一、mmdetection的安装

mmdetection的github链接：https://github.com/open-mmlab/mmdetection
mmdetection的官方文档：https://mmdetection.readthedocs.io/en/latest/

1. mmdetection的安装

1. 创建conda环境

```
conda create -n mmdetection python=3.7 -y
source activate mmdetection
```

2. 安装pytorch及torchvision

```
conda install pytorch==1.6.0 torchvision==0.7.0 cudatoolkit=10.2 -c pytorch
#也可以上官网选择特定的版本
# https://pytorch.org/
```

3. 安装mmcv
   可以在这个网址https://github.com/open-mmlab/mmcv#install-with-pip, 查找自己的环境所需的mmcv包。

```
pip install mmcv-full==latest+torch1.6.0+cu102 -f
https://download.openmmlab.com/mmcv/dist/index.html
```

4. clone mmdetection的代码库

```
git clone https://github.com/open-mmlab/mmdetection.git
cd mmdetection培养桃红
```

5. 安装mmdetection

```
pip install -r requirements/build.txt
pip install -v -e . # 这里有个点不能忘了
```

这里采用`pip install -v -e .`安装的是最小运行依赖， 如果想要使用可选择的依赖，例如`albumentations`和`imagecorruptions`可以直接运行下边的代码

```
pip install -v -e .[optional]
```

2. 验证mmdetection是否安装成功

在`demo`文件夹下创建`demo.py`,输入以下的内容：

```
from mmdet.apis import init_detector, inference_detector

config_file = 'configs/faster_rcnn/faster_rcnn_r50_fpn_1x_coco.py'
device = 'cuda:0'
# init a detector
model = init_detector(config_file, device=device)
# inference the demo image
inference_detector(model, 'demo/demo.jpg')
```

然后运行：

```
python demo/demo.py
```

二、mmdetection2.6标准数据集标准模型的训练与推理

mmdetection的github链接：https://github.com/open-mmlab/mmdetection
mmdetection的官方文档：https://mmdetection.readthedocs.io/en/latest/

主要实现如下的功能：

- 使用现有的模型对给定的图像进行推理
- 在标准数据集上测试已有的模型
- 在标准数据集上训练预定义的模型

1. 使用现有的模型进行推理

对于推理来说就是采用现有的模型来检测图像中的物体，在mmdetection中，模型在config文件中定义，而模型的参数存在于checkpoint文件中。

这里采用faster rcnn的config文件以及checkpoint文件作为基础来进行如下的操作。

1.1 高层次的推理API

mmdetection为照片的推理提供了高层的API，这里是一个简单的样例代码：

```
from mmdet.apis import init_detector, inference_detector
import mmcv

# Specify the path to model config and checkpoint file
config_file = 'configs/faster_rcnn/faster_rcnn_r50_fpn_1x_coco.py'
checkpoint_file = 'checkpoints/faster_rcnn_r50_fpn_1x_coco_20200130-047c8118.pth'

# build the model from a config file and a checkpoint file
model = init_detector(config_file, checkpoint_file, device='cuda:0')

# test a single image and show the results
img = 'test.jpg'  # or img = mmcv.imread(img), which will only load it once
```

```
result = inference_detector(model, img)
# visualize the results in a new window
# 可视化推理检测的结果
model.show_result(img, result)
# or save the visualization results to image files
# 将推理的结果保存
model.show_result(img, result, out_file='result.jpg')

# test a video and show the results
# 测试视频片段的推理结果
video = mmcv.VideoReader('video.mp4')
for frame in video:
    result = inference_detector(model, frame)
    model.show_result(frame, result, wait_time=1)
```

在`demo/inference_demo.ipynb`中可以找到这个推理的脚本代码

1.2 python3.7支持的异步接口

对于python3.7来说，mmdetection支持异步接口，通过cuda流的使用，可以在不阻止cpu和gpu绑定的推理代码的情况下，在单线程应用中能够更好的提升cpu与gpu的利用率， 可以在不同的输入数据样本之间或某个推理管道的不同模型之间同时进行推理。

在`test/async_benchmark.py`中可以对比同步与异步接口的速度。

```
import asyncio
import torch
from mmdet.apis import init_detector, async_inference_detector
from mmdet.utils.contextmanagers import concurrent

async def main():
    config_file = 'configs/faster_rcnn/faster_rcnn_r50_fpn_1x_coco.py'
    checkpoint_file = 'checkpoints/faster_rcnn_r50_fpn_1x_coco_20200130-047c8118.pth'
    device = 'cuda:0'
    model = init_detector(config_file, checkpoint=checkpoint_file, device=device)

    # queue is used for concurrent inference of multiple images
    streamqueue = asyncio.Queue()
    # queue size defines concurrency level
    streamqueue_size = 3

    for _ in range(streamqueue_size):
        streamqueue.put_nowait(torch.cuda.Stream(device=device))

    # test a single image and show the results
    img = 'test.jpg'  # or img = mmcv.imread(img), which will only load it once

    async with concurrent(streamqueue):
        result = await async_inference_detector(model, img)

    # visualize the results in a new window
    model.show_result(img, result)
```

```
    # or save the visualization results to image files
    model.show_result(img, result, out_file='result.jpg')


asyncio.run(main())
```

1.4 Demos

在 `demo` 文件夹中有两个推理的demo代码。

1.4.1 Image demo

有三个必要的参数，依次为测试图像的路径， config文件的路径， checkpoint文件的路径。还可以指定cpu还是gpu运行， 默认值为gpu， 保留bbox的分数阈值，默认为0.3

```
python demo/image_demo.py \
    ${IMAGE_FILE} \
    ${CONFIG_FILE} \
    ${CHECKPOINT_FILE} \
    [--device ${GPU_ID}] \
    [--score-thr ${SCORE_THR}]
```

例子：

```
python demo/image_demo.py demo/demo.jpg \
    configs/faster_rcnn_r50_fpn_1x_coco.py \
    checkpoints/faster_rcnn_r50_fpn_1x_coco_20200130-047c8118.pth \
    --device cpu
```

1.4.2 webcam demo(测试摄像头)

基本的参数同上，必须指定的参数为config文件的路径，checkpoint文件的路径，其他的gpu或者cpu设备， 摄像头id号， 得分阈值可以选择指定

```
python demo/webcam_demo.py \
    ${CONFIG_FILE} \
    ${CHECKPOINT_FILE} \
    [--device ${GPU_ID}] \
    [--camera-id ${CAMERA-ID}] \
    [--score-thr ${SCORE_THR}]
```

例子：

```
python demo/webcam_demo.py \
    configs/faster_rcnn_r50_fpn_1x_coco.py \
    checkpoints/faster_rcnn_r50_fpn_1x_coco_20200130-047c8118.pth
```

2. 在标准数据集上测试已经存在的模型

为了测试模型的准确率，往往需要在标准数据集上测试模型，mmdetection支持coco，voc，cityscapes等一些其他的标准数据集，在目标检测中最常用的数据集格式就是coco数据集，我的习惯都会将自己的数据集转换为这种标准格式，进行后续的训练验证测试。

2.1 测试现有的模型

mmdetection提供了测试现有模型的测试脚本。

- 单GPU
- 多GPU的单节点
- 多节点

对于不同的环境选择合适的测试脚本。

```
# single-gpu testing
python tools/test.py \
    ${CONFIG_FILE} \
    ${CHECKPOINT_FILE} \
    [--out ${RESULT_FILE}] \
    [--eval ${EVAL_METRICS}] \
    [--show]

# multi-gpu testing
bash tools/dist_test.sh \
    ${CONFIG_FILE} \
    ${CHECKPOINT_FILE} \
    ${GPU_NUM} \
    [--out ${RESULT_FILE}] \
    [--eval ${EVAL_METRICS}]
```

`tools/dist_test.sh` 支持多节点的测试。

其中除了必要的cofig文件与checkpoint文件，还有其他可选择的参数：

- `RESULT_FILE`：输出结果的文件名采用pickle格式。如果未指定，结果将不会保存到文件中。
- `EVAL_METRICS`：要根据结果评估的项目。允许的值取决于数据集，例如 `proposal_fast`，`proposal`，`bbox`，`segm` 可用于COCO， `mAP`，`recall` 为PASCAL VOC。可以通过 `cityscapes` 所有COCO度量标准来评估城市景观。
- `--show`：如果指定，检测结果将绘制在图像上并显示在新窗口中。它仅适用于单个GPU测试，并用于调试和可视化。请确保您的环境中有GUI。否则，您可能会遇到类似的错误。`cannot connect to X server`
- `--show-dir`：如果指定，检测结果将绘制在图像上并保存到指定目录。它仅适用于单个GPU测试，并用于调试和可视化。您不需要环境中的GUI即可使用此选项。
- `--show-score-thr`：如果指定，则分数低于此阈值的检测将被删除。

2.2 一些例子

假设已经下载checkpoint文件到`checkpoint/`文件夹中。

1. 测试faster rcnn并可视化结果，按任意建来查看下一张图像

```
python tools/test.py \
  configs/faster_rcnn/faster_rcnn_r50_fpn_1x_coco.py \
  checkpoints/faster_rcnn_r50_fpn_1x_coco_20200130-047c8118.pth \
  --show
```

2. 测试Faster R-CNN，并保存绘制的图像以供将来可视化。

```
python tools/test.py \
  configs/faster_rcnn/faster_rcnn_r50_fpn_1x.py \
  checkpoints/faster_rcnn_r50_fpn_1x_coco_20200130-047c8118.pth \
  --show-dir faster_rcnn_r50_fpn_1x_results
```

3. 在PASCAL VOC上测试Faster R-CNN（不保存测试结果）并评估mAP。、

```
python tools/test.py \
  configs/pascal_voc/faster_rcnn_r50_fpn_1x_voc.py \
  checkpoints/faster_rcnn_r50_fpn_1x_voc0712_20200624-c9895d40.pth \
  --eval mAP
```

4. 使用8个GPU测试Mask R-CNN，并评估bbox和mask AP。

```
./tools/dist_test.sh \
  configs/mask_rcnn_r50_fpn_1x_coco.py \
  checkpoints/mask_rcnn_r50_fpn_1x_coco_20200205-d4b0c5d6.pth \
  8 \
  --out results.pkl \
  --eval bbox segm
```

5. 使用8个GPU测试Mask R-CNN，并评估**分类**bbox和mask AP。

```
./tools/dist_test.sh \
  configs/mask_rcnn/mask_rcnn_r50_fpn_1x_coco.py \
  checkpoints/mask_rcnn_r50_fpn_1x_coco_20200205-d4b0c5d6.pth \
  8 \
  --out results.pkl \
  --eval bbox segm \
  --options "classwise=True"
```

6. 在具有8个GPU的COCO test-dev上测试Mask R-CNN，并生成JSON文件以提交给官方评估服务器。

```
./tools/dist_test.sh \
    configs/mask_rcnn/mask_rcnn_r50_fpn_1x_coco.py \
    checkpoints/mask_rcnn_r50_fpn_1x_coco_20200205-d4b0c5d6.pth \
    8 \
    -format-only \
    --options "jsonfile_prefix=./mask_rcnn_test-dev_results"
```

此命令生成两个JSON文件`mask_rcnn_test-dev_results.bbox.json`和`mask_rcnn_test-dev_results.segm.json`。

7. 在Cityscapes上使用8个GPU测试 Mask R-CNN，并生成txt和png文件提交给官方评估服务器。

```
./tools/dist_test.sh \
    configs/cityscapes/mask_rcnn_r50_fpn_1x_cityscapes.py \
    checkpoints/mask_rcnn_r50_fpn_1x_cityscapes_20200227-afe51d5a.pth \
    8 \
    --format-only \
    --options "txtfile_prefix=./mask_rcnn_cityscapes_test_results"
```

生成的png和txt将在`./mask_rcnn_cityscapes_test_results`目录下。

3. 在标准数据集上训练预训练模型

mmdetection也提供了开禧那个即用的工具来训练检测模型，这部分来显示如何训练预定的模型。

注意：默认的学习率是8gpu，每张gpu上两幅图像，也就是batch size = 16的学习率，对于不同的训练环境可以按照线性的策略来修改学习率。

3.1 单cpu训练

在`tools/train.py`中提供了在单gpu上的基本训练策略，基本使用方式是：

```
python tools/train.py \
    ${CONFIG_FILE} \
    [optional arguments]
```

在训练的过程中，日至文件与checkpoint文件会被保存在工作目录，这个工作目录可以由config文件中的work_dir 或者通过命令行--work-dir指定。

默认情况下，每个epoch都会验证一次模型的准确率， 在config文件中可以调整模型验证步骤的间隔。

```
# evaluate the model every 12 epoch.
evaluation = dict(interval=12)
```

下边是其他的一些可选择的参数：

- `--no-validate` （**不建议**）：禁用在训练期间进行评估。
- `--work-dir ${WORK_DIR}`：覆盖工作目录。
- `--resume-from ${CHECKPOINT_FILE}`：从先前的checkpoint文件继续训练。
- `--options 'Key=value'`：覆盖使用的配置中的其他设置。

注意： 在config文件中存在两个参数，`load_from`与`resume_from`,这两个参数是不同的。

resume_from不仅加载模型的权重，而且加载优化器的状态，epoch从指定的checkpoint文件中得到，这个参数通常应用于在训练过程中偶然断开的恢复继续训练。而load_from仅仅加载模型的权重，并且训练过程从0开始，通常在finetune微调时使用。

3.2 多GPU训练

在`tools/dist_train.sh`中提供了多gpu训练的脚本，基本的使用方式是：

```
./tools/dist_train.sh \
    ${CONFIG_FILE} \
    ${GPU_NUM} \
    [optional arguments]
```

其他可选的参数与单gpu训练时一致。

3.2.1 同时启动多个jobs

如果要在一台计算机上启动多个jobs，例如，在具有8个GPU的计算机上执行2个4-GPU训练jobs，则需要为每个job指定不同的端口（默认为29500），以避免通信冲突。

如果`dist_train.sh`用于启动训练jobs，则可以在命令中设置端口。

```
CUDA_VISIBLE_DEVICES=0,1,2,3 PORT=29500 ./tools/dist_train.sh ${CONFIG_FILE} 4
CUDA_VISIBLE_DEVICES=4,5,6,7 PORT=29501 ./tools/dist_train.sh ${CONFIG_FILE} 4
```

3.2.2 在多个节点上训练

MMDetection依靠`torch.distributed`软件包进行分布式训练。

3.2.3 使用Slurm管理jobs

Slurm是用于计算集群的良好作业调度系统。在Slurm管理的群集上，您可以`slurm_train.sh`用来生成训练作业。它支持单节点和多节点训练。

基本用法如下：

```
[GPUS=${GPUS}] ./tools/slurm_train.sh ${PARTITION} ${JOB_NAME} ${CONFIG_FILE} ${WORK_DIR}
```

下面是一个使用16 GPU在一个名为dev的slurm分区来训练Mask rcnn的例子，并且设置work_dir为某个共享文件系统。

```
GPUS=16 ./tools/slurm_train.sh dev mask_r50_1x configs/mask_rcnn_r50_fpn_1x_coco.py
/nfs/xxxx/mask_rcnn_r50_fpn_1x
```

您可以检查源代码以查看完整的参数和环境变量。

使用Slurm时，需要通过以下方式之一设置端口选项：

1. 通过设置端口`--options`。推荐这样做，因为它不会更改原始配置。

```
CUDA_VISIBLE_DEVICES=0,1,2,3 GPUS=4 ./tools/slurm_train.sh ${PARTITION} ${JOB_NAME}
config1.py ${WORK_DIR} --options 'dist_params.port=29500'
CUDA_VISIBLE_DEVICES=4,5,6,7 GPUS=4 ./tools/slurm_train.sh ${PARTITION} ${JOB_NAME}
config2.py ${WORK_DIR} --options 'dist_params.port=29501'
```

2. 修改配置文件以设置不同的通信端口。

   在中`config1.py`，设置

   ```
   dist_params = dict(backend='nccl', port=29500)
   ```

   在中`config2.py`，设置

   ```
   dist_params = dict(backend='nccl', port=29501)
   ```

   然后，您可以使用`config1.py`和启动两个作业`config2.py`。

   ```
   CUDA_VISIBLE_DEVICES=0,1,2,3 GPUS=4 ./tools/slurm_train.sh ${PARTITION} ${JOB_NAME}
   config1.py ${WORK_DIR}
   CUDA_VISIBLE_DEVICES=4,5,6,7 GPUS=4 ./tools/slurm_train.sh ${PARTITION} ${JOB_NAME}
   config2.py ${WORK_DIR}
   ```

以上基本上就是官方英文文档的翻译。

三、mmdetection2.6利用自定义数据集训练模型

在这篇文章中，介绍如何采用自定义数据集来训练，测试，推理预定义的模型。

基本的步骤如下：

- 准备自定义数据集
- 准备config文件
- 在自定义数据集上训练，测试，推理模型

1. 准备自定义数据集

mmdetection支持三种方法来自定义数据集：

1. 将数据集组织为coco格式
2. 将数据集组织为middle格式
3. 应用一个新的数据集

前两种方式更简单，更容易操作一点

在这篇文章中，提供一个将自己数据集转换为coco数据集格式的例子。

注意： 对于评估mask AP这种势力分割任务， mmdetection 仅仅支持COCO格式的数据集，因此如果进行实例分割就需要将数据集转换为coco格式的数据集。

1.1 coco标注格式

对于实例分割任务， coco格式的数据集必要的键为：

```
{
  "images": [image],
  "annotations": [annotation],
  "categories": [category]
}


image = {
  "id": int,
  "width": int,
  "height": int,
  "file_name": str,
}

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

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

需要将自己的数据集转换为coco格式的标注信息，然后利用coco格式的数据集load数据，使用CocoDataset来训练与评估模型。

2. 准备config文件

假设采用Mask RCNN+FPN这种结构来训练检测器。假设这个config文件位于`configs/ballon`中，命名为`mask_rcnn_r50_caffe_fpn_mstrain-poly_1x_balloon.py`

config文件的内容如下：

```
# The new config inherits a base config to highlight the necessary modification
_base_ = 'mask_rcnn/mask_rcnn_r50_caffe_fpn_mstrain-poly_1x_coco.py'

# We also need to change the num_classes in head to match the dataset's annotation
model = dict(
  roi_head=dict(
    bbox_head=dict(num_classes=1),
    mask_head=dict(num_classes=1)))

# Modify dataset related settings
dataset_type = 'COCODataset'
```

```
classes = ('balloon',)
data = dict(
  train=dict(
    img_prefix='balloon/train/',
    classes=classes,
    ann_file='balloon/train/annotation_coco.json'),
  val=dict(
    img_prefix='balloon/val/',
    classes=classes,
    ann_file='balloon/val/annotation_coco.json'),
  test=dict(
    img_prefix='balloon/val/',
    classes=classes,
    ann_file='balloon/val/annotation_coco.json'))

# We can use the pre-trained Mask RCNN model to obtain higher performance
load_from = 'checkpoints/mask_rcnn_r50_caffe_fpn_mstrain-poly_3x_coco_bbox_mAP-0.408__segm_mAP-0.37_20200504_163245-42aa3d00.pth'
```

3. 训练测试 推理模型

训练

```
python tools/train.py configs/ballon/mask_rcnn_r50_caffe_fpn_mstrain-poly_1x_balloon.py
```

测试

```
python tools/test.py configs/ballon/mask_rcnn_r50_caffe_fpn_mstrain-poly_1x_balloon.py work_dirs/mask_rcnn_r50_caffe_fpn_mstrain-poly_1x_balloon.py/latest.pth --eval bbox segm
```

四、mmdetection2.6的config文件

mmdetection合并了模块化与继承设计的思想来构成config系统，利用这种系统可以方便的执行多样化的实验。如果

想要检查配置文件，可以运行如下的代码来查看完整的配置文件：

```
python tools/print_config.py /PATH/TO/CONFIG
```

1. 配置文件的结构

在 config/_base_ 中存在4中基本的组件类型，分别为：dataset(数据集), model(模型）, schedule（学习率调整的粗略），default_runtime（默认运行时设置）。使用Faster R-CNN，Mask R-CNN，Cascade R-CNN，RPN，SSD中的一个可以轻松构造许多方法。来自 __base__ 中的组件称为*promitive*.

对于同一文件夹下的所有配置，建议仅具有**一个** *原始（promitive）*配置。所有其他配置应从*原始（promitive）*配置继承。这样，继承级别的最大值为3。

便于理解，构建爱你模型推荐从现有的方法中继承。例如，如果有一些修改是基于faster rcnn的， 那么使用者可能通过指定 _base_ = ../faster_rcnn/faster_rcnn_r50_fpn_1x_coco.py 首先继承基本的faster rcnn架构， 然后修改必要的config文件。

如果构建了一种没有基于任何架构的完全新方法，可以在configs文件夹下创建一个新的文件夹。

2. config文件命名风格

利用下面的文件命名方法命名：

```
{model}_[model setting]_{backbone}_{neck}_[norm setting]_[misc]_[gpu x batch_per_gpu]_{schedule}_{dataset}
```

{xxx} 是必填字段， [yyy] 是可选字段。

- {model}：模型的类型， 例如 faster_rcnn ， mask_rcnn 等。
- [model setting]：某些模型的特定设置，例如 without_semantic for htc ， moment for reppoints 等。
- {backbone}：骨干类型，例如 r50 （ResNet-50），x101 （ResNeXt-101）。
- {neck}：neck的类型， 例如 fpn ， pafpn ， nasfpn ， c4 。
- [norm_setting]：如果不特别指定，默认使用 bn ，否则使用其他标准图层类型可以是 gn （Group Normalization），syncbn （Synchronized Batch Normalization）。 gn-head / gn-neck 表示GN仅应用于头部/颈部，而 gn-all 表示GN应用于整个模型，例如，骨架，颈部，头部。
- [misc]：其他的设置/模型的插件，例如 dconv ， gcb ， attention ， albu ， mstrain 。
- [gpu x batch_per_gpu]：GPU数目和每个GPU的样本数目， 默认使用 8x2 。
- {schedule}：训练进度表，可选的有 1x ， 2x ， 20e ， 1x 和 2x 默认12与24 epoch。 20e 在级联模型中采用，表示20个epoch。对于 1x / 2x ，初始学习率在第8/16和11/22个epoch衰减10倍。对于 20e ，初始学习率在第16和19个epoch衰减10倍。
- {dataset}：数据集， 例如 coco ， cityscapes ， voc_0712 ， wider_face 。

3. mask rcnn的config文件

下边简单介绍maskrcnn + resnet50 + fpn的config文件介绍，文件存在于 configs/__base__/models/mask_rcnn_r50_fpn.py 。

```
model = dict(
  type='MaskRCNN', # 检测器的名名称
  pretrained=
  'torchvision://resnet50', # 加载的ImageNet预训练权重。
  backbone=dict( # backbone的cofig文件
    type='ResNet', # backbone的类型, 更多的细节参考 https://github.com/open-mmlab/mmdetection/blob/master/mmdet/models/backbones/resnet.py#L288。
    depth=50, # backbone的深度, 通常情况下resnet采用50， resnext采用101
```

```
    num_stages=4, # backbone的段的数目.
    out_indices=(0, 1, 2, 3), # 每个stage所产生的feature map的索引
    frozen_stages=1, # 冻结第一个stage不训练
    norm_cfg=dict( # 正则化层的config参数
      type='BN', # 正则化层
      requires_grad=True), # 是否训练BN中的gamma与beta
    norm_eval=True, # 是否冻结BN中的统计量
    style='pytorch'), # backbone的风格，可选的有pytorch与caffe, 'pytorch' 参数表示stride为2的过程在3x3的卷积层中，而 'caffe' 表示stride 2的层在1x1的卷积层中）。
  neck=dict(
    type='FPN', # 检测器的neck是FPN. 支持 'NASFPN', 'PAFPN'等. 更多细节可参考https://github.com/open-mmlab/mmdetection/blob/master/mmdet/models/necks/fpn.py#L10.
    in_channels=[256, 512, 1024, 2048], # FPN的输入channels, 这与neck的输出通道一致
    out_channels=256, # 金字塔特征映射的每个级别的输出通道
    num_outs=5), # 输出尺度的数目
  rpn_head=dict(
    type='RPNHead', # RPN head is 'RPNHead', 框架同时支持 'GARPNHead'等. 更多细节参考https://github.com/open-mmlab/mmdetection/blob/master/mmdet/models/dense_heads/rpn_head.py#L12
    in_channels=256, # 每个输入特征图的通道数， 这与neck的输出通道数一致。
    feat_channels=256, # 卷积层的特征通道
    anchor_generator=dict( # 生成anchor的配置
      type='AnchorGenerator', # 大多数方法使用AnchorGenerator, 而ssd检测器采用`SSDAnchorGenerator`. 更多细节参考https://github.com/open-mmlab/mmdetection/blob/master/mmdet/core/anchor/anchor_generator.py#L10
      scales=[8], # anchor的基本尺度，一个featuremap的anchor的面积通过公式计算： scale * base_sizes
      ratios=[0.5, 1.0, 2.0], # anchor的宽高比
      strides=[4, 8, 16, 32, 64]), # anchor生成器的步长，这与FPN特征的步长一致。 如果base_size没有设置，这个步长将会作为base_size.
    bbox_coder=dict( # 边界框编码器的配置，在训练与测试时编码与解码边界框
      type='DeltaXYWHBBoxCoder', # 边界框编码器的配置. 大多数的方法采用'DeltaXYWHBBoxCoder'. 更多的细节参考https://github.com/open-mmlab/mmdetection/blob/master/mmdet/core/bbox/coder/delta_xywh_bbox_coder.py#L9
      target_means=[0.0, 0.0, 0.0, 0.0], # 目标的均值，这个值用来编码与解码边界框
      target_stds=[1.0, 1.0, 1.0, 1.0]), # 目标的方差，这个值用来编码与解码边界框
    loss_cls=dict( # 分类分支的loss的配置
      type='CrossEntropyLoss', # 分类分支loss的类型，除了交叉商损失外，也支持focal loss.
      use_sigmoid=True, # RPN曾通常为二分类任务，用来区分前景与背景，采用sigmoid激活函数
      loss_weight=1.0), # 分类分支loss的权重
    loss_bbox=dict( # 回归分支lossfunction的config配置
      type='L1Loss', # 回归损失的类型，除了l1 loss之外，也支持smooth l1 loss与各种IOU loss. 更多的细节参考https://github.com/open-mmlab/mmdetection/blob/master/mmdet/models/losses/smooth_l1_loss.py#L56
      loss_weight=1.0)), # 回归分支的损失权重
  roi_head=dict( # ROIHead封装了级联检测器或者两部法的第二个stage
    type='StandardRoIHead', # ROIHead的类型.更多的细节参考https://github.com/open-mmlab/mmdetection/blob/master/mmdet/models/roi_heads/standard_roi_head.py#L10.
    bbox_roi_extractor=dict( # 用于bbox回归的ROI特征提取器config.
      type='SingleRoIExtractor', # ROI特征提取其的类型,大多数方法采用SingleRoIExtractor. 更多的细节参考https://github.com/open-mmlab/mmdetection/blob/master/mmdet/models/roi_heads/roi_extractors/single_level.py#L10
      roi_layer=dict( # ROI层的config
        type='RoIAlign', # ROIAlign的类型, 同时也支持DeformRoIPoolingPack and ModulatedDeformRoIPoolingPack. Refer to https://github.com/open-mmlab/mmdetection/blob/master/mmdet/ops/roi_align/roi_align.py#L79 for details.
        output_size=7, # featuremap的输出尺寸.
        sampling_ratio=0), # 当提取ROI特征时的采样率， 如果采样率为0， 表示自适应率
      out_channels=256, # 提取得到的特征的输出通道.
```

```
        featmap_strides=[4, 8, 16, 32]), # 多尺度特征图的步长， 应该与backbone的输出一致
    bbox_head=dict( # 在ROI head中 的box headconfig
        type='Shared2FCBBoxHead', # box head的步长, Refer to https://github.com/open-mmlab/mmdetection/blob/master/mmdet/models/roi_heads/bbox_heads/convfc_bbox_head.py#L177 for implementation details.
        in_channels=256, #bbox的输入通道，应该与roi特征提取器的输出一致.
        fc_out_channels=1024, # 全链接层的输出通道.
        roi_feat_size=7, # roi特征的尺寸
        num_classes=80, # 分类的类别数目
        bbox_coder=dict( # 在第二个stage中的bbox编码器的config
            type='DeltaXYWHBBoxCoder', # bbox编码器的类型. 大多数方法采用'DeltaXYWHBBoxCoder' .
            target_means=[0.0, 0.0, 0.0, 0.0], # Means used to encode and decode box
            target_stds=[0.1, 0.1, 0.2, 0.2]), # 编解码的方差，这个方差更小，因为此时的bbox已经西那个对比较准确. [0.1, 0.1, 0.2, 0.2] is a conventional setting.
        reg_class_agnostic=False, # 是否回归分支是不知道类别的.
        loss_cls=dict( # 分类分支的lossfunction
            type='CrossEntropyLoss',
            use_sigmoid=False, # 是否采用sigmoid
            loss_weight=1.0), # 分类loss的权重
        loss_bbox=dict(
            type='L1Loss',
            loss_weight=1.0)),
    mask_roi_extractor=dict( # bbox回归的特征提取器.
        type='SingleRoIExtractor', # ROI特征提取器的类型，大多数方法采用SingleRoIExtractor.
        roi_layer=dict( # 用于实例分割的特征提取ROI层
            type='RoIAlign', # Type of RoI Layer, DeformRoIPoolingPack and ModulatedDeformRoIPoolingPack are also supported
            output_size=14, # The output size of feature maps.
            sampling_ratio=0), # Sampling ratio when extracting the RoI features.
        out_channels=256, # Output channels of the extracted feature.
        featmap_strides=[4, 8, 16, 32]), # Strides of multi-scale feature maps.
    mask_head=dict( # mask预测head
        type='FCNMaskHead', # Type of mask head, refer to https://github.com/open-mmlab/mmdetection/blob/master/mmdet/models/roi_heads/mask_heads/fcn_mask_head.py#L21 for implementation details.
        num_convs=4, # mask head中卷积层的数目.
        in_channels=256, # 输入通道数目， 应该与mask特征提取层的输出通道数目一致.
        conv_out_channels=256, # 卷积层的输出通道数目.
        num_classes=80, # 分割的类别数目.
        loss_mask=dict( # Config of loss function for the mask branch.
            type='CrossEntropyLoss', # Type of loss used for segmentation
            use_mask=True, # 是否仅仅训练mask正确的位置
            loss_weight=1.0)))) # Loss weight of mask branch.


train_cfg = dict( #rpn与rcnn的训练超参数设置。
  rpn=dict( # rpn的训练config配置
    assigner=dict( # assigner的config
        type='MaxIoUAssigner', # assigner的类型, 大多数检测器采用MaxIoUAssigner. Refer to https://github.com/open-mmlab/mmdetection/blob/master/mmdet/core/bbox/assigners/max_iou_assigner.py#L10 for more details.
        pos_iou_thr=0.7, # IoU >= threshold 0.7 will be taken as positive samples
        neg_iou_thr=0.3, # IoU < threshold 0.3 will be taken as negative samples
        min_pos_iou=0.3, # The minimal IoU threshold to take boxes as positive samples
        match_low_quality=True, # Whether to match the boxes under low quality (see API doc for more details).
        ignore_iof_thr=-1), # IoF threshold for ignoring bboxes
```

```
    sampler=dict(  # Config of positive/negative sampler
      type='RandomSampler',  # Tsampler的类型, 也支持其他的PseudoSampler and other samplers. Refer to https://github.com/open-mmlab/mmdetection/blob/master/mmdet/core/bbox/samplers/random_sampler.py#L8 for implementation details.
      num=256,  # samples的数目
      pos_fraction=0.5,  # 所有的样本中正样本的比例
      neg_pos_ub=-1,  # The upper bound of negative samples based on the number of positive samples.
      add_gt_as_proposals=False),  # 是否在采样后将GT作为正阳本加入samples
    allowed_border=-1,  # The border allowed after padding for valid anchors.
    pos_weight=-1,  # 训练过程中正样本的去
    debug=False),  # 是否设置debug模式
  rpn_proposal=dict(  # 训练过程中生成proposal的config。
    nms_across_levels=False,  # 是否跨层次执行nms
    nms_pre=2000,  # nms之前的proposal的数目
    nms_post=1000,  # nms保留的样本的数目
    max_num=1000,  # nms之后使用的proposal数目
    nms_thr=0.7,  # nms中使用的阈值
    min_bbox_size=0),  # 最小的box的尺寸
  rcnn=dict(  # The config for the roi heads.
    assigner=dict(  # Config of assigner for second stage, this is different for that in rpn
      type='MaxIoUAssigner',  # Type of assigner, 现在所有的roi_head均使用MaxIoUAssigner. Refer to https://github.com/open-mmlab/mmdetection/blob/master/mmdet/core/bbox/assigners/max_iou_assigner.py#L10 for more details.
      pos_iou_thr=0.5,  # IoU >= threshold 0.5 will be taken as positive samples
      neg_iou_thr=0.5,  # IoU< threshold 0.5 will be taken as negative samples
      min_pos_iou=0.5,  # The minimal IoU threshold to take boxes as positive samples
      match_low_quality=False,  # Whether to match the boxes under low quality (see API doc for more details).
      ignore_iof_thr=-1),  # IoF threshold for ignoring bboxes
    sampler=dict(
      type='RandomSampler',  # Type of sampler, PseudoSampler and other samplers are also supported. Refer to https://github.com/open-mmlab/mmdetection/blob/master/mmdet/core/bbox/samplers/random_sampler.py#L8 for implementation details.
      num=512,  # Number of samples
      pos_fraction=0.25,  # The ratio of positive samples in the total samples.
      neg_pos_ub=-1,  # The upper bound of negative samples based on the number of positive samples.
      add_gt_as_proposals=True
    ),  # Whether add GT as proposals after sampling.
    mask_size=28,  # Size of mask
    pos_weight=-1,  # The weight of positive samples during training.
    debug=False))  # Whether to set the debug mode


test_cfg = dict(  # Config for testing hyperparameters for rpn and rcnn
  rpn=dict(  # The config to generate proposals during testing
    nms_across_levels=False,  # Whether to do NMS for boxes across levels
    nms_pre=1000,  # The number of boxes before NMS
    nms_post=1000,  # The number of boxes to be kept by NMS
    max_num=1000,  # The number of boxes to be used after NMS
    nms_thr=0.7,  # The threshold to be used during NMS
    min_bbox_size=0),  # The allowed minimal box size
  rcnn=dict(  # The config for the roi heads.
    score_thr=0.05,  # Threshold to filter out boxes
    nms=dict(  # Config of nms in the second stage
      type='nms',  # Type of nms
```

```
        iou_thr=0.5),  # NMS threshold
      max_per_img=100,  # Max number of detections of each image
      mask_thr_binary=0.5))  # Threshold of mask prediction


dataset_type = 'CocoDataset'  # 数据集的类型
data_root = 'data/coco/'  # 数据的根目录
img_norm_cfg = dict(  # 图像正则化config
  mean=[123.675, 116.28, 103.53],  # Mean values used to pre-training the pre-trained backbone models
  std=[58.395, 57.12, 57.375],  # Standard variance used to pre-training the pre-trained backbone models
  to_rgb=True
)  # The channel orders of image used to pre-training the pre-trained backbone models


train_pipeline = [  # Training pipeline
  dict(type='LoadImageFromFile'),  # First pipeline to load images from file path
  dict(
    type='LoadAnnotations',  # Second pipeline to load annotations for current image
    with_bbox=True,  # Whether to use bounding box, True for detection
    with_mask=True,  # Whether to use instance mask, True for instance segmentation
    poly2mask=False),  # Whether to convert the polygon mask to instance mask, set False for
acceleration and to save memory
  dict(
    type='Resize',  # Augmentation pipeline that resize the images and their annotations
    img_scale=(1333, 800),  # The largest scale of image
    keep_ratio=True
  ),  # whether to keep the ratio between height and width.
  dict(
    type='RandomFlip',  # Augmentation pipeline that flip the images and their annotations
    flip_ratio=0.5),  # The ratio or probability to flip
  dict(
    type='Normalize',  # Augmentation pipeline that normalize the input images
    mean=[123.675, 116.28, 103.53],  # These keys are the same of img_norm_cfg since the
    std=[58.395, 57.12, 57.375],  # keys of img_norm_cfg are used here as arguments
    to_rgb=True),
  dict(
    type='Pad',  # Padding config
    size_divisor=32),  # The number the padded images should be divisible
  dict(type='DefaultFormatBundle'),  # Default format bundle to gather data in the pipeline
  dict(
    type='Collect',  # Pipeline that decides which keys in the data should be passed to the detector
    keys=['img', 'gt_bboxes', 'gt_labels', 'gt_masks'])
]


test_pipeline = [
  dict(type='LoadImageFromFile'),  # First pipeline to load images from file path
  dict(
    type='MultiScaleFlipAug',  # An encapsulation that encapsulates the testing augmentations
    img_scale=(1333, 800),  # Decides the largest scale for testing, used for the Resize pipeline
    flip=False,  # Whether to flip images during testing
    transforms=[
      dict(type='Resize',  # Use resize augmentation
         keep_ratio=True),  # Whether to keep the ratio between height and width, the img_scale set here
will be supressed by the img_scale set above.
      dict(type='RandomFlip'),  # Thought RandomFlip is added in pipeline, it is not used because
flip=False
      dict(
```

```
            type='Normalize',  # Normalization config, the values are from img_norm_cfg
            mean=[123.675, 116.28, 103.53],
            std=[58.395, 57.12, 57.375],
            to_rgb=True),
        dict(
            type='Pad',  # Padding config to pad images divisable by 32.
            size_divisor=32),
        dict(
            type='ImageToTensor',  # convert image to tensor
            keys=['img']),
        dict(
            type='Collect',  # Collect pipeline that collect necessary keys for testing.
            keys=['img'])
    ])
]



data = dict(
  samples_per_gpu=2,  # 每个gpu上的images， 这里乘以gpu的数目即为batch size
  workers_per_gpu=2,  # Worker to pre-fetch data for each single GPU
  train=dict(  # Train dataset config
    type='CocoDataset',  # Type of dataset, refer to https://github.com/open-
mmlab/mmdetection/blob/master/mmdet/datasets/coco.py#L19 for details.
    ann_file='data/coco/annotations/instances_train2017.json',  # Path of annotation file
    img_prefix='data/coco/train2017/',  # Prefix of image path
    pipeline=[  # pipeline, this is passed by the train_pipeline created before.
      dict(type='LoadImageFromFile'),
      dict(
        type='LoadAnnotations',
        with_bbox=True,
        with_mask=True,
        poly2mask=False),
      dict(type='Resize', img_scale=(1333, 800), keep_ratio=True),
      dict(type='RandomFlip', flip_ratio=0.5),
      dict(
        type='Normalize',
        mean=[123.675, 116.28, 103.53],
        std=[58.395, 57.12, 57.375],
        to_rgb=True),
      dict(type='Pad', size_divisor=32),
      dict(type='DefaultFormatBundle'),
      dict(
        type='Collect',
        keys=['img', 'gt_bboxes', 'gt_labels', 'gt_masks'])
    ]),
  val=dict(  # Validation dataset config
    type='CocoDataset',
    ann_file='data/coco/annotations/instances_val2017.json',
    img_prefix='data/coco/val2017/',
    pipeline=[  # Pipeline is passed by test_pipeline created before
      dict(type='LoadImageFromFile'),
      dict(
        type='MultiScaleFlipAug',
        img_scale=(1333, 800),
        flip=False,
        transforms=[
          dict(type='Resize', keep_ratio=True),
```

```
            dict(type='RandomFlip'),
            dict(
              type='Normalize',
              mean=[123.675, 116.28, 103.53],
              std=[58.395, 57.12, 57.375],
              to_rgb=True),
            dict(type='Pad', size_divisor=32),
            dict(type='ImageToTensor', keys=['img']),
            dict(type='Collect', keys=['img'])
          ])
    ]),
  test=dict(  # Test dataset config, modify the ann_file for test-dev/test submission
    type='CocoDataset',
    ann_file='data/coco/annotations/instances_val2017.json',
    img_prefix='data/coco/val2017/',
    pipeline=[  # Pipeline is passed by test_pipeline created before
      dict(type='LoadImageFromFile'),
      dict(
        type='MultiScaleFlipAug',
        img_scale=(1333, 800),
        flip=False,
        transforms=[
          dict(type='Resize', keep_ratio=True),
          dict(type='RandomFlip'),
          dict(
            type='Normalize',
            mean=[123.675, 116.28, 103.53],
            std=[58.395, 57.12, 57.375],
            to_rgb=True),
          dict(type='Pad', size_divisor=32),
          dict(type='ImageToTensor', keys=['img']),
          dict(type='Collect', keys=['img'])
        ])
    ],
    samples_per_gpu=2  # Batch size of a single GPU used in testing
    ))

evaluation = dict(  # The config to build the evaluation hook, refer to https://github.com/open-mmlab/mmdetection/blob/master/mmdet/core/evaluation/eval_hooks.py#L7 for more details.
  interval=1, #每1个epoch评估一次，这里也可以修改
  metric=['bbox', 'segm']) # 评估标准
optimizer = dict(  # Config used to build optimizer, support all the optimizers in PyTorch whose arguments are also the same as those in PyTorch
  type='SGD',  # Type of optimizers, refer to https://github.com/open-mmlab/mmdetection/blob/master/mmdet/core/optimizer/default_constructor.py#L13 for more details
  lr=0.02,  # Learning rate of optimizers, see detail usages of the parameters in the documentaion of PyTorch
  momentum=0.9,  # Momentum
  weight_decay=0.0001)  # Weight decay of SGD
optimizer_config = dict(  # Config used to build the optimizer hook, refer to https://github.com/open-mmlab/mmcv/blob/master/mmcv/runner/hooks/optimizer.py#L8 for implementation details.
  grad_clip=None)  # Most of the methods do not use gradient clip
lr_config = dict(  # Learning rate scheduler config used to register LrUpdater hook
  policy='step',  # The policy of scheduler, also support CosineAnnealing, Cyclic, etc. Refer to details of supported LrUpdater from https://github.com/open-mmlab/mmcv/blob/master/mmcv/runner/hooks/lr_updater.py#L9.
  warmup='linear',  # The warmup policy, also support `exp` and `constant`.
  warmup_iters=500,  # The number of iterations for warmup
```

```
    warmup_ratio=
    0.001,  # The ratio of the starting learning rate used for warmup
    step=[8, 11])  # Steps to decay the learning rate
total_epochs = 12  # Total epochs to train the model
checkpoint_config = dict(  # Config to set the checkpoint hook, Refer to https://github.com/open-mmlab/mmcv/blob/master/mmcv/runner/hooks/checkpoint.py for implementation.
    interval=1)  # The save interval is 1
log_config = dict(  # config to register logger hook
    interval=50,  # Interval to print the log
    hooks=[
        # dict(type='TensorboardLoggerHook')  # The Tensorboard logger is also supported
        dict(type='TextLoggerHook')
    ])  # The logger used to record the training process.
dist_params = dict(backend='nccl')  # Parameters to setup distributed training, the port can also be set.
log_level = 'INFO'  # The level of logging.
load_from = None  # 加载预训练模型权重，这里不是resume训练采用
resume_from = None  # 这里resume训练采用
workflow = [('train', 1)]  # Workflow for runner. [('train', 1)] means there is only one workflow and the workflow named 'train' is executed once. The workflow trains the model by 12 epochs according to the total_epochs.
work_dir = 'work_dir'  # 这个路径用来存储模型与日志文件
```

4. 一些问题：

在继承关系中有时可以设置 `_delete_=True` 来忽视在base config中的一些键。在mmcv中可以看到具体的一些指导细节。

在mmdetection中，在mask rcnn的config文件中来修改backbone。

```
model = dict(
    type='MaskRCNN',
    pretrained='torchvision://resnet50',
    backbone=dict(
        type='ResNet',
        depth=50,
        num_stages=4,
        out_indices=(0, 1, 2, 3),
        frozen_stages=1,
        norm_cfg=dict(type='BN', requires_grad=True),
        norm_eval=True,
        style='pytorch'),
    neck=dict(...),
    rpn_head=dict(...),
    roi_head=dict(...))
```

`ResNet` and `HRNet` 使用不同的键来构建

```
_base_ = '../mask_rcnn/mask_rcnn_r50_fpn_1x_coco.py'
model = dict(
    pretrained='open-mmlab://msra/hrnetv2_w32',
    backbone=dict(
        _delete_=True,
        type='HRNet',
```

```
    extra=dict(
      stage1=dict(
        num_modules=1,
        num_branches=1,
        block='BOTTLENECK',
        num_blocks=(4, ),
        num_channels=(64, )),
      stage2=dict(
        num_modules=1,
        num_branches=2,
        block='BASIC',
        num_blocks=(4, 4),
        num_channels=(32, 64)),
      stage3=dict(
        num_modules=4,
        num_branches=3,
        block='BASIC',
        num_blocks=(4, 4, 4),
        num_channels=(32, 64, 128)),
      stage4=dict(
        num_modules=3,
        num_branches=4,
        block='BASIC',
        num_blocks=(4, 4, 4, 4),
        num_channels=(32, 64, 128, 256)))),
  neck=dict(...))
```

`_delete_=True` 在 `backbone` 中利用新键来取代旧键。

4.1 在配置中使用中间变量

一些中间变量在配置文件中是非常重要的，例如，datasets中的train_pipeline与test_pipeline。值得注意的是，在子配置文件中修改中间变量时，用户需要再次将中间变量传递到相应的字段中。例如，我们想使用多尺度策略来训练Mask R-CNN。 `train_pipeline` / `test_pipeline` 是我们要修改的中间变量。

```
_base_ = './mask_rcnn_r50_fpn_1x_coco.py'
img_norm_cfg = dict(
  mean=[123.675, 116.28, 103.53], std=[58.395, 57.12, 57.375], to_rgb=True)
train_pipeline = [
  dict(type='LoadImageFromFile'),
  dict(type='LoadAnnotations', with_bbox=True, with_mask=True),
  dict(
    type='Resize',
    img_scale=[(1333, 640), (1333, 672), (1333, 704), (1333, 736),
          (1333, 768), (1333, 800)],
    multiscale_mode="value",
    keep_ratio=True),
  dict(type='RandomFlip', flip_ratio=0.5),
  dict(type='Normalize', **img_norm_cfg),
  dict(type='Pad', size_divisor=32),
  dict(type='DefaultFormatBundle'),
  dict(type='Collect', keys=['img', 'gt_bboxes', 'gt_labels', 'gt_masks']),
]
```

```
test_pipeline = [
  dict(type='LoadImageFromFile'),
  dict(
    type='MultiScaleFlipAug',
    img_scale=(1333, 800),
    flip=False,
    transforms=[
      dict(type='Resize', keep_ratio=True),
      dict(type='RandomFlip'),
      dict(type='Normalize', **img_norm_cfg),
      dict(type='Pad', size_divisor=32),
      dict(type='ImageToTensor', keys=['img']),
      dict(type='Collect', keys=['img']),
    ])
]
data = dict(
  train=dict(pipeline=train_pipeline),
  val=dict(pipeline=test_pipeline),
  test=dict(pipeline=test_pipeline))
```

这里我们首先定义了新的train_pipeline与test_pipeline并且将他们传给data。

五、mmdetection2.6自定义数据集

官方推荐的集中自定义数据集的方式：

- 将自己的数据集组织为标准的数据集格式（常用COCO）
- 将自己的数据集组织为中间格式
- 利用datawraper自定义新的数据集

1. 将数据集组织为coco数据集格式

coco数据集的标注格式：

整体为一个字典形式，主要的键为：`images`，`annotations`，`categorties`

- `images`的值为一个列表，列表的每个元素为如下所示的元素信息
- `annotations`的值为一个列表， 每个元素为如下所示的信息
- `categorties`的值为一个列表，每个元素为如下所示的信息，id从0开始。

```
'images': [
  {
    'file_name': 'COCO_val2014_000000001268.jpg',
    'height': 427,
    'width': 640,
    'id': 1268
  },
  ...
],

'annotations': [
  {
    'segmentation': [[192.81,
```

```
            247.09,
            ...
            219.03,
            249.06]],  # if you have mask labels
        'area': 1035.749,
        'iscrowd': 0,
        'image_id': 1268,
        'bbox': [192.81, 224.8, 74.73, 33.43],
        'category_id': 16,
        'id': 42986
    },
    ...
],

'categories': [
    {'id': 0, 'name': 'car'},
 ]
```

最简单的方式，就是将自己的数据集组织为coco数据集的标注 格式，这样训练的过程中仅仅需要在config文件中修改**数据集的路径**与**类别**即可。

2. 将数据集组织为middle格式

mmdetection定义一种比较简单的数据集格式，标注文件的信息是一个字典列表，每个字典对应着一张图片，如下所示：

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

转换为上述的格式之后，有两种数据集的使用方式，一种在线的使用方式，一种离线的方式：

- 在线方式

重新写一个继承自`CustomDataset`的类。重写 `load_annotations(self, ann_file)` and `get_ann_info(self, idx)` 这两个方法。

- 离线的方法

将数据集转换为标准的COCO或者VOC格式，然后直接使用`CustomDataset`。

3. 简单的自定义数据集的例子

假设我们现有的标注数据的格式为txt文件标注

```
#分别为图片名称， 图片宽高， bbox的数目， bbox坐标与类别id
000001.jpg
1280 720
2
10 20 40 60 1
20 40 50 60 2

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

然后创建一个新的文件 `mmdet/datasets/my_dataset.py` 来加载数据集。

```
import mmcv
import numpy as np

from .builder import DATASETS
from .custom import CustomDataset


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

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

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

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

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

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

```
            labels.append(int(anns[4]))

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

    return data_infos

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

然后在config文件中，使用`MyDataset`。

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

5. 使用dataset wrappers来自定义数据集

mmdetection支持很多中数据集wrapper来混合数据集或者在训练时修改数据集分布。现在着吃三种数据集包装器（wrapper）：

- `RepeatDataset`：只需重复整个数据集。
- `ClassBalancedDataset`：以类平衡的方式重复数据集。
- `ConcatDataset`：concat数据集。

5.1 RepeatDataset

使用`RepeatDataset`作为数据集包装器来重复数据集。例如重复原始的数据集`Dataset_A`。config文件如下：

```
dataset_A_train = dict(
    type='RepeatDataset',
    times=N,
    dataset=dict(  # This is the original config of Dataset_A
      type='Dataset_A',
      ...
      pipeline=train_pipeline
    )
  )
```

5.2 ClassBalancedDataset

使用`ClassBalancedDataset`作为wrapper来依据类别的频率来重复数据集，重复的数据集需要初始化函数`self.get_cat_ids(idx)`来支持`ClassBalancedDataset`。例如使用过采样率`oversample_thr=1e-3`来重复`Dataset_A`。

```
dataset_A_train = dict(
    type='ClassBalancedDataset',
    oversample_thr=1e-3,
    dataset=dict(  # This is the original config of Dataset_A
      type='Dataset_A',
      ...
      pipeline=train_pipeline
    )
  )
```

5.3 ConcatDataset

有三种方法堆叠数据集

5.3.1 两个数据集是同样的类型

采用如下的方式：

```
dataset_A_train = dict(
  type='Dataset_A',
  ann_file = ['anno_file_1', 'anno_file_2'],
  pipeline=train_pipeline
)
```

这种方式在测试验证过程中，两个数据集会分开进行测试，如果想要整体进行测试，需要`separate_eval=False`

```
dataset_A_train = dict(
  type='Dataset_A',
  ann_file = ['anno_file_1', 'anno_file_2'],
  separate_eval=False,
  pipeline=train_pipeline
)
```

5.3.2 两个数据集不同

```
dataset_A_train = dict()
dataset_B_train = dict()

data = dict(
  imgs_per_gpu=2,
  workers_per_gpu=2,
  train = [
    dataset_A_train,
    dataset_B_train
  ],
  val = dataset_A_val,
  test = dataset_A_test
  )
```

在测试过程中，这种方式支持分离的方式进行测试

5.3.3 明确定义concat的方式

```
dataset_A_val = dict()
dataset_B_val = dict()

data = dict(
  imgs_per_gpu=2,
  workers_per_gpu=2,
  train=dataset_A_train,
  val=dict(
    type='ConcatDataset',
    datasets=[dataset_A_val, dataset_B_val],
    separate_eval=False))
```

使用`separate_eval=False`在测试验证过程中，将所有的数据集当作以整个数据集进行测试。

注意：

1. 该选项`separate_eval=False`假定数据集`self.data_infos`在评估期间使用。因此，COCO数据集不支持此行为，因为COCO数据集不完全依赖于`self.data_infos`评估。因此，不建议结合使用不同类型的数据集并对其进行整体评估。
2. 不支持评估`ClassBalancedDataset`，`RepeatDataset`因此不支持评估这些类型的串联数据集。

更加复杂的方式，重复两个数据集分别N， M次，使用如下的方式：

```
dataset_A_train = dict(
  type='RepeatDataset',
  times=N,
  dataset=dict(
    type='Dataset_A',
    ...
    pipeline=train_pipeline
  )
)
dataset_A_val = dict(
  ...
  pipeline=test_pipeline
)
dataset_A_test = dict(
```

```
    ...
    pipeline=test_pipeline
)
dataset_B_train = dict(
    type='RepeatDataset',
    times=M,
    dataset=dict(
        type='Dataset_B',
        ...
        pipeline=train_pipeline
    )
)
data = dict(
    imgs_per_gpu=2,
    workers_per_gpu=2,
    train = [
        dataset_A_train,
        dataset_B_train
    ],
    val = dataset_A_val,
    test = dataset_A_test
)
```

6. 调整数据集的类别

可以通过调整数据集的类别来训练数据集的子集，例如现存的数据集为20类， 最终可以调整训练所使用的数据集的类别来仅仅训练其中的三类，mmdetection可以自动滤除其他的类别。

```
classes = ('person', 'bicycle', 'car')
data = dict(
    train=dict(classes=classes),
    val=dict(classes=classes),
    test=dict(classes=classes))
```

mmdetection2.0也支持从文件中读取数据集类别的名称，例如txt文件：

```
person
bicycle
car
```

使用如下的方法进行操作：

```
classes = 'path/to/classes.txt'
data = dict(
    train=dict(classes=classes),
    val=dict(classes=classes),
    test=dict(classes=classes))
```

注意：

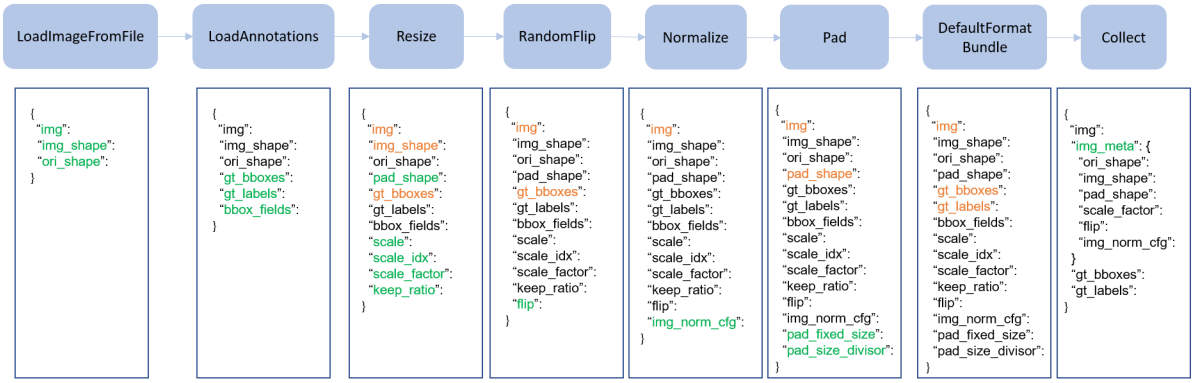
- 在MMDetection v2.5.0之前，如果设置了类别名称，则数据集将自动过滤出空的GT图像，并且无法通过config禁用它。这是不受欢迎的行为，并且会引起混淆，因为如果未设置类别名称，则数据集仅在`filter_empty_gt=True`和时过滤空的GT图像`test_mode=False`。在MMDetection v2.5.0之后，我们将图像过滤过程与类别修改解耦，即，无论是否设置了类别，数据集都只会在`filter_empty_gt=True`和`test_mode=False`时过滤不含GT的图像。因此，设置类别仅会影响用于训练的类别的注释，并且用户可以决定是否自己过滤不含GT的图像。
- 由于中间格式仅具有框标签且不包含类名称，因此在使用时`CustomDataset`，用户无法通过config过滤出空的GT映像，而只能离线进行。

六、mmdetection2.6自定义数据集pipeline

1. data pipeline的基本使用

mmdetection的数据读取方式分为两部分，第一部分为数据集，第二部分为data pipeline，通常数据集定义如何处理标注信息，而pipeline定义处理数据字典的所有步骤，一个pipeline由一系列的操作组成，每一个操作都采用一个dict作为输入，并且也输出一个dict作为下一个操作的输入。

下图是一个经典的pipeline，蓝色的方块是piplline的基本操作，随着pipeline的移动，每个操作都可以加入新的键（绿色的字）作为结果的dict。并且更新已经存在的键值（橘色的字）。



这些操作被分为数据加载，预处理，格式化，测试时增强。

faster rcnn的pipeline的例子：

```
img_norm_cfg = dict(
    mean=[123.675, 116.28, 103.53], std=[58.395, 57.12, 57.375], to_rgb=True)
train_pipeline = [
    dict(type='LoadImageFromFile'),
    dict(type='LoadAnnotations', with_bbox=True),
    dict(type='Resize', img_scale=(1333, 800), keep_ratio=True),
    dict(type='RandomFlip', flip_ratio=0.5),
    dict(type='Normalize', **img_norm_cfg),
    dict(type='Pad', size_divisor=32),
```

```
    dict(type='DefaultFormatBundle'),
    dict(type='Collect', keys=['img', 'gt_bboxes', 'gt_labels']),
]
test_pipeline = [
    dict(type='LoadImageFromFile'),
    dict(
        type='MultiScaleFlipAug',
        img_scale=(1333, 800),
        flip=False,
        transforms=[
            dict(type='Resize', keep_ratio=True),
            dict(type='RandomFlip'),
            dict(type='Normalize', **img_norm_cfg),
            dict(type='Pad', size_divisor=32),
            dict(type='ImageToTensor', keys=['img']),
            dict(type='Collect', keys=['img']),
        ])
]
```

下边列出了各种operation添加，更新与移除的dict行为。

1.1 数据加载

- `LoadImageFromFile`

添加：img，img_shape，ori_shape

- `LoadAnnotations`

添加：gt_bboxes，gt_bboxes_ignore，gt_labels，gt_masks，gt_semantic_seg，bbox_fields，mask_fields

- `LoadProposals`

添加：proposals

1.2 预处理

- `Resize`

添加：scale，scale_idx，pad_shape，scale_factor，keep_ratio
更新：img，img_shape，* bbox_fields，* mask_fields，* seg_fields

- `RandomFlip`

添加：flip
更新：img，* bbox_fields，* mask_fields，* seg_fields

- `Pad`

添加：pad_fixed_size，pad_size_divisor
更新：img，pad_shape，* mask_fields，* seg_fields

- `RandomCrop`

更新：img，pad_shape，gt_bboxes，gt_labels，gt_masks，* bbox_fields

- `Normalize`

添加：img_norm_cfg
更新：img

- `SegRescale`

更新：gt_semantic_seg

- `PhotoMetricDistortion`

更新：img

- `Expand`

更新：img，gt_bboxes

- `MinIoURandomCrop`

更新：img，gt_bboxes，gt_labels

- `Corrupt`

更新：img

1.3 格式化

- `ToTensor`

更新：specified by `keys`.

- `ImageToTensor`

更新：specified by `keys`.

- `Transpose`

更新：specified by `keys`.

- `ToDataContainer`

更新：specified by `fields`.

- `DefaultFormatBundle`

更新：img，proposals，gt_bboxes，gt_bboxes_ignore，gt_labels，gt_masks，gt_semantic_seg

- `Collect`

添加：img_meta（img_meta的键由指定`meta_keys`）
删除：除由所指定的键外的所有其他键

1.4 测试时增强

- `MultiScaleFlipAug`

2. 扩展并使用自定义的pipeline

1. 新建一个python文件，例如： `my_pipeline.py`，输入为一个dict输出也为一个dict

```
from mmdet.datasets import PIPELINES

@PIPELINES.register_module()
class MyTransform:

  def __call__(self, results):
    results['dummy'] = True
    return results
```

2. Import这个新类

```
from .my_pipeline import MyTransform
```

3. 在config文件中使用这个pipeline

```
img_norm_cfg = dict(
  mean=[123.675, 116.28, 103.53], std=[58.395, 57.12, 57.375], to_rgb=True)
train_pipeline = [
  dict(type='LoadImageFromFile'),
  dict(type='LoadAnnotations', with_bbox=True),
  dict(type='Resize', img_scale=(1333, 800), keep_ratio=True),
  dict(type='RandomFlip', flip_ratio=0.5),
  dict(type='Normalize', **img_norm_cfg),
  dict(type='Pad', size_divisor=32),
  dict(type='MyTransform'),
  dict(type='DefaultFormatBundle'),
  dict(type='Collect', keys=['img', 'gt_bboxes', 'gt_labels']),
]
```

七、mmdetection2.6 自定义模型

mmdetection将目标检测模型的基本组件分为五类：

- backbone： 利用全卷积网络来提取特征，例如resnet或者mobilenet
- neck： 在beckbone与head之间的部分，例如FPN与PAFPN
- head：针对特定任务的组成部分，例如：bbox预测，mask预测

- roi extractor： 从feature map中提取roi 特征，例如： ROI Align
- loss： head中计算loss的部分。例如与focal loss 与 GHM loss

每个部分的添加基本上要经历以下几个过程：

定义新的模块——>导入模块——> 在config文件中使用这个模块

1、 添加新的backbone

这里利用mobilenet展示如何开发新的组件。

1.1 定义新的backbone（例如mobilenet）

创建一个新文件 `mmdet/models/backbones/mobilenet.py`.

```
import torch.nn as nn

from ..builder import BACKBONES


@BACKBONES.register_module()
class MobileNet(nn.Module):

  def __init__(self, arg1, arg2):
    pass

  def forward(self, x):  # should return a tuple
    pass

  def init_weights(self, pretrained=None):
    pass
```

1.2 导入模块

两种方法：

1. 加入下边一行代码到 `mmdet/models/backbones/__init__.py` 中。

   ```
   from .mobilenet import MobileNet
   ```

2. 不修改原始框架的代码，在config文件中加入下边的代码：

   ```
   custom_imports = dict(
     imports=['mmdet.models.backbones.mobilenet'],
     allow_failed_imports=False)
   ```

1.3 在config文件中使用backbone

```
model = dict(
  ...
  backbone=dict(
    type='MobileNet',
    arg1=xxx,
    arg2=xxx),
  ...
```

2、 添加新的necks

2.1 定义一个neck

创建一个新的文件 `mmdet/models/necks/pafpn.py` .

```
from ..builder import NECKS

@NECKS.register
class PAFPN(nn.Module):

  def __init__(self,
         in_channels,
         out_channels,
         num_outs,
         start_level=0,
         end_level=-1,
         add_extra_convs=False):
    pass

  def forward(self, inputs):
    # implementation is ignored
    pass
```

2.2 导入这个模块

两种方法：

1. 加入下边一行代码到 `mmdet/models/necks/__init__.py` 中。

   ```
   from .pafpn import PAFPN
   ```

2. 不修改原始框架的代码，在config文件中加入下边的代码：

   ```
   custom_imports = dict(
     imports=['mmdet.models.necks.pafpn'],
     allow_failed_imports=False)
   ```

2.3 在config文件中使用新的neck

```
neck=dict(
  type='PAFPN',
  in_channels=[256, 512, 1024, 2048],
  out_channels=256,
  num_outs=5)
```

3、 添加新的heads

以Double Head R-CNN作为例子来开发一个新的head。

3.1 定义一个head

首先新建一个文件 `mmdet/models/roi_heads/bbox_heads/double_bbox_head.py` ， 并在其中加入一个新的bbox head。Double Head R-CNN采用了一个新型的bbox head来进行 目标检测，为了应用bbox head，需要实现下边所示的三个模块。

```
from mmdet.models.builder import HEADS
from .bbox_head import BBoxHead

@HEADS.register_module()
class DoubleConvFCBBoxHead(BBoxHead):
  r"""Bbox head used in Double-Head R-CNN

                      /-> cls
          /-> shared convs ->
                      \-> reg
  roi features
                      /-> cls
          \-> shared fc    ->
                      \-> reg
  """  # noqa: W605qi

  def __init__(self,
         num_convs=0,
         num_fcs=0,
         conv_out_channels=1024,
         fc_out_channels=1024,
         conv_cfg=None,
         norm_cfg=dict(type='BN'),
         **kwargs):
    kwargs.setdefault('with_avg_pool', True)
    super(DoubleConvFCBBoxHead, self).__init__(**kwargs)

  def init_weights(self):
    # conv layers are already initialized by ConvModule

  def forward(self, x_cls, x_reg):
```

接下来， 如果必要的话，还需要实现ROI head，从 `StandardRoIHead` 继承 `DoubleHeadRoIHead` 。 `standardRoIHead` 已经实现了如下的函数：

```
import torch

from mmdet.core import bbox2result, bbox2roi, build_assigner, build_sampler
from ..builder import HEADS, build_head, build_roi_extractor
from .base_roi_head import BaseRoIHead
from .test_mixins import BBoxTestMixin, MaskTestMixin


@HEADS.register_module()
class StandardRoIHead(BaseRoIHead, BBoxTestMixin, MaskTestMixin):
    """Simplest base roi head including one bbox head and one mask head.
    """

    def init_assigner_sampler(self):

    def init_bbox_head(self, bbox_roi_extractor, bbox_head):

    def init_mask_head(self, mask_roi_extractor, mask_head):

    def init_weights(self, pretrained):

    def forward_dummy(self, x, proposals):


    def forward_train(self,
                      x,
                      img_metas,
                      proposal_list,
                      gt_bboxes,
                      gt_labels,
                      gt_bboxes_ignore=None,
                      gt_masks=None):

    def _bbox_forward(self, x, rois):

    def _bbox_forward_train(self, x, sampling_results, gt_bboxes, gt_labels,
                            img_metas):

    def _mask_forward_train(self, x, sampling_results, bbox_feats, gt_masks,
                            img_metas):

    def _mask_forward(self, x, rois=None, pos_inds=None, bbox_feats=None):


    def simple_test(self,
                    x,
                    proposal_list,
                    img_metas,
                    proposals=None,
                    rescale=False):
        """Test without augmentation."""
```

Double Head主要修改bbox_forward的逻辑部分，并且直接从`StandardRoIHead`继承其他的函数。

在`mmdet/models/roi_heads/double_roi_head.py`中实现新的ROI Head

```
from ..builder import HEADS
from .standard_roi_head import StandardRoIHead


@HEADS.register_module()
class DoubleHeadRoIHead(StandardRoIHead):
  """RoI head for Double Head RCNN

  https://arxiv.org/abs/1904.06493
  """

  def __init__(self, reg_roi_scale_factor, **kwargs):
    super(DoubleHeadRoIHead, self).__init__(**kwargs)
    self.reg_roi_scale_factor = reg_roi_scale_factor

  def _bbox_forward(self, x, rois):
    bbox_cls_feats = self.bbox_roi_extractor(
      x[:self.bbox_roi_extractor.num_inputs], rois)
    bbox_reg_feats = self.bbox_roi_extractor(
      x[:self.bbox_roi_extractor.num_inputs],
      rois,
      roi_scale_factor=self.reg_roi_scale_factor)
    if self.with_shared_head:
      bbox_cls_feats = self.shared_head(bbox_cls_feats)
      bbox_reg_feats = self.shared_head(bbox_reg_feats)
    cls_score, bbox_pred = self.bbox_head(bbox_cls_feats, bbox_reg_feats)

    bbox_results = dict(
      cls_score=cls_score,
      bbox_pred=bbox_pred,
      bbox_feats=bbox_cls_feats)
    return bbox_results
```

3.2 导入这个模块

两种方法：

1. 将实现的模块加入到`mmdet/models/bbox_heads/__init__.py`与`mmdet/models/roi_heads/__init__.py`中

   ```
   from .double_bbox_head import DoubleConvFCBBoxHead
   from .double_roi_head import DoubleHeadRoIHead
   ```

2. 不修改原始框架的代码，在config文件中加入下边的代码：

   ```
   custom_imports=dict(
     imports=['mmdet.models.roi_heads.double_roi_head',
   'mmdet.models.bbox_heads.double_bbox_head'])
   ```

3.3 在config文件中使用新的head

```
#  Double Head R-CNN的config文件
_base_ = '../faster_rcnn/faster_rcnn_r50_fpn_1x_coco.py'
model = dict(
  roi_head=dict(
    type='DoubleHeadRoIHead',
    reg_roi_scale_factor=1.3,
    bbox_head=dict(
      _delete_=True,
      type='DoubleConvFCBBoxHead',
      num_convs=4,
      num_fcs=2,
      in_channels=256,
      conv_out_channels=1024,
      fc_out_channels=1024,
      roi_feat_size=7,
      num_classes=80,
      bbox_coder=dict(
        type='DeltaXYWHBBoxCoder',
        target_means=[0., 0., 0., 0.],
        target_stds=[0.1, 0.1, 0.2, 0.2]),
      reg_class_agnostic=False,
      loss_cls=dict(
        type='CrossEntropyLoss', use_sigmoid=False, loss_weight=2.0),
      loss_bbox=dict(type='SmoothL1Loss', beta=1.0, loss_weight=2.0))))
```

4、 添加新的losses

4.1 定义一个losss

假设定义一个bbox回归的loss 函数`Myloss`, 新建一个文件`mmdet/models/losses/my_loss.py`

```
import torch
import torch.nn as nn

from ..builder import LOSSES
from .utils import weighted_loss

@weighted_loss
def my_loss(pred, target):
  assert pred.size() == target.size() and target.numel() > 0
  loss = torch.abs(pred - target)
  return loss

@LOSSES.register_module()
class MyLoss(nn.Module):
```

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

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

4.2 导入这个模块

两种方法：

1. 加入下边一行代码到 `mmdet/models/losses/__init__.py` 中。

   ```
   from .my_loss import MyLoss, my_loss
   ```

2. 不修改原始框架的代码，在config文件中加入下边的代码：

   ```
   custom_imports=dict(
       imports=['mmdet.models.losses.my_loss'])
   ```

4.3 在config文件中使用新的loss

使用时，针对定义的loss函数的作用，直接修改相应的 `loss_xxx` 字段，这里的myloss是bbox回归，因此修改下边的字段：

```
loss_bbox=dict(type='MyLoss', loss_weight=1.0))
```

八、mmdetection2.6 自定义losses

首先介绍mdmetection的loss的计算pipeline，然后逐步介绍如何自定义loss function。主要的修改包括调整与加权两种。

1、 loss的计算pipeline

给定输入的预测结果与真实值目标，还有loss函数的权重，loss函数将输入tensor映射为最终的loss张量，步骤如下：

1. 按照loss函数的计算得到element-wise或者sample-wise的损失
2. 利用一个权重tensor按照element-wise的方式加权损失函数
3. 将loss的tensor降维为一个标量
4. 加权这个标量损失

2、 调整损失

这种方式主要与上述的1，3，4步骤相关。大多数修改都在config文件中进行，以focal loss为例。

```
# focal loss 的代码定义
@LOSSES.register_module()
class FocalLoss(nn.Module):

  def __init__(self,
          use_sigmoid=True,
          gamma=2.0,
          alpha=0.25,
          reduction='mean',
          loss_weight=1.0):
```

```
# config文件中focal loss函数的使用
loss_cls=dict(
  type='FocalLoss',
  use_sigmoid=True,
  gamma=2.0,
  alpha=0.25,
  loss_weight=1.0)
```

2.1 调整超参数

在focal loss 中gamma与beta是一种超参数。可以在config文件中直接修改

```
loss_cls=dict(
  type='FocalLoss',
  use_sigmoid=True,
  gamma=1.5,
  alpha=0.5,
  loss_weight=1.0)
```

2.2 调整reduction的方式

可选的有`sum`与`mean`，可以直接在config文件中修改

```
loss_cls=dict(
  type='FocalLoss',
  use_sigmoid=True,
  gamma=2.0,
  alpha=0.25,
  loss_weight=1.0,
  reduction='sum')
```

2.3 调整loss的权重

这个权重是一个标量，多任务中控制不同人物的loss函数的权重表示。

```
loss_cls=dict(
  type='FocalLoss',
  use_sigmoid=True,
  gamma=2.0,
  alpha=0.25,
  loss_weight=0.5)
```

3、 加权loss

加权loss意味着按照element-wise的方式重新加权loss值，具体操作就是直接采用与loss唯独相同的权重tensor与loss tensor相乘。这样不同的位置的loss就会被不同程度的放缩。损失权重在不同模型之间有所不同，并且与上下文高度相关，但是总的来说，存在两种损失权重：`label_weights`分类损失和`bbox_weights`bbox回归损失。您可以`get_target`在相应头的方法中找到它们。在这里我们以ATSSHead为例，它继承了AnchorHead，但是覆盖了它的`get_targets`方法，产生了`label_weights`和`bbox_weights`。

```
class ATSSHead(AnchorHead):

  ...

  def get_targets(self,
          anchor_list,
          valid_flag_list,
          gt_bboxes_list,
          img_metas,
          gt_bboxes_ignore_list=None,
          gt_labels_list=None,
          label_channels=1,
          unmap_outputs=True):
```

九、mmdetection2.6 自定义Runtime设置

1、 自定义优化器设置

1.1 自定义pytorch支持的优化器

支持很多的pytorch定义的优化器，唯一需要修改的就是去在`config`文件中修改`optimizer`字段的值。例如如果想要使用`ADAM`，可以使用如下的方式修改：

```
optimizer = dict(type='Adam', lr=0.0003, weight_decay=0.0001)
```

如果想要修改学习率，使用者可以在`optimizer` config文件中修改`lr`。使用者可以直接按照pytorch的API doc 设置arguments。

1.2 自定义优化器

1.2.1 定义新的优化器

假设想要添加`MyOptimizer`的新优化器，有`a`，`b`，`c`三个参数，需要创建一个新的路径`mmdet/core/optimizer`。然后在config文件中应用一个新的优化器`mmdet/core/optimizer/my_optimizer.py`：

```
from .registry import OPTIMIZERS
from torch.optim import Optimizer


@OPTIMIZERS.register_module()
class MyOptimizer(Optimizer):

  def __init__(self, a, b, c)
```

1.2.2 将新的优化器添加到注册器中

上边定义完成的模块想要在config文件中使用，就必须import到一个明明空间中，两种方法，实现这个工作：

- 直接在`__init__.py`文件中，导入，在`mmdet/core/optimizer/__init__.py`中

  ```
  from .my_optimizer import MyOptimizer
  ```

- 在config文件中实现

```
custom_imports = dict(imports=['mmdet.core.optimizer.my_optimizer'],
allow_failed_imports=False)
```

该模块`mmdet.core.optimizer.my_optimizer`将在程序开始时导入，然后自动注册`MyOptimizer`这个类。请注意，仅应该导入包含`MyOptimizer`的包。`mmdet.core.optimizer.my_optimizer.MyOptimizer`**无法**直接导入。

实际上，使用这种导入的方式，使用者可以直接使用不同的文件路径结构，只要模块的路径在`PYTHONPATH`中可以找到即可。

1.2.3 在config文件中指定优化器

在config文件中的`optimizer`字段中使用`MyOptimizer`：

```
# 原始的优化器使用方法
optimizer = dict(type='SGD', lr=0.02, momentum=0.9, weight_decay=0.0001)
# 自定义优化器使用方法jiu
optimizer = dict(type='MyOptimizer', a=a_value, b=b_value, c=c_value)
```

1.3 自定义优化器constructor

一些模型对于优化器的参数有一些特定的设置，例如对于BN层的权重衰减。使用者可以使用自定义优化器的构建器来执行参数的微调。

```
from mmcv.utils import build_from_cfg

from mmcv.runner.optimizer import OPTIMIZER_BUILDERS, OPTIMIZERS
from mmdet.utils import get_root_logger
from .my_optimizer import MyOptimizer


@OPTIMIZER_BUILDERS.register_module()
class MyOptimizerConstructor(object):

  def __init__(self, optimizer_cfg, paramwise_cfg=None):

  def __call__(self, model):

    return my_optimizer
```

默认优化器构建器在这里实现,可以作为一个新的优化器构建器的模板。

1.4 其他的设置

优化器没有实现的技巧，可以通过优化器构建器实现。下边列出了一些常用稳定与加速训练的通用设置。

- 使用梯度裁剪来稳定训练

  ```
  optimizer_config = dict(
    _delete_=True, grad_clip=dict(max_norm=35, norm_type=2))
  ```

  如果config文件继承自base config，那么已经设置了optimizer_config， 可能需要`_delete_=True`来去除不必要的设置。

- 使用动量表加速模型收敛

支持使用动量进度表来一句学习率修改模型的动量，这种方式可以使得模型收敛的更快。动量进程表通常使用LR scheduler，例如接下来的config文件用在3D目标检测中来加速模型的收敛。

```
lr_config = dict(
  policy='cyclic',
  target_ratio=(10, 1e-4),
  cyclic_times=1,
  step_ratio_up=0.4,
)
momentum_config = dict(
  policy='cyclic',
  target_ratio=(0.85 / 0.95, 1),
  cyclic_times=1,
  step_ratio_up=0.4,
)
```

2、 自定义训练schedules

默认情况下，我们使用步长调整的学习率使用`1x`的方式。在MMCV中这称为`StepLRHook`， 我们也支持许多其他的schedule。例如 `CosineAnnealing` and `Poly` schedule。

- Poly schedule:

  ```
  lr_config = dict(policy='poly', power=0.9, min_lr=1e-4, by_epoch=False)
  ```

- ConsineAnnealing schedule:

```
lr_config = dict(
    policy='CosineAnnealing',
    warmup='linear',
    warmup_iters=1000,
    warmup_ratio=1.0 / 10,
    min_lr_ratio=1e-5)
```

3、 自定义workflow

wokflow是一个（phrase， epoch）的列表，使用这个参数来指定运行的顺序与epoches，默认的设置为

```
workflow = [('train', 1)]
```

这意味着训练1epoch， 有事使用者可能想要在验证集上检查某些评价指标，例如loss与准确率，这种情况下，我们可以设置：

```
[('train', 1), ('val', 1)]
```

这样训练一个epoch，验证一个epoch交替执行。

注意：

- 验证时，模型的参数不能更改
- config文件中的total_epochs，仅仅影响训练过程，对于val的过程没有影响。
- Workflows `[('train', 1), ('val', 1)]` and `[('train', 1)]` 不会改变 `EvalHook` 的行为，因为e `EvalHook` 在`after_train_epoch`调用验证的workflow仅仅影响在 `after_val_epoch`中调用的hooks. 因此，`[('train', 1), ('val', 1)]` 和 `[( ‘train’ , 1)]`唯一的不同点就是 runner将会在每个训练过程之后计算验证集的损失。

4、 自定义hooks

4.1 自定义hooks

4.1.1 实现新的hook

mmdetection支持自定义训练过程的hooks，使用者可以简单的修改config文件直接在mmdet或者给予mmdet的自己代码库中应用自己的hook

```
from mmcv.runner import HOOKS, Hook


@HOOKS.register_module()
class MyHook(Hook):

    def __init__(self, a, b):
        pass
```

```
  def before_run(self, runner):
    pass

  def after_run(self, runner):
    pass

  def before_epoch(self, runner):
    pass

  def after_epoch(self, runner):
    pass

  def before_iter(self, runner):
    pass

  def after_iter(self, runner):
    pass
```

基于hook的函数功能，仅仅需要指定在训练过程中这个hook需要完成的功能即可。在`before_run`, `after_run`, `before_epoch`, `after_epoch`, `before_iter`, and `after_iter`这些函数中修改即可。

4.1.2 注册新的hook

然后我们需要将，自定义的`MyHook`导入，假设存在`mmdet/core/utils/my_hook.py`这个文件中，两种方法来导入这个模块：

- 修改 `mmdet/core/utils/__init__.py` 来导入模块

  ```
  from .my_hook import MyHook
  ```

- 在config文件中修改

  ```
  custom_imports = dict(imports=['mmdet.core.utils.my_hook'], allow_failed_imports=False)
  ```

4.1.3 修改config文件

```
custom_hooks = [
  dict(type='MyHook', a=a_value, b=b_value)
]
```

也可以设置hook的优先级通过加入这个键值对 `priority`值为 `'NORMAL'` 或 `'HIGHEST'`：

```
custom_hooks = [
  dict(type='MyHook', a=a_value, b=b_value, priority='NORMAL')
]
```

默认的优先级为 NORMAL

4.2 使用MMCV实现的HOOKs

如果mmcv中存在对应的hook，可以直接使用：

```
custom_hooks = [
   dict(type='MyHook', a=a_value, b=b_value, priority='NORMAL')
]
```

4.3 修改默认运行时的hooks

有一些通用的hook，这些hook没有通过 custom_hooks 注册，这些hooks是：

- log_config
- checkpoint_config
- evaluation
- lr_config
- optimizer_config
- momentum_config

在这些hooks中，只有 log_config 具有 VERY_LOW 的优先级其他的都是 NORMAL 优先级。上面的介绍已经说明了如何修改 optimizer_config , momentum_config 和 lr_config 。这里介绍 log_config , checkpoint_config , and evaluation 这几个hooks。

4.3.1 checkpoint config

mmcv使用 checkpoint_config 来初始化 CheckpointHook 。

```
checkpoint_config = dict(interval=1)
```

使用者可以设置 max_keep_ckpts 来仅仅保存少数的几个checkpoint文件，或者使用 save_optimizer 决定是否保存优化器的状态.

4.3.2 log config

log_config 打包了很多的logger hooks并且能够设置间隔。现在MMCV支持 WandbLoggerHook , MlflowLoggerHook , and TensorboardLoggerHook 。更多细节参考：
https://mmcv.readthedocs.io/en/latest/api.html#mmcv.runner.LoggerHook

```
log_config = dict(
   interval=50,
   hooks=[
      dict(type='TextLoggerHook'),
      dict(type='TensorboardLoggerHook')
   ])
```

4.3.3 evaluation config

这个config文件用来初始化`EvalHook`，除了intervals这个参数之外，其他的变量例如`metric`将会传递给`dataset.evaluate()`

```
evaluation = dict(interval=1, metric='bbox')
```

十、mmdetection2.6 模型微调

mmdetection提供了丰富的预训练模型文件，我们可以直接采用mdmetection提供的预训练模型在我们的数据集上微调。预训练模型参考https://mmdetection.readthedocs.io/en/latest/model_zoo.html,其实也可以在configs文件夹中的readme文件中找到相应的模型链接。

在一个新的数据集上微调模型需要如下的两个步骤：

- 按照mmdetection2.6自定义数据集的方式给框架添加新数据集的支持。
- 修改config文件（这里讨论）

1、 继承base configs

mmdetection2.6采用config的继承模式，来防止写一整个模型的config文件容易造成的编码bug，例如当微调mask rcnn时，新的config文件继承

`_base_/models/mask_rcnn_r50_fpn.py`来构建基本的模型，如果使用Cityscapes数据集，新的config文件需要继承

`_base_/datasets/cityscapes_instance.py`，对于运行时设置，config文件必须继承`_base_/default_runtime.py`。

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

如果不想要继承，也可以自己写所有的config文件。

2、 修改head

简单修改`num_classes`即可。

```
model = dict(
  pretrained=None,
  roi_head=dict(
    bbox_head=dict(
      type='Shared2FCBBoxHead',
      in_channels=256,
      fc_out_channels=1024,
```

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

3、修改数据集

按照[mmdetection2.6自定义数据集]的方式给框架添加新数据集的支持。

4、 修改training schedule

直接在config文件中共修改即可

```
# optimizer
# lr is set for a batch size of 8
optimizer = dict(type='SGD', lr=0.01, momentum=0.9, weight_decay=0.0001)
optimizer_config = dict(grad_clip=None)
# learning policy
lr_config = dict(
  policy='step',
  warmup='linear',
  warmup_iters=500,
  warmup_ratio=0.001,
  # [7] yields higher performance than [6]
  step=[7])
total_epochs = 8  # actual epoch = 8 * 8 = 64
log_config = dict(interval=100)
```

5、 使用预训练模型

修改config文件中的`load_from`参数即可

```
load_from = '下载的权重的保存路径，也可以直接放上官方的urls'
```

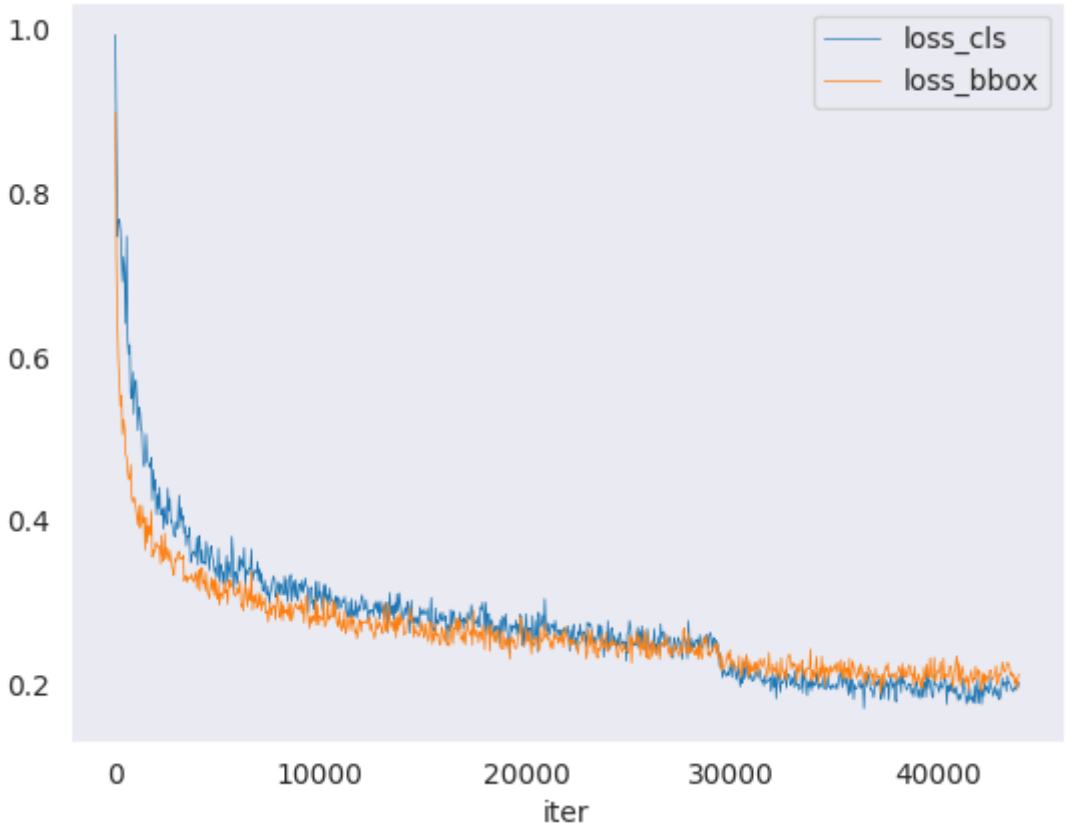
十一、mmdetection2.6 有用的工具

在tools文件夹下， 除了训练与测试的script，也提供了很多的其他的有用工具。

1. 日志分析

`tools/analyze_logs.py` 利用给定的训练日志文件，可以打印出loss函数与map的曲线，运行`pip install serborn`来安装对应的依赖。

```
python tools/analyze_logs.py plot_curve [--keys ${KEYS}] [--title ${TITLE}] [--legend ${LEGEND}] [--backend ${BACKEND}] [--style ${STYLE}] [--out ${OUT_FILE}]
```



Examples:

- 打印分类损失

  ```
  python tools/analyze_logs.py plot_curve log.json --keys loss_cls --legend loss_cls
  ```

- 打印分类与回归的损失，并保存为一个pdf文件

  ```
  python tools/analyze_logs.py plot_curve log.json --keys loss_cls loss_bbox --out losses.pdf
  ```

- 在同一个图中对比两个runs的map

  ```
  python tools/analyze_logs.py plot_curve log1.json log2.json --keys bbox_mAP --legend run1 run2
  ```

- 计算平均训练速度

```
python tools/analyze_logs.py cal_train_time log.json [--include-outliers]
```

训练速度的计算结果输出如下所示：.

```
-----Analyze train time of work_dirs/some_exp/20190611_192040.log.json-----
slowest epoch 11, average time is 1.2024
fastest epoch 1, average time is 1.1909
time std over epochs is 0.0028
average iter time: 1.1959 s/iter
```

2. 可视化

2.1 可视化数据集

`tools/browse_dataset.py` 这个脚本程序，用来帮助使用者来浏览目标检测的数据集，包括图片与bbox，或者保存图片到指定的目录。

```
python tools/browse_dataset.py ${CONFIG} [-h] [--skip-type ${SKIP_TYPE[SKIP_TYPE...]}] [--output-dir ${OUTPUT_DIR}] [--not-show] [--show-interval ${SHOW_INTERVAL}]
```

2.2 可视化模型

首先按照这里的描述转换模型为ONNX，现在仅仅RetainNet支持这种操作，其他的模型将在后来的版本中逐步支持，然后使用Netron工具来实现可视化。

2.3 可视化预测结果

如果需要轻量化的GUI来可视化检测结果，可以参考DetVisGUI project。

3. 误差分析

`tools/coco_error_analysis.py` 利用不同的标准分析了每一类的结果，也可以绘图来提供有用的信息。

```
python tools/coco_error_analysis.py ${RESULT} ${OUT_DIR} [-h] [--ann ${ANN}] [--types ${TYPES[TYPES...]}]
```

4. 模型复杂度

`tools/get_flops.py` 是一个利用 flops-counter.pytorch 来计算FLOPS与参数的方法。

```
python tools/get_flops.py ${CONFIG_FILE} [--shape ${INPUT_SHAPE}]
```

得到的最终结果如下图所示：

```
==============================
Input shape: (3, 1280, 800)
Flops: 239.32 GFLOPs
Params: 37.74 M
==============================
```

注意：这个工具依然在实验阶段，并且不能够保证结果完全正确。可以使用这个结果进行简单的比较，但是在论文或者技术报告中，使用这个结果时必须要好好检查一下。

- FLOPS与输入图像的代销相关，然而模型参数没有关系，默认的输入尺寸是（1， 3， 1280, 800）
- 一些自定义的操作例如GN等没有算在FLOPS中。更多信息参考 mmcv.cnn.get_model_complexity_info()
- 两步法的FLOPS依赖于proposal的数目。

5. 模型转换

5.1 mmdetection模型转换为ONNX（实验阶段）

mmdetection提供了一个脚本将模型转换为ONNX。

```
python tools/pytorch2onnx.py ${CONFIG_FILE} ${CHECKPOINT_FILE} --output_file ${ONNX_FILE} [--shape ${INPUT_SHAPE} --verify]
```

现在这个工具依然在实验阶段，好多自定义的操作不支持。

5.2 mmdetection1.x版本的模型转换为mmdetection2.x

`tools/upgrade_model_version.py`更新先前版本的checkpoint权重为新的版本。不保证全部实现。

```
python tools/upgrade_model_version.py ${IN_FILE} ${OUT_FILE} [-h] [--num-classes NUM_CLASSES]
```

5.3 RegNet模型转换为mmdetection

`tools/regnet2mmdet.py`将pycls预训练的regnet模型转换为mmdetection风格的模型。

```
python tools/regnet2mmdet.py ${SRC} ${DST} [-h]
```

5.4 Detectron resnet转换为Pytorch

`tools/detectron2pytorch.py`将原始detectron框架的resnet预训练模型转换为pytorch风格的模型。

```
python tools/detectron2pytorch.py ${SRC} ${DST} ${DEPTH} [-h]
```

5.5 准备一个待发布的模型

`tools/publish_model.py`帮助使用者准备自己的待发布的模型。

在你讲模型传到AWS之前，你可能想要：

- 将模型的权重转换为cpu tensor
- 删除优化器的状态
- 计算checkpoint文件的的哈希值，并且将hash id加到文件名后边

```
python tools/publish_model.py ${INPUT_FILENAME} ${OUTPUT_FILENAME}
```

例子：

```
python tools/publish_model.py work_dirs/faster_rcnn/latest.pth faster_rcnn_r50_fpn_1x_20190801.pth
```

最终的输出名字为`faster_rcnn_r50_fpn_1x_20190801-{hash id}.pth`

6. 数据集转换

`tools/convert_datasets/`提供了工具来转换Cityscapes数据集与Pascal VOC数据集为COCO格式的数据集。

```
python tools/convert_datasets/cityscapes.py ${CITYSCAPES_PATH} [-h] [--img-dir ${IMG_DIR}] [--gt-dir ${GT_DIR}] [-o ${OUT_DIR}] [--nproc ${NPROC}]
python tools/convert_datasets/pascal_voc.py ${DEVKIT_PATH} [-h] [-o ${OUT_DIR}]
```

7. 其他

7.1 评估标准

`tools/eval_metric.py`按照config文件评估某个结果pickle文件。

```
python tools/eval_metric.py ${CONFIG} ${PKL_RESULTS} [-h] [--format-only] [--eval ${EVAL[EVAL ...]}]
                            [--cfg-options ${CFG_OPTIONS [CFG_OPTIONS ...]}]
                            [--eval-options ${EVAL_OPTIONS [EVAL_OPTIONS ...]}]
```

7.2 打印整个config文件

`tools/print_config.py`打印整个config文件，包括import的内容

```
python tools/print_config.py ${CONFIG} [-h] [--options ${OPTIONS [OPTIONS...]}]
```

7.3 测试模型的鲁棒性

参考robustness_benchmarking.md.

专栏所有文章请点击下列文章列表查看：

知乎专栏：小哲AI专栏文章分类索引跳转查看

AI研习社专栏：小哲AI专栏文章分类索引

专栏所有文章都可以在github仓库中找到pdf版本：https://github.com/lxztju/notes