MMSegmentation

发布 *1.0.0rc0*

MMSegmentation Authors

2022 年 09 月 30 日

开始你的第一步

CHAPTER 1

开始（待更新）

1.1 依赖

- Linux or macOS (Windows 下支持需要 mmcv-full，但运行时可能会有一些问题。)
- Python 3.6+
- PyTorch 1.3+
- CUDA 9.2+ (如果您基于源文件编译 PyTorch, CUDA 9.0 也可以使用)
- GCC 5+
- MMCV

可编译的 MMSegmentation 和 MMCV 版本如下所示，请对照对应版本安装以避免安装问题。

注意: 如果您已经安装好 mmcv，您首先需要运行 `pip uninstall mmcv`。如果 mmcv 和 mmcv-full 同时被安装，会报错 `ModuleNotFoundError`。

1.2 安装

a. 创建一个 conda 虚拟环境并激活它

```
conda create -n open-mmlab python=3.10 -y
conda activate open-mmlab
```

b. 按照官方教程 安装 PyTorch 和 totchvision，这里我们使用 PyTorch1.11.0 和 CUDA11.3，您也可以切换至其他版本

```
conda install pytorch=1.11.0 torchvision cudatoolkit=11.3 -c pytorch
```

c. 按照 官方教程 安装 MMCV ，mmcv 或 mmcv-full 和 MMSegmentation 均兼容，但对于 CCNet 和 PSANet，mmcv-full 里的 CUDA 运算是必须的

在 Linux 下安装 mmcv：

为了安装 MMCV, 我们推荐使用下面的这种预编译好的 MMCV.

```
pip install mmcv-full -f https://download.openmmlab.com/mmcv/dist/{cu_version}/{torch_
↪version}/index.html
```

请替换 url 里面的 {cu_version} 和 {torch_version} 为您想要使用的版本. mmcv-full 仅在 PyTorch 1.x.0 上面编译, 因为在 1.x.0 和 1.x.1 之间通常是兼容的. 如果您的 PyTorch 版本是 1.x.1, 您可以安装用 PyTorch 1.x.0 编译的 mmcv-full 而它通常是可以正常使用的. 例如, 用 CUDA 11.1 and PyTorch 1.11.0 安装使用 mmcv-full, 使用如下命令:

```
pip install mmcv-full -f https://download.openmmlab.com/mmcv/dist/cu113/torch1.11/
↪index.html
```

请查看 这里 来找到适配不同 PyTorch 和 CUDA 版本的 MMCV.

您也可以采用下面的命令来从源码编译 MMCV (可选)

```
git clone https://github.com/open-mmlab/mmcv.git
cd mmcv
MMCV_WITH_OPS=1 pip install -e .  # package mmcv-full, which contains cuda ops, will␣
↪be installed after this step
# OR pip install -e .  # package mmcv, which contains no cuda ops, will be installed␣
↪after this step
cd ..
```

重点: 如果您已经安装了 MMCV, 您需要先运行 pip uninstall mmcv. 因为如果 mmcv 和 mmcv-full 被同时安装, 将会报错 ModuleNotFoundError.

在 Windows 下安装 mmcv (有风险)：

对于 Windows，MMCV 的安装需要本地 C++ 编译工具，例如 cl.exe。请添加编译工具至%PATH%。

如果您已经在电脑上安装好 Windows SDK 和 Visual Studio，cl.exe 的一个典型路径看起来如下：

```
C:\Program Files (x86)\Microsoft Visual Studio\2019\Professional\VC\Tools\MSVC\14.26.
↪28801\bin\Hostx86\x64
```

或者您需要从网上下载 cl 编译工具并安装至路径。

随后，从 github 克隆 mmcv 并通过 pip 安装：

```
git clone https://github.com/open-mmlab/mmcv.git
cd mmcv
pip install -e .
```

或直接:

```
pip install mmcv
```

当前，mmcv-full 并不完全在 windows 上支持。

d. 安装 MMSegmentation

```
pip install mmsegmentation # 安装最新版本
```

或者

```
pip install git+https://github.com/open-mmlab/mmsegmentation.git # 安装 master 分支
```

此外，如果您想安装 dev 模式的 MMSegmentation, 运行如下命令：

```
git clone https://github.com/open-mmlab/mmsegmentation.git
cd mmsegmentation
pip install -e .  # 或者 "python setup.py develop"
```

注意:

1. 当在 windows 下训练和测试模型时，请确保路径下所有的’\’被替换成‘/’，在 python 代码里可以使用`.replace('\\', '/')`处理路径的字符串
2. `version+git_hash` 也将被保存进 meta 训练模型里，即 0.5.0+c415a2e
3. 当 MMsegmentation 以 `dev` 模式被安装时，本地对代码的修改将不需要重新安装即可产生作用
4. 如果您想使用 `opencv-python-headless` 替换 `opencv-python`，您可以在安装 MMCV 前安装它
5. 一些依赖项是可选的。简单的运行 `pip install -e .` 将仅安装最必要的一些依赖。为了使用可选的依赖项如 `cityscapessripts`，要么手动使用 `pip install -r requirements/optional.txt` 安装，要么专门从 pip 下安装 (即 `pip install -e .[optional]`，其中选项可设置为 `all`, `tests`, `build`, 和 `optional`)

1.2.1 完整的安装脚本

Linux

这里便是一个完整安装 MMSegmentation 的脚本，使用 conda 并链接了数据集的路径（以您的数据集路径为 $DATA_ROOT 来安装）。

```
conda create -n open-mmlab python=3.10 -y
conda activate open-mmlab

conda install pytorch=1.11.0 torchvision cudatoolkit=11.3 -c pytorch
pip install mmcv-full -f https://download.openmmlab.com/mmcv/dist/cu113/torch1.11.0/
↪index.html
git clone https://github.com/open-mmlab/mmsegmentation.git
cd mmsegmentation
pip install -e .  # 或者 "python setup.py develop"

mkdir data
ln -s $DATA_ROOT data
```

Windows (有风险)

这里便是一个完整安装 MMSegmentation 的脚本，使用 conda 并链接了数据集的路径（以您的数据集路径为%DATA_ROOT% 来安装）。注意：它必须是一个绝对路径。

```
conda create -n open-mmlab python=3.10 -y
conda activate open-mmlab

conda install pytorch=1.11.0 torchvision cudatoolkit=11.3 -c pytorch
set PATH=full\path\to\your\cpp\compiler;%PATH%
pip install mmcv

git clone https://github.com/open-mmlab/mmsegmentation.git
cd mmsegmentation
pip install -e .  # 或者 "python setup.py develop"

mklink /D data %DATA_ROOT%
```

使用多版本 MMSegmentation 进行开发

训练和测试脚本已经修改了 PYTHONPATH 来确保使用当前路径的 MMSegmentation。

为了使用当前环境默认安装的 MMSegmentation 而不是正在工作的 MMSegmentation，您可以在那些脚本里移除下面的内容：

```
PYTHONPATH="$(dirname $0)/..":$PYTHONPATH
```

1.3 验证

为了验证 MMSegmentation 和它所需要的环境是否正确安装，我们可以使用样例 python 代码来初始化一个 segmentor 并推理一张 demo 图像。

```
from mmseg.apis import inference_model, init_model
import mmcv

config_file = 'configs/pspnet/pspnet_r50-d8_512x1024_40k_cityscapes.py'
checkpoint_file = 'checkpoints/pspnet_r50-d8_512x1024_40k_cityscapes_20200605_003338-
↪2966598c.pth'

# 从一个 config 配置文件和 checkpoint 文件里创建分割模型
model = init_model(config_file, checkpoint_file, device='cuda:0')

# 测试一张样例图片并得到结果
img = 'test.jpg'  # 或者 img = mmcv.imread(img), 这将只加载图像一次.
result = inference_model(model, img)
# 在新的窗口里可视化结果
model.show_result(img, result, show=True)
# 或者保存图片文件的可视化结果
# 您可以改变 segmentation map 的不透明度 (opacity)，在 (0, 1] 之间。
model.show_result(img, result, out_file='result.jpg', opacity=0.5)

# 测试一个视频并得到分割结果
video = mmcv.VideoReader('video.mp4')
for frame in video:
   result = inference_model(model, frame)
   model.show_result(frame, result, wait_time=1)
```

当您完成 MMSegmentation 的安装时，上述代码应该可以成功运行。

我们还提供一个 demo 脚本去可视化单张图片。

```
python demo/image_demo.py ${IMAGE_FILE} ${CONFIG_FILE} ${CHECKPOINT_FILE} [--device $
↪{DEVICE_NAME}] [--palette-thr ${PALETTE}]
```

样例：

```
python demo/image_demo.py demo/demo.jpg configs/pspnet/pspnet_r50-d8_512x1024_40k_
↪cityscapes.py \
    checkpoints/pspnet_r50-d8_512x1024_40k_cityscapes_20200605_003338-2966598c.pth --
↪device cuda:0 --palette cityscapes
```

推理的 demo 文档可在此查询：demo/inference_demo.ipynb 。

CHAPTER 2

训练 & 测试

2.1 学习配置文件（待更新）

我们整合了模块和继承设计到我们的配置里，这便于做很多实验。如果您想查看配置文件，您可以运行 `python tools/print_config.py /PATH/TO/CONFIG` 去查看完整的配置文件。您还可以传递参数 `--cfg-options xxx.yyy=zzz` 去查看更新的配置。

2.1.1 配置文件的结构

在 `config/_base_` 文件夹下面有 4 种基本组件类型：数据集 (dataset)，模型 (model)，训练策略 (schedule) 和运行时的默认设置 (default runtime)。许多方法都可以方便地通过组合这些组件进行实现。这样，像 DeepLabV3, PSPNet 这样的模型可以容易地被构造。被来自 `_base_` 下的组件来构建的配置叫做 *原始配置 (primitive)*。

对于所有在同一个文件夹下的配置文件，推荐**只有一个**对应的**原始配置**文件。所有其他的配置文件都应该继承自这个**原始配置**文件。这样就能保证配置文件的最大继承深度为 3。

为了便于理解，我们推荐社区贡献者继承已有的方法配置文件。例如，如果一些修改是基于 DeepLabV3，使用者首先首先应该通过指定 `_base_ = ../deeplabv3/deeplabv3_r50_512x1024_40ki_cityscapes.py` 来继承基础 DeepLabV3 结构，再去修改配置文件里其他内容以完成继承。

如果您正在构建一个完整的新模型，它完全没有和已有的方法共享一些结构，您可能需要在 `configs` 下面创建一个文件夹 `xxxnet`。更详细的文档，请参照 mmcv 。

2.1.2 配置文件命名风格

我们按照下面的风格去命名配置文件，社区贡献者被建议使用同样的风格。

```
{model}_{backbone}_[misc]_[gpu x batch_per_gpu]_{resolution}_{iterations}_{dataset}
```

`{xxx}` 是被要求的文件 `[yyy]` 是可选的。

- `{model}`: 模型种类，例如 `psp`，`deeplabv3` 等等
- `{backbone}`: 主干网络种类，例如 `r50` (ResNet-50)，`x101` (ResNeXt-101)
- `[misc]`: 模型中各式各样的设置/插件，例如 `dconv`，`gcb`，`attention`，`mstrain`
- `[gpu x batch_per_gpu]`: GPU 数目和每个 GPU 的样本数，默认为 `8x2`
- `{iterations}`: 训练迭代轮数，如 `160k`
- `{dataset}`: 数据集，如 `cityscapes`，`voc12aug`，`ade`

2.1.3 PSPNet 的一个例子

为了帮助使用者熟悉这个流行的语义分割框架的完整配置文件和模块，我们在下面对使用 ResNet50V1c 的 PSPNet 的配置文件做了详细的注释说明。更多的详细使用和其他模块的替代项请参考 API 文档。

```
norm_cfg = dict(type='SyncBN', requires_grad=True)  # 分割框架通常使用 SyncBN
model = dict(
    type='EncoderDecoder',  # 分割器 (segmentor) 的名字
    pretrained='open-mmlab://resnet50_v1c',  # 将被加载的 ImageNet 预训练主干网络
    backbone=dict(
        type='ResNetV1c',  # 主干网络的类别。 可用选项请参考 mmseg/models/backbones/resnet.
↪py
        depth=50,  # 主干网络的深度。通常为 50 和 101。
        num_stages=4,  # 主干网络状态 (stages) 的数目，这些状态产生的特征图作为后续的 head 的输
入。
        out_indices=(0, 1, 2, 3),  # 每个状态产生的特征图输出的索引。
        dilations=(1, 1, 2, 4),  # 每一层 (layer) 的空心率 (dilation rate)。
        strides=(1, 2, 1, 1),  # 每一层 (layer) 的步长 (stride)。
        norm_cfg=dict(  # 归一化层 (norm layer) 的配置项。
            type='SyncBN',  # 归一化层的类别。通常是 SyncBN。
            requires_grad=True),  # 是否训练归一化里的 gamma 和 beta。
        norm_eval=False,  # 是否冻结 BN 里的统计项。
        style='pytorch',  # 主干网络的风格，'pytorch' 意思是步长为 2 的层为 3x3 卷积， 'caffe
↪' 意思是步长为 2 的层为 1x1 卷积。
        contract_dilation=True),  # 当空洞 > 1，是否压缩第一个空洞层。
    decode_head=dict(
        type='PSPHead',  # 解码头 (decode head) 的类别。 可用选项请参考 mmseg/models/
↪decode_heads。
```

(下页继续)

(续上页)

```
        in_channels=2048,  # 解码头的输入通道数。
        in_index=3,  # 被选择的特征图 (feature map) 的索引。
        channels=512,  # 解码头中间态 (intermediate) 的通道数。
        pool_scales=(1, 2, 3, 6),  # PSPHead 平均池化 (avg pooling) 的规模 (scales)。 细节
请参考文章内容。
        dropout_ratio=0.1,  # 进入最后分类层 (classification layer) 之前的 dropout 比例。
        num_classes=19,  # 分割前景的种类数目。 通常情况下, cityscapes 为 19, VOC 为 21,
ADE20k 为 150。
        norm_cfg=dict(type='SyncBN', requires_grad=True),  # 归一化层的配置项。
        align_corners=False,  # 解码里调整大小 (resize) 的 align_corners 参数。
        loss_decode=dict(  # 解码头 (decode_head) 里的损失函数的配置项。
            type='CrossEntropyLoss',  # 在分割里使用的损失函数的类别。
            use_sigmoid=False,  # 在分割里是否使用 sigmoid 激活。
            loss_weight=1.0)),  # 解码头里损失的权重。
    auxiliary_head=dict(
        type='FCNHead',  # 辅助头 (auxiliary head) 的种类。可用选项请参考 mmseg/models/
↪decode_heads。
        in_channels=1024,  # 辅助头的输入通道数。
        in_index=2,  # 被选择的特征图 (feature map) 的索引。
        channels=256,  # 辅助头中间态 (intermediate) 的通道数。
        num_convs=1,  # FCNHead 里卷积 (convs) 的数目. 辅助头里通常为 1。
        concat_input=False,  # 在分类层 (classification layer) 之前是否连接 (concat) 输入和
卷积的输出。
        dropout_ratio=0.1,  # 进入最后分类层 (classification layer) 之前的 dropout 比例。
        num_classes=19,  # 分割前景的种类数目。 通常情况下, cityscapes 为 19, VOC 为 21,
ADE20k 为 150。
        norm_cfg=dict(type='SyncBN', requires_grad=True),  # 归一化层的配置项。
        align_corners=False,  # 解码里调整大小 (resize) 的 align_corners 参数。
        loss_decode=dict(  # 辅助头 (auxiliary head) 里的损失函数的配置项。
            type='CrossEntropyLoss',  # 在分割里使用的损失函数的类别。
            use_sigmoid=False,  # 在分割里是否使用 sigmoid 激活。
            loss_weight=0.4)))  # 辅助头里损失的权重。默认设置为 0.4。
train_cfg = dict()  # train_cfg 当前仅是一个占位符。
test_cfg = dict(mode='whole')  # 测试模式, 选项是 'whole' 和 'sliding'. 'whole': 整张图像
全卷积 (fully-convolutional) 测试。 'sliding': 图像上做滑动裁剪窗口 (sliding crop window)。
dataset_type = 'CityscapesDataset'  # 数据集类型, 这将被用来定义数据集。
data_root = 'data/cityscapes/'  # 数据的根路径。
img_norm_cfg = dict(  # 图像归一化配置, 用来归一化输入的图像。
    mean=[123.675, 116.28, 103.53],  # 预训练里用于预训练主干网络模型的平均值。
    std=[58.395, 57.12, 57.375],  # 预训练里用于预训练主干网络模型的标准差。
    to_rgb=True)  # 预训练里用于预训练主干网络的图像的通道顺序。
crop_size = (512, 1024)  # 训练时的裁剪大小
train_pipeline = [  # 训练流程
```

(下页继续)

(续上页)

```
    dict(type='LoadImageFromFile'),  # 第 1 个流程，从文件路径里加载图像。
    dict(type='LoadAnnotations'),  # 第 2 个流程，对于当前图像，加载它的注释信息。
    dict(type='Resize',  # 变化图像和其注释大小的数据增广的流程。
        img_scale=(2048, 1024),  # 图像的最大规模。
        ratio_range=(0.5, 2.0)), # 数据增广的比例范围。
    dict(type='RandomCrop',  # 随机裁剪当前图像和其注释大小的数据增广的流程。
        crop_size=(512, 1024),  # 随机裁剪图像生成 patch 的大小。
        cat_max_ratio=0.75),  # 单个类别可以填充的最大区域的比例。
    dict(
        type='RandomFlip',  # 翻转图像和其注释大小的数据增广的流程。
        flip_ratio=0.5),  # 翻转图像的概率
    dict(type='PhotoMetricDistortion'),  # 光学上使用一些方法扭曲当前图像和其注释的数据增广的流
程。
    dict(
        type='Normalize',  # 归一化当前图像的数据增广的流程。
        mean=[123.675, 116.28, 103.53],  # 这些键与 img_norm_cfg 一致，因为 img_norm_cfg␣
↪被
        std=[58.395, 57.12, 57.375],  # 用作参数。
        to_rgb=True),
    dict(type='Pad',  # 填充当前图像到指定大小的数据增广的流程。
        size=(512, 1024),  # 填充的图像大小。
        pad_val=0,  # 图像的填充值。
        seg_pad_val=255),  # 'gt_semantic_seg' 的填充值。
    dict(type='DefaultFormatBundle'),  # 流程里收集数据的默认格式捆。
    dict(type='Collect',  # 决定数据里哪些键被传递到分割器里的流程。
        keys=['img', 'gt_semantic_seg'])
]
test_pipeline = [
    dict(type='LoadImageFromFile'),  # 第 1 个流程，从文件路径里加载图像。
    dict(
        type='MultiScaleFlipAug',  # 封装测试时数据增广 (test time augmentations)。
        img_scale=(2048, 1024),  # 决定测试时可改变图像的最大规模。用于改变图像大小的流程。
        flip=False,  # 测试时是否翻转图像。
        transforms=[
            dict(type='Resize',  # 使用改变图像大小的数据增广。
                 keep_ratio=True),  # 是否保持宽和高的比例，这里的图像比例设置将覆盖上面的图像规
模大小的设置。
            dict(type='RandomFlip'),  # 考虑到 RandomFlip 已经被添加到流程里，当␣
↪flip=False 时它将不被使用。
            dict(
                type='Normalize',  # 归一化配置项，值来自 img_norm_cfg。
                mean=[123.675, 116.28, 103.53],
                std=[58.395, 57.12, 57.375],
```

(下页继续)

(续上页)

```
                to_rgb=True),
            dict(type='ImageToTensor', # 将图像转为张量
                keys=['img']),
            dict(type='Collect', # 收集测试时必须的键的收集流程。
                keys=['img'])
        ])
]
data = dict(
    samples_per_gpu=2,  # 单个 GPU 的 Batch size
    workers_per_gpu=2,  # 单个 GPU 分配的数据加载线程数
    train=dict(  # 训练数据集配置
        type='CityscapesDataset',  # 数据集的类别, 细节参考自 mmseg/datasets/。
        data_root='data/cityscapes/',  # 数据集的根目录。
        img_dir='leftImg8bit/train',  # 数据集图像的文件夹。
        ann_dir='gtFine/train',  # 数据集注释的文件夹。
        pipeline=[  # 流程, 由之前创建的 train_pipeline 传递进来。
            dict(type='LoadImageFromFile'),
            dict(type='LoadAnnotations'),
            dict(
                type='Resize', img_scale=(2048, 1024), ratio_range=(0.5, 2.0)),
            dict(type='RandomCrop', crop_size=(512, 1024), cat_max_ratio=0.75),
            dict(type='RandomFlip', flip_ratio=0.5),
            dict(type='PhotoMetricDistortion'),
            dict(
                type='Normalize',
                mean=[123.675, 116.28, 103.53],
                std=[58.395, 57.12, 57.375],
                to_rgb=True),
            dict(type='Pad', size=(512, 1024), pad_val=0, seg_pad_val=255),
            dict(type='DefaultFormatBundle'),
            dict(type='Collect', keys=['img', 'gt_semantic_seg'])
        ]),
    val=dict(  # 验证数据集的配置
        type='CityscapesDataset',
        data_root='data/cityscapes/',
        img_dir='leftImg8bit/val',
        ann_dir='gtFine/val',
        pipeline=[  # 由之前创建的 test_pipeline 传递的流程。
            dict(type='LoadImageFromFile'),
            dict(
                type='MultiScaleFlipAug',
                img_scale=(2048, 1024),
                flip=False,
```

(下页继续)

(续上页)

```
                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='ImageToTensor', keys=['img']),
                    dict(type='Collect', keys=['img'])
                ])
        ]),
    test=dict(
        type='CityscapesDataset',
        data_root='data/cityscapes/',
        img_dir='leftImg8bit/val',
        ann_dir='gtFine/val',
        pipeline=[
            dict(type='LoadImageFromFile'),
            dict(
                type='MultiScaleFlipAug',
                img_scale=(2048, 1024),
                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='ImageToTensor', keys=['img']),
                    dict(type='Collect', keys=['img'])
                ])
        ]))
log_config = dict(  # 注册日志钩 (register logger hook) 的配置文件。
    interval=50,  # 打印日志的间隔
    hooks=[
        # dict(type='TensorboardLoggerHook')  # 同样支持 Tensorboard 日志
        dict(type='TextLoggerHook', by_epoch=False)
    ])
dist_params = dict(backend='nccl')  # 用于设置分布式训练的参数，端口也同样可被设置。
log_level = 'INFO'  # 日志的级别。
```

(下页继续)

(续上页)

```
load_from = None  # 从一个给定路径里加载模型作为预训练模型，它并不会消耗训练时间。
resume_from = None  # 从给定路径里恢复检查点 (checkpoints)，训练模式将从检查点保存的轮次开始恢复
训练。
workflow = [('train', 1)]  # runner 的工作流程。 [('train', 1)] 意思是只有一个工作流程而且工作
流程 'train' 仅执行一次。根据 `runner.max_iters` 工作流程训练模型的迭代轮数为 40000 次。
cudnn_benchmark = True  # 是否是使用 cudnn_benchmark 去加速，它对于固定输入大小的可以提高训练速
度。
optimizer = dict(  # 用于构建优化器的配置文件。支持 PyTorch 中的所有优化器，同时它们的参数与
PyTorch 里的优化器参数一致。
    type='SGD',  # 优化器种类，更多细节可参考 https://github.com/open-mmlab/mmcv/blob/
↪master/mmcv/runner/optimizer/default_constructor.py#L13。
    lr=0.01,  # 优化器的学习率，参数的使用细节请参照对应的 PyTorch 文档。
    momentum=0.9,  # 动量 (Momentum)
    weight_decay=0.0005)  # SGD 的衰减权重 (weight decay)。
optimizer_config = dict()  # 用于构建优化器钩 (optimizer hook) 的配置文件，执行细节请参考␣
↪https://github.com/open-mmlab/mmcv/blob/master/mmcv/runner/hooks/optimizer.py#L8。
lr_config = dict(
    policy='poly',  # 调度流程的策略，同样支持 Step, CosineAnnealing, Cyclic 等. 请从␣
↪https://github.com/open-mmlab/mmcv/blob/master/mmcv/runner/hooks/lr_updater.py#L9 参
考 LrUpdater 的细节。
    power=0.9,  # 多项式衰减 (polynomial decay) 的幂。
    min_lr=0.0001,  # 用来稳定训练的最小学习率。
    by_epoch=False)  # 是否按照每个 epoch 去算学习率。
runner = dict(
    type='IterBasedRunner', # 将使用的 runner 的类别 (例如 IterBasedRunner 或␣
↪EpochBasedRunner)。
    max_iters=40000) # 全部迭代轮数大小，对于 EpochBasedRunner 使用 `max_epochs` 。
checkpoint_config = dict(  # 设置检查点钩子 (checkpoint hook) 的配置文件。执行时请参考␣
↪https://github.com/open-mmlab/mmcv/blob/master/mmcv/runner/hooks/checkpoint.py。
    by_epoch=False,  # 是否按照每个 epoch 去算 runner。
    interval=4000)  # 保存的间隔
evaluation = dict(  # 构建评估钩 (evaluation hook) 的配置文件。细节请参考 mmseg/core/
↪evaluation/eval_hook.py。
    interval=4000,  # 评估的间歇点
    metric='mIoU')  # 评估的指标
```

2.1.4 FAQ

忽略基础配置文件里的一些域内容。

有时，您也许会设置 `_delete_=True` 去忽略基础配置文件里的一些域内容。您也许可以参照 mmcv 来获得一些简单的指导。

在 MMSegmentation 里，例如为了改变 PSPNet 的主干网络的某些内容：

```
norm_cfg = dict(type='SyncBN', requires_grad=True)
model = dict(
    type='MaskRCNN',
    pretrained='torchvision://resnet50',
    backbone=dict(
        type='ResNetV1c',
        depth=50,
        num_stages=4,
        out_indices=(0, 1, 2, 3),
        dilations=(1, 1, 2, 4),
        strides=(1, 2, 1, 1),
        norm_cfg=norm_cfg,
        norm_eval=False,
        style='pytorch',
        contract_dilation=True),
    decode_head=dict(...),
    auxiliary_head=dict(...))
```

`ResNet` 和 `HRNet` 使用不同的关键词去构建。

```
_base_ = '../pspnet/psp_r50_512x1024_40ki_cityscpaes.py'
norm_cfg = dict(type='SyncBN', requires_grad=True)
model = dict(
    pretrained='open-mmlab://msra/hrnetv2_w32',
    backbone=dict(
        _delete_=True,
        type='HRNet',
        norm_cfg=norm_cfg,
        extra=dict(
            stage1=dict(
                num_modules=1,
                num_branches=1,
                block='BOTTLENECK',
                num_blocks=(4, ),
                num_channels=(64, )),
            stage2=dict(
```

(下页继续)

(续上页)

```
                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)))),
    decode_head=dict(...),
    auxiliary_head=dict(...))
```

delete=True 将用新的键去替换 backbone 域内所有老的键。

使用配置文件里的中间变量

配置文件里会使用一些中间变量，例如数据集里的 train_pipeline/test_pipeline。需要注意的是，在子配置文件里修改中间变量时，使用者需要再次传递这些变量给对应的域。例如，我们想改变在训练或测试时，PSPNet 的多尺度策略 (multi scale strategy)，train_pipeline/test_pipeline 是我们想要修改的中间变量。

```
_base_ = '../pspnet/psp_r50_512x1024_40ki_cityscapes.py'
crop_size = (512, 1024)
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'),
    dict(type='Resize', img_scale=(2048, 1024), ratio_range=(1.0, 2.0)),  # 改成 [1.,␣
↪2.]
    dict(type='RandomCrop', crop_size=crop_size, cat_max_ratio=0.75),
    dict(type='RandomFlip', flip_ratio=0.5),
    dict(type='PhotoMetricDistortion'),
    dict(type='Normalize', **img_norm_cfg),
    dict(type='Pad', size=crop_size, pad_val=0, seg_pad_val=255),
```

(下页继续)

(续上页)

```
    dict(type='DefaultFormatBundle'),
    dict(type='Collect', keys=['img', 'gt_semantic_seg']),
]
test_pipeline = [
    dict(type='LoadImageFromFile'),
    dict(
        type='MultiScaleFlipAug',
        img_scale=(2048, 1024),
        img_ratios=[0.5, 0.75, 1.0, 1.25, 1.5, 1.75],  # 改成多尺度测试 (multi scale␣
↪testing)。
        flip=False,
        transforms=[
            dict(type='Resize', keep_ratio=True),
            dict(type='RandomFlip'),
            dict(type='Normalize', **img_norm_cfg),
            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))
```

我们首先定义新的 `train_pipeline/test_pipeline` 然后传递到 `data` 里。

同样的，如果我们想从 `SyncBN` 切换到 `BN` 或者 `MMSyncBN`，我们需要配置文件里的每一个 `norm_cfg`。

```
_base_ = '../pspnet/psp_r50_512x1024_40ki_cityscpaes.py'
norm_cfg = dict(type='BN', requires_grad=True)
model = dict(
    backbone=dict(norm_cfg=norm_cfg),
    decode_head=dict(norm_cfg=norm_cfg),
    auxiliary_head=dict(norm_cfg=norm_cfg))
```

2.2 准备数据集（待更新）

推荐用软链接，将数据集根目录链接到 `$MMSEGMENTATION/data` 里。如果您的文件夹结构是不同的，您也许可以试着修改配置文件里对应的路径。

```
mmsegmentation
├── mmseg
```

(下页继续)

(续上页)

```
├── tools
├── configs
├── data
|   ├── cityscapes
|   |   ├── leftImg8bit
|   |   |   ├── train
|   |   |   ├── val
|   |   ├── gtFine
|   |   |   ├── train
|   |   |   ├── val
|   ├── VOCdevkit
|   |   ├── VOC2012
|   |   |   ├── JPEGImages
|   |   |   ├── SegmentationClass
|   |   |   ├── ImageSets
|   |   |   |   ├── Segmentation
|   |   ├── VOC2010
|   |   |   ├── JPEGImages
|   |   |   ├── SegmentationClassContext
|   |   |   ├── ImageSets
|   |   |   |   ├── SegmentationContext
|   |   |   |   |   ├── train.txt
|   |   |   |   |   ├── val.txt
|   |   |   ├── trainval_merged.json
|   |   ├── VOCaug
|   |   |   ├── dataset
|   |   |   |   ├── cls
|   ├── ade
|   |   ├── ADEChallengeData2016
|   |   |   ├── annotations
|   |   |   |   ├── training
|   |   |   |   ├── validation
|   |   |   ├── images
|   |   |   |   ├── training
|   |   |   |   ├── validation
|   ├── CHASE_DB1
|   |   ├── images
|   |   |   ├── training
|   |   |   ├── validation
|   |   ├── annotations
|   |   |   ├── training
|   |   |   ├── validation
|   ├── DRIVE
```

(下页继续)

(续上页)

```
|   |   ├── images
|   |   |   ├── training
|   |   |   ├── validation
|   |   ├── annotations
|   |   |   ├── training
|   |   |   ├── validation
|   ├── HRF
|   |   ├── images
|   |   |   ├── training
|   |   |   ├── validation
|   |   ├── annotations
|   |   |   ├── training
|   |   |   ├── validation
|   ├── STARE
|   |   ├── images
|   |   |   ├── training
|   |   |   ├── validation
|   |   ├── annotations
|   |   |   ├── training
|   |   |   ├── validation
|   ├── dark_zurich
|   |   ├── gps
|   |   |   ├── val
|   |   |   └── val_ref
|   |   ├── gt
|   |   |   └── val
|   |   ├── LICENSE.txt
|   |   ├── lists_file_names
|   |   |   ├── val_filenames.txt
|   |   |   └── val_ref_filenames.txt
|   |   ├── README.md
|   |   └── rgb_anon
|   |   |   ├── val
|   |   |   └── val_ref
|   ├── NighttimeDrivingTest
|   |   ├── gtCoarse_daytime_trainvaltest
|   |   |   └── test
|   |   |       └── night
|   |   └── leftImg8bit
|   |   |   └── test
|   |   |       └── night
|   ├── loveDA
|   |   ├── img_dir
```

(下页继续)

(续上页)

```
|   |   |   ├── train
|   |   |   ├── val
|   |   |   ├── test
|   |   ├── ann_dir
|   |   |   ├── train
|   |   |   ├── val
|   ├── potsdam
|   |   ├── img_dir
|   |   |   ├── train
|   |   |   ├── val
|   |   ├── ann_dir
|   |   |   ├── train
|   |   |   ├── val
|   ├── vaihingen
|   |   ├── img_dir
|   |   |   ├── train
|   |   |   ├── val
|   |   ├── ann_dir
|   |   |   ├── train
|   |   |   ├── val
|   ├── iSAID
|   |   ├── img_dir
|   |   |   ├── train
|   |   |   ├── val
|   |   |   ├── test
|   |   ├── ann_dir
|   |   |   ├── train
|   |   |   ├── val
```

2.2.1 Cityscapes

注册成功后，数据集可以在 这里 下载。

通常情况下，`**labelTrainIds.png` 被用来训练 cityscapes。基于 cityscapesscripts, 我们提供了一个 脚本, 去生成 `**labelTrainIds.png`。

```
# --nproc 8 意味着有 8 个进程用来转换，它也可以被忽略。
python tools/convert_datasets/cityscapes.py data/cityscapes --nproc 8
```

2.2.2 Pascal VOC

Pascal VOC 2012 可以在 这里 下载。此外，许多最近在 Pascal VOC 数据集上的工作都会利用增广的数据，它们可以在 这里 找到。

如果您想使用增广后的 VOC 数据集，请运行下面的命令来将数据增广的标注转成正确的格式。

```
# --nproc 8 意味着有 8 个进程用来转换，它也可以被忽略。
python tools/convert_datasets/voc_aug.py data/VOCdevkit data/VOCdevkit/VOCaug --nproc␣
↪8
```

关于如何拼接数据集 (concatenate) 并一起训练它们，更多细节请参考 拼接连接数据集 。

2.2.3 ADE20K

ADE20K 的训练集和验证集可以在 这里 下载。您还可以在 这里 下载验证集。

2.2.4 Pascal Context

Pascal Context 的训练集和验证集可以在 这里 下载。注册成功后，您还可以在 这里 下载验证集。

为了从原始数据集里切分训练集和验证集，您可以在 这里 下载 trainval_merged.json。

如果您想使用 Pascal Context 数据集，请安装 细节 然后再运行如下命令来把标注转换成正确的格式。

```
python tools/convert_datasets/pascal_context.py data/VOCdevkit data/VOCdevkit/VOC2010/
↪trainval_merged.json
```

2.2.5 CHASE DB1

CHASE DB1 的训练集和验证集可以在 这里 下载。

为了将 CHASE DB1 数据集转换成 MMSegmentation 的格式，您需要运行如下命令：

```
python tools/convert_datasets/chase_db1.py /path/to/CHASEDB1.zip
```

这个脚本将自动生成正确的文件夹结构。

2.2.6 DRIVE

DRIVE 的训练集和验证集可以在 这里 下载。在此之前，您需要注册一个账号，当前‘1st_manual’并未被官方提供，因此需要您从其他地方获取。

为了将 DRIVE 数据集转换成 MMSegmentation 格式，您需要运行如下命令：

```
python tools/convert_datasets/drive.py /path/to/training.zip /path/to/test.zip
```

这个脚本将自动生成正确的文件夹结构。

2.2.7 HRF

首先，下载 healthy.zip glaucoma.zip, diabetic_retinopathy.zip, healthy_manualsegm.zip, glaucoma_manualsegm.zip 以及 diabetic_retinopathy_manualsegm.zip 。

为了将 HRF 数据集转换成 MMSegmentation 格式，您需要运行如下命令：

```
python tools/convert_datasets/hrf.py /path/to/healthy.zip /path/to/healthy_manualsegm.
↪zip /path/to/glaucoma.zip /path/to/glaucoma_manualsegm.zip /path/to/diabetic_
↪retinopathy.zip /path/to/diabetic_retinopathy_manualsegm.zip
```

这个脚本将自动生成正确的文件夹结构。

2.2.8 STARE

首先，下载 stare-images.tar, labels-ah.tar 和 labels-vk.tar 。

为了将 STARE 数据集转换成 MMSegmentation 格式，您需要运行如下命令：

```
python tools/convert_datasets/stare.py /path/to/stare-images.tar /path/to/labels-ah.
↪tar /path/to/labels-vk.tar
```

这个脚本将自动生成正确的文件夹结构。

2.2.9 Dark Zurich

因为我们只支持在此数据集上测试模型，所以您只需下载验证集 。

2.2.10 Nighttime Driving

因为我们只支持在此数据集上测试模型，所以您只需下载测试集 。

2.2.11 LoveDA

可以从 Google Drive 里下载 LoveDA 数据集 。

或者它还可以从 zenodo 下载, 您需要运行如下命令:

```
# Download Train.zip
wget https://zenodo.org/record/5706578/files/Train.zip
# Download Val.zip
wget https://zenodo.org/record/5706578/files/Val.zip
# Download Test.zip
wget https://zenodo.org/record/5706578/files/Test.zip
```

对于 LoveDA 数据集，请运行以下命令下载并重新组织数据集

```
python tools/convert_datasets/loveda.py /path/to/loveDA
```

请参照 这里 来使用训练好的模型去预测 LoveDA 测试集并且提交到官网。

关于 LoveDA 的更多细节可以在这里 找到。

2.2.12 ISPRS Potsdam

Potsdam 数据集是一个有着 2D 语义分割内容标注的城市遥感数据集。数据集可以从挑战主页 获得。需要其中的‘2_Ortho_RGB.zip’和‘5_Labels_all_noBoundary.zip’。

对于 Potsdam 数据集，请运行以下命令下载并重新组织数据集

```
python tools/convert_datasets/potsdam.py /path/to/potsdam
```

使用我们默认的配置，将生成 3456 张图片的训练集和 2016 张图片的验证集。

2.2.13 ISPRS Vaihingen

Vaihingen 数据集是一个有着 2D 语义分割内容标注的城市遥感数据集。

数据集可以从挑战 主页. 需要其中的‘ISPRS_semantic_labeling_Vaihingen.zip’和‘ISPRS_semantic_labeling_Vaihingen_ground_truth_eroded_COMPLETE.zip’。

对于 Vaihingen 数据集，请运行以下命令下载并重新组织数据集

```
python tools/convert_datasets/vaihingen.py /path/to/vaihingen
```

使用我们默认的配置 (clip_size=512, stride_size=256)，将生成 344 张图片的训练集和 398 张图片的验证集。

2.2.14 iSAID

iSAID 数据集 (训练集/验证集/测试集) 的图像可以从 DOTA-v1.0 下载.

iSAID 数据集 (训练集/验证集) 的注释可以从 iSAID 下载.

该数据集是一个大规模的实例分割 (也可以用于语义分割) 的遥感数据集.

下载后，在数据集转换前，您需要将数据集文件夹调整成如下格式.

```
|   ├── iSAID
|   |   ├── train
|   |   |   ├── images
|   |   |   |   ├── part1.zip
|   |   |   |   ├── part2.zip
|   |   |   |   ├── part3.zip
|   |   |   ├── Semantic_masks
|   |   |   |   ├── images.zip
|   |   ├── val
|   |   |   ├── images
|   |   |   |   ├── part1.zip
|   |   |   ├── Semantic_masks
|   |   |   |   ├── images.zip
|   |   ├── test
|   |   |   ├── images
|   |   |   |   ├── part1.zip
|   |   |   |   ├── part2.zip
```

```
python tools/convert_datasets/isaid.py /path/to/iSAID
```

使用我们默认的配置 (patch_width=896, patch_height=896, overlap_area=384)，将生成 33978 张图片的训练集和 11644 张图片的验证集。

2.3 使用预训练模型推理（待更新）

我们提供测试脚本来评估完整数据集（Cityscapes, PASCAL VOC, ADE20k 等）上的结果，同时为了使其他项目的整合更容易，也提供一些高级 API。

2.3.1 测试一个数据集

- 单卡 GPU
- CPU
- 单节点多卡 GPU
- 多节点

您可以使用以下命令来测试一个数据集。

```
# 单卡 GPU 测试
python tools/test.py ${配置文件} ${检查点文件} [--out ${结果文件}] [--eval ${评估指标}] [--
↪show]

# CPU: 如果机器没有 GPU, 则跟上述单卡 GPU 测试一致
# CPU: 如果机器有 GPU, 那么先禁用 GPU 再运行单 GPU 测试脚本
export CUDA_VISIBLE_DEVICES=-1 # 禁用 GPU
python tools/test.py ${配置文件} ${检查点文件} [--out ${结果文件}] [--eval ${评估指标}] [--
↪show]

# 多卡 GPU 测试
./tools/dist_test.sh ${配置文件} ${检查点文件} ${GPU 数目} [--out ${结果文件}] [--eval ${评
估指标}]
```

可选参数:

- `RESULT_FILE`: pickle 格式的输出结果的文件名，如果不专门指定，结果将不会被专门保存成文件。（MMseg v0.17 之后，args.out 将只会保存评估时的中间结果或者是分割图的保存路径。）
- `EVAL_METRICS`: 在结果里将被评估的指标。这主要取决于数据集，`mIoU` 对于所有数据集都可获得，像 Cityscapes 数据集可以通过 `cityscapes` 命令来专门评估，就像标准的 `mIoU` 一样。
- `--show`: 如果被指定，分割结果将会在一张图像里画出来并且在另一个窗口展示。它仅仅是用来调试与可视化，并且仅针对单卡 GPU 测试。请确认 GUI 在您的环境里可用，否则您也许会遇到报错 `cannot connect to X server`
- `--show-dir`: 如果被指定，分割结果将会在一张图像里画出来并且保存在指定文件夹里。它仅仅是用来调试与可视化，并且仅针对单卡 GPU 测试。使用该参数时，您的环境不需要 GUI。
- `--eval-options`: 评估时的可选参数，当设置 `efficient_test=True` 时，它将会保存中间结果至本地文件里以节约 CPU 内存。请确认您本地硬盘有足够的存储空间（大于 20GB）。（MMseg v0.17 之

后，efficient_test 不再生效，我们重构了 test api，通过使用一种渐近式的方式来提升评估和保存结果的效率。）

例子:

假设您已经下载检查点文件至文件夹 checkpoints/ 里。

1. 测试 PSPNet 并可视化结果。按下任何键会进行到下一张图

```
python tools/test.py configs/pspnet/pspnet_r50-d8_512x1024_40k_cityscapes.py \
    checkpoints/pspnet_r50-d8_512x1024_40k_cityscapes_20200605_003338-2966598c.
↪pth \
    --show
```

2. 测试 PSPNet 并保存画出的图以便于之后的可视化

```
python tools/test.py configs/pspnet/pspnet_r50-d8_512x1024_40k_cityscapes.py \
    checkpoints/pspnet_r50-d8_512x1024_40k_cityscapes_20200605_003338-2966598c.
↪pth \
    --show-dir psp_r50_512x1024_40ki_cityscapes_results
```

3. 在数据集 PASCAL VOC (不保存测试结果) 上测试 PSPNet 并评估 mIoU

```
python tools/test.py configs/pspnet/pspnet_r50-d8_512x1024_20k_voc12aug.py \
    checkpoints/pspnet_r50-d8_512x1024_20k_voc12aug_20200605_003338-c57ef100.pth \
    --eval mAP
```

4. 使用 4 卡 GPU 测试 PSPNet，并且在标准 mIoU 和 cityscapes 指标里评估模型

```
./tools/dist_test.sh configs/pspnet/pspnet_r50-d8_512x1024_40k_cityscapes.py \
    checkpoints/pspnet_r50-d8_512x1024_40k_cityscapes_20200605_003338-2966598c.
↪pth \
    4 --out results.pkl --eval mIoU cityscapes
```

注意：在 cityscapes mIoU 和我们的 mIoU 指标会有一些差异 (~0.1%) 。因为 cityscapes 默认是根据类别样本数的多少进行加权平均，而我们对所有的数据集都是采取直接平均的方法来得到 mIoU。

5. 在 cityscapes 数据集上 4 卡 GPU 测试 PSPNet，并生成 png 文件以便提交给官方评估服务器

首先,在配置文件里添加内容:configs/pspnet/pspnet_r50-d8_512x1024_40k_cityscapes.py，

```
data = dict(
    test=dict(
        img_dir='leftImg8bit/test',
        ann_dir='gtFine/test'))
```

随后，进行测试。

```
./tools/dist_test.sh configs/pspnet/pspnet_r50-d8_512x1024_40k_cityscapes.py \
    checkpoints/pspnet_r50-d8_512x1024_40k_cityscapes_20200605_003338-2966598c.
↪pth \
    4 --format-only --eval-options "imgfile_prefix=./pspnet_test_results"
```

您会在文件夹 `./pspnet_test_results` 里得到生成的 png 文件。您也许可以运行 `zip -r results.zip pspnet_test_results/` 并提交 zip 文件给 evaluation server 。

6. 在 Cityscapes 数据集上使用 CPU 高效内存选项来测试 DeeplabV3+ `mIoU` 指标 (没有保存测试结果)

```
python tools/test.py \
configs/deeplabv3plus/deeplabv3plus_r18-d8_512x1024_80k_cityscapes.py \
deeplabv3plus_r18-d8_512x1024_80k_cityscapes_20201226_080942-cff257fe.pth \
--eval-options efficient_test=True \
--eval mIoU
```

使用 `pmap` 可查看 CPU 内存情况，`efficient_test=True` 会使用约 2.25GB 的 CPU 内存，`efficient_test=False` 会使用约 11.06GB 的 CPU 内存。这个可选参数可以节约很多 CPU 内存。（MMseg v0.17 之后, `efficient_test` 参数将不再生效, 我们使用了一种渐近的方式来更加有效快速地评估和保存结果。）

7. 在 LoveDA 数据集上 1 卡 GPU 测试 PSPNet，并生成 png 文件以便提交给官方评估服务器

首先，在配置文件里添加内容：`configs/pspnet/pspnet_r50-d8_512x512_80k_loveda.py`,

```
data = dict(
    test=dict(
        img_dir='img_dir/test',
        ann_dir='ann_dir/test'))
```

随后，进行测试。

```
python ./tools/test.py configs/pspnet/pspnet_r50-d8_512x512_80k_loveda.py \
    checkpoints/pspnet_r50-d8_512x512_80k_loveda_20211104_155728-88610f9f.pth \
    --format-only --eval-options "imgfile_prefix=./pspnet_test_results"
```

您会在文件夹 `./pspnet_test_results` 里得到生成的 png 文件。您也许可以运行 `zip -r -j Results.zip pspnet_test_results/` 并提交 zip 文件给 evaluation server 。

CHAPTER 3

实用工具

3.1 可视化

3.2 常用工具（待更新）

除了训练和测试的脚本，我们在 `tools/` 文件夹路径下还提供许多有用的工具。

3.2.1 计算参数量（params）和计算量（FLOPs）(试验性)

我们基于 flops-counter.pytorch 提供了一个用于计算给定模型参数量和计算量的脚本。

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

您将得到如下的结果：

```
==============================
Input shape: (3, 2048, 1024)
Flops: 1429.68 GMac
Params: 48.98 M
==============================
```

注意: 这个工具仍然是试验性的，我们无法保证数字是正确的。您可以拿这些结果做简单的实验的对照，在写技术文档报告或者论文前您需要再次确认一下。

(1) 计算量与输入的形状有关，而参数量与输入的形状无关，默认的输入形状是 (1, 3, 1280, 800)；(2) 一些运算操作，如 GN 和其他定制的运算操作没有加入到计算量的计算中。

3.2.2 发布模型

在您上传一个模型到云服务器之前，您需要做以下几步：(1) 将模型权重转成 CPU 张量；(2) 删除记录优化器状态 (optimizer states) 的相关信息；(3) 计算检查点文件 (checkpoint file) 的哈希编码（hash id）并且将哈希编码加到文件名中。

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

例如，

```
python tools/publish_model.py work_dirs/pspnet/latest.pth psp_r50_hszhao_200ep.pth
```

最终输出文件将是 `psp_r50_512x1024_40ki_cityscapes-{hash id}.pth`。

3.2.3 导出 ONNX (试验性)

我们提供了一个脚本来导出模型到 ONNX 格式。被转换的模型可以通过工具 Netron 来可视化。除此以外，我们同样支持对 PyTorch 和 ONNX 模型的输出结果做对比。

```
python tools/pytorch2onnx.py \
    ${CONFIG_FILE} \
    --checkpoint ${CHECKPOINT_FILE} \
    --output-file ${ONNX_FILE} \
    --input-img ${INPUT_IMG} \
    --shape ${INPUT_SHAPE} \
    --rescale-shape ${RESCALE_SHAPE} \
    --show \
    --verify \
    --dynamic-export \
    --cfg-options \
      model.test_cfg.mode="whole"
```

各个参数的描述:

- `config` : 模型配置文件的路径
- `--checkpoint` : 模型检查点文件的路径
- `--output-file`: 输出的 ONNX 模型的路径。如果没有专门指定，它默认是 `tmp.onnx`
- `--input-img` : 用来转换和可视化的一张输入图像的路径
- `--shape`: 模型的输入张量的高和宽。如果没有专门指定，它将被设置成 `test_pipeline` 的 `img_scale`

- `--rescale-shape`: 改变输出的形状。设置这个值来避免 OOM，它仅在 `slide` 模式下可以用
- `--show`: 是否打印输出模型的结构。如果没有被专门指定，它将被设置成 `False`
- `--verify`: 是否验证一个输出模型的正确性 (correctness)。如果没有被专门指定，它将被设置成 `False`
- `--dynamic-export`: 是否导出形状变化的输入与输出的 ONNX 模型。如果没有被专门指定，它将被设置成 `False`
- `--cfg-options`: 更新配置选项

注意: 这个工具仍然是试验性的，目前一些自定义操作还没有被支持

3.2.4 评估 ONNX 模型

我们提供 `tools/deploy_test.py` 去评估不同后端的 ONNX 模型。

先决条件

- 安装 onnx 和 onnxruntime-gpu

```
pip install onnx onnxruntime-gpu
```

- 参考 如何在 MMCV 里构建 tensorrt 插件 安装 TensorRT (可选)

使用方法

```
python tools/deploy_test.py \
    ${CONFIG_FILE} \
    ${MODEL_FILE} \
    ${BACKEND} \
    --out ${OUTPUT_FILE} \
    --eval ${EVALUATION_METRICS} \
    --show \
    --show-dir ${SHOW_DIRECTORY} \
    --cfg-options ${CFG_OPTIONS} \
    --eval-options ${EVALUATION_OPTIONS} \
    --opacity ${OPACITY} \
```

各个参数的描述:

- `config`: 模型配置文件的路径
- `model`: 被转换的模型文件的路径
- `backend`: 推理的后端，可选项：`onnxruntime`，`tensorrt`
- `--out`: 输出结果成 pickle 格式文件的路径

- `--format-only` : 不评估直接给输出结果的格式。通常用在当您想把结果输出成一些测试服务器需要的特定格式时。如果没有被专门指定，它将被设置成 `False`。注意这个参数是用 `--eval` 来 **手动添加**
- `--eval`: 评估指标，取决于每个数据集的要求，例如 “mIoU” 是大多数据集的指标而 “cityscapes” 仅针对 Cityscapes 数据集。注意这个参数是用 `--format-only` 来 **手动添加**
- `--show`: 是否展示结果
- `--show-dir`: 涂上结果的图像被保存的文件夹的路径
- `--cfg-options`: 重写配置文件里的一些设置，`xxx=yyy` 格式的键值对将被覆盖到配置文件里
- `--eval-options`: 自定义的评估的选项，`xxx=yyy` 格式的键值对将成为 `dataset.evaluate()` 函数的参数变量
- `--opacity`: 涂上结果的分割图的透明度，范围在 (0, 1] 之间

结果和模型

注意: TensorRT 仅在使用 `whole mode` 测试模式时的配置文件里可用。

3.2.5 导出 TorchScript (试验性)

我们同样提供一个脚本去把模型导出成 TorchScript 格式。您可以使用 pytorch C++ API LibTorch 去推理训练好的模型。被转换的模型能被像 Netron 的工具来可视化。此外，我们还支持 PyTorch 和 TorchScript 模型的输出结果的比较。

```
python tools/pytorch2torchscript.py \
    ${CONFIG_FILE} \
    --checkpoint ${CHECKPOINT_FILE} \
    --output-file ${ONNX_FILE}
    --shape ${INPUT_SHAPE}
    --verify \
    --show
```

各个参数的描述:

- `config` : pytorch 模型的配置文件的路径
- `--checkpoint` : pytorch 模型的检查点文件的路径
- `--output-file`: TorchScript 模型输出的路径，如果没有被专门指定，它将被设置成 `tmp.pt`
- `--input-img` : 用来转换和可视化的输入图像的路径
- `--shape`: 模型的输入张量的宽和高。如果没有被专门指定，它将被设置成 `512 512`
- `--show`: 是否打印输出模型的追踪图 (traced graph)，如果没有被专门指定，它将被设置成 `False`
- `--verify`: 是否验证一个输出模型的正确性 (correctness)，如果没有被专门指定，它将被设置成 `False`

注意: 目前仅支持 PyTorch>=1.8.0 版本

注意: 这个工具仍然是试验性的，一些自定义操作符目前还不被支持

例子:

- 导出 PSPNet 在 cityscapes 数据集上的 pytorch 模型

```
python tools/pytorch2torchscript.py configs/pspnet/pspnet_r50-d8_512x1024_40k_
↪cityscapes.py \
--checkpoint checkpoints/pspnet_r50-d8_512x1024_40k_cityscapes_20200605_003338-
↪2966598c.pth \
--output-file checkpoints/pspnet_r50-d8_512x1024_40k_cityscapes_20200605_003338-
↪2966598c.pt \
--shape 512 1024
```

3.2.6 导出 TensorRT (试验性)

一个导出 ONNX 模型成 TensorRT 格式的脚本

先决条件

- 按照 ONNXRuntime in mmcv 和 TensorRT plugin in mmcv，用 ONNXRuntime 自定义运算 (custom ops) 和 TensorRT 插件安装 `mmcv-full`
- 使用 pytorch2onnx 将模型从 PyTorch 转成 ONNX

使用方法

```
python ${MMSEG_PATH}/tools/onnx2tensorrt.py \
    ${CFG_PATH} \
    ${ONNX_PATH} \
    --trt-file ${OUTPUT_TRT_PATH} \
    --min-shape ${MIN_SHAPE} \
    --max-shape ${MAX_SHAPE} \
    --input-img ${INPUT_IMG} \
    --show \
    --verify
```

各个参数的描述:

- `config` : 模型的配置文件
- `model` : 输入的 ONNX 模型的路径
- `--trt-file` : 输出的 TensorRT 引擎的路径
- `--max-shape` : 模型的输入的最大形状
- `--min-shape` : 模型的输入的最小形状

- `--fp16` : 做 fp16 模型转换
- `--workspace-size` : 在 GiB 里的最大工作空间大小 (Max workspace size)
- `--input-img` : 用来可视化的图像
- `--show` : 做结果的可视化
- `--dataset` : Palette provider, 默认为 `CityscapesDataset`
- `--verify` : 验证 ONNXRuntime 和 TensorRT 的输出
- `--verbose` : 当创建 TensorRT 引擎时，是否详细做信息日志。默认为 False

注意: 仅在全图测试模式 (whole mode) 下测试过

3.3 其他内容

3.3.1 打印完整的配置文件

`tools/print_config.py` 会逐字逐句的打印整个配置文件，展开所有的导入。

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

各个参数的描述:

- `config` : pytorch 模型的配置文件的路径
- `--graph` : 是否打印模型的图 (models graph)
- `--cfg-options`: 自定义替换配置文件的选项

3.3.2 对训练日志 (training logs) 画图

`tools/analyze_logs.py` 会画出给定的训练日志文件的 loss/mIoU 曲线，首先需要 `pip install seaborn` 安装依赖包。

```
python tools/analyze_logs.py xxx.log.json [--keys ${KEYS}] [--legend ${LEGEND}] [--
↪backend ${BACKEND}] [--style ${STYLE}] [--out ${OUT_FILE}]
```

示例:

- 对 mIoU, mAcc, aAcc 指标画图

```
python tools/analyze_logs.py log.json --keys mIoU mAcc aAcc --legend mIoU mAcc␣
↪aAcc
```

- 对 loss 指标画图

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

3.3.3 转换其他仓库的权重

tools/model_converters/ 提供了若干个预训练权重转换脚本，支持将其他仓库的预训练权重的 key 转换为与 MMSegmentation 相匹配的 key。

ViT Swin MiT Transformer 模型

- ViT

tools/model_converters/vit2mmseg.py 将 timm 预训练模型转换到 MMSegmentation。

```
python tools/model_converters/vit2mmseg.py ${SRC} ${DST}
```

- Swin

  tools/model_converters/swin2mmseg.py 将官方预训练模型转换到 MMSegmentation。

```
python tools/model_converters/swin2mmseg.py ${SRC} ${DST}
```

- SegFormer

  tools/model_converters/mit2mmseg.py 将官方预训练模型转换到 MMSegmentation。

```
python tools/model_converters/mit2mmseg.py ${SRC} ${DST}
```

3.4 模型服务

为了用 TorchServe 服务 MMSegmentation 的模型，您可以遵循如下流程:

3.4.1 1. 将 model 从 MMSegmentation 转换到 TorchServe

```
python tools/mmseg2torchserve.py ${CONFIG_FILE} ${CHECKPOINT_FILE} \
--output-folder ${MODEL_STORE} \
--model-name ${MODEL_NAME}
```

注意: ${MODEL_STORE} 需要设置为某个文件夹的绝对路径

3.4.2 2. 构建 mmseg-serve 容器镜像 (docker image)

```
docker build -t mmseg-serve:latest docker/serve/
```

3.4.3 3. 运行 mmseg-serve

请查阅官方文档: 使用容器运行 TorchServe

为了在 GPU 环境下使用, 您需要安装 nvidia-docker. 若在 CPU 环境下使用，您可以忽略添加 --gpus 参数。

示例:

```
docker run --rm \
--cpus 8 \
--gpus device=0 \
-p8080:8080 -p8081:8081 -p8082:8082 \
--mount type=bind,source=$MODEL_STORE,target=/home/model-server/model-store \
mmseg-serve:latest
```

阅读关于推理 (8080), 管理 (8081) 和指标 (8082) APIs 的 文档 。

3.4.4 4. 测试部署

```
curl -O https://raw.githubusercontent.com/open-mmlab/mmsegmentation/master/resources/
↪3dogs.jpg
curl http://127.0.0.1:8080/predictions/${MODEL_NAME} -T 3dogs.jpg -o 3dogs_mask.png
```

得到的响应将是一个 “.png” 的分割掩码.

您可以按照如下方法可视化输出:

```
import matplotlib.pyplot as plt
import mmcv
plt.imshow(mmcv.imread("3dogs_mask.png", "grayscale"))
plt.show()
```

看到的东西将会和下图类似:

然后您可以使用 `test_torchserve.py` 比较 torchserve 和 pytorch 的结果，并将它们可视化。

```
python tools/torchserve/test_torchserve.py ${IMAGE_FILE} ${CONFIG_FILE} ${CHECKPOINT_
↪FILE} ${MODEL_NAME}
[--inference-addr ${INFERENCE_ADDR}] [--result-image ${RESULT_IMAGE}] [--device $
↪{DEVICE}]
```

示例:

```
python tools/torchserve/test_torchserve.py \
demo/demo.png \
configs/fcn/fcn_r50-d8_512x1024_40k_cityscapes.py \
checkpoint/fcn_r50-d8_512x1024_40k_cityscapes_20200604_192608-efe53f0d.pth \
fcn
```

3.5 模型部署

CHAPTER 4

基本概念

4.1 数据流

4.2 数据结构

为了统一模型和各功能模块之间的输入和输出的接口，在 OpenMMLab 2.0 MMEngine 中定义了一套抽象数据结构，实现了基础的增/删/查/改功能，支持不同设备间的数据迁移，也支持了如 `.cpu()`，`.cuda()`，`.get()` 和 `.detach()` 的类字典和张量的操作。具体可以参考 MMEngine 文档。

同样的，MMSegmentation 亦遵循了 OpenMMLab 2.0 各模块间的接口协议，定义了 `SegDataSample` 用来封装语义分割任务所需要的数据。

4.2.1 语义分割数据 SegDataSample

SegDataSample 包括了三个主要数据字段 `gt_sem_seg`，`pred_sem_seg` 和 `seg_logits`，分别用来存放标注信息，预测结果和预测的未归一化前的 logits 值。

以下示例代码展示了 `SegDataSample` 的使用方法：

```
import torch
from mmengine.structures import PixelData
from mmseg.structures import SegDataSample

img_meta = dict(img_shape=(4, 4, 3),
```

(下页继续)

(续上页)

```
                  pad_shape=(4, 4, 3))
data_sample = SegDataSample()
# 定义 gt_segmentations 用于封装模型的输出信息
gt_segmentations = PixelData(metainfo=img_meta)
gt_segmentations.data = torch.randint(0, 2, (1, 4, 4))

# 增加和处理 SegDataSample   中的属性
data_sample.gt_sem_seg = gt_segmentations
assert 'gt_sem_seg' in data_sample
assert 'data' in data_sample.gt_sem_seg
assert 'img_shape' in data_sample.gt_sem_seg.metainfo_keys()
print(data_sample.gt_sem_seg.shape)
'''
(4, 4)
'''
print(data_sample)
'''
<SegDataSample(

    META INFORMATION

    DATA FIELDS
    gt_sem_seg: <PixelData(

            META INFORMATION
            img_shape: (4, 4, 3)
            pad_shape: (4, 4, 3)

            DATA FIELDS
            data: tensor([[[1, 1, 1, 0],
                           [1, 0, 1, 1],
                           [1, 1, 1, 1],
                           [0, 1, 0, 1]]])
        ) at 0x1c2b4156460>
) at 0x1c2aae44d60>
'''

# 删除和修改 SegDataSample   中的属性
data_sample = SegDataSample()
gt_segmentations = PixelData(metainfo=img_meta)
gt_segmentations.data = torch.randint(0, 2, (1, 4, 4))
data_sample.gt_sem_seg = gt_segmentations
data_sample.gt_sem_seg.set_metainfo(dict(img_shape=(4,4,9), pad_shape=(4,4,9)))
```

(下页继续)

(续上页)

```
del data_sample.gt_sem_seg.img_shape

# 类张量的操作
data_sample = SegDataSample()
gt_segmentations = PixelData(metainfo=img_meta)
gt_segmentations.data = torch.randint(0, 2, (1, 4, 4))
cuda_gt_segmentations = gt_segmentations.cuda()
cuda_gt_segmentations = gt_segmentations.to('cuda:0')
cpu_gt_segmentations = cuda_gt_segmentations.cpu()
cpu_gt_segmentations = cuda_gt_segmentations.to('cpu')
```

4.2.2 在 SegDataSample 中自定义新的属性

如果你想在 SegDataSample 中自定义新的属性，你可以参考下面的 SegDataSample 示例:

```
class SegDataSample(BaseDataElement):
    ...

    @property
    def xxx_property(self) -> xxxData:
        return self._xxx_property

    @xxx_property.setter
    def xxx_property(self, value: xxxData) -> None:
        self.set_field(value, '_xxx_property', dtype=xxxData)

    @xxx_property.deleter
    def xxx_property(self) -> None:
        del self._xxx_property
```

这样一个新的属性 xxx_property 就将被增加到 SegDataSample 里面了。

4.3 模型

数据集

4.4 数据增广

模型评测

4.5 训练引擎

4.6 训练技巧（待更新）

MMSegmentation 支持如下训练技巧：

4.6.1 主干网络和解码头组件使用不同的学习率 (Learning Rate, LR)

在语义分割里，一些方法会让解码头组件的学习率大于主干网络的学习率，这样可以获得更好的表现或更快的收敛。

在 MMSegmentation 里面，您也可以在配置文件里添加如下行来让解码头组件的学习率是主干组件的 10 倍。

```
optimizer=dict(
    paramwise_cfg = dict(
        custom_keys={
            'head': dict(lr_mult=10.)}))
```

通过这种修改，任何被分组到 `'head'` 的参数的学习率都将乘以 10。您也可以参照 MMCV 文档 获取更详细的信息。

4.6.2 在线难样本挖掘 (Online Hard Example Mining, OHEM)

对于训练时采样，我们在 这里 做了像素采样器。如下例子是使用 PSPNet 训练并采用 OHEM 策略的配置：

```
_base_ = './pspnet_r50-d8_512x1024_40k_cityscapes.py'
model=dict(
    decode_head=dict(
        sampler=dict(type='OHEMPixelSampler', thresh=0.7, min_kept=100000)) )
```

通过这种方式，只有置信分数在 0.7 以下的像素值点会被拿来训练。在训练时我们至少要保留 100000 个像素值点。如果 `thresh` 并未被指定，前 `min_kept` 个损失的像素值点才会被选择。

4.6.3 类别平衡损失 (Class Balanced Loss)

对于不平衡类别分布的数据集，您也许可以改变每个类别的损失权重。这里以 cityscapes 数据集为例：

```
_base_ = './pspnet_r50-d8_512x1024_40k_cityscapes.py'
model=dict(
    decode_head=dict(
        loss_decode=dict(
            type='CrossEntropyLoss', use_sigmoid=False, loss_weight=1.0,
            # DeepLab 对 cityscapes 使用这种权重
            class_weight=[0.8373, 0.9180, 0.8660, 1.0345, 1.0166, 0.9969, 0.9754,
                          1.0489, 0.8786, 1.0023, 0.9539, 0.9843, 1.1116, 0.9037,
                          1.0865, 1.0955, 1.0865, 1.1529, 1.0507])))
```

class_weight 将被作为 weight 参数，传递给 CrossEntropyLoss。详细信息请参照 PyTorch 文档 。

4.6.4 同时使用多种损失函数 (Multiple Losses)

对于训练时损失函数的计算，我们目前支持多个损失函数同时使用。以 unet 使用 DRIVE 数据集训练为例，使用 CrossEntropyLoss 和 DiceLoss 的 1:3 的加权和作为损失函数。配置文件写为:

```
_base_ = './fcn_unet_s5-d16_64x64_40k_drive.py'
model = dict(
    decode_head=dict(loss_decode=[dict(type='CrossEntropyLoss', loss_name='loss_ce',␣
↪loss_weight=1.0),
            dict(type='DiceLoss', loss_name='loss_dice', loss_weight=3.0)]),
    auxiliary_head=dict(loss_decode=[dict(type='CrossEntropyLoss', loss_name='loss_ce
↪',loss_weight=1.0),
            dict(type='DiceLoss', loss_name='loss_dice', loss_weight=3.0)]),
    )
```

通过这种方式，确定训练过程中损失函数的权重 loss_weight 和在训练日志里的名字 loss_name。

注意：loss_name 的名字必须带有 loss_ 前缀，这样它才能被包括在反传的图里。

4.6.5 在损失函数中忽略特定的 label 类别

默认设置 avg_non_ignore=False，即每个像素都用来计算损失函数。尽管其中的一些像素属于需要被忽略的类别。

对于训练时损失函数的计算，我们目前支持使用 avg_non_ignore 和 ignore_index 来忽略 label 特定的类别。这样损失函数将只在非忽略类别像素中求平均值，会获得更好的表现。这里是相关 PR。以 unet 使用 Cityscapes 数据集训练为例，在计算损失函数时，忽略 label 为 0 的背景，并且仅在不被忽略的像素上计算均值。配置文件写为:

```
_base_ = './fcn_unet_s5-d16_4x4_512x1024_160k_cityscapes.py'
model = dict(
    decode_head=dict(
        ignore_index=0,
        loss_decode=dict(
            type='CrossEntropyLoss', use_sigmoid=False, loss_weight=1.0, avg_non_
↪ignore=True),
    auxiliary_head=dict(
        ignore_index=0,
        loss_decode=dict(
            type='CrossEntropyLoss', use_sigmoid=False, loss_weight=1.0, avg_non_
↪ignore=True)),
    ))
```

通过这种方式，确定训练过程中损失函数的权重 `loss_weight` 和在训练日志里的名字 `loss_name`。

注意：`loss_name` 的名字必须带有 `loss_` 前缀，这样它才能被包括在反传的图里。

CHAPTER 5

自定义组件

5.1 自定义模型（待更新）

5.1.1 自定义优化器 (optimizer)

假设您想增加一个新的叫 MyOptimizer 的优化器，它的参数分别为 a, b, 和 c。您首先需要在一个文件里实现这个新的优化器，例如在 mmseg/core/optimizer/my_optimizer.py 里面：

```
from mmcv.runner import OPTIMIZERS
from torch.optim import Optimizer


@OPTIMIZERS.register_module
class MyOptimizer(Optimizer):

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

然后增加这个模块到 mmseg/core/optimizer/__init__.py 里面，这样注册器 (registry) 将会发现这个新的模块并添加它：

```
from .my_optimizer import MyOptimizer
```

之后您可以在配置文件的 optimizer 域里使用 MyOptimizer，如下所示，在配置文件里，优化器被 optimizer 域所定义：

```
optimizer = dict(type='SGD', lr=0.02, momentum=0.9, weight_decay=0.0001)
```

为了使用您自己的优化器，域可以被修改为：

```
optimizer = dict(type='MyOptimizer', a=a_value, b=b_value, c=c_value)
```

我们已经支持了 PyTorch 自带的全部优化器，唯一修改的地方是在配置文件里的 `optimizer` 域。例如，如果您想使用 `ADAM`，尽管数值表现会掉点，还是可以如下修改：

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

使用者可以直接按照 PyTorch 文档教程 去设置参数。

5.1.2 定制优化器的构造器 (optimizer constructor)

对于优化，一些模型可能会有一些特别定义的参数，例如批归一化 (BatchNorm) 层里面的权重衰减 (weight decay)。使用者可以通过定制优化器的构造器来微调这些细粒度的优化器参数。

```
from mmcv.utils import build_from_cfg

from mmcv.runner import OPTIMIZER_BUILDERS
from .cocktail_optimizer import CocktailOptimizer


@OPTIMIZER_BUILDERS.register_module
class CocktailOptimizerConstructor(object):

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

    def __call__(self, model):

        return my_optimizer
```

5.1.3 开发和增加新的组件（Module）

MMSegmentation 里主要有 2 种组件：

- 主干网络 (backbone): 通常是卷积网络的堆叠，来做特征提取，例如 ResNet, HRNet
- 解码头 (decoder head): 用于语义分割图的解码的组件（得到分割结果）

添加新的主干网络

这里我们以 MobileNet 为例，展示如何增加新的主干组件：

1. 创建一个新的文件 mmseg/models/backbones/mobilenet.py

```
import torch.nn as nn

from ..registry 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
```

2. 在 mmseg/models/backbones/__init__.py 里面导入模块

```
from .mobilenet import MobileNet
```

3. 在您的配置文件里使用它

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

增加新的解码头 (decoder head) 组件

在 MMSegmentation 里面，对于所有的分割头，我们提供一个基类解码头 BaseDecodeHead 。所有新建的解码头都应该继承它。这里我们以 PSPNet 为例，展示如何开发和增加一个新的解码头组件：

首先，在 mmseg/models/decode_heads/psp_head.py 里添加一个新的解码头。PSPNet 中实现了一个语义分割的解码头。为了实现一个解码头，我们只需要在新构造的解码头中实现如下的 3 个函数：

```
@HEADS.register_module()
class PSPHead(BaseDecodeHead):

    def __init__(self, pool_scales=(1, 2, 3, 6), **kwargs):
        super(PSPHead, self).__init__(**kwargs)

    def init_weights(self):

    def forward(self, inputs):
```

接着，使用者需要在 mmseg/models/decode_heads/__init__.py 里面添加这个模块，这样对应的注册器 (registry) 可以查找并加载它们。

PSPNet 的配置文件如下所示：

```
norm_cfg = dict(type='SyncBN', requires_grad=True)
model = dict(
    type='EncoderDecoder',
    pretrained='pretrain_model/resnet50_v1c_trick-2cccc1ad.pth',
    backbone=dict(
        type='ResNetV1c',
        depth=50,
        num_stages=4,
        out_indices=(0, 1, 2, 3),
        dilations=(1, 1, 2, 4),
        strides=(1, 2, 1, 1),
        norm_cfg=norm_cfg,
        norm_eval=False,
        style='pytorch',
        contract_dilation=True),
    decode_head=dict(
        type='PSPHead',
        in_channels=2048,
        in_index=3,
        channels=512,
        pool_scales=(1, 2, 3, 6),
        dropout_ratio=0.1,
        num_classes=19,
        norm_cfg=norm_cfg,
        align_corners=False,
        loss_decode=dict(
            type='CrossEntropyLoss', use_sigmoid=False, loss_weight=1.0)))
```

增加新的损失函数

假设您想添加一个新的损失函数 MyLoss 到语义分割解码器里。为了添加一个新的损失函数，使用者需要在 mmseg/models/losses/my_loss.py 里面去实现它。weighted_loss 可以对计算损失时的每个样本做加权。

```
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 = self.loss_weight * my_loss(
            pred, target, weight, reduction=reduction, avg_factor=avg_factor)
        return loss
```

然后使用者需要在 mmseg/models/losses/__init__.py 里面添加它：

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

为了使用它，修改 loss_xxx 域。之后您需要在解码头组件里修改 loss_decode 域。loss_weight 可以

被用来对不同的损失函数做加权。

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

5.2 自定义数据集（待更新）

5.2.1 通过重新组织数据来定制数据集

最简单的方法是将您的数据集进行转化，并组织成文件夹的形式。

如下的文件结构就是一个例子。

```
├── data
|   ├── my_dataset
|   |   ├── img_dir
|   |   |   ├── train
|   |   |   |   ├── xxx{img_suffix}
|   |   |   |   ├── yyy{img_suffix}
|   |   |   |   ├── zzz{img_suffix}
|   |   |   ├── val
|   |   ├── ann_dir
|   |   |   ├── train
|   |   |   |   ├── xxx{seg_map_suffix}
|   |   |   |   ├── yyy{seg_map_suffix}
|   |   |   |   ├── zzz{seg_map_suffix}
|   |   |   ├── val
```

一个训练对将由 img_dir/ann_dir 里同样首缀的文件组成。

如果给定 `split` 参数，只有部分在 img_dir/ann_dir 里的文件会被加载。我们可以对被包括在 split 文本里的文件指定前缀。

除此以外，一个 split 文本如下所示：

```
xxx
zzz
```

只有

`data/my_dataset/img_dir/train/xxx{img_suffix}`, `data/my_dataset/img_dir/train/zzz{img_suffix}`, `data/my_dataset/ann_dir/train/xxx{seg_map_suffix}`, `data/my_dataset/ann_dir/train/zzz{seg_map_suffix}` 将被加载。

注意：标注是跟图像同样的形状 (H, W)，其中的像素值的范围是 `[0, num_classes - 1]`。您也可以使用 pillow 的 `'P'` 模式去创建包含颜色的标注。

5.2.2 通过混合数据去定制数据集

MMSegmentation 同样支持混合数据集去训练。当前它支持拼接 (concat), 重复 (repeat) 和多图混合 (multi-image mix) 数据集。

重复数据集

我们使用 `RepeatDataset` 作为包装 (wrapper) 去重复数据集。例如，假设原始数据集是 `Dataset_A`，为了重复它，配置文件如下：

```
dataset_A_train = dict(
        type='RepeatDataset',
        times=N,
        dataset=dict(  # 这是 Dataset_A 数据集的原始配置
            type='Dataset_A',
            ...
            pipeline=train_pipeline
        )
    )
```

拼接数据集

有 2 种方式去拼接数据集。

1. 如果您想拼接的数据集是同样的类型，但有不同的标注文件，您可以按如下操作去拼接数据集的配置文件：

    1. 您也许可以拼接两个标注文件夹 `ann_dir`

    ```
    dataset_A_train = dict(
        type='Dataset_A',
        img_dir = 'img_dir',
        ann_dir = ['anno_dir_1', 'anno_dir_2'],
        pipeline=train_pipeline
    )
    ```

    2. 您也可以去拼接两个 `split` 文件列表

    ```
    dataset_A_train = dict(
        type='Dataset_A',
        img_dir = 'img_dir',
        ann_dir = 'anno_dir',
        split = ['split_1.txt', 'split_2.txt'],
    ```

(下页继续)

(续上页)

```
    pipeline=train_pipeline
)
```

3. 您也可以同时拼接 ann_dir 文件夹和 split 文件列表

```
dataset_A_train = dict(
    type='Dataset_A',
    img_dir = 'img_dir',
    ann_dir = ['anno_dir_1', 'anno_dir_2'],
    split = ['split_1.txt', 'split_2.txt'],
    pipeline=train_pipeline
)
```

在这样的情况下，ann_dir_1 和 ann_dir_2 分别对应于 split_1.txt 和 split_2.txt

2. 如果您想拼接不同的数据集，您可以如下去拼接数据集的配置文件：

```
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
    )
```

一个更复杂的例子如下：分别重复 Dataset_A 和 Dataset_B N 次和 M 次，然后再去拼接重复后的数据集

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

(下页继续)

(续上页)

```
    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
)
```

多图混合集

我们使用 `MultiImageMixDataset` 作为包装 (wrapper) 去混合多个数据集的图片。`MultiImageMixDataset` 可以被类似 mosaic 和 mixup 的多图混合数据增广使用。

`MultiImageMixDataset` 与 `Mosaic` 数据增广一起使用的例子:

```
train_pipeline = [
    dict(type='RandomMosaic', prob=1),
    dict(type='Resize', img_scale=(1024, 512), keep_ratio=True),
    dict(type='RandomFlip', prob=0.5),
    dict(type='Normalize', **img_norm_cfg),
    dict(type='DefaultFormatBundle'),
    dict(type='Collect', keys=['img', 'gt_semantic_seg']),
]
```

(下页继续)

(续上页)

```
train_dataset = dict(
    type='MultiImageMixDataset',
    dataset=dict(
        classes=classes,
        palette=palette,
        type=dataset_type,
        reduce_zero_label=False,
        img_dir=data_root + "images/train",
        ann_dir=data_root + "annotations/train",
        pipeline=[
            dict(type='LoadImageFromFile'),
            dict(type='LoadAnnotations'),
        ]
    ),
    pipeline=train_pipeline
)
```

5.3 自定义数据流程（待更新）

5.3.1 数据流程的设计

按照通常的惯例，我们使用 `Dataset` 和 `DataLoader` 做多线程的数据加载。`Dataset` 返回一个数据内容的字典，里面对应于模型前传方法的各个参数。因为在语义分割中，输入的图像数据具有不同的大小，我们在 MMCV 里引入一个新的 `DataContainer` 类别去帮助收集和分发不同大小的输入数据。

更多细节，请查看这里 。

数据的准备流程和数据集是解耦的。通常一个数据集定义了如何处理标注数据（annotations）信息，而一个数据流程定义了准备一个数据字典的所有步骤。一个流程包括了一系列操作，每个操作里都把一个字典作为输入，然后再输出一个新的字典给下一个变换操作。

这些操作可分为数据加载 (data loading)，预处理 (pre-processing)，格式变化 (formatting) 和测试时数据增强 (test-time augmentation)。

下面的例子就是 PSPNet 的一个流程：

```
img_norm_cfg = dict(
    mean=[123.675, 116.28, 103.53], std=[58.395, 57.12, 57.375], to_rgb=True)
crop_size = (512, 1024)
train_pipeline = [
    dict(type='LoadImageFromFile'),
    dict(type='LoadAnnotations'),
```

(下页继续)

(续上页)

```
    dict(type='Resize', img_scale=(2048, 1024), ratio_range=(0.5, 2.0)),
    dict(type='RandomCrop', crop_size=crop_size, cat_max_ratio=0.75),
    dict(type='RandomFlip', flip_ratio=0.5),
    dict(type='PhotoMetricDistortion'),
    dict(type='Normalize', **img_norm_cfg),
    dict(type='Pad', size=crop_size, pad_val=0, seg_pad_val=255),
    dict(type='DefaultFormatBundle'),
    dict(type='Collect', keys=['img', 'gt_semantic_seg']),
]
test_pipeline = [
    dict(type='LoadImageFromFile'),
    dict(
        type='MultiScaleFlipAug',
        img_scale=(2048, 1024),
        # img_ratios=[0.5, 0.75, 1.0, 1.25, 1.5, 1.75],
        flip=False,
        transforms=[
            dict(type='Resize', keep_ratio=True),
            dict(type='RandomFlip'),
            dict(type='Normalize', **img_norm_cfg),
            dict(type='ImageToTensor', keys=['img']),
            dict(type='Collect', keys=['img']),
        ])
]
```

对于每个操作，我们列出它添加、更新、移除的相关字典域 (dict fields)：

数据加载 Data loading

`LoadImageFromFile`

- 增加: img, img_shape, ori_shape

`LoadAnnotations`

- 增加: gt_semantic_seg, seg_fields

预处理 Pre-processing

`Resize`

- 增加: scale, scale_idx, pad_shape, scale_factor, keep_ratio
- 更新: img, img_shape, *seg_fields

`RandomFlip`

- 增加: flip
- 更新: img, *seg_fields

`Pad`

- 增加: pad_fixed_size, pad_size_divisor
- 更新: img, pad_shape, *seg_fields

`RandomCrop`

- 更新: img, pad_shape, *seg_fields

`Normalize`

- 增加: img_norm_cfg
- 更新: img

`SegRescale`

- 更新: gt_semantic_seg

`PhotoMetricDistortion`

- 更新: img

格式 Formatting

`ToTensor`

- 更新: 由 `keys` 指定

`ImageToTensor`

- 更新: 由 `keys` 指定

`Transpose`

- 更新: 由 `keys` 指定

`ToDataContainer`

- 更新: 由 `keys` 指定

`DefaultFormatBundle`

- 更新: img, gt_semantic_seg

`Collect`

- 增加: img_meta (the keys of img_meta is specified by `meta_keys`)
- 移除: all other keys except for those specified by `keys`

测试时数据增强 Test time augmentation

`MultiScaleFlipAug`

5.3.2 拓展和使用自定义的流程

1. 在任何一个文件里写一个新的流程，例如 `my_pipeline.py`，它以一个字典作为输入并且输出一个字典

```python
from mmseg.datasets import PIPELINES

@PIPELINES.register_module()
class MyTransform:

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

2. 导入一个新类

```python
from .my_pipeline import MyTransform
```

3. 在配置文件里使用它

```python
img_norm_cfg = dict(
    mean=[123.675, 116.28, 103.53], std=[58.395, 57.12, 57.375], to_rgb=True)
crop_size = (512, 1024)
train_pipeline = [
    dict(type='LoadImageFromFile'),
    dict(type='LoadAnnotations'),
    dict(type='Resize', img_scale=(2048, 1024), ratio_range=(0.5, 2.0)),
    dict(type='RandomCrop', crop_size=crop_size, cat_max_ratio=0.75),
    dict(type='RandomFlip', flip_ratio=0.5),
    dict(type='PhotoMetricDistortion'),
    dict(type='Normalize', **img_norm_cfg),
    dict(type='Pad', size=crop_size, pad_val=0, seg_pad_val=255),
    dict(type='MyTransform'),
```

(下页继续)

(续上页)

```
    dict(type='DefaultFormatBundle'),
    dict(type='Collect', keys=['img', 'gt_semantic_seg']),
]
```

5.4 自定义运行设定（待更新）

5.4.1 自定义优化设定

自定义 PyTorch 支持的优化器

我们已经支持 PyTorch 自带的所有优化器，唯一需要修改的地方是在配置文件里的 `optimizer` 域里面。例如，如果您想使用 `ADAM` (注意如下操作可能会让模型表现下降)，可以使用如下修改：

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

为了修改模型的学习率，使用者仅需要修改配置文件里 optimizer 的 `lr` 即可。使用者可以参照 PyTorch 的 API 文档 直接设置参数。

自定义自己实现的优化器

1. 定义一个新的优化器

一个自定义的优化器可以按照如下去定义：

假如您想增加一个叫做 `MyOptimizer` 的优化器，它的参数分别有 `a`, `b`, 和 `c`。您需要创建一个叫 `mmseg/core/optimizer` 的新文件夹。然后再在文件，即 `mmseg/core/optimizer/my_optimizer.py` 里面去实现这个新优化器：

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


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

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

2. 增加优化器到注册表 (registry)

为了让上述定义的模块被框架发现，首先这个模块应该被导入到主命名空间 (main namespace) 里。有两种方式可以实现它。

- 修改 mmseg/core/optimizer/__init__.py 来导入它

  新的被定义的模块应该被导入到 mmseg/core/optimizer/__init__.py 这样注册表将会发现新的模块并添加它

```
from .my_optimizer import MyOptimizer
```

- 在配置文件里使用 custom_imports 去手动导入它

```
custom_imports = dict(imports=['mmseg.core.optimizer.my_optimizer'], allow_failed_
↪imports=False)
```

mmseg.core.optimizer.my_optimizer 模块将会在程序运行的开始被导入，并且 MyOptimizer 类将会自动注册。需要注意只有包含 MyOptimizer 类的包 (package) 应当被导入。而 mmseg.core.optimizer.my_optimizer.MyOptimizer **不能**被直接导入。

事实上，使用者完全可以用另一个按这样导入方法的文件夹结构，只要模块的根路径已经被添加到 PYTHONPATH 里面。

3. 在配置文件里定义优化器

之后您可以在配置文件的 optimizer 域里面使用 MyOptimizer 在配置文件里，优化器被定义在 optimizer 域里，如下所示：

```
optimizer = dict(type='SGD', lr=0.02, momentum=0.9, weight_decay=0.0001)
```

为了使用您自己的优化器，这个域可以被改成：

```
optimizer = dict(type='MyOptimizer', a=a_value, b=b_value, c=c_value)
```

自定义优化器的构造器 (constructor)

有些模型可能需要在优化器里有一些特别参数的设置，例如批归一化层 (BatchNorm layers) 的权重衰减 (weight decay)。使用者可以通过自定义优化器的构造器去微调这些细粒度参数。

```
from mmcv.utils import build_from_cfg

from mmcv.runner.optimizer import OPTIMIZER_BUILDERS, OPTIMIZERS
from mmseg.utils import get_root_logger
```

(下页继续)

(续上页)

```
from .my_optimizer import MyOptimizer


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

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

    def __call__(self, model):

        return my_optimizer
```

默认的优化器构造器的实现可以参照 这里 ，它也可以被用作新的优化器构造器的模板。

额外的设置

优化器没有实现的一些技巧应该通过优化器构造器 (optimizer constructor) 或者钩子 (hook) 去实现，如设置基于参数的学习率 (parameter-wise learning rates)。我们列出一些常见的设置，它们可以稳定或加速模型的训练。如果您有更多的设置，欢迎在 PR 和 issue 里面提交。

- **使用梯度截断 (gradient clip) 去稳定训练**:

  一些模型需要梯度截断去稳定训练过程，如下所示

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

  如果您的配置继承自已经设置了 `optimizer_config` 的基础配置 (base config)，您可能需要 `_delete_=True` 来重写那些不需要的设置。更多细节请参照 配置文件文档 。

- **使用动量计划表 (momentum schedule) 去加速模型收敛**:

  我们支持动量计划表去让模型基于学习率修改动量，这样可能让模型收敛地更快。动量计划表经常和学习率计划表 (LR scheduler) 一起使用，例如如下配置文件就在 3D 检测里经常使用以加速收敛。更多细节请参考 CyclicLrUpdater 和 CyclicMomentumUpdater 的实现。

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

(下页继续)

(续上页)

```
    target_ratio=(0.85 / 0.95, 1),
    cyclic_times=1,
    step_ratio_up=0.4,
)
```

5.4.2 自定义训练计划表

我们根据默认的训练迭代步数 40k/80k 来设置学习率，这在 MMCV 里叫做 `PolyLrUpdaterHook` 。我们也支持许多其他的学习率计划表：这里 ，例如 `CosineAnnealing` 和 `Poly` 计划表。下面是一些例子：

- 步计划表 Step schedule:

```
lr_config = dict(policy='step', step=[9, 10])
```

- 余弦退火计划表 ConsineAnnealing schedule:

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

5.4.3 自定义工作流 (workflow)

工作流是一个专门定义运行顺序和轮数 (running order and epochs) 的列表 (phase, epochs)。默认情况下它设置成：

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

意思是训练是跑 1 个 epoch。有时候使用者可能想检查模型在验证集上的一些指标（如损失 loss，精确性 accuracy），我们可以这样设置工作流：

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

于是 1 个 epoch 训练，1 个 epoch 验证将交替运行。

注意:

1. 模型的参数在验证的阶段不会被自动更新
2. 配置文件里的关键词 `total_epochs` 仅控制训练的 epochs 数目，而不会影响验证时的工作流
3. 工作流 `[('train', 1), ('val', 1)]` 和 `[('train', 1)]` 将不会改变 `EvalHook` 的行为，因为 `EvalHook` 被 `after_train_epoch` 调用而且验证的工作流仅仅影响通过调用 `after_val_epoch`

的钩子 (hooks)。因此，`[('train', 1), ('val', 1)]` 和 `[('train', 1)]` 的区别仅在于 runner 将在每次训练 epoch 结束后计算在验证集上的损失

5.4.4 自定义钩 (hooks)

使用 MMCV 实现的钩子 (hooks)

如果钩子已经在 MMCV 里被实现，如下所示，您可以直接修改配置文件来使用钩子：

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

修改默认的运行时间钩子 (runtime hooks)

以下的常用的钩子没有被 `custom_hooks` 注册：

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

在这些钩子里，只有 logger hook 有 `VERY_LOW` 优先级，其他的优先级都是 `NORMAL`。上述提及的教程已经包括了如何修改 `optimizer_config`，`momentum_config` 和 `lr_config`。这里我们展示我们如何处理 `log_config`，`checkpoint_config` 和 `evaluation`。

检查点配置文件 (Checkpoint config)

MMCV runner 将使用 `checkpoint_config` 去初始化 `CheckpointHook`.

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

使用者可以设置 `max_keep_ckpts` 来仅保存一小部分检查点或者通过 `save_optimizer` 来决定是否保存优化器的状态字典 (state dict of optimizer)。更多使用参数的细节请参考 这里 。

日志配置文件 (Log config)

log_config 包裹了许多日志钩 (logger hooks) 而且能去设置间隔 (intervals)。现在 MMCV 支持 WandbLoggerHook，MlflowLoggerHook 和 TensorboardLoggerHook。详细的使用请参照 文档 。

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

评估配置文件 (Evaluation config)

evaluation 的配置文件将被用来初始化 EvalHook 。除了 interval 键，其他的像 metric 这样的参数将被传递给 dataset.evaluate() 。

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

CHAPTER 6

迁移文档

CHAPTER 7

mmseg.apis

CHAPTER 8

mmseg.datasets

8.1 datasets

8.2 transforms

CHAPTER 9

mmseg.engine

9.1 hooks

9.2 optimizers

CHAPTER 10

mmseg.evaluation

10.1 metrics

CHAPTER 11

mmseg.models

CHAPTER 12

mmseg.ops

CHAPTER 13

mmseg.registry

CHAPTER 14

mmseg.structures

14.1 structures

14.2 sampler

CHAPTER 15

mmseg.utils

CHAPTER 16

mmseg.visualization

CHAPTER 17

标准与模型库

17.1 共同设定

- 我们默认使用 4 卡分布式训练
- 所有 PyTorch 风格的 ImageNet 预训练网络由我们自己训练，和 论文 保持一致。我们的 ResNet 网络是基于 ResNetV1c 的变种，在这里输入层的 7x7 卷积被 3 个 3x3 取代
- 为了在不同的硬件上保持一致，我们以 `torch.cuda.max_memory_allocated()` 的最大值作为 GPU 占用率，同时设置 `torch.backends.cudnn.benchmark=False`。注意，这通常比 `nvidia-smi` 显示的要少
- 我们以网络 forward 和后处理的时间加和作为推理时间，除去数据加载时间。我们使用脚本 `tools/benchmark.py` 来获取推理时间，它在 `torch.backends.cudnn.benchmark=False` 的设定下，计算 200 张图片的平均推理时间
- 在框架中，有两种推理模式
  - `slide` 模式（滑动模式）：测试的配置文件字段 `test_cfg` 会是 `dict(mode='slide', crop_size=(769, 769), stride=(513, 513))`. 在这个模式下，从原图中裁剪多个小图分别输入网络中进行推理。小图的大小和小图之间的距离由 `crop_size` 和 `stride` 决定，重合区域会进行平均
  - `whole` 模式（全图模式）：测试的配置文件字段 `test_cfg` 会是 `dict(mode='whole')`. 在这个模式下，全图会被直接输入到网络中进行推理。对于 769x769 下训练的模型，我们默认使用 `slide` 进行推理，其余模型用 `whole` 进行推理

- 对于输入大小为 8x+1（比如 769），我们使用 `align_corners=True`。其余情况，对于输入大小为 8x (比如 512，1024)，我们使用 `align_corners=False`

17.2 基线

17.2.1 FCN

请参考 FCN 获得详细信息。

17.2.2 PSPNet

请参考 PSPNet 获得详细信息。

17.2.3 DeepLabV3

请参考 DeepLabV3 获得详细信息。

17.2.4 PSANet

请参考 PSANet 获得详细信息。

17.2.5 DeepLabV3+

请参考 DeepLabV3+ 获得详细信息。

17.2.6 UPerNet

请参考 UPerNet 获得详细信息。

17.2.7 NonLocal Net

请参考 NonLocal Net 获得详细信息。

17.2.8 EncNet

请参考 EncNet 获得详细信息。

17.2.9 CCNet

请参考 CCNet 获得详细信息。

17.2.10 DANet

请参考 DANet 获得详细信息。

17.2.11 APCNet

请参考 APCNet 获得详细信息。

17.2.12 HRNet

请参考 HRNet 获得详细信息。

17.2.13 GCNet

请参考 GCNet 获得详细信息。

17.2.14 DMNet

请参考 DMNet 获得详细信息。

17.2.15 ANN

请参考 ANN 获得详细信息。

17.2.16 OCRNet

请参考 OCRNet 获得详细信息。

17.2.17 Fast-SCNN

请参考 Fast-SCNN 获得详细信息。

17.2.18 ResNeSt

请参考 ResNeSt 获得详细信息。

17.2.19 Semantic FPN

请参考 Semantic FPN 获得详细信息。

17.2.20 PointRend

请参考 PointRend 获得详细信息。

17.2.21 MobileNetV2

请参考 MobileNetV2 获得详细信息。

17.2.22 MobileNetV3

请参考 MobileNetV3 获得详细信息。

17.2.23 EMANet

请参考 EMANet 获得详细信息。

17.2.24 DNLNet

请参考 DNLNet 获得详细信息。

17.2.25 CGNet

请参考 CGNet 获得详细信息。

17.2.26 Mixed Precision (FP16) Training

请参考 Mixed Precision (FP16) Training 在 BiSeNetV2 训练的样例 获得详细信息。

17.3 速度标定（待更新）

17.3.1 硬件

- 8 NVIDIA Tesla V100 (32G) GPUs
- Intel(R) Xeon(R) Gold 6148 CPU @ 2.40GHz

17.3.2 软件环境

- Python 3.7
- PyTorch 1.5
- CUDA 10.1
- CUDNN 7.6.03
- NCCL 2.4.08

17.3.3 训练速度

为了公平比较，我们全部使用 ResNet-101V1c 进行标定。输入大小为 1024x512，批量样本数为 2。

训练速度如下表，指标为每次迭代的时间，以秒为单位，越低越快。

注意：DeepLabV3+ 的输出步长为 8。

CHAPTER 18

模型库统计数据

- 论文数量: 44
  - ALGORITHM: 34
  - BACKBONE: 10
- 模型数量: 590
  - [ALGORITHM] ANN (16 ckpts)
  - [ALGORITHM] APCNet (12 ckpts)
  - [BACKBONE] BEiT (2 ckpts)
  - [ALGORITHM] BiSeNetV1 (11 ckpts)
  - [ALGORITHM] BiSeNetV2 (4 ckpts)
  - [ALGORITHM] CCNet (16 ckpts)
  - [ALGORITHM] CGNet (2 ckpts)
  - [BACKBONE] ConvNeXt (6 ckpts)
  - [ALGORITHM] DANet (16 ckpts)
  - [ALGORITHM] DeepLabV3 (41 ckpts)
  - [ALGORITHM] DeepLabV3+ (42 ckpts)
  - [ALGORITHM] DMNet (12 ckpts)
  - [ALGORITHM] DNLNet (12 ckpts)

- [ALGORITHM] DPT (1 ckpts)
- [ALGORITHM] EMANet (4 ckpts)
- [ALGORITHM] EncNet (12 ckpts)
- [ALGORITHM] ERFNet (1 ckpts)
- [ALGORITHM] FastFCN (12 ckpts)
- [ALGORITHM] Fast-SCNN (1 ckpts)
- [ALGORITHM] FCN (41 ckpts)
- [ALGORITHM] GCNet (16 ckpts)
- [BACKBONE] HRNet (37 ckpts)
- [ALGORITHM] ICNet (12 ckpts)
- [ALGORITHM] ISANet (16 ckpts)
- [ALGORITHM] K-Net (7 ckpts)
- [BACKBONE] MAE (1 ckpts)
- [BACKBONE] MobileNetV2 (8 ckpts)
- [BACKBONE] MobileNetV3 (4 ckpts)
- [ALGORITHM] NonLocal Net (16 ckpts)
- [ALGORITHM] OCRNet (24 ckpts)
- [ALGORITHM] PointRend (4 ckpts)
- [ALGORITHM] PSANet (16 ckpts)
- [ALGORITHM] PSPNet (54 ckpts)
- [BACKBONE] ResNeSt (8 ckpts)
- [ALGORITHM] SegFormer (13 ckpts)
- [ALGORITHM] Segmenter (5 ckpts)
- [ALGORITHM] Semantic FPN (4 ckpts)
- [ALGORITHM] SETR (7 ckpts)
- [ALGORITHM] STDC (4 ckpts)
- [BACKBONE] Swin Transformer (6 ckpts)
- [BACKBONE] Twins (12 ckpts)
- [ALGORITHM] UNet (25 ckpts)
- [ALGORITHM] UPerNet (16 ckpts)

- [BACKBONE] Vision Transformer (11 ckpts)

CHAPTER 19

English

CHAPTER 20

简体中文

CHAPTER 21

Indices and tables

- genindex
- search