基于RT-DETR_HGNetV2的实时垃圾桶目标检测

1.背景综述

1.1 论文简介

众所周知，实时目标检测(Real-Time Object Detection)在2022年一直被YOLO系列检测器统治着，YOLO版本更是炒到了v8。今年四月百度飞桨的PaddleDetection团队发布了一个名为 RT-DETR 的检测器，宣告其推翻了YOLO对实时检测领域统治。论文标题很直接：《DETRs Beat YOLOs on Real-time Object Detection》，直译就是 RT-DETR在实时目标检测中击败YOLO家族！

事实上，端到端目标检测模型（DETRs）一直具有相当良好的性能，然而，DETRs 的高计算成本限制了其实际应用，使其无法充分发挥无后处理（如非最大抑（NMS））的优势。

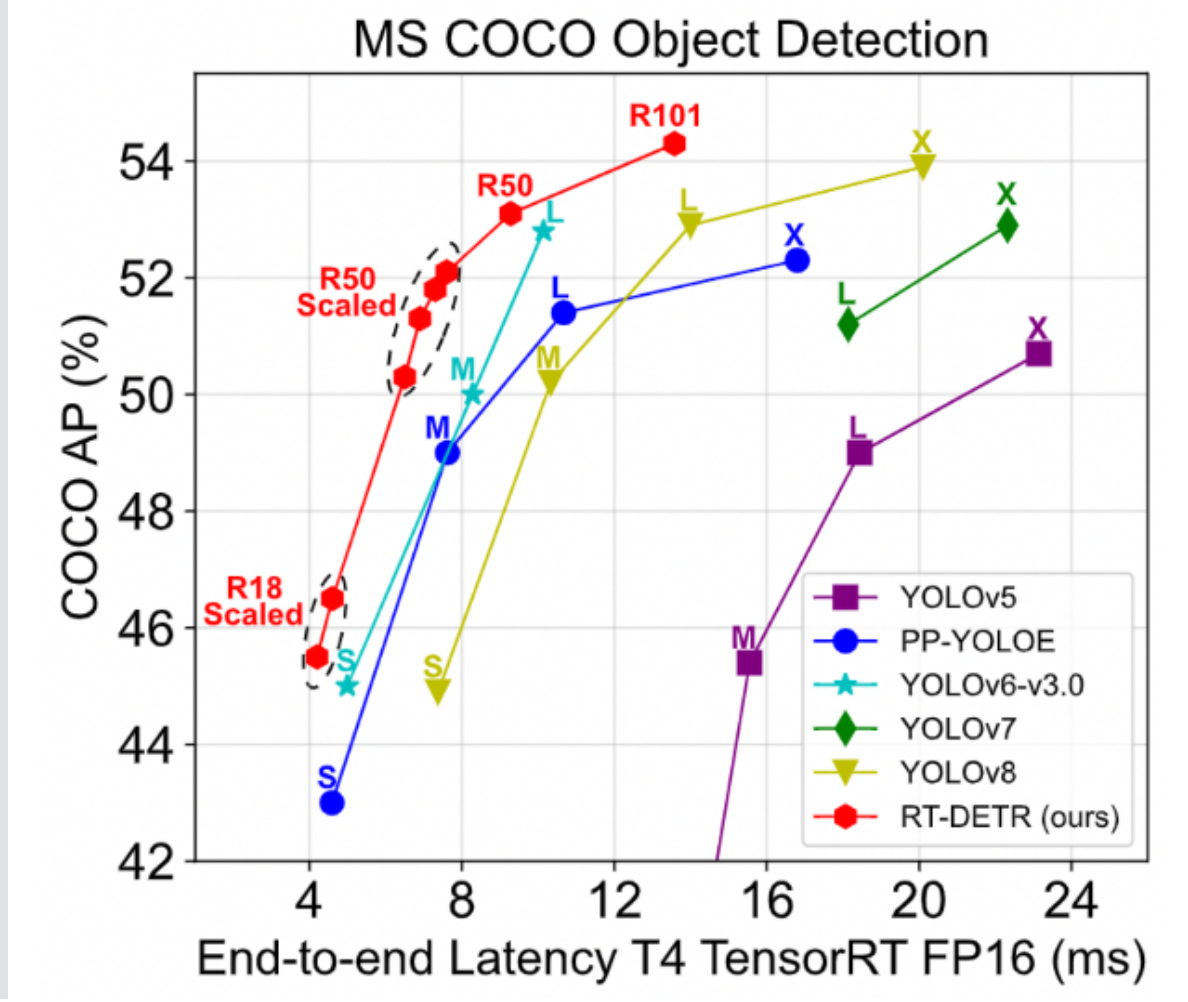
百度飞桨的PaddleDetection团队首先分析了 NMS 对现有实时目标检测器精度和速度的负面影响，并建立了端到端速度基准。为了解决上述问题，他们提出了一种实时目标检测转换器（RT-DETR），它是第一个实时端到端目标检测器。具体来说，他们设计了一种高效的混合编码器来高效处理多尺度特征。内交互和跨尺度融合，从而高效处理多尺度特征。感知 IoU 的查询选择，通过向解码器提供更高质量的初始对象查询，进一步提高性能。此外，他们提出的检测器还支持通过使用不同的解码器层来灵活调整推理速度，而不需要重新解码。并且他们提出的检测器还支持通过使用不同的解码器层灵活调整推理速度，而无需重新训练，这为在各种实时场景中的实际应用提供了便利。

据他们自己所言：

> *Our RT-DETR-L achieves* 53.0% *AP on COCO val2017 and* 114 *FPS on T4 GPU, while RT-DETR-X achieves* 54.8% *AP and* 74 *FPS, outperforming the state of-the-art YOLO detectors of the same scale in both speed and accuracy. Furthermore, our RT-DETR-R50 achieves 53.1% AP and* 108 *FPS, outperforming DINO Deformable DETR-R50 by* 2.2% *AP in accuracy and by about* 21 *times in FPS.*

DETRs在目标检测的进度与速度上都已经胜过了之前的所有的yolo模型

> RT-DETR 是 PaddleDetection 团队提出的第一个实时端到端目标检测模型，在速度和精度方面都优于相同规模的所有 YOLO 检测器，同时也在精度上超过了全部使用相同骨干网络的 DETR 检测器。下图展示了 RT-DETR 模型的测试效果：



接下来，我们运用这一模型，基于论文中提供的数据，对选择的模型进行优化，调参 ，完成对选择的所有模型的训练完成对垃圾桶状态的检测。尽力超过原有在各项指标上超过论文中给出的结果。

1.2 背景简介

1.2.1 实时对象检测器

经过多年的不断发展，YOLO 系列已成为实时物体检测器的代名词，可大致分为两类：基于锚的和无锚的。 从这些检测器的性能来看，锚不再是限制 YOLO 发展的主要因素。 然而，上述检测器产生了无数冗余的边界框，需要在后处理阶段使用 NMS 将其取出。遗憾的是，这导致YOLO系列出现了性能瓶颈，NMS 的参数过高对检测器的精度和速度产生了明显影响。

1.2.2 端对端的对象检测器

Carionetal.首先提出了基于Transformer的端到端目标检测器，命名为 DETR（DEtectionTRans-former）。 通过采用这种策略，DETR 简化了检测管线，并缓解了 NMS 带来的性能瓶颈。尽管对比YOLO系列有明显的优势，但 DETR 仍然存在两个主要问题：训练收敛速度慢和查询难以优化。 具体来说，Deformable-DETR通过提高注意力机制的效率，加快了具有多尺度特征的训练收敛速度；Con-conditional-DETR和Anchor-DETR降低了查询的优化难度；DAB-DETR引入了 4 个参考点，并逐层对预测框进行优化。 DN-DETR通过引入查询变异来加速训练收敛性。Group-DETR[6]通过引入组对多的分配来加速训练。DINO在前人成果的基础上，取得了最新成果。

1.2.3 用于物体检测的多尺度特征

FPN引入了一个融合相邻尺度特征的特征金字塔网络，随后的研究对这一结构进行了扩展和增强，并在实时物体检测器中得到了广泛应用。 Zhuetal.首先在 DETR 中引入了多尺度特征，并改善了性能和转换速度，但这导致了 DETR 计算成本的显著增加。虽然可变形注意机制在一定程度上降低了计算成本，但纳入多尺度特征仍然会导致较高的计算负担。 EffcientDETR通过将对象查询初始化为高密度前置，减少了编码器和解码器的层数；SparseDETR有选择地更新了编码器令牌中预计被解码器引用的内容，从而减少了计算开销。 LiteDETR通过以交错的方式降低流量级特征的更新频率，提高了编码器的效率。

2.RT-DETR_HGNetV2模型基础知识

结合PaddleDetection开源的代码（代码网址：https://github.com/PaddlePaddle/PaddleDetection/tree/develop/configs/rtdetr ） 来看，RT-DETR是基于先前DETR里精度最高的DINO检测模型进行修改得到的，但为了提高检测速度，RT-DETR针对实时检测做了很多方面的改进。作者团队正是先前PP-YOLOE和PP-YOLO论文的PaddleDetection，完全可以起名为PP-DETR，可能是为了突出RT这个实时性的意思吧。

2.1 结构

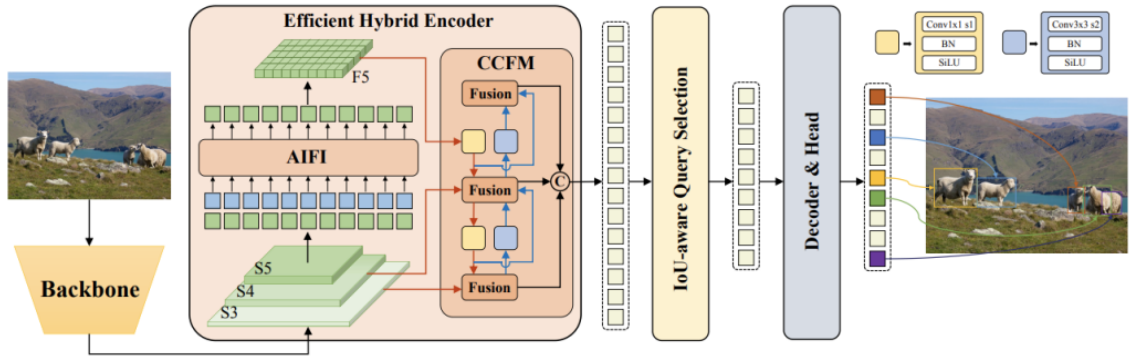


Figure 3: **Overview of RT-DETR.** We first leverage features of the last three stages of the backbone $\{S_3, S_4, S_5\}$ as the input to the encoder. The efficient hybrid encoder transforms multi-scale features into a sequence of image features through intra-scale feature interaction (AIFI) and cross-scale feature-fusion module (CCFM). The IoU-aware query selection is employed to select a fixed number of image features to serve as initial object queries for the decoder. Finally, the decoder with auxiliary prediction heads iteratively optimizes object queries to generate boxes and confidence scores.

RT-DETR模型结构

(1)Backbone层： 采用了经典的ResNet和百度自研的HGNet-v2两种，backbone是可以Scaled，应该就是常见的s， m， l， x分大中小几个版本，不过可能由于还要对比众多高精度的DETR系列所以只公布了HGNetv2的L和X两个版本，也分别对标经典的ResNet50和ResNet101，不同于DINO等DETR类检测器使用最后4个stage输出，RT-DETR为了提速只需要最后3个，这样也符合YOLO的风格；

(2) Neck层： 起名为HybridEncoder，其实是相当于DETR中的Encoder，其也类似于经典检测模型模型常用的FPN，论文里分析了Encoder计算量是比较冗余的，作者解耦了基于Transformer的这种全局特征编码，设计了AIFI (尺度内特征交互）和 CCFM(跨尺度特征融合）结合的新的高效混合编码器也就是Efficient Hybrid Encoder ，此外把encoder_layer层数由6减小到1层，并且由几个通道维度区分L和X两个版本，配合CCFM中RepBlock数量一起调节宽度深度实现Scaled RT-DETR；

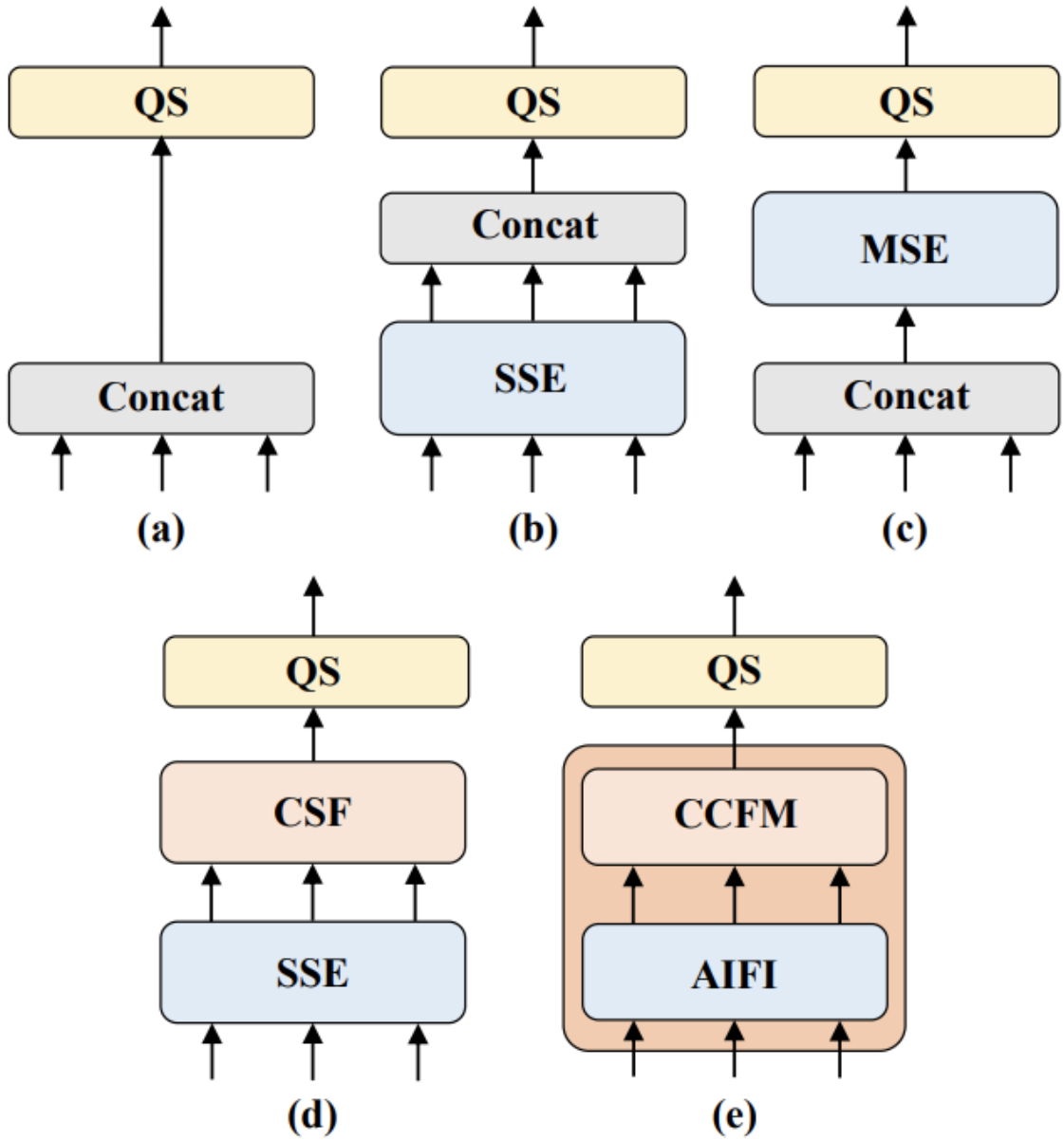


Figure 5: The set of variants with different types of encoders. **QS** represents the query selection, **SSE** represents the single-scale encoder, **MSE** represents the multi-scale encoder, and **CSF** represents cross-scale fusion.

HybridEncoder设计思路

(3)Transformer： 起名为RTDETRTransformer，基于DINO Transformer中的decoder改动的不多；

(4)Head和Loss： 和DINOHead基本一样，从RT-DETR的配置文件其实也可以看出neck+transformer+detr_head其实就是一整个Transformer，拆开写还是有点像YOLO类的风格。而训练加入了IoU-Aware的query selection，这个思路也是针对分类score和iou未必一致而设计的，改进后提供了更高质量（高分类分数和高IoU分数）的decoder特征；

(5)Reader和训练策略: Reader采用的是YOLO风格的简单640尺度，没有DETR类检测器复杂的多尺度resize，其实也就是原先他们PPYOLOE系列的reader，都是非常基础的数据增强，0均值1方差的NormalizeImage大概是为了节省部署时图片前处理的耗时，然后也没有用到别的YOLO惯用的mosaic等trick；训练策略和优化器，采用的是DETR类检测器常用的AdamW，毕竟模型主体还是DETR类的；

2.2 源码讲解

结合官方提供的RT-DETR_HGNetV2源码，其主要文件主要分为以下四个部分：
(1) rtdetr_reader.yml;
(2) rtdetr_r50vd.yml;
(3) optimizer_6x.yml;
(4) rtdetr_hgnetv2_l_6x_coco.yml
接下来我们将对其进行一一讲解

2.2.1 rtdetr_reader

rtdetr_reader是一个训练配置文件，用于指定训练数据读取器（TrainReader）、验证数据读取器（EvalReader）和测试数据读取器（TestReader）的配置。

```
worker_num: 4
```

这部分指定了数据读取时使用的工作线程数量为4。

```
TrainReader:
  sample_transforms:
    - Decode: {}
    - RandomDistort: {prob: 0.8}
    - RandomExpand: {fill_value: [123.675, 116.28, 103.53]}
    - RandomCrop: {prob: 0.8}
    - RandomFlip: {}
  batch_transforms:
    - BatchRandomResize: {target_size: [480, 512, 544, 576, 608, 640, 640, 640,
672, 704, 736, 768, 800], random_size: True, random_interp: True, keep_ratio:
False}
    - NormalizeImage: {mean: [0., 0., 0.], std: [1., 1., 1.], norm_type: none}
    - NormalizeBox: {}
    - BboxXYXY2XYWH: {}
    - Permute: {}
  batch_size: 4
  shuffle: true
  drop_last: true
  collate_batch: false
  use_shared_memory: true
```

这部分指定了训练数据读取器（TrainReader）的配置，包括样本变换（sample_transforms）和批次变换（batch_transforms）的操作序列，批次大小（batch_size）为4，是否打乱数据顺序（shuffle）为True，是否丢弃最后一个不完整的批次（drop_last）为True，是否在批次组装时进行拼接操作（collate_batch）为False，是否使用共享内存（use_shared_memory）为True。

样本变换的操作序列包括解码（Decode）、随机扭曲（RandomDistort，概率为0.8）、随机扩展（RandomExpand，使用填充值[123.675, 116.28, 103.53]）、随机裁剪（RandomCrop，概率为0.8）和随机翻转（RandomFlip）。

批次变换的操作序列包括批次随机调整大小（BatchRandomResize，目标大小为[480, 512, 544, 576, 608, 640, 640, 640, 672, 704, 736, 768, 800]，随机大小为True，随机插值方法为True，保持长宽比为False）、图像归一化（NormalizeImage，均值为[0., 0., 0.]，标准差为[1., 1., 1.]，归一化类型为none）、边界框归一化（NormalizeBox）、边界框坐标格式转换（BboxXYXY2XYWH）和维度置换（Permute）。

```
EvalReader:
  sample_transforms:
    - Decode: {}
    - Resize: {target_size: [640, 640], keep_ratio: False, interp: 2}
    - NormalizeImage: {mean: [0., 0., 0.], std: [1., 1., 1.], norm_type: none}
    - Permute: {}
  batch_size: 4
  shuffle: false
  drop_last: false
```

这部分指定了验证数据读取器（EvalReader）的配置，包括样本变换的操作序列，批次大小（batch_size）为4，是否打乱数据顺序（shuffle）为False，是否丢弃最后一个不完整的批次（drop_last）为False。

样本变换的操作序列包括解码（Decode）、调整大小（Resize，目标大小为[640, 640]，保持长宽比为False，插值方法为2）、图像归一化（NormalizeImage，均值为[0., 0., 0.]，标准差为[1., 1., 1.]，归一化类型为none）和维度置换（Permute）。

```
TestReader  inputs_def:
    image_shape: [3, 640, 640]
  sample_transforms:
    - Decode: {}
    - Resize: {target_size: [640, 640], keep_ratio: False, interp: 2}
    - NormalizeImage: {mean: [0., 0., 0.], std: [1., 1., 1.], norm_type: none}
    - Permute: {}
  batch_size: 1
  shuffle: false
  drop_last: false
```

这部分指定了测试数据读取器（TestReader）的配置，包括输入定义（image_shape为输入图像的形状，[3, 640, 640]表示通道数为3，高度和宽度为640）、样本变换的操作序列、批次大小（batch_size）为1，是否打乱数据顺序（shuffle）为False，是否丢弃最后一个不完整的批次（drop_last）为False。

样本变换的操作序列包括解码（Decode）、调整大小（Resize，目标大小为[640, 640]，保持长宽比为False，插值方法为2）、图像归一化（NormalizeImage，均值为[0., 0., 0.]，标准差为[1., 1., 1.]，归一化类型为none）和维度置换（Permute）。

这些配置用于定义数据读取器在训练、验证和测试过程中的数据处理流程，包括数据变换、批次大小、打乱顺序等操作，以及输入图像的形状定义。

2.2.2 rtdetr_r50vd

是一个模型配置文件，用于指定DETR（Detection Transformer）模型的结构、权重路径、训练参数等。

```
architecture: DETR
pretrain_weights:
https://paddledet.bj.bcebos.com/models/pretrained/ResNet50_vd_ssld_v2_pretrained.
pdparams
norm_type: sync_bn
use_ema: True
ema_decay: 0.9999
ema_decay_type: "exponential"
ema_filter_no_grad: True
hidden_dim: 256
use_focal_loss: True
eval_size: [640, 640]
```

这部分指定了DETR模型的一些基本配置，包括模型架构为DETR，预训练权重的下载链接，使用的归一化方法为同步批归一化（sync_bn），是否使用指数移动平均（EMA）为True，EMA的衰减率为0.9999，EMA的衰减类型为指数衰减（exponential），是否在EMA过程中过滤无梯度参数为True，隐藏维度为256，是否使用焦点损失（focal loss）为True，评估尺寸为[640, 640]。

```
DETR:
  backbone: ResNet
  neck: HybridEncoder
  transformer: RTDETRTransformer
  detr_head: DINOHead
  post_process: DETRPostProcess
```

这部分定义了DETR模型的各个组件，包括主干网络（backbone）、特征融合模块（neck）、Transformer模块（transformer）、DETR头部（detr_head）以及后处理模块（post_process）。

```
ResNet:
  depth: 50
  variant: d
  norm_type: bn
  freeze_at: 0
  return_idx: [1, 2, 3]
  lr_mult_list: [0.1, 0.1, 0.1, 0.1]
  num_stages: 4
  freeze_stem_only: True
```

这部分定义了ResNet主干网络的配置，包括网络的深度为50，变体为"d"，归一化方法为批归一化（bn），冻结的层数为0，要返回的特征层索引为[1, 2, 3]，各层的学习率倍率为[0.1, 0.1, 0.1, 0.1]，总共的阶段数为4，只冻结主干网络的起始部分。

```yaml
HybridEncoder:
  hidden_dim: 256
  use_encoder_idx: [2]
  num_encoder_layers: 1
  encoder_layer:
    name: TransformerLayer
    d_model: 256
    nhead: 8
    dim_feedforward: 1024
    dropout: 0.
    activation: 'gelu'
  expansion: 1.0
```

这部分定义了特征融合模块（HybridEncoder）的配置，包括隐藏维度为256，使用的编码器索引为[2]，编码器层数为1，每个编码器层使用的是TransformerLayer，其中d_model为256，nhead为8，dim_feedforward为1024，dropout为0.，激活函数为'gelu'，扩展因子为1.0。

```yaml
RTDETRTransformer:
  num_queries: 300
  position_embed_type: sine
  feat_strides: [8, 16, 32]
  num_levels: 3
  nhead: 8
  num_decoder_layers: 6
  dim_feedforward: 1024
  dropout: 0.0
  activation: relu
  num_denoising: 100
  label_noise_ratio: 0.5
  box_noise_scale: 1.0
  learnt_init_query: False
```

这部分定义了Transformer模块（RTDETRTransformer）的配置，包括查询数目为300，位置编码类型为正弦编码（sine），特征步长为[8, 16, 32]，金字塔层数为3，注意力头数为8，解码层数为6，前馈网络隐藏层维度为1024，dropout为0.0，激活函数为ReLU，去噪次数为100，标签噪声比例为0.5，边界框噪声缩放因子为1.0，是否学习初始查询向量为False。

```yaml
DINOHead:
  loss:
    name: DINOLoss
    loss_coeff: {class: 1, bbox: 5, giou: 2}
    aux_loss: True
    use_vfl: True
    matcher:
      name: HungarianMatcher
      matcher_coeff: {class: 2, bbox: 5, giou: 2}
```

这部分定义了DETR头部（DINOHead）的配置，包括损失函数为DINOLoss，损失系数为{class: 1, bbox: 5, giou: 2}，是否使用辅助损失为True，是否使用可变焦损失（VFL）为True，匹配器为HungarianMatcher，匹配器系数为{class: 2, bbox: 5, giou: 2}。

```
DETRPostProcess:
  num_top_queries: 300
```

这部分定义了后处理模块（DETRPostProcess）的配置，包括保留的顶部查询数目为300。

2.2.3 optimizer_6x

optimizer_6x是一个训练配置文件的一部分，类似于官方给出的训练格式示范文件，用于指定训练过程中的学习率和优化器的配置。

```
epoch: 72
```

这部分指定了训练的总轮数为72。

```
LearningRate:
  base_lr: 0.0001
  schedulers:
  - !PiecewiseDecay
    gamma: 1.0
    milestones: [100]
    use_warmup: true
  - !LinearWarmup
    start_factor: 0.001
    steps: 2000
```

这部分指定了学习率的配置，包括基础学习率（base_lr）为0.0001和学习率调度器（schedulers）的配置。

学习率调度器使用了两个策略，分别是PiecewiseDecay和LinearWarmup。PiecewiseDecay是分段衰减策略，其中gamma为1.0表示不进行衰减，milestones为[100]表示在第100个epoch时进行衰减。use_warmup为true表示使用预热策略。

LinearWarmup是线性预热策略，其中start_factor为0.001表示初始学习率为base_lr的0.001倍，steps为2000表示进行2000个步骤进行线性预热。

```
OptimizerBuilder:
  clip_grad_by_norm: 0.1
  regularizer: false
  optimizer:
    type: AdamW
    weight_decay: 0.0001
```

这部分指定了优化器的配置，包括梯度裁剪（clip_grad_by_norm）为0.1，正则化（regularizer）为false。

优化器类型为AdamW，weight_decay为0.0001表示使用权重衰减（L2正则化）的权重衰减系数为0.0001。

这些配置用于定义训练过程中的学习率和优化器的设置，包括学习率的衰减策略、预热策略，以及优化器的类型、权重衰减等设置。

2.2.4 rtdetr_hgnetv2_l_6x_coco

rtdetr_hgnetv2_l_6x_coco是一个模型配置文件，它包含了以下设置：

```
_BASE_: [
  '../datasets/coco_detection.yml',
  '../runtime.yml',
  '_base_/optimizer_6x.yml',
  '_base_/rtdetr_r50vd.yml',
  '_base_/rtdetr_reader.yml',
]
```

这部分定义了模型配置的基础设置，包括数据集、运行时配置、优化器设置、模型结构以及读取器设置等。通过引用其他配置文件路径，将这些配置文件中定义的设置合并到当前配置中。

```
weights: output/rtdetr_hgnetv2_l_6x_coco/model_final
```

这行代码指定了模型的权重路径，表示要加载训练好的模型权重。

```
pretrain_weights:
https://bj.bcebos.com/v1/paddledet/models/pretrained/PPHGNetV2_L_ssld_pretrained.
pdparams
```

这行代码指定了预训练权重的下载链接，如果模型权重不存在，则会从指定的链接下载预训练权重。

```
find_unused_parameters: True
```

这行代码设置为True，表示在多GPU训练时允许未使用的参数。

```
log_iter: 200
```

这行代码设置了日志输出的间隔，每训练200个迭代输出一次日志信息。

```
DETR:
  backbone: PPHGNetV2
```

这部分指定了DETR模型的设置，其中`backbone`指定了使用的主干网络结构为PPHGNetV2。

```
PPHGNetV2:
  arch: 'L'
  return_idx: [1, 2, 3]
  freeze_stem_only: True
  freeze_at: 0
  freeze_norm: True
  lr_mult_list: [0., 0.05, 0.05, 0.05, 0.05]
```

这部分定义了PPHGNetV2主干网络的具体配置。`arch`指定了使用的网络版本为L，`return_idx`指定了主干网络的返回层索引，`freeze_stem_only`设置为True表示只冻结主干网络的起始部分，`freeze_at`指定了冻结的层数，`freeze_norm`设置为True表示冻结BatchNorm层的参数，`lr_mult_list`指定了各个层的学习率倍率。

2.3 精度对比

（1）对比YOLO系列：

同级别下RT-DETR比所有的YOLO都更高，而且这还只是RT-DETR训练72个epoch的结果，先前精度最高的YOLOv8都是需要训500个epoch的，其他YOLO也基本都需要训300epoch，这个训练时间成本就不在一个级别了，对于训练资源有限的用户或项目是非常友好的。之前各大YOLO模型在COCO数据集上，同级别的L版本都还没有突破53 mAP的，X版本也没有突破54 mAP的，唯一例外的YOLO还是RT-DETR团队他们先前搞的PP-YOLOE+，借助objects365预训练只80epoch X版本就刷到了54.7 mAP，而蒸馏后的L版本更刷到了54.0 mAP，实在是太强了。此外RT-DETR的参数量FLOPs上也非常可观，换用HGNetv2后更优，虽然模型结构类似DETR但设计的这么轻巧还是很厉害了。

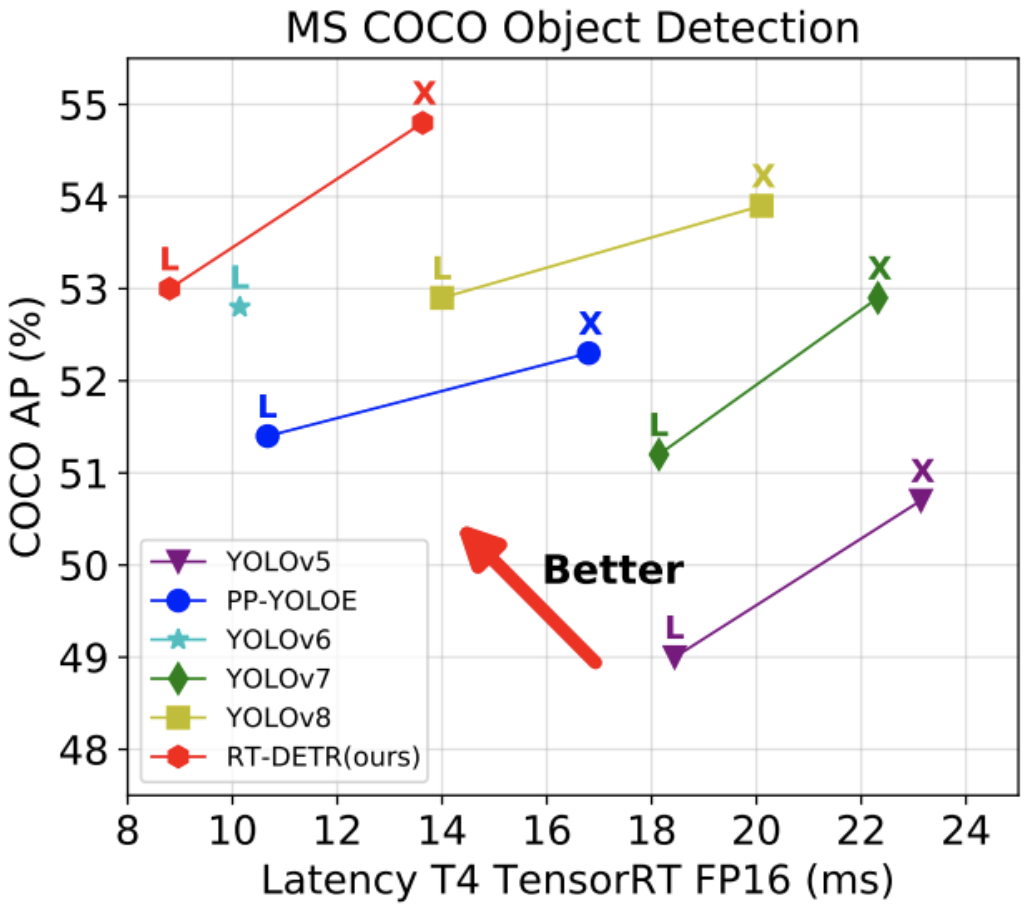


Figure 1: Compared to other real-time object detectors, our proposed detector achieves state-of-the-art performance in both speed and accuracy.

和YOLO对比

(2) 对比DETR系列：

DETR类在COCO上常用的尺度都是800x1333，以往都是以Res50 backbone刷上45 mAP甚至50 mAP为目标，而RT-DETR在采用了YOLO风格的640x640尺度情况下，也不需要熬时长训几百个epoch 就能轻松突破50mAP，精度也远高于所有DETR类模型。此外值得注意的是，RT-DETR只需要300个queries，设置更大比如像DINO的900个肯定还会更高，但是应该会变慢很多意义不大。

Model	Backbone	#Epochs	#Params (M)	GFLOPs	$FPS_{bs=1}$	AP^{val}	AP^{val}_{50}	AP^{val}_{75}	AP^{val}_{S}	AP^{val}_{M}	AP^{val}_{L}
Real-time Object Detectors											
YOLOv5-L[10]	-	300	46	109	54	49.0	67.3	-	-	-	-
YOLOv5-X[10]	-	300	86	205	43	50.7	68.9	-	-	-	-
PPYOLOE-L[36]	CSPRepResNet	300	52	110	94	51.4	68.9	55.6	31.4	55.3	66.1
PPYOLOE-X[36]	CSPRepResNet	300	98	206	60	52.3	69.9	56.5	33.3	56.3	66.4
YOLOv6-L[14]	-	300	59	150	99	52.8	70.3	57.7	34.4	58.1	70.1
YOLOv7-L[33]	-	300	36	104	55	51.2	69.7	55.5	35.2	55.9	66.7
YOLOv7-X[33]	-	300	71	189	45	52.9	71.1	57.4	36.9	57.7	68.6
YOLOv8-L[11]	-	-	43	165	71	52.9	69.8	57.5	35.3	58.3	69.8
YOLOv8-X[11]	-	-	68	257	50	53.9	71.0	58.7	35.7	59.3	70.7
End-to-end Object Detectors											
DETR-DC5 [4]	R50	500	41	187	-	43.3	63.1	45.9	22.5	47.3	61.1
DETR-DC5 [4]	R101	500	60	253	-	44.9	64.7	47.7	23.7	49.5	62.3
Anchor-DETR-DC5 [35]	R50	50	39	172	-	44.2	64.7	47.5	24.7	48.2	60.6
Anchor-DETR-DC5 [35]	R101	50	-	-	-	45.1	65.7	48.8	25.8	49.4	61.6
Conditional-DETR-DC5 [23]	R50	108	44	195	-	45.1	65.4	48.5	25.3	49.0	62.2
Conditional-DETR-DC5 [23]	R101	108	63	262	-	45.9	66.8	49.5	27.2	50.3	63.3
Efficient-DETR [37]	R50	36	35	210	-	45.1	63.1	49.1	28.3	48.4	59.0
Efficient-DETR [37]	R101	36	54	289	-	45.7	64.1	49.5	28.2	49.1	60.2
SMCA-DETR [6]	R50	108	40	152	-	45.6	65.5	49.1	25.9	49.3	62.6
SMCA-DETR [6]	R101	108	58	218	-	46.3	66.6	50.2	27.2	50.5	63.2
Deformable-DETR [43]	R50	50	40	173	-	46.2	65.2	50.0	28.8	49.2	61.7
DAB-Deformable-DETR [20]	R50	50	48	195	-	46.9	66.0	50.8	30.1	50.4	62.5
DN-Deformable-DETR [16]	R50	50	48	195	-	48.6	67.4	52.7	31.0	52.0	63.7
DAB-Deformable-DETR++ [16]	R50	50	47	-	-	48.7	67.2	53.0	31.4	51.6	63.9
DN-Deformable-DETR++ [16]	R50	50	47	-	-	49.5	67.6	53.8	31.3	52.6	65.4
DINO-Deformable-DETR [40]	R50	36	47	279	5	50.9	69.0	55.3	34.6	54.1	64.6
Real-time End-to-end Object Detector (ours)											
RT-DETR-R50	R50	72	42	136	**108**	**53.1**	**71.3**	**57.7**	34.8	58.0	70.0
RT-DETR-R101	R101	72	76	259	**74**	**54.3**	**72.7**	58.6	36.0	58.8	**72.1**
RT-DETR-L	HGNetv2	72	32	110	**114**	**53.0**	**71.6**	57.3	34.6	57.3	**71.2**
RT-DETR-X	HGNetv2	72	67	234	**74**	**54.8**	**73.1**	**59.4**	35.7	**59.6**	**72.9**

Table 2: Main results. Real-time detectors and our RT-DETR share a common input size of 640, and end-to-end detectors use an input size of (800, 1333). The end-to-end speed results are reported on T4 GPU with TensorRT FP16 using official pre-trained models followed the method proposed in Sec. 3. (**Note:** We do not test the speed of DETRs, except for DINO-Deformable-DERT for comparison, as they are not real time detectors.)

论文精度速度表格

2.4 速度对比

来看下RT-DETR和各大YOLO和DETR的速度对比：

（1）对比YOLO系列：

首先纯模型也就是去NMS后的速度上，RT-DETR由于轻巧的设计也已经快于大部分YOLO，然后实际端到端应用的时候还是得需要加上NMS的...嗯等等，DETR类检测器压根就不需要NMS，所以一旦端到端使用，RT-DETR依然轻装上阵一路狂奔，而YOLO系列就需要带上NMS负重前行了，NMS参数设置的不好比如为了拉高recall就会严重拖慢YOLO系列端到端的整体速度。

（2）对比DETR系列：

其实结论是显而易见的，以往DETR几乎都是遵循着800x1333尺度去训和测，这个速度肯定会比640尺度慢很多，即使换到640尺度训和测，精度首先肯定会更低的多，而其原生设计庞大的参数量计算量也注定了会慢于轻巧设计的RT-DETR。RT-DETR的轻巧快速是encoder高效设计、通道数、encoder数、query数等方面全方位改良过的。

3 参数调节

3.1 参数说明

参数名称	数值样例	说明
轮次（Epochs）	40	可修改，模型训练的轮次，不同模型的训练轮次单位有所差异(Epochs or Iters)。训练轮次越大，训练耗时越长，最后的精度也往往较高。

参数名称	数值样例	说明
批大小（Batch Size）	8	可修改，模型训练批处理大小（单卡），**实际的总Batch Size等于该数值乘以GPU数量。**Batch Size越大，显存占用越高，训练耗时也越短（轮次单位为Epochs时）。如果您遇到显存不足的问题，可以尝试使用较小的Batch Size。
学习率（Learning Rate）	0.0001	可修改，模型优化器的学习率。**如果减小Batch Size，一般需要等比例减小学习率。**
选择设备	GPU/CPU	可选GPU或者CPU作为训练设备，**强烈建议使用GPU进行训练，CPU训练耗时很长且不稳定。**
设备编号	0 1 2 3	当【设备选择】为GPU时，可选设备编号，**使用空格分隔。**
热启动步数（WarmUp Steps）	100	在训练初始阶段以较小学习率训练的轮次，有助于训练稳定收敛。
log 打印间隔(Log Interval) / step	10	log信息打印间隔，决定了在模型训练时底部运行日志的刷新频率。
评估、保存间隔(Eval Interval) / epoch	5	评估和报错模型的间隔。在训练时，每隔若干个epoch将会进行一次模型评估（基于验证集）和权重文件的保存（便于断点训练）。**该间隔不宜过小，否则保存大量权重文件会占据大量存储空间，甚至超出存储限制。**
输出目录	./output	可修改，模型训练过程中，日志文件和模型文件存储路径。**训练过程会在该目录下输出train.log，记录完整的训练日志。**
开始训练	-	可点击，点击后开始训练。
断点训练权重路径	-	可修改，如果之前有训练模型，想在之前中断的基础上继续训练，可以在这里填入模型checkpoint的路径（.pdparams文件），然后点击【开始训练】即可继续进行训练。
预训练权重路径	-	可修改，如果您想使用相关的预训练模型进行微调，则可以把预训练模型路径（.pdparams文件）放在这里。
混合精度训练模式（规划中，敬请期待）	O1	可修改，是否开启混合精度训练模式及其开启后的级别。
是否动转静训练（规划中，敬请期待）	否	可修改，是否开启混合精度训练。

3.2 调节参数

在调节参数过程中，我分别进行了如下尝试：

- epoch为14，其余均保持默认值，得到结果为map = 0.77，此时已经高于论文中所给出的0.764
- epoch为25，学习率改为0.001，得到结果为map = 0.223，猜测学习率过高导致过拟合
- epoch为50，其余均保持默认值，得到结果如下

> Average Precision (AP) @[IoU=0.50:0.95 | area= all | maxDets=100] = 0.787
> Average Precision (AP) @[IoU=0.50 | area= all | maxDets=100] = 0.891
> Average Precision (AP) @[IoU=0.75 | area= all | maxDets=100] = 0.850
> Average Precision (AP) @[IoU=0.50:0.95 | area= small | maxDets=100] = 0.590
> Average Precision (AP) @[IoU=0.50:0.95 | area=medium | maxDets=100] = 0.730
> Average Precision (AP) @[IoU=0.50:0.95 | area= large | maxDets=100] = 0.834
> Average Recall (AR) @[IoU=0.50:0.95 | area= all | maxDets= 1] = 0.526
> Average Recall (AR) @[IoU=0.50:0.95 | area= all | maxDets= 10] = 0.863
> Average Recall (AR) @[IoU=0.50:0.95 | area= all | maxDets=100] = 0.879
> Average Recall (AR) @[IoU=0.50:0.95 | area= small | maxDets=100] = 0.750
> Average Recall (AR) @[IoU=0.50:0.95 | area=medium | maxDets=100] = 0.835
> Average Recall (AR) @[IoU=0.50:0.95 | area= large | maxDets=100] = 0.912
> [11/21 16:05:17] ppdet.engine INFO: Total sample number: 1608, average FPS: 40.529900250500354

显然已经优于论文所用模型

- 参考论文以后，我发现该模型epoch达到72时已经能够得到较好的效果，因此我将epoch设置为72，其余均保持默认值，然而训练结果查看日志发现，在第62轮时达到0.817，训练完成后疑似过拟合，map降至0.763，同时对第62轮训练得到的权重文件进行评估，发现map降至0.756

> **Average Precision (AP) @[IoU=0.50:0.95 | area= all | maxDets=100] = 0.756**
> **Average Precision (AP) @[IoU=0.50 | area= all | maxDets=100] = 0.832**
> **Average Precision (AP) @[IoU=0.75 | area= all | maxDets=100] = 0.807**
> **Average Precision (AP) @[IoU=0.50:0.95 | area= small | maxDets=100] = 0.629**
> **Average Precision (AP) @[IoU=0.50:0.95 | area=medium | maxDets=100] = 0.720**
> **Average Precision (AP) @[IoU=0.50:0.95 | area= large | maxDets=100] = 0.782**
> **Average Recall (AR) @[IoU=0.50:0.95 | area= all | maxDets= 1] = 0.497**
> **Average Recall (AR) @[IoU=0.50:0.95 | area= all | maxDets= 10] = 0.804**
> **Average Recall (AR) @[IoU=0.50:0.95 | area= all | maxDets=100] = 0.821**
> **Average Recall (AR) @[IoU=0.50:0.95 | area= small | maxDets=100] = 0.736**
> **Average Recall (AR) @[IoU=0.50:0.95 | area=medium | maxDets=100] = 0.790**
> **Average Recall (AR) @[IoU=0.50:0.95 | area= large | maxDets=100] = 0.842**

- 接下来，我将epoch设置为60-65，第六十轮运行结果如下

> **Average Precision (AP) @[IoU=0.50:0.95 | area= all | maxDets=100] = 0.795**
> **Average Precision (AP) @[IoU=0.50 | area= all | maxDets=100] = 0.891**
> **Average Precision (AP) @[IoU=0.75 | area= all | maxDets=100] = 0.850**
> **Average Precision (AP) @[IoU=0.50:0.95 | area= small | maxDets=100] = 0.593**
> **Average Precision (AP) @[IoU=0.50:0.95 | area=medium | maxDets=100] = 0.727**
> **Average Precision (AP) @[IoU=0.50:0.95 | area= large | maxDets=100] = 0.843**
> **Average Recall (AR) @[IoU=0.50:0.95 | area= all | maxDets= 1] = 0.538**
> **Average Recall (AR) @[IoU=0.50:0.95 | area= all | maxDets= 10] = 0.868**
> **Average Recall (AR) @[IoU=0.50:0.95 | area= all | maxDets=100] = 0.880**
> **Average Recall (AR) @[IoU=0.50:0.95 | area= small | maxDets=100] = 0.725**
> **Average Recall (AR) @[IoU=0.50:0.95 | area=medium | maxDets=100] = 0.830**
> **Average Recall (AR) @[IoU=0.50:0.95 | area= large | maxDets=100] = 0.917**