
MMSegmentation

发布 *0.30.0*

MMSegmentation Authors

2023 年 07 月 22 日

1	依赖	1
2	安装	3
2.1	最佳实践	3
2.2	验证安装	4
2.3	自定义安装	5
2.4	故障解决	7
3	准备数据集	9
3.1	Cityscapes	12
3.2	Pascal VOC	13
3.3	ADE20K	13
3.4	Pascal Context	13
3.5	CHASE DB1	13
3.6	DRIVE	14
3.7	HRF	14
3.8	STARE	14
3.9	Dark Zurich	14
3.10	Nighttime Driving	15
3.11	LoveDA	15
3.12	ISPRS Potsdam	15
3.13	ISPRS Vaihingen	16
3.14	iSAID	16
3.15	ImageNetS	17
4	标准与模型库	19
4.1	共同设定	19
4.2	基线	20

4.3	速度标定	23
5	模型库统计数据	25
6	训练一个模型	29
6.1	使用单台机器训练	30
6.2	使用多台机器训练	31
6.3	使用 slurm 管理任务	32
7	使用预训练模型推理	35
7.1	测试一个数据集	35
8	教程 1: 学习配置文件	39
8.1	配置文件的结构	39
8.2	配置文件命名风格	40
8.3	PSPNet 的一个例子	40
8.4	FAQ	46
9	教程 2: 自定义数据集	49
9.1	通过重新组织数据来定制数据集	49
9.2	通过混合数据去定制数据集	50
10	教程 3: 自定义数据流程	55
10.1	数据流程的设计	55
10.2	拓展和使用自定义的流程	58
11	教程 4: 自定义模型	59
11.1	自定义优化器 (optimizer)	59
11.2	定制优化器的构造器 (optimizer constructor)	60
11.3	开发和增加新的组件 (Module)	60
12	教程 5: 训练技巧	65
12.1	主干网络和解码头组件使用不同的学习率 (Learning Rate, LR)	65
12.2	在线难样本挖掘 (Online Hard Example Mining, OHEM)	66
12.3	类别平衡损失 (Class Balanced Loss)	66
12.4	同时使用多种损失函数 (Multiple Losses)	66
12.5	在损失函数中忽略特定的 label 类别	67
13	教程 6: 自定义运行设定	69
13.1	自定义优化设定	69
13.2	自定义训练计划表	72
13.3	自定义工作流 (workflow)	72
13.4	自定义钩 (hooks)	73
14	常用工具	75

14.1	计算参数量 (params) 和计算量 (FLOPs) (试验性)	75
14.2	发布模型	76
14.3	导出 ONNX (试验性)	76
14.4	评估 ONNX 模型	77
14.5	导出 TorchScript (试验性)	78
14.6	导出 TensorRT (试验性)	79
15	其他内容	81
15.1	打印完整的配置文件	81
15.2	对训练日志 (training logs) 画图	81
15.3	转换其他仓库的权重	82
16	模型服务	83
16.1	1. 将 model 从 MMSegmentation 转换到 TorchServe	83
16.2	2. 构建 mmseg-serve 容器镜像 (docker image)	83
16.3	3. 运行 mmseg-serve	84
16.4	4. 测试部署	84
17	模型集成	87
17.1	使用方法	87
17.2	各个参数的描述:	87
17.3	模型集成结果	88
18	常见问题解答 (FAQ)	89
18.1	安装	89
18.2	如何获知模型训练时需要的显卡数量	90
18.3	auxiliary head 是什么	90
18.4	为什么日志文件没有被创建	90
18.5	运行测试脚本时如何输出绘制分割掩膜的图像	90
18.6	如何处理二值分割任务?	91
18.7	reduce_zero_label 的作用	92
19	NPU (华为昇腾)	93
19.1	使用方法	93
19.2	经过验证的模型	93
20	English	95
21	简体中文	97
22	mmseg.apis	99
23	mmseg.core	101
23.1	seg	101
23.2	evaluation	101

23.3	utils	101
24	mmseg.datasets	103
24.1	datasets	103
24.2	pipelines	103
25	mmseg.models	105
25.1	segmentors	105
25.2	backbones	105
25.3	decode_heads	105
25.4	losses	105
26	Indices and tables	107

CHAPTER 1

依赖

在本节中，我们将演示如何用 PyTorch 准备一个环境。

MMSegmentation 可以在 Linux、Windows 和 MacOS 上运行。它需要 Python 3.6 以上，CUDA 9.2 以上和 PyTorch 1.3 以上。

注解：如果您对 PyTorch 有经验并且已经安装了它，请跳到下一节。否则，您可以按照以下步骤进行准备。

第一步从[官方网站](#)下载并安装 Miniconda。

第二步创建并激活一个 conda 环境。

```
conda create --name openmmlab python=3.8 -y
conda activate openmmlab
```

第三步按照[官方说明](#)安装 PyTorch。

在 GPU 平台上：

```
conda install pytorch torchvision -c pytorch
```

在 CPU 平台上：

```
conda install pytorch torchvision cpuonly -c pytorch
```


我们建议用户遵循我们的最佳实践来安装 MMSegmentation，同时整个过程是高度可定制的。更多信息见自定义安装部分。

2.1 最佳实践

第一步使用 MIM 安装 MMCV

```
pip install -U openmim
mim install mmcv-full
```

第二步安装 MMSegmentation

根据具体需求，我们支持两种安装模式：

- **从源码安装（推荐）**：如果基于 MMSegmentation 框架开发自己的任务，需要添加新的功能，比如新的模型或是数据集，或者使用我们提供的各种工具。
- **作为 Python 包安装**：只是希望调用 MMSegmentation 的接口，或者在自己的项目中导入 MMSegmentation 中的模块。

2.1.1 从源码安装

```
git clone https://github.com/open-mmlab/msegmentation.git
cd msegmentation
pip install -v -e .
# "-v" 指详细说明, 或更多的输出
# "-e" 表示在可编辑模式下安装项目, 因此对代码所做的任何本地修改都会生效, 从而无需重新安装。
```

2.1.2 作为 Python 包安装

```
pip install msegmentation
```

注意: 如果你想使用 `albuementations`, 我们建议使用 `pip install-U albuementations --no-binary qudida,albuementations` 进行安装。如果您仅使用 `pip install albuementations>=0.3.2` 进行安装, 它将同时安装 `opencv-python-headless` (即使您已经安装了 `opencv-python`)。我们建议在安装了 `albuementations` 后检查环境, 以确保没有同时安装 `opencv-python` 和 `opencv-python-headless`, 因为如果两者都安装了, 可能会导致意外问题。请参阅[官方文档](#)了解更多信息。

2.2 验证安装

为了验证 `MMSegmentation` 是否安装正确, 我们提供了一些示例代码来执行模型推理。

第一步我们需要下载配置文件和模型权重文件。

```
mim download msegmentation --config pspnet_r50-d8_512x1024_40k_cityscapes --dest .
```

下载将需要几秒钟或更长时间, 这取决于你的网络环境。完成后, 你会在当前文件夹中发现两个文件 `pspnet_r50-d8_512x1024_40k_cityscapes.py` 和 `pspnet_r50-d8_512x1024_40k_cityscapes_20200605_003338-2966598c.pth`。

第二步验证推理示例

如果您是**从源码安装**的 `MMSegmentation`, 那么直接运行以下命令进行验证:

```
python demo/image_demo.py demo/demo.png pspnet_r50-d8_512x1024_40k_cityscapes.py_
↪pspnet_r50-d8_512x1024_40k_cityscapes_20200605_003338-2966598c.pth --device cpu --
↪out-file result.jpg
```

你会在你的当前文件夹中看到一个新的图像 `result.jpg`, 其中的分割掩膜覆盖在所有对象上。

如果您是**作为 Python 包安装**, 那么可以打开您的 Python 解释器, 复制并粘贴如下代码:

```
from mmseg.apis import inference_segmentor, init_segmentor
import mmcv

config_file = 'pspnet_r50-d8_512x1024_40k_cityscapes.py'
checkpoint_file = 'pspnet_r50-d8_512x1024_40k_cityscapes_20200605_003338-2966598c.pth'

# 通过配置文件和模型权重文件构建模型
model = init_segmentor(config_file, checkpoint_file, device='cuda:0')

# 对单张图片进行推理并展示结果
img = 'test.jpg' # or img = mmcv.imread(img), which will only load it once
result = inference_segmentor(model, img)
# 在新窗口中可视化推理结果
model.show_result(img, result, show=True)
# 或将可视化结果存储在文件中
# 你可以修改 opacity 在 (0,1] 之间的取值来改变绘制好的分割图的透明度
model.show_result(img, result, out_file='result.jpg', opacity=0.5)

# 对视频进行推理并展示结果
video = mmcv.VideoReader('video.mp4')
for frame in video:
    result = inference_segmentor(model, frame)
    model.show_result(frame, result, wait_time=1)
```

你可以修改上面的代码来测试一张图片或一段视频，这两种方式都可以验证安装是否成功。

2.3 自定义安装

2.3.1 CUDA 版本

在安装 PyTorch 时，您需要指定 CUDA 的版本。如果您不清楚应该选择哪一个，请遵循我们的建议。

- 对于 Ampere 架构的 NVIDIA GPU，例如 GeForce 30 系列以及 NVIDIA A100，CUDA 11 是必需的。
- 对于更早的 NVIDIA GPU，CUDA 11 是向后兼容 (backward compatible) 的，但 CUDA 10.2 能够提供更好的兼容性，也更加轻量。

请确保您的 GPU 驱动版本满足最低的版本需求，参阅[这张表](#)。

注解：如果按照我们的最佳实践进行安装，CUDA 运行时库就足够了，因为我们提供相关 CUDA 代码的预编译，您不需要进行本地编译。但如果您希望从源码进行 MMCV 的编译，或是进行其他 CUDA 算子的开发，那么就必须安装完整的 CUDA 工具链，参见 [NVIDIA 官网](#)，另外还需要确保该 CUDA 工具链的版本与 PyTorch

安装时的配置相匹配（如用 `conda install` 安装 PyTorch 时指定的 `cuda-toolkit` 版本）。

2.3.2 不使用 MIM 安装 MMCV

MMCV 包含 C++ 和 CUDA 扩展，因此其对 PyTorch 的依赖比较复杂。MIM 会自动解析这些依赖，选择合适的 MMCV 预编译包，使安装更简单，但它并不是必需的。

要使用 `pip` 而不是 MIM 来安装 MMCV，请遵照 [MMCV 安装指南](#)。它需要您用指定 `url` 的形式手动指定对应的 PyTorch 和 CUDA 版本。

举个例子，如下命令将会安装基于 PyTorch 1.10.x 和 CUDA 11.3 编译的 `mmcv-full`。

```
pip install mmcv-full -f https://download.openmmlab.com/mmcv/dist/cu113/torch1.10/
↪ index.html
```

2.3.3 在 CPU 环境中安装

MMSegmentation 可以仅在 CPU 环境中安装，在 CPU 模式下，您可以完成训练（需要 MMCV 版本 $\geq 1.4.4$ ）、测试和模型推理等所有操作。

2.3.4 在 Google Colab 中安装

Google Colab 通常已经包含了 PyTorch 环境，因此我们只需要安装 MMCV 和 MMSegmentation 即可，命令如下：

第一步使用 MIM 安装 MMCV

```
!pip3 install openmim
!mim install mmcv-full
```

第二步从源码安装 MMSegmentation

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

第三步验证

```
import mmdet
print(mmdet.__version__)
# 预期输出: 0.24.1 或其他版本号
```

注解: 在 Jupyter 中, 感叹号 ! 用于执行外部命令, 而 %cd 是一个魔法命令, 用于切换 Python 的工作路径。

2.3.5 通过 Docker 使用 MMSegmentation

我们提供了一个 Dockerfile 来构建一个镜像。请确保你的 docker 版本 ≥ 19.03 。

```
# build an image with PyTorch 1.11, CUDA 11.3
# If you prefer other versions, just modified the Dockerfile
docker build -t mmsegmentation docker/
```

用以下命令运行 Docker 镜像:

```
docker run --gpus all --shm-size=8g -it -v {DATA_DIR}:/mmpose/data mmpose
```

2.4 故障解决

如果你在安装过程中遇到一些问题, 请先查看 [FAQ](#) 页面。

如果没有找到解决方案, 你也可以在 [GitHub](#) 上打开一个问题。

CHAPTER 3

准备数据集

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
├── tools
├── configs
├── data
│   ├── cityscapes
│   │   ├── leftImg8bit
│   │   │   ├── train
│   │   │   └── val
│   │   ├── gtFine
│   │   │   ├── train
│   │   │   └── val
│   ├── VOCdevkit
│   │   ├── VOC2012
│   │   │   ├── JPEGImages
│   │   │   ├── SegmentationClass
│   │   │   ├── ImageSets
│   │   │   └── Segmentation
│   │   └── VOC2010
│   │       ├── JPEGImages
│   │       └── SegmentationClassContext
```

(下页继续)

(续上页)

```

| | | | | 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
| | | | images
| | | | | training
| | | | | validation
| | | | annotations
| | | | | training
| | | | | validation
| | | HRF
| | | | images
| | | | | training
| | | | | validation
| | | | annotations
| | | | | training
| | | | | validation
| | | STARE
| | | | images
| | | | | training
| | | | | validation
| | | | annotations
| | | | training

```

(下页继续)

(续上页)

```

| | | └─ 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
| | | └─ train
| | | └─ val
| | | └─ test
| | └─ ann_dir
| | | └─ train
| | | └─ val
| └─ potsdam
| | └─ img_dir
| | | └─ train
| | | └─ val
| | └─ ann_dir
| | | └─ train
| | | └─ val
| └─ vaihingen
| | └─ img_dir
| | | └─ train
| | | └─ val
| | └─ ann_dir
| | | └─ train

```

(下页继续)

(续上页)

```

| | | | └─ val
| | | └─ iSAID
| | | └─ img_dir
| | | └─ train
| | | └─ val
| | | └─ test
| | └─ ann_dir
| | └─ train
| | └─ val
| └─ ImageNetS
| | └─ ImageNetS919
| | | └─ train-semi
| | | └─ train-semi-segmentation
| | | └─ validation
| | | └─ validation-segmentation
| | | └─ test
| | └─ ImageNetS300
| | | └─ train-semi
| | | └─ train-semi-segmentation
| | | └─ validation
| | | └─ validation-segmentation
| | | └─ test
| | └─ ImageNetS50
| | | └─ train-semi
| | | └─ train-semi-segmentation
| | | └─ validation
| | | └─ validation-segmentation
| | | └─ test

```

3.1 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

```

3.2 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](#)。

3.3 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。

3.4 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
```

3.5 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。

3.6 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。

3.7 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。

3.8 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。

3.9 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](#)。

3.10 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。

3.11 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。

3.12 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。

3.13 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。

3.14 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
```

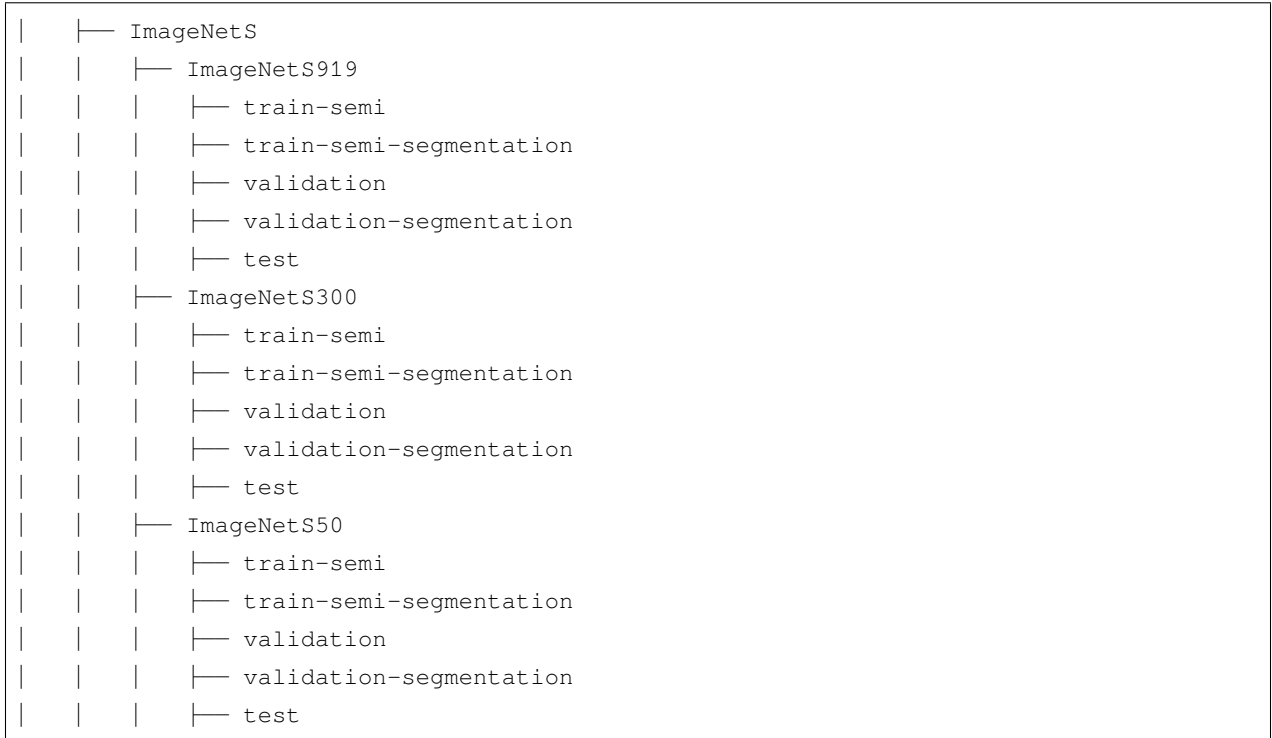
```
python tools/convert_datasets/isaid.py /path/to/iSAID
```

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。

3.15 ImageNetS

ImageNet-S 是用于大规模无监督/半监督语义分割任务的数据集。

ImageNet-S 数据集可在[ImageNet-S](#)获取。



4.1 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 8×1 (比如 769), 我们使用 `align_corners=True`。其余情况, 对于输入大小为 $8 \times$ (比如 512, 1024), 我们使用 `align_corners=False`

4.2 基线

4.2.1 FCN

请参考 [FCN](#) 获得详细信息。

4.2.2 PSPNet

请参考 [PSPNet](#) 获得详细信息。

4.2.3 DeepLabV3

请参考 [DeepLabV3](#) 获得详细信息。

4.2.4 PSANet

请参考 [PSANet](#) 获得详细信息。

4.2.5 DeepLabV3+

请参考 [DeepLabV3+](#) 获得详细信息。

4.2.6 UPerNet

请参考 [UPerNet](#) 获得详细信息。

4.2.7 NonLocal Net

请参考 [NonLocal Net](#) 获得详细信息。

4.2.8 EncNet

请参考 [EncNet](#) 获得详细信息。

4.2.9 CCNet

请参考 [CCNet](#) 获得详细信息。

4.2.10 DANet

请参考 [DANet](#) 获得详细信息。

4.2.11 APCNet

请参考 [APCNet](#) 获得详细信息。

4.2.12 HRNet

请参考 [HRNet](#) 获得详细信息。

4.2.13 GCNet

请参考 [GCNet](#) 获得详细信息。

4.2.14 DMNet

请参考 [DMNet](#) 获得详细信息。

4.2.15 ANN

请参考 [ANN](#) 获得详细信息。

4.2.16 OCRNet

请参考 [OCRNet](#) 获得详细信息。

4.2.17 Fast-SCNN

请参考 [Fast-SCNN](#) 获得详细信息。

4.2.18 ResNeSt

请参考 [ResNeSt](#) 获得详细信息。

4.2.19 Semantic FPN

请参考 [Semantic FPN](#) 获得详细信息。

4.2.20 PointRend

请参考 [PointRend](#) 获得详细信息。

4.2.21 MobileNetV2

请参考 [MobileNetV2](#) 获得详细信息。

4.2.22 MobileNetV3

请参考 [MobileNetV3](#) 获得详细信息。

4.2.23 EMANet

请参考 [EMANet](#) 获得详细信息。

4.2.24 DNLNet

请参考 [DNLNet](#) 获得详细信息。

4.2.25 CGNet

请参考 [CGNet](#) 获得详细信息。

4.2.26 Mixed Precision (FP16) Training

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

4.3 速度标定

4.3.1 硬件

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

4.3.2 软件环境

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

4.3.3 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。

模型库统计数据

- 论文数量: 47
 - ALGORITHM: 35
 - BACKBONE: 11
 - DATASET: 1
- 模型数量: 610
 - [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)
- [DATASET] [ImageNet-S](#) (3 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)
- [BACKBONE] [PoolFormer](#) (5 ckpts)
- [ALGORITHM] [PSANet](#) (16 ckpts)
- [ALGORITHM] [PSPNet](#) (54 ckpts)
- [BACKBONE] [ResNeSt](#) (8 ckpts)
- [ALGORITHM] [SegFormer](#) (13 ckpts)
- [ALGORITHM] [Segmenter](#) (5 ckpts)
- [ALGORITHM] [SegNeXt](#) (4 ckpts)
- [ALGORITHM] [Semantic FPN](#) (4 ckpts)
- [ALGORITHM] [SETR](#) (7 ckpts)
- [ALGORITHM] [STDC](#) (4 ckpts)

- [BACKBONE] Swin Transformer (8 ckpts)
- [BACKBONE] Twins (12 ckpts)
- [ALGORITHM] UNet (25 ckpts)
- [ALGORITHM] UPerNet (22 ckpts)
- [BACKBONE] Vision Transformer (11 ckpts)

CHAPTER 6

训练一个模型

MMSegmentation 可以执行分布式训练和非分布式训练，分别使用 `MMDistributedDataParallel` 和 `MMDDataParallel` 命令。

所有的输出（日志 `log` 和检查点 `checkpoints`）将被保存到工作路径文件夹里，它可以通过配置文件里的 `work_dir` 指定。

在一定迭代轮次后，我们默认在验证集上评估模型表现。您可以在训练配置文件中添加间隔参数来改变评估间隔。

```
evaluation = dict(interval=4000) # 每 4000 iterations 评估一次模型的性能
```

*** 重要提示 ***: 在配置文件里的默认学习率是针对 4 卡 GPU 和 2 张图/GPU (此时 `batchsize = 4x2 = 8`) 来设置的。同样，您也可以使用 8 卡 GPU 和 1 张图/GPU 的设置，因为所有的模型均使用 cross-GPU 的 SyncBN 模式。

我们可以在训练速度和 GPU 显存之间做平衡。当模型或者 Batch Size 比较大的时，可以传递 `--cfg-options model.backbone.with_cp=True`，使用 `with_cp` 来节省显存，但是速度会更慢，因为原先使用 `with_cp` 时，是逐层反向传播 (Back Propagation, BP)，不会保存所有的梯度。

6.1 使用单台机器训练

6.1.1 使用单卡 GPU 训练

```
python tools/train.py ${CONFIG_FILE} [可选参数]
```

如果您想在命令里定义工作文件夹路径，您可以添加一个参数`--work-dir ${工作路径}`。

6.1.2 使用 CPU 训练

如果计算机没有 GPU，那么使用 CPU 训练的流程和使用单 GPU 训练的流程一致。如果计算机有 GPU 但是想使用 CPU，我们仅需要在训练流程开始前禁用 GPU。

```
export CUDA_VISIBLE_DEVICES=-1
```

之后运行单 GPU 训练脚本即可。

警告：我们不推荐用户使用 CPU 进行训练，这太过缓慢。我们支持这个功能是为了方便用户在没有 GPU 的机器上进行调试。

6.1.3 使用多卡 GPU 训练

```
sh tools/dist_train.sh ${CONFIG_FILE} ${GPUS} [可选参数]
```

可选参数可以为：

- `--no-validate` (**不推荐**): 训练时代码库默认会在每 `k` 轮迭代后在验证集上进行评估，如果不需评估使用命令 `--no-validate`
- `--work-dir ${工作路径}`: 在配置文件里重写工作路径文件夹
- `--resume-from ${检查点文件}`: 继续使用先前的检查点 (checkpoint) 文件 (可以继续训练过程)
- `--load-from ${检查点文件}`: 从一个检查点 (checkpoint) 文件里加载权重 (对另一个任务进行精调)
- `--deterministic`: 选择此模式会减慢训练速度，但结果易于复现

`resume-from` 和 `load-from` 的区别:

- `resume-from` 加载出模型权重和优化器状态包括迭代轮数等
- `load-from` 仅加载模型权重，从第 0 轮开始训练

示例:

```
# 模型的权重和日志将会存储在这个路径下: WORK_DIR=work_dirs/pspnet_r50-d8_512x512_80k_ade20k/
# 如果 work_dir 没有被设定, 它将会被自动生成
sh tools/dist_train.sh configs/pspnet/pspnet_r50-d8_512x512_80k_ade20k.py 8 --work_
↪dir work_dirs/pspnet_r50-d8_512x512_80k_ade20k/ --deterministic
```

注意: 在训练时, 模型的和日志保存在 “work_dirs/” 下的配置文件的相同文件夹结构中。不建议使用自定义的 “work_dirs/”, 因为验证脚本可以从配置文件名中推断工作目录。如果你想在其他地方保存模型的权重, 请使用符号链接, 例如:

```
ln -s ${YOUR_WORK_DIRS} ${MMSEG}/work_dirs
```

6.1.4 在单个机器上启动多个任务

如果您在单个机器上启动多个任务, 例如在 8 卡 GPU 的一个机器上有 2 个 4 卡 GPU 的训练任务, 您需要特别对每个任务指定不同的端口 (默认为 29500) 来避免通讯冲突。否则, 将会有报错信息 `RuntimeError: Address already in use`。

如果您使用命令 `dist_train.sh` 来启动一个训练任务, 您可以在命令行的用环境变量 `PORT` 设置端口:

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

6.2 使用多台机器训练

如果您想使用由 `ethernet` 连接起来的多台机器, 您可以使用以下命令:

在第一台机器上:

```
NNODES=2 NODE_RANK=0 PORT=$MASTER_PORT MASTER_ADDR=$MASTER_ADDR sh tools/dist_train.
↪sh $CONFIG $GPUS
```

在第二台机器上:

```
NNODES=2 NODE_RANK=1 PORT=$MASTER_PORT MASTER_ADDR=$MASTER_ADDR sh tools/dist_train.
↪sh $CONFIG $GPUS
```

但是, 如果您不使用高速网路连接这几台机器的话, 训练将会非常慢。

6.3 使用 slurm 管理任务

Slurm 是一个很好的计算集群作业调度系统。在由 Slurm 管理的集群中，可以使用 `slurm_train.sh` 来进行训练。它同时支持单节点和多节点训练。

在多台机器上训练：

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

这里有一个在 dev 分区上使用 16 块 GPUs 来训练 PSPNet 的例子：

```
GPUS=16 sh tools/slurm_train.sh dev pspr50 configs/psenet/psenet_r50-d8_512x1024_40k_  
→cityscapes.py work_dirs/psenet_r50-d8_512x1024_40k_cityscapes/
```

当使用 `slurm_train.sh` 在一个节点上启动多个任务时，需要指定不同的端口号，这里提供了三种设置：

方式 1：

在 `config1.py` 中设置：

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

在 `config2.py` 中设置：

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

然后就可以使用 `config1.py` 和 `config2.py` 启动两个作业：

```
CUDA_VISIBLE_DEVICES=0,1,2,3 GPUS=4 sh tools/slurm_train.sh ${PARTITION} ${JOB_NAME}┐  
→config1.py tmp_work_dir_1  
CUDA_VISIBLE_DEVICES=4,5,6,7 GPUS=4 sh tools/slurm_train.sh ${PARTITION} ${JOB_NAME}┐  
→config2.py tmp_work_dir_2
```

方式 2：

您可以设置不同的通信端口，而不需要修改配置文件，但必须设置 “`cfg-options`”，以覆盖配置文件中的默认端口。

```
CUDA_VISIBLE_DEVICES=0,1,2,3 GPUS=4 sh tools/slurm_train.sh ${PARTITION} ${JOB_NAME}┐  
→config1.py tmp_work_dir_1 --cfg-options dist_params.port=29500  
CUDA_VISIBLE_DEVICES=4,5,6,7 GPUS=4 sh tools/slurm_train.sh ${PARTITION} ${JOB_NAME}┐  
→config2.py tmp_work_dir_2 --cfg-options dist_params.port=29501
```

方式 3：

您可以使用环境变量 ‘`MASTER_PORT`’ 在命令中设置端口：

```
CUDA_VISIBLE_DEVICES=0,1,2,3 GPUS=4 MASTER_PORT=29500 sh tools/slurm_train.sh $
↪{PARTITION} ${JOB_NAME} config1.py tmp_work_dir_1
CUDA_VISIBLE_DEVICES=4,5,6,7 GPUS=4 MASTER_PORT=29501 sh tools/slurm_train.sh $
↪{PARTITION} ${JOB_NAME} config2.py tmp_work_dir_2
```

使用预训练模型推理

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。

7.1 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]
```

(下页继续)

(续上页)

```
# 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片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片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片

```
python tools/test.py configs/psenet/psenet_r50-d8_512x1024_40k_cityscapes.py \
    checkpoints/psenet_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/psenet/psenet_r50-d8_512x1024_40k_cityscapes.py \
    checkpoints/psenet_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/psenet/psenet_r50-d8_512x1024_20k_voc12aug.py \
    checkpoints/psenet_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/psenet/psenet_r50-d8_512x1024_40k_cityscapes.py \
    checkpoints/psenet_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/psenet/psenet_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/psenet/psenet_r50-d8_512x1024_40k_cityscapes.py \
    checkpoints/psenet_r50-d8_512x1024_40k_cityscapes_20200605_003338-2966598c.
↪pth \
    4 --format-only --eval-options "imgfile_prefix=./psenet_test_results"
```

您会在文件夹 ./psenet_test_results 里得到生成的 png 文件。您也许可以运行 `zip -r results.zip psenet_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/psenet/psenet_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](#)。

教程 1: 学习配置文件

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。

8.1 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 [mmdcv](#)。

8.2 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

8.3 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(
```

(下页继续)

(续上页)

```

    type='PSPHead', # 解码头 (decode head) 的类别。 可用选项请参考 mmseg/models/
    ↳ decode_heads。
    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。

```

(下页继续)

(续上页)

```

crop_size = (512, 1024) # 训练时的裁剪大小
train_pipeline = [ # 训练流程
    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。

```

(下页继续)

(续上页)

```

        mean=[123.675, 116.28, 103.53],
        std=[58.395, 57.12, 57.375],
        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',

```

(下页继续)

(续上页)

```

        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'])
        ])
    ],
    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=[ # 训练期间执行的钩子
        dict(type='TextLoggerHook', by_epoch=False),
        dict(type='TensorboardLoggerHook', by_epoch=False),
        dict(type='MMSegWandbHook', by_epoch=False, # 还支持 Wandb 记录器, 它需要安装
↪ `wandb`。

```

(下页继续)

(续上页)

```

        init_kwargs={'entity': "OpenMMLab", # 用于登录 wandb 的实体
                      'project': "mmseg", # WandB 中的项目名称
                      'config': cfg_dict}), # 检查 https://docs.wandb.ai/ref/
        ↪python/init 以获取更多初始化参数
    ])

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。
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

```

(下页继续)

8.4 FAQ

8.4.1 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_cityscapes.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(
```

(续上页)

```

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

```

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

8.4.2 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.]

```

(下页继续)

(续上页)

```

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], # 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_cityscapes.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))

```

教程 2: 自定义数据集

9.1 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。

9.2 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。

9.2.1 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
    )
)
```


9.2.2 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'],
    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,
```

(下页继续)

(续上页)

```
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(  
    ...  
    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,
```

(下页继续)

(续上页)

```

    test = dataset_A_test
)

```

9.2.3 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` 的多图混合数据图广使用。

`MultiImageMixDataset` 与 `Mosaic` 数据图广一起使用的例子：

```

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']),
]

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
)

```


教程 3: 自定义数据流程

10.1 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'),
    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),
```

(下页继续)

(续上页)

```

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):

10.1.1 数据加载 Data loading

LoadImageFromFile

- 增加: img, img_shape, ori_shape

LoadAnnotations

- 增加: gt_semantic_seg, seg_fields

10.1.2 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`

10.1.3 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`

10.1.4 测试时数据增强 Test time augmentation

MultiScaleFlipAug

10.2 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

```
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

```
from .my_pipeline import MyTransform
```

3. 在配置文件里使用它

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


11.1 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。

11.2 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, optimizer_cfg, paramwise_cfg=None):

    def __call__(self, model):

        return my_optimizer
```

11.3 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）

11.3.1 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 ..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
```

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),
    ...)
```

11.3.2 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)))
```

11.3.3 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)
```

MMSegmentation 支持如下训练技巧:

12.1 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。

12.2 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。

12.3 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。

12.4 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。

12.5 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。

13.1 自定义优化设定

13.1.1 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。

13.1.2 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)
```

13.1.3 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
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。

13.1.4 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己已经设置了 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',
    target_ratio=(0.85 / 0.95, 1),
    cyclic_times=1,
    step_ratio_up=0.4,
)
```

13.2 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)
```

13.3 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 workflow：

```
[('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

13.4 自定义钩 (hooks)

13.4.1 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')
]
```

13.4.2 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')
```


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。

14.1 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。

14.2 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。

14.3 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

14.4 评估 ONNX 模型

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

14.4.1 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)

14.4.2 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

14.4.3 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。

14.5 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/psenet/psenet_r50-d8_512x1024_40k_
↪cityscapes.py \
--checkpoint checkpoints/psenet_r50-d8_512x1024_40k_cityscapes_20200605_003338-
↪2966598c.pth \
--output-file checkpoints/psenet_r50-d8_512x1024_40k_cityscapes_20200605_003338-
↪2966598c.pt \
--shape 512 1024
```

14.6 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

15.1 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

15.2 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
```

15.3 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。

15.3.1 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}
```


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

16.1 1. 将 model 从 `MMSegmentation` 转换到 `TorchServe`

```
python tools/mmdet2torchserve.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

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

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

16.3 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](#)。

16.4 4. 测试部署

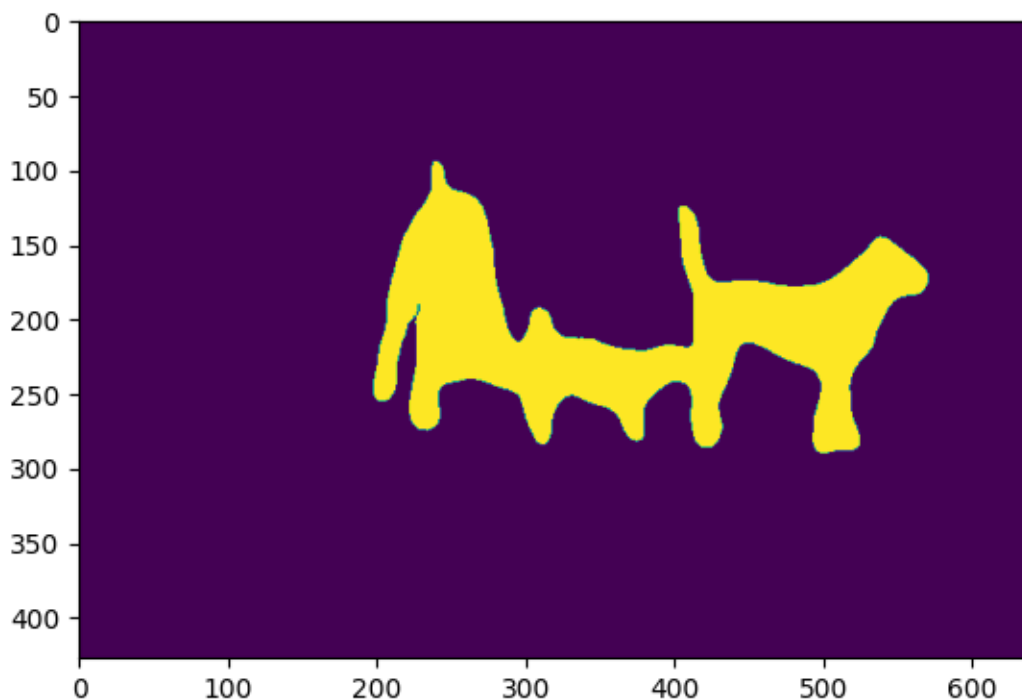
```
curl -O https://raw.githubusercontent.com/open-mmlab/mmdetection/master/resources/
↪3dogs.jpg
curl http://127.0.0.1:8080/predictions/${MODEL_NAME} -T 3dogs.jpg -o 3dogs_mask.png
```

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()
```

看到的東西將會和下图类似:



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
```


我们提供了 `tools/model_ensemble.py` 完成对多个模型的预测概率进行集成的脚本

17.1 使用方法

```
python tools/model_ensemble.py \  
  --config ${CONFIG_FILE1} ${CONFIG_FILE2} ... \  
  --checkpoint ${CHECKPOINT_FILE1} ${CHECKPOINT_FILE2} ...\  
  --aug-test \  
  --out ${OUTPUT_DIR}\  
  --gpus ${GPU_USED}
```

17.2 各个参数的描述:

- `--config`: 集成模型的配置文件的路径
- `--checkpoint`: 集成模型的权重文件的路径
- `--aug-test`: 是否使用翻转和多尺度预测
- `--out`: 模型集成结果的保存文件夹路径
- `--gpus`: 模型集成使用的 `gpu-id`

17.3 模型集成结果

- 模型集成会对每一张输入，形状为 $[H, W]$ ，产生一张未渲染的分割掩膜文件 (segmentation mask)，形状为 $[H, W]$ ，分割掩膜中的每个像素点的值代表该位置分割后的像素类别。
- 模型集成结果的文件名会采用和 Ground Truth 一致的文件命名，如 Ground Truth 文件名称为 1.png，则模型集成结果文件也会被命名为 1.png，并放置在 --out 指定的文件夹中。

常见问题解答 (FAQ)

我们在这里列出了使用时的一些常见问题及其相应的解决方案。如果您发现有一些问题被遗漏，请随时提 PR 丰富这个列表。如果您无法在此获得帮助，请使用 [issue 模板](#) 创建问题，但是请在模板中填写所有必填信息，这有助于我们更快定位问题。

18.1 安装

兼容的 MMSegmentation 和 MMCV 版本如下。请安装正确版本的 MMCV 以避免安装问题。

如果你安装了 mmcv，你需要先运行 `pip uninstall mmcv`。如果 mmcv 和 mmcv-full 都安装了，会出现 “ModuleNotFoundError”。

- “No module named ‘mmcv.ops’” ; “No module named ‘mmcv._ext’” .
 1. 使用 `pip uninstall mmcv` 卸载环境中现有的 mmcv。
 2. 按照安装说明安装 mmcv-full。

18.2 如何获知模型训练时需要的显卡数量

- 看模型的 config 文件的命名。可以参考学习配置文件中的配置文件命名风格部分。比如，对于名字为 segformer_mit-b0_8x1_1024x1024_160k_cityscapes.py 的 config 文件，8x1 代表训练其对应的模型需要的卡数为 8，每张卡中的 batch size 为 1。
- 看模型的 log 文件。点开该模型的 log 文件，并在其中搜索 nGPU，在 nGPU 后的数字个数即训练时所需的卡数。比如，在 log 文件中搜索 nGPU 得到 nGPU 0,1,2,3,4,5,6,7 的记录，则说明训练该模型需要使用八张卡。

18.3 auxiliary head 是什么

简单来说，这是一个提高准确率的深度监督技术。在训练阶段，decode_head 用于输出语义分割的结果，auxiliary_head 只是增加了一个辅助损失，其产生的分割结果对你的模型结果没有影响，仅在在训练中起作用。你可以阅读这篇[论文](#)了解更多信息。

18.4 为什么日志文件没有被创建

在训练脚本中，我们在第 167 行调用 get_root_logger 方法，然后 mmseg 的 get_root_logger 方法调用 mmcv 的 get_logger，mmcv 将返回在 ‘mmsegmentation/tools/train.py’ 中使用参数 log_file 初始化的同一个 logger。在训练期间只存在一个用 log_file 初始化的 logger。

参 考：<https://github.com/open-mmlab/mmcv/blob/21bada32560c7ed7b15b017dc763d862789e29a8/mmcv/utils/logging.py#L9-L16>

如果你发现日志文件没有被创建，可以检查 mmcv.utils.get_logger 是否在其他地方被调用。

18.5 运行测试脚本时如何输出绘制分割掩膜的图像

在测试脚本中，我们提供了 show-dir 参数来控制是否输出绘制的图像。用户可以运行以下命令：

```
python tools/test.py {config} {checkpoint} --show-dir {/path/to/save/image} --opacity-  
→ 1
```


18.6 如何处理二值分割任务？

MMSegmentation 使用 `num_classes` 和 `out_channels` 来控制模型最后一层 `self.conv_seg` 的输出。更多细节可以参考 [这里](#)。

`num_classes` 应该和数据集本身类别个数一致，当是二值分割时，数据集只有前景和背景两类，所以 `num_classes` 为 2。 `out_channels` 控制模型最后一层的输出的通道数，通常和 `num_classes` 相等，但当二值分割时候，可以有两种处理方法，分别是：

- 设置 `out_channels=2`，在训练时以 **Cross Entropy Loss** 作为损失函数，在推理时使用 `F.softmax()` 归一化 logits 值，然后通过 `argmax()` 得到每个像素的预测结果。
- 设置 `out_channels=1`，在训练时以 **Binary Cross Entropy Loss** 作为损失函数，在推理时使用 `F.sigmoid()` 和 `threshold` 得到预测结果，`threshold` 默认为 0.3。

对于实现上述两种计算二值分割的方法，需要在 `decode_head` 和 `auxiliary_head` 的配置里修改。下面是对样例 `pspnet_unet_s5-d16.py` 做出的对应修改。

- (1) `num_classes=2`, `out_channels=2` 并在 `CrossEntropyLoss` 里面设置 `use_sigmoid=False`。

```
decode_head=dict(
    type='PSPHead',
    in_channels=64,
    in_index=4,
    num_classes=2,
    out_channels=2,
    loss_decode=dict(
        type='CrossEntropyLoss', use_sigmoid=False, loss_weight=1.0)),
auxiliary_head=dict(
    type='FCNHead',
    in_channels=128,
    in_index=3,
    num_classes=2,
    out_channels=2,
    loss_decode=dict(
        type='CrossEntropyLoss', use_sigmoid=False, loss_weight=0.4)),
```

- (2) `num_classes=2`, `out_channels=1` 并在 `CrossEntropyLoss` 里面设置 `use_sigmoid=True`。

```
decode_head=dict(
    type='PSPHead',
    in_channels=64,
    in_index=4,
    num_classes=2,
    out_channels=1,
```

(下页继续)

(续上页)

```
loss_decode=dict(  
    type='CrossEntropyLoss', use_sigmoid=True, loss_weight=1.0)),  
auxiliary_head=dict(  
    type='FCNHead',  
    in_channels=128,  
    in_index=3,  
    num_classes=2,  
    out_channels=1,  
    loss_decode=dict(  
        type='CrossEntropyLoss', use_sigmoid=True, loss_weight=0.4)),
```

18.7 reduce_zero_label 的作用

数据集中 `reduce_zero_label` 参数类型为布尔类型, 默认为 `False`, 它的功能是为了忽略数据集 label 0. 具体做法是将 label 0 改为 255, 其余 label 相应编号减 1, 同时 decode head 里将 255 设为 ignore index, 即不参与 loss 计算. 以下是 `reduce_zero_label` 具体实现逻辑:

```
if self.reduce_zero_label:  
    # avoid using underflow conversion  
    gt_semantic_seg[gt_semantic_seg == 0] = 255  
    gt_semantic_seg = gt_semantic_seg - 1  
    gt_semantic_seg[gt_semantic_seg == 254] = 255
```

注意: 使用 `reduce_zero_label` 请确认数据集原始类别个数, 如果只有两类, 需要关闭 `reduce_zero_label` 即设置 `reduce_zero_label=False`.

19.1 使用方法

首先，请参考[MMCV](#) 安装带有 NPU 支持的 [MMCV](#) 与 [MMEEngine](#)。使用如下命令，可以利用 4 个 NPU 训练模型（以 deeplabv3 为例）：

```
bash tools/dist_train.sh configs/deeplabv3/deeplabv3_r50-d8_512x1024_40k_cityscapes.  
→py 4
```

或者，使用如下命令，在一个 NPU 上训练模型（以 deeplabv3 为例）：

```
python tools/train.py configs/deeplabv3/deeplabv3_r50-d8_512x1024_40k_cityscapes.py
```

19.2 经过验证的模型

注意：

- 如果没有特别标记，NPU 上的结果与使用 FP32 的 GPU 上的结果结果相同。

以上所有模型权重及训练日志均由华为昇腾团队提供

CHAPTER 20

English

CHAPTER 21

简体中文

CHAPTER 22

mmseg.apis

CHAPTER 23

mmseg.core

23.1 seg

23.2 evaluation

23.3 utils

CHAPTER 24

mmseg.datasets

24.1 datasets

24.2 pipelines

25.1 segmentors

25.2 backbones

25.3 decode_heads

25.4 losses

CHAPTER 26

Indices and tables

- `genindex`
- `search`