 实现 2.5 的依次调用.
3. 最后是数据迭代器和训练 pipeline四.

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	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$
	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

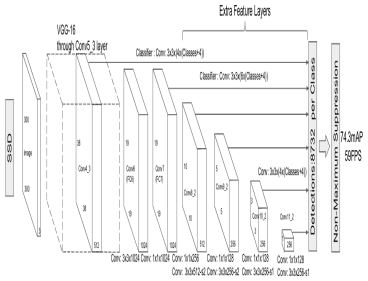


图 4: ssd

1.4 Necks

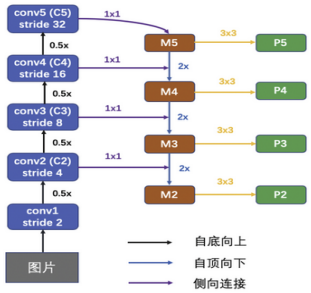


图 5: fpn

展开描述: 目标尺度和金字塔层级的关系, 这些层级对应的尺度与损失函数, 样本采样等的关系.

1.5 Heads

anchor_heads, bbox_heads, mask_heads 都将有 loss 的计算, 是核心点, 需详细描述.

1.5.1 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 → 再筛选 (即 ssd_head.py 中使用 indices 操作进行筛选))

1.6 Losses

CrossEntropyLoss, SmoothL1Loss, Fcoalloss, BalancedL1Loss, IoULoss, BoundedIoULoss, ArcLoss

1.7 Detectors

OneStage, TwoStage 和能改动的地方.

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5, 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5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(后续说明), 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).

RandomSampler, 保持设定的平衡比例, 随机采样. 其他采样待补.

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
```

第二节　数据处理

coco 数据等格式，多格式的转换

第三节　模型结构

OneStage，TwoStage, 其他结构的变体.

第四节　训练 pipeline

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

案例

第六节　新增模型

RetinaFace, BlazeFace, FaceBoxes 等

第七节　计划

1. 基本骨架网络说明,ssd-vgg, 将其和配置文件 ssd300,ssd500 中的 head 参数对齐
2. anchor target 通用的和具体到某算法
3. neck 层,fpn 的扩展和变体
4. 损失函数基本和具体到 maskrcnn head 部分
5. Detectors 中 maskrcnn 的 RPNHead,bboxhead, 中有关 get_anchors, anchor_target 函数部分
6. ops 中的 ROIAlign