

Lecture 6

实践 MMDetection

王若晖
2022年3月

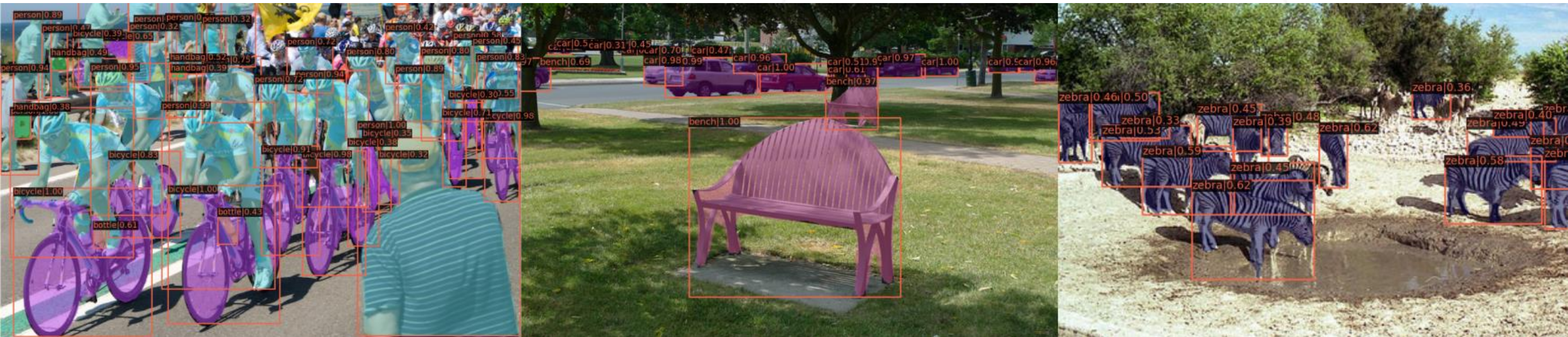
▶ 本节内容:

- MMDetection 项目概览
- MMDetection 运行环境搭建 (基于Openbayes计算平台)
- 使用 MMDetection 进行模型推理
- 使用 MMDetection 训练模型, 检测图像中水果



github.com/open-mmlab/mmdetection

- 2018-10 发布
- 2019-07 v1.0
- 2020-05 v2.0





任务支持

目标检测

实例分割

覆盖广泛

440+ 个
预训练模型

60+ 篇
论文复现

常用学术数据集

算法丰富

两阶段检测器

一阶段检测器

级联检测器

无锚框检测器

Transformer

使用方便

训练工具

测试工具

推理 API

科研论文



2018 年至今

谷歌学术引用 **超过 943 次**;

仅计算机视觉三大顶会上

被 **超过 100 篇论文** 作为基础代码库;

工业落地



商汤、腾讯、阿里、华为、
国内外初创公司,



学术比赛

COCO 2018 实例分割**冠军**

COCO 2019 实例分割**冠军**

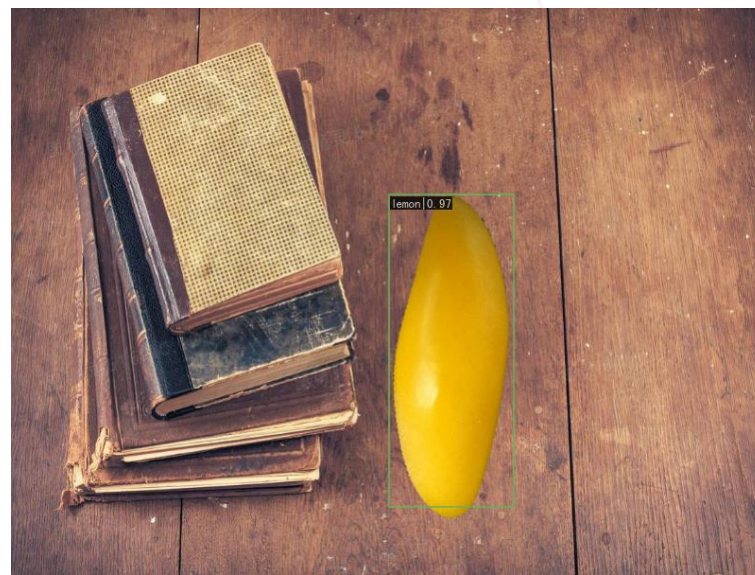
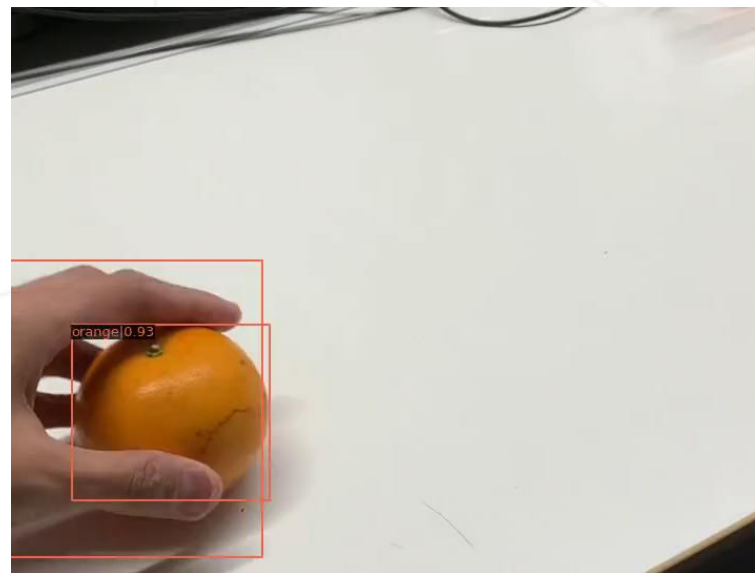
Open Images 2019 物体检测**冠军**

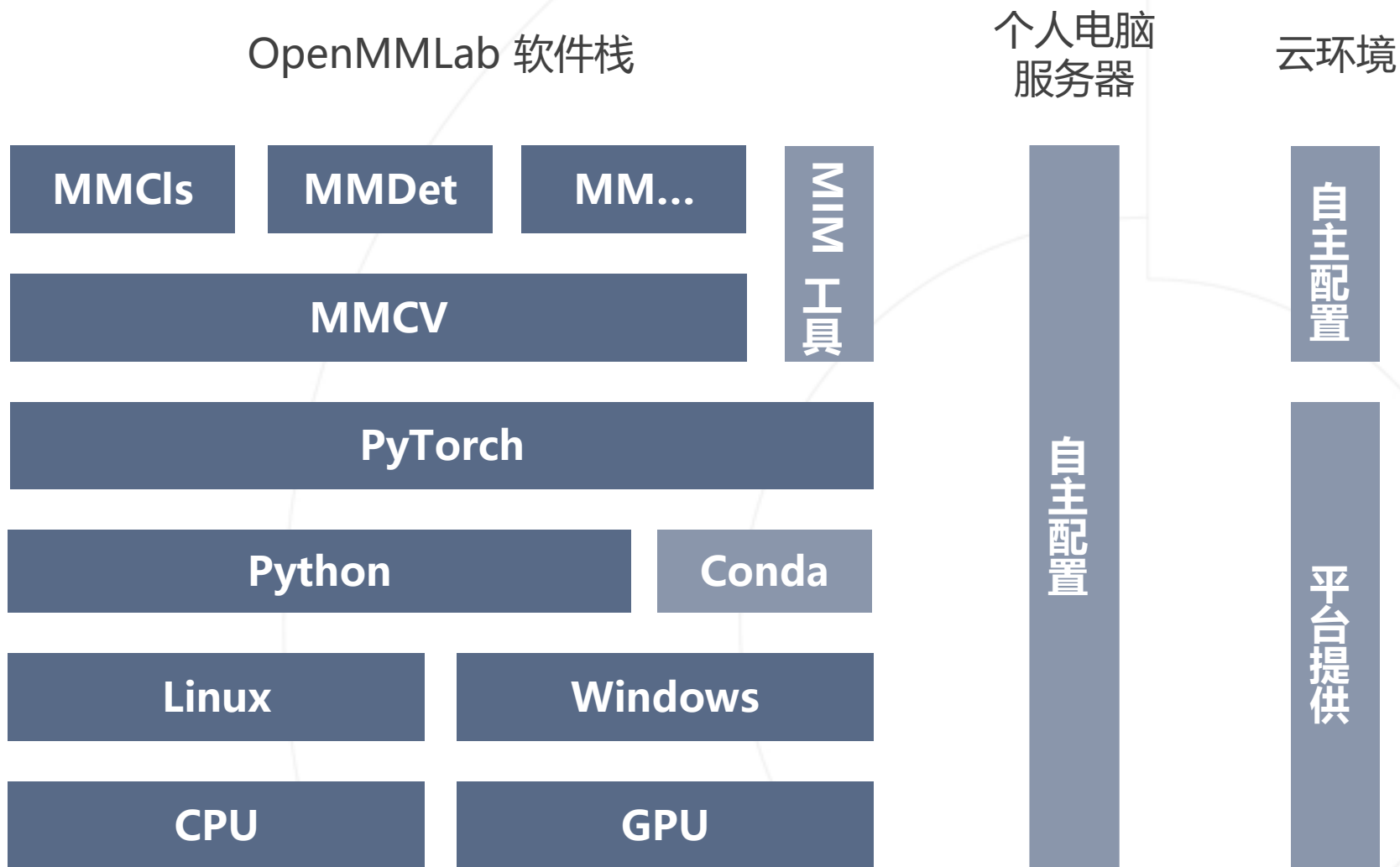
Global Wheat Detection**冠军**

Crowd Human 人体检测**冠军**

Materialist(FGVC6) 2019**冠军**

- MMDetection 提供 400 余个性能优良的预训练模型，开箱即用，几行 Python API 即可调用强大的检测能力
- MMDetection 涵盖 60 余个目标检测算法，并提供方便易用的工具，经过简单的配置文件改写和调参就可以训练自己的目标检测模型





➤ 深度学习模型的训练涉及几个方面：

- 模型结构 模型有几层、每层多少通道数等等
- 数据集 用什么数据训练模型：数据集划分、数据文件路径、数据增强策略等等
- 训练策略 梯度下降算法、学习率参数、batch_size、训练总轮次、学习率变化策略等等
- 运行时 GPU、分布式环境配置等等
- 一些辅助功能 如打印日志、定时保存checkpoint等等

- 在 MMDetection（及其他 OpenMMLab 项目中），所有这些项目都涵盖在一个**配置文件**中，一个配置文件定义了一个完整的训练过程
 - model 字段定义模型
 - data 字段定义数据
 - optimizer、lr_config 等字段定义训练策略
 - load_from 字段定义与训练模型参数文件

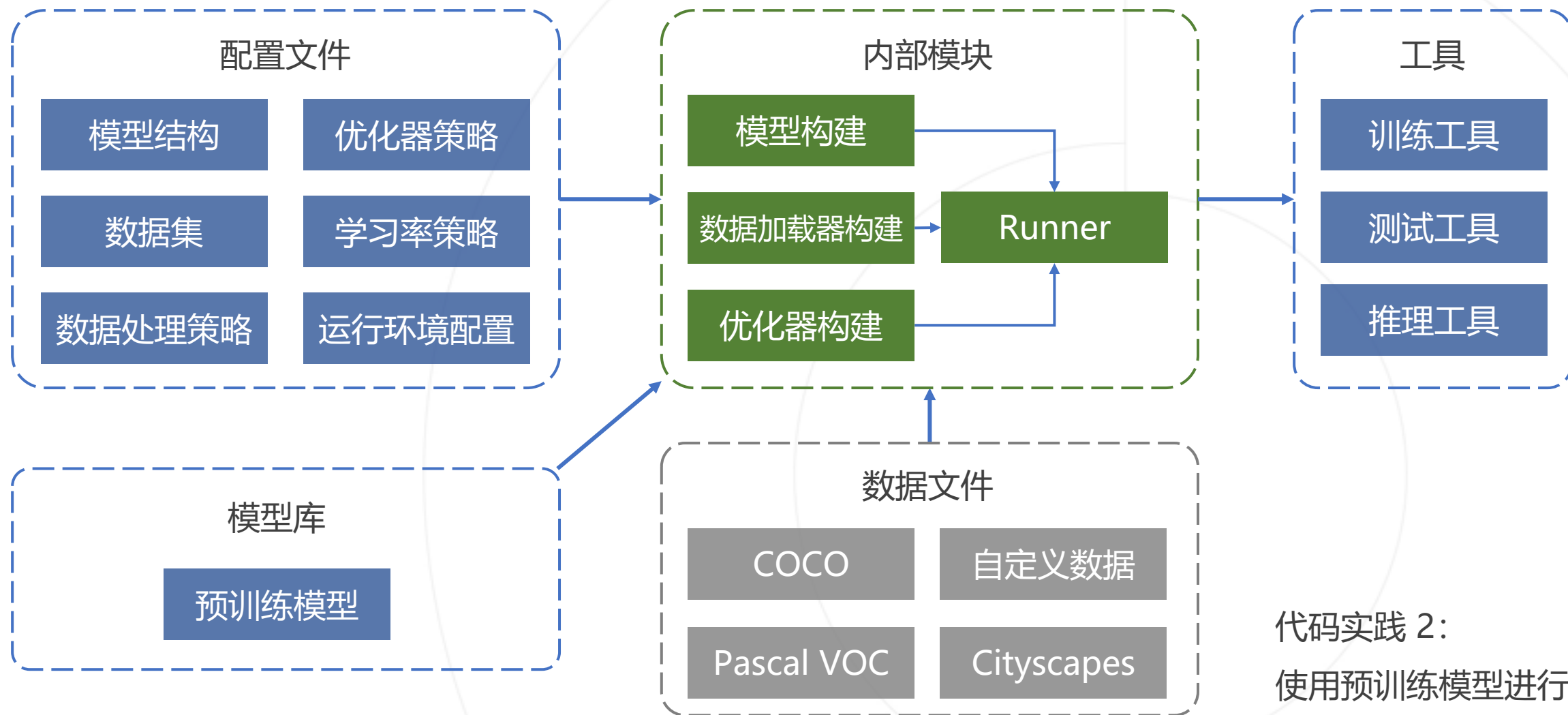
open-mmlab / mmdetection

<https://github.com/open-mmlab/mmdetection>

dev_scripts	
github	
configs	配置文件
demo	
docker	
docs	
mmdet	核心工具包
requirements	
resources	
tests	
tools	训练、推理、测试工具

众多预训练模型可以从这里找到

apis	训练、推理、测试的高层次 API
core	anchor、bbox、mask 等模块的实现
datasets	数据集支持、数据预处理与数据增强
models	检测模型的实现
utils	辅助工具
__init__.py	
version.py	



同样适用于其他 OpenMMLab 工具包

代码实践 2:
使用预训练模型进行推理

通常基于微调训练：

- 使用基于COCO预训练的检测模型作为梯度下降的“起点”
- 使用自己的数据进行“微调训练”，通常需要降低学习率

具体到 MMDetection，需要：

- 选择一个基础模型，下载对应的配置文件和预训练模型的参数文件
- 将数据整理成MMDetection支持的格式，如COCO格式或者自定义格式
- 修改配置文件：
 - 修改配置文件中的数据路径
 - 修改模型的分类头
 - 设置加载预训练模型
 - 修改优化器配置（学习率、训练轮次等）
 - 修改一些杂项

配置文件的修改可以通过**继承**的方式，不用把一整个配置文件贴过来再一条一条改

代码实践 3：

训练水果检测的模型

微软于2014年提出，最常用的是2017年版本

- 全集 33W 张图像
- 针对多种任务进行了标注
- 80类、150W 物体标注用于**目标检测与实例分割**



完整的 COCO 数据集包含一系列单独存放的图片，和若干“标注文件”（annotation files），每个标注文件定义了一个数据子集，包含对应的图片路径，以及图片上的所有标注信息



Data Explorer

Version 2 (27.6 GB)

- ▼ coco2017
 - ▼ annotations
 - { captions_train2017.json
 - { captions_val2017.json
 - { instances_train2017.json
 - { instances_val2017.json
 - { person_keypoints_train2017.json
 - { person_keypoints_val2017.json
 - ▶ test2017
 - ▶ train2017
 - ▼ val2017
 - 000000000139.jpg
 - 000000000285.jpg
 - 000000000632.jpg
 - 000000000724.jpg
 - 000000000776.jpg

MMDetection 会按照 COCO 的格式去读取数据集，因此我们也可以将自己的数据集按照 COCO 的格式进行组织，MMDetection 就可以正确读取数据和标注信息

所有标注信息存储在一个 JSON 对象中
包含以下字段:

```
{  
  "info" : info,  
  "images" : [image],  
  "annotations" : [annotation],  
  "licenses" : [license],  
  "categories" : [categories],  
}
```

所有图像的信息存储在一个列表中, 每个图像对应一个 JSON 对象, 包含以下字段

```
image {  
  "id" : int,  
  "width" : int,  
  "height" : int,  
  "file_name" : str,  
  "license" : int,  
  "flickr_url" : str,  
  "coco_url" : str,  
  "date_captured" : datetime,  
}
```

所有标注信息存储在一个 JSON 对象中
包含以下字段：

```
{  
  "info" : info,  
  "images" : [image],  
  "annotations" : [annotation],  
  "licenses" : [license],  
  "categories" : [categories],  
}
```

所有类别的信息存储在一个列表中，每个类别包含以下字段：

```
categories [{  
  "id" : int,  
  "name" : str,  
  "supercategory" : str,  
}]
```

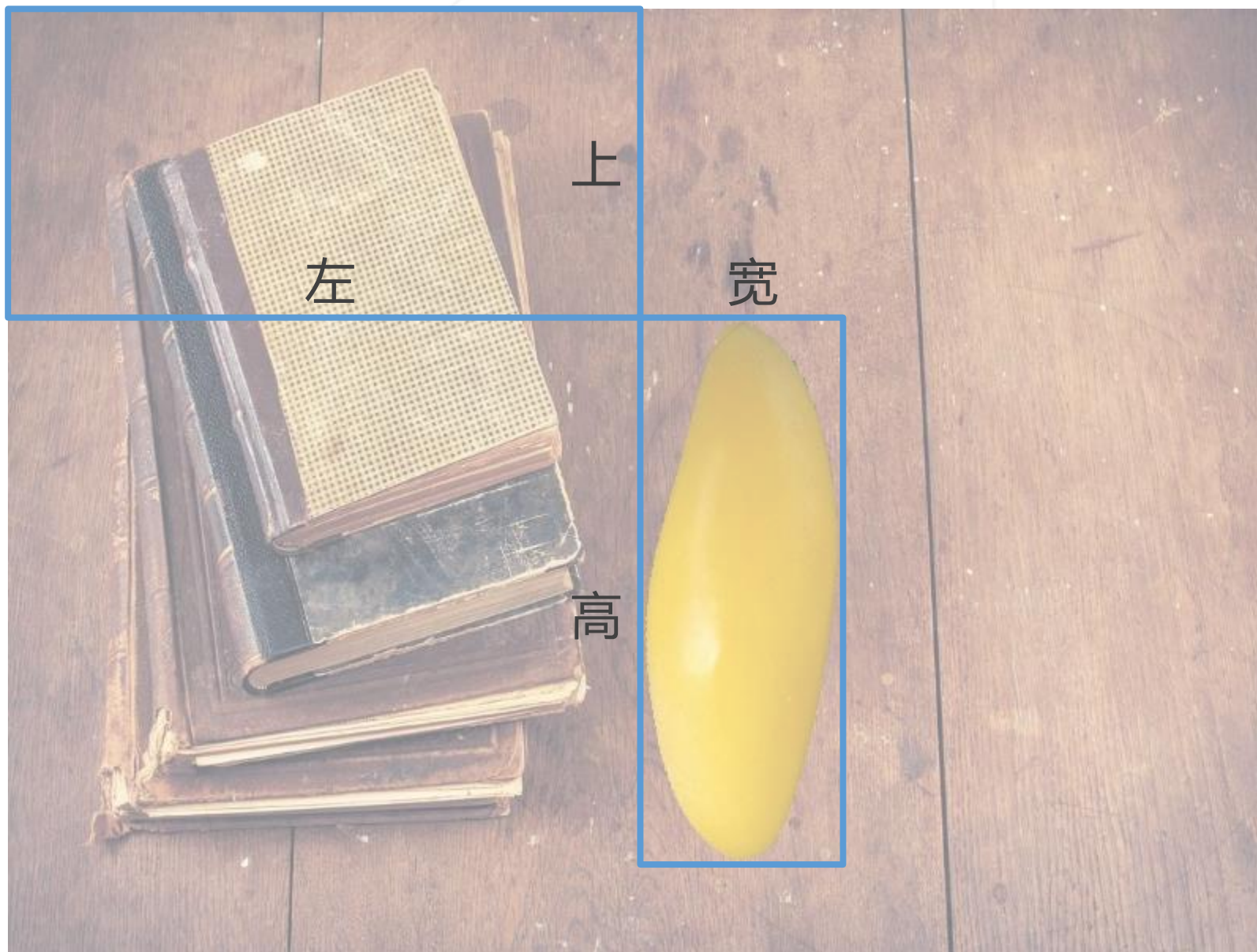


所有标注信息存储在一个 JSON 对象中
包含以下字段:

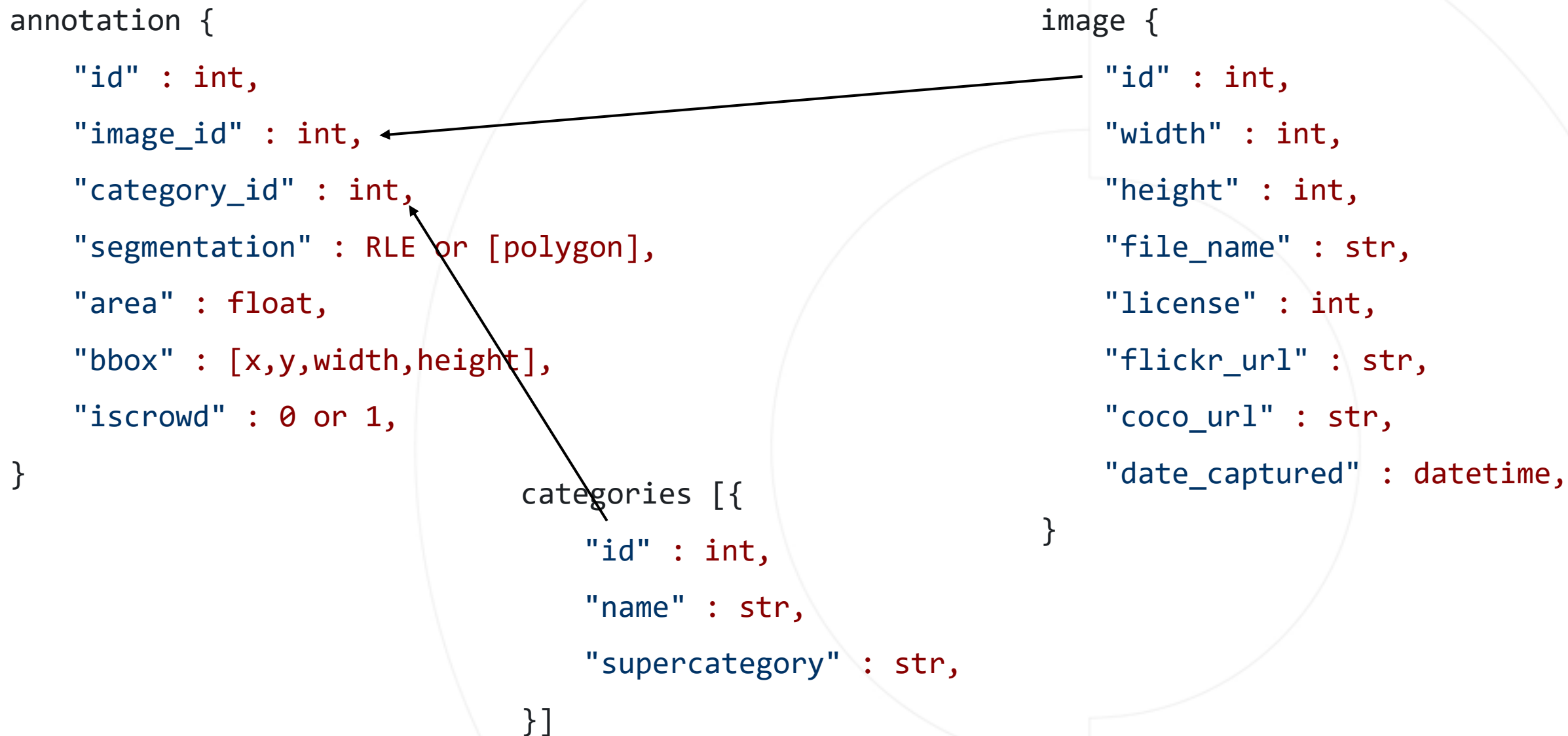
```
{  
  "info" : info,  
  "images" : [image],  
  "annotations" : [annotation],  
  "licenses" : [license],  
  "categories" : [categories],  
}
```

所有标注信息存储在一个列表中, 每个标注对应
图像上一个物体标注, 包含以下字段

```
annotation {  
  "id" : int,  
  "image_id" : int,  
  "category_id" : int,  
  "segmentation" : RLE or [polygon],  
  "area" : float,  
  "bbox" : [x,y,width,height],  
  "iscrowd" : 0 or 1,  
}
```



```
annotation {  
    "id" : int,  
    "image_id" : int,  
    "category_id" : int,  
    "segmentation" : RLE or [polygon],  
    "area" : float,  
    "bbox" : [x,y,width,height],  
    "iscrowd" : 0 or 1,  
}  
  
categories [{  
    "id" : int,  
    "name" : str,  
    "supercategory" : str,  
}]  
  
image {  
    "id" : int,  
    "width" : int,  
    "height" : int,  
    "file_name" : str,  
    "license" : int,  
    "flickr_url" : str,  
    "coco_url" : str,  
    "date_captured" : datetime,  
}
```



COCO 数据集的根目录

配置 batch_size

配置 Dataloader 进程数

针对训练、验证、测试子集，
分别配置：

- 数据类型 = COCO 数据集
- 标注文件的路径
- 图像目录的路径
- 数据处理流水线

```
dataset_type = 'CocoDataset'
data_root = 'data/coco/'
data = dict(
    samples_per_gpu=2,
    workers_per_gpu=2,
    train=dict(
        type=dataset_type,
        ann_file=data_root + 'annotations/instances_train2017.json',
        img_prefix=data_root + 'train2017/',
        pipeline=train_pipeline),
    val=dict(
        type=dataset_type,
        ann_file=data_root + 'annotations/instances_val2017.json',
        img_prefix=data_root + 'val2017/',
        pipeline=test_pipeline),
    test=dict(
        type=dataset_type,
        ann_file=data_root + 'annotations/instances_val2017.json',
        img_prefix=data_root + 'val2017/',
        pipeline=test_pipeline))
```


一个列表包含所有图像，
每个图像包含所有物体标注

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

自定义数据集的根目录

自定义数据集的类别名

配置 batch_size

配置 Dataloader 进程数

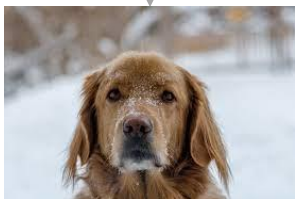
针对训练、验证、测试子集，
分别配置：

- 数据类型 = 自定义数据集
- 标注文件的路径（自定义格式中包含ndarray，需要用pkl格式）
- 图像目录的路径
- 数据处理流水线

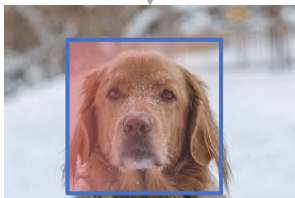
```
dataset_type = 'CustomDataset'
data_root = 'dataset_path/'
classes= ('classname1', 'classname2', ...)
data = dict(
    samples_per_gpu=2,
    workers_per_gpu=2,
    train=dict(
        type=dataset_type,
        ann_file='train/custom_anno.pkl',
        img_prefix='train',
        pipeline=train_pipeline),
    val=dict(
        type=dataset_type,
        ann_file='val/custom_anno.pkl',
        img_prefix='val',
        pipeline=test_pipeline),
    test=dict(
        type=dataset_type,
        ann_file='val/custom_anno.pkl',
        img_prefix='val',
        pipeline=test_pipeline))
```

~/dog/img1.jpg

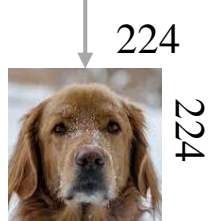
读取图像



随机裁剪



缩放
随机翻转



数据格式
转换

Tensor

像素归一化的均值
和标准差

定义数据加载流水线

从文件中读取图像

随机裁剪与缩放

随机水平翻转

像素值归一化

将数据转换为
PyTorch Tensor

```
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='RandomResizedCrop', size=224),  
    dict(type='RandomFlip', flip_prob=0.5,  
    direction='horizontal'),  
    dict(type='Normalize', **img_norm_cfg),  
    dict(type='ImageToTensor', keys=['img']),  
    dict(type='ToTensor', keys=['gt_label']),  
    dict(type='Collect', keys=['img', 'gt_label'])  
]
```

```
{'img_info', 'ann_info', ...} ← Dataset.load_annotation()
```



```
{'img'=array(h,w,3), ...}
```

```
{'gt_bboxes'=array(n,4),
```

```
'gt_labels'=array(n,...)}
```

```
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='DefaultFormatBundle'),
```

```
dict(type='Collect', keys=['img', 'gt_bboxes', 'gt_labels']),
```

```
]
```

在['img']上应用数据增强, 同时在
['gt_bboxes']上做对应操作

将数据转变为对应类型的 torch.Tensor

保留'img', 'gt_bboxes',
'gt_labels' 三个 key



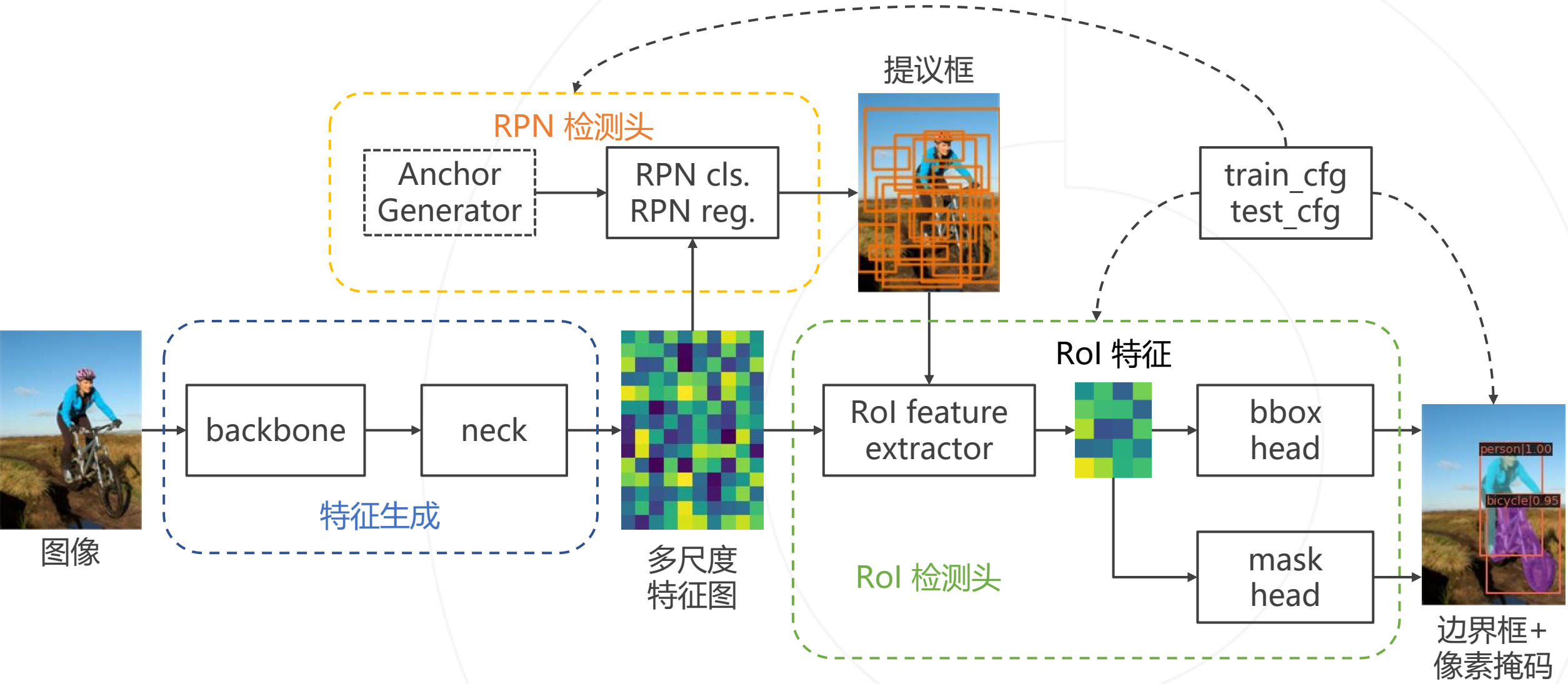
检测模型

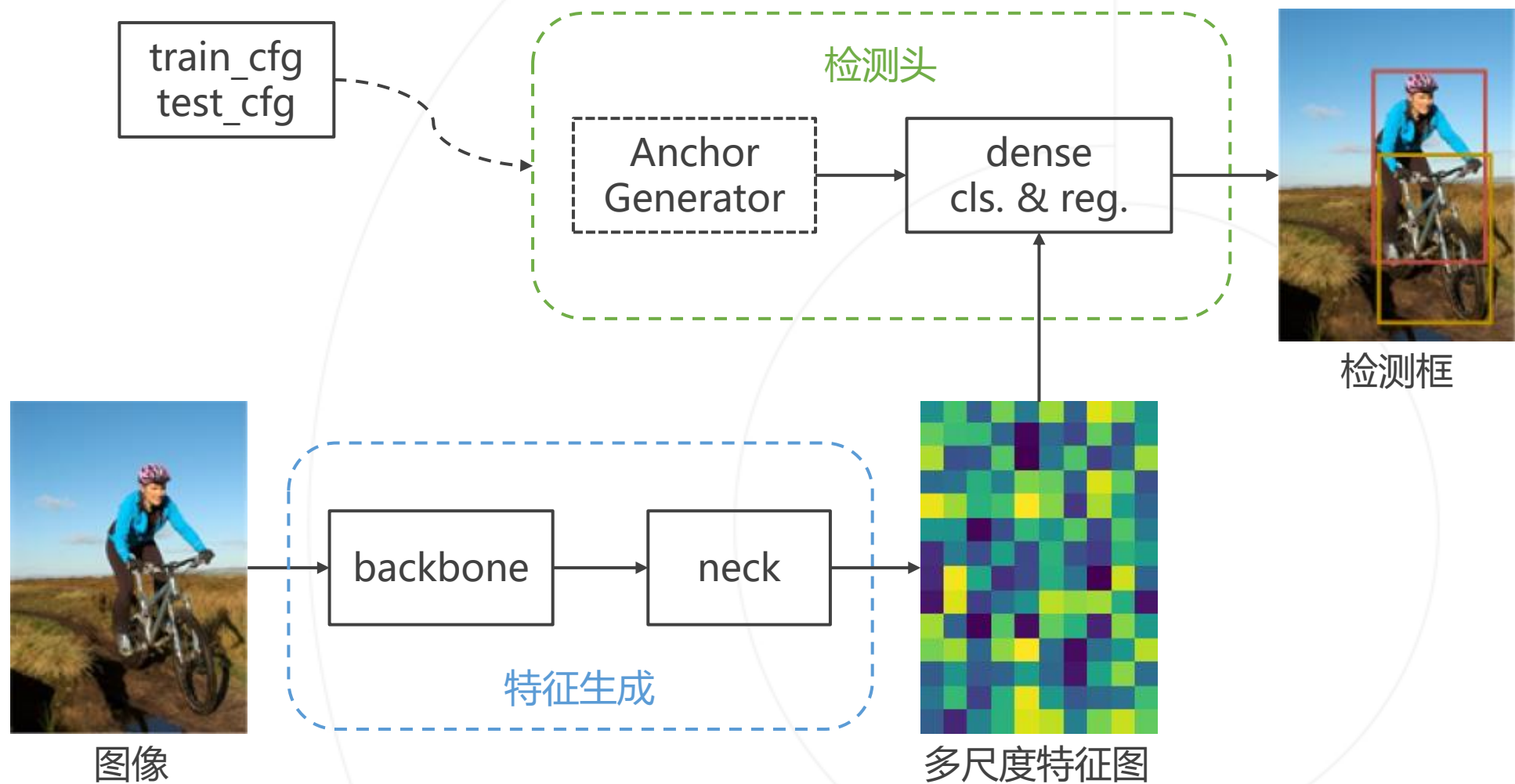
通常使用SGD算法配合不同的学习率策略：

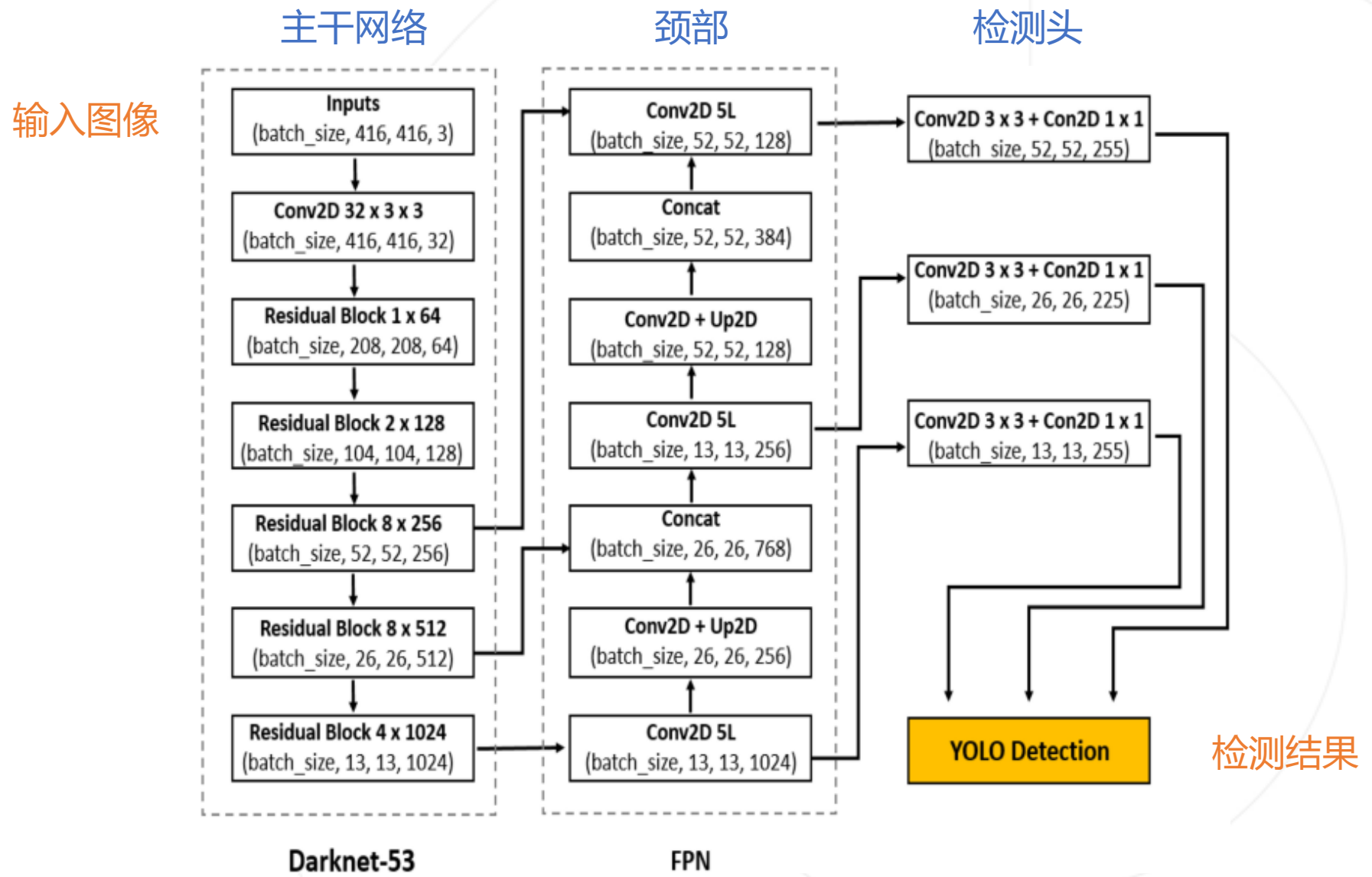
学习率策略	第一次降低	第二次降低	总轮数
1x	8 轮	11 轮	12 轮
2x	16 轮	22 轮	24 轮
20e	16 轮	19 轮	20 轮

所有策略均包含一个学习率升温过程。

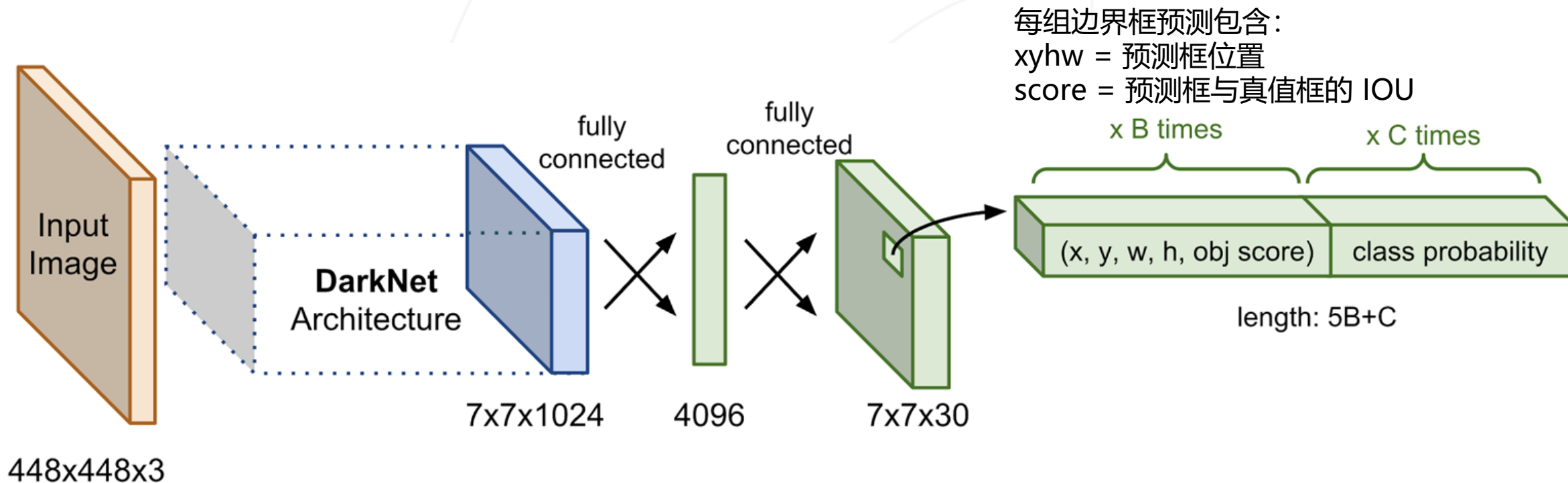
```
optimizer = dict(type='SGD', lr=0.02, momentum=0.9,
weight_decay=0.0001)
optimizer_config = dict(grad_clip=None)
# learning policy
lr_config = dict(
    policy='step',
    warmup='linear',
    warmup_iters=500,
    warmup_ratio=0.001,
    step=[8, 11])
runner = dict(type='EpochBasedRunner', max_epochs=12)
```



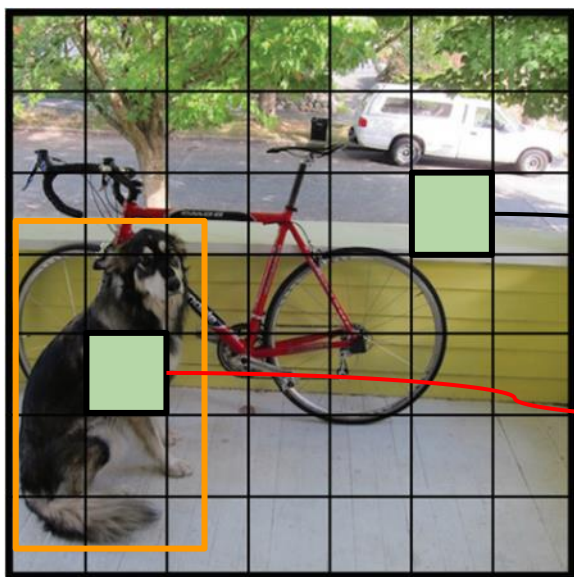




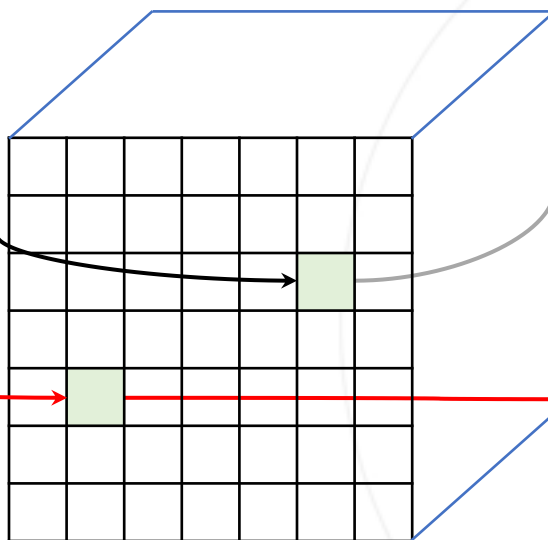
- 最早的单阶段算法之一
- DarkNet 结构的主干网络产生特征图
- 全连接层产生全部空间位置的预测结果，每个位置包含 C 维分类概率和 B 组边界框预测



- 将原图切分成 $S \times S$ 大小的格子，对应预测图上 $S \times S$ 个像素位置。
- 如果原图上某个物体的中心落于某个格子内，则对应位置的预测值应给出该物体的类别和边界框的位置（基于格子边界的偏移量）。
- 其余位置应预测为背景类别，不关心边界框预测结果。



将图像划分为 7×7 的格子



7×7 分辨率的预测图
每个位置 $5B+C$ 个通道
= B 个框+ C 个类别的预测

类别预测=背景
边界框不计算 loss

类别预测 = 狗
边界框预测:
 X, Y = 相对于格子边界的偏移量
 W, H = 窗大小 / 图像大小
分数 = 预测框与真值框的交并比

多任务学习：回归、分类共同计入损失函数，通过 λ 控制权重

$$\lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^B \mathbb{1}_{ij}^{\text{obj}} \left[(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2 \right] \\ + \lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^B \mathbb{1}_{ij}^{\text{obj}} \left[\left(\sqrt{w_i} - \sqrt{\hat{w}_i} \right)^2 + \left(\sqrt{h_i} - \sqrt{\hat{h}_i} \right)^2 \right]$$

边界框需要产生**物体**预测时，
计算边界框坐标的回归损失

$$+ \sum_{i=0}^{S^2} \sum_{j=0}^B \mathbb{1}_{ij}^{\text{obj}} (C_i - \hat{C}_i)^2 \\ + \lambda_{\text{noobj}} \sum_{i=0}^{S^2} \sum_{j=0}^B \mathbb{1}_{ij}^{\text{noobj}} (C_i - \hat{C}_i)^2$$

边界框需要产生**类别（物体/背景）**预测时，
计算边界框置信度的回归损失

$$+ \sum_{i=0}^{S^2} \mathbb{1}_i^{\text{obj}} \sum_{c \in \text{classes}} (p_i(c) - \hat{p}_i(c))^2$$

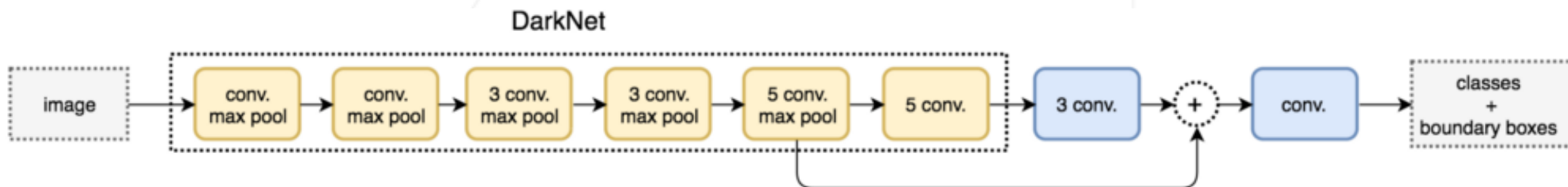
边界框需要产生**物体**预测时，
计算C个类别概率的回归损失

- 快！在Pascal VOC 数据集上，使用自己设计的 DarkNet 结构可以达到实时速度，使用相同的 VGG 可以达到 3 倍于 Faster RCNN 的速度
- 不依赖锚框，直接回归边界框

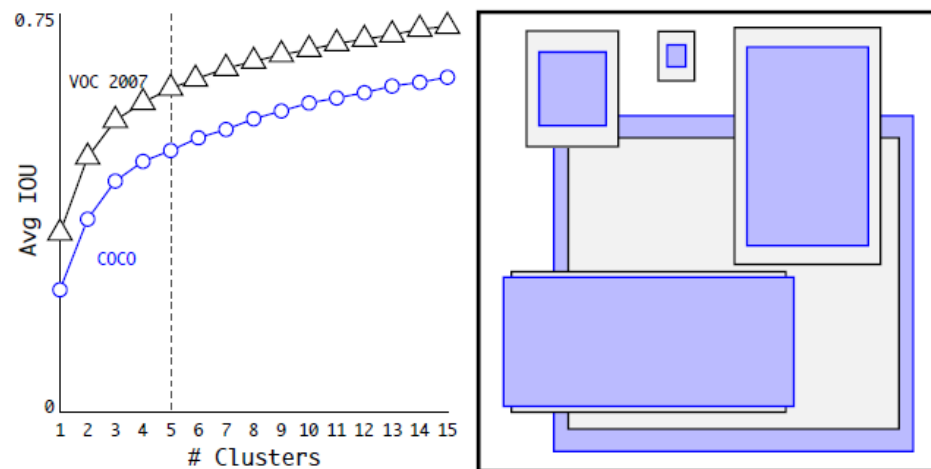
检测算法	主干网络	训练集	检测精度 (mAP)	检测速度 (FPS)
Fast YOLO	9层 DarkNet	VOC 2007 + 2012	52.7	155
YOLO	24层 DarkNet	VOC 2007 + 2012	63.4	45
Faster R-CNN VGG16	VGG 16	VOC 2007 + 2012	73.2	7
YOLO VGG 16	VGG 16	VOC 2007 + 2012	66.4	21

- 由于每个格子只能预测 1 个物体，因此对重叠物体、尤其是大量重叠的小物体容易产生漏检
- 直接回归边界框有难度，回归误差较大

- 新的 DarkNet-19 结构的主干网络，加入 BN



- 加入锚框，并使用聚类方法设定锚框尺寸





上海交大-OpenMMLab课程答疑



该二维码7天内(4月5日前)有效, 重新进入将更新

谢谢大家

- 主干网络加入残差结构 → DarkNet 53
- 加入类 FPN 结构，基于多尺度特征图预测

