第3章　目标检测

一直以来，目标检测都是计算机视觉领域基本的且具有挑战性的问题，受到了研究学者的广泛关注，它与图像分类以及图像分割任务一起构成了计算机视觉领域的热点性研究问题。图像分类是针对整个图像进行类别的判断，更关注图像整体表达的含义。而目标检测则是识别图像中可能存在的预定义目标实例。如图 3-0-1 所示，在图像分类中，只需要对整个图像给出预测结果，即识别出“猫”。而对于目标检测，则需要识别出图像中存在的预定义的目标实例（猫、狗、鸭子等），并给出每个实例的位置、大小和类别，即通过表达不同类别含义的矩形框包裹图像中的不同实例，通常情况下对于每个目标实例的矩形框使用中心点坐标和长宽(x,y,w,h)或左上角、右下角的坐标(x_1,y_1,x_2,y_2)表示。

图 3-0-1　图像分类和目标检测

近几年，基于深度学习的目标检测取得了突飞猛进的发展，如图 3-0-2 所示，可以在多种不同的场景检测多个目标实例，同时这些目标的类别也变得更加丰富，从最开始的人脸、行人，发展到了可以识别现如今生活中常见的各类物品。伴随着目标检测的发展，其在人脸检测、智能计数、视觉搜索引擎以及航拍图像分析等应用领域中发挥着不可替代的作用。

我们把深度学习广泛地应用于目标检测之前的方法称为传统的目标检测方法。传统的目标检测方法（目标提取方法）一般情况下分为三个阶段：第一阶段，在给定的图像上选择若干候选区域；第二阶段，通过各种方法对候选区域进行所需特征的提取；第三阶段，使用经过预处理的分类器或者回归器对特征进行分类。其中，区域选择是通过使用不同尺寸的窗口在图像中进行滑动操作选取图像的某一部分作为候选区域；特征提取是提取每个候选区域的人工设计的视觉特征，但是由于人工特征是根据目标的形状、颜色、纹理、边缘等因素设计的，具有很强的针对性。因此，为了检测不同的目标会设计和使用不同的特征，比如人脸检测任务中使用的 Haar 特征，行人检测任务中常用 HOG 特征。特征提取器所提取特征的质量将直接影响分类器或者回归器的准确性，但是设计一个适用于多类目标且鲁棒性较

图 3-0-2　多种场景下的目标检测

好的特征是比较难的。综上，可以看出，传统目标提取方法具有两个缺点：一是区域选择策略；二是人工设计特征的局限性。

传统的依靠手工提取特征完成各类任务的方式一度盛行，一直到 2012 年，Krizhevsky 等人提出了一种名为 AlexNet 的深度卷积神经网络（DCNN），它在大规模视觉识别挑战赛（ILSRVC）中突破了图像分类准确性的纪录。从那时起，计算机视觉领域的研究重点开始转移到深度学习的方法。R-CNN 方法可以说是卷积神经网络在目标检测领域的里程碑式的方法，它开启了目标检测领域的新篇章，由此目标检测领域也取得了显著性的突破。基于深度学习的目标检测方法根据其原理有几种不同的划分方式。其中比较经典的划分方式是根据其检测的流程分为一阶段目标检测算法和两阶段目标检测算法。

（1）两阶段目标检测算法，其将目标提取过程主要分为两个阶段：第一个阶段是产生候选区域（region proposals），得到可能存在目标的区域；第二个阶段是修正候选区域中的目标位置并判断目标类别。这类算法的典型代表是基于区域（region-based）的 R-CNN 系列算法，包含 R-CNN、Fast R-CNN 和 Faster R-CNN 等。

（2）一阶段目标检测算法，其移除了产生候选区域的阶段，直接通过图像预测目标的位置和类别，这类算法的典型代表有：SSD、YOLO 等。

目前主流的目标检测算法还可以依据其是否需要先验候选框，划分为基于 Anchor 和不基于 Anchor 的目标检测方法。Anchor 的本质是先验框，在设计了不同尺度和比例的先验框后，网络会学习如何区分和修正这些先验框：是否包含 object、包含什么类别的 object，以及修正先验框的位置。但是，由于 Anchor 要先验地人为设定，设定的数目和尺寸都将会直接影响检测算法的效果。基于这种原因，很多人做了改进，提出了 Anchor Free 的方法，例如，CornerNet、CenterNet、ExtremeNet 等不依赖 Anchor 来实现目标检测的方法。除此之外，近几年基于 Transformer 的目标检测方法大放异彩，取得了不凡的成绩。

3.1　实践一：基于 Faster RCNN 模型的瓷砖瑕疵检测(两阶段目标检测)

在本节,我们将使用 PaddleDetection 来实现 Faster RCNN 网络进行瓷砖表面瑕疵检测。

基于 Faster RCNN 模型的瓷砖瑕疵检测

Faster RCNN 是两阶段目标检测方法的代表之作。Faster RCNN 丢弃了离线的候选框的提取过程,将目标检测变成一个端到端的过程,从而大大节省了推理时间,并且使得检测任务变得更加容易。如图 3-1-1 所示,Faster RCNN 的方法可分为 4 个步骤。

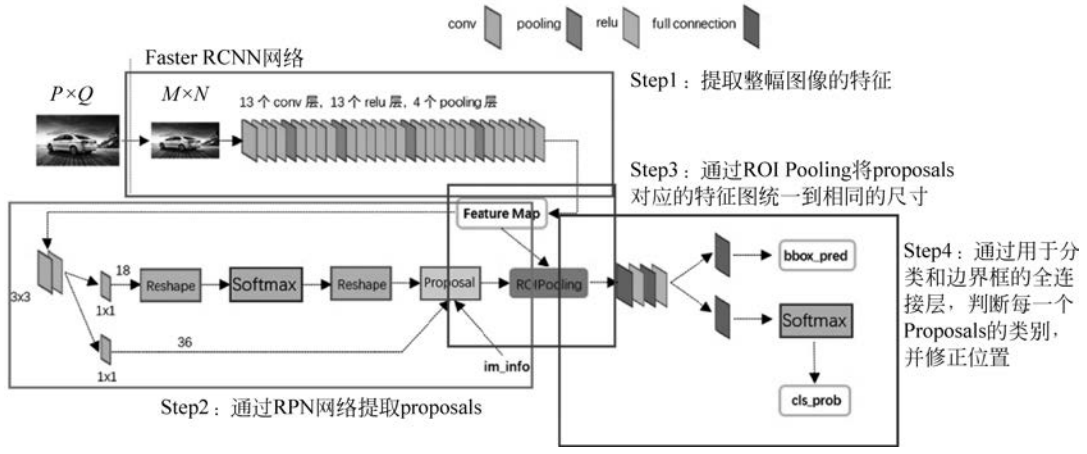


图 3-1-1　Faster RCNN 网络结构

首先,将整张图片作为输入送到卷积神经网络中进行特征提取,得到特征图;其次,将卷积特征图输入到候选框生成网络(Region Proposal Network,RPN)中进行候选框的预测(可能存在目标的区域),这个预测包含两部分,一部分是对预设的默认框进行一个背景和前景的二分类判别,另一部分是对预设的默认框进行一次中心位置偏移量和宽高的回归,从而得到一组稀疏的候选框;再次,将得到的候选框所对应在特征图上的特征区域通过一个 ROI 池化调整到固定尺寸;最后,通过两个全连接层对其进行 $n+1$ 类(n 个目标类+背景类)的分类和中心位置偏移量以及长宽的二次回归。

步骤 1:数据集介绍及预处理

瓷砖经过复杂的工艺生产出来后,需要经过质量检测和包装等步骤才能投放市场。人工智能技术的发展,赋能了越来越多的传统制造业。在质检领域,通过智能化手段代替人工检测,可以大大节约时间和人力成本,并且检测质量也能得到提升。

本次瓷砖瑕疵检测的数据集共包含 5388 张图像。如图 3-1-2 所所示,数据集包括砖渣、落脏、滴墨等 6 个类别,本次实践的任务也就是检测出图像中存在的瑕疵位置并区分瑕疵的种类。

数据集分为图像和标注两个部分。如图 3-1-3 所示,train_imgs 目录下存储着用于训练和验证的图像,train_annos.json 下则存储着所有图像对应的标注。

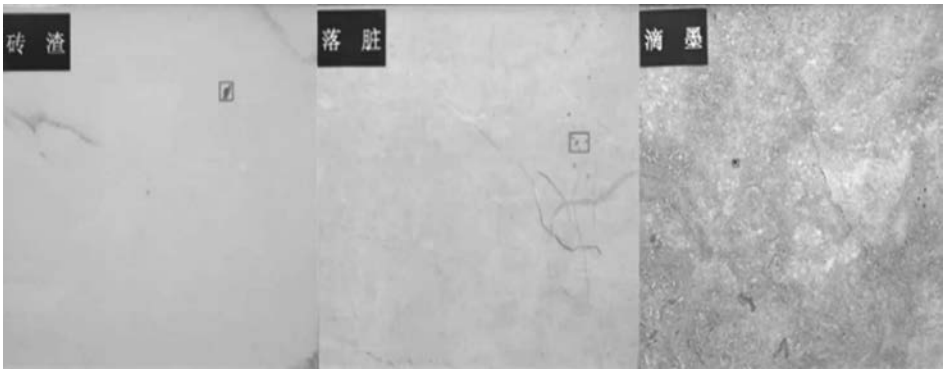


图 3-1-2　瓷砖瑕疵示例

图 3-1-3　数据集格式列表

目标检测有两类经典的标注格式，分别是以 PASCAL VOC 数据集为代表的 XML 格式数据集和以 COCO 数据集为代表的 JSON 格式的数据集。本次实践中数据集的格式不同于上述两种数据格式，如图 3-1-4 所示，每个框内表示一个目标实例，其中 name、image_height 和 image_width 则分别表示目标实例所在的图像文件名以及图像的长和宽；category 表示的是目标实例所有对应的类别（1～6 分别表示不同的瓷砖瑕疵类别）；bbox 表示的是包裹目标实例的矩形框，其中 0、1 表示矩形框的左上角点的坐标，2、3 表示矩形框的右下角点的坐标。

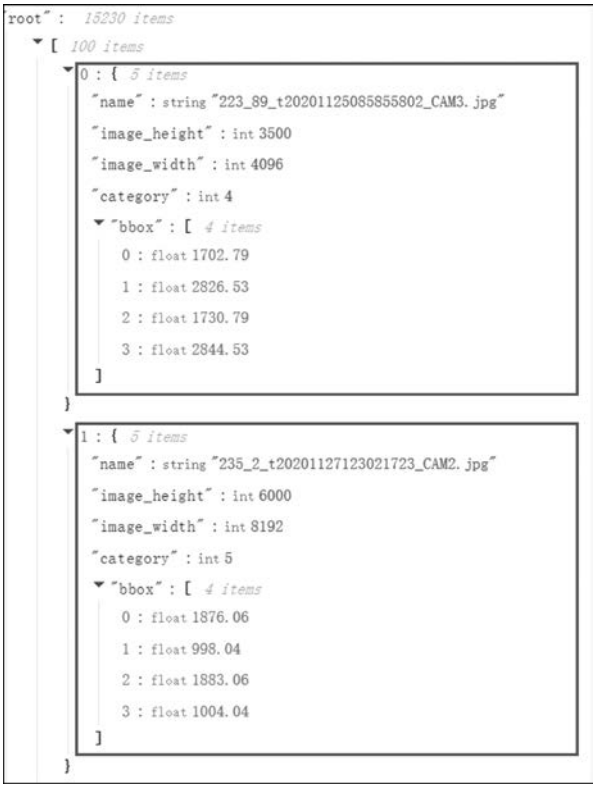


图 3-1-4　标注文件示例

我们在使用 PaddleDetection 进行目标检测之前，需要将标注文件转换为 COCO 的标注格，并按照 9∶1 的比例划分训练集和测试集。在这里我们通过 Fabric2COCO 类来实现数据标注格式的转化和数据集的划分。

```
#训练集,划分 90 % 作为验证集
fabric2coco = Fabric2COCO()
train_instance = fabric2coco.to_coco(anno_dir,img_dir)
fabric2coco.save_coco_json(train_instance, "/home/aistudio/work/PaddleDetection - release -
2.2/dataset/coco/annotations/" + 'instances_{}.json'.
format("train"))
#验证集,划分 10 % 作为验证集
fabric2coco_val = Fabric2COCO(is_mode = "val")
val_instance = fabric2coco_val.to_coco(anno_dir,img_dir)
fabric2coco_val.save_coco_json(train_instance, "/home/aistudio/work/PaddleDetection -
release - 2.2/dataset/coco/annotations/" + 'instances_{}.json'
.format("val"))
```

进行转换后，会得到如图 3-1-5 所示的目录。其中 train 和 val 目录下存储的是分别用于训练和验证的图像，annotations 目录下存储的则是 instances_train.json 和 instances_val.json 两个文件分别对应转换后的训练集和验证集合的标注文件。

图 3-1-5　生成目录

以 instances_train.json 为例，转换后得到的数据标注如图 3-1-6 所示，左侧框内表示训练集中用于训练的图像名称、ID 和图像对应的长和宽。右侧框内表示的则是单个目标实

图 3-1-6　生成标注示例

例，其中 image_id 表示的是目标实例所存在的图像（与右侧图像 ID 相对应），categoory_id 表示的则是目标实例所对应的类别（1～6 分别表示不同的瓷砖瑕疵类别），bbox 中存储的由原来的矩形框角点坐标转换成中心点坐标和矩形框的长宽，area 表示的是矩形框的面积。

步骤 2：PaddleDection 及环境安装

PaddleDetection 为基于飞桨 PaddlePaddle 的端到端目标检测套件，内置 30 多个模型算法及 250 多个预训练模型，覆盖目标检测、实例分割、跟踪、关键点检测等方向，其中包括服务器端和移动端高精度、轻量级产业级 SOTA 模型、冠军方案和学术前沿算法，并提供配置化的网络模块组件、十余种数据增强策略和损失函数等高阶优化支持和多种部署方案，在打通数据处理、模型开发、训练、压缩、部署全流程的基础上，提供丰富的案例及教程，加速算法产业落地应用。

经过长时间的产业实践打磨，PaddleDetection 已拥有顺畅、卓越的使用体验，被工业质检、遥感图像检测、无人巡检、新零售、互联网、科研等十多个行业广泛使用，如图 3-1-7 所示。

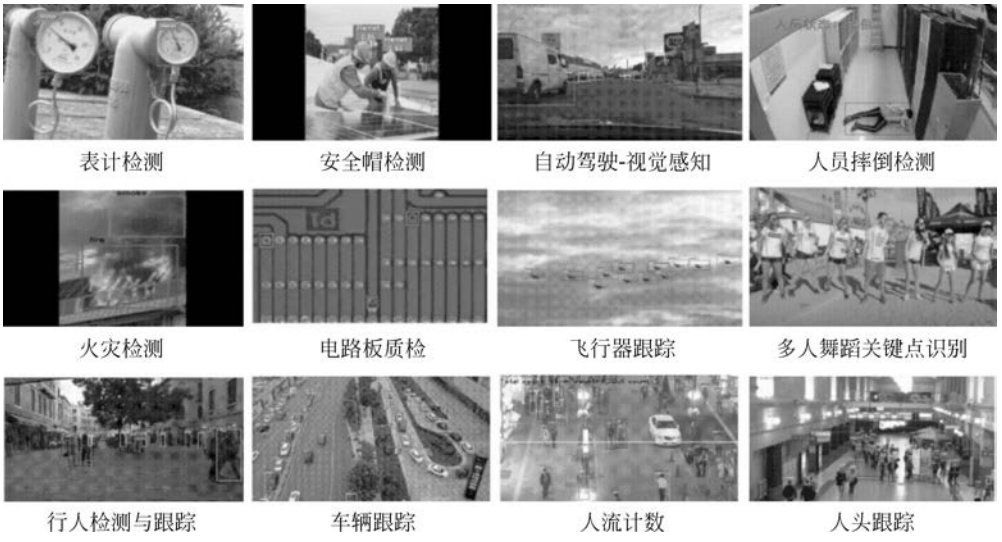


图 3-1-7　PaddleDetection 应用示例

PaddleDetection 具有以下特点。

模型丰富：包含目标检测、实例分割、人脸检测、关键点检测、多目标跟踪等 250 多个预训练模型，涵盖多种全球竞赛冠军方案。

使用简洁：模块化设计，解耦各个网络组件，开发者轻松搭建、试用各种检测模型及优化策略，快速得到高性能、定制化的算法。

端到端打通：从数据增强、组网、训练、压缩、部署端到端打通，并完备支持云端/边缘端多架构、多设备部署。

高性能：基于飞桨的高性能内核，模型训练速度及显存占用优势明显。支持 FP16 训练，支持多机训练。

在使用 PaddleDetection 时我们可以根据任务的需要，在图 3-1-8 中选择不同的模型、特征提取网络、组件和数据增强方式。比如，我们在进行目标检测可以选择两阶段的 Faster

Architectures	Backbones	Components	Data Augmentation

• **Object Detection**
  ○ Faster RCNN
  ○ FPN
  ○ Cascade-RCNN
  ○ Libra RCNN
  ○ Hybrid Task RCNN
  ○ PSS-Det
  ○ RetinaNet
  ○ YOLOv3
  ○ YOLOv4
  ○ PP-YOLOv1/v2
  ○ PP-YOLO-Tiny
  ○ PP-YOLOE
  ○ YOLOX
  ○ SSD
  ○ CornerNet-Squeeze
  ○ FCOS
  ○ TTFNet
  ○ PP-PicoDet
  ○ DETR
  ○ Deformable DETR
  ○ Swin Transformer
  ○ Sparse RCNN
• **Instance Segmentation**
  ○ Mask RCNN
  ○ SOLOv2
• **Face Detection**
  ○ FaceBoxes
  ○ BlazeFace
  ○ BlazeFace-NAS
• **Multi-Object-Tracking**
  ○ JDE
  ○ FairMOT
  ○ DeepSORT
• **KeyPoint-Detection**
  ○ HRNet
  ○ HigherHRNet
	• ResNet(&vd)
	• ResNeXt(&vd)
	• SENet
	• Res2Net
	• HRNet
	• Hourglass
	• CBNet
	• GCNet
	• DarkNet
	• CSPDarkNet
	• VGG
	• MobileNetv1/v3
	• GhostNet
	• Efficientnet
	• BlazeNet
		• **Common**
		  ○ Sync-BN
		  ○ Group Norm
		  ○ DCNv2
		  ○ Non-local
		• **KeyPoint**
		  ○ DarkPose
		• **FPN**
		  ○ BiFPN
		  ○ BFP
		  ○ HRFPN
		  ○ ACFPN
		• **Loss**
		  ○ Smooth-L1
		  ○ GIoU/DIoU/CIoU
		  ○ IoUAware
		• **Post-processing**
		  ○ SoftNMS
		  ○ MatrixNMS
		• **Speed**
		  ○ FP16 training
		  ○ Multi-machine training
			• Resize
			• Lighting
			• Flipping
			• Expand
			• Crop
			• Color Distort
			• Random Erasing
			• Mixup
			• AugmentHSV
			• Mosaic
			• Cutmix
			• Grid Mask
			• Auto Augment
			• Random Perspective

图 3-1-8　PaddleDetection 组件

RCNN、一阶段的 YOLO 系列以及基于 Transformer 的目标检测模型 DETR、Swin Transformer 等，同时也可以根据我们对精度和速度的要求选择不同的特征提取网络和组件（对小目标要求较高时使用 HRNet，对速度要求较高时使用 MobileNet 等）。除此之外，还可以根据实际的需求选择不同的数据增强方法等。

在使用 PaddleDetection 进行目标检测之前，我们首先要下载 PaddleDetection 的源码（可以通过 git 下载，也可以解压下载好的压缩包），然后安装 PaddleDetection 所需要的依赖并编译安装 paddledet。

```
#下载
git clone https://github.com/PaddlePaddle/PaddleDetection.git
#解压
!unzip -o /home/aistudio/data/data113827/PaddleDetection-release-2.2_tile.zip -d /home/aistudio/work/
!pip install -r requirements.txt
!python setup.py install
```

步骤 3：模型训练及验证

在处理好数据和部署好环境后，我们就可以通过 train.py 开始训练网络。在使用 train.py() 函数训练网络的时候，我们还需要通过加载配置文件来配置我们的训练过程。如图 3-1-9 所示，在配置文件中可以设置迭代的总轮数、预训练的权重、检测类别、数据集路径、优化器以及 Faster RCNN 网络中的各种参数配置（特征提取网络、网络深度、FPN 网络参数、RPN 网络参数设置等），同时还可以通过--eval 参数表示在训练过程中在验证集上验证模型。

```
!python tools/train.py \
    -c /home/aistudio/work/faster_rcnn_r50_fpn_2x.yml -eval
```

```
architecture: FasterRCNN
max_iters: 30000
use_gpu: true
snapshot_iter: 2000
log_iter: 10
save_dir: output
pretrain_weights: https://paddledet.bj.bcebos.com/model
metric: COCO
# If set `true`, for traffic output json
#save_prediction_only: true
save_prediction_only: false
weights: output/faster_rcnn_r50_fpn_2x/model_final
num_classes: 7
snapshot_epoch: 1
norm_type: sync_bn

epoch: 20

worker_num: 0
TrainDataset:
  !COCODataSet
  image_dir: train
  anno_path: annotations/instances_train.json
  dataset_dir: /home/aistudio/work/PaddleDetection-re
  data_fields: ['image', 'gt_bbox', 'gt_class', 'is_c

FasterRCNN:
  backbone: ResNet
  neck: FPN
  rpn_head: RPNHead
  bbox_head: BBoxHead
  # post process
  bbox_post_process: BBoxPostProcess

ResNet:
  depth: 50
  norm_type: bn
  freeze_at: 2
  return_idx: [0,1,2,3]
  num_stages: 4

FPN:
  out_channel: 256
```

```
# RPNHead
RPNHead:
  # anchor generator
  anchor_generator:
    aspect_ratios: [0.5, 1.0, 2.0]
    anchor_sizes: [[32], [64], [128], [256], [512]]
    strides: [4, 8, 16, 32, 64]
  # rpn_target_assign
  rpn_target_assign:
    batch_size_per_im: 256
    fg_fraction: 0.5
    negative_overlap: 0.3
    positive_overlap: 0.7
    use_random: True
  # 训练时生成proposal的参数
  train_proposal:
    min_size: 0.0
    nms_thresh: 0.7
    pre_nms_top_n: 2000
    post_nms_top_n: 1000
    topk_after_collect: True
  # 评估时生成proposal的参数
  test_proposal:
    min_size: 0.0
    nms_thresh: 0.7
    pre_nms_top_n: 1000
    post_nms_top_n: 1000

# BBoxHead
BBoxHead:
  # TwoFCHead as BBoxHead
  head: TwoFCHead
  # roi align
  roi_extractor:
    resolution: 7
    sampling_ratio: 0
    aligned: True
  # bbox_assigner
  bbox_assigner: BBoxAssigner
```

```
# BBoxAssigner
BBoxAssigner:
  # batch_size_per_im
  batch_size_per_im: 512
  # 背景阈值
  bg_thresh: 0.5
  # 前景阈值
  fg_thresh: 0.5
  # 前景比例
  fg_fraction: 0.25
  # 是否随机采样
  use_random: True

# TwoFCHead
TwoFCHead:
  # TwoFCHead特征维度
  out_channel: 1024


# BBoxPostProcess
BBoxPostProcess:
  # 解码
  decode: RCNNBox
  # nms
  nms:
    # 使用MultiClassNMS
    name: MultiClassNMS
    keep_top_k: 100
    score_threshold: 0.05
    nms_threshold: 0.5
```

图 3-1-9　配置文件

训练开始后会随着训练的进行，输出迭代的轮数、batch 的批次、学习率、Faster RCNN 网络中 RPN 网络和预测网络的分类、回归损失，以及总损失等，如图 3-1-10 所示。

```
[06/06 22:16:31] ppdet.engine INFO: Epoch: [0] [  10/2389] learning_rate: 0.000347 loss_rpn_cls: 0.689849 loss_rpn_reg: 0.113589 loss_bbox_c
ls: 0.264905 loss_bbox_reg: 0.000366 loss: 1.077134 eta: 3 days, 1:12:20 batch_cost: 5.7971 data_cost: 4.3453 ips: 0.3450 images/s
[06/06 22:17:25] ppdet.engine INFO: Epoch: [0] [  20/2389] learning_rate: 0.000360 loss_rpn_cls: 0.584538 loss_rpn_reg: 0.095285 loss_bbox_c
ls: 0.084861 loss_bbox_reg: 0.000559 loss: 0.764139 eta: 3 days, 0:06:37 batch_cost: 5.3459 data_cost: 3.8599 ips: 0.3741 images/s
```

图 3-1-10　训练过程中的部分输出结果

训练完成后可以通过执行 eval.py 开启验证模型，与训练时相似，也需要给定模型的配置文件，除此之外还需要给定训练阶段得到权重文件。

```
!python tools/eval.py \
    -c /home/aistudio/work/faster_rcnn_r50_fpn_2x.yml -o weights=
    output/faster_rcnn_r50_fpn_2x/best_model.pdparams
```

也可以通过执行 infer.py 用训练好的模型进行预测。在这里，需要给定模型的配置文件、训练好的权重和用于预测的图像路径。

```
!python -u tools/infer.py \
  -c /home/aistudio/work/faster_rcnn_r50_fpn_2x.yml \
  --output_dir=infer_output/ \
  --save_txt=True \
     -o weights=
    output/faster_rcnn_r50_fpn_2x/best_model.pdparams \
    --infer_img=/home/aistudio/work/235_7_t20201127123214965_CAM2.jpg
```

将预测后的图片局部放大后可以看到图 3-1-11 的检测结果。

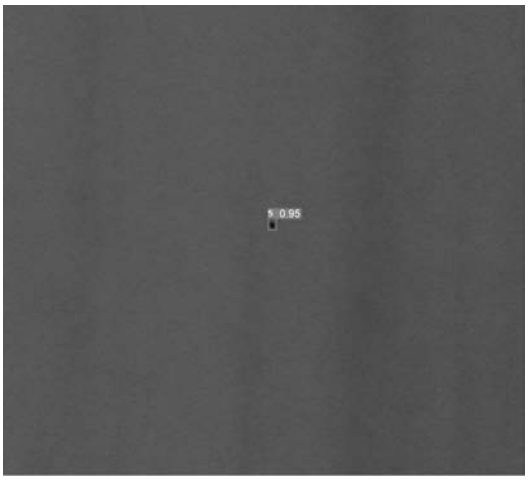


图 3-1-11　检测结果示例

3.2　实践二：基于 YOLOV3/PP-YOLO 模型的昆虫检测（一阶段目标检测）

本节将使用 YOLOV3 和 PP-YOLO 来实现昆虫识别。

R-CNN 系列算法需要先产生候选区域，再对候选区域进行分类和位置的预测，这类算法被称为两阶段目标检测算法。近几年，很多研究人员相继提出一系列一阶段的检测算法，直接从图像中预测目标，从而涉及候选区域提议的过程。

Joseph Redmon 等人在 2015 年提出 YOLO(You Only Look Once)算法，该算法通常也被称为 YOLOV1；2016 年，他们对算法进行改进，又提出 YOLOV2 版本；2018 年该算法发展出 YOLOV3 版本。YOLO3 采用了 Darknet-53 的网络结构(含有 53 个卷积层)，它借鉴了残差网络的做法，在一些层之间设置了跳跃链接，并在三个不同的尺度上进行预测。

PP-YOLO 是 PaddleDetecion 中基于 YOLOV3 精度速度优化的实战实践，通过几乎不增加预测计算量的优化方法尽可能地提高 YOLOV3 模型的精度，最终在 COCO test-dev2017 数据集上精度达到 45.9%，单卡 V100 预测速度为 72.9FPS。图 3-2-1 是 PP-YOLO 模型和当时 SOTA 的目标检测算法在 COCO test-dev 数据集的精度和 V100 上预测速度的对比图。

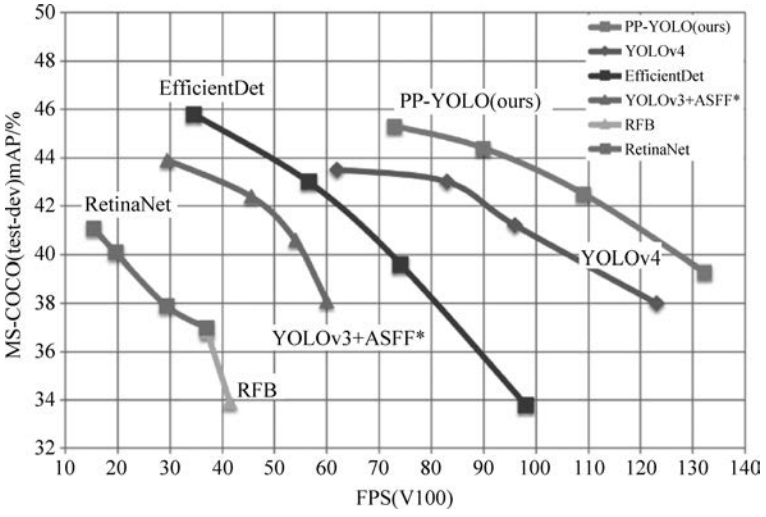


图 3-2-1　网络效果对比

基于 YOLOV3 模型的昆虫检测

3.2.1　基于 YOLOV3 模型的昆虫检测

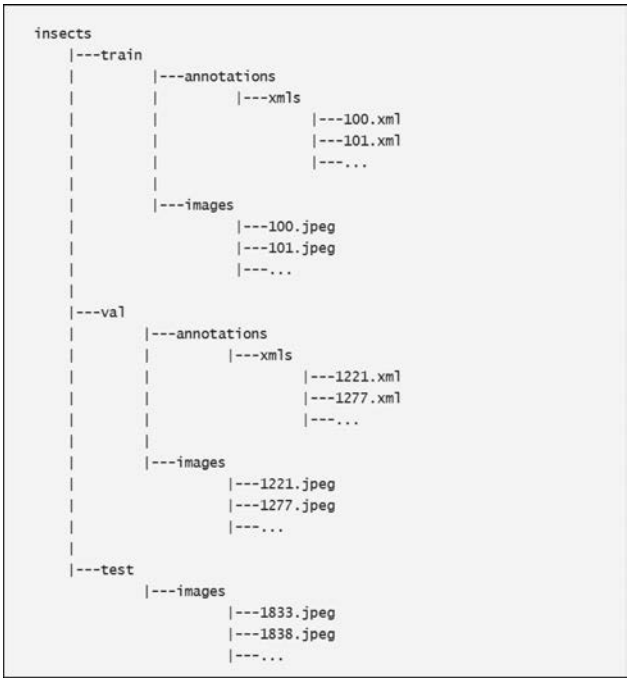
步骤 1：认识 AI 识虫数据集与数据下载

本次实践采用百度与北京林业大学合作开发的林业病虫害防治项目用到的 AI 识虫数据集，如图 3-2-2 所示，图片中有不同种类的昆虫，本次实践的目标就是检测出图像中昆虫的位置并区分它们的类别。数据集可以在 AIstudio 中下载：https://aistudio.baidu.com/aistudio/datasetdetail/19638。

图 3-2-2　数据集图像示例

该数据集提供了 2183 张图像，其中训练集 1693 张，验证集 245 张，测试集 245 张，共包含 Boerner、Leconte、Linnaeus、acuminatus、armandi、coleoptera 等多种昆虫。数据集格式如图 3-2-3 所示，分为 train、val 和 test 三个文件夹，每个文件夹下图像和标注文件分别存储在 annotations 和 images 下。

昆虫数据集采用了与 PASCAL VOC 数据集相同



```
insects
    |---train
    |         |---annotations
    |         |         |---xmls
    |         |                   |---100.xml
    |         |                   |---101.xml
    |         |                   |---...
    |         |
    |         |---images
    |                   |---100.jpeg
    |                   |---101.jpeg
    |                   |---...
    |
    |---val
    |         |---annotations
    |         |         |---xmls
    |         |                   |---1221.xml
    |         |                   |---1277.xml
    |         |                   |---...
    |         |
    |         |---images
    |                   |---1221.jpeg
    |                   |---1277.jpeg
    |                   |---...
    |
    |---test
              |---images
                        |---1833.jpeg
                        |---1838.jpeg
                        |---...
```

图 3-2-3　数据集结构

的 XML 标注格式，如图 3-2-4 所示，filename 标签对下记录的是图像名称；size 标签对下记录的是图像的宽、高和图像的通道数；每个 object 标签对下记录的是图像中每个目标实例

```
<annotation>
    <folder>石膏宇</folder>
    <filename>581.jpeg</filename>
    <path>/home/sxy/已拍摄图片/石膏宇/石膏宇/石膏宇/581.jpeg</path>
    <source>
        <database>Unknown</database>
    </source>
    <size>
        <width>1274</width>
        <height>1274</height>
        <depth>3</depth>
    </size>
    <segmented>0</segmented>
    <object>
        <name>Boerner</name>
        <pose>Unspecified</pose>
        <truncated>0</truncated>
        <difficult>0</difficult>
        <bndbox>
            <xmin>812</xmin>
            <ymin>566</ymin>
            <xmax>866</xmax>
            <ymax>694</ymax>
        </bndbox>
    </object>
    <object>
        <name>Leconte</name>
        <pose>Unspecified</pose>
        <truncated>0</truncated>
        <difficult>0</difficult>
        <bndbox>
            <xmin>632</xmin>
            <ymin>415</ymin>
            <xmax>753</xmax>
            <ymax>523</ymax>
        </bndbox>
    </object>
</annotation>
```

图 3-2-4　标注文件

的信息。其中，name 标签对表示目标实例的类别，bndbox 标签对则是记录的目标实例矩形框的左上角和右下角坐标。

步骤 2：数据加载

（1）数据读取。

在本次实践中，我们需要通过编写代码从 xml 文件中提取标注信息。首先，通过 get_annotations 读取 xml 中的标注信息，并返回一个图像中所有目标实例的类别和位置（x，y，w，h），在这里我们需要用 ElementTree 来解析 xml 格式的文件，获取图像的名称、宽、高以及通道数。

```
def get_annotations(cname2cid, datadir):
  filenames = os.listdir(os.path.join(datadir, 'annotations', 'xmls'))
  records = []
  ct = 0
  for fname in filenames:
    fid = fname.split('.')[0]
    fpath = os.path.join(datadir, 'annotations', 'xmls', fname)
    img_file = os.path.join(datadir, 'images', fid + '.jpeg')
    tree = ET.parse(fpath)
    if tree.find('id') is None:
      im_id = np.array([ct])
    else:
      im_id = np.array([int(tree.find('id').text)])
    objs = tree.findall('object')
    im_w = float(tree.find('size').find('width').text)
    im_h = float(tree.find('size').find('height').text)
    gt_bbox = np.zeros((len(objs), 4), dtype=np.float32)
    gt_class = np.zeros((len(objs), ), dtype=np.int32)
    is_crowd = np.zeros((len(objs), ), dtype=np.int32)
    difficult = np.zeros((len(objs), ), dtype=np.int32)
```

通过遍历所有的 object 标签对，依次读取图像中每个目标实例的标注，并针对每个实例构建一个字典。最终对于图像中所有的目标实例返回一个实例列表：

```
for i, obj in enumerate(objs):
  cname = obj.find('name').text
  gt_class[i] = cname2cid[cname]
  _difficult = int(obj.find('difficult').text)
  x1 = float(obj.find('bndbox').find('xmin').text)
  y1 = float(obj.find('bndbox').find('ymin').text)
  x2 = float(obj.find('bndbox').find('xmax').text)
  y2 = float(obj.find('bndbox').find('ymax').text)
  x1 = max(0, x1)
  y1 = max(0, y1)
  x2 = min(im_w - 1, x2)
  y2 = min(im_h - 1, y2)
  # 这里使用 xywh 格式来表示目标物体真实框
  gt_bbox[i] = [(x1 + x2)/2.0 , (y1 + y2)/2.0, x2 - x1 + 1., y2 - y1 + 1.]
  is_crowd[i] = 0
  difficult[i] = _difficult
```

```
voc_rec = {
  'im_file': img_file,
  'im_id': im_id,
  'h': im_h,
    'w': im_w,
    'is_crowd': is_crowd,
  'gt_class': gt_class,
  'gt_bbox': gt_bbox,
  'gt_poly': [],
  'difficult': difficult
  }
if len(objs) != 0:
  records.append(voc_rec)
```

检测网络训练的过程中，需要同时输入图像、目标矩形框和目标类别，因此需要通过 get_img_data_from_file()函数，使用 cv2.imread()函数加载图像，并将目标实例的坐标转化为相对值。最终返回图像矩阵、图像的大小以及图像中所有目标实例位置和类别。

```
def get_img_data_from_file(record):
  im_file = record['im_file']
  h = record['h']
  w = record['w']
  is_crowd = record['is_crowd']
  gt_class = record['gt_class']
  gt_bbox = record['gt_bbox']
  difficult = record['difficult']
  img = cv2.imread(im_file)
  img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
  # check if h and w in record equals that read from img
  assert img.shape[0] == int(h)
  assert img.shape[1] == int(w)
  gt_boxes, gt_labels = get_bbox(gt_bbox, gt_class)
  # gt_bbox 用相对值
  gt_boxes[:, 0] = gt_boxes[:, 0] / float(w)
  gt_boxes[:, 1] = gt_boxes[:, 1] / float(h)
  gt_boxes[:, 2] = gt_boxes[:, 2] / float(w)
  gt_boxes[:, 3] = gt_boxes[:, 3] / float(h)
  return img, gt_boxes, gt_labels, (h, w)
```

对于一般的检测任务来说，一幅图像上往往会有多个目标物体(每幅图像上的目标数目并不固定)，这样就无法固定每幅图像的目标实例列表的长度(以坐标位置为例，长度为 4×目标数目)。因此在输入网络之前需要统一所有图像的标注长度，通过 get_bbox 将目标矩形框和标签都填充至 50，对于多出图像中目标的部分用 0 补齐。

```
def get_bbox(gt_bbox, gt_class):
  MAX_NUM = 50
  gt_bbox2 = np.zeros((MAX_NUM, 4))
  gt_class2 = np.zeros((MAX_NUM,))
  for i in range(len(gt_bbox)):
    gt_bbox2[i, :] = gt_bbox[i, :]
    gt_class2[i] = gt_class[i]
    if i >= MAX_NUM:
```

```
            break
    return gt_bbox2, gt_class2
```

（2）数据预处理。

在训练之前，通常会对图像做一些随机的变化，产生相似但又不完全相同的样本。其主要作用是扩大训练数据集，抑制过拟合，提升模型的泛化能力，在检测任务中常用的方法主要有以下几种。

随机改变亮度、对比度和颜色：每次加载数据时，在一定范围内随机改变图像的亮度、对比度和颜色的值。

```
def random_distort(img):
  # 随机改变亮度
  def random_brightness(img, lower = 0.5, upper = 1.5):
    e = np.random.uniform(lower, upper)
    return ImageEnhance.Brightness(img).enhance(e)
  # 随机改变对比度
  def random_contrast(img, lower = 0.5, upper = 1.5):
    e = np.random.uniform(lower, upper)
    return ImageEnhance.Contrast(img).enhance(e)
  # 随机改变颜色
  def random_color(img, lower = 0.5, upper = 1.5):
    e = np.random.uniform(lower, upper)
    return ImageEnhance.Color(img).enhance(e)
  ops = [random_brightness, random_contrast, random_color]
  np.random.shuffle(ops)
  img = Image.fromarray(img)
  img = ops[0](img)
  img = ops[1](img)
  img = ops[2](img)
  img = np.asarray(img)
  return img
```

随机填充：每次加载数据时，以一定的概率在图像边缘处添加一定范围内的随机边框。但需要注意的是，填充会改变图像的大小，因此标注也要相应地做出调整（如图 3-2-5 所示）。

```
def random_expand(img,gtboxes,max_ratio = 4.,fill = None, keep_ratio = True, thresh = 0.5):
  if random.random() > thresh:
    return img, gtboxes
  if max_ratio < 1.0:
    return img, gtboxes
  h, w, c = img.shape
  ratio_x = random.uniform(1, max_ratio)
  if keep_ratio:
    ratio_y = ratio_x
  else:
    ratio_y = random.uniform(1, max_ratio)
  oh = int(h * ratio_y)
  ow = int(w * ratio_x)
  off_x = random.randint(0, ow - w)
  off_y = random.randint(0, oh - h)
  out_img = np.zeros((oh, ow, c))
  if fill and len(fill) == c:
```

```
    for i in range(c):
      out_img[:, :, i] = fill[i] * 255.0
  out_img[off_y:off_y + h, off_x:off_x + w, :] = img
  gtboxes[:, 0] = ((gtboxes[:, 0] * w) + off_x) / float(ow)
  gtboxes[:, 1] = ((gtboxes[:, 1] * h) + off_y) / float(oh)
  gtboxes[:, 2] = gtboxes[:, 2] / ratio_x
  gtboxes[:, 3] = gtboxes[:, 3] / ratio_y
  return out_img.astype('uint8'), gtboxes
```

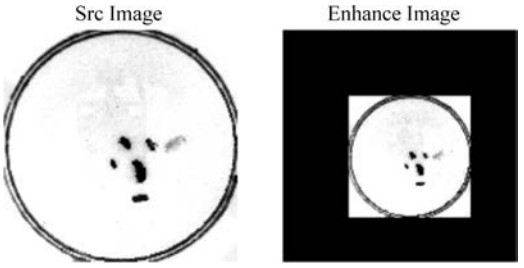


图 3-2-5　随机填充效果

随机裁剪：对图像进行随机的裁剪，但需要注意的是，裁剪会改变图像的大小，因此标注也要相应地调整。

```
def box_crop(boxes, labels, crop, img_shape):
  x, y, w, h = map(float, crop)
  im_w, im_h = map(float, img_shape)
  boxes = boxes.copy()
boxes[:, 0], boxes[:, 2] = (boxes[:, 0] - boxes[:, 2] / 2) * im_w, (boxes[:, 0] + boxes[:, 2] /
2) * im_wboxes[:, 1], boxes[:, 3] = (boxes[:, 1] - boxes[:, 3] / 2) * im_h, (boxes[:, 1] +
boxes[:, 3] / 2) * im_hcrop_box = np.array([x, y, x + w, y + h])
centers = (boxes[:, :2] + boxes[:, 2:]) / 2.0
  mask = np.logical_and(crop_box[:2] <= centers, centers <= crop_box[2:]).all(axis = 1)
  boxes[:, :2] = np.maximum(boxes[:, :2], crop_box[:2])
  boxes[:, 2:] = np.minimum(boxes[:, 2:], crop_box[2:])
  boxes[:, :2] -= crop_box[:2]
  boxes[:, 2:] -= crop_box[:2]
  mask = np.logical_and(mask, (boxes[:, :2] < boxes[:, 2:]).all(axis = 1))
  boxes = boxes * np.expand_dims(mask.astype('float32'), axis = 1)
  labels = labels * mask.astype('float32')
  boxes[:, 0], boxes[:, 2] = (boxes[:, 0] + boxes[:, 2]) / 2 / w, ( boxes[:, 2] - boxes[:, 0]) /
  w boxes[:, 1], boxes[:, 3] = (boxes[:, 1] + boxes[:, 3]) / 2 / h, (boxes[:, 3] - boxes[:, 1]) /
  h return boxes, labels, mask.sum()
```

随机缩放：对图像的大小进行调整。因为标注会转换成图像中相对坐标位置的形式，因此缩放不会对标注造成影响。

```
def random_interp(img, size, interp = None):
  interp_method = [
    cv2.INTER_NEAREST,
    cv2.INTER_LINEAR,
    cv2.INTER_AREA,
    cv2.INTER_CUBIC,
    cv2.INTER_LANCZOS4,
  ]
```

```
    if not interp or interp not in interp_method:
      interp = interp_method[random.randint(0, len(interp_method) - 1)]
    h, w, _ = img.shape
    im_scale_x = size / float(w)
    im_scale_y = size / float(h)
    img = cv2.resize(
      img, None, None, fx=im_scale_x, fy=im_scale_y, interpolation=interp)
    return img
```

随机翻转：对图像按照中心进行对称翻转，相应的标注也要调整。

```
def random_flip(img, gtboxes, thresh=0.5):
    if random.random() > thresh:
      img = img[:, ::-1, :]
      gtboxes[:, 0] = 1.0 - gtboxes[:, 0]
    return img, gtboxes
```

随机打乱标注框的排列顺序：每幅图像存在一个至多个目标实例，每次训练时，随机打乱这些实例标注的顺序。

```
def shuffle_gtbox(gtbox, gtlabel):
    gt = np.concatenate([gtbox, gtlabel[:, np.newaxis]], axis=1)
    idx = np.arange(gt.shape[0])
    np.random.shuffle(idx)
    gt = gt[idx, :]
    return gt[:, :4], gt[:, 4]
```

在读取数据的过程中，我们会按顺序进行上述的数据增强方法，以扩充样本的多样性。通过这种方式每次送入网络的数据都不尽相同：

```
def image_augment(img, gtboxes, gtlabels, size, means=None):
    # 随机改变亮暗、对比度和颜色等
    img = random_distort(img)
    # 随机填充
    img, gtboxes = random_expand(img, gtboxes, fill=means)
    # 随机裁剪
    img, gtboxes, gtlabels, = random_crop(img, gtboxes, gtlabels)
    # 随机缩放
    img = random_interp(img, size)
    # 随机翻转
    img, gtboxes = random_flip(img, gtboxes)
    # 随机打乱真实框排列顺序
    gtboxes, gtlabels = shuffle_gtbox(gtboxes, gtlabels)
    return img.astype('float32'), gtboxes.astype('float32'), gtlabels.astype('int32')
```

接下来，我们通过 get_img_data 来调用前面的函数，实现数据的读入，首先通过 get_img_data_from_file 读取图像、标注文件和图像尺寸，之后通过 image_augment 对图像进行数据增广，最后再将得到的图像进行归一化，并将维度从[H，W，C]调整为[C，H，W]。

```
def get_img_data(record, size=640):
    img, gt_boxes, gt_labels, scales = get_img_data_from_file(record)
    img, gt_boxes, gt_labels = image_augment(img, gt_boxes, gt_labels, size)
    mean = [0.485, 0.456, 0.406]
    std = [0.229, 0.224, 0.225]
```

```
    mean = np.array(mean).reshape((1, 1, -1))
    std = np.array(std).reshape((1, 1, -1))
    img = (img / 255.0 - mean) / std
    img = img.astype('float32').transpose((2, 0, 1))
    return img, gt_boxes, gt_labels, scales
```

最后，是数据加载的最后一步，也是最重要的一步，定义数据读取类 TrainDataset。在 init()函数中，我们通过 get_annotations 获取所有图像的标注和图像所在的路径；在 getitem()函数中通过 get_img_data 返回图像和标注。

```
class TrainDataset(paddle.io.Dataset):
  def __init__(self, datadir, mode = 'train'):
    self.datadir = datadir
    cname2cid = get_insect_names()
    self.records = get_annotations(cname2cid, datadir)
    self.img_size = 640 #get_img_size(mode)
  def __getitem__(self, idx):
    record = self.records[idx]
    # print("print: ", record)
    img, gt_bbox, gt_labels, im_shape = get_img_data(record, size = self.img_size)
    return img, gt_bbox, gt_labels, np.array(im_shape)
```

步骤 3：搭建 YOLOV3 网络

首先介绍在本实践中使用到的 API 接口。

```
paddle.nn.functional.leaky_relu(x,
                                negative_slope = 0.01,
                                name = None):
```

该接口用于实现 leaky_relu 的激活层。

- x(Tensor)：输入 Tensor，数据类型为 float32、float64。
- negative_slope(float，可选)：x＜0 时的斜率。默认值为 0.01。
- name(str，可选)：操作的名称(可选，默认值为 None)。

paddle.add(x, y, name = None)：该接口是逐元素相加算子，输入 x 与输入 y 逐元素相加，并将各个位置的输出元素保存到返回结果中。

- x(Tensor)：输入 Tensor，数据类型为 float32、float64、int32、int64。
- y(Tensor)：输入 Tensor，数据类型为 float32、float64、int32、int64。
- name(str，可选)：操作的名称(可选，默认值为 None)。

```
paddle.vision.ops.yolo_loss(x,
                            gt_box,
                            gt_label,
                            anchors,
                            anchor_mask,
                            class_num,
                            ignore_thresh,
                            downsample_ratio,
                            gt_score = None,
                            use_label_smooth = True,
```

```
                name = None,
                scale_x_y = 1.0):
```

该运算通过给定的预测结果和真实框计算 YOLOV3 损失。

- x(Tensor) ：YOLOV3 损失运算的输入张量，这是一个形状为[N，C，H，W]的四维 Tensor。H 和 W 应该相同，第二维(C)存储框的位置信息，以及每个 anchor box 的置信度得分和 one-hot 分类。数据类型为 float32 或 float64。
- gt_box(Tensor) ：真实框，应该是[N，B，4]的形状。第三维用来承载 x、y、w、h，其中 x、y 是真实框的中心坐标，w、h 是框的宽度和高度，且 x、y、w、h 将除以输入图片的尺寸，缩放到[0，1]区间内。N 是 batch size，B 是图像中所含有的的最多的 box 数目。数据类型为 float32 或 float64。
- gt_label(Tensor) ：真实框的类 id，应该形为[N，B]。数据类型为 int32。
- anchors(list|tuple) ：指定 anchor 框的宽度和高度，将逐对进行解析。
- anchor_mask(list|tuple) ：当前 YOLOV3 损失计算中使用 anchor 的 mask 索引。
- class_num(int) ：要预测的类别数。
- ignore_thresh(float) ：一定条件下忽略某框置信度损失的忽略阈值。
- downsample_ratio(int) ：网络输入 YOLOV3 loss 中的下采样率，因此第一、第二和第三个 loss 的下采样率应分别为 32，16，8。
- gt_score(Tensor)：真实框的混合得分，形为[N，B]。默认为 None。数据类型为 float32 或 float64。
- use_label_smooth(bool)：是否使用平滑标签。默认为 True。
- name(str，可选)：操作的名称(可选，默认值为 None)。
- scale_x_y(float，可选)：缩放解码边界框的中心点。默认值为 1.0。

(1) YOLOV3 标签分配。

IoU 是目标检测过程中常用的标准，用于反映两个框之间的交并比。因此，在进行网络搭建之前，首先要定义用于计算 IoU 的函数 box_iou_xywh。

```
def box_iou_xywh(box1, box2):
  x1min, y1min = box1[0] - box1[2]/2.0, box1[1] - box1[3]/2.0
  x1max, y1max = box1[0] + box1[2]/2.0, box1[1] + box1[3]/2.0
  s1 = box1[2] * box1[3]
  x2min, y2min = box2[0] - box2[2]/2.0, box2[1] - box2[3]/2.0
  x2max, y2max = box2[0] + box2[2]/2.0, box2[1] + box2[3]/2.0
  s2 = box2[2] * box2[3]
  xmin = np.maximum(x1min, x2min)
  ymin = np.maximum(y1min, y2min)
  xmax = np.minimum(x1max, x2max)
  ymax = np.minimum(y1max, y2max)
  inter_h = np.maximum(ymax - ymin, 0.)
  inter_w = np.maximum(xmax - xmin, 0.)
  intersection = inter_h * inter_w
  union = s1 + s2 - intersection
  iou = intersection / union
  return iou
```

YOLOV3 在训练的过程中首先需要产生锚框，并根据标注对候选框分配标签。每一个 objectness 标注为 1 的锚框，会有一个真实的标注框跟它对应，该锚框所属物体类别，是其所对应的真实框包含的物体类别。这里使用 one-hot 向量来表示类别标签 label。比如一共有 10 个分类，而真实的标注框里面包含的物体类别是第 2 类，则 label 为(0,1,0,0,0,0,0,0,0,0)，具体的过程如图 3-2-6 所示。

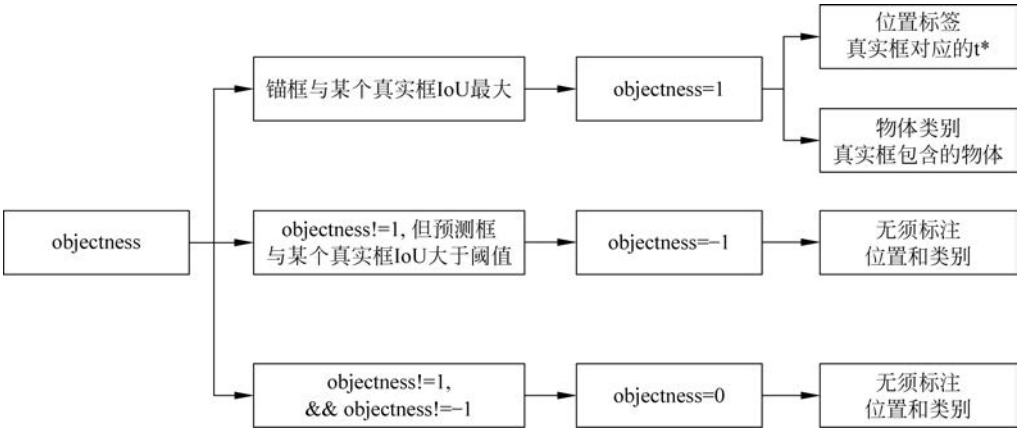


图 3-2-6　标签分配过程

(2) YOLOV3 特征提取网络。

YOLOV3 算法使用的特征提取网络是 Darknet53。Darknet53 在 ImageNet 图像分类任务上取得了很好的成绩，网络的具体结构如图 3-2-7 所示。在检测任务中，将图中 C0 后面的平均池化、全连接层和 Softmax 去掉，保留从输入到 C0 部分的网络结构，作为检测模型的基础网络结构，也称为骨干网络。YOLOV3 模型会在骨干网络的基础上，再添加检测相关的网络模块。

因为 DarkNet53 的网络层数比较多，因此我们采用了模块化的搭建形式。首先，搭建卷积＋批归一化层的子模块 ConvBNLayer()函数，它由一层卷积和一层批归一化层组成，根据输入可以选择是否使用 leaky_relu 作为激活函数。

```
class ConvBNLayer(paddle.nn.Layer):
  def __init__(self, ch_in, ch_out,
           kernel_size=3, stride=1, groups=1,
           padding=0, act="leaky"):
    super(ConvBNLayer, self).__init__()

    self.conv = paddle.nn.Conv2D(
      in_channels=ch_in,
      out_channels=ch_out,
      kernel_size=kernel_size,
      stride=stride,
      padding=padding,
      groups=groups,
      weight_attr=paddle.ParamAttr(
        initializer=paddle.nn.initializer.Normal(0., 0.02)),
      bias_attr=False)
```

```
    self.batch_norm = paddle.nn.BatchNorm2D(
      num_features = ch_out,
      weight_attr = paddle.ParamAttr(
        initializer = paddle.nn.initializer.Normal(0.,0.02),
        regularizer = paddle.regularizer.L2Decay(0.)),
      bias_attr = paddle.ParamAttr(
        initializer = paddle.nn.initializer.Constant(0.0),
        regularizer = paddle.regularizer.L2Decay(0.)))
    self.act = act
  def forward(self, inputs):
    out = self.conv(inputs)
    out = self.batch_norm(out)
    if self.act == 'leaky':
      out = F.leaky_relu(x = out, negative_slope = 0.1)
    return out
```

DarkNet53网络结构图

	类型	输出通道数	卷积核	输出特征图大小	
	Softmax			1000	
	全连接			1000	
	平均池化	1024	全局池化	1×1	
4×残差块	残差			8×8	C0
	卷积	1024	3×3		
	卷积	512	1×1		
	卷积	1024	3×3/2	8×8	
8×残差块	残差			16×16	C1
	卷积	512	3×3		
	卷积	256	1×1		
	卷积	512	3×3/2	16×16	
8×残差块	残差			32×32	C2
	卷积	256	3×3		
	卷积	128	1×1		
	卷积	256	3×3/2	32×32	
2×残差块	残差			64×64	
	卷积	128	3×3		
	卷积	64	1×1		
	卷积	128	3×3/2	64×64	
1×残差块	残差			128×128	
	卷积	64	3×3		
	卷积	32	1×1		
	卷积	64	3×3/2	128×128	
	卷积	32	3×3	256×256	

图 3-2-7　Darknet53 网络结构

DownSample 类是在网络中用于下采样的模块，在 DarkNet53 中下采样是通过步长为 2

的卷积层实现的，可以实现特征分别率减半。

```
class DownSample(paddle.nn.Layer):
  # 下采样，图片尺寸减半，具体实现方式是使用 stirde = 2 的卷积
  def __init__(self,
           ch_in,
           ch_out,
           kernel_size = 3,
           stride = 2,
           padding = 1):
    super(DownSample, self).__init__()
    self.conv_bn_layer = ConvBNLayer(
      ch_in = ch_in,
      ch_out = ch_out,
      kernel_size = kernel_size,
      stride = stride,
      padding = padding)
    self.ch_out = ch_out
  def forward(self, inputs):
    out = self.conv_bn_layer(inputs)
    return out
```

在 DarkNet53 中，引入了 ResNet 跳跃连接的思路和残差结构。通过 BasicBlock 类定义 DarkNet53 中的基本残差结构。对于输入 x，经过两次卷积＋批归一化结构后，通过 paddle.add 与原始的输入 x 相加。

```
class BasicBlock(paddle.nn.Layer):
  def __init__(self, ch_in, ch_out):
    super(BasicBlock, self).__init__()
    self.conv1 = ConvBNLayer(
      ch_in = ch_in,
      ch_out = ch_out,
      kernel_size = 1,
      stride = 1,
      padding = 0
      )
    self.conv2 = ConvBNLayer(
      ch_in = ch_out,
      ch_out = ch_out * 2,
      kernel_size = 3,
      stride = 1,
      padding = 1
      )
  def forward(self, inputs):
    conv1 = self.conv1(inputs)
    conv2 = self.conv2(conv1)
    out = paddle.add(x = inputs, y = conv2)
    return out
```

LayerWarp 类以 BasicBlock 为基础，组合多个残差结构，构成 Darknet53 网络的一个层级。

```
class LayerWarp(paddle.nn.Layer):
  def __init__(self, ch_in, ch_out, count, is_test = True):
```

```
    super(LayerWarp,self).__init__()
    self.basicblock0 = BasicBlock(ch_in,
      ch_out)
    self.res_out_list = []
    for i in range(1, count):
      res_out = self.add_sublayer("basic_block_%d" % (i), # 使用 add_sublayer 添加子层
        BasicBlock(ch_out*2,
          ch_out))
      self.res_out_list.append(res_out)
  def forward(self,inputs):
    y = self.basicblock0(inputs)
    for basic_block_i in self.res_out_list:
      y = basic_block_i(y)
    return y
```

设计完用于构建网络的各个子模块后，接下来就要通过这些模块来搭建 DarkNet53，构建 YOLOV3 的特征提取网络。具体地，根据图 3-2-7 的网络结构示意，首先通过 ConvBNLayer 实现第一个卷积层，再通过 DownSample 实现特征图的下采样。

```
# DarkNet 每组残差块的个数，来自 DarkNet 的网络结构图
DarkNet_cfg = {53: ([1, 2, 8, 8, 4])}
class DarkNet53_conv_body(paddle.nn.Layer):
  def __init__(self):
    super(DarkNet53_conv_body, self).__init__()
    self.stages = DarkNet_cfg[53]
    self.stages = self.stages[0:5]
    # 第一层卷积
    self.conv0 = ConvBNLayer(
      ch_in=3,
      ch_out=32,
      kernel_size=3,
      stride=1,
      padding=1)
    # 下采样，使用 stride=2 的卷积来实现
    self.downsample0 = DownSample(
      ch_in=32,
      ch_out=32 * 2)
```

接下来通过循环地调用 LayerWarp 实现图 3-2-7 中的由不同数量残差结构组成的卷积单元（框线内的部分），每两个单元之间通过 DownSample（带有步长的卷积）实现特征图的下采样。同时考虑后面网络结构的需要，把 C0，C1，C2 特征图都作为返回值。

```
# 添加各个层级的实现
self.darknet53_conv_block_list = []
self.downsample_list = []
for i, stage in enumerate(self.stages):
  conv_block = self.add_sublayer(
    "stage_%d" % (i),
    LayerWarp(32*(2**(i+1)),
    32*(2**i),
    stage))
  self.darknet53_conv_block_list.append(conv_block)
# 两个层级之间使用 DownSample 将尺寸减半
```

```
for i in range(len(self.stages) - 1):
  downsample = self.add_sublayer(
    "stage_%d_downsample" % i,
    DownSample(ch_in=32*(2**(i+1)),
      ch_out=32*(2**(i+2))))
  self.downsample_list.append(downsample)
def forward(self,inputs):
  out = self.conv0(inputs)
  out = self.downsample0(out)
  blocks = []
  for i, conv_block_i in enumerate(self.darknet53_conv_block_list):
                                                    #依次将各个层级作用在输入上面
    out = conv_block_i(out)
    blocks.append(out)
    if i < len(self.stages) - 1:
      out = self.downsample_list[i](out)
  return blocks[-1:-4:-1]                           # 将 C0, C1, C2 作为返回值
```

(3) YOLOV3 预测网络特征提取。

通过 Darknet53 和上采样得到的特征,并不能直接用于模型预测,还需要经过一系列的卷积过程。因此,通过 YOLODetectionBlock 来进一步提取特征,YOLODetectionBlock 由 6 组卷积和批归一化的结构组成,同时返回中间和最后的特征图。

```
class YOLODetectionBlock(paddle.nn.Layer):
  def __init__(self,ch_in,ch_out,is_test=True):
    super(YOLODetectionBlock, self).__init__()
    self.conv0 = ConvBNLayer(
      ch_in=ch_in,ch_out=ch_out,kernel_size=1,stride=1,padding=0)
    self.conv1 = ConvBNLayer(
      ch_in=ch_out,
      ch_out=ch_out*2,
      kernel_size=3,
      stride=1,
      padding=1)
    self.conv2 = ConvBNLayer(
      ch_in=ch_out*2,
      ch_out=ch_out,
      kernel_size=1,
      stride=1,
      padding=0)
    self.conv3 = ConvBNLayer(
      ch_in=ch_out,
      ch_out=ch_out*2,
      kernel_size=3,
      stride=1,
      padding=1)
    self.route = ConvBNLayer(
      ch_in=ch_out*2,
      ch_out=ch_out,
      kernel_size=1,
      stride=1,
      padding=0)
    self.tip = ConvBNLayer(
```

```
      ch_in = ch_out,
      ch_out = ch_out * 2,
      kernel_size = 3,
      stride = 1,
      padding = 1)
  def forward(self, inputs):
    out = self.conv0(inputs)
    out = self.conv1(out)
    out = self.conv2(out)
    out = self.conv3(out)
    route = self.route(out)
    tip = self.tip(route)
    return route, tip
```

（4）YOLOV3 上采样部分。

YOLOV3 将在三个不同尺度的特征图上进行预测，因此需要根据 Darknet53 提取的特征图和 Upsample 类构建用于预测的多个尺度的特征图：

```
class Upsample(paddle.nn.Layer):
  def __init__(self, scale = 2):
    super(Upsample,self).__init__()
    self.scale = scale

  def forward(self, inputs):
    # get dynamic upsample output shape
    shape_nchw = paddle.shape(inputs)
    shape_hw = paddle.slice(shape_nchw, axes = [0], starts = [2], ends = [4])
    shape_hw.stop_gradient = True
    in_shape = paddle.cast(shape_hw, dtype = 'int32')
    out_shape = in_shape * self.scale
    out_shape.stop_gradient = True

    # reisze by actual_shape
    out = paddle.nn.functional.interpolate(
      x = inputs, scale_factor = self.scale, mode = "NEAREST")
    return out
```

（5）YOLOV3 整体结构。

在实现 YOLOV3 的各个组件之后，接下来要定义 YOLOV3 模型的整体结构，其中包括 init、forward 和 get_loss。

在 init 部分，通过 DarkNet53_conv_body() 搭建特征提取网络 DarkNet53，并通过 YOLODetectionBlock 和 Upsample 构建用于预测的三种尺度的特征图。对于每种尺度特种图使用 $K(C+5)$ 的 1×1 卷积进行预测，其中 C 是预测类别，K 是每个尺度特征图上预设的锚点种类数量。

其中损失部分调用了飞桨平台用于计算 YOLOV3 损失的接口 paddle.vision.ops.yolo_loss，YOLOV3 损失包括三个主要部分：框位置损失、目标性损失、分类损失。L1 损失用于框坐标 (w,h)，同时，sigmoid 交叉熵损失用于框坐标 (x,y)、目标性损失和分类损失。

每个真实框将在所有 anchor 中找到最匹配的 anchor，对该 anchor 的预测将会计算全部（三种）损失，但是没有匹配 GT box（ground truth box，真实框）的 anchor 的预测只会产生目

标性损失。为了权衡大框(box)和小框(box)之间的框坐标损失,框坐标损失将与比例权重相乘而得。

$$\text{loss}=(\text{loss}_{xy}+\text{loss}_{wh})\times\text{weight}_{\text{box}}+\text{loss}_{\text{conf}}+\text{loss}_{\text{class}}$$

YOLOV3 loss 前的网络输出形状为$[N,C,H,W]$,H 和 W 应该相同,用来指定网格(grid)大小。每个网格点预测 S 个边界框(bounding boxes),S 由每个尺度中 anchors 簇的个数指定。在第二维(表示通道的维度)中,C 的值应为 $S\times$(class_num+5),class_num 是源数据集的对象种类数(如 coco 中为 80),另外,除了存储 4 个边界框位置坐标 x,y,w,h,还包括边界框以及每个 anchor 框的 one-hot 关键字的置信度得分。

```
class YOLOV3(paddle.nn.Layer):
  def __init__(self, num_classes=7):
    super(YOLOV3,self).__init__()
    self.num_classes = num_classes
    # 提取图像特征的骨干代码
    self.block = DarkNet53_conv_body()
    self.block_outputs = []
    self.YOLO_blocks = []
    self.route_blocks_2 = []
    # 生成 3 个层级的特征图 P0, P1, P2
    for i in range(3):
      # 添加从 ci 生成 ri 和 ti 的模块
      YOLO_block = self.add_sublayer(
        "YOLO_detecton_block_%d" % (i),
        YOLODetectionBlock(
                ch_in=512//(2**i)*2 if i==0 else 512//(2**i)*2 + 512//(2**i),
                ch_out = 512//(2**i)))
      self.YOLO_blocks.append(YOLO_block)
      num_filters = 3 * (self.num_classes + 5)
      block_out = self.add_sublayer(
        "block_out_%d" % (i),
        paddle.nn.Conv2D(in_channels=512//(2**i)*2,
           out_channels=num_filters,
           kernel_size=1,
           stride=1,
           padding=0,
           weight_attr=paddle.ParamAttr(
             initializer=paddle.nn.initializer.Normal(0., 0.02)),
           bias_attr=paddle.ParamAttr(
             initializer=paddle.nn.initializer.Constant(0.0),
             regularizer=paddle.regularizer.L2Decay(0.))))
      self.block_outputs.append(block_out)
      if i<2:
        # 对 ri 进行卷积
        route = self.add_sublayer("route2_%d"%i,
                        ConvBNLayer(ch_in=512//(2**i),
                              ch_out=256//(2**i),
                              kernel_size=1,
                              stride=1,
                              padding=0))
        self.route_blocks_2.append(route)
```

```
        # 将 ri 放大以便跟 c_{i+1}保持同样的尺寸
        self.upsample = Upsample()
```

在 forward()函数中确定 YOLOV3 网络结构的各层之间前向传播的先后顺序。

```
def forward(self, inputs):
  outputs = []
  blocks = self.block(inputs)
  for i, block in enumerate(blocks):
    if i > 0:
      # 将 r_{i-1}经过卷积和上采样之后得到特征图,与这一级的 ci 进行拼接
      block = paddle.concat([route, block], axis=1)
    # 从 ci 生成 ti 和 ri
    route, tip = self.YOLO_blocks[i](block)
    # 从 ti 生成 pi
    block_out = self.block_outputs[i](tip)
    # 将 pi 放入列表
    outputs.append(block_out)
    if i < 2:
      # 对 ri 进行卷积调整通道数
      route = self.route_blocks_2[i](route)
      # 对 ri 进行放大,使其尺寸和 c_{i+1}保持一致
      route = self.upsample(route)
  return outputs
```

通过 paddle.vision.ops.YOLO_loss 直接计算损失函数,过程更简洁,速度也更快。

```
def get_loss(self, outputs, gtbox, gtlabel, gtscore=None,
       anchors = [10, 13, 16, 30, 33, 23, 30, 61, 62, 45, 59, 119, 116, 90, 156, 198, 373, 326],
       anchor_masks = [[6, 7, 8], [3, 4, 5], [0, 1, 2]],
       ignore_thresh=0.7,
       use_label_smooth=False):
    self.losses = []
    downsample = 32
    for i, out in enumerate(outputs): # 对三个层级分别求损失函数
      anchor_mask_i = anchor_masks[i]
      loss = paddle.vision.ops.YOLO_loss(
          x=out, # out 是 P0, P1, P2 中的一个
          gt_box=gtbox,                          # 真实框坐标
          gt_label=gtlabel,                      # 真实框类别
          gt_score=gtscore,                      # 真实框得分,使用 mixup 训练技巧时需要
          anchors=anchors,                       # 锚框尺寸,包含[w0, h0, w1, h1, ..., w8, h8]共
                                                   9 个锚框的尺寸
          anchor_mask=anchor_mask_i,             # 筛选锚框的 mask
          class_num=self.num_classes,            # 分类类别数
          ignore_thresh=ignore_thresh,           # 当预测框与真实框 IoU > ignore_thresh,标注
                                                   objectness = -1
          downsample_ratio=downsample,           # 特征图相对于原图缩小倍数
          use_label_smooth=False)                # 使用 label_smooth
      self.losses.append(paddle.mean(loss))      #mean 对每张图片求和
      downsample = downsample // 2               # 下一级特征图的缩放倍数会减半
    return sum(self.losses)                      # 对每个层级求和
```

步骤 4：训练 YOLOV3 网络

训练过程如图 3-2-8 所示,输入图片经过特征提取后得到三个层级的输出特征图 P0

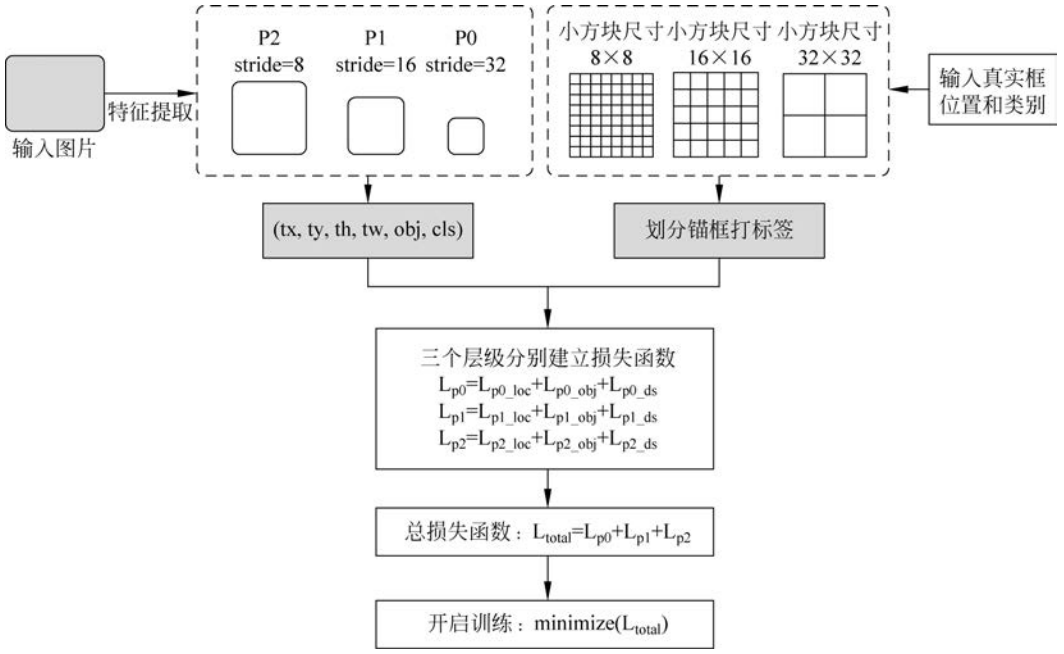


图 3-2-8　YOLOV3 训练过程流程图

(stride=32)、P1(stride=16)和 P2(stride=8)，相应地分别使用不同大小的小方块区域去生成对应的锚框和预测框，并对这些锚框进行标注。

P0 层级特征图，对应使用 32×32 大小的小方块，在每个区域中心生成大小分别为[116,90]、[156,198]、[373,326]的三种锚框。

P1 层级特征图，对应使用 16×16 大小的小方块，在每个区域中心生成大小分别为[30,61]、[62,45]、[59,119]的三种锚框。

P2 层级特征图，对应使用 8×8 大小的小方块，在每个区域中心生成大小分别为[10,13]、[16,30]、[33,23]的三种锚框。

将三个层级的特征图与对应锚框之间的标签关联起来，并建立损失函数，总的损失函数等于三个层级的损失函数相加。通过极小化损失函数，可以开启端到端的训练过程。

```
def train():
  model = YOLOV3(num_classes = NUM_CLASSES)  #创建模型
  learning_rate = get_lr()
  opt = paddle.optimizer.Momentum(
          learning_rate = learning_rate,
          momentum = 0.9,
          weight_decay = paddle.regularizer.L2Decay(0.0005),
          parameters = model.parameters())     #创建优化器
  MAX_EPOCH = 1
  for epoch in range(MAX_EPOCH):
    for i, data in enumerate(train_loader()):
      img, gt_boxes, gt_labels, img_scale = data
      gt_scores = np.ones(gt_labels.shape).astype('float32')
      gt_scores = paddle.to_tensor(gt_scores)
      img = paddle.to_tensor(img)
```

```
gt_boxes = paddle.to_tensor(gt_boxes)
gt_labels = paddle.to_tensor(gt_labels)
outputs = model(img)                                    #前向传播,输出[P0, P1, P2]
loss = model.get_loss(outputs, gt_boxes, gt_labels, gtscore=gt_scores,
                      anchors = ANCHORS,
                      anchor_masks = ANCHOR_MASKS,
                      ignore_thresh=IGNORE_THRESH,
                      use_label_smooth=False)    # 计算损失函数

loss.backward()                                         # 反向传播计算梯度
opt.step()                                              # 更新参数
opt.clear_grad()
```

训练过程如图 3-2-9 所示，在训练过程中会输出全部数据训练的轮数、batch 迭代的次数和训练时的损失。

```
2021-02-21 13:16:32[TRAIN]epoch 0, iter 0, output loss: [17515.46]
2021-02-21 13:16:45[TRAIN]epoch 0, iter 10, output loss: [711.6523]
2021-02-21 13:16:58[TRAIN]epoch 0, iter 20, output loss: [177.56128]
2021-02-21 13:17:09[TRAIN]epoch 0, iter 30, output loss: [100.74901]
2021-02-21 13:17:22[TRAIN]epoch 0, iter 40, output loss: [109.4012]
2021-02-21 13:17:36[TRAIN]epoch 0, iter 50, output loss: [86.60315]
2021-02-21 13:17:49[TRAIN]epoch 0, iter 60, output loss: [73.88124]
2021-02-21 13:18:02[TRAIN]epoch 0, iter 70, output loss: [51.598812]
2021-02-21 13:18:15[TRAIN]epoch 0, iter 80, output loss: [66.547485]
2021-02-21 13:18:27[TRAIN]epoch 0, iter 90, output loss: [65.25056]
2021-02-21 13:18:40[TRAIN]epoch 0, iter 100, output loss: [87.08785]
2021-02-21 13:18:52[TRAIN]epoch 0, iter 110, output loss: [76.32029]
2021-02-21 13:19:06[TRAIN]epoch 0, iter 120, output loss: [72.307175]
2021-02-21 13:19:20[TRAIN]epoch 0, iter 130, output loss: [78.60363]
2021-02-21 13:19:34[TRAIN]epoch 0, iter 140, output loss: [61.50921]
2021-02-21 13:19:47[TRAIN]epoch 0, iter 150, output loss: [57.60893]
2021-02-21 13:20:00[TRAIN]epoch 0, iter 160, output loss: [48.932396]
2021-02-21 13:20:14[TRAIN]epoch 0, iter 170, output loss: [70.52108]
2021-02-21 13:20:28[TRAIN]epoch 0, iter 180, output loss: [57.51571]
2021-02-21 13:20:42[TRAIN]epoch 0, iter 190, output loss: [54.175972]
2021-02-21 13:20:54[TRAIN]epoch 0, iter 200, output loss: [55.90041]
2021-02-21 13:21:07[TRAIN]epoch 0, iter 210, output loss: [52.864914]
```

图 3-2-9　YOLOV3 训练过程中的部分输出

步骤 5：YOLOV3 预测模型

模型的预测过程如图 3-2-10 所示，可以分为两步：

（1）通过网络输出计算出预测框位置和所属类别的得分；

（2）使用非极大值抑制来消除重叠较大的预测框。

在 YOLOV3 类中添加 get_pred()函数，将网络输出的特征转换成网络预测的矩形框坐标和矩形框对应的类别：

```
def get_pred(self,
             outputs,
             im_shape=None,
             anchors=[10, 13, 16, 30, 33, 23, 30, 61, 62, 45, 59, 119, 116, 90, 156, 198, 373, 326],
             anchor_masks=[[6, 7, 8], [3, 4, 5], [0, 1, 2]],
```

```
        valid_thresh = 0.01):
downsample = 32
total_boxes = []
total_scores = []
for i, out in enumerate(outputs):
  anchor_mask = anchor_masks[i]
  anchors_this_level = []
  for m in anchor_mask:
    anchors_this_level.append(anchors[2 * m])
    anchors_this_level.append(anchors[2 * m + 1])
  boxes, scores = paddle.vision.ops.YOLO_box(
      x = out,
      img_size = im_shape,
      anchors = anchors_this_level,
      class_num = self.num_classes,
      conf_thresh = valid_thresh,
      downsample_ratio = downsample,
      name = "YOLO_box" + str(i))
  total_boxes.append(boxes)
  total_scores.append(
        paddle.transpose(
        scores, perm = [0, 2, 1]))
  downsample = downsample // 2
YOLO_boxes = paddle.concat(total_boxes, axis = 1)
YOLO_scores = paddle.concat(total_scores, axis = 2)
return YOLO_boxes, YOLO_scores
```

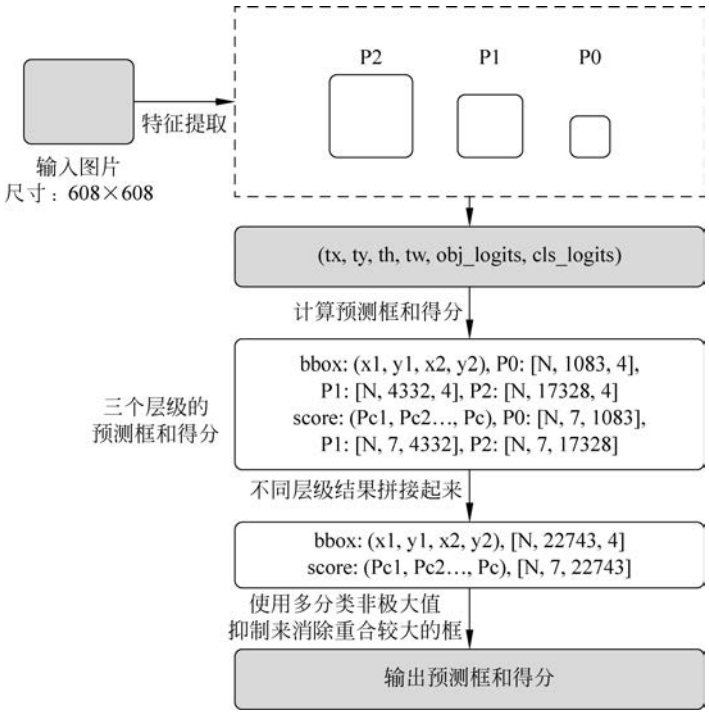


图 3-2-10　YOLOV3 网络预测过程流程图

因为每个目标可能会被不同的锚框覆盖，可能会被预测出多次，因此需要定义 multiclass_nms()函数，对 YOLOV3 的预测结果进行非极大值抑制，对于重叠的矩形框只保留置信度最高的目标：

```
def multiclass_nms(bboxes, scores, score_thresh = 0.01, nms_thresh = 0.45, pre_nms_topk = 1000,
pos_nms_topk = 100):
  batch_size = bboxes.shape[0]
  class_num = scores.shape[1]
  rets = []
  for i in range(batch_size):
    bboxes_i = bboxes[i]
    scores_i = scores[i]
    ret = []
    for c in range(class_num):
      scores_i_c = scores_i[c]
      keep_inds = nms(bboxes_i, scores_i_c, score_thresh, nms_thresh, pre_nms_topk, i = i, c = c)
      if len(keep_inds) < 1:
        continue
      keep_bboxes = bboxes_i[keep_inds]
      keep_scores = scores_i_c[keep_inds]
      keep_results = np.zeros([keep_scores.shape[0], 6])
      keep_results[:, 0] = c
      keep_results[:, 1] = keep_scores[:]
      keep_results[:, 2:6] = keep_bboxes[:, :]
      ret.append(keep_results)
    if len(ret) < 1:
      rets.append(ret)
      continue
    ret_i = np.concatenate(ret, axis = 0)
    scores_i = ret_i[:, 1]
    if len(scores_i) > pos_nms_topk:
      inds = np.argsort(scores_i)[::-1]
      inds = inds[:pos_nms_topk]
      ret_i = ret_i[inds]
    rets.append(ret_i)
return rets
```

最后我们通过定义 test()函数使用训练好的 YOLOV3 模型进行预测。首先使用 YOLOV3 中的 forward()函数提取图像的预测特征，然后通过 get_pred 将预测特征转换为网络预测的矩形框和对应的类别，最后通过 multiclass_nms 去除重叠的预测结果，得到网络最终的预测结果。

```
def test():
  model = YOLOV3(num_classes = NUM_CLASSES)
  params_file_path = '/home/aistudio/YOLO_epoch0'
  model_state_dict = paddle.load(params_file_path)
  model.load_dict(model_state_dict)
  model.eval()
  total_results = []
  test_loader = test_data_loader(TESTDIR, batch_size= 1, mode = 'test')
  for i, data in enumerate(test_loader()):
    img_name, img_data, img_scale_data = data
```

```
img = paddle.to_tensor(img_data)
img_scale = paddle.to_tensor(img_scale_data)
outputs = model.forward(img)
bboxes, scores = model.get_pred(outputs,
                im_shape=img_scale,
                anchors=ANCHORS,
                anchor_masks=ANCHOR_MASKS,
                valid_thresh = VALID_THRESH)
bboxes_data = bboxes.numpy()
scores_data = scores.numpy()
result = multiclass_nms(bboxes_data, scores_data,
          score_thresh=VALID_THRESH,
          nms_thresh=NMS_THRESH,
          pre_nms_topk=NMS_TOPK,
          pos_nms_topk=NMS_POSK)
for j in range(len(result)):
  result_j = result[j]
  img_name_j = img_name[j]
  total_results.append([img_name_j, result_j.tolist()])
```

预测完成后，通过可视化，可以得到如图 3-2-11 所示的结果，不同类型的昆虫被表示为由不同类别的矩形框包裹着，看起来训练的结果还不错。

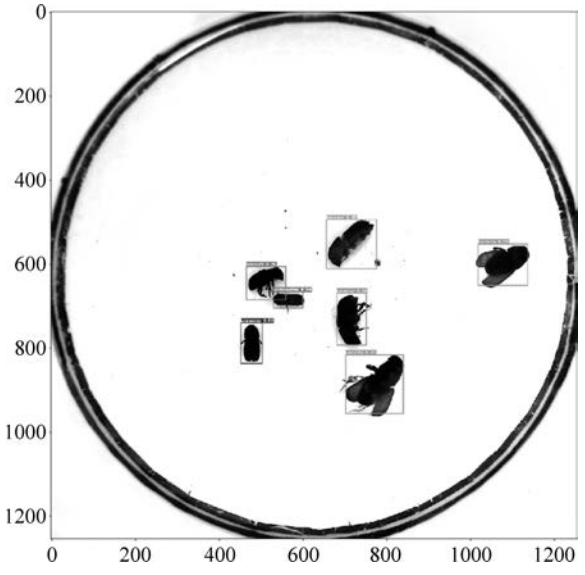


图 3-2-11　预测结果示例

至此，我们就完成了 YOLOV3 网络的搭建、训练和预测过程，你学会了吗？

3.2.2　基于 PP-YOLO 模型的昆虫检测

在使用 YOLOV3 实现昆虫识别的实践中，我们通过大量的代码实现了数据预处理、数据加载、模型构建，以及模型训练测试的过程。在这里我们将使用 PaddleDetection 快速地实现基于 PP-YOLO 进行昆虫识别。

基于 PP-YOLO 模型的昆虫检测

步骤 1：认识 AI 识虫数据集

这里使用的数据集也是昆虫识别的数据集，包含 1693 张训练图像、245 张验证图像和 245 张测试图像。与之前不同的是，数据存储的目录结构做出相应的调整，与 PASCAL VOC 的目录结构相匹配。如图 3-2-12 所示，目录下有标注、图像和图像划分设置三个文件夹。其中，标注文件夹和图像文件夹下各自分为测试、训练和验证三个文件夹，并分别存储对应测试、训练和验证集的标注和图像。图像划分设置文件夹下存储训练、验证和测试的名单以及类别列表。

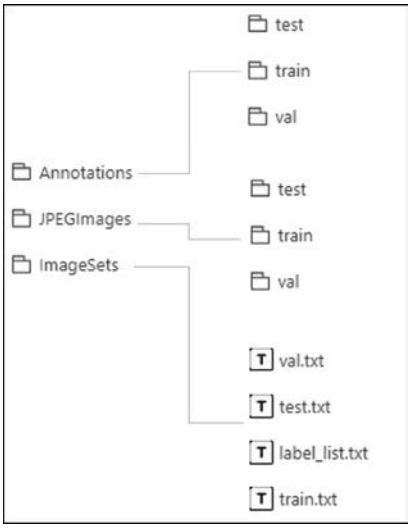


图 3-2-12　数据存储的目录结构

步骤 2：环境安装

在本次实践中，我们要使用 PaddleDetection，因此首先要下载并安装 PaddleDetection 的环境。与 3.1 节的实践一样，通过 git clone 命令下载 PaddleDetection 源码，并安装 PaddleDetection 所需的依赖。

```
!git clone https://github.com/PaddlePaddle/PaddleDetection
 % cd PaddleDetection
!pip install -r requirements.txt
```

步骤 3：模型训练、验证和评估

在完成数据部署和环境准备后，可以直接通过执行 train.py 来训练网络。在这里我们需要使用与 PP-YOLO 相对应的配置文件。在如图 3-2-13 所示配置文件中，可以设置数据

```
architecture: YOLOv3
use_gpu: true
max_iters: 25000
log_smooth_window: 20
log_iter: 20
save_dir: output
snapshot_iter: 1000
metric: VOC
pretrain_weights: https://paddle-image
weights: output/ppyolo/model_final
num_classes: 7
use_fine_grained_loss: true
use_ema: true
ema_decay: 0.9998

YOLOv3:
  backbone: ResNet
  yolo_head: YOLOv3Head
  use_fine_grained_loss: true

ResNet:
  norm_type: sync_bn
  freeze_at: 0
  freeze_norm: false
  norm_decay: 0.
  depth: 50
  feature_maps: [3, 4, 5]
  variant: d
  dcn_v2_stages: [5]

YOLOv3Head:
  anchor_masks: [[6, 7, 8], [3, 4, 5], [0, 1, 2]]
  anchors: [[10, 13], [16, 30], [33, 23],
           [30, 61], [62, 45], [59, 119],
           [116, 90], [156, 198], [373, 326]]
  norm_decay: 0.
  coord_conv: true
  iou_aware: true
  iou_aware_factor: 0.4
  scale_x_y: 1.05
  spp: true
  yolo_loss: YOLOv3Loss
  nms: MatrixNMS
  drop_block: true

YOLOv3Loss:
  batch_size: 12
  ignore_thresh: 0.7
  scale_x_y: 1.05
  label_smooth: false
  use_fine_grained_loss: true
  iou_loss: IouLoss
  iou_aware_loss: IouAwareLoss

IouLoss:
  loss_weight: 2.5
  max_height: 608
  max_width: 608

IouAwareLoss:
  loss_weight: 1.0
  max_height: 608
  max_width: 608

TrainReader:
  inputs_def:
    fields: ['image', 'gt_bbox', 'gt_class', 'gt_score']
    num_max_boxes: 50
  dataset:
    !VOCDataSet
      dataset_dir: dataset/insect
      anno_path: ImageSets/train.txt
      image_dir: ImageSets
      label_list: ImageSets/label_list.txt
      use_default_label: false
      with_background: false
  sample_transforms:
    - !DecodeImage
      to_rgb: True
      with_mixup: True
    - !MixupImage
      alpha: 1.5
      beta: 1.5
    - !ColorDistort {}
    - !RandomExpand
      fill_value: [123.675, 116.28, 103.53]
    - !RandomCrop {}
    - !RandomFlipImage
      is_normalized: false
    - !NormalizeBox {}
    - !PadBox
      num_max_boxes: 50
    - !BboxXYXY2XYWH {}
  batch_transforms:
  - !RandomShape
    sizes: [320, 352, 384, 416, 448, 480, 512, 544, 576, 608]
    random_inter: True
  - !NormalizeImage
    mean: [0.485, 0.456, 0.406]
    std: [0.229, 0.224, 0.225]
```

图 3-2-13　PP-YOLO 配置文件

集的数据格式、数据读取阶段的各种数据增强方法和 PP-YOLO 的各种网络结构、参数配置等。

```
! python tools/train.py -c ../../work/ppyolo.yml --eval
```

训练完成后可以通过执行 eval.py 和 infer.py 来进行模型的验证和预测。与训练阶段不同的是，除了要给定配置文件外，在验证阶段还需要给出训练好的权重、预测时需要给出训练好的权重和需要预测的图像。

```
! python tools/eval.py -c ../../work/ppyolo.yml -o weights=../../work/best_model
! python tools/infer.py -c ../../work/ppyolo.yml --infer_img=dataset/insect/JPEGImages/
test/1898.jpeg -o weights=../../work/best_model
```

3.3　实践三：基于 DETR 模型的目标检测

基于 DETR 模型的目标检测

本节将使用 DETR 来实现 COCO 数据集上的目标检测。

DETR 即 Detection Transformer，是 Facebook AI 的研究者提出的一种借助基于 Transformer 的编码器-解码器体系结构进行目标检测的方法。它是第一个将 Transformer 成功整合为检测 pipeline 中心构建块的目标检测框架。与之前的目标检测方法相比，DETR 有效地消除了对许多手工设计的组件的需求，例如非最大抑制（Non-Maximum Suppression，NMS）、锚点（Anchor）生成等。

本书提出了一个非常简单的端到端的框架，DETR 的网络结构很简单，分为三个部分：第一部分是一个传统 CNN，用于提取图片特征到更高维度；第二部分是一个 Transformer 的 Encoder 和 Decoder，用来提取 Bounding Box；第三部分是 Bipartite matching loss，用来训练网络。

步骤 1：COCO 数据集与数据下载

MS COCO 的全称是 Microsoft Common Objects in Context，是微软团队提供的一个可以用来进行图像识别的数据集，与 ImageNet 竞赛一样，被视为计算机视觉领域最受关注和最权威的比赛之一。

COCO 数据集是一个大型的、丰富的目标检测（Image Detection）、语义分割（Semantic Segmentation）和图像标题（Image Captioning）数据集。其数据主要来源于复杂的日常场景（如图 3-3-1 所示），共包含超过 33 万张图像（其中 22 万张是有标注的图像），150 万个目标，80 个目标类别（Object Categories，例如行人、汽车、大象等），91 种类别（Stuff Categoris，例如，草、墙、天空等），每张图像包含 5 句图像的语句描述，且有 250000 个带关键点标注的行人。

本次实践采用的是 COCO2017 的目标检测数据，训练集 118287 张图，验证集 5000 张图，共计 123287 张图。如图 3-3-2 所示，训练图像和验证图像分别存储在 train2017 和 val2017 文件夹中，annotations 存储的是对应训练集和验证集的标注，其中 instances_train2017 和 instances_val2017 是需要的标注文件，其余的文件分别对应图像标题和人体关键点的标注（本次实践不需要）。

在本次实践中，并不需要 instances_train2017 中所有的标注信息。如图 3-3-3 所示，在

图 3-3-1　COCO 数据集示例

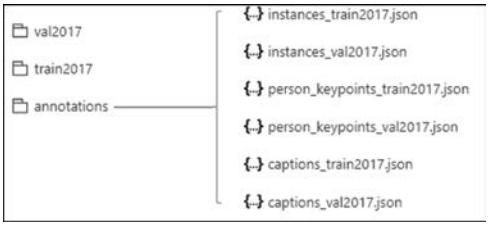


图 3-3-2　目录结构


```
"images": [
    {
        "license": 4,
        "file_name": "000000397133.jpg",
        "coco_url": "http://images.cocodataset.org/val2017/000000397133.jpg",
        "height": 427,
        "width": 640,
        "date_captured": "2013-11-14 17:02:52",
        "flickr_url": "http://farm7.staticflickr.com/6116/6255196340_da26cf2c9e_z.jpg",
        "id": 397133
    },
    {
        "license": 1,
        "file_name": "000000037777.jpg",
        "coco_url": "http://images.cocodataset.org/val2017/000000037777.jpg",
        "height": 230,
        "width": 352,
        "date_captured": "2013-11-14 20:55:31",
        "flickr_url": "http://farm9.staticflickr.com/8429/7839199426_f6d48aa585_z.jpg",
        "id": 37777
    },
    {
        "license": 4,
        "file_name": "000000252219.jpg",
        "coco_url": "http://images.cocodataset.org/val2017/000000252219.jpg",
        "height": 428,
        "width": 640,
        "date_captured": "2013-11-14 22:32:02",
        "flickr_url": "http://farm4.staticflickr.com/3446/3232237447_13d84bd0a1_z.jpg",
        "id": 252219
    },
    {
        "license": 1,
        "file_name": "000000087038.jpg",
        "coco_url": "http://images.cocodataset.org/val2017/000000087038.jpg",
        "height": 480,
        "width": 640,
        "date_captured": "2013-11-14 23:11:37",
        "flickr_url": "http://farm8.staticflickr.com/7355/8825114508_b0fa4d7168_z.jpg",
        "id": 87038
    },
    {
        "license": 6,
        "file_name": "000000174482.jpg",
        "coco_url": "http://images.cocodataset.org/val2017/000000174482.jpg",
        "height": 388,
        "width": 640,
        "date_captured": "2013-11-14 23:16:55",
        "flickr_url": "http://farm8.staticflickr.com/7020/6478877255_242f741dd1_z.jpg",
        "id": 174482
    },
```

```
{
    "segmentation": [
        [
            364.34,
            399.94,
            356.67,
            402.07,
            350.85,
            402.21,
            347.72,
            401.64,
            346.16,
            398.52,
            345.03,
            394.26,
            347.44,
            391.56,
            350.42,
            389.01,
            354.96,
            387.87,
            360.08,
            389.72,
            363.91,
            392.56,
            365.75,
            395.11
        ]
    ],
    "area": 219.91114999999954,
    "iscrowd": 0,
    "image_id": 535523,
    "bbox": [
        345.03,
        387.87,
        20.72,
        14.34
    ],
    "category_id": 60,
    "id": 1082722
},
```


图 3-3-3　标注示例

本次实践中需要用到图像信息中 file_name 记录的图片名称、height 和 width 记录的图像高和宽和标注文件中 category_id 记录的标注框对应类别、bbox 记录的标注框坐标以及 image_id 中记录的标注框所对应的图像 id。

步骤 2：数据加载

接下来，我们要实现网络训练过程中的数据加载部分。在网络训练中，数据加载部分除了需要提供图像和标注加载的功能外，还需要提供数据增强和分布式读取的功能，分别来实现这些功能。

本次实践使用的 COCO 数据集，标注通过 JSON 的格式存储。因此，要实现一个 COCODataSet 类。COCODataSet 类要完成：①解析标注文件，构建图像、标签数据；②对图像进行数据增强，并相应地对标注进行调整。

COCODataSet 类的主要函数包括 init、getitem 和 parse_dataset。接下来，分别针对这几个函数展开介绍。因为 COCODataSet 类是针对兼容 COCO 数据集多个任务设计的，所以代码中不仅考虑了检测的部分，也考虑了分割、关键点等任务。本节的内容将主要针对实践所涉及的目标检测部分。

init()函数用于在构建 COCODataSet 实例时进行初始化，确定数据存放的目录、图像路径、标注路径以及需要加载的数据内容(COCO 数据集除了包含检测标注外还包含分割、关键点等标注，因此我们需要给定需要加载的数据内容)。

```
def __init__(self,
             dataset_dir=None,
             image_dir=None,
             anno_path=None,
             data_fields=['image'],
             sample_num=-1,
             load_crowd=False,
             allow_empty=False,
             empty_ratio=1.,
    use_default_label=None):
    super(COCODataSet, self).__init__()
    self.dataset_dir = dataset_dir if dataset_dir is not None else ''
    self.anno_path = anno_path
    self.image_dir = image_dir if image_dir is not None else ''
    self.data_fields = data_fields
    self.sample_num = sample_num
    …… ……
```

parse_dataset()函数是 COCODataSet 中用于加载并解析所有标注信息的函数。在这里将分段介绍 parse_dataset()函数。

在 parse_dataset()函数中，需要使用 COCO 数据集提供的方法来读取标注文件中的一些信息。具体地，COCO 标注中的目标实例通过绑定图像 ID 来确定与图像的对应关系。因此，通过 getImgIds()函数来获取所有图像的 ID。除此之外，COCO 数据集 80 个类的编号并不是 0～79，所以需要构建顺序编号的类别 ID。这里就需要通过 getCatIds()函数来获取所有 COCO 数据集中的所有类别编号。

```
from pycocotools.coco import COCO
coco = COCO(anno_path)
img_ids = coco.getImgIds()
img_ids.sort()
cat_ids = coco.getCatIds()
records = []
self.catid2clsid = dict({catid: i for i, catid in enumerate(cat_ids)})
self.cname2cid = dict({
  coco.loadCats(catid)[0]['name']: clsid
  for catid, clsid in self.catid2clsid.items()
})
```

在获取完所有图像 ID 和对类别进行编码之后，就需要针对每个图象构建它的标注信息。在遍历图像内存在的标注之前，先构建针对图像的初始字典，其中包括图像的存储路径、图像的 ID 以及图像的长宽。

```
for img_id in img_ids:
    img_anno = coco.loadImgs([img_id])[0]      #加载存储图像信息的字典
im_fname = img_anno['file_name']
    im_w = float(img_anno['width'])
    im_h = float(img_anno['height'])
    im_path = os.path.join(image_dir,
                  im_fname) if image_dir else im_fname
    coco_rec = {
      'im_file': im_path,
      'im_id': np.array([img_id]),
      'h': im_h,
      'w': im_w,
    } if 'image' in self.data_fields else {}      #构建单张图像的字典
```

接下来，通过给定图像 ID，使用 coco.getAnnIds 读取与图像关联的所有目标的标注信息。一张图像往往存在多个目标实例，也就对应着多个目标标注信息。针对每个目标矩形框，将中心点坐标和框长宽转化为左上角和右下角点坐标后，添加进 bboxes 中。

```
if not self.load_image_only:
  ins_anno_ids = coco.getAnnIds(                #获取图像中包含的目标 ID
    imgIds=[img_id], iscrowd=None if self.load_crowd else False)
  instances = coco.loadAnns(ins_anno_ids)       #提取图像中的目标标注信息
  bboxes = []
  for inst in instances:                        #获得每一个实例目标
    x1, y1, box_w, box_h = inst['bbox']
    x2 = x1 + box_w
    y2 = y1 + box_h
    eps = 1e-5
    if inst['area'] > 0 and x2 - x1 > eps and y2 - y1 > eps:
      inst['clean_bbox'] = [
        round(float(x), 3) for x in [x1, y1, x2, y2]
      ]
      bboxes.append(inst)
```

对于图像中的实例，除了需要包裹目标实例矩形框的坐标外，还需要矩形框所对应的类别。因此针对一张图像所有的目标实例，我们构建存储目标矩形框坐标和标注的数组，并生

成单张图像的标注字典。

```
num_bbox = len(bboxes)
gt_bbox = np.zeros((num_bbox, 4), dtype = np.float32)
gt_theta = np.zeros((num_bbox, 1), dtype = np.int32)
gt_class = np.zeros((num_bbox, 1), dtype = np.int32)
is_crowd = np.zeros((num_bbox, 1), dtype = np.int32)
gt_poly = [None] * num_bbox
for i, box in enumerate(bboxes):
  catid = box['category_id']
  gt_class[i][0] = self.catid2clsid[catid]    #得到标注类别
  gt_bbox[i, :] = box['clean_bbox']           #得到标注 box 坐标
  # xc, yc, w, h, theta
gt_rec = {                                    #生成单张图像的标注信息(字典的形式)
  'is_crowd': is_crowd,                       #区分是单个实例还是一组对象
  'gt_class': gt_class,
  'gt_bbox': gt_bbox,
  'gt_poly': gt_poly,
}
```

最后,将图像的信息字典和图像内的标注字典合并在一起就完成了单张图像的所有信息加载。遍历所有训练的图像,就得到了训练过程中需要的信息。

```
    for k, v in gt_rec.items():
      if k in self.data_fields:
          coco_rec[k] = v
    records.append(coco_rec)
self.roidbs = records
```

getitem()函数用于在网络训练迭代的过程中提供训练所需要的数据。在 DETR 中返回的是图像、图像的宽高信息和图像中存在的目标矩形框的位置和类别。在 getitem 中还会随着迭代轮数的变化,根据设定的 Mixup 数据增强方式,对返回的图像和标注进行调整。

```
def __getitem__(self, idx):
  roidb = copy.deepcopy(self.roidbs[idx])
  if self.mixup_epoch == 0 or self._epoch < self.mixup_epoch:
    n = len(self.roidbs)
    idx = np.random.randint(n)
    roidb = [roidb, copy.deepcopy(self.roidbs[idx])]
  elif self.cutmix_epoch == 0 or self._epoch < self.cutmix_epoch:
    n = len(self.roidbs)
    idx = np.random.randint(n)
    roidb = [roidb, copy.deepcopy(self.roidbs[idx])]
  elif self.mosaic_epoch == 0 or self._epoch < self.mosaic_epoch:
    n = len(self.roidbs)
    roidb = [roidb, ] + [
      copy.deepcopy(self.roidbs[np.random.randint(n)])
      for _ in range(3)
    ]
  if isinstance(roidb, Sequence):
    for r in roidb:
      r['curr_iter'] = self._curr_iter
  else:
    roidb['curr_iter'] = self._curr_iter
```

```
    self._curr_iter += 1
    return self.transform(roidb)
```

我们可能会觉得奇怪，在 getitem()函数中，我们通过索引得到的是 parse_dataset()函数中生成的图像存储路径，并没有直接得到图像。其实，在 getitem()函数中是通过 transform(roidb)来实现图像的加载和数据增强的过程的。在 COCODataSet 类实例化后，我们会调用 set_transform 设置对图像和标注的加载以及增强方式。

```
def set_transform(self, transform):
    self.transform = transform
```

还要定义用于数据处理的各个类。目标检测网络对输入图片的格式、大小有一定的要求，数据灌入模型前，需要对数据进行预处理操作，使图片满足网络训练以及预测的需要。同时，为了使网络见过更多富有变化的数据，增强网络的泛化能力，还会进行一些数据增广。

本次实践中用到的数据预处理方法如下。

- 图像解码：将图像转为 Numpy 格式。
- 图像翻转：将图像进行翻转。
- 随机选择：在不同的预处理随机之间随机选择一个转换。
- 调整图片大小：将原图片中短边尺寸统一缩放到 384。
- 图像裁剪：将图像的长宽统一裁剪为 384×384，确保模型读入的图片数据大小统一。
- 归一化(Normalization)：通过规范化手段，把神经网络每层中任意神经元的输入值分布改变成均值为 0、方差为 1 的标准正态分布，使得最优解的寻优过程明显会变得平缓，训练过程更容易收敛。
- 通道变换：图像的数据格式为[H，W，C](高度、宽度和通道数)，而神经网络使用的训练数据的格式为[C，H，W]，因此需要对图像数据重新排列，例如[384，384，3]变为[3，384，384]。

在实现这些方法之前，要先定义一个预处理的父类 BaseOperator，其他的数据预处理类都要继承这个父类。在 BaseOperator 类中，通过__call__的 BaseOperator 类实例对象可以像调用普通函数那样，以“对象名()”的形式使用。其他类在继承 BaseOperator 类后，只需要重新 apply 方法即可。

```
class BaseOperator(object):
    def __init__(self, name=None):
        if name is None:
            name = self.__class__.__name__
        self._id = name + '_' + str(uuid.uuid4())[-6:]
    def apply(self, sample, context=None):
        return sample
    def __call__(self, sample, context=None):
        if isinstance(sample, Sequence):
            for i in range(len(sample)):
                sample[i] = self.apply(sample[i], context)
        else:
            sample = self.apply(sample, context)
        return sample
    def __str__(self):
        return str(self._id)
```

图像解码类 Decode：用于加载图像，并将图像转化为 Numpy 的格式。因为继承了 BaseOperator 的方法，在 Decode 类中只需要通过重写 apply()函数就可以实现图像的加载。具体地，使用 opencv 通过图像路径加载图形，确保加载的图像为 RGB 格式，并根据加载图像的长宽补充、修正通过 COCO 提供的 JSON 标注文件得到的图像长宽。最后，将图像转换为 Numpy 的格式。

```
class Decode(BaseOperator):
  def __init__(self):
    super(Decode, self).__init__()
  def apply(self, sample, context=None):
    """ load image if 'im_file' field is not empty but 'image' is"""
    im = sample['image']
    data = np.frombuffer(im, dtype='uint8')
    im = cv2.imdecode(data, 1) # BGR mode, but need RGB mode
    if 'keep_ori_im' in sample and sample['keep_ori_im']:
      sample['ori_image'] = im
    im = cv2.cvtColor(im, cv2.COLOR_BGR2RGB)
    sample['image'] = im
    if 'h' not in sample:
      sample['h'] = im.shape[0]
    elif sample['h'] != im.shape[0]:
      sample['h'] = im.shape[0]
    if 'w' not in sample:
      sample['w'] = im.shape[1]
    elif sample['w'] != im.shape[1]:
      sample['w'] = im.shape[1]
    sample['im_shape'] = np.array(im.shape[:2], dtype=np.float32)
    sample['scale_factor'] = np.array([1., 1.], dtype=np.float32)
    return sample
```

图像翻转类 RandomFlip 类：用于对图像进行反转。同时，对于标注的矩形框也要做出相应的调整，确保无论图像怎么变化，矩形框总能正确地包裹住目标。

```
class RandomFlip(BaseOperator):
  def __init__(self, prob=0.5):
    super(RandomFlip, self).__init__()
    self.prob = prob
  def apply_image(self, image):
    return image[:, ::-1, :]
  def apply_bbox(self, bbox, width):
    oldx1 = bbox[:, 0].copy()
    oldx2 = bbox[:, 2].copy()
    bbox[:, 0] = width - oldx2
    bbox[:, 2] = width - oldx1
    return bbox
  def apply(self, sample, context=None):
    if np.random.uniform(0, 1) < self.prob:
      im = sample['image']
      height, width = im.shape[:2]
      im = self.apply_image(im)
      if 'gt_bbox' in sample and len(sample['gt_bbox']) > 0:
        sample['gt_bbox'] = self.apply_bbox(sample['gt_bbox'], width)
```

```
        sample['flipped'] = True
        sample['image'] = im
      return sample
```

根据短边随机调整图像类 RandomShortSideResize：以图像的最短边为基础，随机放大或缩小短边的长度。在所缩放短边的同时，保证整个图像的长宽比例不变，进而缩放整个图像，从而实现图像分辨率随机调整。与图像反转相同，标注的矩形框也要做出相应的调整。其中，get_size_with_aspect_ratio()函数用于计算缩放因子，resize()函数根据缩放因子实现图像的调整，apply_bbox 则对应地调整标注的信息。

```
class RandomShortSideResize(BaseOperator):
  def __init__(self,
           short_side_sizes,
           max_size=None,
           interp=cv2.INTER_LINEAR,
           random_interp=False):
    super(RandomShortSideResize, self).__init__()
    ...

  def get_size_with_aspect_ratio(self, image_shape, size, max_size=None):
    ...
    return (ow, oh)
  def resize(self,
         sample,
         target_size,
         max_size=None,
         interp=cv2.INTER_LINEAR):
    ...
    return sample

  def apply_bbox(self, bbox, scale, size):
    im_scale_x, im_scale_y = scale
    resize_w, resize_h = size
    bbox[:, 0::2] *= im_scale_x
    bbox[:, 1::2] *= im_scale_y
    bbox[:, 0::2] = np.clip(bbox[:, 0::2], 0, resize_w)
    bbox[:, 1::2] = np.clip(bbox[:, 1::2], 0, resize_h)
    return bbox.astype('float32')

  def apply(self, sample, context=None):
    target_size = random.choice(self.short_side_sizes)
    interp = random.choice(
      self.interps) if self.random_interp else self.interp
    return self.resize(sample, target_size, self.max_size, interp)
```

随机裁剪类 RandomSizeCrop：根据给定的 min_size 和 max_size 对图像进裁剪。在 RandomSizeCrop 中通过 get_crop_params()函数获得裁剪过程中所必需的参数（裁剪位置、大小），并通过 crop 实现图像的裁剪。同样地，在对图像进行裁剪的时候，标注的矩形框也要通过 apply_bbox()函数做出相应地调整。

```
class RandomSizeCrop(BaseOperator):
  def __init__(self, min_size, max_size):
```

```
    super(RandomSizeCrop, self).__init__()
    self.min_size = min_size
    self.max_size = max_size
    from paddle.vision.transforms.functional import crop as paddle_crop
    self.paddle_crop = paddle_crop
  @staticmethod
  def get_crop_params(img_shape, output_size):
    h, w = img_shape
    th, tw = output_size
    if w == tw and h == th:
      return 0, 0, h, w
    i = random.randint(0, h - th + 1)
    j = random.randint(0, w - tw + 1)
    return i, j, th, tw

  def crop(self, sample, region):
    image_shape = sample['image'].shape[:2]
    sample['image'] = self.paddle_crop(sample['image'], *region)
    keep_index = None
    if 'gt_bbox' in sample and len(sample['gt_bbox']) > 0:
      sample['gt_bbox'] = self.apply_bbox(sample['gt_bbox'], region)
      bbox = sample['gt_bbox'].reshape([-1, 2, 2])
      area = (bbox[:, 1, :] - bbox[:, 0, :]).prod(axis=1)
      keep_index = np.where(area > 0)[0]
      sample['gt_bbox'] = sample['gt_bbox'][keep_index] if len(
        keep_index) > 0 else np.zeros(
          [0, 4], dtype=np.float32)
      sample['gt_class'] = sample['gt_class'][keep_index] if len(
        keep_index) > 0 else np.zeros(
          [0, 1], dtype=np.float32)
    return sample

  def apply_bbox(self, bbox, region):
    i, j, h, w = region
    region_size = np.asarray([w, h])
    crop_bbox = bbox - np.asarray([j, i, j, i])
    crop_bbox = np.minimum(crop_bbox.reshape([-1, 2, 2]), region_size)
    crop_bbox = crop_bbox.clip(min=0)
    return crop_bbox.reshape([-1, 4]).astype('float32')

  def apply(self, sample, context=None):
    h = random.randint(self.min_size,
              min(sample['image'].shape[0], self.max_size))
    w = random.randint(self.min_size,
              min(sample['image'].shape[1], self.max_size))
    region = self.get_crop_params(sample['image'].shape[:2], [h, w])
    return self.crop(sample, region)
```

随机选择预处理类 RandomSelect：从两种数据预处理方式组合中随机选择一种。可以根据输入 p 控制两种预处理组合的倾向性。

```
class RandomSelect(BaseOperator):
  def __init__(self, transforms1, transforms2, p=0.5):
```

```
    super(RandomSelect, self).__init__()
    self.transforms1 = Compose(transforms1)
    self.transforms2 = Compose(transforms2)
    self.p = p

  def apply(self, sample, context=None):
    if random.random() < self.p:
      return self.transforms1(sample)
    return self.transforms2(sample)
```

图像归一化类 NormalizeImage：对输入的图像进行归一化。NormalizeImage 类提供了两种选项：①将图像的像素值映射到 0 到 1；②每个像素点减去均值再除以方差。

```
class NormalizeImage(BaseOperator):
  def __init__(self, mean=[0.485, 0.456, 0.406], std=[1, 1, 1],
               is_scale=True):
    super(NormalizeImage, self).__init__()
    self.mean = mean
    self.std = std
    self.is_scale = is_scale
    from functools import reduce
    if reduce(lambda x, y: x * y, self.std) == 0:
      raise ValueError('{}: std is invalid!'.format(self))

  def apply(self, sample, context=None):
    im = sample['image']
    im = im.astype(np.float32, copy=False)
    mean = np.array(self.mean)[np.newaxis, np.newaxis, :]
    std = np.array(self.std)[np.newaxis, np.newaxis, :]
    if self.is_scale:
      im = im / 255.0
    im -= mean
    im /= std
    sample['image'] = im
    return sample
```

标注矩形框坐标归一化类 NormalizeBox：将标注矩形框的坐标归一化，由原来的绝对坐标位置转化为相对整个图像而言的相对坐标位置。简单来说，就是坐标点分别除以图像的长或宽。

```
class NormalizeBox(BaseOperator):

  def __init__(self):
    super(NormalizeBox, self).__init__()
  def apply(self, sample, context):
    im = sample['image']
    gt_bbox = sample['gt_bbox']
    height, width, _ = im.shape
    for i in range(gt_bbox.shape[0]):
      gt_bbox[i][0] = gt_bbox[i][0] / width
      gt_bbox[i][1] = gt_bbox[i][1] / height
      gt_bbox[i][2] = gt_bbox[i][2] / width
      gt_bbox[i][3] = gt_bbox[i][3] / height
```

```
    sample['gt_bbox'] = gt_bbox
    return sample
```

Permute 类：用于更改图片通道为(C,H,W)。图像在加载进来后通道为(H,W,C)并不满足网络的输入要求,因此需要通过 Permute 类调将其调整为(C,H,W)。

```
class Permute(BaseOperator):
  def __init__(self):
    super(Permute, self).__init__()

  def apply(self, sample, context=None):
    im = sample['image']
    im = im.transpose((2, 0, 1))
    sample['image'] = im
    return sample
```

在完成 COCODataSet 和各种预处理类后,我们就要实现最终的数据读取类 BaseDataLoader。BaseDataLoader 通过接收 COCODataSet 的实例,调用 COCODataSet 的方法和 paddle.io.DataLoader 实现 DETR 网络数据的加载,根据 batch 的设置为 DETR 批量地提供预处理后的图像和标注数据。其中,Compose、BatchCompose 类分别根据接收到预处理参数组合构建预处理的过程。

```
class BaseDataLoader(object):
  def __init__(self,
          sample_transforms=[],
          batch_transforms=[],
          batch_size=1,
          shuffle=False,
          drop_last=True,
          num_classes=80,
          collate_batch=True,
          use_shared_memory=False,
          **kwargs):
    self._sample_transforms = Compose(
      sample_transforms, num_classes=num_classes)
    self._batch_transforms = BatchCompose(batch_transforms, num_classes,
                            collate_batch)
    ...
  def __call__(self,
          dataset,
          worker_num,
          batch_sampler=None,
          return_list=False):
    self.dataset = dataset
    self.dataset.parse_dataset()
    self.dataset.set_transform(self._sample_transforms)
    self.dataset.set_kwargs(**self.kwargs)
    ...
    self.dataloader = DataLoader(
      dataset=self.dataset,
      batch_sampler=self._batch_sampler,
      collate_fn=self._batch_transforms,
```

```
            num_workers = worker_num,
            return_list = return_list,
            use_shared_memory = use_shared_memory)
        self.loader = iter(self.dataloader)
        return self
...
```

步骤 3：DETR 模型构建

在这部分，我们要开始构建 DETR 的模型。如图 3-3-4 所示，DETR 首先将一张三通道图片输入到 backbone 为 CNN 的网络中，提取图片特征；其次，把图像特征和位置信息结合后输入到 transformer 模型的编码器和解码器中；最后，通过预测网络得到最终的检测结果，每个结果就是一个 box，每个 box 表示一个元组，包含物体的类别和检测框位置。接下来将分别实现 DETR 的每个部分。

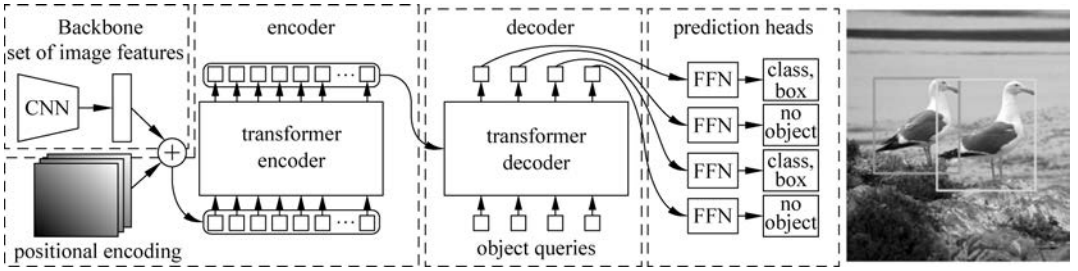


图 3-3-4　DETR 详细网络结构

（1）Resnet 特征提取网络。

在这里，我们使用 ResNet 作为 backbone 来提取图片的特征。ResNet 网络在 2015 年由微软实验室的何凯明等提出，斩获当年 ImageNet 竞赛中分类、目标检测任务的第一名、COCO 数据集中目标检测、图像分割的第一名。

在此过程中，我们主要使用如下接口进行网络结构的堆叠。

paddle. nn. Sequential(* **layers**)：顺序容器。子 Layer 将按构造函数参数的顺序添加到此容器中。传递给构造函数的参数可以是 Layers 或可迭代的 name Layer 元组。

- layers(tuple)：Layers 或可迭代的 name Layer 对。

通过定义一个继承了 paddle. nn. Layer 的 ResNet 类来实现 ResNet 网络。在 init()函数中我们需要输入搭建 ResNet 网络的一些必要参数。

```
class ResNet(nn.Layer):
  def __init__(self,
               depth = 50, #ResNet depth, should be 18, 34, 50, 101, 152.
               ch_in = 64, #output channel of first stage, default 64
               variant = 'b', #ResNet variant,
               lr_mult_list = [1.0, 1.0, 1.0, 1.0], #learning rate ratio
               groups = 1, # group convolution cardinality
               base_width = 64, # base width of each group convolution
               norm_type = 'bn', # normalization type
               norm_decay = 0, # weight decay
               freeze_norm = True, # freeze normalization layers
```

```
        freeze_at = 0, # freeze the backbone at which stage
        return_idx = [0, 1, 2, 3], # freeze the backbone
        dcn_v2_stages = [ - 1], # deformable conv v2
        num_stages = 4, # total num of stages
        std_senet = False):
    super(ResNet, self).__ init __()
    self._model_type = 'ResNet' if groups == 1 else 'ResNeXt'
    self.depth = depth
    self.variant = variant
    self.groups = groups
    self.base_width = base_width
    self.norm_type = norm_type
    self.norm_decay = norm_decay
    self.freeze_norm = freeze_norm
    self.freeze_at = freeze_at
    self.return_idx = return_idx
    self.num_stages = num_stages
    self.dcn_v2_stages = dcn_v2_stages
```

在 init()函数中，还需要完成 ResNet 各个网络层的搭建。通过调用 resnet.py 文件中的 NameAdapter 方法获取网络层的名称以匹配预训练权重。因为 Resnet 的第一个卷积层与后面的网络层都不相同，因此，要通过 ConvNormLayer()函数单独实现第卷积＋BN 层（步长为 2，大小为 7×7 的卷积核）。

```
na = NameAdapter(self)
conv1_name = na.fix_c1_stage_name()
conv_def = [[3, ch_in, 7, 2, conv1_name]]
self.conv1 = nn.Sequential()
for (c_in, c_out, k, s, _name) in conv_def:
  self.conv1.add_sublayer(
    _name,
    ConvNormLayer(
      ch_in = c_in,
      ch_out = c_out,
      filter_size = k,
      stride = s,
      groups = 1,
      act = 'relu',
      norm_type = norm_type,
      norm_decay = norm_decay,
      freeze_norm = freeze_norm,
      lr = 1.0))
```

ResNet 除第一层网络参数需要单独设定之外，其余的网络层都有着相似的结构。因此通过循环调用 Blocks()类，并赋予不同的参数来快速实现。最终网络层的结构被存储在 res_layers 中（具体可参考 2.4 节和 resnet.py 文件）。

```
ch_out_list = [64, 128, 256, 512]
    block = BottleNeck if depth >= 50 else BasicBlock
    self._out_channels = [block.expansion * v for v in ch_out_list]
    self._out_strides = [4, 8, 16, 32]
    self.res_layers = []
```

```
for i in range(num_stages):
  lr_mult = lr_mult_list[i]
  stage_num = i + 2
  res_name = "res{}".format(stage_num)
  res_layer = self.add_sublayer(
    res_name,
    Blocks(
      block,
      self.ch_in,
      ch_out_list[i],
      count=block_nums[i],
      name_adapter=na,
      stage_num=stage_num,
      variant=variant,
      groups=groups,
      base_width=base_width,
      lr=lr_mult,
      norm_type=norm_type,
      norm_decay=norm_decay,
      freeze_norm=freeze_norm,
      dcn_v2=(i in self.dcn_v2_stages),
      std_senet=std_senet))
  self.res_layers.append(res_layer)
  self.ch_in = self._out_channels[i]
```

ResNet 前向传播比较简单，只需要将我们第一层网络结构和 res_layers 的网络结构按顺序传播就可以（残差的结构在 Blocks 类中已经实现了）。

```
def forward(self, inputs):
  x = inputs['image']
  conv1 = self.conv1(x)
  x = F.max_pool2d(conv1, kernel_size=3, stride=2, padding=1)
  outs = []
  for idx, stage in enumerate(self.res_layers):
    x = stage(x)
    if idx in self.return_idx:
      outs.append(x)
  return outs
```

（2）位置编码。

在 DETR 中与 2.5 节相似地使用了位置编码（PositionEmbedding）。在 DETR 实践中通过 PositionEmbedding 类来实现 DETR 的位置编码。PositionEmbedding 提供了 sine 和 learned 两种位置编码方式。learned 是一种可学习的方法，即 emmbeding 向量从网络中学习；sine 方法对于特征图 $z_0 \in R^{d\times H\times W}$，构建相应的位置编码 $\mathrm{PE}\in R^{d\times H\times W}$，对于位置（$h$，$w$），前 $d/2$ 维表示 H 方向的位置编码，后 $d/2$ 维表示 W 方向的位置编码：

$$\mathrm{PE}_{(\mathrm{POS},2i)} = \sin\left(\frac{\mathrm{pos}}{\mathrm{temperature}^{2i/d}}\right)$$

$$\mathrm{PE}_{(\mathrm{POS},2i+1)} = \cos\left(\frac{\mathrm{pos}}{\mathrm{temperature}^{2i/d}}\right)$$

在 PositionEmbedding 类中，init()函数需要根据选定 sine()或者 learned 的方法进行设

置。如果是 sine()的方法，要给定特征的维度 d、sin()中的分母底数 temperature。而 learned 的方法则通过 paddle.nn.Embedding 来实现。

```
class PositionEmbedding(nn.Layer):
  def __init__(self,
          num_pos_feats=128,
          temperature=10000,
          normalize=True,
          scale=None,
          embed_type='sine',
          num_embeddings=50,
          offset=0.):
    super(PositionEmbedding, self).__init__()
    assert embed_type in ['sine', 'learned']
    self.embed_type = embed_type
    self.offset = offset
    self.eps = 1e-6
    if self.embed_type == 'sine':
      self.num_pos_feats = num_pos_feats
      self.temperature = temperature
      self.normalize = normalize
      if scale is not None and normalize is False:
        raise ValueError("normalize should be True if scale is passed")
      if scale is None:
        scale = 2 * math.pi
      self.scale = scale
    elif self.embed_type == 'learned':
      self.row_embed = nn.Embedding(num_embeddings, num_pos_feats)
      self.col_embed = nn.Embedding(num_embeddings, num_pos_feats)
```

在 forward()函数中，sine 的方法通过 cumsum 方法分别计算 W、H 两个方向上 pos 值的信息，再分别根据公式计算得到对应的矩阵；而 learned 方法则根据输入特征的 W、H，分别通过 init 中的 row_embed 和 col_embed 构建对应的矩阵。最后，得到的两个方向的矩阵经过连接和维度变化后就得到了位置编码。

```
def forward(self, mask):
  if self.embed_type == 'sine':
    mask = mask.astype('float32')
    y_embed = mask.cumsum(1, dtype='float32')
    x_embed = mask.cumsum(2, dtype='float32')
    if self.normalize:
      y_embed = (y_embed + self.offset) / (
        y_embed[:, -1:, :] + self.eps) * self.scale
      x_embed = (x_embed + self.offset) / (
        x_embed[:, :, -1:] + self.eps) * self.scale
    dim_t = 2 * (paddle.arange(self.num_pos_feats) //
           2).astype('float32')
    dim_t = self.temperature**(dim_t / self.num_pos_feats)
    pos_x = x_embed.unsqueeze(-1) / dim_t
    pos_y = y_embed.unsqueeze(-1) / dim_t
    pos_x = paddle.stack(
      (pos_x[:, :, :, 0::2].sin(), pos_x[:, :, :, 1::2].cos()),
```

```
            axis = 4).flatten(3)
        pos_y = paddle.stack(
            (pos_y[:, :, :, 0::2].sin(), pos_y[:, :, :, 1::2].cos()),
            axis = 4).flatten(3)
        pos = paddle.concat((pos_y, pos_x), axis = 3).transpose([0, 3, 1, 2])
        return pos
    elif self.embed_type == 'learned':
        h, w = mask.shape[ - 2:]
        i = paddle.arange(w)
        j = paddle.arange(h)
        x_emb = self.col_embed(i)
        y_emb = self.row_embed(j)
        pos = paddle.concat(
            [x_emb.unsqueeze(0).repeat(h, 1, 1),
             y_emb.unsqueeze(1).repeat(1, w, 1)],
            axis = - 1).transpose([2, 0, 1]).unsqueeze(0).tile(mask.shape[0],
                                                               1, 1, 1)
        return pos
```

（3）Transformer 结构。

接下来要完成 Transformer 的结构。如图 3-3-5 所示，Transformer 分为 Encoder 和 Decoder 两部分，具体实现过程如下。

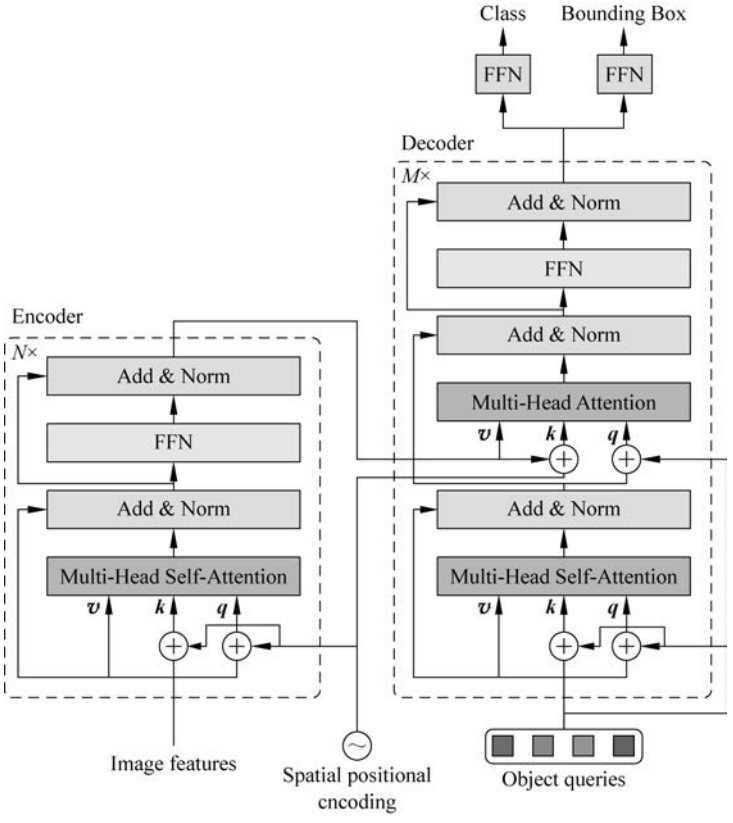


图 3-3-5　Transformer 结构示意图

在 Transformer 中，一个非常重要的部分是实现多头自注意力。因此，通过 MultiHeadAttention 类来实现多头自注意力的网络结构。在 init() 函数中我们需要给定输入以及输出特征的维度 embed_dim、多头自注意力机制中头的数目 num_heads 和是否使用 dropout 等，并生成 q、k、v 所需要的 Linear 层。

```
class MultiHeadAttention(nn.Layer):
  def __init__(self,
           embed_dim,
           num_heads,
           dropout = 0.,
           kdim = None,
           vdim = None,
           need_weights = False):
    super(MultiHeadAttention, self).__init__()
    self.embed_dim = embed_dim
    ...
    self.head_dim = embed_dim // num_heads
    if self._qkv_same_embed_dim:
      self.in_proj_weight = self.create_parameter(
          shape = [embed_dim, 3 * embed_dim],
          attr = None,
          dtype = self._dtype,
          is_bias = False)
      self.in_proj_bias = self.create_parameter(
        shape = [3 * embed_dim],
        attr = None,
        dtype = self._dtype,
        is_bias = True)
    else:
      self.q_proj = nn.Linear(embed_dim, embed_dim)
      self.k_proj = nn.Linear(self.kdim, embed_dim)
      self.v_proj = nn.Linear(self.vdim, embed_dim)
    self.out_proj = nn.Linear(embed_dim, embed_dim)
    self._type_list = ('q_proj', 'k_proj', 'v_proj')
    self._reset_parameters()
```

接下来，在前向传播 forward() 函数中实现自注意力的过程：q 乘以 k 的转置，在进行 scaling 和 softmax 后跟 v 做乘积就完成了 self.attention 的过程。

```
def forward(self, query, key = None, value = None, attn_mask = None):
  key = query if key is None else key
  value = query if value is None else value
  q, k, v = (self.compute_qkv(t, i)
         for i, t in enumerate([query, key, value]))
  product = paddle.matmul(x = q, y = k, transpose_y = True)
  scaling = float(self.head_dim) * * - 0.5
  product = product * scaling
  if attn_mask is not None:
     attn_mask = _convert_attention_mask(attn_mask, product.dtype)
     product = product + attn_mask
  weights = F.softmax(product)
  if self.dropout:
```

```
    weights = F.dropout(
      weights,
      self.dropout,
      training = self.training,
      mode = "upscale_in_train")
  out = paddle.matmul(weights, v)
  out = paddle.transpose(out, perm = [0, 2, 1, 3])
  out = paddle.reshape(x = out, shape = [0, 0, out.shape[2] * out.shape[3]])
  out = self.out_proj(out)
  outs = [out]
  if self.need_weights:
    outs.append(weights)
  return out if len(outs) == 1 else tuple(outs)
```

Encoder 部分如下。

Transformer 编码的过程由多个如图 3-3-6 所示的 Encoder 结构组成，其中包含多头自注意力、残差、归一化和前馈神经网络。接下来我们通过 TransformerEncoderLayer 来实现单个 Encoder。

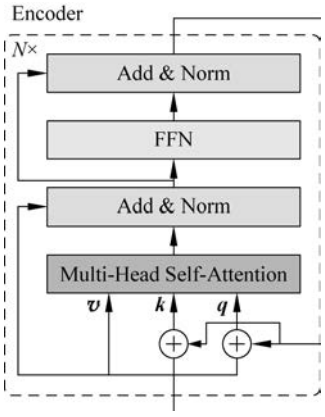


图 3-3-6　Encoder 结构示意图

在 TransformerEncoderLayer 的 init() 函数中，要实例化 Encoder 结构中所需要的各个网络层，主要包括多头自注意力层、FFN、归一化和 Dropout 层。

```
class TransformerEncoderLayer(nn.Layer):
  def __init__(self,
          d_model,
          nhead,
          dim_feedforward = 2048,
          dropout = 0.1,
          activation = "relu",
          attn_dropout = None,
          act_dropout = None,
          normalize_before = False):
    super(TransformerEncoderLayer, self).__init__()
    attn_dropout = dropout if attn_dropout is None else attn_dropout
    act_dropout = dropout if act_dropout is None else act_dropout
    self.normalize_before = normalize_before
    self.self_attn = MultiHeadAttention(d_model, nhead, attn_dropout)
    self.linear1 = nn.Linear(d_model, dim_feedforward)
    self.dropout = nn.Dropout(act_dropout, mode = "upscale_in_train")
    self.linear2 = nn.Linear(dim_feedforward, d_model)
    self.norm1 = nn.LayerNorm(d_model)
    self.norm2 = nn.LayerNorm(d_model)
    self.dropout1 = nn.Dropout(dropout, mode = "upscale_in_train")
    self.dropout2 = nn.Dropout(dropout, mode = "upscale_in_train")
    self.activation = getattr(F, activation)
    self._reset_parameters()
```

在 TransformerEncoderLayer 的前向过程中，按照图 3-3-6 所示流程进行。$\boldsymbol{q}$，$\boldsymbol{k}$ 由最初输入的 src 加上 pos 的位置编码构成，进入自注意力层后会对 $\boldsymbol{q}$，$\boldsymbol{k}$，$\boldsymbol{v}$ 进行 reshape。之后进行残差、FNN 等操作。而归一化则分为两种情况，一种情况是在输入多头自注意力层和

FFN 前进行归一化，另一种情况是在这两个层输出后再进行归一化。

```
def forward(self, src, src_mask = None, pos_embed = None):
  src_mask = _convert_attention_mask(src_mask, src.dtype)
  residual = src
  if self.normalize_before:
    src = self.norm1(src)
  q = k = self.with_pos_embed(src, pos_embed)
  src = self.self_attn(q, k, value = src, attn_mask = src_mask)
  src = residual + self.dropout1(src)
  if not self.normalize_before:
    src = self.norm1(src)
  residual = src
  if self.normalize_before:
    src = self.norm2(src)
  src = self.linear2(self.dropout(self.activation(self.linear1(src))))
  src = residual + self.dropout2(src)
  if not self.normalize_before:
    src = self.norm2(src)
  return src
```

通过 TransformerEncoderLayer 可以实现单个的 Encoder 结构。接下来，定义 TransformerDecoder 类来实现 Transformer 的整体的 Encoder 结构。Encoder 通常有 6 层，也就是上一层 Encoder 的输出作为下一层 Encoder 的输入，直到第 6 层最后输出 memory，这个 memory 将作为 Decoder 的输入（使用_get_clones()方法将其复制多次）。

```
class TransformerEncoder(nn.Layer):
  def __init__(self, encoder_layer, num_layers, norm = None):
    super(TransformerEncoder, self).__init__()
    self.layers = _get_clones(encoder_layer, num_layers)
    self.num_layers = num_layers
    self.norm = norm
  def forward(self, src, src_mask = None, pos_embed = None):
    src_mask = _convert_attention_mask(src_mask, src.dtype)
    output = src
    for layer in self.layers:
      output = layer(output, src_mask = src_mask, pos_embed = pos_embed)
    if self.norm is not None:
      output = self.norm(output)
    return output
```

Decoder 部分如下。

Transformer 的 Decoder 部分跟 Encoder 部分相似，由多个小的 Decoder 结构组成，但是在输入上存在差异。如图 3-3-7 所示，自注意力层的输出将作为多头注意力层中的 $\boldsymbol{q}$，而 $\boldsymbol{k}$ 和 $\boldsymbol{v}$ 则来自 Encoder 部分的输出，其中 $\boldsymbol{k}$ 还要加上位置编码。

我们通过 TransformerDecoderLayer 来实现单个的 Decoder 结构。在 init()函数中，需要实例化 Decoder 结构中的多头自注意力层、线性层、dropout 和激活函数。

```
class TransformerDecoderLayer(nn.Layer):
  def __init__(self,
          d_model,
          nhead,
```

```
        dim_feedforward = 2048,
        dropout = 0.1,
        activation = "relu",
        attn_dropout = None,
        act_dropout = None,
        normalize_before = False):
super(TransformerDecoderLayer, self).__ init __()
attn_dropout = dropout if attn_dropout is None else attn_dropout
act_dropout = dropout if act_dropout is None else act_dropout
self.normalize_before = normalize_before
self.self_attn = MultiHeadAttention(d_model, nhead, attn_dropout)
self.cross_attn = MultiHeadAttention(d_model, nhead, attn_dropout)
self.linear1 = nn.Linear(d_model, dim_feedforward)
self.dropout = nn.Dropout(act_dropout, mode = "upscale_in_train")
self.linear2 = nn.Linear(dim_feedforward, d_model)
self.norm1 = nn.LayerNorm(d_model)
self.norm2 = nn.LayerNorm(d_model)
self.norm3 = nn.LayerNorm(d_model)
self.dropout1 = nn.Dropout(dropout, mode = "upscale_in_train")
self.dropout2 = nn.Dropout(dropout, mode = "upscale_in_train")
self.dropout3 = nn.Dropout(dropout, mode = "upscale_in_train")
self.activation = getattr(F, activation)
self._reset_parameters()
```

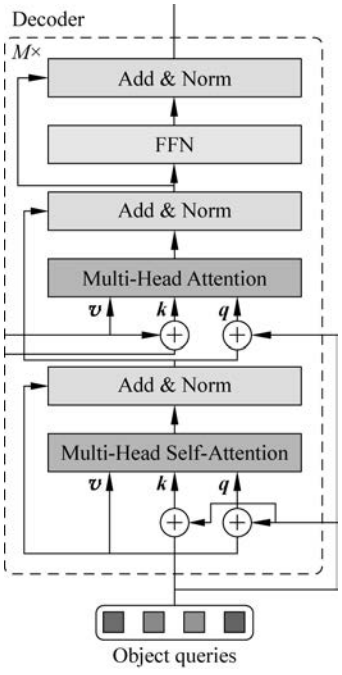


图 3-3-7　Decoder 结构示意图

TransformerDecoderLayer 前向传播的过程中，图 3-3-7 下半部分的 q，k 由 tgt 加上 query_pos(query 可以理解为对 anchor 的编码，并且这个 anchor 是一个全参数可学习的)的向量构成，且 $q=k$。经过自注意力层、残差和归一化后，加上 query_pos 作为上半部分多头自注意力层的 q。上半部分多头自注意力层的 k 和 v 则分别为 Encoder 部分的输出加上

query_pos 和 encoder 部分的输出。上半部分多头自注意力层的输出再经过残差、FNN 和归一化等操作后就得到了单个 Decoder 结构的输出。

```
def forward(self,
            tgt,
            memory,
            tgt_mask = None,
            memory_mask = None,
            pos_embed = None,
            query_pos_embed = None):
    tgt_mask = _convert_attention_mask(tgt_mask, tgt.dtype)
    memory_mask = _convert_attention_mask(memory_mask, memory.dtype)
    residual = tgt
    if self.normalize_before:
      tgt = self.norm1(tgt)
    q = k = self.with_pos_embed(tgt, query_pos_embed)
    tgt = self.self_attn(q, k, value = tgt, attn_mask = tgt_mask)
    tgt = residual + self.dropout1(tgt)
    if not self.normalize_before:
      tgt = self.norm1(tgt)
    residual = tgt
    if self.normalize_before:
      tgt = self.norm2(tgt)
    q = self.with_pos_embed(tgt, query_pos_embed)
    k = self.with_pos_embed(memory, pos_embed)
    tgt = self.cross_attn(q, k, value = memory, attn_mask = memory_mask)
    tgt = residual + self.dropout2(tgt)
    if not self.normalize_before:
      tgt = self.norm2(tgt)
    residual = tgt
    if self.normalize_before:
      tgt = self.norm3(tgt)
    tgt = self.linear2(self.dropout(self.activation(self.linear1(tgt))))
    tgt = residual + self.dropout3(tgt)
    if not self.normalize_before:
      tgt = self.norm3(tgt)
    return tgt
```

DETR 的解码器由多个 Decoder 模块组成，接下来通过 TransformerDecoder 类来实现 DETR 的整个 Decoder 过程，与 Encoder 部分相似，通过_get_clones 复制多个 Decoder 的结构，以前一个 Decoder 的输出作为下一个 Decoder 的输入。

```
class TransformerDecoder(nn.Layer):
  def __init__(self,
               decoder_layer,
               num_layers,
               norm = None,
               return_intermediate = False):
    super(TransformerDecoder, self).__init__()
    self.layers = _get_clones(decoder_layer, num_layers)
    self.num_layers = num_layers
    self.norm = norm
    self.return_intermediate = return_intermediate
```

```
def forward(self,
            tgt,
            memory,
            tgt_mask = None,
            memory_mask = None,
            pos_embed = None,
            query_pos_embed = None):
  tgt_mask = _convert_attention_mask(tgt_mask, tgt.dtype)
  memory_mask = _convert_attention_mask(memory_mask, memory.dtype)
  output = tgt
  intermediate = []
  for layer in self.layers:
    output = layer(
      output,
      memory,
      tgt_mask = tgt_mask,
      memory_mask = memory_mask,
      pos_embed = pos_embed,
        query_pos_embed = query_pos_embed)
    if self.return_intermediate:
      intermediate.append(self.norm(output))
  if self.norm is not None:
    output = self.norm(output)
  if self.return_intermediate:
    return paddle.stack(intermediate)
  return output.unsqueeze(0)
```

Transformer 部分如下。

完成 Transformer 的 Encoder 和 Decoder 之后，就可以实现 DETR 的 Transformer 整体结构。在 DETRTransformer 类的 init() 函数中，分别实例化 Transformer 中需要的 Encoder 结构、Decoder 结构、位置编码、query_pos 和用来降维的 1×1 卷积。

```
class DETRTransformer(nn.Layer):
  def __init__(self,
               num_queries = 100,
               position_embed_type = 'sine',
               return_intermediate_dec = True,
               backbone_num_channels = 2048,
               hidden_dim = 256,
               nhead = 8,
               num_encoder_layers = 6,
               num_decoder_layers = 6,
               dim_feedforward = 2048,
               dropout = 0.1,
               activation = "relu",
               attn_dropout = None,
               act_dropout = None,
               normalize_before = False):
    super(DETRTransformer, self).__init__()
    self.hidden_dim = hidden_dim
    self.nhead = nhead
    encoder_layer = TransformerEncoderLayer(
```

```
      hidden_dim, nhead, dim_feedforward, dropout, activation,
      attn_dropout, act_dropout, normalize_before)
    encoder_norm = nn.LayerNorm(hidden_dim) if normalize_before else None
    self.encoder = TransformerEncoder(encoder_layer, num_encoder_layers,
                         encoder_norm)
    decoder_layer = TransformerDecoderLayer(
      hidden_dim, nhead, dim_feedforward, dropout, activation,
      attn_dropout, act_dropout, normalize_before)
    decoder_norm = nn.LayerNorm(hidden_dim)
    self.decoder = TransformerDecoder(
      decoder_layer,
      num_decoder_layers,
      decoder_norm,
      return_intermediate=return_intermediate_dec)
    self.input_proj = nn.Conv2D(
      backbone_num_channels, hidden_dim, kernel_size=1)
    self.query_pos_embed = nn.Embedding(num_queries, hidden_dim)
    self.position_embedding = PositionEmbedding(
      hidden_dim // 2,
      normalize=True if position_embed_type == 'sine' else False,
      embed_type=position_embed_type)
```

Transformer 的前向过程，以 Resnet 的输出和 mask（由于在读取数据时对图像进行了随机的变化，再加上对图像进行随机裁剪，所以同一 batch 的数据尺寸存在差异，但是同一 batch 输入 resnet 的大小需要保持一致，就需要对图像进行 padding（全 0）操作以保证同一 batch 的尺寸相同。具体来说就是找到该 batch 下最大的 W 和最大的 H，然后 batch 下所有的图像根据这个最大的 $W \times H$ 进行 padding。因此还需要一个 mask 来记录 padding 前的原始图像在 padding 后的图像中的位置）为输入。首先，对 Resnet 的特征进行降维，并将维度由 $[B, C, H, W]$ 转化为 $[B, H \times W, C]$，然后根据 mask 进行位置编码。最终，图像、mask 和位置编码经过 Encoder 和 Decoder 之后就得到了最终的输出。

```
def forward(self, src, src_mask=None):
  src_proj = self.input_proj(src[-1])
  bs, c, h, w = src_proj.shape
  src_flatten = src_proj.flatten(2).transpose([0, 2, 1])
  if src_mask is not None:
    src_mask = F.interpolate(
      src_mask.unsqueeze(0).astype(src_flatten.dtype),
      size=(h, w))[0].astype('bool')
  else:
    src_mask = paddle.ones([bs, h, w], dtype='bool')
  pos_embed = self.position_embedding(src_mask).flatten(2).transpose(
    [0, 2, 1])
  src_mask = _convert_attention_mask(src_mask, src_flatten.dtype)
  src_mask = src_mask.reshape([bs, 1, 1, -1])
  memory = self.encoder(
    src_flatten, src_mask=src_mask, pos_embed=pos_embed)
  query_pos_embed = self.query_pos_embed.weight.unsqueeze(0).tile(
    [bs, 1, 1])
  tgt = paddle.zeros_like(query_pos_embed)
  output = self.decoder(
```

```
        tgt,
        memory,
        memory_mask = src_mask,
        pos_embed = pos_embed,
        query_pos_embed = query_pos_embed)
    return (output, memory.transpose([0, 2, 1]).reshape([bs, c, h, w]),
            src_proj, src_mask.reshape([bs, 1, 1, h, w]))
```

（4）匈牙利算法。

DETR 中不再设定 anchor，而是直接推断出一个包含 N 个结果的预测集合，其中 N 被设置为明显大于图像中物体数量的数值。而匈牙利算法就是用来匹配这些预测的结果和真实的标注（在这里就不对匈牙利算法展开介绍了）。

```
class HungarianMatcher(nn.Layer):
  def __init__(self,
               matcher_coeff = {'class': 1,'bbox': 5, 'giou': 2},
               use_focal_loss = False,alpha = 0.25,gamma = 2.0):
    super(HungarianMatcher, self).__init__()
...
  def forward(self, boxes, logits, gt_bbox, gt_class):
...
```

（5）DETRLOSS。

DETR 的 loss 由两部分组成：分类损失和边界框损失。其中，分类损失使用的是交叉熵损失，而边界框损失则由 L1 Loss（计算 x、y、W、H 的绝对值误差）和 GIoU Loss 组成。在这里通过 DETRLoss 类来实现 DETR 网络的损失部分。

```
class DETRLoss(nn.Layer):
  def __init__(self,
               num_classes = 80,
               matcher = 'HungarianMatcher',
               loss_coeff = {
                 'class': 1,
                 'bbox': 5,
                 'giou': 2,
                 'no_object': 0.1,
                 'mask': 1,
                 'dice': 1
               },
               aux_loss = True,
               use_focal_loss = False):
...
```

DETR 在计算损失时，首先将网络预测的结果和标注通过 HungarianMatcher 方法实现一一匹配，然后根据匹配的结果计算分类回归损失和边界框回归损失。

```
def forward(self,
            boxes,
            logits,
            gt_bbox,
            gt_class,
            masks = None,
```

```
        gt_mask = None):
    match_indices = self.matcher(boxes[ - 1].detach(), logits[ - 1].detach(),
                        gt_bbox, gt_class)
    num_gts = sum(len(a) for a in gt_bbox)
    ...
    total_loss = dict()
    total_loss.update(
       self._get_loss_class(logits[ - 1], gt_class, match_indices,
                    self.num_classes, num_gts))
    total_loss.update(
       self._get_loss_bbox(boxes[ - 1], gt_bbox, match_indices, num_gts))
    return total_loss
```

_get_loss_class 和 _get_loss_bbox 分别用于计算分类回归损失和边界框回归损失。其中，_get_loss_class 的交叉熵损失通过调用 paddle.nn.functional.cross_entropy 来实现。

```
  def _get_loss_class(self, logits, gt_class, match_indices, bg_index,
          num_gts):
target_label = paddle.full(logits.shape[:2], bg_index, dtype = 'int64')
bs, num_query_objects = target_label.shape
if sum(len(a) for a in gt_class) > 0:
   index, updates = self._get_index_updates(num_query_objects,
                            gt_class, match_indices)
   target_label = paddle.scatter(
      target_label.reshape([ - 1, 1]), index, updates.astype('int64'))
   target_label = target_label.reshape([bs, num_query_objects])
return {
 F.cross_entropy(
              logits, target_label, weight = self.loss_coeff['class'])         }
```

_get_loss_bbox 通过调用 paddle.nn.functional.l1_loss 和 GIoULoss 类来计算预测框和标注框之间的 L1 损失和 GIoU 损失。

```
def _get_loss_bbox(self, boxes, gt_bbox, match_indices, num_gts):
  loss = dict()
  if sum(len(a) for a in gt_bbox) == 0:
    loss['loss_bbox'] = paddle.to_tensor([0.])
    loss['loss_giou'] = paddle.to_tensor([0.])
    return loss
  src_bbox, target_bbox = self._get_src_target_assign(boxes, gt_bbox,
                                  match_indices)
  loss['loss_bbox'] = self.loss_coeff['bbox'] * F.l1_loss(
    src_bbox, target_bbox, reduction = 'sum') / num_gts
  loss['loss_giou'] = self.giou_loss(
    bbox_cxcywh_to_xyxy(src_bbox), bbox_cxcywh_to_xyxy(target_bbox))
  loss['loss_giou'] = loss['loss_giou'].sum() / num_gts
  loss['loss_giou'] = self.loss_coeff['giou'] * loss['loss_giou']
  return loss
```

(6) DETRHead。

DETR 的 Head 以 Transformer 中 Decoder 部分的输出为输入，通过 FFN 来实现最后的分类和边界框回归。

```
class DETRHead(nn.Layer):
  def __init__(self,
          num_classes=80,
          hidden_dim=256,
          nhead=8,
          num_mlp_layers=3,
          loss='DETRLoss',
          fpn_dims=[1024, 512, 256],
          with_mask_head=False,
          use_focal_loss=False):
    super(DETRHead, self).__init__()
    ...
    self.score_head = nn.Linear(hidden_dim, self.num_classes)
    self.bbox_head = MLP(hidden_dim,
                hidden_dim,
                output_dim=4,
                num_layers=num_mlp_layers)
    ...
  ...
  def forward(self, out_transformer, body_feats, inputs=None):
    feats, memory, src_proj, src_mask = out_transformer
    outputs_logit = self.score_head(feats)
    outputs_bbox = F.sigmoid(self.bbox_head(feats))
    outputs_seg = None
    if self.training:
      gt_mask = self.get_gt_mask_from_polygons(
        inputs['gt_poly'],
        inputs['pad_mask']) if 'gt_poly' in inputs else None
      return self.loss(
        outputs_bbox,
        outputs_logit,
        inputs['gt_bbox'],
        inputs['gt_class'],
        masks=outputs_seg,
        gt_mask=gt_mask)
    else:
      return (outputs_bbox[-1], outputs_logit[-1], outputs_seg)
```

（7）DETR。

前面分别定义了 DETR 的 backbone、transformer、DETRHead 和 DETR 的损失，它们共同组成了 DETR 模型。

```
class DETR(nn.Layer):
  def __init__(self,
          backbone,
          transformer,
          detr_head,
          post_process='DETRBBoxPostProcess',
          data_format='NCHW'):
    super(DETR, self).__init__()
    self.backbone = backbone
    self.transformer = transformer
    self.detr_head = detr_head
```

```
    self.post_process = post_process
    self.data_format = data_format
```

在前向传播的过程中，DETR 以图像和标注为输入，通过 Backbone 提取图像特征，并将提取的特征送入 transformer，最终通过 detr_head 返回最后的损失（预测阶段测返回对应的预测结果）。

```
def forward(self, inputs):
  if self.data_format == 'NHWC':
    image = inputs['image']
    inputs['image'] = paddle.transpose(image, [0, 2, 3, 1])
  self.inputs = inputs
  self.model_arch()
  if self.training:
    body_feats = self.backbone(self.inputs)
    out_transformer = self.transformer(body_feats, self.inputs['pad_mask'])
    losses = detr_head(out_transformer, body_feats, self.inputs)
    losses.update({'loss':paddle.add_n([v for k, v in losses.items() if 'log' not in k])
  })
  return loss
  else:
    body_feats = self.backbone(self.inputs)
    out_transformer = self.transformer(body_feats, self.inputs['pad_mask'])
           preds = self.detr_head(out_transformer, body_feats)
    bbox, bbox_num = self.post_process(preds, self.inputs['im_shape'],
                        self.inputs['scale_factor'])
    output = {"bbox": bbox_pred,"bbox_num": bbox_num}
    return output
```

步骤 4：DETR 训练

在 DETR 的训练阶段，首先要实例化前面定义的模型，用于提取图像特征的 ResNet、DETR 的 Transformer 结构，用于匹配预测结果和标志的 HungarianMatcher、损失 DETRLoss 和用于 DETR 的预测头部网络（DETRBBoxPostProcess 在预测阶段使用，用于对结果进行后处理，从而得到类别和检测框坐标），然后的实例化 model 就是我们要训练的 DETR 模型：

```
backbone = ResNet(depth = 50, norm_type = 'bn', freeze_at = 0, return_idx = [3],
                                lr_mult_list = [0.0, 0.1, 0.1, 0.1], num_stages = 4)
transformer = DETRTransformer(num_queries = 100, position_embed_type = 'sine',
            nhead = 8, num_encoder_layers = 6, num_decoder_layers = 6, dim_feedforward = 2048,
            dropout = 0.1, activation = 'relu', hidden_dim = 256, backbone_num_channels = 2048)
matcher = HungarianMatcher(matcher_coeff = {'class': 1, 'bbox': 5, 'giou': 2},
            use_focal_loss = False)
loss = DETRLoss(loss_coeff = {'class': 1, 'bbox': 5, 'giou': 2, 'no_object': 0.1,
            'mask': 1, 'dice': 1}, aux_loss = True, num_classes = 80, use_focal_loss = False,
            matcher = matcher)
detr_head = DETRHead(num_mlp_layers = 3, num_classes = 80, hidden_dim = 256, use
            _focal_loss = False, nhead = 8, fpn_dims = [], loss = loss)
post_process = DETRBBoxPostProcess(num_classes = 80, use_focal_loss = False)
model = DETR(backbone = backbone,
```

```
                transformer = transformer,
                detr_head = detr_head,
                post_process = post_process)
```

完成模型的实例化后，接下来要实现训练阶段所需的数据集、优化器、设置训练过程中的学习率、权重衰减。在 sample_transforms 和 batch_transforms 设置图像预处理和 batch 上预处理的操作。

```
def train(model, start_epoch, epoch):
...
  dataset = COCODataSet(dataset_dir = '/home/aistudio/dataset/', image_dir = 'train2017',anno_
path = 'annotations/instances_train2017.json',data_fields = ['image', 'gt_bbox', 'gt_class',
'is_crowd'])
  sample_transforms = [
                    {Decode: {}}, {RandomFlip: {'prob': 0.5}}, {RandomSelect: {'transforms1':
                    [{RandomShortSideResize: {'short_side_sizes': [480, 512, 544, 576, 608,
                    640, 672, 704, 736, 768, 800], 'max_size': 1333}}], 'transforms2':
                    [{RandomShortSideResize: {'short_side_sizes': [400, 500, 600]}},
                    {RandomSizeCrop: {'min_size': 384, 'max_size': 600}},
                    {RandomShortSideResize: {'short_side_sizes': [480, 512, 544, 576, 608,
                    640, 672, 704, 736, 768, 800], 'max_size': 1333}}]}}, {NormalizeImage:
                    {'is_scale': True, 'mean': [0.485, 0.456, 0.406], 'std': [0.229, 0.224,
                    0.225]}}, {NormalizeBox: {}}, {BboxXYXY2XYWH: {}}, {Permute: {}}]
  batch_transforms = [
                    {PadMaskBatch: {'pad_to_stride': 1, 'return_pad_mask': True}}]
  loader = BaseDataLoader(sample_transforms, batch_transforms, batch_size = 2,
                    shuffle = True, drop_last = True,collate_batch = False, use_shared_memory
                    = False)(dataset, 0)
  # build optimizer in train mode
  steps_per_epoch = len(loader)
  # 设置学习率、优化器
  schedulers = PiecewiseDecay(gamma = 0.1,milestones = [400],use_warmup = False)
  lr_ = LearningRate(base_lr = 0.0001, schedulers = schedulers)
  optimizer_ = OptimizerBuilder(clip_grad_by_norm = 0.1, regularizer = False, optimizers =
{'type': 'AdamW', 'weight_decay': 0.0001})
  lr = lr_(steps_per_epoch)
  optimizers = optimizer_(lr,model.parameters())
```

DETR 训练的过程与前面的实践相似，在每次迭代的过程中将加载数据输入 DETR 模型进行前向传播并计算损失，根据损失进行反向传播，执行一次优化器并进行参数更新、清空梯度，这样就完成了一次迭代训练。

```
for epoch_id in range(start_epoch, epoch):
  status['mode'] = 'train'
  status['epoch_id'] = epoch_id
  _compose_callback.on_epoch_begin(status)
  loader.dataset.set_epoch(epoch_id)
  model.train()
  iter_tic = time.time()
  for step_id, data in enumerate(loader):
    status['data_time'].update(time.time() - iter_tic)
    status['step_id'] = step_id
    _compose_callback.on_step_begin(status)
```

```
    outputs = model(data)
    loss = outputs['loss']
    loss.backward()
    optimizers.step()
    curr_lr = optimizers.get_lr()
    lr.step()
    optimizers.clear_grad()
    status['learning_rate'] = curr_lr
    if _nranks < 2 or _local_rank == 0:
      status['training_staus'].update(outputs)
    status['batch_time'].update(time.time() - iter_tic)
    _compose_callback.on_step_end(status)
    iter_tic = time.time()
```

步骤 5：DETR 的验证和预测

DETR 的验证阶段与训练过程相似，需要先实现用于验证的数据集，但不需要在对图像做增广的操作，只需要进行归一化等基础操作。同时，也不需要优化器和反向传播，每次迭代通过模型返回预测结果与标注计算精度即可。

```
def _eval_with_loader(model):
  status = {}
  _callbacks = [LogPrinter(model)]
  _compose_callback = ComposeCallback(_callbacks)
dataset = COCODataSet(dataset_dir = '/home/aistudio/dataset/', image_dir = 'val2017'
                      ,anno_path = 'annotations/instances_val2017.json')
  _eval_batch_sampler = paddle.io.BatchSampler(dataset, batch_size = 1)
  sample_transforms = [{Decode: {}}, {Resize: {'target_size': [800, 1333],
                      'keep_ratio': True}}, {NormalizeImage: {'is_scale': True, 'mean': [0.485,
                      0.456, 0.406], 'std': [0.229, 0.224, 0.225]}}, {Permute: {}}]
  batch_transforms = [{PadMaskBatch:{'pad_to_stride': -1,
                       'return_pad_mask': True}}]
  loader = BaseDataLoader(sample_transforms, batch_transforms, batch_size = 1,
                      shuffle = False, drop_last = False, drop_empty = False)(dataset,
                       4, _eval_batch_sampler)
  _metrics = _init_metrics(dataset = dataset)
  sample_num = 0
  tic = time.time()
  _compose_callback.on_epoch_begin(status)
  status['mode'] = 'eval'
  model.eval()
  for step_id, data in enumerate(loader):
    status['step_id'] = step_id
    _compose_callback.on_step_begin(status)
    outs = model(data)
    for metric in _metrics:
      metric.update(data, outs)
    sample_num += data['im_id'].numpy().shape[0]
    _compose_callback.on_step_end(status)
```

在 DETR 预测的过程中也要对预测的图像进行归一化、调整尺寸等基础的操作。图像在进行处置之后送入网络，对网络的输出进行处理就完成了网络预测的过程。至此，DETR 的网络实践就学习完了，快去动手实践吧！