mmdet 简略解析

Sisyphes

2020 年 5 月 24 日

目录

第一节　一些废话

拖拖拉拉，勉强走完，才发现，这才是开始。一眼望去，很多地方都比较粗糙，那些需要用实验丰富的地方，时间肯定是被偷了。虽然不排除存在对初学者有帮助的地方，但真正有帮助的或许是 mmsdet(目前不存在)，一个简单，轻量，高性能的检测库。

第二节　结构设计

19 年 7 月，Kai Chen 等人写了一篇文章 MMDetection, 介绍了他们在 mmdetection上的一些工作。包括 mmdetection 的设计逻辑，已实现的算法等。猜：Kai Chen 在不知道经历了一些什么之后，觉得对各种实现迥异的检测算法抽象一些公共的组件出来也许是一件不错的事。这里尝试对代码做一些简单的解析，见下。

组件设计:

- BackBone: 特征提取骨架网络,ResNet,ResneXt,ssd_vgg, hrnet 等。
- Neck: 连接骨架和头部. 多层级特征融合,FPN,BFP,PAFPN 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: 对特征图上的预选框做 pool 得到大小统一的 roi。
- RoIHead (BBoxHead/MaskHead): 在特征图上对 roi 做类别分类或位置回归等 (1.x)。
- ROIHead:bbox 或 mask 的 roi_extractor+head(2.0, 合并了 extractor 和 head)
- OneStage: BackBone + Neck + DenseHead
- TwoStage: BackBone + Neck + DenseHead + RoIExtractor + RoIHead : 1.x
- TwoStage: BackBone + Neck + DenseHead + RoIHead(2.0)

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和一些实用脚本

mmdet:

apis: 分布式环境设定 (1.x,2.0 移植到 mmcv), 推断, 测试, 训练基础代码

core: anchor 生成,bbox,mask 编解码, 变换, 标签锚定, 采样等, 模型评估, 加速, 优化器，后处理等

datasets:coco,voc 等数据类, 数据 pipelines 的统一格式, 数据增强，数据采样

models: 模型组件 (backbone,head,loss,neck)，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,masked_conv，focal_loss 等

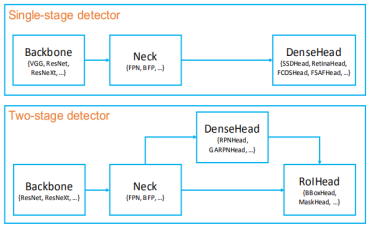


图 1: Framework

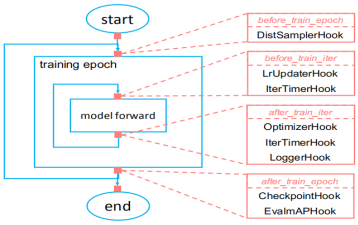


图 2: Trainning pipeline

2.1　总体逻辑

从 tools/train.py 中能看到整体可分如下 4 个步骤:

1. mmcv.Config.fromfile 从配置文件解析配置信息, 并做适当更新, 包括环境搜集，预加载模型文件, 分布式设置，日志记录等

2. mmdet.models 中的 build_detector 根据配置信息构造模型

2.1 build 系列函数调用 build_from_cfg 函数, 按 type 关键字从注册表中获取相应的对象, 对象的具名参数在注册文件中赋值。

2.2 registr.py 放置了模型的组件注册器。其中注册器的 register_module 成员函数是一个装饰器功能函数,在具体的类对象 A 头上装饰 @X.register

_module，并同时在 A 对象所在包的初始化文件中调用 A，即可将 A 保存到 registry.module_dict 中, 完成注册。
2.3 目前包含 BACKBONES,NECKS,ROI_EXTRACTORS,SHARED_HEADS,HEADS,LOSSES,DETECTORS 七个模型相关注册器，另外还有数据类，优化器等注册器。

3. build_dataset 根据配置信息获取数据类
3.1 coco，cityscapes，voc，wider_face 等数据 (数据类扩展见后续例子)。

4.train_detector 模型训练流程
4.1 数据 loader 化, 模型分布式化，优化器选取
4.2 进入 runner 训练流程 (来自 mmcv 库，采用 hook 方式，整合了 pytorch 训练流程)
4.2 训练 pipelines 可见2，具体细节见后续展开。

后续说说配置文件，注册机制和训练逻辑。

第三节　配置, 注册

3.1　配置类

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. 这里帖部分代码，以供学习。

```
class Config(object):
    ...
    @staticmethod
    def _file2dict(filename):
        filename = osp.abspath(osp.expanduser(filename))
        check_file_exist(filename)
        if filename.endswith('.py'):
            with tempfile.TemporaryDirectory() as temp_config_dir:
                shutil.copyfile(filename,
                                osp.join(temp_config_dir, '_tempconfig.py'))
                sys.path.insert(0, temp_config_dir)
                mod = import_module('_tempconfig')
                sys.path.pop(0)
```

```
14                cfg_dict = {
15                    name: value
16                    for name, value in mod.__dict__.items()
17                    if not name.startswith('__')
18                }
19                # delete imported module
20                del sys.modules['_tempconfig']
21            elif filename.endswith(('.yml', '.yaml', '.json')):
22                import mmcv
23                cfg_dict = mmcv.load(filename)
24            else:
25                raise IOError('Only py/yml/yaml/json type are supported now!')
26
27            cfg_text = filename + '\n'
28            with open(filename, 'r') as f:
29                cfg_text += f.read()
30            # 2.0新增的配置文件的组合继承
31            if '_base_' in cfg_dict:
32                cfg_dir = osp.dirname(filename)
33                base_filename = cfg_dict.pop('_base_')
34                base_filename = base_filename if isinstance(
35                    base_filename, list) else [base_filename]
36
37                cfg_dict_list = list()
38                cfg_text_list = list()
39                for f in base_filename:
40                    # 递归, 可搜索staticmethod and recursion
41                    # 静态方法调静态方法, 类方法调静态方法
42                    _cfg_dict, _cfg_text = Config._file2dict(osp.join(cfg_dir, f
    ))
43                    cfg_dict_list.append(_cfg_dict)
44                    cfg_text_list.append(_cfg_text)
45
46                base_cfg_dict = dict()
47                for c in cfg_dict_list:
48                    if len(base_cfg_dict.keys() & c.keys()) > 0:
49                        raise KeyError('Duplicate key is not allowed among bases
    ')
50                    base_cfg_dict.update(c)
51                # 合并
52                Config._merge_a_into_b(cfg_dict, base_cfg_dict)
53                cfg_dict = base_cfg_dict
54
55                # merge cfg_text
56                cfg_text_list.append(cfg_text)
57                cfg_text = '\n'.join(cfg_text_list)
```

```
58
59          return cfg_dict, cfg_text
60
61      ...
62      # 获取key值
63      def __getattr__(self, name):
64          return getattr(self._cfg_dict, name)
65      # 序列化
66      def __getitem__(self, name):
67          return self._cfg_dict.__getitem__(name)
68      # 将字典属性化主要用了__setattr__
69      def __setattr__(self, name, value):
70          if isinstance(value, dict):
71              value = ConfigDict(value)
72          self._cfg_dict.__setattr__(name, value)
73      # 更新key值
74      def __setitem__(self, name, value):
75          if isinstance(value, dict):
76              value = ConfigDict(value)
77          self._cfg_dict.__setitem__(name, value)
78      # 迭代器
79      def __iter__(self):
80          return iter(self._cfg_dict)
```

主要考虑点是自己怎么实现类似的东西，核心点就是 python 的基本魔法函数的应用，可同时参考 yacs。

3.2　注册器

把基本对象放到一个继承了字典的对象中，实现了对象的灵活管理。

```
1  import inspect
2  from functools import partial
3  import mmcv
4
5  class Registry(object):
6      # 2.0 放到mmcv中
7
8      def __init__(self, name):
9          self._name = name
10         self._module_dict = dict()
11     @property
12     def name(self):
13         return self._name
14     @property
15     def module_dict(self):
```

```
16          return self._module_dict
17
18      def get(self, key):
19          return self._module_dict.get(key, None)
20
21      def _register_module(self, module_class, force=False):
22          """Register a module.
23
24          Args:
25              module (:obj:`nn.Module`): Module to be registered.
26          """
27          if not inspect.isclass(module_class):
28              raise TypeError('module must be a class, but got {}'.format(
29                  type(module_class)))
30          module_name = module_class.__name__
31          if not force and module_name in self._module_dict:
32              raise KeyError('{} is already registered in {}'.format(
33                  module_name, self.name))
34          self._module_dict[module_name] = module_class   # 类名:类
35
36      def register_module(self, cls=None, force=False):
37          # 作为类cls的装饰器
38          if cls is None:
39              # partial函数(类)固定参数, 返回新对象,递归不是很清楚
40              return partial(self.register_module, force=force)
41          self._register_module(cls, force=force)   # 将cls装进当前Registry对
    象的中_module_dict
42          return cls    # 返回类
43
44  def build_from_cfg(cfg, registry, default_args=None):
45    assert isinstance(cfg, dict) and 'type' in cfg
46    assert isinstance(default_args, dict) or default_args is None
47    args = cfg.copy()
48    obj_type = args.pop('type')
49    if mmcv.is_str(obj_type):
50      # 从注册类中拿出obj_type类
51      obj_cls = registry.get(obj_type)
52      if obj_cls is None:
53        raise KeyError('{} is not in the {} registry'.format(
54          obj_type, registry.name))
55    elif inspect.isclass(obj_type):
56      obj_cls = obj_type
57    else:
58      raise TypeError('type must be a str or valid type, but got {}'.format(
59        type(obj_type)))
60    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)。
```

第四节　数据处理

数据处理可能是炼丹师接触最为密集的了，因为通常情况，除了数据的离线处理，写个数据类，就可以炼丹了。但本节主要涉及数据的在线处理，更进一步应该是检测分割数据的 pytorch 处理方式。虽然 mmdet 将常用的数据都实现了，而且也实现了中间通用数据格式，但，这和模型，损失函数，性能评估的实现也相关，比如你想把官网的 centernet 完整的改成 mmdet 风格，就能看到 (看起来没必要)。

4.1　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, 有 pipeline，将各个函数及对应参数以字典形式放到列表里，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，具体为把不同序列 (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 (见代码注释)。而被注册的数据类在 datasets 下一些熟知的数据名文件中。其中，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: 数据增强相关操作，代码如下：

```
  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
```

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，因此可以快速替换为 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。

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，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 (此处为 1.x 版本，2.0 版本将 loader 移植到了 builder.py 中)，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。

返回 maskrcnn 的配置文件 (1.x,2.0 看 base config)，可以看到训练和测试的不同之处：LoadAnnotations，MultiScaleFlipAug，DefaultFormatBundle 和 Collect。额外提示，虽然测试没有 LoadAnnotations，根据 CustomDataset 可知，它仍需标注文件，这和 inference 的 pipeline 不同，也即这里的 test 实为 evaluate。

```
# 序列中的dict可以随意删减，增加，属于数据增强调参内容
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']),
        ])
]
```

最后这些所有操作被 Compose 串联起来，代码如下：

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

  def __init__(self, transforms):
    assert isinstance(transforms, collections.abc.Sequence)  # 列表是序列结
    构
    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 (方法)。扩展的增强类均需实现 ___call___ 方法，这和 pytorch 原始方法是一致的。

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 (这个和模型损失函数，评估等相关，统一结构，使得代码统一)。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。

collate_fn 是 DataLoader 中将序列 dataset 组织成 batch 大小的函数，这里帖三个普通例子:

```
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 等暂略。

```
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)
```

第五节　训练流程

训练流程的包装过程大致如下:tools/train.py->apis/train.py->mmcv/runner.py->mmcv/hook.py(后面是分散的)，其中 runner 维护了数据信息，优化器，日志系统，训练 loop 中的各节点信息，模型保存，学习率等. 另外补

充一点, 以上包装过程, 在 mmdet 中无处不在, 包括 mmcv 的代码也是对日常频繁使用的函数进行了统一封装.

5.1　训练逻辑

图见2, 注意它的四个层级. 代码上, 主要查看 apis/train.py, mmcv 中的 runner 相关文件. 核心围绕 Runner,Hook 两个类. Runner 将模型, 批处理函数 batch_pro cessor, 优化器作为基本属性, 训练过程中与训练状态, 各节点相关的信息被记录在 mode,_hooks,_epoch,_iter,_inner_iter,_max_epochs, _max_iters 中, 这些信息维护了训练过程中插入不同 hook 的操作方式. 理清训练流程只需看 Runner 的成员函数 run. 在 run 里会根据 mode 按配置中 workflow 的 epoch 循环调用 train 和 val 函数, 跑完所有的 epoch. 比如 train:

```
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
```

上面需要说明的是自定义 hook 类, 自定义 hook 类需继承 mmcv 的 Hook 类, 其默认了 6+8+4 个成员函数, 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). 这里的 call_hook 函数定义如下:

```
def call_hook(self, fn_name):
```

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

容易看出, 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      # runner需要管理各节点信息的原因之一
            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, 而当前学习率即为返回值.

第六节 Core

6.1 anchor

anchor 首先来源于 rcnn, 在我看来, anchor 利用强监督的特点, 在原图上构造了完整的 bbox 空间 (非数学意义的空间), 然后根据人为标定 bbox, 选出有效的拟合集合 (子空间, 非严格表达), 从而使得优化变得更有效 (原本是实现了 end2end)。

AnchorGenerator 类为不同特征层生成 anchor, 其中特征层上单个像素的 anchor 是以此像素为中心, 按给定的尺度和扩展比率生成, 也即一个像素对应的 anchor 数为 len(scales)*len(ratios)。

输入参数 base_size, scales, ratios 一般作用于非 retinta 类网络, 含义分别表示: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. 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 (v2.0 不取整):

```
1
2                    [[-21.,  -9.,  24.,  12.],
3                    [-43., -21.,  46.,  24.],
4                    [-14., -14.,  17.,  17.],
5                    [-30., -30.,  33.,  33.],
6                    [ -9., -21.,  12.,  24.],
7                    [-21., -43.,  24.,  46.]]
```

而 retina 等网络会存在 octave_base_scale,scales_per_octave, 其含义和上面保持对应。

因为此处得到的 anchor 是以特征图为坐标系的, 所以要得到原图上的 anchor, 还得把中心点变为对应到原图的上的点。那么回到源代码, 容易看出, gen_single_level_base_anchors 得到单个特征层的所有 anchor,

gen_base_anchors 将不同特征层的 anchor 汇集到列表里,single_level_grid_anchors 将单个特征层的 anchor 变为原图中，grid_anchors 则将所有特征汇集到列表中，列表每个元素为一个 tensor, 记录了对应特征层上的所有 anchor 坐标。代码涉及到两个技巧:

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

# 2.single_level_grid_anchors
shifts = torch.stack([shift_xx, shift_yy, shift_xx, shift_yy], dim=-1)
shifts = shifts.type_as(base_anchors)
# first feat_w elements correspond to the first row of shifts
# add A anchors (1, A, 4) to K shifts (K, 1, 4) to get
# shifted anchors (K, A, 4), reshape to (K*A, 4)
# base_anchors:(fw*hw*num_anchors, 4), shifts:(fw*hw, 4)
all_anchors = base_anchors[None, :, :] + shifts[:, None, :]    #此处直接理解较难
all_anchors = all_anchors.view(-1, 4)
```

另一个 PointGenerator 代码如下:

```
class PointGenerator(object):
  def _meshgrid(self, x, y, row_major=True):
    xx = x.repeat(len(y))
    yy = y.view(-1, 1).repeat(1, len(x)).view(-1)
    if row_major:
      return xx, yy       #  (xx[i],yy[j])为一个矩阵A_xy的坐标(i,j), dim(xx)=1时
    else:
      return yy, xx

  def grid_points(self, featmap_size, stride=16, device='cuda'):
    feat_h, feat_w = featmap_size
    shift_x = torch.arange(0., feat_w, device=device) * stride    # 特征网格对应到原图网格
    shift_y = torch.arange(0., feat_h, device=device) * stride
    shift_xx, shift_yy = self._meshgrid(shift_x, shift_y)
    stride = shift_x.new_full((shift_xx.shape[0], ), stride)
    shifts = torch.stack([shift_xx, shift_yy, stride], dim=-1)    # x, y和对应点的stride大小
    all_points = shifts.to(device)
    return all_points
```

代码上，主要的难点在于高维张量的处理。

6.2 bbox

6.2.1 coder

bbox 的编码是基础重要的，一家之言：anchor genertor 让优化空间合理，bbox 编码让优化更合理 (guided anchor 处在这两者之间)。通常 bbox 连归一化都不做的，效果不会好到哪里去 (可以试试 retinaface)。但是在不同方法体系里面，就不好说了，比如以中心点表示方式的 CenterNet，对 w, h 做回归，就没有编码，直接拟合 $\Delta w, \Delta h$，但它是在下采样 1/4 后的特征图上，也即尺度上还是除了 4。总归而言，要让优化对象存在一个合理的空间上，才是本质的。

one, two stage 的 bbox 编码方式，来源于 14 年 Ross Girshick 等人写的 rcnn，其编码为

$$
\begin{aligned}
t_x &= (G_x - P_x) / P_w \\
t_y &= (G_y - P_y) / P_h \\
t_w &= \log (G_w / P_w) \\
t_h &= \log (G_h / P_h)
\end{aligned}
$$

此处 P 是 anchor，G 是标框。

解码为：

$$
\begin{aligned}
\hat{G}_x &= P_w d_x(P) + P_x \\
\hat{G}_y &= P_h d_y(P) + P_y \\
\hat{G}_w &= P_w \exp (d_w(P)) \\
\hat{G}_h &= P_h \exp (d_h(P))
\end{aligned}
$$

那么代码如何实现呢?

```python
class BaseBBoxCoder(metaclass=ABCMeta):

  def __init__(self, **kwargs):
    pass
  @abstractmethod
  def encode(self, bboxes, gt_bboxes):
    pass
  @abstractmethod
  def decode(self, bboxes, bboxes_pred):
    pass

def bbox2delta(proposals, gt, means=(0., 0., 0., 0.), stds=(1., 1., 1., 1.)):
```

```
13    assert proposals.size() == gt.size()
14
15    proposals = proposals.float()    # 浮点数
16    gt = gt.float()
17    px = (proposals[..., 0] + proposals[..., 2]) * 0.5    # 中心点(x1+x2)/2
18    py = (proposals[..., 1] + proposals[..., 3]) * 0.5
19    pw = proposals[..., 2] - proposals[..., 0]
20    ph = proposals[..., 3] - proposals[..., 1]
21
22    gx = (gt[..., 0] + gt[..., 2]) * 0.5
23    gy = (gt[..., 1] + gt[..., 3]) * 0.5
24    gw = gt[..., 2] - gt[..., 0]    # 宽(x2 - x1) = w
25    gh = gt[..., 3] - gt[..., 1]
26
27    dx = (gx - px) / pw    # \eqref{1}
28    dy = (gy - py) / ph    # \eqref{2}
29    dw = torch.log(gw / pw)
30    dh = torch.log(gh / ph)    # \eqref{4}
31    deltas = torch.stack([dx, dy, dw, dh], dim=-1)    # 最后一维度stack
32
33    means = deltas.new_tensor(means).unsqueeze(0)    # new_tensor和unsqueeze(
        扩张维度)
34    stds = deltas.new_tensor(stds).unsqueeze(0)
35    deltas = deltas.sub_(means).div_(stds)    # sub_ 和 sub 的区别
36
37    return deltas
38
39 gx = torch.addcmul(px, 1, pw, dx)  # gx = px + pw * dx
40 gy = torch.addcmul(py, 1, ph, dy)  # gy = py + ph * dy
41
42 # torch.addcmul(input, tensor1, tensor2, *, value=1, out=None) → Tensor
43 # outi=input_i+value×tensor_1i ×tensor_2i
```

stack, new_tensor, unsqueeze, div_ 可以学习一下。另外为何 Δ 坐标系 (说法不严谨，但可以理解，cv 论文经常这样) 要做高斯归一化变换呢? 可参考Gaussian YOLOv3。我的理解，将优化空间映射到高斯 $\mu - \sigma$ 分布中。

在解码函数 delta2bbox 有如下函数需要注意一下 (tensor.*)：repeat, clamp, expand_as, exp, torch.addcmul, view_as。

另外一种编码方式，来自 19 年 Chenchen Zhu 等人写的FSAF。将 (x1, y1,x2, y2) 编码为 (top, bottom, left, right)，含义自明。代码上需要注意的是 w, h 的归一化以及整体的归一化因子，默认除以 4。此编码对应的 bbox loss 也将变为 Iou 系列 Loss，若说明，也应该是在模型解析部分补充。

6.2.2 assigners

assign 主要给特征图上的 anchor 赋予有意义的监督信息，从而使优化更为有效。那么自然他的参数至少包含 anchors,gt_bboxes,gt_labels。实际因为数据的复杂，或许有 gt_bboxes_ignore 等信息。assign 的机制可总结为如下四点:

1. 将所有框置为背景 (label 置-1)
2. 将与所有 gts 的 iou 小于 neg_iou_thr 的置为 0
3. 将与所有 gt 的 max(iou) 大于 pos_iou_thr 的框置为对应 gt
4. 对每个 gt, 将与其 iou 最大的框置为 gt

关于上述四点: 用生成的 bbox 去拟合实际的 bbox, 那么实际的 bbox 至少有一个可以优化的对象。

有了这些认识，代码就相对容易了。

```
@BBOX_ASSIGNERS.register_module()
class MaxIoUAssigner(BaseAssigner):
    #  关键代码(BaseAssigner,抽象基类)
    # assign:
    ...    #  gt太多，就cpu上计算
    overlaps = self.iou_calculator(gt_bboxes, bboxes)  # gt在前，更快吧
    ...
    # assign_wrt_overlaps
    # 1. 默认全部为背景-1
    assigned_gt_inds = overlaps.new_full((num_bboxes, ),
                                         -1,
                                         dtype=torch.long)
    # 每个bbox与所有gt的最大iou值和index
    max_overlaps, argmax_overlaps = overlaps.max(dim=0)
    # 每个gt与所有bbox的最大iou值和index
    gt_max_overlaps, gt_argmax_overlaps = overlaps.max(dim=1)
    # 2  负样本置0
    assigned_gt_inds[(max_overlaps >= 0)
                     & (max_overlaps < self.neg_iou_thr)] = 0
    # 3 argmax_overlaps的值gt的维度上的index,但其本身的维度是bbox的维度
    pos_inds = max_overlaps >= self.pos_iou_thr
    assigned_gt_inds[pos_inds] = argmax_overlaps[pos_inds] + 1

    # 4 bbox和gt的max(iou)对应的gt未必是此bbox的最好设定，得从gt的max(iou)为出发点，会更好。
    if self.match_low_quality:
```

```
            for i in range(num_gts):
                if gt_max_overlaps[i] >= self.min_pos_iou:
                    if self.gt_max_assign_all:
                        max_iou_inds = overlaps[i, :] == gt_max_overlaps[i]
                        assigned_gt_inds[max_iou_inds] = i + 1
                    else:
                        assigned_gt_inds[gt_argmax_overlaps[i]] = i + 1
    # 最终结果被AssignResult(num_gts, assigned_gt_inds, max_overlaps, labels
    =assigned_labels)
    # 包装，方便调试，查看算法的数据状态
```

6.2.3 sampler

sample 主要为 two-stage 服务. 已实现的有 RandomSampler，CombinedSampler,OHEMSampler, InstanceBalancedPosSampler, IoUBalancedNegSampler, 这些均继承自抽象类 BaseSampler, 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 可算中间类，意思自明。另外，采样只是在做 bbox 的 index 选取，并不会改变 bbox 等的形状，这在最后算损失函数时，只需要乘以选取的 mask 即可。

6.2.4 iou_calculators

torch 版本的 iou 计算，主要注意点是张量计算，广播机制。

```
# bboxes1 (Tensor): shape (m, 4) in <x1, y1, x2, y2> format or empty.
# bboxes2 (Tensor): shape (n, 4) in <x1, y1, x2, y2> format or empty.
lt = torch.max(bboxes1[:, None, :2], bboxes2[:, :2])  # 这里m!=n, [m, None,
    2] + [n, 2] —> [m, n, 2]
# None 技巧
```

6.3 evaluate

mAP 的含义可以参考mmAP 浅析一 我就不滥竽充数了, 将其部分摘录于此:

” 我们都知道，评价一个图像分类结果的性能，只需要看预测类别是否正确即可，在一个数据集上面，我们可以很容易地得出一个平均准确率。可

是目标检测的输出目标数量和真实目标数量都是不固定的（前文提到的非结构化的特性），因此评判时要考虑的就不仅是“对错”这么简单了，我们需要考虑的有：如果漏掉了一个目标对性能有多大损伤？如果多检测出了一个目标对性能有多大损伤？如果检测出来的位置信息有所偏差对性能有多大损伤？进一步地，在这么多的检测结果中，总有一些是检测器十分笃定的，有一些是模棱两可的。如果检测器对多检测出来的那个目标本身也不太确定呢？如果检测器最满意最信任的那个检测结果出错了呢？换言之：一个检测结果对性能的影响，是否应该和检测器对它的满意程度（置信度）相关？以及，检测错了一个稀有的物体和检测错了一个常见的物体所带来的性能损伤是否应该相同？.....

刚刚提到的所有问题，mmAP 都要一一给出答案。

首先是位置偏差问题。有的使用场景对位置的准确度要求不高，有的则要求精确定位。因此，mmAP 先按位置准确度的需求进行划分，设置一组 IOU 阈值，这组阈值为 (0.5, 0.55, 0.6, ..., 0.9, 0.95)，如果 DT 与 GT 的 IOU 超过阈值，则视作检测成功。这样每给定一个阈值就可以计算出一个性能（也就是 mAP，后面详述），然后对这些性能取平均（也就是 mmAP，后面详述）就是整个检测算法的性能了。

然后是类别平衡问题，这一问题在分类领域非常常见，“将一个白血病患者错分为健康的人“和“将一个健康的人错分为白血病患者“是一样的吗？显然不是，因为白血病患者本身就属于少数，如果一个分类器把所有人都无脑地判断为健康，其正确率就是健康的人/全部人。这个分类器的正确率很高但是完全失去了判断病患的功能。mmAP 为了公平的评价检测器在各个类别上的性能，采用了类别间求平均的方法：先给定一个 IOU 阈值，然后将所有的 GT 和 DT 按照类别先进行划分，用同一类的所有 GT 和 DT 计算出一个性能（也就是 AP，马上详述），然后对所有类别的性能取平均（mAP），就是检测算法在这个 IOU 阈值下的性能。mmAP 为了公平的评价检测器在各个类别上的性能，采用了类别间求平均的方法：先给定一个 IOU 阈值，然后将所有的 GT 和 DT 按照类别先进行划分，用同一类的所有 GT 和 DT 计算出一个性能（也就是 AP，马上详述），然后对所有类别的性能取平均（mAP），就是检测算法在这个 IOU 阈值下的性能。

现在我们来看看，给定了一个 IOU 阈值、并给定了一个类别，如何具体地计算检测的性能。首先，我们要先对所有的检测结果排序，得分越高的排序越靠前，然后依次判断检测是否成功。将排序后的所有结果定义为

DTs，所有同类别的真实目标定义为 GTs。先依序遍历一遍 DTs 中的所有 DT，每个 DT 和全部 GT 都计算一个 IOU，如果最大的 IOU 超过了给定的阈值，那么视为检测成功，算作 TP（True Positive），并且最大 IOU 对应的 GT 被视为匹配成功；如果该 DT 与所有 GT 的 IOU 都没超过阈值，自然就是 FP（False Positive）；同时，每当一个 GT 被检测成功后，都会从 GTs 中“被取走”，以免后续的检测结果重复匹配。因此如果有多个检测结果都与同一个 GT 匹配，那么分数最高的那个会被算为 TP，其余均为 FP。遍历完成后，我们就知道了所有 DTs 中，哪些是 TP，哪些是 FP，而由于被匹配过的 GT 都会“被取走”，因此 GTs 中剩下的就是没有被匹配上的 FN（False Negative）。

有了 TP、FP、FN 的定义，就可以方便地得出准确率（Precison，P，即所有的检测结果中多少是对的）和召回率（Recall，R，即所有的真实目标中有多少被检测出来了），两者的定义分别为：P = TP / (TP + FP), R = TP / (TP + FN) = TP / len(GTs)。

但是，单纯地用 Precision 和 Recall 来评价整个检测器并不公平，因为有的检测任务要求更高的 Recall，“错检”几个影响不大；有的检测任务则要求更高的 Precision，“漏检”几个影响不大。因此我们需要对 Precision 和 Recall 做一个整体的评估，而这个评估就是前文提到的 AP（Average Precision），其定义非常简单，对排序好的 det 结果进行“截取”，依次观察 det 结果的前 1 个（也就是只有第一个结果）、前 2 个、...、前 N 个，每次观察都能够得到一个对应的 P 和 R，随着观察数量的增大，R 一定会变大或不变。因此可以以 R 为横轴，P 为纵轴，将每次的“截取”观察到的 P 和 R 画成一个点（R，P）。值得注意的是，当“截取”到的新结果为 FP 时，因为 R 没有变化所以并不会有新的（R，P）点诞生。最后利用这些（R，P）点绘制成 P-R 曲线，定义：$AP = \int_0^1 P\mathrm{d}R$。通俗点讲，AP 就是这个 P-R 曲线下的面积。AP 计算了不同 Recall 下的 Precision，综合性地评价了检测器，并不会对 P 和 R 有任何“偏好”，同时，检测分数越高的结果对 AP 的影响越大，分数越低的对 AP 的影响越小。实际上在计算 AP 时，都要对 P-R 曲线做一次修正，将 P 值修正为当 R>R0 时最大的 P（R0 即为该点对应的 R），即 $AP = \int_0^1 max(\{P(r)|r \geq R\})\mathrm{d}R$

除了 AP、mAP、mmAP 之外，还有一个重要的性能是 Recall，有时我们也需要关心“检测器能达到的最大 Recall 是多少？尽管此时的 Precision 可能非常低”，AR 就是度量 Recall 的。每给定一个 IOU 阈值和类别，都

会得到一个 P-R 曲线，该曲线 P 不为 0 时的最大的 R，就称为该 IOU 阈值下该类别的 Recall（其实是“最大 Recall”），在类别尺度上平均后，就是该 IOU 阈值下的 AR，通常我们会用 AR[0.5:0.95] 表示所有 IOU 阈值下 AR 的平均值（也就是 mAR）。值得注意的是，AR 并不是和 mmAP 同等量级的性能度量指标，因为 AP 综合考虑了 P 和 R，而 AR 只是考虑了 Recall。计算 AR 通常是用于帮助我们分析检测器性能特点的。在两个检测结果的 mmAP 相同时，更高的 AR 并不意味着更好的检测效果，而仅仅意味着“更扁平的 P-R 曲线”（可自行脑补该曲线）。”

mmAP 浅析二对 mAP 做了更进一步的探讨。

mmdet 中模型性能评估代码有多种，coco，cityspaces 格式调用了各自默认的 api，而被转化为 mmdet 中间通用格式的 (xml 等)，则评估调用 mean_ap, recall 函数。这里对代码做一些简单的注释。

```
def average_precision(recalls, precisions, mode='area'):
    """
    Returns:
        float or ndarray: calculated average precision
    """
    no_scale = False
    if recalls.ndim == 1:
        no_scale = True
        recalls = recalls[np.newaxis, :]     #  扩展成scale格式
        precisions = precisions[np.newaxis, :]
    assert recalls.shape == precisions.shape and recalls.ndim == 2
    num_scales = recalls.shape[0]
    ap = np.zeros(num_scales, dtype=np.float32)    # 多个scale的ap
    if mode == 'area':
        zeros = np.zeros((num_scales, 1), dtype=recalls.dtype)
        ones = np.ones((num_scales, 1), dtype=recalls.dtype)
        mrec = np.hstack((zeros, recalls, ones))   # 水平合并(x轴),若合并对象多维,则对应维度水平合并, (num_scales, 3)
        mpre = np.hstack((zeros, precisions, zeros))
        for i in range(mpre.shape[1] - 1, 0, -1):    # 从后往前递推替换当前值为相邻两向量对应的较大值
            mpre[:, i - 1] = np.maximum(mpre[:, i - 1], mpre[:, i])  # 返回两向量对应位置的较大值
        for i in range(num_scales):
            ind = np.where(mrec[i, 1:] != mrec[i, :-1])[0]     # (r,1) != (0, r)
            ap[i] = np.sum(
                (mrec[i, ind + 1] - mrec[i, ind]) * mpre[i, ind + 1])  # R差*P
    elif mode == '11points':
```

```
        for i in range(num_scales):
            for thr in np.arange(0, 1 + 1e-3, 0.1):
                precs = precisions[i, recalls[i, :] >= thr]    # 召回率大于11等分阈值对应的精度
                prec = precs.max() if precs.size > 0 else 0    # 实际处理,取精度的极大值(修正后的P-R曲线)
                ap[i] += prec
            ap /= 11
    else:
        raise ValueError(
            'Unrecognized mode, only "area" and "11points" are supported')
    if no_scale:
        ap = ap[0]
    return ap
```

```
def tpfp_default(det_bboxes,
                 gt_bboxes,
                 gt_bboxes_ignore=None,
                 iou_thr=0.5,
                 area_ranges=None):
    """Check if detected bboxes are true positive or false positive.
    Returns:
        tuple[np.ndarray]: (tp, fp) whose elements are 0 and 1. The shape of
            each array is (num_scales, m).
    """
    # an indicator of ignored gts
    gt_ignore_inds = np.concatenate(
        (np.zeros(gt_bboxes.shape[0], dtype=np.bool),
         np.ones(gt_bboxes_ignore.shape[0], dtype=np.bool)))
    # stack gt_bboxes and gt_bboxes_ignore for convenience
    gt_bboxes = np.vstack((gt_bboxes, gt_bboxes_ignore))

    num_dets = det_bboxes.shape[0]
    num_gts = gt_bboxes.shape[0]
    if area_ranges is None:
        area_ranges = [(None, None)]
    num_scales = len(area_ranges)
    # tp and fp are of shape (num_scales, num_gts), each row is tp or fp of a certain scale
    tp = np.zeros((num_scales, num_dets), dtype=np.float32)   # 构造数据mask
    fp = np.zeros((num_scales, num_dets), dtype=np.float32)

    # if there is no gt bboxes in this image, then all det bboxes, within area range are false positives
    if gt_bboxes.shape[0] == 0:
        if area_ranges == [(None, None)]:
            fp[...] = 1
```

```
        else:
            det_areas = (det_bboxes[:, 2] - det_bboxes[:, 0]) * (
                det_bboxes[:, 3] - det_bboxes[:, 1])
            for i, (min_area, max_area) in enumerate(area_ranges):
                fp[i, (det_areas >= min_area) & (det_areas < max_area)] = 1
        return tp, fp

    ious = bbox_overlaps(det_bboxes, gt_bboxes)     #这里的bbox_overlaps可以去掉循环
    # for each det, the max iou with all gts
    ious_max = ious.max(axis=1)
    # for each det, which gt overlaps most with it
    ious_argmax = ious.argmax(axis=1)       # 若torch.Tensor格式, 则可一次得到value, idx
    # sort all dets in descending order by scores
    sort_inds = np.argsort(-det_bboxes[:, -1])
    # 细节上需注意数据mask的构造问题:gt_area_ignore,gt_covered,tp,fp
    for k, (min_area, max_area) in enumerate(area_ranges):
        gt_covered = np.zeros(num_gts, dtype=bool)
        # if no area range is specified, gt_area_ignore is all False
        if min_area is None:
            gt_area_ignore = np.zeros_like(gt_ignore_inds, dtype=bool)
        else:
            gt_areas = (gt_bboxes[:, 2] - gt_bboxes[:, 0]) * (
                gt_bboxes[:, 3] - gt_bboxes[:, 1])
            gt_area_ignore = (gt_areas < min_area) | (gt_areas >= max_area)
        for i in sort_inds:
            if ious_max[i] >= iou_thr:
                matched_gt = ious_argmax[i]
                if not (gt_ignore_inds[matched_gt]
                        or gt_area_ignore[matched_gt]):
                    if not gt_covered[matched_gt]:
                        gt_covered[matched_gt] = True
                        tp[k, i] = 1
                    else:
                        fp[k, i] = 1
                # otherwise ignore this detected bbox, tp = 0, fp = 0     #细节
            elif min_area is None:
                fp[k, i] = 1
            else:
                bbox = det_bboxes[i, :4]
                area = (bbox[2] - bbox[0]) * (bbox[3] - bbox[1])
                if area >= min_area and area < max_area:
                    fp[k, i] = 1
    return tp, fp
```

```python
def eval_map(det_results,
             annotations,
             scale_ranges=None,
             iou_thr=0.5,
             dataset=None,
             logger=None,
             nproc=4):
    """Evaluate mAP of a dataset.
    Returns:
        tuple: (mAP, [dict, dict, ...])
    """
    assert len(det_results) == len(annotations)
    # 这里det_results数据，每张图的结果(一个列表)，都按类别排序了

    num_imgs = len(det_results)
    num_scales = len(scale_ranges) if scale_ranges is not None else 1
    num_classes = len(det_results[0])  # positive class num
    area_ranges = ([(rg[0]**2, rg[1]**2) for rg in scale_ranges]
                   if scale_ranges is not None else None)

    pool = Pool(nproc)
    eval_results = []
    #  类别循环
    for i in range(num_classes):
        # get gt and det bboxes of this class
        cls_dets, cls_gts, cls_gts_ignore = get_cls_results(
            det_results, annotations, i)
        # choose proper function according to datasets to compute tp and fp
        if dataset in ['det', 'vid']:
            tpfp_func = tpfp_imagenet
        else:
            tpfp_func = tpfp_default
        # compute tp and fp for each image with multiple processes
        tpfp = pool.starmap(
            tpfp_func,
            zip(cls_dets, cls_gts, cls_gts_ignore,
                [iou_thr for _ in range(num_imgs)],
                [area_ranges for _ in range(num_imgs)]))  # zip取对应列作为一套参数，进入多进程
        tp, fp = tuple(zip(*tpfp))  #  元组格式值不被改变
        # calculate gt number of each scale, ignored gts or gts beyond the specific scale are not counted
        num_gts = np.zeros(num_scales, dtype=int)
        for j, bbox in enumerate(cls_gts):
            if area_ranges is None:
                num_gts[0] += bbox.shape[0]
```

```
45              else:
46                  gt_areas = (bbox[:, 2] - bbox[:, 0]) * (
47                      bbox[:, 3] - bbox[:, 1])
48                  for k, (min_area, max_area) in enumerate(area_ranges):
49                      num_gts[k] += np.sum((gt_areas >= min_area)
50                                           & (gt_areas < max_area))
51          # sort all det bboxes by score, also sort tp and fp
52          cls_dets = np.vstack(cls_dets)
53          num_dets = cls_dets.shape[0]
54          sort_inds = np.argsort(-cls_dets[:, -1])
55          tp = np.hstack(tp)[:, sort_inds]
56          fp = np.hstack(fp)[:, sort_inds]    # 对应同时排序
57          # calculate recall and precision with tp and fp
58          tp = np.cumsum(tp, axis=1)    # [1,2,3]-->[1,+2,1+2+3]..
59          fp = np.cumsum(fp, axis=1)
60          eps = np.finfo(np.float32).eps
61          recalls = tp / np.maximum(num_gts[:, np.newaxis], eps)
62          precisions = tp / np.maximum((tp + fp), eps)
63          # calculate AP
64          if scale_ranges is None:
65              recalls = recalls[0, :]
66              precisions = precisions[0, :]
67              num_gts = num_gts.item()
68          mode = 'area' if dataset != 'voc07' else '11points'
69          ap = average_precision(recalls, precisions, mode)
70          # 当前所有图的一类检测结果
71          eval_results.append({
72              'num_gts': num_gts,
73              'num_dets': num_dets,
74              'recall': recalls,
75              'precision': precisions,
76              'ap': ap
77          })
78      pool.close()
79      if scale_ranges is not None:
80          # shape (num_classes, num_scales)
81          all_ap = np.vstack([cls_result['ap'] for cls_result in eval_results
    ])
82          all_num_gts = np.vstack(
83              [cls_result['num_gts'] for cls_result in eval_results])
84          mean_ap = []
85          for i in range(num_scales):
86              if np.any(all_num_gts[:, i] > 0):
87                  mean_ap.append(all_ap[all_num_gts[:, i] > 0, i].mean())
88              else:
89                  mean_ap.append(0.0)
```

```
90      else:
91          aps = []
92          for cls_result in eval_results:
93              if cls_result['num_gts'] > 0:
94                  aps.append(cls_result['ap'])
95          mean_ap = np.array(aps).mean().item() if aps else 0.0       #  所有类
        的平均ap—>mAP
96
97      print_map_summary(
98          mean_ap, eval_results, dataset, area_ranges, logger=logger)   # 好看
        的打印
99
100     return mean_ap, eval_results
```

需要注意一点，模型在训练中评估，需要走 mmdet 的训练流程，也就是说，评估函数需要被注册到对应的钩子中去，于是需要在 Hook 中实现 after_train_epoch。因为 mmdet 的数据类，实现了对应的 evaulate, 所以这里的 EvalHook 可按如下形式实现:

```
1  class EvalHook(Hook):
2      ...
3
4      def after_train_epoch(self, runner):
5          if not self.every_n_epochs(runner, self.interval):
6              return
7          from mmdet.apis import single_gpu_test
8          results = single_gpu_test(runner.model, self.dataloader, show=False)
9          self.evaluate(runner, results)
10
11     def evaluate(self, runner, results):
12         eval_res = self.dataloader.dataset.evaluate(
13             results, logger=runner.logger, **self.eval_kwargs)
14         for name, val in eval_res.items():
15             runner.log_buffer.output[name] = val
16         runner.log_buffer.ready = True
```

总结：原理，怎么组织数据格式，多加练习 numpy 的基本函数。

6.4 optimizer

torch 的优化器基本完备，mmdet 中这部分代码上也没什么可说的，所以这里主要说一些和理论相关的内容。

6.4.1 优化算法

1. The Loss Surfaces of Multilayer Networks(2015.Anna Choromanska)
2. Escaping From Saddle Points —Online Stochastic Gradient for Tensor Decomposition(2015.Rong Ge)
3. Efficient approaches for escaping higher order saddle points in non-convex optimization(2016.Anima Anandkumar)
4. How to Escape Saddle Points Efficiently(2017.Chi Jin)
5. Hessian-based Analysis of Large Batch Training and Robustness to Adversaries(2018)
6. Over- Deep Neural Networks Have No Strict Local Minima For Any Continuous Activations (2018)
7. Gradient descent with identity initialization efficiently learns positive definite linear transformations by deep residual networks (2018)
8. Deep linear neural networks with arbitrary loss: All local minima are global (2017)

1,2,3 主要来自此篇博文: Escaping from Saddle Points ，论文 1 说明了在深度学习中, 几乎所有的局部极值点和全局最优点的函数差值都不大, 因此只要收敛到局部极小值点就差不多了. 而文章 3 又指出寻找一般非凸函数的局部极值点是一个 NP-hard 问题. 除此之外, 文章 3 主要考虑了非凸优化问题中的 degenerate saddle points ，并给出利用三阶导数信息来逃离鞍点并收敛到三阶局部最下值的算法. 文章 2 中作者定义了 strict saddle 概念, 并给出随机梯度下降法能在多项式时间内收敛到局部极值点的证明 (有限制条件), 也即 $y = x - \eta\nabla f(x) + \epsilon$, 其中 ϵ 就是梯度随机因子. 毕竟鞍点是不稳定的, 加扰动因子, 是很容易逃离鞍点附近的. 但是, 问题是如果鞍点附近平稳区域太大, 或者逃离的方向不太对, 怎么办? 这样就很容易跑到无穷小或无穷大去了. 关于这一点, 博主又更新了一篇文章 Saddles Again, 文章表明一般优化算法很难收敛到鞍点, 除非你精心设计初始点, 调节参数等.

考虑简单的不定二次型,$f(x) = \frac{1}{2}\sum_{i=1}^{d} a_i x_i^2$ 其中前 k 个系数为正, 后 $d-k$ 个系数为负数, 因此我们很容易得到其梯度迭代形式:

$$x^{(k+1)} = x^{(k)} - t\nabla f\left(x^{(k)}\right)$$

考虑第 i 维度, 容易得到

$$x_i^{(k)} = (1 - ta_i)^k \, x_i^{(0)}$$

因 $\exists a_i > 0$, 于是从任何非零点开始都将以指数般的速度发散到无穷远. 当且仅当初始点为 0 时, 方才收敛, 但凡有一点扰动, 都将发散 (原文似乎有一些不严密的地方). 文章 4 也是在梯度方向上增加扰动, 给出了几乎与维度无关的收敛到二阶驻点的多项式时间复杂度算法. 而且收敛速度和众所周知的梯度下降收敛到一阶驻点的收敛速度同步差仅一个对数因子. 该方法可直接用于矩阵分解问题.

文章 5 实验了使用不同大小批量训练时, 模型的收敛邻域的局部几何之间的区别. 结果显示, 此前人们普遍相信的鞍点困扰优化的论据其实并不存在在大批量训练过程中, 真正的原因是大批量训练的模型会逐渐收敛于具有更大的谱的区域, Hessian 矩阵的谱越大, 其对应的极值点更尖锐, 因此泛化更低, 但, 锐度可能不是唯一的因素.

DL 中的优化算法，这里采用Juliuszh 的逻辑说明:

约定: 优化参数：w ，目标函数：$f(w)$ ，初始学习率 α，epoch t。

参数更新框架:

1. 计算梯度：$g_t = \nabla f\left(w_t\right)$
2. 计算一，二阶动量：$m_t = \phi\left(g_1, g_2, \cdots, g_t\right); V_t = \psi\left(g_1, g_2, \cdots, g_t\right)$
3. 计算下降梯度：$\eta_t = \alpha \cdot m_t / \sqrt{V_t}$
4. 根据下降梯度更新参数：$w_{t+1} = w_t - \eta_t$

各个算法的差别:

- SGD: $m_t = g_t; V_t = I^2$，没有动量
- SGDM:$m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$, 一阶动量是各个时刻梯度方向的指数移动平均值，约等于最近 $\frac{1}{1-\beta_1}$ 个时刻的梯度向量和的平均值, 于是下降方向由当前梯度方向和累计梯度方向共同决定.
- SGDN: $g_t = \nabla f\left(w_t - \alpha \cdot m_{t-1} / \sqrt{V_{t-1}}\right)$, 按累计梯度方向走一步后的梯度方向和历史动量相结合.
- AdaDelta:$V_t = \sum_{\tau=1}^{t} g_\tau^2$，用二阶动量去度量历史更新频率, 来动态调节学习率 (频率越快学习率越小).

- AdaDelta / RMSProp: $V_t = \beta_2 * V_{t-1} + (1 - \beta_2)\, g_t^2$, 利用指数移动平均值来计算过去一段时间梯度的二阶动量, 避免学习率递减的过快.
- Adam: $m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t, V_t = \beta_2 * V_{t-1} + (1 - \beta_2)\, g_t^2$, 同时考虑了一二阶动量, 且都是指数移动平均值.
- Nadam: Nesterov + Adam.

问题:

- 1. SGD 收敛速度慢, 且不容易逃离局部最优点。从6.4.3的实验发现, 针对同样的参数, 其他优化方法在给定的函数上也很难跳出局部最优点。
- 2. Adam 所谓的学习率自适应并不是真的自适应, 在一些优化函数上可能导致不收敛 (后期学习率有震荡现象), 可能错过全局最优 (跳出去了就很难回来)

经验:

- 优先考虑 SGD+Nesterov Momentum 或者 Adam
- Adam 等自适应学习率算法对于稀疏数据具有优势，且收敛速度很快；但精调参数的 SGD（+Momentum）往往能够取得更好的最终结果。
- 先用 Adam 快速下降，再用 SGD 调优:Improving Generalization Performance by Switching from Adam to SGD

最后补充两博文：张戎(导师沈维孝)。7 种优化函数在一元函数上的实验7 优化一元函数。

6.4.2 初始化

初始化对寻找更优的极值点来说，极为重要。这节我以几个具体的二元函数为例，来直观说明此问题。而关于神经网络的初始化，对模型性能的提升同样很重要，在同样的优化策略下，好的初始化方式，也许会带来几个百分点的性能提升，相反糟糕的初始化也许让你不经怀疑模型是否有问题等严重影响判断的现象。然而，神经网络的初始化和本节讲的具体的二元函数的初始化还是有其不同的地方，首先，神经是没有具体表达式的，其初始化

的是所有可能的函数集的函数的参数，因此不同的初始化方式，将覆盖不同的函数集合，从而也必将影响最终模型所代表的函数与最优函数的差距。

目前深度学习中，初始化方式从 pytorch 官方文档 torch.init 中可知大概有 11 种方式，比如 uniform, normal, xavier_uniform, kaiming_uniform, orthogonal，sparse，constant 等。

在上一小小节有人在一元函数上做了 7 种优化器的实验，这里我给出一些二元函数的实验例子。

$$
\begin{aligned}
&f(x,y) = (x^2+y-11)^2 + (x+y^2-7)^2 \quad (\text{ himmelblau }) \\
&f(x,y) = x^3 - 3xy^2 \quad (\text{ monkey sad}\,dle) \\
&\text{compleax peaks :} \\
&f(x,y) = 3*(1-x)^2 * e^{-x^2-(y+1)^2} - 10*(x/5 - x^3 - y^5)* \\
&\quad e^{-\left(x^2+y^2\right)} - \frac{e^{-(x+1)^2-y^2}}{3} - \frac{\sin\left(x^2+y^2\right)-\cos\left(x^2+y^2\right)}{2} \\
&f(x,y) = \sin(wx)^2 * \sin(wy)^2 * e^{\frac{x+y}{\sigma^2}} - \\
&\qquad 2*\sin(2*w(x+2))^2 * \sin(2*w(y+2))^2 * e^{\frac{x+y}{\sigma^2}} + \\
&\qquad \sin(3*w(x-2))^2 * \sin(3*w(y-2))^2 * e^{\frac{x+y}{\sigma^2}}
\end{aligned}
$$

前两个是熟知的函数，后两个是我构造的例子，主要出发点是找一个有很多波峰波谷的函数。

```
import numpy as np
import torch
from matplotlib import pyplot as plt
from mpl_toolkits.mplot3d import Axes3D


def himmelblau(x):
  return (x[0]**2 + x[1]-11)**2 + (x[0] + x[1]**2 -7)**2

def monkey_saddle(x):
  return (x[0]**3 - 3*x[0]*x[1]**2)

def many_peaks(x):
  return 3*(1-x[0])**2 * torch.exp(-x[0]**2-(x[1]+1)**2) - 10*(x[0]/5 - x
    [0]**3 - x[1]**5) * \
    torch.exp(-x[0]**2 - x[1]**2) - torch.exp((x[0]+1)**2-x[1]**2)/3 - \
      (torch.sin(x[0]**2+x[1]**2) - torch.cos(x[0]**2+x[1]**2))/2

def complex_peaks(x, w=2, sigmma=2):
```

```
19    return torch.sin(w*x[0]**2 * torch.sin(w*x[1])**2 * torch.exp(x[0]+x[1])/
        sigmma**2) \
20      - 2*torch.sin(2*w*(x[0]+2))**2 * torch.sin(2*w*(x[1]+2))**2 * torch.exp
        ((x[0]+x[1])/sigmma**2) \
21        + torch.sin(3*w*(x[0]-2))**2 * torch.sin(3*w*(x[1]-2))**2 * torch.exp
        ((x[0]+x[1])/sigmma**2)
22
23
24 # x = np.arange(-4, 4, 0.01)
25 # y = np.arange(-4, 4, 0.01)
26 # for complex_peaks function
27 x = np.linspace(2, 6, 400)
28 y = np.linspace(2, 6, 400)
29 X, Y = np.meshgrid(x, y)
30
31 # Z = himmelblau([X, Y])
32 # Z = monkey_saddle([X, Y])
33 Z = many_peaks([torch.from_numpy(X), torch.from_numpy(Y)]).numpy()
34 # Z = complex_peaks([torch.from_numpy(X), torch.from_numpy(Y)]).numpy()
35
36 fig = plt.figure('himmeblau')
37 ax = fig.gca(projection='3d')
38 ax.plot_surface(X, Y, Z, cmap='Accent')
39
40 grad_x_sets = []
41 grad_y_sets = []
42 grad_z_sets = []
43
44 x_0 = torch.tensor([0., 0.], requires_grad=True)
45 optimizer = torch.optim.SGD([x_0], lr=0.01, momentum=0.05)
46 # optimizer = torch.optim.RMSprop([x_0], lr=10, alpha=0.9)
47 # optimizer = torch.optim.RMSprop([x_0], lr=10, alpha=0.9, momentum=0.85)
48 # optimizer = torch.optim.Adam([x_0], betas =(0.9,0.99), lr=20)
49 # lr_scheduler = CyclicLR(optimizer, base_lr=1e-2, max_lr=1e-0)
50
51 for step in range (100) :
52   pred = himmelblau(x_0)
53   # pred = many_peaks(x_0)
54   # pred = complex_peaks(x_0)
55
56   optimizer.zero_grad()
57   # lr_scheduler.batch_step()
58   # print('lr:', lr_scheduler.get_lr())
59   pred.backward()
60   optimizer.step()
61
```

```
62    grad_x_sets.append(x_0.tolist()[0])
63    grad_y_sets.append(x_0.tolist()[1])
64    grad_z_sets.append(pred.tolist())
65
66    if step % 2==1:
67      print('ste {}:x={}, f(x)={}'.format(step, x_0.tolist(),pred.item()))
68
69  ax.plot(grad_x_sets, grad_y_sets, grad_z_sets, c='black', lw=4, label='
        gradient decent curve')
70  plt.show()
```

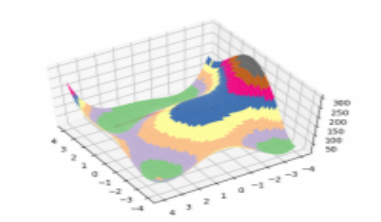


图 3: himmelblau

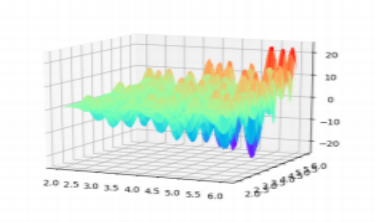


图 4: complex peaks

关于 himmelblau 函数，容易算出其四个极小值点为 (3,2), (-3.7794,-3.2832), (3.5845, -1.8481), (-2.8050, 3.1313) 四极值点的中心点几乎是 (0, 0) 点，明显 (0, 0) 点于 (3, 2) 点最近，于是初始化值若为 (0, 0)，则优化到的极值点一般就是 (3, 2)(图例 himmelblau 中红色线条为带动量的 SGD 优化器优化路线), 但是当你学习率较大，比如 lr=10, 使用 adam，则结果为第二个极值点，lr=20 时，结果为第三个极值点等。对于 complex_peaks，观察其在 $[2,6]\times[2,6]$ 区域内的图形，容易看出极值点密集，选取不同初始值并使用不同的优化器，你会发现很难找到全局的极值点，几乎都停留在初始值附近的波谷之中。也许神经网络的极值点也大概如此，只是波峰波谷之间的差距没有 complex_peaks 函数来得大，但我们仍然要适当跳出局部极值点，向全局极值点逼近，关于这个问题，我想下一小节的动态调整学习率，会有一定的帮助 (事实上对鞍点的帮助更大，这点我将利用下一小节的策略，补充 monkey_saddle 函数的一些实验情况)。

6.4.3 学习率

涉及学习率的问题，可以有初始化学习率，学习率的变化策略两点。对于一个简单的曲面，我们可以根据曲面的一些性质，动态调节学习率，加速

收敛，但由于 DL 高度非凸且复杂，以及实际应用中的大数据，我们只能在整体上做一些调节，比如熟知的 2x step，分步减小，周期性变化Cyclical lr，前者假设 first step 已经优化到极值点附近，后者增加了探索性。最佳初始化学习率，可参考 find start lr 这里将其改为 pytorch1.x 版本。

```

def find_lr(init_value = 1e-8, final_value=10., beta = 0.98):
  num = len(trn_loader)-1
  mult = (final_value / init_value) ** (1/num)
  lr = init_value
  optimizer.param_groups[0]['lr'] = lr
  avg_loss = 0.
  best_loss = 0.
  batch_num = 0
  losses = []
  log_lrs = []
  for data in trn_loader:
    batch_num += 1
    #As before, get the loss for this mini-batch of inputs/outputs
    inputs,labels = data
    inputs, labels = inputs.to(device), labels.to(device)
    optimizer.zero_grad()
    outputs = net(inputs)
    loss = criterion(outputs, labels)
    #Compute the smoothed loss
    avg_loss = beta * avg_loss + (1-beta) *loss.data[0]
    smoothed_loss = avg_loss / (1 - beta**batch_num)
    #Stop if the loss is exploding
    if batch_num > 1 and smoothed_loss > 4 * best_loss:
      return log_lrs, losses
    #Record the best loss
    if smoothed_loss < best_loss or batch_num==1:
      best_loss = smoothed_loss
    #Store the values
    losses.append(smoothed_loss)
    log_lrs.append(math.log10(lr))
    #Do the SGD step
    loss.backward()
    optimizer.step()
    #Update the lr for the next step
    lr *= mult
    optimizer.param_groups[0]['lr'] = lr
  return log_lrs, losses
```

这里用 CycleLR，来测试一下上小节给的函数，实验参数略，实验结果总结起来大致有：针对 complex_peak 函数，当学习率 < 1 时，除 SGD

外各个优化器都很难跳出局部最优；学习率过小且带动量，则连局部极值点都很难收敛到；学习率大于一定阈值，各优化器均可以跳出去... ，对于 himmelblau 函数，当其收敛到某一极值点后，各优化器在循环学习率更新策略下均很难跳出去。以下为梯度更新方向示例图。

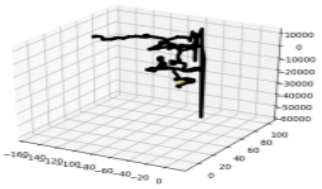


图 5: rmsprop momentum

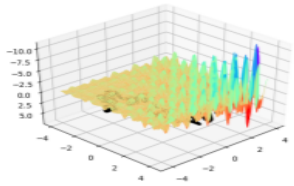


图 6: sgd mometum

想象一下，要想跳出局部极值点，必须增加探索的范围，或者在极值区域附近，疯狂增加学习率，并记录很多中间结果，然后采用回溯的方式，进行更新。只是这样，代码复杂，速度很慢，而且实际情况是 DL 很多极值点的高度都差不多 (6.4.1)。

6.5 post_processing

nms 包含文字，旋转框，多边形，mask，等，相关描述略，代码可参考本地 lib_functions 文件夹。

第七节 Detectors

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

7.0.1 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.

首先 backbone 为 resnet50，(resnet 系列结构参见9), 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14, 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14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7.3.1, 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).

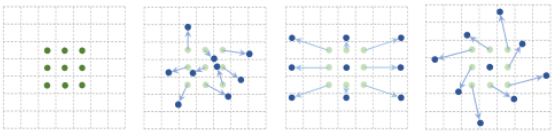
RoIAlign 在 ops 中, 经 cuda 加速, 详解略. 其中 roi_extractors 中的特征层级映射函数如下:

```
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
```

7.0.2 RepPoints



(a) (b) (c) (d)

Figure 1: Illustration of the sampling locations in 3×3 standard and deformable convolutions. (a) regular sampling grid (green points) of standard convolution. (b) deformed sampling locations (dark blue points) with augmented offsets (light blue arrows) in deformable convolution. (c)(d) are special cases of (b), showing that the deformable convolution generalizes various transformations for scale, (anisotropic) aspect ratio and rotation.

图 7: dcn

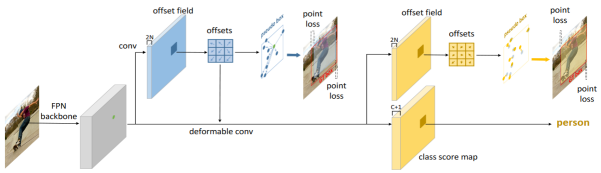


Figure 2. Overview of the proposed RPDet (RepPoints detector). While feature pyramidal networks (FPN) [24] are adopted as the backbone, we only draw the afterwards pipeline of one scale of FPN feature maps for clear illustration. Note all scales of FPN feature maps share the same afterwards network architecture and the same model weights.

图 8: reppoints

2019 年，Ze Yang 等人利用DCNv2的特性，实现了结构点表示物体的检测算法 RepPoints。其中 DCNv2 从 DCNv1 进化而来，出发点如7所示，改变了卷积的采样点。其卷积数学表达式如下：$y(p) = \sum_{k=1}^{K} w_k \cdot x\left(p + p_k + \Delta p_k\right) \cdot \Delta m_k$ 对应的池化层为：$y(k) = \sum_{j=1}^{n_k} x\left(p_{kj} + \Delta p_k\right) \cdot \Delta m_k / n_k$

其中 $w_k, p, p_k, \Delta p_k, \Delta m_k$ 分别表示卷积核参数，卷积中心点 (图上)，当前卷积点，当前点的偏移量，当前点的权重因子，就这么简单。不规则采样点和图像中物体的不规则形状是吻合的，所以直观上讲，这个卷积是很有意义的。实验上你可以将以前的 conv 策略性的替换为 dcn，也许会带来性能的提升 (ps 后续很多新检测算法实验，证明了这一点)。

问题是如何利用 DCNv2 不规则卷积，来做检测算法呢?

1. 通过定位和分类的直接监督来学习可形变卷积的偏移量，使得偏移量具有可解释性 (@ 陀飞轮)。

2. 可以通过采样点来直接生成伪框 (pseudo box)，不需要另外学习边界框，并且分类和定位是有联系的 (@ 陀飞轮)。

从8可以看出，DCN 卷积得到 $2N$ 的偏移量，将其限制到标定的 bbox，类似于 rpn，然后重复 DCN 得到 $2N, C+1$ 分别作送入 bbox 回归和 class 分类 (refine)。也即上面所说的两点。其中的关键点是学习的 points，需要限制到标定的 bbox 中，也即学习的引导。因为是点表示，所以一个自然的问题是比如人脸检测，那么所得到的点的几何结构是一致的吗? 更精细的问题是，可以得到物体的边缘点吗?

这需要对算法更细致的研究了，可以参考一下官方说明:

“这里面“可学习的采样点集合”其实是更本质的概念，这些采样点同时用来提取语义对齐的特征，又用来表示物体的几何形态。deformable convolution 是其中一种基于采样点的特征提取方法，但同时还可以有其它的实现。几何形态监督的方式也可以是更多样的，除了物体检测中的 bounding box 标注外，还可以利用更精细的几何标注来进行监督。我们最近有一个工作就提出了不同的特征提取方法，以及不同的几何监督方式。

通常来说，用 anchor 去覆盖 4d 空间是困难的，所以一般需要不同尺度和不同长宽比的多种 anchor，而且赋予这个 anchor box 类别标签也相对麻烦，需要计算和 ground-truth box 的 IoU。与之对应的，要覆盖 2d 空间和赋予类别都很容易，典型的 2d 表示包括 center point，corner point 等等。

回归 vs. 验证是一个很古老的问题，通常来讲求解验证问题比求解回归问题更容易，但是效率更低。在物体检测里面，通常的做法是先做粗验证，后用回归方法来做精确定位。基于 anchor 的设计就是典型的验证思路，后续回归只在一个很小的范围内进行。而 anchor-free 方法更多依赖回归来求

解问题，通常讲更难学习，但是因为有了 deep 的特征，以及多阶段的定位方法，效果能比肩依赖验证更多的 anchor-based 方法，同时框架更简洁。

RepPoints 不仅能用来解决物体检测问题，它在表示物体的精细结构上也有很大潜力，例如预测物体的 contour(后续会公布)。”

7.0.3 Retinanet

见 RetinaHead。

7.1 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 9: resnet

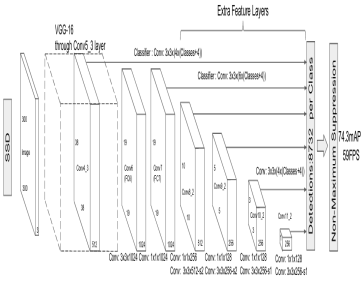


图 10: ssd

7.1.1 backbone 及改进

出于对Universal approximation theorem的考察，人们认为加深网络，在复杂问题上会带来更好的效果，但实验情况并非如此，于是 2015 年ResNet出现，旨在解决模型网络加深带来的性能下降问题 (degradation problem)。其结构参数见9。其中 $\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}$ 为 Basicbolck 基础块，$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}$ 为 Bottleneck 基础块，后接池化，全连接和概率 softmax 映射，Bottleneck 计算量为 Basicbolck 的 5.9%，池化替代全连接，更本质，一来计算量极大减小，二来过滤或者综合出了更有效的特征。一点需要注意一下，基础块都 × 了一个数，重复计算可以增加感受野，同时增加了非线性算算子个数，能提取更抽象的特征，因为上下层尺度只减小了一半，所以 stride=2 只会在重复块中的某一个上做操作，这个是可以调节的。

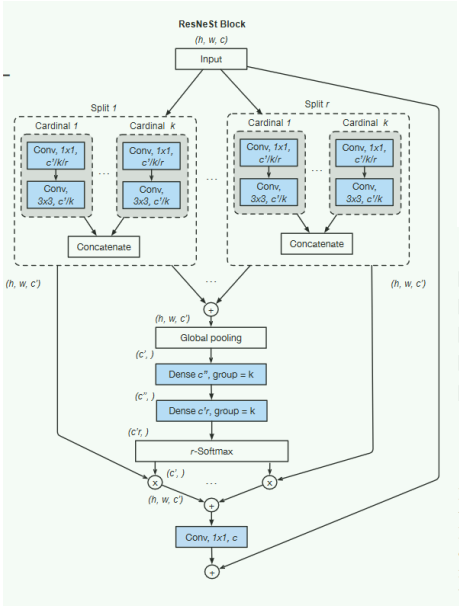


图 11: ResNeSt

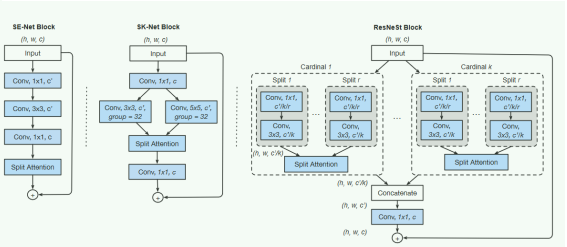
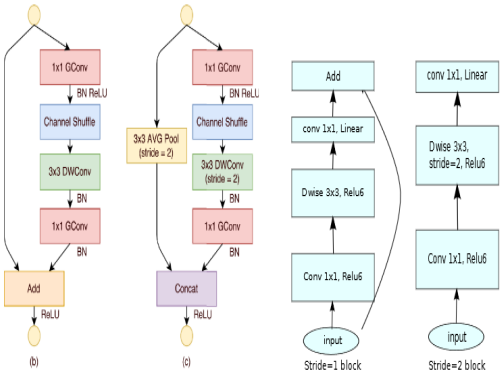


Fig. 1: Comparing our ResNeSt block with SE-Net [30] and SK-Net [38]. A detailed view of Split-Attention unit is shown in Figure 2. For simplicity, we show ResNeSt block in cardinality-major view (the featuremap groups with same cardinal group index reside next to each other). We use radix-major in the real implementation, which can be modularized and accelerated by group convolution and standard CNN layers (see supplementary material).

图 12: ResNeSt_SE_SK



(c) ShuffleNet [20]　　(d) Mobilenet V2

图 13: mobilenetv2

那么 ResNet 是怎么解决梯度消失问题的呢？由 Skip Connection 组成的残差块 (residual blocks)。

比如将第 l 层到 $l+1$ 层的激活函数改到 addition 之前 (参考ResNetV2) 的残差模块的表示方式：$\mathbf{x}_{l+1} = \mathbf{x}_l + \mathcal{F}(\mathbf{x}_l, \mathcal{W}_l)$，递归形式 $\mathbf{x}_L = \mathbf{x}_l + \sum_{i=l}^{L-1} \mathcal{F}(\mathbf{x}_i, \mathcal{W}_i)$，则其梯度如下：

$$\frac{\partial loss}{\partial \mathbf{x}_l} = \frac{\partial loss}{\partial \mathbf{x}_L}\frac{\partial \mathbf{x}_L}{\partial \mathbf{x}_l} = \frac{\partial loss}{\partial \mathbf{x}_L}\left(1 + \frac{\partial}{\partial \mathbf{x}_l}\sum_{i=l}^{L-1}\mathcal{F}(\mathbf{x}_i, \mathcal{W}_i)\right)$$

从 BP 公式我们知道梯度消失来源于梯度向量叠乘后太小，这里均有 +1，极大抑制了梯度消失问题。

ResNet 的改进文章有很多，主要可分为模型性能，模型轻量和功能扩展。如上 V2，Bag 包含了性能，轻量上的改进，而resnest集成了三方面的改进。resnest 结构见12，从结构图基本可以看到该残差模块的含义。现' 按图索骥'，给出对应的公式 (对比论文)，即可完成清晰的理解。

首先对输入特征图按 channel 分解成 $G = KR$ 部分，其中 K 是 cardinal group 数，R 是 radix group 数，一句话，组中组。每个原子特征图经过 F_i 映射成 U_i，$i \in \{1, 2, ..., G\}$。从图中能看到 F_i 为一个 $1 \times 1, 3 \times 3$ 卷积组成，这里好奇为何不是 131 结构，然后一个 R 小组接一个 Split Attention 结构，然后 K 个 Split Attention 结果 concatente 在一起和输入图相加，组成一个残差块。图中有个细节，c, c'，这个也是 channel 降维减小计算量吧 (出于这个考虑，也许可以圆一下为何不是 131 结构)。K 个 R 基组经过 F 映射后做求和处理得到 K 基组单位元的表示，$\hat{U}^k = \sum_{j=R(k-1)+1}^{Rk} U_j$，其中 $\hat{U}^k \in \mathbb{R}^{H \times W \times C/K}$，每个 K 特征图在 channel 维度上做一个全局池化得到该特征图的 Global contextual information $s^k \in \mathbb{R}^{C/K}$，其中第 c 个 channle $s_c^k = \frac{1}{H \times W}\sum_{i=1}^{H}\sum_{j=1}^{W}\hat{U}_c^k(i, j)$，这些 Global contextual information 做一个全连接映射 $\mathcal{G}$，得到 K 组中每个 R 组中各特征的 channel 的重要系数 $a_i^k(c)$，

$$a_i^k(c) = \begin{cases} \frac{\exp\left(\mathcal{G}_i^c\left(s^k\right)\right)}{\sum_{j=0}^{R}\exp\left(\mathcal{G}_j^c(s^k)\right)} & \text{if } R > 1 \\ \frac{1}{1+\exp\left(-\mathcal{G}_i^c(s^k)\right)} & \text{if } R = 1 \end{cases}$$

最后用此系数加权求和 R 组各特征图 U，其中第 c channel 为 $V_c^k = \sum_{i=1}^{R} a_i^k(c) U_{R(k-1)+i}$，得到 Attention 后的 K 组结果，然后 concat 一下得到映射特征，即 $V = \text{Concat}\left\{V^1, V^2, \ldots V^K\right\}$，最后执行 Skip Connection，$Y = V + X$ 得到残差块的最终输出。X 上可做操作，可调节。

以上基本囊括了 ResNet 的改进内容。接着说说 mmdet 的代码部分。

代码上，Resnet 由 BasicBlock，Bottleneck，make_res_layer 构成。其中 make_res_layer 重复构造残差模块。需要说明的是在 resnet.py 中引进了 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

def forward(self, x):
  # 首先是两个下采样:kernel_size=7和maxpool, 此时尺度为原始1/4
  x = self.conv1(x)
  x = self.norm1(x)
  x = self.relu(x)
  x = self.maxpool(x)
  # 然后是几个尺度减半的特征层 make_res_layer, 对应fpn层
  outs = []
  for i, layer_name in enumerate(self.res_layers):
    res_layer = getattr(self, layer_name)
    x = res_layer(x)
    if i in self.out_indices:
      # 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)
      outs.append(x)
  return tuple(outs)
```

说明: resnest 被 ECCV2020 strong recject 了，其中的丰富历史略。

7.2 Necks

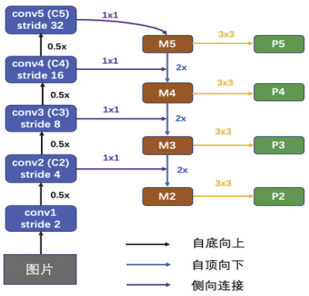


图 14: fpn

提到数字图像的多尺度，首先想到的是 David Lowe 于 1999 发表的 SIFT 算法。而 DL 中的多尺度结构 FPN，算了接了 SIFT 的衣钵。2017 年，Tsung-Yi Lin 等人提出FPN，其摘要中给出解释: A top-down architecture

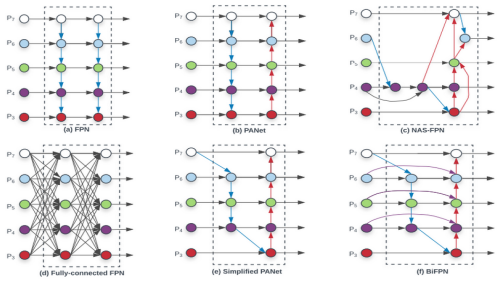


图 15: bifpn

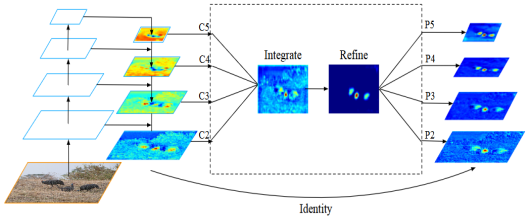


Figure 4: Pipeline and heatmap visualization of balanced feature pyramid.

图 16: bfpn

with lateral connections is developed forbuilding high-level semantic feature maps at all scales。从中我们看到几个关键词，自顶向下，横向连接，高级语义特征，各种尺度。

想象一下，一张图被 1/2 下采样了几次，那么同样的物体在不同从尺度上，表示的信息将至少对半折少 (像素信息)，那么合并各层信息将有助于有效信息的叠加，抑制某些物体的消失。因此容易看出这几个关键词的一些可变化的地方。自顶向下，自底向上，连接方式可从全连接开始缩减，获取高级语义特征的方式，不同尺度特征对齐的方式，更多尺度。若采用事后诸葛亮的逻辑，那么后续确实有一些文章是在围绕这个几个点做改进工作。比如看一眼15，不过 BiFPN增加了各层的权重因子，这和后来横飞的 Attention 是一致的。bifpn 的实现参考YAEDet.Pytorch，如果想看更好看的实现，可参考bifpn.pytorch，实现细节上和论文有出入 (YAEDet.Pytorch)，所以自行采坑。类似于简化全连接可参考HRFPN，看图就行，大同小异。另外 mmdet 中实现了BFPN，从16可以看到，是对原始 fpn 后的特征做了一次残差操作。具体就是特征做了一次合并和解耦 (融合，分离)，然后和输入特征相加。

fpn 的改进远不止以上内容，因为其他很多改进可以合并到**??**小目标改进上，所以这里暂时省去。

最后，FPN 的代码相对比较简单，这里选择三点说明。

```
# 1. 一行代码:
# add extra conv layers (e.g., RetinaNet)
extra_levels = num_outs - self.backbone_end_level + self.start_level

# 参数说明，num_outs: 最终fpn输出的尺度个数，backbone_end_level: 输入尺度的第几层作为基础fpn输出的最后一层，
# start_level: 输入尺度作为基础fpn输出尺度的开始层
# 上面没解释清楚，本来fpn没有额外加层的，你可以理解基础fpn就是原始的fpn，输入多少个层，选一个开始和结束，作为截断就是
#输出结果，现在有了额外层，所以，fpn从基础fpn变为最终fpn。num_outs就是你想输出的最终尺度个数，额外差的尺度特征，从
# 基础fpn被截断的层数(x=backbone_end_level - start_level)末尾开始，往下再下采样y=num_outs-x)即得到最终的fpn特征。

# 2. 一个基础模块: ConvModule，有conv_cfg， norm_cfg，act_cfg三个配置， 作为最小卷积配置单元，注意一下。

# 3. 额外加层: add_extra_convs为False时，Maskrcnn，Fasterrcnn 会在基础fpn上外增加num_outs - used_backbone_levels(x,如上)个max_pool层，
# 为True时，Retina增加额外y层3*3，stride=2的基础ConvModule模块。
```

另外经典 fpn 的自顶向下连接方式从14图中容易理解，注意代码中使用的 F.interpolate 支持 3,4,5 维的张量。

7.3 Heads

head 主要处理特征层上的分类和回归任务，它汇集了 anchor 生成，bbox 编码，target 设定，以及损失函数的计算。mmdet v2 版本将 head 分成两部分，one stage 的 dense_heads, two stage 的 roi_heads。因 one stage 更简单, 适用性更好，同时 fcos, fovea, reppoints 等 head 需独立实现，所以这里选取 anchor_head ，retina_head 以做简要说明。

7.3.1 AnchorHead

AnchorHead 的基本组成元素在初始化对象时完成，参数信息一目了然。

```
@HEADS.register_module()
class AnchorHead(nn.Module):
    """Anchor-based head (RPN, RetinaNet, SSD, etc.).
    """

    def __init__(self,
                 num_classes,
                 in_channels,
                 feat_channels=256,
                 anchor_generator=dict(
                     type='AnchorGenerator',
                     scales=[8, 16, 32],
                     ratios=[0.5, 1.0, 2.0],
                     strides=[4, 8, 16, 32, 64]),
                 bbox_coder=dict(
                     type='DeltaXYWHBBoxCoder',
                     target_means=(.0, .0, .0, .0),
                     target_stds=(1.0, 1.0, 1.0, 1.0)),
                 reg_decoded_bbox=False,    # 算损失函数时将预测的bbox解码(若True,一般效果不会好吧)
                 background_label=None,
                 loss_cls=dict(
                     type='CrossEntropyLoss',
                     use_sigmoid=True,
                     loss_weight=1.0),
                 loss_bbox=dict(
```

```
                        type='SmoothL1Loss', beta=1.0 / 9.0, loss_weight
    =1.0),
                    train_cfg=None,
                    test_cfg=None):
            super(AnchorHead, self).__init__()
```

其中每部分内容已在其他章节说明，在 head 中，我们只需要关心怎么给 anchor 打标签，头部网络怎么设计，loss 怎么计算，以及如何解析预测结果。这四部分内容分别见 (_get_targets_single, get_targets),(_init_layers),(loss_single, get_anchors, loss), (_get_bboxes_single,get_bboxes)。其中损失函数见7.4。

因 get_target, loss 在不同尺度上计算是类似的，于是作者设计了一个公共函数 multi_apply, 将同一操作作用于不同尺度上，得到所有结果。

```
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.

这里利用 _get_targets_single 函数来解释 multi_apply 函数:

```
def _get_targets_single(self,
                        flat_anchors,
                        valid_flags,
                        gt_bboxes,
                        gt_bboxes_ignore,
                        gt_labels,
                        img_meta,
                        label_channels=1,
                        unmap_outputs=True):
    """Compute regression and classification targets for anchors in
        a single image.
    """
    # 1. 有效anhcor的index
    inside_flags = anchor_inside_flags(flat_anchors, valid_flags,
                                       img_meta['img_shape'][:2],
                                       self.train_cfg.allowed_border)
    if not inside_flags.any():
```

```
        return (None, ) * 6
    # 2. 根据gt和有效anchor采样
    anchors = flat_anchors[inside_flags, :]
    # assign 见core章节
    assign_result = self.assigner.assign(
        anchors, gt_bboxes, gt_bboxes_ignore,
        None if self.sampling else gt_labels)
    sampling_result = self.sampler.sample(assign_result, anchors,
                                          gt_bboxes)

    #   3. 构造损失函数计算变量
    num_valid_anchors = anchors.shape[0]
    bbox_targets = torch.zeros_like(anchors)
    bbox_weights = torch.zeros_like(anchors)
    labels = anchors.new_full((num_valid_anchors, ),
                              self.background_label,
                              dtype=torch.long)
    #   3.1 label_weights: 记录张量上需要被计算的点, 细节见损失函数章节
    label_weights = anchors.new_zeros(num_valid_anchors, dtype=torch.float)
    #   3.2 对采样后的anchor做一些编码
    pos_inds = sampling_result.pos_inds
    neg_inds = sampling_result.neg_inds
    if len(pos_inds) > 0:
        if not self.reg_decoded_bbox:
            pos_bbox_targets = self.bbox_coder.encode(
                sampling_result.pos_bboxes, sampling_result.pos_gt_bboxes)
        else:
            pos_bbox_targets = sampling_result.pos_gt_bboxes
        bbox_targets[pos_inds, :] = pos_bbox_targets
        bbox_weights[pos_inds, :] = 1.0
        ...

    # 4. 将anchor unmap 回原图
    ...

    return (labels, label_weights, bbox_targets, bbox_weights, pos_inds,
            neg_inds)

#   在  get_targets中
#   调用\_get\_targets\_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)

#   然后调用_get_targets_single
result = multi_apply(
        self._get_targets_single,,
        concat_anchor_list,
        concat_valid_flag_list,
        gt_bboxes_list,
        gt_bboxes_ignore_list,
        gt_labels_list,
        img_metas,
        label_channels=label_channels,
        unmap_outputs=unmap_outputs)

(all_labels, all_label_weights, all_bbox_targets, all_bbox_weights,
pos_inds_list, neg_inds_list) = results[:6]

# 最后调用images_to_levels 将 target 按level排序(不同尺度的target会记录相应的anchor个数,作为level分界线)
```

此时 multi_apply 的 *args 参数:

- concat_anchor_list: get_anchor 而得, 是一个 list[list[Ten sors]] 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*K, 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

```
@force_fp32(apply_to=('cls_scores', 'bbox_preds'))
def loss(self,
             cls_scores,
             bbox_preds,
             gt_bboxes,
             gt_labels,
```

```
            img_metas,
            gt_bboxes_ignore=None):
    featmap_sizes = [featmap.size()[-2:] for featmap in cls_scores]
    assert len(featmap_sizes) == self.anchor_generator.num_levels

    device = cls_scores[0].device
    # 1.生成anchor
    anchor_list, valid_flag_list = self.get_anchors(
        featmap_sizes, img_metas, device=device)
    label_channels = self.cls_out_channels if self.use_sigmoid_cls else 1
    # 2. 标定anchor
    cls_reg_targets = self.get_targets(
        anchor_list,
        valid_flag_list,
        gt_bboxes,
        img_metas,
        gt_bboxes_ignore_list=gt_bboxes_ignore,
        gt_labels_list=gt_labels,
        label_channels=label_channels)
    if cls_reg_targets is None:
        return None
    (labels_list, label_weights_list, bbox_targets_list, bbox_weights_list,
        num_total_pos, num_total_neg) = cls_reg_targets
    num_total_samples = (
        num_total_pos + num_total_neg if self.sampling else num_total_pos)

    # anchor number of multi levels
    num_level_anchors = [anchors.size(0) for anchors in anchor_list[0]]
    # concat all level anchors and flags to a single tensor
    concat_anchor_list = []
    for i in range(len(anchor_list)):
        concat_anchor_list.append(torch.cat(anchor_list[i]))
    all_anchor_list = images_to_levels(concat_anchor_list,
                                       num_level_anchors)

    losses_cls, losses_bbox = multi_apply(
        self.loss_single,
        cls_scores,   # init_layers 网络头部预测, 调用forward_single
        bbox_preds,    # init_layers 网络头部预测
        all_anchor_list,
        labels_list,
        label_weights_list,
        bbox_targets_list,
        bbox_weights_list,
        num_total_samples=num_total_samples)
    return dict(loss_cls=losses_cls, loss_bbox=losses_bbox)
```

7.3.2 SSDHead

ssd 结构的检测网络，目前已有 ssd300,ssd512, 结构细节参考10. 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))

说明: 此为 1.x 版本。

7.3.3 RetinaHead

RetinaHead 继承 AnchorHead, 只需要实现 _init_layers 和 forward_single 即可，这点 retina_head 一看即明。需要额外注意一点的是，retina 在 fpn 后增加了几层 ConvModule。

7.3.4 FCOSHead

fcoshead 并非继承 anchorhead, 代码得从 0 解析，故略。

这里补一点 fcos 的论文简略解读 (每个算法细节都很多，具体实践也会有大量的额外 trick, 并非一点文字可以说明，具体坑还得自愿踩)。

物体检测的算法基本可以用以下这段话来描述:

在某一度量 (..) 下, 找出与人为标定的物体的框的集合, 利用这个集合按不同策略生成更有效的正负样本, 完成物体的位置预测.

以 IOU 为度量的模型有 Faster R-CNN, SSD 和 YOLOv2, v3 等. 它从标定框与虚拟框的偏移集中学习物体的标定框与虚拟框的映射关系, 以此来完成物体定位. 这里的隐含点为映射变量为物体的表示向量, 这是根据类别标签优化而得. FCOS 可以称为以 center-ness 为度量的检测模型. 它将标定框内的聚中心的所有点作为物体的有效表示点, 学习其与标定框的中心偏移量, 完成物体的检测.

因为两者度量方式不同, 导致后续操作差别比较大,iou 度量的 anchor-based 模型, 需要给每个位置点设计不同形状的 anchor 作为候选框, 然后在所有尺度上进行筛选 (或不同尺度分别筛选), 这个过程较为细致, 且超参数较多, 对不同任务较为敏感. 后者 anchor-free 则省去了 anhor 构造的过程, 可以直接在选出的点上做位置偏移预测，比较简单.

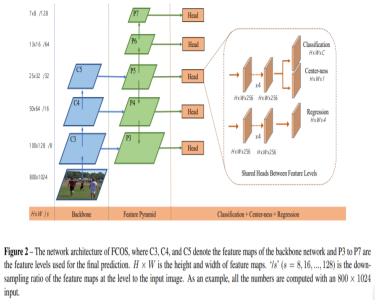


Figure 2 – The network architecture of FCOS, where C3, C4, and C5 denote the feature maps of the backbone network and P3 to P7 are the feature levels used for the final prediction. $H \times W$ is the height and width of feature maps. '/s' ($s = 8, 16, ..., 128$) is the down-sampling ratio of the feature maps at the level to the input image. As an example, all the numbers are computed with an 800×1024 input.

图 17: fcos

从以上结构图能清晰的看到 FCOS 算法的的三大模块. 它的核心点在 Classification + Center-ness + Regression. (代码以 maskrcnn-benchmark 为基础, 核心在 rpn/fcos 的三个文件中 (后续).)

这里有几点需要说明:

- 限制不同特征层的预测框范围 $\{P_3, P_4, P_5, P_6, P_7\}$ 对应范围区间 $0, 64, 128, 256, 512$ and ∞.
- P_i, 层的位置回归做如下动态调整 $\exp(s_i x)$ 以适应不同特征层回归不同范围.
- 位置预测的是中心点距离物体左上下右的距离, 且位置回归函数是 UnitBox 中的 IOU loss.
- IOU loss:
$$\begin{aligned} &\square \text{ Ground that } \tilde{x} = (\tilde{x}_t, x_b, \tilde{x}_l, \tilde{x}_r) \\ &\square \text{ predictors} \quad x = (x_t, x_b, x_l, x_r) \\ &\ell_2 \text{ loss } = \|\square - \square\|_2^2 \\ &\text{IoU loss } = -\ln \frac{\text{Intersection}(\square,\square)}{\text{Union}(\square,\square)} \end{aligned}$$
- centerness $^* = \sqrt{\frac{\min(l^*,r^*)}{\max(l^*,r^*)} \times \frac{\min(t^*,b^*)}{\max(t^*,b^*)}}$ 以此来筛选更有代表的物体表示点.

- 在多个 box 内的点, 选择 box 面积最小的那个作为其 label box.

最终损失函数为:

$$L\left(\{\boldsymbol{p}_{x,y}\},\{\boldsymbol{t}_{x,y}\}\right)=\frac{1}{N_{\text{pos}}}\sum_{\boldsymbol{x},y}L_{\text{cls}}\left(\boldsymbol{p}_{x,y},c^*_{x,y}\right)+\frac{\lambda}{N_{\text{pos}}}\sum_{x,y}\mathbb{I}_{\{c^*_{x,y}>0\}}L_{\text{reg}}\left(\boldsymbol{t}_{x,y},\boldsymbol{t}^*_{x,y}\right)$$

7.4 Losses

7.4.1 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:

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

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.

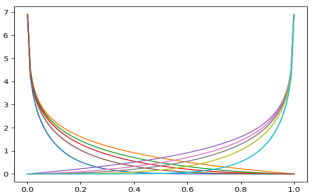


图 18: FcoalLoss

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.

7.4.2 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:

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

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

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

这里使用的 $\mathrm{IoU_B}$ 定义为:

$$\begin{matrix}\mathrm{IoU_B}\left(x,b_t\right)=\max\left(0,\frac{w_t-2|\Delta x|}{w_t+2|\Delta x|}\right)\\ \mathrm{IoU_B}\left(w,b_t\right)=\min\left(\frac{w}{w_t},\frac{w_t}{w}\right)\end{matrix}.$$

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。

第八节　简单实践

8.1　更改模型

8.1.1　增加模块

本身就包含了很多更改配置文件。

8.1.2　模型瘦身

已有模型参数思路: 各自参数往小的方向改，比如结构的重复次数，channel 的数目，或者采用更多的 $1*1$ 替换 $3*3$，甚至可使用 $m*n$ 联合 $n*m$ 替换 $m*m$ 卷积等。检测模型基本组件 (mmdet 中的理解) 替换:backbone 更改为熟知的轻量 backbone，比如 mobilev2。自行设计思路: 这就考虑任务特性，自身对模型的认识水平和经验问题了，后面补充一个样例，作为参考。

8.2　抽离模型

8.2.1　retinanet_resnet18

只需要将需要的部分代码抽离出来，作为单独的 inference 模型，可参考本地代码 retinanet_resnet18。

8.3　新增模型

8.3.1　centernet

先看原始论文 Objects as Points 以及官方代码。然后 mmsdet/centernet/core 下的 gaussian_radius 函数的解读可参考 centernet heatmap 解读，对推广的 FocalLoss 的更细致的分析可参考 centernet 解读，想做一些其他任务尝试的可参考 centernet 一些尝试。

8.3.2　代码说明

这里只是将 mmdet 的注册方式和配置文件类拿来用了一下，从原始代码centernet office code 将需要的部分抽离出来，将 pose 模型改成了 backbone+heathead 结合成 HeatMap 的形式，其他文件做了和自己对 mmdet 的理解保持一致方面的些调整。很多部分没有按照 mmdet 的方式去改写，想了下改动有点大，数据，pipeline，训练方式等，就算了。

datasets 将原始的数据类 ___getitem___ 分开实现的方式合并了，原始这样实现主要是不同的方向不同的 getitem 所致吧。ops 里的 dcn 等就没有加进来了。这个有参考抽离模型一节。

一些说明点：

0 没有 fpn，resnet 的 stem_layer 出来后，做了三次下采样，然后又三次转置卷积回去了，也就是得到了 heatmap 特征图。

1 dcn 替换在转置卷积部分

2 detector 中的 base_detector，cdetdector 只是用来测试的

3 loss 的实现上，张量的选取 tensor[index] 没有 tensor * mask 速度快

4 核心地方在 core 中，这和 mmdet 中 anchor 相关的意思一致，core/imgage.py 中的射影点是原始图 scale 后的中心点，中心点关于 scale/2 左移，以及这两点关于图像原点组成的一个三角形 (见 get_3rd_point)。

5 ___getitem___ 中将输入图像射影到 (512, 512) 大小 (2**n)，然后在构造的 512/4=128 大小的 heatmap mask 上赋值对应的 hm，wh，reg，其中 hm 是将原始框射影到 128 大小的 heatmap mask 上，根据对应的框中心，进行赋值；wh 就是射影后的 w，h 值，reg 为中心向上取整后的误差，也即中心偏移量。

整体上 centernet 的思想是非常简洁的，效果也很好，以及后续的 centertrack 的 end2end 式追踪方案，不过后者的前后帧处理方式优化点似乎较多。

8.4　numpy,torch 某些基础函数

- 损失函数部分
  - torch 相关:new_full,nonzero,numel,expand,where: 分段函数。
- 模型评估部分
  - numpy 相关:cumsum,finfo,vstack,hstack,concatenate,argmax,
  - argsort,zeros_like,newaxis
  - torch 相关:type_as,stack,x[:,None]*y[None,:],contiguous,view,
  - max(和 numpy 的区别)
- 其他
  - permute, transpose

第九节　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.

9.1　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).

9.1.1　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).

9.1.2 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++:

9.1.3　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7.0.1.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.

9.1.4　Anchor Free 系列

FCOS 见7.3.4, CenterNet 见8.3.1。

9.2　总结

检测算法总结略。时间较紧张，很多标点错误，一些内容很仓促，实验也欠缺，先就这样吧。剩下的一点小想法 (mmsdet)，也不知道会不会有机会完成。去寻找其他有意义的事情吧。

第十节　官方文档 2.0 伪译

2.0 相比 1.x，就代码组织上，在模块化这方面有了更好的贯彻。能拆分的就拆分，比如配置文件，数据的信息整合、变换、采样迭代，以前版本命名不严格的一律改掉，比如 anchor_heads。

10.1　配置系统

1.x 版本是将所有配置信息放到一个 x.py 配置文件中，2.0 增加了配置文件的模块化和继承能力，这样在实验中能提高组合不同部分的效率。执行　python tools/print_config.py /PATH/TO/CONFIG 能看到配置信息。–options xxx.yyy=zzz 可看到更新信息。

基础配置文件在 config/_base_ 中，有 dataset, model, schedule, default_runtime 四个部分，对应 1.x 版本单个配置文件的不同部分。

10.2　使用预训练模型

将 coco 数据训练的模型作为 CitySpace 等数据预训练模型，需要做以下五处改动

10.2.1　继承基础配置文件

基础模型继承自 mask_rcnn_r50_fpn，数据继承自 cityscapes 风格，训练 schedules 继承自默认的 default_runtime，在配置文件顶部增加如下代码:

```

_base_ = [
  '../_base_/models/mask_rcnn_r50_fpn.py',
  '../_base_/datasets/cityscapes_instance.py', '../_base_/default_runtime.
  py'
]
```

10.2.2　更改头部

如果新旧模型的类别不同，则需要改一下类别数目。

```

model = dict(
  pretrained=None,
```

```
    roi_head=dict(
        bbox_head=dict(
            type='Shared2FCBBoxHead',
            in_channels=256,
            fc_out_channels=1024,
            roi_feat_size=7,
            num_classes=8,          # new num_classes
            target_means=[0., 0., 0., 0.],
            target_stds=[0.1, 0.1, 0.2, 0.2],
            reg_class_agnostic=False,
            loss_cls=dict(
                type='CrossEntropyLoss', use_sigmoid=False, loss_weight=1.0)
    ,
            loss_bbox=dict(type='SmoothL1Loss', beta=1.0, loss_weight=1.0)),
        mask_head=dict(
            type='FCNMaskHead',
            num_convs=4,
            in_channels=256,
            conv_out_channels=256,
            num_classes=8,
            loss_mask=dict(
                type='CrossEntropyLoss', use_mask=True, loss_weight=1.0)))
```

预训练的模型权重除最后的预测层不会加载预用模型权值，其他均会被加载。

10.2.3　更改数据

仿照 VOC, WIDER FACE, COCO and Cityscapes 数据类重写自己的数据整合方式。改一下数据类的名字即可。具体改写细节见下小节。

10.2.4　改写训练 schedule

优化器，训练超参数等的修改。

```

# optimizer
# lr is set for a batch size of 8
optimizer = dict(type='SGD', lr=0.01, momentum=0.9, weight_decay=0.0001)
optimizer_config = dict(grad_clip=None)
# learning policy
lr_config = dict(
  policy='step',
  warmup='linear',
  warmup_iters=500,
```

```
    warmup_ratio=0.001,
    # [7] yields higher performance than [6]
    step=[7])
total_epochs = 8  # actual epoch = 8 * 8 = 64
log_config = dict(interval=100)
```

10.2.5　使用预训练模型

```
load_from = 'https://s3.ap-northeast-2.amazonaws.com/open-mmlab/mmdetection/
    models/mask_rcnn_r50_fpn_2x_20181010-41d35c05.pth'
```

10.3　增加新数据类

10.3.1　转成公用格式

最简单的方式就是将自己的数据脚本转换成 coco 或者 voc 格式。然后更改配置文件中的数据信息。比如 coco 格式，在 configs/my_custom_config.py 中有:

```
...
# dataset settings
dataset_type = 'CocoDataset'
classes = ('a', 'b', 'c', 'd', 'e')     #   自己的五类名字
...
data = dict(
  samples_per_gpu=2,
  workers_per_gpu=2,
  train=dict(
    type=dataset_type,
    classes=classes,
    ann_file='path/to/your/train/data',
    ...),
  val=dict(
    type=dataset_type,
    classes=classes,
    ann_file='path/to/your/val/data',
    ...),
  test=dict(
    type=dataset_type,
    classes=classes,
    ann_file='path/to/your/test/data',
    ...))
...
```

10.3.2　中间格式

mmdet 提供了和 coco，voc 等兼容的中间格式：

```
[
  {
    'filename': 'a.jpg',
    'width': 1280,
    'height': 720,
    'ann': {
      'bboxes': <np.ndarray, float32> (n, 4),
      'labels': <np.ndarray, int64> (n, ),
      'bboxes_ignore': <np.ndarray, float32> (k, 4),
      'labels_ignore': <np.ndarray, int64> (k, ) (optional field)
    }
  },
  ...
]
```

使用方式：

1. 在线转换
   写一个继承自 CustomDataset 的类，并重写 load_annotations(self, ann_file) 和 get_ann_info(self, idx) 两个方法。参考 CocoDataset，VOCDataset。

2. 离线转换
   将标注文件转成中间格式，保存成 pickle 或 json 文件，参看 pascal_voc.py，然后调用 CustomDataset 即可。

一个例子，假设标注文件格式如下

```
#
000001.jpg
1280 720
2
10 20 40 60 1
20 40 50 60 2
#
000002.jpg
1280 720
3
50 20 40 60 2
20 40 30 45 2
30 40 50 60 3
```

我们可以写一个继承自 CustomDataset 的新类如下:

```
import mmcv
import numpy as np

from .builder import DATASETS
from .custom import CustomDataset

@DATASETS.register_module()
class MyDataset(CustomDataset):

  CLASSES = ('person', 'bicycle', 'car', 'motorcycle')

  def load_annotations(self, ann_file):
    ann_list = mmcv.list_from_file(ann_file)

    data_infos = []
    for i, ann_line in enumerate(ann_list):
      if ann_line != '#':
        continue

      img_shape = ann_list[i + 2].split(' ')
      width = int(img_shape[0])
      height = int(img_shape[1])
      bbox_number = int(ann_list[i + 3])

      anns = ann_line.split(' ')
      bboxes = []
      labels = []
      for anns in ann_list[i + 4:i + 4 + bbox_number]:
        bboxes.append([float(ann) for ann in anns[:4]])
        labels.append(int(anns[4]))

      data_infos.append(
        dict(
          filename=ann_list[i + 1],
          width=width,
          height=height,
          ann=dict(
            bboxes=np.array(bboxes).astype(np.float32),
            labels=np.array(labels).astype(np.int64))
        ))

    return data_infos

  def get_ann_info(self, idx):
    return self.data_infos[idx]['ann']
```

```

# 配置文件做如下更改:

dataset_A_train = dict(
    type='MyDataset',
    ann_file = 'image_list.txt',
    pipeline=train_pipeline
)
```

数据合并，Repeat 或 Concatemate，顾名思义，将同一种数据重复多次，或不同数据 concate 成一个更大的数据。

如果你只想训练某数据的指定类别，只需要做如下改动:

```
classes = ('person', 'bicycle', 'car')
# classes = 'path/to/classes.txt'  # 或者类别从文件中读取
data = dict(
  train=dict(classes=classes),
  val=dict(classes=classes),
  test=dict(classes=classes))
```

10.4　自定义数据管道

经典的数据管道如下:

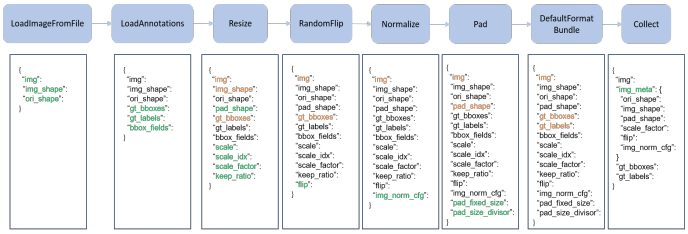


图 19: data pipeline

蓝色块为管道算子，一个算子为一个数据增强算法，从左到右，依次字典进，字典出。关于数据结构，可参考第二节数据处理。这里绿色为算子作用后新增的 keys，橙色为算子作用与已有 keys 的 values 后的更新标记。

10.4.1　扩展 pipelines

```

# 1. 实现新增强函数 my\_pipeline.py
# 和pytorch原始transforms中的增强方式一样, 实现__call__方法的类即可
```

```python
from mmdet.datasets import PIPELINES

@PIPELINES.register_module()
class MyTransform:

    def __call__(self, results):        # 输入的是mmdet设定的字典格式
        results['dummy'] = True
    return results

# 2. 导入新类.
from .my_pipeline import MyTransform

# 3. 配置文件调用

img_norm_cfg = dict(
    mean=[123.675, 116.28, 103.53], std=[58.395, 57.12, 57.375], to_rgb=True
    )
train_pipeline = [
    dict(type='LoadImageFromFile'),
    dict(type='LoadAnnotations', with_bbox=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='MyTransform'),
    dict(type='DefaultFormatBundle'),
    dict(type='Collect', keys=['img', 'gt_bboxes', 'gt_labels']),
]
```

10.5　增加新模块

每一个组件都如上小节所给的三部曲，实现，导入，修改配置文件。每一步，mmdet 中都有一些范例，分别看一下，就能实现改写。

10.5.1　优化器

一些模型可能需要对某部分参数做特殊优化处理，比如批归一化层的权重衰减。我们可以通过自定义优化器构造函数来进行细粒度参数调优。

```python
from mmcv.utils import build_from_cfg

from mmdet.core.optimizer import OPTIMIZER_BUILDERS, OPTIMIZERS
from mmdet.utils import get_root_logger
from .cocktail_optimizer import CocktailOptimizer
```

```
6
7  @OPTIMIZER_BUILDERS.register_module
8  class CocktailOptimizerConstructor(object):
9
10   def __init__(self, optimizer_cfg, paramwise_cfg=None):
11
12   def __call__(self, model):
13
14     return my_optimizer
```

10.5.2　开发新组件

- backbone，参考 MobilenetV2
- neck, 参考 PAFPN
- head 参考 Retinaface
- roi extractor，　参考 DCN ROIAlign

关于 head 相关组件，核心的点我认为在于数据的流向。基础变换层由 forward 函数得到变换的结果，怎么放到 Loss 中去，其中所涉及到的数据细节操作，是比较关键的。mmdet 中会在 head 模块中实现对应的 loss 函数，最终被汇集到检测模型的 loss 中去。分别调用各自的 loss.step, 进行权重更新。

这里贴一个新增 loss 样例:

```
1
2  # 1.在 mmdet/models/losses/my_loss.py实现新的box回归函数
3  import torch
4  import torch.nn as nn
5
6  from ..builder import LOSSES
7  from .utils import weighted_loss
8
9  @weighted_loss          #  加权损失函数，可参考损失函数章节
10 def my_loss(pred, target):
11   assert pred.size() == target.size() and target.numel() > 0
12   loss = torch.abs(pred - target)
13   return loss
14
15 @LOSSES.register_module
16 class MyLoss(nn.Module):
17
```

```
  def __init__(self, reduction='mean', loss_weight=1.0):
    super(MyLoss, self).__init__()
    self.reduction = reduction
    self.loss_weight = loss_weight

  def forward(self,
        pred,
        target,
        weight=None,
        avg_factor=None,
        reduction_override=None):
    assert reduction_override in (None, 'none', 'mean', 'sum')
    reduction = (
      reduction_override if reduction_override else self.reduction)
    loss_bbox = self.loss_weight * my_loss(
      pred, target, weight, reduction=reduction, avg_factor=avg_factor)

# 2. 然后在mmdet/models/losses/__init__.py.中注册
from .my_loss import MyLoss, my_loss

# 3. 配置文件使用
loss_bbox=dict(type='MyLoss', loss_weight=1.0))
```

10.6　1.x 模型升级到 2.0

执行脚本　tools/upgrade_model_version.py。可能有小于 1% 的绝对 AP 减小，具体可参见 configs/legacy。

10.7　2.0 和 1.x 的不同之处

主要有四点不同：坐标系，基础代码约定，训练超参数，模块设计。

10.7.1　坐标系

新坐标系与detectron2一致 treats the center of the most left-top pixel as (0, 0) rather than the left-top corner of that pixel. 这句话的意思是将 $bbox = [x1, y1, x1 + w - 1, y1 + h - 1]$ 改为 $bbox = [x1, y1, x1 + w, y1 + h]$，这样更加自然和精确 (假设长或宽为 1, 则 box 退缩为点或线，这是有问题的)，同理 xyxy2xywh 的长宽就不在 +1 了，生成的 anhor 的中心偏移也不在是 0.5 而是 0 了。与此相关的改动有 Box 的编解码，与 iou 计算相关的 nms，assinger。另外，现在的坐标为 float，1 .xx 为 int，与此相关的有

ahchor 与特征网格的中心对齐问题，这对 anchor-based 的方法在性能上有一定影响 (变好)，ROIAlign 也能更好的对齐，mask cropping and pasting 更精准，利用新的 RoIAlign 去 crop mask targets，会得到更好的结果，因为没有取整等误差了，而且在训练上也有 0.1s/iter 的速度提升 (少了取整操作)。

10.7.2 Codebase Conventions

- 类别设定，1.x 中 0 为背景，$[1, k]$ 为 k 类对象，2.0 中 k 为背景，$[0, k-1]$ 为 k 类对象。
- bbox 分配方案就低质量分配上得到了改进。支持了更多长宽尺度输入，以上均有微弱性能提升。
- 配置名称约定改动为 $[model]\_(modelsetting)\_[backbone]\_[neck]\_(normsetting)\_(misc)\_(gpuxbatch)\_[schedule]\_[dataset].py$，

10.7.3 训练超参数

一些训练参数的优化

- nms 后的 rpn 的 proposals 从 2000 改为 1000,(nms_post=1000, max_num=1000)，mask, bbox AP 有 0.2% 的提升。
- Mask, Faster R-CNN 框回归损失函数 Smooth L1 改为 L1 带来 0.6% 的提升,Cascade R-CNN and HTC 保持原样。
- RoIAlign layer 采样数设置为 0 ，0.2% 提升
- 默认设置不在使用梯度截断，这样训练更快，但 RepPoints 保持是为了训练稳定以及更好的结果。
- 默认的 warmup ratio 从 1/3 改为 0.001，这样更平滑，同时也是因为去掉了梯度截断。

10.8 版本变化补充

2.0 相对于 1.x 版本，从代码上主要有这些改变:

1. core 内部做了一些改写，比如 anchor 中的 tagret 移走了，增加了分割数据结构，抽象了 bbox 的编解码, 新增 iou_calculators，看起来 evaluation 中的 bbox_overlaps 不是最优的。

2. 数据类做了更彻底的分解，将 1.x 版本的 loader 中的 sample 分离成和 pipeline 同级别，data_loader 放到数据类中的 builder，结构更清晰。

3. models 中 anchor_heads 改为 dense_heads；bbox_heads, mask_heads 等 two stage 相关的合并到 roi_heads 中 (和 2 思想一致)

4. 所有 registry.py 均合并到 builder.py 中，更为简洁。

5. necks 增加了 pafpn, detectors 增加了 fsaf,core/mask

6. 配置文件采用了组合继承的方式。

7. 优化了训练，测试流程，速度提升明显，版本支持 pytorch1.5，1.2 以下均不在支持，定制化模块做了优化。