mmengine

发布 *0.2.0*

mmengine contributors

2023 年 02 月 07 日

开始你的第一步

您可以在页面左下角切换中英文文档。

CHAPTER 1

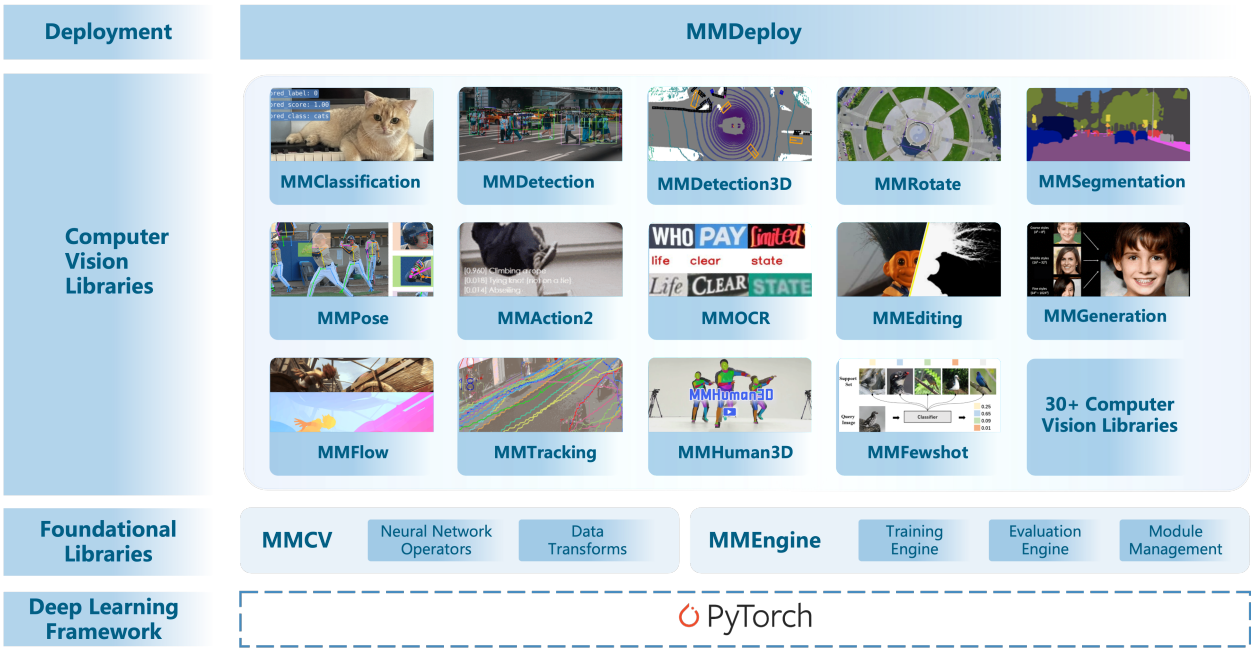
介绍

MMEngine 是一个用于深度学习模型训练的基础库，基于 PyTorch，支持在 Linux、Windows、macOS 上运行。它具有如下三个亮点：

1. 通用：MMEngine 实现了一个高级的通用训练器，它能够：
   - 支持用少量代码训练不同的任务，例如仅使用 80 行代码就可以训练 imagenet（pytorch example 400 行）
   - 轻松兼容流行的算法库如 TIMM、TorchVision 和 Detectron2 中的模型
2. 统一：MMEngine 设计了一个接口统一的开放架构，使得
   - 用户可以仅依赖一份代码实现所有任务的轻量化，例如 MMRazor 1.x 相比 MMRazor 0.x 优化了 40% 的代码量
   - 上下游的对接更加统一便捷，在为上层算法库提供统一抽象的同时，支持多种后端设备。目前 MMEngine 支持 Nvidia CUDA、Mac MPS、AMD、MLU 等设备进行模型训练。
3. 灵活：MMEngine 实现了“乐高”式的训练流程，支持了
   - 根据迭代数、loss 和评测结果等动态调整的训练流程、优化策略和数据增强策略，例如早停（early stopping）机制等
   - 任意形式的模型权重平均，如 Exponential Momentum Average (EMA) 和 Stochastic Weight Averaging (SWA)
   - 训练过程中针对任意数据和任意节点的灵活可视化和日志控制
   - 对神经网络模型中各个层的优化配置进行细粒度调整

- 混合精度训练的灵活控制

1.1 架构



上图展示了 MMEngine 在 OpenMMLab 2.0 中的层次。MMEngine 实现了 OpenMMLab 算法库的新一代训练架构，为 OpenMMLab 中的 30 多个算法库提供了统一的执行基座。其核心组件包含训练引擎、评测引擎和模块管理等。

1.2 模块介绍

MMEngine 将训练过程中涉及的组件和它们的关系进行了抽象，如上图所示。不同算法库中的同类型组件具有相同的接口定义。

1.2.1 核心模块与相关组件

训练引擎的核心模块是执行器（*Runner*）。执行器负责执行训练、测试和推理任务并管理这些过程中所需要的各个组件。在训练、测试、推理任务执行过程中的特定位置，执行器设置了钩子（*Hook*）来允许用户拓展、插入和执行自定义逻辑。执行器主要调用如下组件来完成训练和推理过程中的循环：

- 数据集（Dataset）：负责在训练、测试、推理任务中构建数据集，并将数据送给模型。实际使用过程中会被数据加载器（DataLoader）封装一层，数据加载器会启动多个子进程来加载数据。
- 模型（*Model*）：在训练过程中接受数据并输出 loss；在测试、推理任务中接受数据，并进行预测。分布式训练等情况下会被模型的封装器（Model Wrapper，如 `MMDistributedDataParallel`）封装一

层。

- 优化器封装（*Optimizer*）：优化器封装负责在训练过程中执行反向传播优化模型，并且以统一的接口支持了混合精度训练和梯度累加。
- 参数调度器（*Parameter Scheduler*）：训练过程中，对学习率、动量等优化器超参数进行动态调整。

在训练间隙或者测试阶段，评测指标与评测器（Metrics & Evaluator）会负责对模型性能进行评测。其中评测器负责基于数据集对模型的预测进行评估。评测器内还有一层抽象是评测指标，负责计算具体的一个或多个评测指标（如召回率、正确率等）。

为了统一接口，OpenMMLab 2.0 中各个算法库的评测器，模型和数据之间交流的接口都使用了数据元素（Data Element）来进行封装。

在训练、推理执行过程中，上述各个组件都可以调用日志管理模块和可视化器进行结构化和非结构化日志的存储与展示。日志管理（Logging Modules）：负责管理执行器运行过程中产生的各种日志信息。其中消息枢纽（MessageHub）负责实现组件与组件、执行器与执行器之间的数据共享，日志处理器（Log Processor）负责对日志信息进行处理，处理后的日志会分别发送给执行器的日志器（Logger）和可视化器（Visualizer）进行日志的管理与展示。可视化器（Visualizer）：可视化器负责对模型的特征图、预测结果和训练过程中产生的结构化日志进行可视化，支持 Tensorboard 和 WanDB 等多种可视化后端。

1.2.2 公共基础模块

MMEngine 中还实现了各种算法模型执行过程中需要用到的公共基础模块，包括

- 配置类（*Config*）：在 OpenMMLab 算法库中，用户可以通过编写 config 来配置训练、测试过程以及相关的组件。
- 注册器（*Registry*）：负责管理算法库中具有相同功能的模块。MMEngine 根据对算法库模块的抽象，定义了一套根注册器，算法库中的注册器可以继承自这套根注册器，实现模块的跨算法库调用。
- 文件读写（File I/O）：为各个模块的文件读写提供了统一的接口，以统一的形式支持了多种文件读写后端和多种文件格式，并具备扩展性。
- 分布式通信原语（Distributed Communication Primitives）：负责在程序分布式运行过程中不同进程间的通信。这套接口屏蔽了分布式和非分布式环境的区别，同时也自动处理了数据的设备和通信后端。
- 其他工具（Utils）：还有一些工具性的模块，如 ManagerMixin，它实现了一种全局变量的创建和获取方式，执行器内很多全局可见对象的基类就是 ManagerMixin。

用户可以进一步阅读教程来了解这些模块的高级用法，也可以参考设计文档了解它们的设计思路与细节。

CHAPTER 2

安装

2.1 环境依赖

- Python 3.6+
- PyTorch 1.6+
- CUDA 9.2+
- GCC 5.4+

2.2 准备环境

1. 使用 conda 新建虚拟环境，并进入该虚拟环境；

```
conda create -n open-mmlab python=3.7 -y
conda activate open-mmlab
```

2. 安装 PyTorch

   在安装 MMEngine 之前，请确保 PyTorch 已经成功安装在环境中，可以参考 PyTorch 官方安装文档。使用以下命令验证 PyTorch 是否安装

```
python -c 'import torch;print(torch.__version__)'
```

2.3 安装 MMEngine

2.3.1 使用 mim 安装

mim 是 OpenMMLab 项目的包管理工具，使用它可以很方便地安装 OpenMMLab 项目。

```
pip install -U openmim
mim install mmengine
```

2.3.2 使用 pip 安装

```
pip install mmengine
```

2.3.3 使用 docker 镜像

1. 构建镜像

   ```
   docker build -t mmengine https://github.com/open-mmlab/mmengine.git#main:docker/
   ↪release
   ```

   更多构建方式请参考 mmengine/docker。

2. 运行镜像

   ```
   docker run --gpus all --shm-size=8g -it mmengine
   ```

2.3.4 源码安装

```
# 如果克隆代码仓库的速度过慢，可以从 https://gitee.com/open-mmlab/mmengine.git 克隆
git clone https://github.com/open-mmlab/mmengine.git
cd mmengine
pip install -e . -v
```

2.4 验证安装

为了验证是否正确安装了 MMEngine 和所需的环境，我们可以运行以下命令

```
python -c 'import mmengine;print(mmengine.__version__)'
```

CHAPTER 3

15 分钟上手 MMEngine

以在 CIFAR-10 数据集上训练一个 ResNet-50 模型为例，我们将使用 80 行以内的代码，利用 MMEngine 构建一个完整的、可配置的训练和验证流程，整个流程包含如下步骤：

1. 构建模型
2. 构建数据集和数据加载器
3. 构建评测指标
4. 构建执行器并执行任务

3.1 构建模型

首先，我们需要构建一个**模型**，在 MMEngine 中，我们约定这个模型应当继承 `BaseModel`，并且其 `forward` 方法除了接受来自数据集的若干参数外，还需要接受额外的参数 `mode`：对于训练，我们需要 `mode` 接受字符串 “loss”，并返回一个包含 “loss” 字段的字典；对于验证，我们需要 `mode` 接受字符串 “predict”，并返回同时包含预测信息和真实信息的结果。

```
import torch.nn.functional as F
import torchvision
from mmengine.model import BaseModel


class MMResNet50(BaseModel):
    def __init__(self):
```

(下页继续)

(续上页)

```
        super().__init__()
        self.resnet = torchvision.models.resnet50()

    def forward(self, imgs, labels, mode):
        x = self.resnet(imgs)
        if mode == 'loss':
            return {'loss': F.cross_entropy(x, labels)}
        elif mode == 'predict':
            return x, labels
```

3.2 构建数据集和数据加载器

其次，我们需要构建训练和验证所需要的**数据集 (Dataset) 和数据加载器 (DataLoader)**。对于基础的训练和验证功能，我们可以直接使用符合 PyTorch 标准的数据加载器和数据集。

```
import torchvision.transforms as transforms
from torch.utils.data import DataLoader

norm_cfg = dict(mean=[0.491, 0.482, 0.447], std=[0.202, 0.199, 0.201])
train_dataloader = DataLoader(batch_size=32,
                              shuffle=True,
                              dataset=torchvision.datasets.CIFAR10(
                                  'data/cifar10',
                                  train=True,
                                  download=True,
                                  transform=transforms.Compose([
                                      transforms.RandomCrop(32, padding=4),
                                      transforms.RandomHorizontalFlip(),
                                      transforms.ToTensor(),
                                      transforms.Normalize(**norm_cfg)
                                  ])))

val_dataloader = DataLoader(batch_size=32,
                            shuffle=False,
                            dataset=torchvision.datasets.CIFAR10(
                                'data/cifar10',
                                train=False,
                                download=True,
                                transform=transforms.Compose([
                                    transforms.ToTensor(),
                                    transforms.Normalize(**norm_cfg)
                                ])))
```

3.3 构建评测指标

为了进行验证和测试，我们需要定义模型推理结果的**评测指标**。我们约定这一评测指标需要继承 BaseMetric，并实现 process 和 compute_metrics 方法。其中 process 方法接受数据集的输出和模型 mode="predict" 时的输出，此时的数据为一个批次的数据，对这一批次的数据进行处理后，保存信息至 self.results 属性。而 compute_metrics 接受 results 参数，这一参数的输入为 process 中保存的所有信息（如果是分布式环境，results 中为已收集的，包括各个进程 process 保存信息的结果），利用这些信息计算并返回保存有评测指标结果的字典。

```
from mmengine.evaluator import BaseMetric

class Accuracy(BaseMetric):
    def process(self, data_batch, data_samples):
        score, gt = data_samples
        # 将一个批次的中间结果保存至 `self.results`
        self.results.append({
            'batch_size': len(gt),
            'correct': (score.argmax(dim=1) == gt).sum().cpu(),
        })

    def compute_metrics(self, results):
        total_correct = sum(item['correct'] for item in results)
        total_size = sum(item['batch_size'] for item in results)
        # 返回保存有评测指标结果的字典，其中键为指标名称
        return dict(accuracy=100 * total_correct / total_size)
```

3.4 构建执行器并执行任务

最后，我们利用构建好的**模型**，**数据加载器**，**评测指标**构建一个**执行器 (Runner)**，同时在其中配置 **优化器**、**工作路径**、**训练与验证配置**等选项，即可通过调用 train() 接口启动训练：

```
from torch.optim import SGD
from mmengine.runner import Runner

runner = Runner(
    # 用以训练和验证的模型，需要满足特定的接口需求
    model=MMResNet50(),
    # 工作路径，用以保存训练日志、权重文件信息
    work_dir='./work_dir',
    # 训练数据加载器，需要满足 PyTorch 数据加载器协议
    train_dataloader=train_dataloader,
    # 优化器包装，用于模型优化，并提供 AMP、梯度累积等附加功能
```

(下页继续)

(续上页)

```
    optim_wrapper=dict(optimizer=dict(type=SGD, lr=0.001, momentum=0.9)),
    # 训练配置，用于指定训练周期、验证间隔等信息
    train_cfg=dict(by_epoch=True, max_epochs=5, val_interval=1),
    # 验证数据加载器，需要满足 PyTorch 数据加载器协议
    val_dataloader=val_dataloader,
    # 验证配置，用于指定验证所需要的额外参数
    val_cfg=dict(),
    # 用于验证的评测器，这里使用默认评测器，并评测指标
    val_evaluator=dict(type=Accuracy),
)

runner.train()
```

最后，让我们把以上部分汇总成为一个完整的，利用 MMEngine 执行器进行训练和验证的脚本：

```
import torch.nn.functional as F
import torchvision
import torchvision.transforms as transforms
from torch.optim import SGD
from torch.utils.data import DataLoader

from mmengine.evaluator import BaseMetric
from mmengine.model import BaseModel
from mmengine.runner import Runner


class MMResNet50(BaseModel):
    def __init__(self):
        super().__init__()
        self.resnet = torchvision.models.resnet50()

    def forward(self, imgs, labels, mode):
        x = self.resnet(imgs)
        if mode == 'loss':
            return {'loss': F.cross_entropy(x, labels)}
        elif mode == 'predict':
            return x, labels


class Accuracy(BaseMetric):
    def process(self, data_batch, data_samples):
        score, gt = data_samples
```

(下页继续)

(续上页)

```
        self.results.append({
            'batch_size': len(gt),
            'correct': (score.argmax(dim=1) == gt).sum().cpu(),
        })

    def compute_metrics(self, results):
        total_correct = sum(item['correct'] for item in results)
        total_size = sum(item['batch_size'] for item in results)
        return dict(accuracy=100 * total_correct / total_size)


norm_cfg = dict(mean=[0.491, 0.482, 0.447], std=[0.202, 0.199, 0.201])
train_dataloader = DataLoader(batch_size=32,
                              shuffle=True,
                              dataset=torchvision.datasets.CIFAR10(
                                  'data/cifar10',
                                  train=True,
                                  download=True,
                                  transform=transforms.Compose([
                                      transforms.RandomCrop(32, padding=4),
                                      transforms.RandomHorizontalFlip(),
                                      transforms.ToTensor(),
                                      transforms.Normalize(**norm_cfg)
                                  ])))

val_dataloader = DataLoader(batch_size=32,
                            shuffle=False,
                            dataset=torchvision.datasets.CIFAR10(
                                'data/cifar10',
                                train=False,
                                download=True,
                                transform=transforms.Compose([
                                    transforms.ToTensor(),
                                    transforms.Normalize(**norm_cfg)
                                ])))

runner = Runner(
    model=MMResNet50(),
    work_dir='./work_dir',
    train_dataloader=train_dataloader,
    optim_wrapper=dict(optimizer=dict(type=SGD, lr=0.001, momentum=0.9)),
    train_cfg=dict(by_epoch=True, max_epochs=5, val_interval=1),
    val_dataloader=val_dataloader,
```

(下页继续)

(续上页)

```
    val_cfg=dict(),
    val_evaluator=dict(type=Accuracy),
)
runner.train()
```

输出的训练日志如下：

```
2022/08/22 15:51:53 - mmengine - INFO -
------------------------------------------------------------
System environment:
    sys.platform: linux
    Python: 3.8.12 (default, Oct 12 2021, 13:49:34) [GCC 7.5.0]
    CUDA available: True
    numpy_random_seed: 1513128759
    GPU 0: NVIDIA GeForce GTX 1660 SUPER
    CUDA_HOME: /usr/local/cuda
...

2022/08/22 15:51:54 - mmengine - INFO - Checkpoints will be saved to /home/mazerun/
↪work_dir by HardDiskBackend.
2022/08/22 15:51:56 - mmengine - INFO - Epoch(train) [1][10/1563]  lr: 1.0000e-03 ␣
↪eta: 0:18:23  time: 0.1414  data_time: 0.0077  memory: 392  loss: 5.3465
2022/08/22 15:51:56 - mmengine - INFO - Epoch(train) [1][20/1563]  lr: 1.0000e-03 ␣
↪eta: 0:11:29  time: 0.0354  data_time: 0.0077  memory: 392  loss: 2.7734
2022/08/22 15:51:56 - mmengine - INFO - Epoch(train) [1][30/1563]  lr: 1.0000e-03 ␣
↪eta: 0:09:10  time: 0.0352  data_time: 0.0076  memory: 392  loss: 2.7789
2022/08/22 15:51:57 - mmengine - INFO - Epoch(train) [1][40/1563]  lr: 1.0000e-03 ␣
↪eta: 0:08:00  time: 0.0353  data_time: 0.0073  memory: 392  loss: 2.5725
2022/08/22 15:51:57 - mmengine - INFO - Epoch(train) [1][50/1563]  lr: 1.0000e-03 ␣
↪eta: 0:07:17  time: 0.0347  data_time: 0.0073  memory: 392  loss: 2.7382
2022/08/22 15:51:57 - mmengine - INFO - Epoch(train) [1][60/1563]  lr: 1.0000e-03 ␣
↪eta: 0:06:49  time: 0.0347  data_time: 0.0072  memory: 392  loss: 2.5956
2022/08/22 15:51:58 - mmengine - INFO - Epoch(train) [1][70/1563]  lr: 1.0000e-03 ␣
↪eta: 0:06:28  time: 0.0348  data_time: 0.0072  memory: 392  loss: 2.7351
...
2022/08/22 15:52:50 - mmengine - INFO - Saving checkpoint at 1 epochs
2022/08/22 15:52:51 - mmengine - INFO - Epoch(val) [1][10/313]    eta: 0:00:03  time:␣
↪0.0122  data_time: 0.0047  memory: 392
2022/08/22 15:52:51 - mmengine - INFO - Epoch(val) [1][20/313]    eta: 0:00:03  time:␣
↪0.0122  data_time: 0.0047  memory: 308
2022/08/22 15:52:51 - mmengine - INFO - Epoch(val) [1][30/313]    eta: 0:00:03  time:␣
↪0.0123  data_time: 0.0047  memory: 308
...
2022/08/22 15:52:54 - mmengine - INFO - Epoch(val) [1][313/313]  accuracy: 35.7000
```

除了以上基础组件，你还可以利用**执行器**轻松地组合配置各种训练技巧，如开启混合精度训练和梯度累积（见优化器封装（*OptimWrapper*））、配置学习率衰减曲线（见评测指标与评测器（*Metrics & Evaluator*））等。

CHAPTER 4

注册器（Registry）

OpenMMLab 的算法库支持了丰富的算法和数据集，因此实现了很多功能相近的模块。例如 ResNet 和 SE-ResNet 的算法实现分别基于 `ResNet` 和 `SEResNet` 类，这些类有相似的功能和接口，都属于算法库中的模型组件。为了管理这些功能相似的模块，MMEngine 实现了注册器。OpenMMLab 大多数算法库均使用注册器来管理它们的代码模块，包括 MMDetection，MMDetection3D，MMClassification 和 MMEditing 等。

4.1 什么是注册器

MMEngine 实现的注册器可以看作一个映射表和模块构建方法（build function）的组合。映射表维护了一个字符串到**类或者函数的映射**，使得用户可以借助字符串查找到相应的类或函数，例如维护字符串 `"ResNet"` 到 `ResNet` 类或函数的映射，使得用户可以通过 `"ResNet"` 找到 `ResNet` 类；而模块构建方法则定义了如何根据字符串查找到对应的类或函数以及如何实例化这个类或者调用这个函数，例如，通过字符串 `"bn"` 找到 `nn.BatchNorm2d` 并实例化 `BatchNorm2d` 模块；又或者通过字符串 `"build_batchnorm2d"` 找到 `build_batchnorm2d` 函数并返回该函数的调用结果。MMEngine 中的注册器默认使用*build_from_cfg* 函数来查找并实例化字符串对应的类或者函数。

一个注册器管理的类或函数通常有相似的接口和功能，因此该注册器可以被视作这些类或函数的抽象。例如注册器 `MODELS` 可以被视作所有模型的抽象，管理了 `ResNet`，`SEResNet` 和 `RegNetX` 等分类网络的类以及 `build_ResNet`,`build_SEResNet` 和 `build_RegNetX` 等分类网络的构建函数。

4.2 入门用法

使用注册器管理代码库中的模块，需要以下三个步骤。

1. 创建注册器
2. 创建一个用于实例化类的构建方法（可选，在大多数情况下可以只使用默认方法）
3. 将模块加入注册器中

假设我们要实现一系列激活模块并且希望仅修改配置就能够使用不同的激活模块而无需修改代码。

首先创建注册器，

```
from mmengine import Registry
# scope 表示注册器的作用域，如果不设置，默认为包名，例如在 mmdetection 中，它的 scope 为 mmdet
ACTIVATION = Registry('activation', scope='mmengine')
```

然后我们可以实现不同的激活模块，例如 `Sigmoid`，`ReLU` 和 `Softmax`。

```
import torch.nn as nn

# 使用注册器管理模块
@ACTIVATION.register_module()
class Sigmoid(nn.Module):
    def __init__(self):
        super().__init__()

    def forward(self, x):
        print('call Sigmoid.forward')
        return x

@ACTIVATION.register_module()
class ReLU(nn.Module):
    def __init__(self, inplace=False):
        super().__init__()

    def forward(self, x):
        print('call ReLU.forward')
        return x

@ACTIVATION.register_module()
class Softmax(nn.Module):
    def __init__(self):
        super().__init__()

    def forward(self, x):
```

(下页继续)

(续上页)

```
        print('call Softmax.forward')
        return x
```

使用注册器管理模块的关键步骤是，将实现的模块注册到注册表 ACTIVATION 中。通过 @ACTIVATION.register_module() 装饰所实现的模块，字符串和类或函数之间的映射就可以由 ACTIVATION 构建和维护，我们也可以通过 ACTIVATION.register_module(module=ReLU) 实现同样的功能。

通过注册，我们就可以通过 ACTIVATION 建立字符串与类或函数之间的映射，

```
print(ACTIVATION.module_dict)
# {
#     'Sigmoid': __main__.Sigmoid,
#     'ReLU': __main__.ReLU,
#     'Softmax': __main__.Softmax
# }
```

注解： 只有模块所在的文件被导入时，注册机制才会被触发，所以我们需要在某处导入该文件或者使用 custom_imports 字段动态导入该模块进而触发注册机制，详情见导入自定义 *Python* 模块。

模块成功注册后，我们可以通过配置文件使用这个激活模块。

```
import torch
input = torch.randn(2)

act_cfg = dict(type='Sigmoid')
activation = ACTIVATION.build(act_cfg)
output = activation(input)
# call Sigmoid.forward
print(output)
```

如果我们想使用 ReLU，仅需修改配置。

```
act_cfg = dict(type='ReLU', inplace=True)
activation = ACTIVATION.build(act_cfg)
output = activation(input)
# call Sigmoid.forward
print(output)
```

如果我们希望在创建实例前检查输入参数的类型（或者任何其他操作），我们可以实现一个构建方法并将其传递给注册器从而实现自定义构建流程。

创建一个构建方法，

```
def build_activation(cfg, registry, *args, **kwargs):
    cfg_ = cfg.copy()
    act_type = cfg_.pop('type')
    print(f'build activation: {act_type}')
    act_cls = registry.get(act_type)
    act = act_cls(*args, **kwargs, **cfg_)
    return act
```

并将 build_activation 传递给 build_func 参数

```
ACTIVATION = Registry('activation', build_func=build_activation, scope='mmengine')

@ACTIVATION.register_module()
class Tanh(nn.Module):
    def __init__(self):
        super().__init__()

    def forward(self, x):
        print('call Tanh.forward')
        return x

act_cfg = dict(type='Tanh')
activation = ACTIVATION.build(act_cfg)
output = activation(input)
# build activation: Tanh
# call Tanh.forward
print(output)
```

注解： 在这个例子中，我们演示了如何使用参数 build_func 自定义构建类的实例的方法。该功能类似于默认的 build_from_cfg 方法。在大多数情况下，使用默认的方法就可以了。

MMEngine 的注册器除了可以注册类，也可以注册函数。

```
FUNCTION = Registry('function', scope='mmengine')

@FUNCTION.register_module()
def print_args(**kwargs):
    print(kwargs)

func_cfg = dict(type='print_args', a=1, b=2)
func_res = FUNCTION.build(func_cfg)
```

4.3 进阶用法

MMEngine 的注册器支持层级注册，利用该功能可实现跨项目调用，即可以在一个项目中使用另一个项目的模块。虽然跨项目调用也有其他方法的可以实现，但 MMEngine 注册器提供了更为简便的方法。

为了方便跨库调用，MMEngine 提供了 20 个根注册器：

- RUNNERS: Runner 的注册器
- RUNNER_CONSTRUCTORS: Runner 的构造器
- LOOPS: 管理训练、验证以及测试流程，如 `EpochBasedTrainLoop`
- HOOKS: 钩子，如 `CheckpointHook`, `ParamSchedulerHook`
- DATASETS: 数据集
- DATA_SAMPLERS: `DataLoader` 的 `Sampler`，用于采样数据
- TRANSFORMS: 各种数据预处理，如 `Resize`, `Reshape`
- MODELS: 模型的各种模块
- MODEL_WRAPPERS: 模型的包装器，如 `MMDistributedDataParallel`，用于对分布式数据并行
- WEIGHT_INITIALIZERS: 权重初始化的工具
- OPTIMIZERS: 注册了 PyTorch 中所有的 `Optimizer` 以及自定义的 `Optimizer`
- OPTIM_WRAPPER: 对 Optimizer 相关操作的封装，如 `OptimWrapper`，`AmpOptimWrapper`
- OPTIM_WRAPPER_CONSTRUCTORS: optimizer wrapper 的构造器
- PARAM_SCHEDULERS: 各种参数调度器，如 `MultiStepLR`
- METRICS: 用于计算模型精度的评估指标，如 `Accuracy`
- EVALUATOR: 用于计算模型精度的一个或多个评估指标
- TASK_UTILS: 任务强相关的一些组件，如 `AnchorGenerator`, `BboxCoder`
- VISUALIZERS: 管理绘制模块，如 `DetVisualizer` 可在图片上绘制预测框
- VISBACKENDS: 存储训练日志的后端，如 `LocalVisBackend`, `TensorboardVisBackend`
- LOG_PROCESSORS: 控制日志的统计窗口和统计方法，默认使用 `LogProcessor`，如有特殊需求可自定义 `LogProcessor`

4.3.1 调用父节点的模块

MMEngine 中定义模块 RReLU，并往 MODELS 根注册器注册。

```
import torch.nn as nn
from mmengine import Registry, MODELS

@MODELS.register_module()
class RReLU(nn.Module):
    def __init__(self, lower=0.125, upper=0.333, inplace=False):
        super().__init__()

    def forward(self, x):
        print('call RReLU.forward')
        return x
```

假设有个项目叫 MMAlpha，它也定义了 MODELS，并设置其父节点为 MMEngine 的 MODELS，这样就建立了层级结构。

```
from mmengine import Registry, MODELS as MMENGINE_MODELS
MODELS = Registry('model', parent=MMENGINE_MODELS, scope='mmalpha')
```

下图是 MMEngine 和 MMAlpha 的注册器层级结构。

可以调用*count_registered_modules* 函数打印已注册到 MMEngine 的模块以及层级结构。

```
from mmengine.registry import count_registered_modules
count_registered_modules()
```

在 MMAlpha 中定义模块 LogSoftmax，并往 MMAlpha 的 MODELS 注册。

```
@MODELS.register_module()
class LogSoftmax(nn.Module):
    def __init__(self, dim=None):
        super().__init__()

    def forward(self, x):
        print('call LogSoftmax.forward')
        return x
```

在 MMAlpha 中使用配置调用 LogSoftmax

```
model = MODELS.build(cfg=dict(type='LogSoftmax'))
```

也可以在 MMAlpha 中调用父节点 MMEngine 的模块。

```
model = MODELS.build(cfg=dict(type='RReLU', lower=0.2))
# 也可以加 scope
model = MODELS.build(cfg=dict(type='mmengine.RReLU'))
```

如果不加前缀，build 方法首先查找当前节点是否存在该模块，如果存在则返回该模块，否则会继续向上查找父节点甚至祖先节点直到找到该模块，因此，如果当前节点和父节点存在同一模块并且希望调用父节点的模块，我们需要指定 scope 前缀。

```
import torch
input = torch.randn(2)
output = model(input)
# call RReLU.forward
print(output)
```

4.3.2 调用兄弟节点的模块

除了可以调用父节点的模块，也可以调用兄弟节点的模块。

假设有另一个项目叫 MMBeta，它和 MMAlpha 一样，定义了 MODELS 以及设置其父节点为 MMEngine 的 MODELS。

```
from mmengine import Registry, MODELS as MMENGINE_MODELS
MODELS = Registry('model', parent=MMENGINE_MODELS, scope='mmbeta')
```

下图是 MMEngine，MMAlpha 和 MMBeta 的注册器层级结构。

在 MMBeta 中调用兄弟节点 MMAlpha 的模块，

```
model = MODELS.build(cfg=dict(type='mmalpha.LogSoftmax'))
output = model(input)
# call LogSoftmax.forward
print(output)
```

调用兄弟节点的模块需要在 type 中指定 scope 前缀，所以上面的配置需要加前缀 mmalpha。

如果需要调用兄弟节点的数个模块，每个模块都加前缀，这需要做大量的修改。于是 MMEngine 引入了*DefaultScope*，Registry 借助它可以很方便地支持临时切换当前节点为指定的节点。

如果需要临时切换当前节点为指定的节点，只需在 cfg 设置 _scope_ 为指定节点的作用域。

```
model = MODELS.build(cfg=dict(type='LogSoftmax', _scope_='mmalpha'))
output = model(input)
# call LogSoftmax.forward
print(output)
```

CHAPTER 5

配置（Config）

MMEngine 实现了抽象的配置类（Config），为用户提供统一的配置访问接口。配置类能够支持不同格式的配置文件，包括 `python`，`json`，`yaml`，用户可以根据需求选择自己偏好的格式。配置类提供了类似字典或者 Python 对象属性的访问接口，用户可以十分自然地进行配置字段的读取和修改。为了方便算法框架管理配置文件，配置类也实现了一些特性，例如配置文件的字段继承等。

在开始教程之前，我们先将教程中需要用到的配置文件下载到本地（建议在临时目录下执行，方便后续删除示例配置文件）：

```
wget https://raw.githubusercontent.com/open-mmlab/mmengine/main/docs/resources/config/
↪config_sgd.py
wget https://raw.githubusercontent.com/open-mmlab/mmengine/main/docs/resources/config/
↪cross_repo.py
wget https://raw.githubusercontent.com/open-mmlab/mmengine/main/docs/resources/config/
↪custom_imports.py
wget https://raw.githubusercontent.com/open-mmlab/mmengine/main/docs/resources/config/
↪demo_train.py
wget https://raw.githubusercontent.com/open-mmlab/mmengine/main/docs/resources/config/
↪example.py
wget https://raw.githubusercontent.com/open-mmlab/mmengine/main/docs/resources/config/
↪learn_read_config.py
wget https://raw.githubusercontent.com/open-mmlab/mmengine/main/docs/resources/config/
↪my_module.py
wget https://raw.githubusercontent.com/open-mmlab/mmengine/main/docs/resources/config/
↪optimizer_cfg.py
wget https://raw.githubusercontent.com/open-mmlab/mmengine/main/docs/resources/config/
↪predefined_var.py
```

(下页继续)

(续上页)

```
wget https://raw.githubusercontent.com/open-mmlab/mmengine/main/docs/resources/config/
↪refer_base_var.py
wget https://raw.githubusercontent.com/open-mmlab/mmengine/main/docs/resources/config/
↪resnet50_delete_key.py
wget https://raw.githubusercontent.com/open-mmlab/mmengine/main/docs/resources/config/
↪resnet50_lr0.01.py
wget https://raw.githubusercontent.com/open-mmlab/mmengine/main/docs/resources/config/
↪resnet50_runtime.py
wget https://raw.githubusercontent.com/open-mmlab/mmengine/main/docs/resources/config/
↪resnet50.py
wget https://raw.githubusercontent.com/open-mmlab/mmengine/main/docs/resources/config/
↪runtime_cfg.py
wget https://raw.githubusercontent.com/open-mmlab/mmengine/main/docs/resources/config/
↪modify_base_var.py
```

5.1 配置文件读取

配置类提供了统一的接口 `Config.fromfile()`，来读取和解析配置文件。

合法的配置文件应该定义一系列键值对，这里举几个不同格式配置文件的例子。

Python 格式：

```python
test_int = 1
test_list = [1, 2, 3]
test_dict = dict(key1='value1', key2=0.1)
```

Json 格式：

```json
{
  "test_int": 1,
  "test_list": [1, 2, 3],
  "test_dict": {"key1": "value1", "key2": 0.1}
}
```

YAML 格式：

```yaml
test_int: 1
test_list: [1, 2, 3]
test_dict:
  key1: "value1"
  key2: 0.1
```

对于以上三种格式的文件，假设文件名分别为 config.py，config.json，config.yml，调用 Config.fromfile('config.xxx') 接口加载这三个文件都会得到相同的结果，构造了包含 3 个字段的配置对象。我们以 config.py 为例，我们先将示例配置文件下载到本地：

然后通过配置类的 fromfile 接口读取配置文件：

```
from mmengine.config import Config

cfg = Config.fromfile('learn_read_config.py')
print(cfg)
```

```
Config (path: learn_read_config.py): {'test_int': 1, 'test_list': [1, 2, 3], 'test_
↪dict': {'key1': 'value1', 'key2': 0.1}}
```

5.2 配置文件的使用

通过读取配置文件来初始化配置对象后，就可以像使用普通字典或者 Python 类一样来使用这个变量了。我们提供了两种访问接口，即类似字典的接口 cfg['key'] 或者类似 Python 对象属性的接口 cfg.key。这两种接口都支持读写。

```
print(cfg.test_int)
print(cfg.test_list)
print(cfg.test_dict)
cfg.test_int = 2

print(cfg['test_int'])
print(cfg['test_list'])
print(cfg['test_dict'])
cfg['test_list'][1] = 3
print(cfg['test_list'])
```

```
1
[1, 2, 3]
{'key1': 'value1', 'key2': 0.1}
2
[1, 2, 3]
{'key1': 'value1', 'key2': 0.1}
[1, 3, 3]
```

注意，配置文件中定义的嵌套字段（即类似字典的字段），在 Config 中会将其转化为 ConfigDict 类，该类继承了 Python 内置字典类型的全部接口，同时也支持以对象属性的方式访问数据。

在算法库中，可以将配置与注册器结合起来使用，达到通过配置文件来控制模块构造的目的。这里举一个在配置文件中定义优化器的例子。

假设我们已经定义了一个优化器的注册器 OPTIMIZERS，包括了各种优化器。那么首先写一个 config_sgd.py：

```
optimizer = dict(type='SGD', lr=0.1, momentum=0.9, weight_decay=0.0001)
```

然后在算法库中可以通过如下代码构造优化器对象。

```
from mmengine import Config, optim
from mmengine.registry import OPTIMIZERS

import torch.nn as nn

cfg = Config.fromfile('config_sgd.py')

model = nn.Conv2d(1, 1, 1)
cfg.optimizer.params = model.parameters()
optimizer = OPTIMIZERS.build(cfg.optimizer)
print(optimizer)
```

```
SGD (
Parameter Group 0
    dampening: 0
    foreach: None
    lr: 0.1
    maximize: False
    momentum: 0.9
    nesterov: False
    weight_decay: 0.0001
)
```

5.3 配置文件的继承

有时候，两个不同的配置文件之间的差异很小，可能仅仅只改了一个字段，我们就需要将所有内容复制粘贴一次，而且在后续观察的时候，不容易定位到具体差异的字段。又有些情况下，多个配置文件可能都有相同的一批字段，我们不得不在这些配置文件中进行复制粘贴，给后续的修改和维护带来了不便。

为了解决这些问题，我们给配置文件增加了继承的机制，即一个配置文件 A 可以将另一个配置文件 B 作为自己的基础，直接继承了 B 中所有字段，而不必显式复制粘贴。

5.3.1 继承机制概述

这里我们举一个例子来说明继承机制。定义如下两个配置文件，

optimizer_cfg.py：

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

resnet50.py：

```
_base_ = ['optimizer_cfg.py']
model = dict(type='ResNet', depth=50)
```

虽然我们在 resnet50.py 中没有定义 optimizer 字段，但由于我们写了 _base_ = ['optimizer_cfg.py']，会使这个配置文件获得 optimizer_cfg.py 中的所有字段。

```
cfg = Config.fromfile('resnet50.py')
print(cfg.optimizer)
```

```
{'type': 'SGD', 'lr': 0.02, 'momentum': 0.9, 'weight_decay': 0.0001}
```

这里 _base_ 是配置文件的保留字段，指定了该配置文件的继承来源。支持继承多个文件，将同时获得这多个文件中的所有字段，但是要求继承的多个文件中**没有**相同名称的字段，否则会报错。

runtime_cfg.py：

```
gpu_ids = [0, 1]
```

resnet50_runtime.py：

```
_base_ = ['optimizer_cfg.py', 'runtime_cfg.py']
model = dict(type='ResNet', depth=50)
```

这时，读取配置文件 resnet50_runtime.py 会获得 3 个字段 model，optimizer，gpu_ids。

```
cfg = Config.fromfile('resnet50_runtime.py')
print(cfg.optimizer)
```

```
{'type': 'SGD', 'lr': 0.02, 'momentum': 0.9, 'weight_decay': 0.0001}
```

通过这种方式，我们可以将配置文件进行拆分，定义一些通用配置文件，在实际配置文件中继承各种通用配置文件，可以减少具体任务的配置流程。

5.3.2 修改继承字段

有时候，我们继承一个配置文件之后，可能需要对其中个别字段进行修改，例如继承了 optimizer_cfg.py 之后，想将学习率从 0.02 修改为 0.01。

这时候，只需要在新的配置文件中，重新定义一下需要修改的字段即可。注意由于 optimizer 这个字段是一个字典，我们只需要重新定义这个字典里面需修改的下级字段即可。这个规则也适用于增加一些下级字段。

resnet50_lr0.01.py:

```
_base_ = ['optimizer_cfg.py', 'runtime_cfg.py']
model = dict(type='ResNet', depth=50)
optimizer = dict(lr=0.01)
```

读取这个配置文件之后，就可以得到期望的结果。

```
cfg = Config.fromfile('resnet50_lr0.01.py')
print(cfg.optimizer)
```

```
{'type': 'SGD', 'lr': 0.01, 'momentum': 0.9, 'weight_decay': 0.0001}
```

对于非字典类型的字段，例如整数，字符串，列表等，重新定义即可完全覆盖，例如下面的写法就将 gpu_ids 这个字段的值修改成了 [0]。

```
_base_ = ['optimizer_cfg.py', 'runtime_cfg.py']
model = dict(type='ResNet', depth=50)
gpu_ids = [0]
```

5.3.3 删除字典中的 key

有时候我们对于继承过来的字典类型字段，不仅仅是想修改其中某些 key，可能还需要删除其中的一些 key。这时候在重新定义这个字典时，需要指定 _delete_=True，表示将没有在新定义的字典中出现的 key 全部删除。

resnet50_delete_key.py:

```
_base_ = ['optimizer_cfg.py', 'runtime_cfg.py']
model = dict(type='ResNet', depth=50)
optimizer = dict(_delete_=True, type='SGD', lr=0.01)
```

这时候，optimizer 这个字典中就只有 type 和 lr 这两个 key，momentum 和 weight_decay 将不再被继承。

```
cfg = Config.fromfile('resnet50_delete_key.py')
print(cfg.optimizer)
```

```
{'type': 'SGD', 'lr': 0.01}
```

5.3.4 引用被继承文件中的变量

有时我们想重复利用 _base_ 中定义的字段内容，就可以通过 {{_base_.xxxx}} 获取来获取对应变量的拷贝。例如：

refer_base_var.py

```
_base_ = ['resnet50.py']
a = {{_base_.model}}
```

解析后发现，a 的值变成了 resnet50.py 中定义的 model

```
cfg = Config.fromfile('refer_base_var.py')
print(cfg.a)
```

```
{'type': 'ResNet', 'depth': 50}
```

我们可以在 json、yaml、python 三种类型的配置文件中，使用这种方式来获取 _base_ 中定义的变量。

尽管这种获取 _base_ 中定义变量的方式非常通用，但是在语法上存在一些限制，无法充分利用 python 类配置文件的动态特性。比如我们想在 python 类配置文件中，修改 _base_ 中定义的变量：

```
_base_ = ['resnet50.py']
a = {{_base_.model}}
a['type'] = 'MobileNet'
```

配置类是无法解析这样的配置文件的（解析时报错）。配置类提供了一种更 pythonic 的方式，让我们能够在 python 类配置文件中修改 _base_ 中定义的变量（python 类配置文件专属特性，目前不支持在 json、yaml 配置文件中修改 _base_ 中定义的变量）。

modify_base_var.py：

```
_base_ = ['resnet50.py']
a = _base_.model
a.type = 'MobileNet'
```

```
cfg = Config.fromfile('modify_base_var.py')
print(cfg.a)
```

```
{'type': 'MobileNet', 'depth': 50}
```

解析后发现，a 的 type 变成了 MobileNet。

5.4 配置文件的导出

在启动训练脚本时，用户可能通过传参的方式来修改配置文件的部分字段，为此我们提供了 dump 接口来导出更改后的配置文件。与读取配置文件类似，用户可以通过 `cfg.dump('config.xxx')` 来选择导出文件的格式。dump 同样可以导出有继承关系的配置文件，导出的文件可以被独立使用，不再依赖于 _base_ 中定义的文件。

基于继承一节定义的 `resnet50.py`, 我们将其加载后导出:

```
cfg = Config.fromfile('resnet50.py')
cfg.dump('resnet50_dump.py')
```

`resnet50_dump.py`

```
optimizer = dict(type='SGD', lr=0.02, momentum=0.9, weight_decay=0.0001)
model = dict(type='ResNet', depth=50)
```

类似的，我们可以导出 json、yaml 格式的配置文件

`resnet50_dump.yaml`

```
model:
  depth: 50
  type: ResNet
optimizer:
  lr: 0.02
  momentum: 0.9
  type: SGD
  weight_decay: 0.0001
```

`resnet50_dump.json`

```
{"optimizer": {"type": "SGD", "lr": 0.02, "momentum": 0.9, "weight_decay": 0.0001},
↪"model": {"type": "ResNet", "depth": 50}}
```

此外，dump 不仅能导出加载自文件的 `cfg`，还能导出加载自字典的 `cfg`

```
cfg = Config(dict(a=1, b=2))
cfg.dump('dump_dict.py')
```

`dump_dict.py`

```
a=1
b=2
```

5.5 其他进阶用法

这里介绍一下配置类的进阶用法，这些小技巧可能使用户开发和使用算法库更简单方便。

5.5.1 预定义字段

有时候我们希望配置文件中的一些字段和当前路径或者文件名等相关，这里举一个典型使用场景的例子。在训练模型时，我们会在配置文件中定义一个工作目录，存放这组实验配置的模型和日志，那么对于不同的配置文件，我们期望定义不同的工作目录。用户的一种常见选择是，直接使用配置文件名作为工作目录名的一部分，例如对于配置文件 `predefined_var.py`，工作目录就是 `./work_dir/predefined_var`。

使用预定义字段可以方便地实现这种需求，在配置文件 `predefined_var.py` 中可以这样写：

```
work_dir = './work_dir/{{fileBasenameNoExtension}}'
```

这里 `{{fileBasenameNoExtension}}` 表示该配置文件的文件名（不含拓展名），在配置类读取配置文件的时候，会将这种用双花括号包起来的字符串自动解析为对应的实际值。

```
cfg = Config.fromfile('./predefined_var.py')
print(cfg.work_dir)
```

```
./work_dir/predefined_var
```

目前支持的预定义字段有以下四种，变量名参考自 VS Code 中的相关字段：

- `{{fileDirname}}` - 当前文件的目录名，例如 `/home/your-username/your-project/folder`
- `{{fileBasename}}` - 当前文件的文件名，例如 `file.py`
- `{{fileBasenameNoExtension}}` - 当前文件不包含扩展名的文件名，例如 file
- `{{fileExtname}}` - 当前文件的扩展名，例如 `.py`

5.5.2 命令行修改配置

有时候我们只希望修改部分配置，而不想修改配置文件本身，例如实验过程中想更换学习率，但是又不想重新写一个配置文件，常用的做法是在命令行传入参数来覆盖相关配置。考虑到我们想修改的配置通常是一些内层参数，如优化器的学习率、模型卷积层的通道数等，因此 MMEngine 提供了一套标准的流程，让我们能够在命令行里轻松修改配置文件中任意层级的参数。

1. 使用 `argparser` 解析脚本运行的参数
2. 使用 `argparse.ArgumentParser.add_argument` 方法时，让 `action` 参数的值为*DictAction*，用它来进一步解析命令行参数中用于修改配置文件的参数
3. 使用配置类的 `merge_from_dict` 方法来更新配置

启动脚本示例如下：

demo_train.py

```
import argparse

from mmengine.config import Config, DictAction


def parse_args():
    parser = argparse.ArgumentParser(description='Train a model')
    parser.add_argument('config', help='train config file path')
    parser.add_argument(
        '--cfg-options',
        nargs='+',
        action=DictAction,
        help='override some settings in the used config, the key-value pair '
        'in xxx=yyy format will be merged into config file. If the value to '
        'be overwritten is a list, it should be like key="[a,b]" or key=a,b '
        'It also allows nested list/tuple values, e.g. key="[(a,b),(c,d)]" '
        'Note that the quotation marks are necessary and that no white space '
        'is allowed.')

    args = parser.parse_args()
    return args


def main():
    args = parse_args()
    cfg = Config.fromfile(args.config)
    if args.cfg_options is not None:
        cfg.merge_from_dict(args.cfg_options)
    print(cfg)


if __name__ == '__main__':
    main()
```

示例配置文件如下：

example.py

```
model = dict(type='CustomModel', in_channels=[1, 2, 3])
optimizer = dict(type='SGD', lr=0.01)
```

我们在命令行里通过 . 的方式来访问配置文件中的深层配置，例如我们想修改学习率，只需要在命令行执行：

```
python demo_train.py ./example.py --cfg-options optimizer.lr=0.1
```

```
Config (path: ./example.py): {'model': {'type': 'CustomModel', 'in_channels': [1, 2,␣
↪3]}, 'optimizer': {'type': 'SGD', 'lr': 0.1}}
```

我们成功地把学习率从 0.01 修改成 0.1。如果想改变列表、元组类型的配置，如上例中的 `in_channels`，则需要在命令行赋值时给 `()`，`[]` 外加上双引号：

```
python demo_train.py ./example.py --cfg-options model.in_channels="[1, 1, 1]"
```

```
Config (path: ./example.py): {'model': {'type': 'CustomModel', 'in_channels': [1, 1,␣
↪1]}, 'optimizer': {'type': 'SGD', 'lr': 0.01}}
```

`model.in_channels` 已经从 [1, 2, 3] 修改成 [1, 1, 1]。

注解： 上述流程只支持在命令行里修改字符串、整型、浮点型、布尔型、None、列表、元组类型的配置项。对于列表、元组类型的配置，里面每个元素的类型也必须为上述七种类型之一。

5.5.3 导入自定义 Python 模块

将配置与注册器结合起来使用时，如果我们往注册器中注册了一些自定义的类，就可能会遇到一些问题。因为读取配置文件的时候，这部分代码可能还没有被执行到，所以并未完成注册过程，从而导致构建自定义类的时候报错。

例如我们新实现了一种优化器 `CustomOptim`，相应代码在 `my_module.py` 中。

```
from mmengine.registry import OPTIMIZERS

@OPTIMIZERS.register_module()
class CustomOptim:
    pass
```

我们为这个优化器的使用写了一个新的配置文件 `custom_imports.py`：

```
optimizer = dict(type='CustomOptim')
```

那么就需要在读取配置文件和构造优化器之前，增加一行 `import my_module` 来保证将自定义的类 `CustomOptim` 注册到 OPTIMIZERS 注册器中：为了解决这个问题，我们给配置文件定义了一个保留字段 `custom_imports`，用于将需要提前导入的 Python 模块，直接写在配置文件中。对于上述例子，就可以将配置文件写成如下：

`custom_imports.py`

```
custom_imports = dict(imports=['my_module'], allow_failed_imports=False)
optimizer = dict(type='CustomOptim')
```

这样我们就不用在训练代码中增加对应的 import 语句，只需要修改配置文件就可以实现非侵入式导入自定义注册模块。

```
cfg = Config.fromfile('custom_imports.py')

from mmengine.registry import OPTIMIZERS

custom_optim = OPTIMIZERS.build(cfg.optimizer)
print(custom_optim)
```

```
<my_module.CustomOptim object at 0x7f6983a87970>
```

5.5.4 跨项目继承配置文件

为了避免基于已有算法库开发新项目时需要复制大量的配置文件，MMEngine 的配置类支持配置文件的跨项目继承。例如我们基于 MMDetection 开发新的算法库，需要使用以下 MMDetection 的配置文件：

```
configs/_base_/schedules/schedule_1x.py
configs/_base_/datasets.coco_instance.py
configs/_base_/default_runtime.py
configs/_base_/models/faster_rcnn_r50_fpn.py
```

如果没有配置文件跨项目继承的功能，我们就需要把 MMDetection 的配置文件拷贝到当前项目，而我们现在只需要安装 MMDetection （如使用 `mim install mmdet`），在新项目的配置文件中按照以下方式继承 MMDetection 的配置文件：

`cross_repo.py`

```
_base_ = [
    'mmdet::_base_/schedules/schedule_1x.py',
    'mmdet::_base_/datasets/coco_instance.py',
    'mmdet::_base_/default_runtime.py',
    'mmdet::_base_/models/faster_rcnn_r50_fpn.py',
]
```

我们可以像加载普通配置文件一样加载 `cross_repo.py`

```
cfg = Config.fromfile('cross_repo.py')
print(cfg.train_cfg)
```

```
{'type': 'EpochBasedTrainLoop', 'max_epochs': 12, 'val_interval': 1, '_scope_': 'mmdet
↪'}
```

通过指定 `mmdet::`，Config 类会去检索 mmdet 包中的配置文件目录，并继承指定的配置文件。实际上，只要算法库的 `setup.py` 文件符合 MMEngine 安装规范，在正确安装算法库以后，新的项目就可以使用上述用法去继承已有算法库的配置文件而无需拷贝。

5.5.5 跨项目获取配置文件

MMEngine 还提供了 `get_config` 和 `get_model` 两个接口，支持对符合 MMEngine 安装规范的算法库中的模型和配置文件做索引并进行 API 调用。通过 `get_model` 接口可以获得构建好的模型。通过 `get_config` 接口可以获得配置文件。

`get_model` 的使用样例如下所示，使用和跨项目继承配置文件相同的语法，指定 `mmdet::`，即可在 mmdet 包中检索对应的配置文件并构建和初始化相应模型。用户可以通过指定 `pretrained=True` 获得已经加载预训练权重的模型以进行训练或者推理。

```
from mmengine.hub import get_model

model = get_model(
    'mmdet::faster_rcnn/faster_rcnn_r50_fpn_1x_coco.py', pretrained=True)
print(type(model))
```

```
http loads checkpoint from path: https://download.openmmlab.com/mmdetection/v2.0/
↪faster_rcnn/faster_rcnn_r50_fpn_1x_coco/faster_rcnn_r50_fpn_1x_coco_20200130-
↪047c8118.pth
<class 'mmdet.models.detectors.faster_rcnn.FasterRCNN'>
```

`get_config` 的使用样例如下所示，使用和跨项目继承配置文件相同的语法，指定 `mmdet::`，即可实现去 mmdet 包中检索并加载对应的配置文件。用户可以基于这样得到的配置文件进行推理修改并自定义自己的算法模型。同时，如果用户指定 `pretrained=True`，得到的配置文件中会新增 `model_path` 字段，指定了对应模型预训练权重的路径。

```
from mmengine.hub import get_config

cfg = get_config(
    'mmdet::faster_rcnn/faster_rcnn_r50_fpn_1x_coco.py', pretrained=True)
print(cfg.model_path)
```

```
https://download.openmmlab.com/mmdetection/v2.0/faster_rcnn/faster_rcnn_r50_fpn_1x_
↪coco/faster_rcnn_r50_fpn_1x_coco_20200130-047c8118.pth
```

CHAPTER 6

执行器（Runner）

深度学习算法的训练、验证和测试通常都拥有相似的流程，因此 MMEngine 提供了执行器以帮助用户简化这些任务的实现流程。用户只需要准备好模型训练、验证、测试所需要的模块构建执行器，便能够通过简单调用执行器的接口来完成这些任务。用户如果需要使用这几项功能中的某一项，只需要准备好对应功能所依赖的模块即可。

用户可以手动构建这些模块的实例，也可以通过编写配置文件，由执行器自动从注册器中构建所需要的模块，我们推荐使用后一种方式。

6.1 手动构建模块来使用执行器

6.1.1 手动构建模块进行训练

如上文所说，使用执行器的某一项功能时需要准备好对应功能所依赖的模块。以使用执行器的训练功能为例，用户需要准备模型、优化器 、参数调度器 还有训练数据集 。

```
# 准备训练任务所需要的模块
import torch
from torch import nn
from torchvision import transforms
from torchvision import datasets
from torch.utils.data import DataLoader
from mmengine.model import BaseModel
from mmengine.optim.scheduler import MultiStepLR
```

(下页继续)

(续上页)

```
# 定义一个多层感知机网络
class Network(BaseModel):
    def __init__(self):
        super().__init__()
        self.mlp = nn.Sequential(nn.Linear(28 * 28, 128), nn.ReLU(), nn.Linear(128,␣
↪128), nn.ReLU(), nn.Linear(128, 10))
        self.loss = nn.CrossEntropyLoss()

    def forward(self, batch_inputs: torch.Tensor, data_samples = None, mode: str =
↪'tensor'):
        x = batch_inputs.flatten(1)
        x = self.mlp(x)
        if mode == 'loss':
            return {'loss': self.loss(x, data_samples)}
        elif mode == 'predict':
            return x.argmax(1)
        else:
            return x

model = Network()

# 构建优化器
optimizer = torch.optim.SGD(model.parameters(), lr=0.01, momentum=0.9)
# 构建参数调度器用于调整学习率
lr_scheduler = MultiStepLR(optimizer, milestones=[2], by_epoch=True)
# 构建手写数字识别 (MNIST) 数据集
train_dataset = datasets.MNIST(root="MNIST", download=True, train=True,␣
↪transform=transforms.ToTensor())
# 构建数据加载器
train_dataloader = DataLoader(dataset=train_dataset, batch_size=10, num_workers=2)
```

在创建完符合上述文档规范的模块的对象后，就可以使用这些模块初始化执行器：

```
from mmengine.runner import Runner


# 训练相关参数设置，按轮次训练，训练 3 轮
train_cfg = dict(by_epoch=True, max_epochs=3)

# 初始化执行器
runner = Runner(model,
                work_dir='./train_mnist',  # 工作目录，用于保存模型和日志
                train_cfg=train_cfg,
```

(下页继续)

(续上页)

```
                train_dataloader=train_dataloader,
                optim_wrapper=dict(optimizer=optimizer),
                param_scheduler=lr_scheduler)
# 执行训练
runner.train()
```

上面的例子中，我们手动构建了一个多层感知机网络和手写数字识别 (MNIST) 数据集，以及训练所需要的优化器和学习率调度器，使用这些模块初始化了执行器，并且设置了训练配置 `train_cfg`，让执行器将模型训练 3 个轮次，最后通过调用执行器的 `train` 方法进行模型训练。

用户也可以修改 `train_cfg` 使执行器按迭代次数控制训练：

```
# 训练相关参数设置，按迭代次数训练，训练 9000 次迭代
train_cfg = dict(by_epoch=False, max_iters=9000)
```

6.1.2 手动构建模块进行测试

再举一个模型测试的例子，模型的测试需要用户准备模型和训练好的权重路径、测试数据集以及评测器：

```
from mmengine.evaluator import BaseMetric


class MnistAccuracy(BaseMetric):
    def process(self, data, preds) -> None:
        self.results.append(((data[1] == preds.cpu()).sum(), len(preds)))
    def compute_metrics(self, results):
        correct, batch_size = zip(*results)
        acc = sum(correct) / sum(batch_size)
        return dict(accuracy=acc)

model = Network()
test_dataset = datasets.MNIST(root="MNIST", download=True, train=False,␣
↪transform=transforms.ToTensor())
test_dataloader = DataLoader(dataset=test_dataset)
metric = MnistAccuracy()
test_evaluator = Evaluator(metric)

# 初始化执行器
runner = Runner(model=model, test_dataloader=test_dataloader, test_evaluator=test_
↪evaluator,
                load_from='./train_mnist/epoch_3.pth', work_dir='./test_mnist')

# 执行测试
```

(下页继续)

(续上页)

```
runner.test()
```

这个例子中我们重新手动构建了一个多层感知机网络，以及测试用的手写数字识别数据集和使用 (Accuracy) 指标的评测器，并使用这些模块初始化执行器，最后通过调用执行器的 `test` 函数进行模型测试。

6.1.3 手动构建模块在训练过程中进行验证

在模型训练过程中，通常会按一定的间隔在验证集上对模型进行验证。在使用 MMEngine 时，只需要构建训练和验证的模块，并在训练配置中设置验证间隔即可

```
# 准备训练任务所需要的模块
optimzier = torch.optim.SGD(model.parameters(), lr=0.01, momentum=0.9)
lr_scheduler = MultiStepLR(milestones=[2], by_epoch=True)
train_dataset = datasets.MNIST(root="MNIST", download=True, train=True,␣
↪transform=transforms.ToTensor())
train_dataloader = DataLoader(dataset=train_dataset, batch_size=10, num_workers=2)

# 准备验证需要的模块
val_dataset = datasets.MNIST(root="MNIST", download=True, train=False,␣
↪transform=transforms.ToTensor())
val_dataloader = Dataloader(dataset=val_dataset)
metric = MnistAccuracy()
val_evaluator = Evaluator(metric)


# 训练相关参数设置
train_cfg = dict(by_epoch=True,  # 按轮次训练
                 max_epochs=5,  # 训练 5 轮
                 val_begin=2,  # 从第 2 个 epoch 开始验证
                 val_interval=1)  # 每隔 1 轮进行 1 次验证

# 初始化执行器
runner = Runner(model=model, optim_wrapper=dict(optimizer=optimzier), param_
↪scheduler=lr_scheduler,
                train_dataloader=train_dataloader, val_dataloader=val_dataloader, val_
↪evaluator=val_evaluator,
                train_cfg=train_cfg, work_dir='./train_val_mnist')
# 执行训练
runner.train()
```

6.2 通过配置文件使用执行器

OpenMMLab 的开源项目普遍使用注册器 + 配置文件的方式来管理和构建模块，MMEngine 中的执行器也推荐使用配置文件进行构建。下面是一个通过配置文件使用执行器的例子：

```
from mmengine import Config
from mmengine.runner import Runner

# 加载配置文件
config = Config.fromfile('configs/resnet/resnet50_8xb32_in1k.py')

# 通过配置文件初始化执行器
runner = Runner.build_from_cfg(config)

# 执行训练
runner.train()

# 执行测试
runner.test()
```

与手动构建模块来使用执行器不同的是，通过调用 Runner 类的 `build_from_cfg` 方法，执行器能够自动读取配置文件中的模块配置，从相应的注册器中构建所需要的模块，用户不再需要考虑训练和测试分别依赖哪些模块，也不需要为了切换训练的模型和数据而大量改动代码。

下面是一个典型的使用配置文件调用 MMClassification 中的模块训练分类器的简单例子：

```
# 工作目录，保存权重和日志
work_dir = './train_resnet'
# 默认注册器域
default_scope = 'mmcls'  # 默认使用 `mmcls` (MMClassification) 注册器中的模块
# 模型配置
model = dict(type='ImageClassifier',
             backbone=dict(type='ResNet', depth=50),
             neck=dict(type='GlobalAveragePooling'),
             head=dict(type='LinearClsHead',num_classes=1000))
# 数据配置
train_dataloader = dict(dataset=dict(type='ImageNet', pipeline=[...]),
                        sampler=dict(type='DefaultSampler', shuffle=True),
                        batch_size=32,
                        num_workers=4)
val_dataloader = ...
test_dataloader = ...

# 优化器配置
```

(下页继续)

(续上页)

```
optim_wrapper = dict(
    optimizer=dict(type='SGD', lr=0.1, momentum=0.9, weight_decay=0.0001))
# 参数调度器配置
param_scheduler = dict(
    type='MultiStepLR', by_epoch=True, milestones=[30, 60, 90], gamma=0.1)
# 验证和测试的评测器配置
val_evaluator = dict(type='Accuracy')
test_evaluator = dict(type='Accuracy')

# 训练、验证、测试流程配置
train_cfg = dict(
    by_epoch=True,
    max_epochs=100,
    val_begin=20,  # 从第 20 个 epoch 开始验证
    val_interval=1  # 每隔一个 epoch 进行一次验证
)
val_cfg = dict()
test_cfg = dict()

# 自定义钩子（可选）
custom_hooks = [...]

# 默认钩子（可选，未在配置文件中写明时将使用默认配置）
default_hooks = dict(
    runtime_info=dict(type='RuntimeInfoHook'),  # 运行时信息钩子
    timer=dict(type='IterTimerHook'),  # 计时器钩子
    sampler_seed=dict(type='DistSamplerSeedHook'),  # 为每轮次的数据采样设置随机种子的钩子
    logger=dict(type='TextLoggerHook'),  # 训练日志钩子
    param_scheduler=dict(type='ParamSchedulerHook'),  # 参数调度器执行钩子
    checkpoint=dict(type='CheckpointHook', interval=1),  # 模型保存钩子
)

# 环境配置（可选，未在配置文件中写明时将使用默认配置）
env_cfg = dict(
    cudnn_benchmark=False,  # 是否使用 cudnn_benchmark
    dist_cfg=dict(backend='nccl'),  # 分布式通信后端
    mp_cfg=dict(mp_start_method='fork')  # 多进程设置
)
# 日志处理器（可选，未在配置文件中写明时将使用默认配置）
log_processor = dict(type='LogProcessor', window_size=50, by_epoch=True)
# 日志等级配置
log_level = 'INFO'
```

(下页继续)

(续上页)

```
# 加载权重的路径 (None 表示不加载)
load_from = None
# 从加载的权重文件中恢复训练
resume = False
```

一个完整的配置文件主要由模型、数据、优化器、参数调度器、评测器等模块的配置，训练、验证、测试等流程的配置，还有执行流程过程中的各种钩子模块的配置，以及环境和日志等其他配置的字段组成。通过配置文件构建的执行器采用了懒初始化 (lazy initialization)，只有当调用到训练或测试等执行函数时，才会根据配置文件去完整初始化所需要的模块。

关于配置文件的更详细的使用方式，请参考配置文件教程

6.3 加载权重或恢复训练

执行器可以通过 load_from 参数加载检查点（checkpoint）文件中的模型权重，只需要将 load_from 参数设置为检查点文件的路径即可。

```
runner = Runner(model=model, test_dataloader=test_dataloader, test_evaluator=test_
↪evaluator,
                load_from='./resnet50.pth')
```

如果是通过配置文件使用执行器，只需修改配置文件中的 load_from 字段即可。

用户也可通过设置 resume=True 来，加载检查点中的训练状态信息来恢复训练。当 load_from 和 resume=True 同时被设置时，执行器将加载 load_from 路径对应的检查点文件中的训练状态。

如果仅设置 resume=True，执行器将会尝试从 work_dir 文件夹中寻找并读取最新的检查点文件。

你可能还想阅读执行器的设计或者执行器的 *API* 文档。

CHAPTER 7

钩子（Hook）

钩子编程是一种编程模式，是指在程序的一个或者多个位置设置位点（挂载点），当程序运行至某个位点时，会自动调用运行时注册到位点的所有方法。钩子编程可以提高程序的灵活性和拓展性，用户将自定义的方法注册到位点便可被调用而无需修改程序中的代码。

7.1 内置钩子

MMEngine 提供了很多内置的钩子，将钩子分为两类，分别是默认钩子以及自定义钩子，前者表示会默认往执行器注册，后者表示需要用户自己注册。

每个钩子都有对应的优先级，在同一位点，钩子的优先级越高，越早被执行器调用，如果优先级一样，被调用的顺序和钩子注册的顺序一致。优先级列表如下：

- HIGHEST (0)
- VERY_HIGH (10)
- HIGH (30)
- ABOVE_NORMAL (40)
- NORMAL (50)
- BELOW_NORMAL (60)
- LOW (70)
- VERY_LOW (90)

- LOWEST (100)

默认钩子

自定义钩子

注解： 不建议修改默认钩子的优先级，因为优先级低的钩子可能会依赖优先级高的钩子。例如 CheckpointHook 的优先级需要比 ParamSchedulerHook 低，这样保存的优化器状态才是正确的状态。另外，自定义钩子的优先级默认为 `NORMAL (50)`。

两种钩子在执行器中的设置不同，默认钩子的配置传给执行器的 `default_hooks` 参数，自定义钩子的配置传给 `custom_hooks` 参数，如下所示：

```
from mmengine.runner import Runner

default_hooks = dict(
    runtime_info=dict(type='RuntimeInfoHook'),
    timer=dict(type='IterTimerHook'),
    sampler_seed=dict(type='DistSamplerSeedHook'),
    logger=dict(type='LoggerHook'),
    param_scheduler=dict(type='ParamSchedulerHook'),
    checkpoint=dict(type='CheckpointHook', interval=1),
)

custom_hooks = [
    dict(type='NaiveVisualizationHook', priority='LOWEST'),
]

runner = Runner(default_hooks=default_hooks, custom_hooks=custom_hooks, ...)
runner.train()
```

下面逐一介绍 MMEngine 中内置钩子的用法。

7.1.1 CheckpointHook

CheckpointHook 按照给定间隔保存模型的权重，如果是分布式多卡训练，则只有主（master）进程会保存权重。`CheckpointHook` 的主要功能如下：

- 按照间隔保存权重，支持按 epoch 数或者 iteration 数保存权重
- 保存最新的多个权重
- 保存最优权重
- 指定保存权重的路径

如需了解其他功能，请阅读*CheckpointHook API 文档*。

下面介绍上面提到的 4 个功能。

- 按照间隔保存权重，支持按 epoch 数或者 iteration 数保存权重

假设我们一共训练 20 个 epoch 并希望每隔 5 个 epoch 保存一次权重，下面的配置即可帮我们实现该需求。

```
# by_epoch 的默认值为 True
default_hooks = dict(checkpoint=dict(type='CheckpointHook', interval=5, by_
↪epoch=True))
```

如果想以迭代次数作为保存间隔，则可以将 by_epoch 设为 False，interval=5 则表示每迭代 5 次保存一次权重。

```
default_hooks = dict(checkpoint=dict(type='CheckpointHook', interval=5, by_
↪epoch=False))
```

- 保存最新的多个权重

如果只想保存一定数量的权重，可以通过设置 max_keep_ckpts 参数实现最多保存 max_keep_ckpts 个权重，当保存的权重数超过 max_keep_ckpts 时，前面的权重会被删除。

```
default_hooks = dict(checkpoint=dict(type='CheckpointHook', interval=5, max_keep_
↪ckpts=2))
```

上述例子表示，假如一共训练 20 个 epoch，那么会在第 5, 10, 15, 20 个 epoch 保存模型，但是在第 15 个 epoch 的时候会删除第 5 个 epoch 保存的权重，在第 20 个 epoch 的时候会删除第 10 个 epoch 的权重，最终只有第 15 和第 20 个 epoch 的权重才会被保存。

- 保存最优权重

如果想要保存训练过程验证集的最优权重，可以设置 save_best 参数，如果设置为 'auto'，则会根据验证集的第一个评价指标（验证集返回的评价指标是一个有序字典）判断当前权重是否最优。

```
default_hooks = dict(checkpoint=dict(type='CheckpointHook', save_best='auto'))
```

也可以直接指定 save_best 的值为评价指标，例如在分类任务中，可以指定为 save_best='top-1'，则会根据 'top-1' 的值判断当前权重是否最优。

除了 save_best 参数，和保存最优权重相关的参数还有 rule，greater_keys 和 less_keys，这三者用来判断 save_best 的值是越大越好还是越小越好。例如指定了 save_best='top-1'，可以指定 rule='greater'，则表示该值越大表示权重越好。

- 指定保存权重的路径

权重默认保存在工作目录（work_dir），但可以通过设置 out_dir 改变保存路径。

```
default_hooks = dict(checkpoint=dict(type='CheckpointHook', interval=5, out_dir='/
↪path/of/directory'))
```

7.1.2 LoggerHook

LoggerHook 负责收集日志并把日志输出到终端或者输出到文件、TensorBoard 等后端。

如果我们希望每迭代 20 次就输出（或保存）一次日志，我们可以设置 `interval` 参数，配置如下：

```
default_hooks = dict(logger=dict(type='LoggerHook', interval=20))
```

如果你对日志的管理感兴趣，可以阅读记录日志（logging）。

7.1.3 ParamSchedulerHook

ParamSchedulerHook 遍历执行器的所有优化器参数调整策略（Parameter Scheduler）并逐个调用 step 方法更新优化器的参数。如需了解优化器参数调整策略的用法请阅读文档。`ParamSchedulerHook` 默认注册到执行器并且没有可配置的参数，所以无需对其做任何配置。

7.1.4 IterTimerHook

IterTimerHook 用于记录加载数据的时间以及迭代一次耗费的时间。`IterTimerHook` 默认注册到执行器并且没有可配置的参数，所以无需对其做任何配置。

7.1.5 DistSamplerSeedHook

DistSamplerSeedHook 在分布式训练时调用 Sampler 的 step 方法以确保 shuffle 参数生效。`DistSamplerSeedHook` 默认注册到执行器并且没有可配置的参数，所以无需对其做任何配置。

7.1.6 RuntimeInfoHook

RuntimeInfoHook 会在执行器的不同钩子位点将当前的运行时信息（如 epoch、iter、max_epochs、max_iters、lr、metrics 等）更新至 message hub 中，以便其他无法访问执行器的模块能够获取到这些信息。`RuntimeInfoHook` 默认注册到执行器并且没有可配置的参数，所以无需对其做任何配置。

7.1.7 EMAHook

EMAHook 在训练过程中对模型执行指数滑动平均操作，目的是提高模型的鲁棒性。注意：指数滑动平均生成的模型只用于验证和测试，不影响训练。

```
custom_hooks = [dict(type='EMAHook')]
runner = Runner(custom_hooks=custom_hooks, ...)
runner.train()
```

EMAHook 默认使用 ExponentialMovingAverage，可选值还有 StochasticWeightAverage 和 MomentumAnnealingEMA。可以通过设置 ema_type 使用其他的平均策略。

```
custom_hooks = [dict(type='EMAHook', ema_type='StochasticWeightAverage')]
```

更多用法请阅读*EMAHook API 文档*。

7.1.8 EmptyCacheHook

EmptyCacheHook 调用 torch.cuda.empty_cache() 释放未被使用的显存。可以通过设置 before_epoch, after_iter 以及 after_epoch 参数控制释显存的时机，第一个参数表示在每个 epoch 开始之前，第二参数表示在每次迭代之后，第三个参数表示在每个 epoch 之后。

```
# 每一个 epoch 结束都会执行释放操作
custom_hooks = [dict(type='EmptyCacheHook', after_epoch=True)]
runner = Runner(custom_hooks=custom_hooks, ...)
runner.train()
```

7.1.9 SyncBuffersHook

SyncBuffersHook 在分布式训练每一轮（epoch）结束时同步模型的 buffer，例如 BN 层的 running_mean 以及 running_var。

```
custom_hooks = [dict(type='SyncBuffersHook')]
runner = Runner(custom_hooks=custom_hooks, ...)
runner.train()
```

7.2 自定义钩子

如果 MMEngine 提供的默认钩子不能满足需求，用户可以自定义钩子，只需继承钩子基类并重写相应的位点方法。

例如，如果希望在训练的过程中判断损失值是否有效，如果值为无穷大则无效，我们可以在每次迭代后判断损失值是否无穷大，因此只需重写 after_train_iter 位点。

```
import torch

from mmengine.registry import HOOKS
from mmengine.hooks import Hook
```

(下页继续)

(续上页)

```
@HOOKS.register_module()
class CheckInvalidLossHook(Hook):
    """Check invalid loss hook.

    This hook will regularly check whether the loss is valid
    during training.

    Args:
        interval (int): Checking interval (every k iterations).
            Defaults to 50.
    """

    def __init__(self, interval=50):
        self.interval = interval

    def after_train_iter(self, runner, batch_idx, data_batch=None, outputs=None):
        """All subclasses should override this method, if they need any
        operations after each training iteration.

        Args:
            runner (Runner): The runner of the training process.
            batch_idx (int): The index of the current batch in the train loop.
            data_batch (dict or tuple or list, optional): Data from dataloader.
            outputs (dict, optional): Outputs from model.
        """
        if self.every_n_train_iters(runner, self.interval):
            assert torch.isfinite(outputs['loss']),\
                runner.logger.info('loss become infinite or NaN!')
```

我们只需将钩子的配置传给执行器的 custom_hooks 的参数，执行器初始化的时候会注册钩子，

```
from mmengine.runner import Runner

custom_hooks = dict(
    dict(type='CheckInvalidLossHook', interval=50)
)
runner = Runner(custom_hooks=custom_hooks, ...)  # 实例化执行器，主要完成环境的初始化以及各种模块的构建
runner.train()  # 执行器开始训练
```

便会在每次模型前向计算后检查损失值。

注意，自定义钩子的优先级默认为 NORMAL (50)，如果想改变钩子的优先级，则可以在配置中设置 priority 字段。

```
custom_hooks = dict(
    dict(type='CheckInvalidLossHook', interval=50, priority='ABOVE_NORMAL')
)
```

也可以在定义类时给定优先级

```
@HOOKS.register_module()
class CheckInvalidLossHook(Hook):

    priority = 'ABOVE_NORMAL'
```

你可能还想阅读钩子的设计或者钩子的 *API* 文档。

CHAPTER 8

模型（Model）

在训练深度学习任务时，我们通常需要定义一个模型来实现算法的主体。在基于 MMEngine 开发时，模型由执行器管理，需要实现 `train_step`，`val_step` 和 `test_step` 方法。

对于检测、识别、分割一类的深度学习任务，上述方法通常为标准的流程，例如在 `train_step` 里更新参数，返回损失；`val_step` 和 `test_step` 返回预测结果。因此 MMEngine 抽象出模型基类*BaseModel*，实现了上述接口的标准流程。我们只需要让模型继承自模型基类，并按照一定的规范实现 `forward`，就能让模型在执行器中运行起来。

模型基类继承自模块基类，能够通过配置 `init_cfg` 灵活的选择初始化方式。

8.1 接口约定

forward: `forward` 的入参需要和 DataLoader 的输出保持一致 (自定义数据处理器除外)，如果 `DataLoader` 返回元组类型的数据 `data`，`forward` 需要能够接受 `*data` 的解包后的参数；如果返回字典类型的数据 `data`，`forward` 需要能够接受 `**data` 解包后的参数。`mode` 参数用于控制 forward 的返回结果：

- `mode='loss'`：`loss` 模式通常在训练阶段启用，并返回一个损失字典。损失字典的 key-value 分别为损失名和可微的 `torch.Tensor`。字典中记录的损失会被用于更新参数和记录日志。模型基类会在 `train_step` 方法中调用该模式的 `forward`。
- `mode='predict'`：`predict` 模式通常在验证、测试阶段启用，并返回列表/元组形式的预测结果，预测结果需要和*process* 接口的参数相匹配。OpenMMLab 系列算法对 `predict` 模式的输出有着更加严格的约定，需要输出列表形式的数据元素。模型基类会在 `val_step`，`test_step` 方法中调用该模式的 `forward`。

- mode='tensor'：tensor 和 predict 模式均返回模型的前向推理结果，区别在于 tensor 模式下，forward 会返回未经后处理的张量，例如返回未经非极大值抑制（nms）处理的检测结果，返回未经 argmax 处理的分类结果。我们可以基于 tensor 模式的结果进行自定义的后处理。

train_step: 调用 loss 模式的 forward 接口，得到损失字典。模型基类基于优化器封装 实现了标准的梯度计算、参数更新、梯度清零流程。

val_step: 调用 predict 模式的 forward，返回预测结果，预测结果会被进一步传给评测器的*process* 接口和钩子（*Hook*）的 after_val_iter 接口。

test_step: 同 val_step，预测结果会被进一步传给 after_test_iter 接口。

基于上述接口约定，我们定义了继承自模型基类的 NeuralNetwork，配合执行器来训练 FashionMNIST：

```
from torch.utils.data import DataLoader
from torch import nn
from torchvision import datasets
from torchvision.transforms import ToTensor
from mmengine.model import BaseModel
from mmengine.evaluator import BaseMetric
from mmengine import Runner


training_data = datasets.FashionMNIST(
    root="data",
    train=True,
    download=True,
    transform=ToTensor()
)

test_data = datasets.FashionMNIST(
    root="data",
    train=False,
    download=True,
    transform=ToTensor()
)

train_dataloader = DataLoader(dataset=training_data, batch_size=64)
test_dataloader = DataLoader(dataset=test_data, batch_size=64)


class NeuralNetwork(BaseModel):
    def __init__(self, data_preprocessor=None):
        super(NeuralNetwork, self).__init__(data_preprocessor)
        self.flatten = nn.Flatten()
        self.linear_relu_stack = nn.Sequential(
```

(下页继续)

(续上页)

```
            nn.Linear(28*28, 512),
            nn.ReLU(),
            nn.Linear(512, 512),
            nn.ReLU(),
            nn.Linear(512, 10),
        )
        self.loss = nn.CrossEntropyLoss()

    def forward(self, img, label, mode='tensor'):
        x = self.flatten(img)
        pred = self.linear_relu_stack(x)
        loss = self.loss(pred, label)
        if mode == 'loss':
            return dict(loss=loss)
        elif mode=='predict':
            return pred.argmax(1), loss.item()
        else:
            return pred


class FashionMnistMetric(BaseMetric):
    def process(self, data, preds) -> None:
        # data 参数为 Dataloader 返回的元组，即 (img, label)
        # predict 为模型 `predict` 模式下，返回的元组，分别为 `pred.argmax(1) 和 `loss``
        self.results.append(((data[1] == preds[0].cpu()).sum(), preds[1],␣
↪len(preds[0])))

    def compute_metrics(self, results):
        correct, loss, batch_size = zip(*results)
        test_loss, correct = sum(loss) / len(self.results), sum(correct) / sum(batch_
↪size)
        return dict(Accuracy=correct, Avg_loss=test_loss)


runner = Runner(
    model=NeuralNetwork(),
    work_dir='./work_dir',
    train_dataloader=train_dataloader,
    optim_wrapper=dict(optimizer=dict(type='SGD', lr=1e-3)),
    train_cfg=dict(by_epoch=True, max_epochs=5, val_interval=1),
    val_cfg=dict(fp16=True),
    val_dataloader=test_dataloader,
    val_evaluator=dict(metrics=FashionMnistMetric()))
runner.train()
```

在这个例子中，NeuralNetwork.forward 存在着以下跨模块的接口约定：

- 由于 train_dataloader 会返回一个 (img, label) 形式的元组，因此 forward 接口的前两个参数分别需要为 img 和 label。
- 由于 forward 在 predict 模式下会返回 (pred, loss) 形式的元组，因此 process 的 preds 参数应当同样为相同形式的元组。

相比于 Pytorch 官方示例，MMEngine 的代码更加简洁，记录的日志也更加丰富。

8.2 数据处理器（DataPreprocessor）

如果你的电脑配有 GPU（或其他能够加速训练的硬件，如 mps、ipu 等），并运行了上节的代码示例。你会发现 Pytorch 的示例是在 CPU 上运行的，而 MMEngine 的示例是在 GPU 上运行的。MMEngine 是在何时把数据和模型从 CPU 搬运到 GPU 的呢?

事实上，执行器会在构造阶段将模型搬运到指定设备，而数据则会在 train_step、val_step、test_step 中，被基础数据处理器（*BaseDataPreprocessor*）搬运到指定设备，进一步将处理好的数据传给模型。数据处理器作为模型基类的一个属性，会在模型基类的构造过程中被实例化。

为了体现数据处理器起到的作用，我们仍然以上一节训练 FashionMNIST 为例，实现了一个简易的数据处理器，用于搬运数据和归一化：

```
from torch.optim import SGD
from mmengine.model import BaseDataPreprocessor, BaseModel


class NeuralNetwork1(NeuralNetwork):

    def __init__(self, data_preprocessor):
        super().__init__(data_preprocessor=data_preprocessor)
        self.data_preprocessor = data_preprocessor

    def train_step(self, data, optimizer):
        img, label = self.data_preprocessor(data)
        loss = self(img, label, mode='loss')['loss'].sum()
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()
        return dict(loss=loss)

    def test_step(self, data):
        img, label = self.data_preprocessor(data)
        return self(img, label, mode='predict')
```

(下页继续)

(续上页)

```
    def val_step(self, data):
        img, label = self.data_preprocessor(data)
        return self(img, label, mode='predict')


class NormalizeDataPreprocessor(BaseDataPreprocessor):

    def forward(self, data, training=False):
        img, label = [item for item in data]
        img = (img - 127.5) / 127.5
        return img, label


model = NeuralNetwork1(data_preprocessor=NormalizeDataPreprocessor())
optimizer = SGD(model.parameters(), lr=0.01)
data = (torch.full((3, 28, 28), fill_value=127.5), torch.ones(3, 10))

model.train_step(data, optimizer)
model.val_step(data)
model.test_step(data)
```

上例中，我们实现了 `BaseModel.train_step`、`BaseModel.val_step` 和 `BaseModel.test_step` 的简化版。数据经 `NormalizeDataPreprocessor.forward` 归一化处理，解包后传给 `NeuralNetwork.forward`，进一步返回损失或者预测结果。如果想实现自定义的参数优化或预测逻辑，可以自行实现 `train_step`、`val_step` 和 `test_step`，具体例子可以参考：使用 *MMEngine* 训练生成对抗网络

注解： 上例中数据处理器的 training 参数用于区分训练、测试阶段不同的批增强策略，`train_step` 会传入 `training=True`，`test_step` 和 `val_step` 则会传入 `trainig=Fasle`。

注解： 通常情况下，我们要求 DataLoader 的 `data` 数据解包后（字典类型的被 **data 解包，元组列表类型被 *data 解包）能够直接传给模型的 `forward`。但是如果数据处理器修改了 data 的数据类型，则要求数据处理器的 `forward` 的返回值与模型 `forward` 的入参相匹配。

CHAPTER 9

模型精度评测（Evaluation）

在模型验证和模型测试中，通常需要对模型精度做定量评测。在 MMEngine 中实现了评测指标（Metric）和评测器（Evaluator）模块来完成这一功能：

- 评测指标：用于根据测试数据和模型预测结果，完成模型特定精度指标的计算。在 OpenMMLab 各算法库中提供了对应任务的常用评测指标，如 MMClassification 中提供了分类正确率指标（Accuracy） 用于计算分类模型的 Top-k 分类正确率。
- 评测器：是评测指标的上层模块，用于在数据输入评测指标前完成必要的格式转换，并提供分布式支持。在模型训练和测试中，评测器由执行器（*Runner*）自动构建。用户亦可根据需求手动创建评测器，进行离线评测。

9.1 在模型训练或测试中进行评测

9.1.1 评测指标配置

在基于 MMEngine 进行模型训练或测试时，执行器会自动构建评测器进行评测，用户只需要在配置文件中通过 `val_evaluator` 和 `test_evaluator` 2 个字段分别指定模型验证和测试阶段的评测指标即可。例如，用户在使用 MMClassification 训练分类模型时，希望在模型验证阶段评测 top-1 和 top-5 分类正确率，可以按以下方式配置：

```
val_evaluator = dict(type='Accuracy', top_k=(1, 5))  # 使用分类正确率评测指标
```

如果需要同时评测多个指标，也可以将 `val_evaluator` 或 `test_evaluator` 设置为一个列表，其中每一项为一个评测指标的配置信息。例如，在使用 MMDetection 训练全景分割模型时，希望在模型测试阶段同时

评测模型的目标检测（COCO AP/AR）和全景分割精度，可以按以下方式配置：

```
test_evaluator = [
    # 目标检测指标
    dict(
        type='COCOMetric',
        metric=['bbox', 'segm'],
        ann_file='annotations/instances_val2017.json',
    ),
    # 全景分割指标
    dict(
        type='CocoPanopticMetric',
        ann_file='annotations/panoptic_val2017.json',
        seg_prefix='annotations/panoptic_val2017',
    )
]
```

9.1.2 自定义评测指标

如果算法库中提供的常用评测指标无法满足需求，用户也可以增加自定义的评测指标。具体的方法可以参考评测指标和评测器设计。

9.2 使用离线结果进行评测

另一种常见的模型评测方式，是利用提前保存在文件中的模型预测结果进行离线评测。此时，由于不存在执行器，用户需要手动构建评测器，并调用评测器的相应接口完成评测。以下是一个离线评测示例：

```
from mmengine.evaluator import Evaluator
from mmengine.fileio import load

# 构建评测器。参数 `metrics` 为评测指标配置
evaluator = Evaluator(metrics=dict(type='Accuracy', top_k=(1, 5)))

# 从文件中读取测试数据。数据格式需要参考具使用的 metric。
data = load('test_data.pkl')

# 从文件中读取模型预测结果。该结果由待评测算法在测试数据集上推理得到。
# 数据格式需要参考具使用的 metric。
predictions = load('prediction.pkl')

# 调用评测器离线评测接口，得到评测结果
# chunk_size 表示每次处理的样本数量，可根据内存大小调整
```

（下页继续）

(续上页)

```
results = evaluator.offline_evaluate(data, predictions, chunk_size=128)
```

CHAPTER 10

优化器封装（OptimWrapper）

MMEngine 实现了优化器封装，为用户提供了统一的优化器访问接口。优化器封装支持不同的训练策略，包括混合精度训练、梯度累加和梯度截断。用户可以根据需求选择合适的训练策略。优化器封装还定义了一套标准的参数更新流程，用户可以基于这一套流程，实现同一套代码，不同训练策略的切换。

10.1 优化器封装 vs 优化器

这里我们分别基于 Pytorch 内置的优化器和 MMEngine 的优化器封装进行单精度训练、混合精度训练和梯度累加，对比二者实现上的区别。

10.1.1 训练模型

1.1 基于 Pytorch 的 SGD 优化器实现单精度训练

```
import torch
from torch.optim import SGD
import torch.nn as nn
import torch.nn.functional as F
****
inputs = [torch.zeros(10, 1, 1)] * 10
targets = [torch.ones(10, 1, 1)] * 10
model = nn.Linear(1, 1)
optimizer = SGD(model.parameters(), lr=0.01)
```

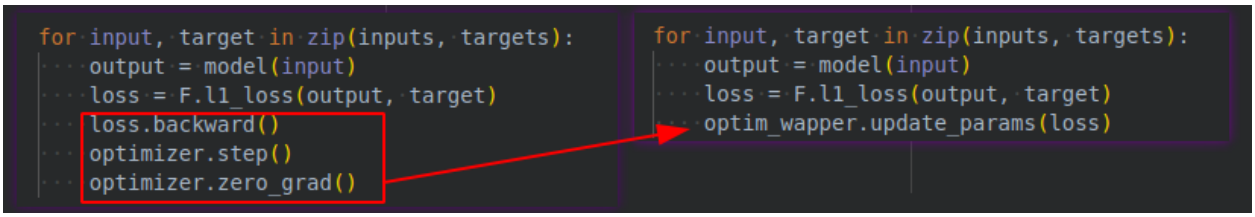
(下页继续)

(续上页)

```
optimizer.zero_grad()

for input, target in zip(inputs, targets):
    output = model(input)
    loss = F.l1_loss(output, target)
    loss.backward()
    optimizer.step()
    optimizer.zero_grad()
```

1.2 使用 MMEngine 的优化器封装实现单精度训练

```
from mmengine.optim import OptimWrapper

optim_wrapper = OptimWrapper(optimizer=optimizer)

for input, target in zip(inputs, targets):
    output = model(input)
    loss = F.l1_loss(output, target)
    optim_wrapper.update_params(loss)
```



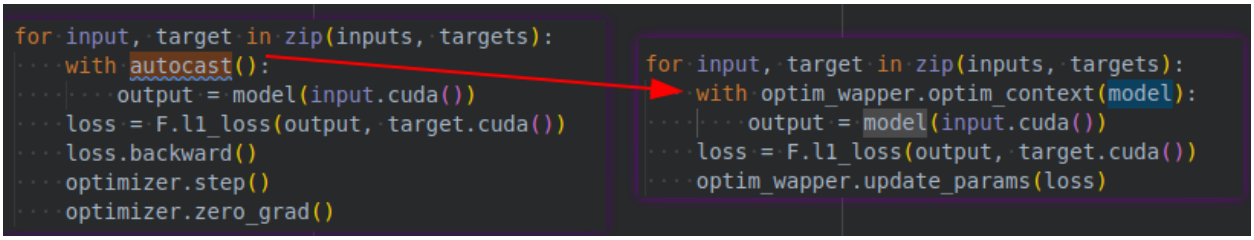
优化器封装的 update_params 实现了标准的梯度计算、参数更新和梯度清零流程，可以直接用来更新模型参数。

2.1 基于 Pytorch 的 SGD 优化器实现混合精度训练

```
from torch.cuda.amp import autocast

model = model.cuda()
inputs = [torch.zeros(10, 1, 1, 1)] * 10
targets = [torch.ones(10, 1, 1, 1)] * 10

for input, target in zip(inputs, targets):
    with autocast():
        output = model(input.cuda())
    loss = F.l1_loss(output, target.cuda())
    loss.backward()
    optimizer.step()
    optimizer.zero_grad()
```

2.2 基于 MMEngine 的优化器封装实现混合精度训练

```
from mmengine.optim import AmpOptimWrapper

optim_wrapper = AmpOptimWrapper(optimizer=optimizer)

for input, target in zip(inputs, targets):
    with optim_wrapper.optim_context(model):
        output = model(input.cuda())
    loss = F.l1_loss(output, target.cuda())
    optim_wrapper.update_params(loss)
```



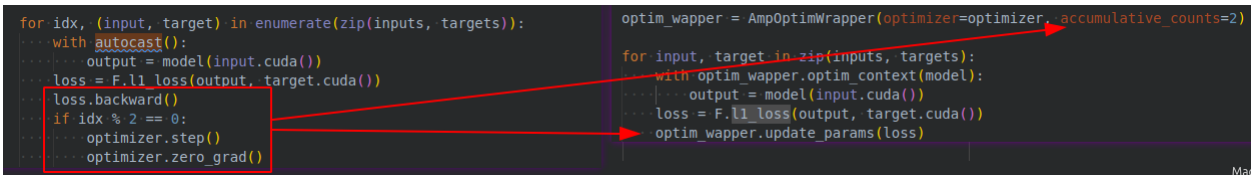
开合混合精度训练需要使用 `AmpOptimWrapper`，他的 optim_context 接口类似 `autocast`，会开启混合精度训练的上下文。除此之外他还能加速分布式训练时的梯度累加，这个我们会在下一个示例中介绍

3.1 基于 Pytorch 的 SGD 优化器实现混合精度训练和梯度累加

```
for idx, (input, target) in enumerate(zip(inputs, targets)):
    with autocast():
        output = model(input.cuda())
    loss = F.l1_loss(output, target.cuda())
    loss.backward()
    if idx % 2 == 0:
        optimizer.step()
        optimizer.zero_grad()
```

3.2 基于 MMEngine 的优化器封装实现混合精度训练和梯度累加

```
optim_wrapper = AmpOptimWrapper(optimizer=optimizer, accumulative_counts=2)

for input, target in zip(inputs, targets):
    with optim_wrapper.optim_context(model):
        output = model(input.cuda())
    loss = F.l1_loss(output, target.cuda())
    optim_wrapper.update_params(loss)
```



我们只需要配置 accumulative_counts 参数，并调用 update_params 接口就能实现梯度累加的功能。除此之外，分布式训练情况下，如果我们配置梯度累加的同时开启了 optim_wrapper 上下文，可以避免梯度累加阶段不必要的梯度同步。

优化器封装同样提供了更细粒度的接口，方便用户实现一些自定义的参数更新逻辑：

- backward：传入损失，用于计算参数梯度，。
- step：同 optimizer.step，用于更新参数。
- zero_grad：同 optimizer.zero_grad，用于参数的梯度。

我们可以使用上述接口实现和 Pytorch 优化器相同的参数更新逻辑：

```
for idx, (input, target) in enumerate(zip(inputs, targets)):
    optimizer.zero_grad()
    with optim_wrapper.optim_context(model):
        output = model(input.cuda())
    loss = F.l1_loss(output, target.cuda())
    optim_wrapper.backward(loss)
    if idx % 2 == 0:
        optim_wrapper.step()
        optim_wrapper.zero_grad()
```

10.1.2 获取学习率/动量：

优化器封装提供了 get_lr 和 get_momentum 接口用于获取优化器的一个参数组的学习率

```
import torch.nn as nn
from torch.optim import SGD

from mmengine.optim import OptimWrapper

model = nn.Linear(1, 1)
optimizer = SGD(model.parameters(), lr=0.01)
optim_wrapper = OptimWrapper(optimizer)

print(optimizer.param_groups[0]['lr'])  # -1.01
print(optimizer.param_groups[0]['momentum'])  # 0
print(optim_wrapper.get_lr())  # {'lr': [0.01]}
print(optim_wrapper.get_momentum())  # {'momentum': [0]}
```

```
0.01
0
{'lr': [0.01]}
{'momentum': [0]}
```

10.1.3 导出/加载状态字典

优化器封装和优化器一样，提供了 state_dict 和 load_state_dict 接口，用于导出/加载优化器状态，对于 AmpOptimWrapper，优化器封装还会额外导出混合精度训练相关的参数：

```
import torch.nn as nn
from torch.optim import SGD
from mmengine.optim import OptimWrapper, AmpOptimWrapper

model = nn.Linear(1, 1)
optimizer = SGD(model.parameters(), lr=0.01)

optim_wapper = OptimWrapper(optimizer=optimizer)
amp_optim_wapper = AmpOptimWrapper(optimizer=optimizer)

# 导出状态字典
optim_state_dict = optim_wapper.state_dict()
amp_optim_state_dict = amp_optim_wapper.state_dict()

print(optim_state_dict)
print(amp_optim_state_dict)
optim_wapper_new = OptimWrapper(optimizer=optimizer)
amp_optim_wapper_new = AmpOptimWrapper(optimizer=optimizer)

# 加载状态字典
amp_optim_wapper_new.load_state_dict(amp_optim_state_dict)
optim_wapper_new.load_state_dict(optim_state_dict)
```

```
{'state': {}, 'param_groups': [{'lr': 0.01, 'momentum': 0, 'dampening': 0, 'weight_
↪decay': 0, 'nesterov': False, 'maximize': False, 'foreach': None, 'params': [0, 1]}
↪]}
{'state': {}, 'param_groups': [{'lr': 0.01, 'momentum': 0, 'dampening': 0, 'weight_
↪decay': 0, 'nesterov': False, 'maximize': False, 'foreach': None, 'params': [0, 1]}
↪], 'loss_scaler': {'scale': 65536.0, 'growth_factor': 2.0, 'backoff_factor': 0.5,
↪'growth_interval': 2000, '_growth_tracker': 0}}
```

10.1.4 使用多个优化器

考虑到生成对抗网络之类的算法通常需要使用多个优化器来训练生成器和判别器，因此优化器封装提供了优化器封装的容器类：OptimWrapperDict 来管理多个优化器封装。OptimWrapperDict 以字典的形式存储优化器封装，并允许用户像字典一样访问、遍历其中的元素，即优化器封装实例。

与普通的优化器封装不同，OptimWrapperDict 没有实现 update_params、optim_context、backward、step 等方法，无法被直接用于训练模型。我们建议直接访问 OptimWrapperDict 管理的优化器实例，来实现参数更新逻辑。

你或许会好奇，既然 OptimWrapperDict 没有训练的功能，那为什么不直接使用 dict 来管理多个优化器。事实上，OptimWrapperDict 的核心功能是支持批量导出/加载所有优化器封装的状态字典；支持获取多个优化器封装的学习率、动量。如果没有 OptimWrapperDict，MMEngine 就需要在很多位置对优化器封装的类型做 if else 判断，以获取所有优化器封装的状态。

```
from torch.optim import SGD
import torch.nn as nn

from mmengine.optim import OptimWrapper, OptimWrapperDict

gen = nn.Linear(1, 1)
disc = nn.Linear(1, 1)
optimizer_gen = SGD(gen.parameters(), lr=0.01)
optimizer_disc = SGD(disc.parameters(), lr=0.01)

optim_wapper_gen = OptimWrapper(optimizer=optimizer_gen)
optim_wapper_disc = OptimWrapper(optimizer=optimizer_disc)
optim_dict = OptimWrapperDict(gen=optim_wapper_gen, disc=optim_wapper_disc)

print(optim_dict.get_lr())  # {'gen.lr': [0.01], 'disc.lr': [0.01]}
print(optim_dict.get_momentum())  # {'gen.momentum': [0], 'disc.momentum': [0]}
```

```
{'gen.lr': [0.01], 'disc.lr': [0.01]}
{'gen.momentum': [0], 'disc.momentum': [0]}
```

如上例所示，OptimWrapperDict 可以非常方便的导出所有优化器封装的学习率和动量，同样的，优化器封装也能够导出/加载所有优化器封装的状态字典。

10.2 在执行器中配置优化器封装

10.2.1 简单配置

优化器封装需要接受 `optimizer` 参数，因此我们首先需要为优化器封装配置 `optimizer`。MMEngine 会自动将 PyTorch 中的所有优化器都添加进 `OPTIMIZERS` 注册表中，用户可以用字典的形式来指定优化器，所有支持的优化器见 PyTorch 优化器列表。

以配置一个 SGD 优化器封装为例：

```
optimizer = dict(type='SGD', lr=0.01, momentum=0.9, weight_decay=0.0001)
optim_wrapper = dict(type='OptimWrapper', optimizer=optimizer)
```

这样我们就配置好了一个优化器类型为 SGD 的优化器封装，学习率、动量等参数如配置所示。考虑到 `OptimWrapper` 为标准的单精度训练，因此我们也可以不配置 `type` 字段：

```
optimizer = dict(type='SGD', lr=0.01, momentum=0.9, weight_decay=0.0001)
optim_wrapper = dict(optimizer=optimizer)
```

要想开启混合精度训练和梯度累加，需要将 `type` 切换成 `AmpOptimWrapper`，并指定 `accumulative_counts` 参数

```
optimizer = dict(type='SGD', lr=0.01, momentum=0.9, weight_decay=0.0001)
optim_wrapper = dict(type='AmpOptimWrapper', optimizer=optimizer, accumulative_
↪counts=2)
```

10.2.2 进阶配置

PyTorch 的优化器支持对模型中的不同参数设置不同的超参数，例如对一个分类模型的骨干（backbone）和分类头（head）设置不同的学习率：

```
from torch.optim import SGD
import torch.nn as nn

model = nn.ModuleDict(dict(backbone=nn.Linear(1, 1), head=nn.Linear(1, 1)))
optimizer = SGD([{'params': model.backbone.parameters()},
     {'params': model.head.parameters(), 'lr': 1e-3}],
    lr=0.01,
    momentum=0.9)
```

上面的例子中，模型的骨干部分使用了 0.01 学习率，而模型的头部则使用了 1e-3 学习率。用户可以将模型的不同部分参数和对应的超参组成一个字典的列表传给优化器，来实现对模型优化的细粒度调整。

在 MMEngine 中，我们通过优化器封装构造器（optimizer wrapper constructor），让用户能够直接通过设置优化器封装配置文件中的 `paramwise_cfg` 字段而非修改代码来实现对模型的不同部分设置不同的超参。

为不同类型的参数设置不同的超参系数

MMEngine 提供的默认优化器封装构造器支持对模型中不同类型的参数设置不同的超参系数。例如，我们可以在 `paramwise_cfg` 中设置 `norm_decay_mult=0`，从而将正则化层（normalization layer）的权重（weight）和偏置（bias）的权值衰减系数（weight decay）设置为 0，来实现 Bag of Tricks 论文中提到的不对正则化层进行权值衰减的技巧。

具体示例如下，我们将 `ToyModel` 中所有正则化层（`head.bn`）的的权重衰减系数设置为 0：

```
from mmengine.optim import build_optim_wrapper
from collections import OrderedDict

class ToyModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.backbone = nn.ModuleDict(
            dict(layer0=nn.Linear(1, 1), layer1=nn.Linear(1, 1)))
        self.head = nn.Sequential(
            OrderedDict(
                linear=nn.Linear(1, 1),
                bn=nn.BatchNorm1d(1)))


optim_wrapper = dict(
    optimizer=dict(type='SGD', lr=0.01, weight_decay=0.0001),
    paramwise_cfg=dict(norm_decay_mult=0))
optimizer = build_optim_wrapper(ToyModel(), optim_wrapper)
```

```
08/23 22:02:43 - mmengine - INFO - paramwise_options -- backbone.layer0.bias:lr=0.01
08/23 22:02:43 - mmengine - INFO - paramwise_options -- backbone.layer0.bias:weight_
↪decay=0.0001
08/23 22:02:43 - mmengine - INFO - paramwise_options -- backbone.layer1.bias:lr=0.01
08/23 22:02:43 - mmengine - INFO - paramwise_options -- backbone.layer1.bias:weight_
↪decay=0.0001
08/23 22:02:43 - mmengine - INFO - paramwise_options -- head.linear.bias:lr=0.01
08/23 22:02:43 - mmengine - INFO - paramwise_options -- head.linear.bias:weight_
↪decay=0.0001
08/23 22:02:43 - mmengine - INFO - paramwise_options -- head.bn.weight:weight_decay=0.
↪0
08/23 22:02:43 - mmengine - INFO - paramwise_options -- head.bn.bias:weight_decay=0.0
```

除了可以对正则化层的权重衰减进行配置外，MMEngine 的默认优化器封装构造器的 `paramwise_cfg` 还支

持对更多不同类型的参数设置超参系数，支持的配置如下：

`lr_mult`：所有参数的学习率系数

`decay_mult`：所有参数的衰减系数

`bias_lr_mult`：偏置的学习率系数（不包括正则化层的偏置以及可变形卷积的 offset），默认值为 1

`bias_decay_mult`：偏置的权值衰减系数（不包括正则化层的偏置以及可变形卷积的 offset），默认值为 1

`norm_decay_mult`：正则化层权重和偏置的权值衰减系数，默认值为 1

`dwconv_decay_mult`：Depth-wise 卷积的权值衰减系数，默认值为 1

`bypass_duplicate`：是否跳过重复的参数，默认为 `False`

`dcn_offset_lr_mult`：可变形卷积（Deformable Convolution）的学习率系数，默认值为 1

为模型不同部分的参数设置不同的超参系数

此外，与上文 PyTorch 的示例一样，在 MMEngine 中我们也同样可以对模型中的任意模块设置不同的超参，只需要在 `paramwise_cfg` 中设置 `custom_keys` 即可。

例如我们想将 `backbone.layer0` 所有参数的学习率设置为 0，衰减系数设置为 0，`backbone` 其余子模块的学习率设置为 1；`head` 所欲参数的学习率设置为 0.01，可以这样配置：

```
optim_wrapper = dict(
    optimizer=dict(type='SGD', lr=0.01, weight_decay=0.0001),
    paramwise_cfg=dict(
        custom_keys={
            'backbone.layer0': dict(lr_mult=0, decay_mult=0),
            'backbone': dict(lr_mult=1),
            'head': dict(lr_mult=0.1)
        }))
optimizer = build_optim_wrapper(ToyModel(), optim_wrapper)
```

```
08/23 22:02:43 - mmengine - INFO - paramwise_options -- backbone.layer0.weight:lr=0.0
08/23 22:02:43 - mmengine - INFO - paramwise_options -- backbone.layer0.weight:weight_
↪decay=0.0
08/23 22:02:43 - mmengine - INFO - paramwise_options -- backbone.layer0.weight:lr_
↪mult=0
08/23 22:02:43 - mmengine - INFO - paramwise_options -- backbone.layer0.weight:decay_
↪mult=0
08/23 22:02:43 - mmengine - INFO - paramwise_options -- backbone.layer0.bias:lr=0.0
08/23 22:02:43 - mmengine - INFO - paramwise_options -- backbone.layer0.bias:weight_
↪decay=0.0
08/23 22:02:43 - mmengine - INFO - paramwise_options -- backbone.layer0.bias:lr_mult=0
08/23 22:02:43 - mmengine - INFO - paramwise_options -- backbone.layer0.bias:decay_
↪mult=0
```

(下页继续)

(续上页)

```
08/23 22:02:43 - mmengine - INFO - paramwise_options -- backbone.layer1.weight:lr=0.01
08/23 22:02:43 - mmengine - INFO - paramwise_options -- backbone.layer1.weight:weight_
↪decay=0.0001
08/23 22:02:43 - mmengine - INFO - paramwise_options -- backbone.layer1.weight:lr_
↪mult=1
08/23 22:02:43 - mmengine - INFO - paramwise_options -- backbone.layer1.bias:lr=0.01
08/23 22:02:43 - mmengine - INFO - paramwise_options -- backbone.layer1.bias:weight_
↪decay=0.0001
08/23 22:02:43 - mmengine - INFO - paramwise_options -- backbone.layer1.bias:lr_mult=1
08/23 22:02:43 - mmengine - INFO - paramwise_options -- head.linear.weight:lr=0.001
08/23 22:02:43 - mmengine - INFO - paramwise_options -- head.linear.weight:weight_
↪decay=0.0001
08/23 22:02:43 - mmengine - INFO - paramwise_options -- head.linear.weight:lr_mult=0.1
08/23 22:02:43 - mmengine - INFO - paramwise_options -- head.linear.bias:lr=0.001
08/23 22:02:43 - mmengine - INFO - paramwise_options -- head.linear.bias:weight_
↪decay=0.0001
08/23 22:02:43 - mmengine - INFO - paramwise_options -- head.linear.bias:lr_mult=0.1
08/23 22:02:43 - mmengine - INFO - paramwise_options -- head.bn.weight:lr=0.001
08/23 22:02:43 - mmengine - INFO - paramwise_options -- head.bn.weight:weight_decay=0.
↪0001
08/23 22:02:43 - mmengine - INFO - paramwise_options -- head.bn.weight:lr_mult=0.1
08/23 22:02:43 - mmengine - INFO - paramwise_options -- head.bn.bias:lr=0.001
08/23 22:02:43 - mmengine - INFO - paramwise_options -- head.bn.bias:weight_decay=0.
↪0001
08/23 22:02:43 - mmengine - INFO - paramwise_options -- head.bn.bias:lr_mult=0.1
```

上例中，模型的状态字典的 key 如下：

```
for name, val in ToyModel().named_parameters():
    print(name)
```

```
backbone.layer0.weight
backbone.layer0.bias
backbone.layer1.weight
backbone.layer1.bias
head.linear.weight
head.linear.bias
head.bn.weight
head.bn.bias
```

custom_keys 中每一个字段的含义如下：

1. `'backbone': dict(lr_mult=1)`：将名字前缀为 backbone 的参数的学习率设置为 1
2. `'backbone.layer0': dict(lr_mult=0, decay_mult=0)`：将名字前缀为 backbone.layer0

的参数学习率设置为 0，衰减系数设置为 0，该配置优先级比第一条高

3. `'head': dict(lr_mult=0.1)`：将名字前缀为 `head` 的参数的学习率设置为 0.1

10.2.3 自定义优化器构造策略

与 MMEngine 中的其他模块一样，优化器封装构造器也同样由注册表管理。我们可以通过实现自定义的优化器封装构造器来实现自定义的超参设置策略。

例如，我们想实现一个叫做 `LayerDecayOptimWrapperConstructor` 的优化器封装构造器，能够对模型不同深度的层自动设置递减的学习率：

```
from mmengine.optim import DefaultOptimWrapperConstructor
from mmengine.registry import OPTIM_WRAPPER_CONSTRUCTORS
from mmengine.logging import print_log


@OPTIM_WRAPPER_CONSTRUCTORS.register_module(force=True)
class LayerDecayOptimWrapperConstructor(DefaultOptimWrapperConstructor):

    def __init__(self, optim_wrapper_cfg, paramwise_cfg=None):
        super().__init__(optim_wrapper_cfg, paramwise_cfg=None)
        self.decay_factor = paramwise_cfg.get('decay_factor', 0.5)

        super().__init__(optim_wrapper_cfg, paramwise_cfg)

    def add_params(self, params, module, prefix='' ,lr=None):
        if lr is None:
            lr = self.base_lr

        for name, param in module.named_parameters(recurse=False):
            param_group = dict()
            param_group['params'] = [param]
            param_group['lr'] = lr
            params.append(param_group)
            full_name = f'{prefix}.{name}' if prefix else name
            print_log(f'{full_name} : lr={lr}', logger='current')

        for name, module in module.named_children():
            chiled_prefix = f'{prefix}.{name}' if prefix else name
            self.add_params(
                params, module, chiled_prefix, lr=lr * self.decay_factor)


class ToyModel(nn.Module):
```

(下页继续)

(续上页)

```
    def __init__(self) -> None:
        super().__init__()
        self.layer = nn.ModuleDict(dict(linear=nn.Linear(1, 1)))
        self.linear = nn.Linear(1, 1)


model = ToyModel()

optim_wrapper = dict(
    optimizer=dict(type='SGD', lr=0.01, weight_decay=0.0001),
    paramwise_cfg=dict(decay_factor=0.5),
    constructor='LayerDecayOptimWrapperConstructor')

optimizer = build_optim_wrapper(model, optim_wrapper)
```

```
08/23 22:20:26 - mmengine - INFO - layer.linear.weight : lr=0.0025
08/23 22:20:26 - mmengine - INFO - layer.linear.bias : lr=0.0025
08/23 22:20:26 - mmengine - INFO - linear.weight : lr=0.005
08/23 22:20:26 - mmengine - INFO - linear.bias : lr=0.005
```

add_params 被第一次调用时，params 参数为空列表（list），module 为模型（model）。详细的重载规则参考优化器封装构造器文档。

类似地，如果想构造多个优化器，也需要实现自定义的构造器：

```
@OPTIM_WRAPPER_CONSTRUCTORS.register_module()
class MultipleOptimiWrapperConstructor:
    ...
```

10.2.4 在训练过程中调整超参

优化器中的超参数在构造时只能设置为一个定值，仅仅使用优化器封装，并不能在训练过程中调整学习率等参数。在 MMEngine 中，我们实现了参数调度器（Parameter Scheduler），以便能够在训练过程中调整参数。关于参数调度器的用法请见优化器参数调整策略

CHAPTER 11

优化器参数调整策略（Parameter Scheduler）

在模型训练过程中，我们往往不是采用固定的优化参数，例如学习率等，会随着训练轮数的增加进行调整。最简单常见的学习率调整策略就是阶梯式下降，例如每隔一段时间将学习率降低为原来的几分之一。PyTorch 中有学习率调度器 LRScheduler 来对各种不同的学习率调整方式进行抽象，但支持仍然比较有限，在 MMEngine 中，我们对其进行了拓展，实现了更通用的参数调度器 `mmengine.optim.scheduler`，可以对学习率、动量等优化器相关的参数进行调整，并且支持多个调度器进行组合，应用更复杂的调度策略。

11.1 参数调度器的使用

这里我们先简单介绍一下如何使用 PyTorch 内置的学习率调度器来进行学习率的调整。下面是参考 PyTorch 官方文档 实现的一个例子，我们构造一个 ExponentialLR，并且在每个 epoch 结束后调用 `scheduler.step()`，实现了随 epoch 指数下降的学习率调整策略。

```
import torch
from torch.optim import SGD
from torch.optim.lr_scheduler import ExponentialLR

model = torch.nn.Linear(1, 1)
dataset = [torch.randn((1, 1, 1)) for _ in range(20)]
optimizer = SGD(model, 0.1)
scheduler = ExponentialLR(optimizer, gamma=0.9)

for epoch in range(10):
    for data in dataset:
```

(下页继续)

(续上页)

```
        optimizer.zero_grad()
        output = model(data)
        loss = 1 - output
        loss.backward()
        optimizer.step()
    scheduler.step()
```

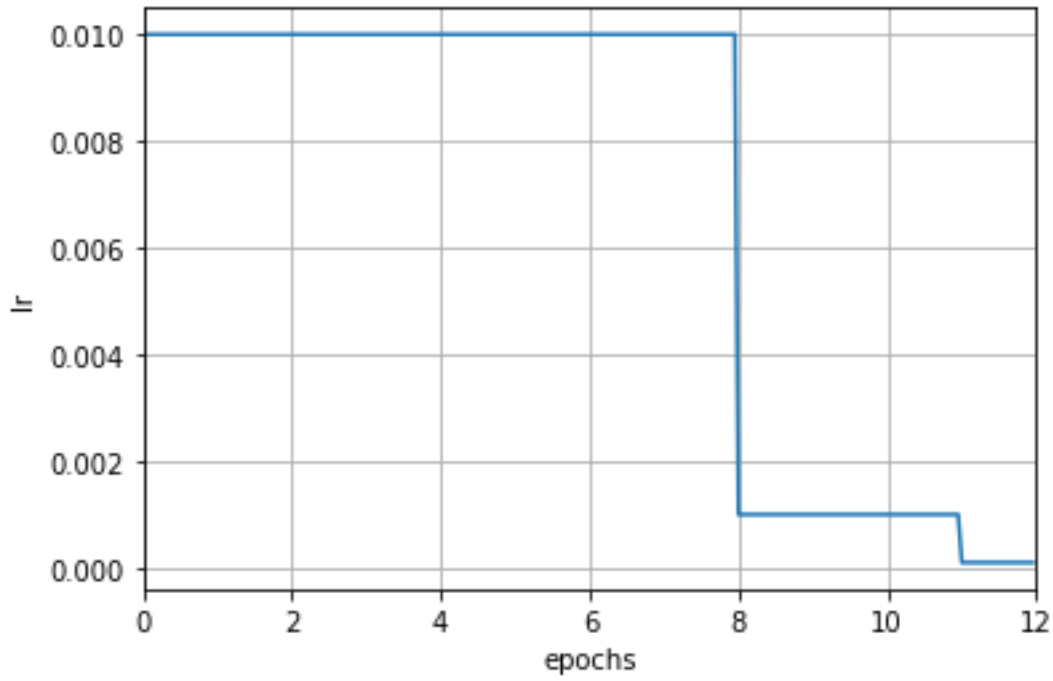
在 mmengine.optim.scheduler 中，我们支持大部分 PyTorch 中的学习率调度器，例如 ExponentialLR，LinearLR，StepLR，MultiStepLR 等，使用方式也基本一致，所有支持的调度器见调度器接口文档。同时增加了对动量的调整，在类名中将 LR 替换成 Momentum 即可，例如 ExponentialMomentum，LinearMomentum。更进一步地，我们实现了通用的参数调度器 ParamScheduler，用于调整优化器的中的其他参数，包括 weight_decay 等。这个特性可以很方便地配置一些新算法中复杂的调整策略。

和 PyTorch 文档中所给示例不同，MMEngine 中通常不需要手动来实现训练循环以及调用 optimizer.step()，而是在执行器（Runner）中对训练流程进行自动管理，同时通过 ParamSchedulerHook 来控制参数调度器的执行。

11.1.1 使用单一的学习率调度器

如果整个训练过程只需要使用一个学习率调度器，那么和 PyTorch 自带的学习率调度器没有差异。

```
from mmengine.optim.scheduler import MultiStepLR

optimizer = SGD(model.parameters(), lr=0.01, momentum=0.9)
scheduler = MultiStepLR(optimizer, milestones=[8, 11], gamma=0.1)
```



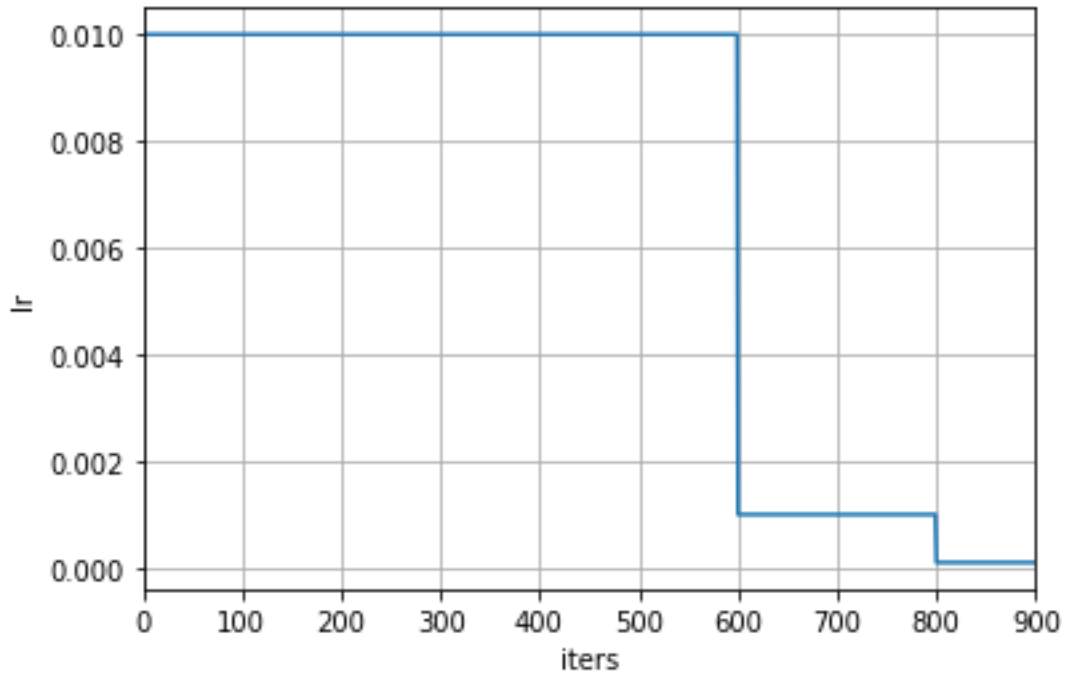
如果配合注册器和配置文件使用的话，我们可以设置配置文件中的 `scheduler` 字段来指定优化器, 执行器（Runner）会根据此字段以及执行器中的优化器自动构建学习率调度器：

```
scheduler = dict(type='MultiStepLR', by_epoch=True, milestones=[8, 11], gamma=0.1)
```

注意这里增加了初始化参数 `by_epoch`，控制的是学习率调整频率，当其为 True 时表示按轮次（epoch）调整，为 False 时表示按迭代次数（iteration）调整，默认值为 True。在上面的例子中，表示按照轮次进行调整，此时其他参数的单位均为 epoch，例如 `milestones` 中的 [8, 11] 表示第 8 和 11 轮次结束时，学习率将会被调整为上一轮次的 0.1 倍。

当修改了学习率调整频率后，调度器中与计数相关设置的含义也会相应被改变。当 `by_epoch=True` 时，milestones 中的数字表示在哪些轮次进行学习率衰减，而当 `by_epoch=False` 时则表示在进行到第几次迭代时进行学习率衰减。下面是一个按照迭代次数进行调整的例子，在第 600 和 800 次迭代结束时，学习率将会被调整为原来的 0.1 倍。

```
scheduler = dict(type='MultiStepLR', by_epoch=False, milestones=[600, 800], gamma=0.1)
```



若用户希望在配置调度器时按轮次填写参数的同时使用基于迭代的更新频率，MMEngine 的调度器也提供了自动换算的方式。用户可以调用 build_iter_from_epoch 方法，并提供每个训练轮次的迭代次数，即可构造按迭代次数更新的调度器对象：

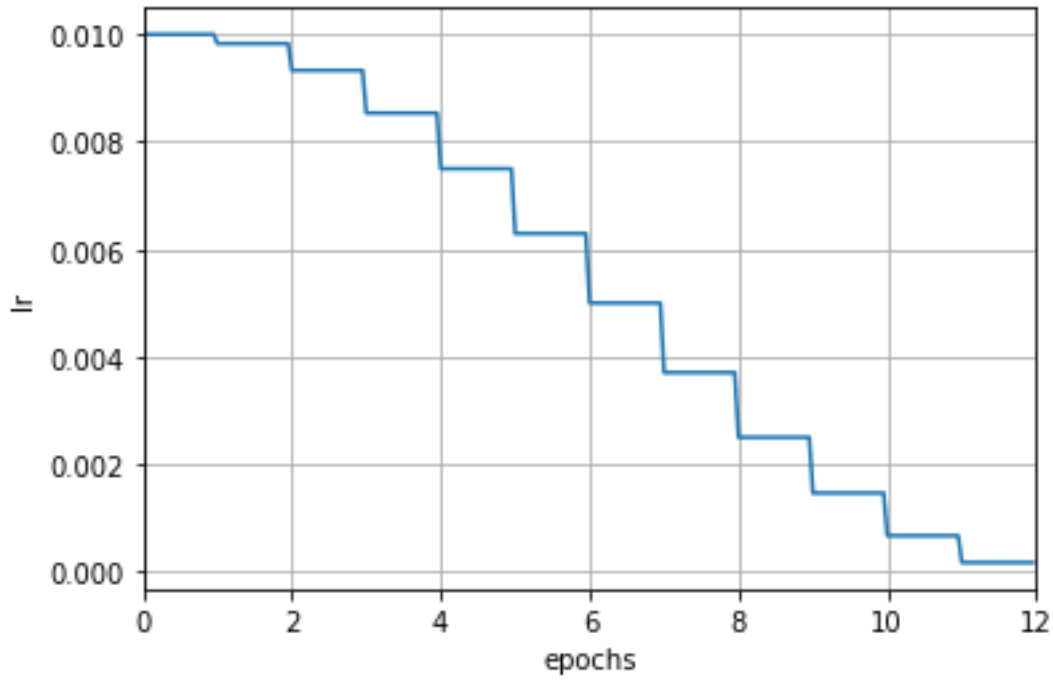
```
epoch_length = len(train_dataloader)
scheduler = MultiStepLR.build_iter_from_epoch(optimizer, milestones=[8, 11], gamma=0.
↪1, epoch_length=epoch_length)
```

如果使用配置文件构建调度器，只需要在配置中加入 `convert_to_iter_based=True`，执行器会自动调用 `build_iter_from_epoch` 将基于轮次的配置文件转换为基于迭代次数的调度器对象：

```
scheduler = dict(type='MultiStepLR', by_epoch=True, milestones=[8, 11], gamma=0.1,␣
↪convert_to_iter_based=True)
```

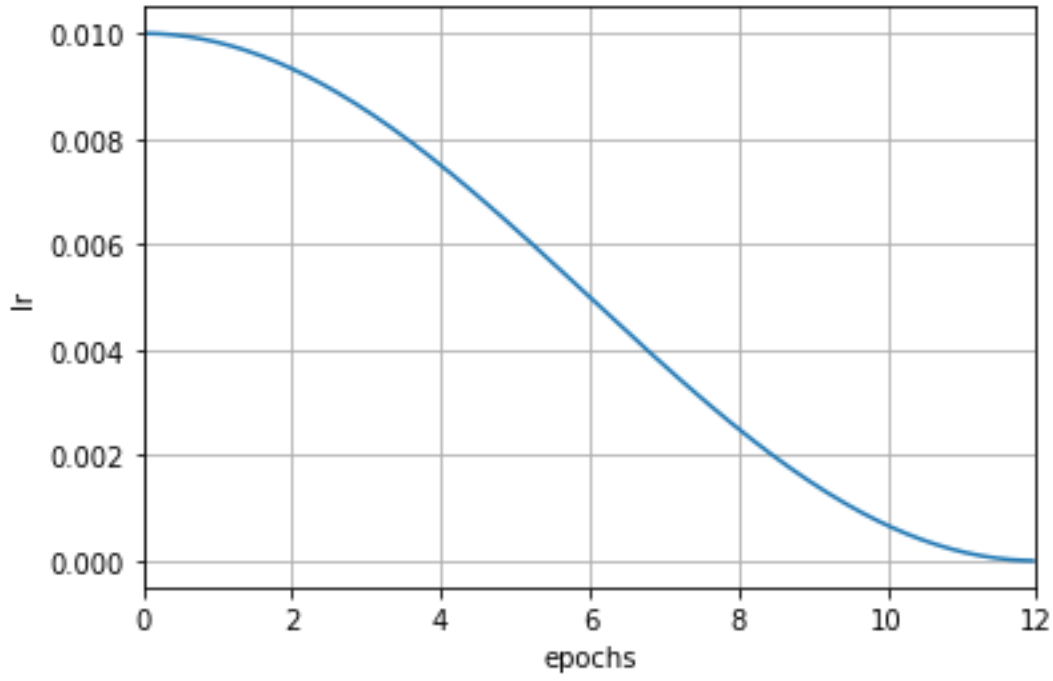
为了能直观感受这两种模式的区别，我们中这里再举一个例子。下面是一个按轮次更新的余弦退火（CosineAnnealing）学习率调度器，学习率仅在每个轮次结束后被修改：

```
scheduler = dict(type='CosineAnnealingLR', by_epoch=True, T_max=12)
```



而在使用自动换算后，学习率会在每次迭代后被修改。从下图可以看出，学习率的变化更为平滑。

```
scheduler = dict(type='CosineAnnealingLR', by_epoch=True, T_max=12, convert_to_iter_
↪based=True)
```

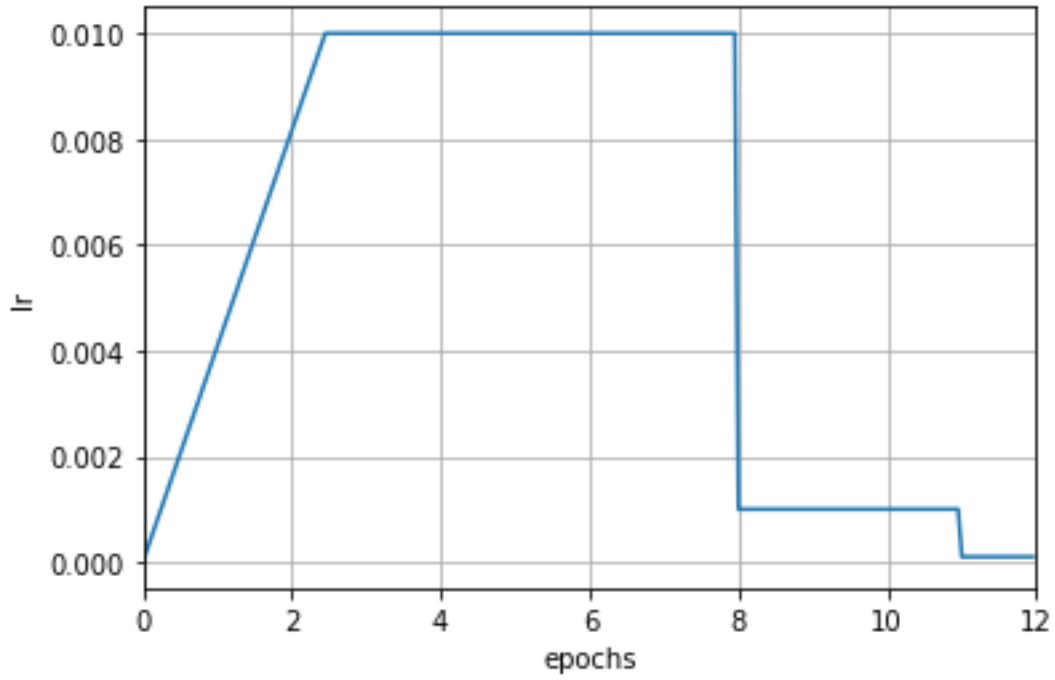


11.1.2 组合多个学习率调度器（以学习率预热为例）

有些算法在训练过程中，并不是自始至终按照某个调度策略进行学习率调整的。最常见的例子是学习率预热，比如在训练刚开始的若干迭代次数使用线性的调整策略将学习率从一个较小的值增长到正常，然后按照另外的调整策略进行正常训练。

MMEngine 支持组合多个调度器一起使用，只需将配置文件中的 `scheduler` 字段修改为一组调度器配置的列表，SchedulerStepHook 可以自动对调度器列表进行处理。下面的例子便实现了学习率预热。

```
scheduler = [
    # 线性学习率预热调度器
    dict(type='LinearLR',
         start_factor=0.001,
         by_epoch=False,  # 按迭代更新学习率
         begin=0,
         end=50),  # 预热前 50 次迭代
    # 主学习率调度器
    dict(type='MultiStepLR',
         by_epoch=True,  # 按轮次更新学习率
         milestones=[8, 11],
         gamma=0.1)
]
```

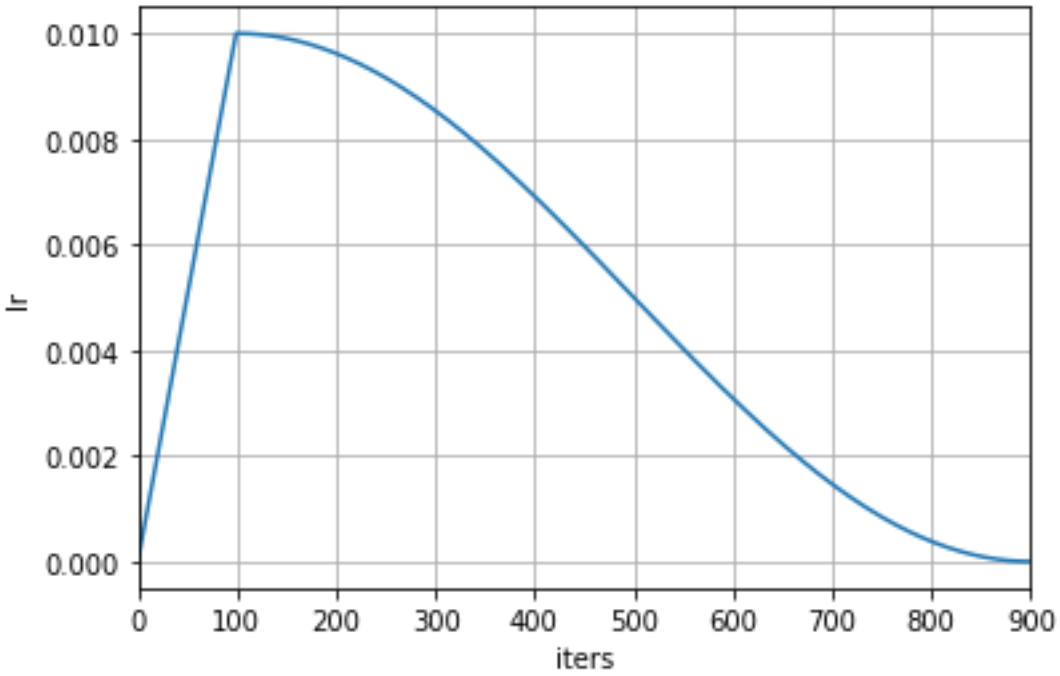


注意这里增加了 `begin` 和 `end` 参数，这两个参数指定了调度器的**生效区间**。生效区间通常只在多个调度器组合时才需要去设置，使用单个调度器时可以忽略。当指定了 `begin` 和 `end` 参数时，表示该调度器只在 [begin, end) 区间内生效，其单位是由 `by_epoch` 参数决定。上述例子中预热阶段 `LinearLR` 的 `by_epoch`

为 False，表示该调度器只在前 50 次迭代生效，超过 50 次迭代后此调度器不再生效，由第二个调度器来控制学习率，即 MultiStepLR。在组合不同调度器时，各调度器的 by_epoch 参数不必相同。

这里再举一个例子：

```
scheduler = [
    # 在 [0, 100) 迭代时使用线性学习率
    dict(type='LinearLR',
         start_factor=0.001,
         by_epoch=False,
         begin=0,
         end=100),
    # 在 [100, 900) 迭代时使用余弦学习率
    dict(type='CosineAnnealingLR',
         T_max=800,
         by_epoch=False,
         begin=100,
         end=900)
]
```



上述例子表示在训练的前 100 次迭代时使用线性的学习率预热，然后在第 100 到第 900 次迭代时使用周期为 800 的余弦退火学习率调度器使学习率按照余弦函数逐渐下降为 0 。

我们可以组合任意多个调度器，既可以使用 MMEngine 中已经支持的调度器，也可以实现自定义的调度器。如果相邻两个调度器的生效区间没有紧邻，而是有一段区间没有被覆盖，那么这段区间的学习率维持不变。而如果两个调度器的生效区间发生了重叠，则对多组调度器叠加使用，学习率的调整会按照调度器配置文件中的顺序触发（行为与 PyTorch 中 ChainedScheduler 一致）。在一般情况下，我们推荐用户在训练的不同

阶段使用不同的学习率调度策略来避免调度器的生效区间发生重叠。如果确实需要将两个调度器叠加使用，则需要十分小心，我们推荐使用学习率可视化工具来可视化叠加后的学习率，以避免学习率的调整与预期不符。

11.2 如何调整其他参数

11.2.1 动量

和学习率一样，动量也是优化器参数组中一组可以调度的参数。动量调度器（momentum scheduler）的使用方法和学习率调度器完全一样。同样也只需要将动量调度器的配置添加进配置文件中的 `scheduler` 字段的列表中即可。

示例：

```
scheduler = [
    # the lr scheduler
    dict(type='LinearLR', ...),
    # the momentum scheduler
    dict(type='LinearMomentum',
         start_factor=0.001,
         by_epoch=False,
         begin=0,
         end=1000)
]
```

11.2.2 通用的参数调度器

MMEngine 还提供了一组通用的参数调度器用于调度优化器的 `param_groups` 中的其他参数，将学习率调度器类名中的 `LR` 改为 `Param` 即可，例如 `LinearParamScheduler`。用户可以通过设置参数调度器的 `param_name` 变量来选择想要调度的参数。

下面是一个通过自定义参数名来调度的例子：

```
scheduler = [
    dict(type='LinearParamScheduler',
         param_name='lr',  # 调度 `optimizer.param_groups` 中名为 'lr' 的变量
         start_factor=0.001,
         by_epoch=False,
         begin=0,
         end=1000)
]
```

这里设置的参数名是 `lr`，因此这个调度器的作用等同于直接使用学习率调度器 `LinearLRScheduler`。

除了动量之外，用户也可以对 optimizer.param_groups 中的其他参数名进行调度，可调度的参数取决于所使用的优化器。例如，当使用带 weight_decay 的 SGD 优化器时，可以按照以下示例对调整 weight_decay：

```
scheduler = [
    dict(type='LinearParamScheduler',
         param_name='weight_decay',  # 调度 `optimizer.param_groups` 中名为 'weight_
↪decay' 的变量
         start_factor=0.001,
         by_epoch=False,
         begin=0,
         end=1000)
]
```

CHAPTER 12

数据变换 (Data Transform)

在 OpenMMLab 算法库中，数据集的构建和数据的准备是相互解耦的。通常，数据集的构建只对数据集进行解析，记录每个样本的基本信息；而数据的准备则是通过一系列的数据变换，根据样本的基本信息进行数据加载、预处理、格式化等操作。

12.1 使用数据变换类

在 MMEngine 中，我们使用各种可调用的数据变换类来进行数据的操作。这些数据变换类可以接受若干配置参数进行实例化，之后通过调用的方式对输入的数据字典进行处理。同时，我们约定所有数据变换都接受一个字典作为输入，并将处理后的数据输出为一个字典。一个简单的例子如下：

注解： MMEngine 中仅约定了数据变换类的规范，常用的数据变换类实现及基类都在 MMCV 中，因此在本篇教程需要提前安装好 MMCV，参见 MMCV 的安装教程。

```
>>> import numpy as np
>>> from mmcv.transforms import Resize
>>>
>>> transform = Resize(scale=(224, 224))
>>> data_dict = {'img': np.random.rand(256, 256, 3)}
>>> data_dict = transform(data_dict)
>>> print(data_dict['img'].shape)
(224, 224, 3)
```

12.2 在配置文件中使用

在配置文件中，我们将一系列数据变换组合成为一个列表，称为数据流水线（Data Pipeline），传给数据集的 `pipeline` 参数。通常数据流水线由以下几个部分组成：

1. 数据加载，通常使用 `LoadImageFromFile`
2. 标签加载，通常使用 `LoadAnnotations`
3. 数据处理及增强，例如 `RandomResize`
4. 数据格式化，根据任务不同，在各个仓库使用自己的变换操作，通常名为 `PackXXXInputs`，其中 XXX 是任务的名称，如分类任务中的 `PackClsInputs`。

以分类任务为例，我们在下图展示了一个典型的数据流水线。对每个样本，数据集中保存的基本信息是一个如图中最左侧所示的字典，之后每经过一个由蓝色块代表的数据变换操作，数据字典中都会加入新的字段（标记为绿色）或更新现有的字段（标记为橙色）。

如果我们希望在测试中使用上述数据流水线，则配置文件如下所示：

```
test_dataloader = dict(
    batch_size=32,
    dataset=dict(
        type='ImageNet',
        data_root='data/imagenet',
        pipeline = [
            dict(type='LoadImageFromFile'),
            dict(type='Resize', size=256, keep_ratio=True),
            dict(type='CenterCrop', crop_size=224),
            dict(type='PackClsInputs'),
        ]
    )
)
```

12.3 常用的数据变换类

按照功能，常用的数据变换类可以大致分为数据加载、数据预处理与增强、数据格式化。我们在 MMCV 中提供了一系列常用的数据变换类：

12.3.1 数据加载

为了支持大规模数据集的加载，通常在数据集初始化时不加载数据，只加载相应的路径。因此需要在数据流水线中进行具体数据的加载。

12.3.2 数据预处理及增强

数据预处理和增强通常是对图像本身进行变换，如裁剪、填充、缩放等。

12.3.3 数据格式化

数据格式化操作通常是对数据进行的类型转换。

12.4 自定义数据变换类

要实现一个新的数据变换类，需要继承 `BaseTransform`，并实现 `transform` 方法。这里，我们使用一个简单的翻转变换（`MyFlip`）作为示例：

```
import random
import mmcv
from mmcv.transforms import BaseTransform, TRANSFORMS

@TRANSFORMS.register_module()
class MyFlip(BaseTransform):
    def __init__(self, direction: str):
        super().__init__()
        self.direction = direction

    def transform(self, results: dict) -> dict:
        img = results['img']
        results['img'] = mmcv.imflip(img, direction=self.direction)
        return results
```

从而，我们可以实例化一个 `MyFlip` 对象，并将之作为一个可调用对象，来处理我们的数据字典。

```
import numpy as np

transform = MyFlip(direction='horizontal')
data_dict = {'img': np.random.rand(224, 224, 3)}
data_dict = transform(data_dict)
processed_img = data_dict['img']
```

又或者，在配置文件的 pipeline 中使用 `MyFlip` 变换

```
pipeline = [
    ...
    dict(type='MyFlip', direction='horizontal'),
    ...
]
```

需要注意的是，如需在配置文件中使用，需要保证 `MyFlip` 类所在的文件在运行时能够被导入。

CHAPTER 13

数据集基类（BaseDataset）

13.1 基本介绍

算法库中的数据集类负责在训练/测试过程中为模型提供输入数据，OpenMMLab 下各个算法库中的数据集有一些共同的特点和需求，比如需要高效的内部数据存储格式，需要支持数据集拼接、数据集重复采样等功能。

因此 **MMEngine** 实现了一个数据集基类（BaseDataset）并定义了一些基本接口，且基于这套接口实现了一些数据集包装（DatasetWrapper）。OpenMMLab 算法库中的大部分数据集都会满足这套数据集基类定义的接口，并使用统一的数据集包装。

数据集基类的基本功能是加载数据集信息，这里我们将数据集信息分成两类，一种是元信息 (meta information)，代表数据集自身相关的信息，有时需要被模型或其他外部组件获取，比如在图像分类任务中，数据集的元信息一般包含类别信息 `classes`，因为分类模型 `model` 一般需要记录数据集的类别信息；另一种为数据信息 (data information)，在数据信息中，定义了具体样本的文件路径、对应标签等的信息。除此之外，数据集基类的另一个功能为不断地将数据送入数据流水线（data pipeline）中，进行数据预处理。

13.1.1 数据标注文件规范

为了统一不同任务的数据集接口，便于多任务的算法模型训练，OpenMMLab 制定了 **OpenMMLab 2.0 数据集格式规范**，数据集标注文件需符合该规范，数据集基类基于该规范去读取与解析数据标注文件。如果用户提供的数据标注文件不符合规定格式，用户可以选择将其转化为规定格式，并使用 OpenMMLab 的算法库基于该数据标注文件进行算法训练和测试。

OpenMMLab 2.0 数据集格式规范规定，标注文件必须为 `json` 或 `yaml`，`yml` 或 `pickle`，`pkl` 格式；标注文件中存储的字典必须包含 `metainfo` 和 `data_list` 两个字段。其中 `metainfo` 是一个字典，里面包含

数据集的元信息；data_list 是一个列表，列表中每个元素是一个字典，该字典定义了一个原始数据（raw data），每个原始数据包含一个或若干个训练/测试样本。

以下是一个 JSON 标注文件的例子（该例子中每个原始数据只包含一个训练/测试样本）：

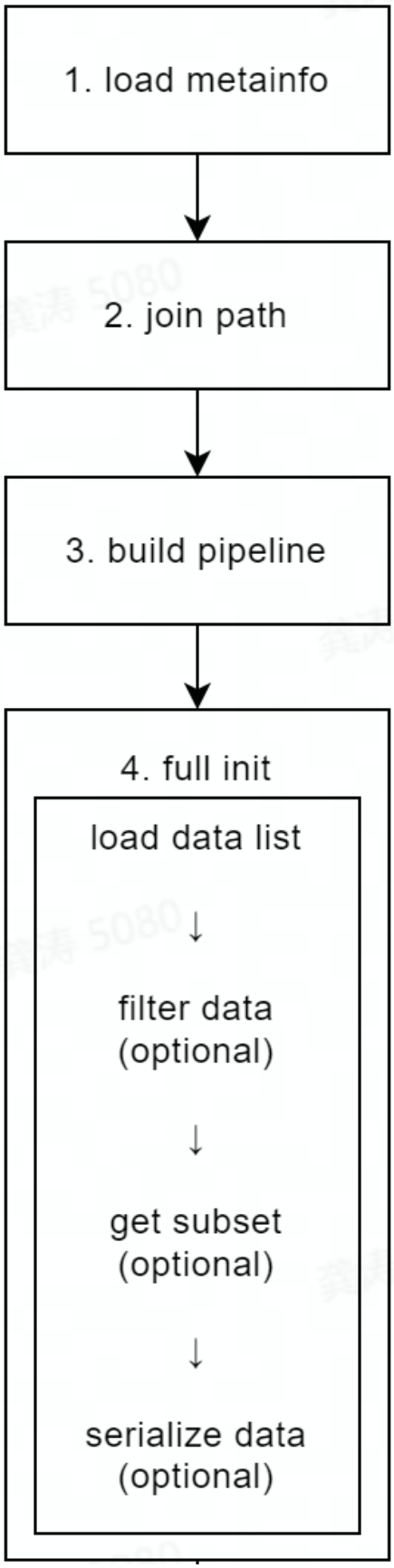
```
{
    'metainfo':
        {
            'classes': ('cat', 'dog'),
            ...
        },
    'data_list':
        [
            {
                'img_path': "xxx/xxx_0.jpg",
                'img_label': 0,
                ...
            },
            {
                'img_path': "xxx/xxx_1.jpg",
                'img_label': 1,
                ...
            },
            ...
        ]
}
```

同时假设数据存放路径如下：

```
data
├── annotations
|   ├── train.json
├── train
|   ├── xxx/xxx_0.jpg
|   ├── xxx/xxx_1.jpg
|   ├── ...
```

13.1.2 数据集基类的初始化流程

数据集基类的初始化流程如下图所示：


1. load metainfo
2. join path
3. build pipeline
4. full init
load data list
filter data
(optional)
get subset
(optional)
serialize data
(optional)

1. load metainfo：获取数据集的元信息，元信息有三种来源，优先级从高到低为：

- __init__() 方法中用户传入的 metainfo 字典；改动频率最高，因为用户可以在实例化数据集时，传入该参数；
- 类属性 BaseDataset.METAINFO 字典；改动频率中等，因为用户可以改动自定义数据集类中的类属性 BaseDataset.METAINFO；
- 标注文件中包含的 metainfo 字典；改动频率最低，因为标注文件一般不做改动。

  如果三种来源中有相同的字段，优先级最高的来源决定该字段的值，这些字段的优先级比较是：用户传入的 metainfo 字典里的字段 > BaseDataset.METAINFO 字典里的字段 > 标注文件中 metainfo 字典里的字段。

2. join path：处理数据与标注文件的路径；
3. build pipeline：构建数据流水线（data pipeline），用于数据预处理与数据准备；
4. full init：完全初始化数据集类，该步骤主要包含以下操作：

- load data list：读取与解析满足 OpenMMLab 2.0 数据集格式规范的标注文件，该步骤中会调用 parse_data_info() 方法，该方法负责解析标注文件里的每个原始数据；
- filter data (可选)：根据 filter_cfg 过滤无用数据，比如不包含标注的样本等；默认不做过滤操作，下游子类可以按自身所需对其进行重写；
- get subset (可选)：根据给定的索引或整数值采样数据，比如只取前 10 个样本参与训练/测试；默认不采样数据，即使用全部数据样本；
- serialize data (可选)：序列化全部样本，以达到节省内存的效果，详情请参考节省内存；默认操作为序列化全部样本。

数据集基类中包含的 parse_data_info() 方法用于将标注文件里的一个原始数据处理成一个或若干个训练/测试样本的方法。因此对于自定义数据集类，用户需要实现 parse_data_info() 方法。

13.1.3 数据集基类提供的接口

与 torch.utils.data.Dataset 类似，数据集初始化后，支持 __getitem__ 方法，用来索引数据，以及 __len__ 操作获取数据集大小，除此之外，OpenMMLab 的数据集基类主要提供了以下接口来访问具体信息：

- metainfo：返回元信息，返回值为字典
- get_data_info(idx)：返回指定 idx 的样本全量信息，返回值为字典
- __getitem__(idx)：返回指定 idx 的样本经过 pipeline 之后的结果（也就是送入模型的数据），返回值为字典
- __len__()：返回数据集长度，返回值为整数型

- `get_subset_(indices)`：根据 `indices` 以 inplace 的方式**修改原数据集类**。如果 `indices` 为 `int`，则原数据集类只包含前若干个数据样本；如果 `indices` 为 `Sequence[int]`，则原数据集类包含根据 `Sequence[int]` 指定的数据样本。
- `get_subset(indices)`：根据 `indices` 以**非** inplace 的方式**返回子数据集类**，即重新复制一份子数据集。如果 `indices` 为 `int`，则返回的子数据集类只包含前若干个数据样本；如果 `indices` 为 `Sequence[int]`，则返回的子数据集类包含根据 `Sequence[int]` 指定的数据样本。

13.2 使用数据集基类自定义数据集类

在了解了数据集基类的初始化流程与提供的接口之后，就可以基于数据集基类自定义数据集类。

13.2.1 对于满足 OpenMMLab 2.0 数据集格式规范的标注文件

如上所述，对于满足 OpenMMLab 2.0 数据集格式规范的标注文件，用户可以重载 `parse_data_info()` 来加载标签。以下是一个使用数据集基类来实现某一具体数据集的例子。

```
import os.path as osp

from mmengine.dataset import BaseDataset


class ToyDataset(BaseDataset):

    # 以上面标注文件为例，在这里 raw_data_info 代表 `data_list` 对应列表里的某个字典：
    # {
    #     'img_path': "xxx/xxx_0.jpg",
    #     'img_label': 0,
    #     ...
    # }
    def parse_data_info(self, raw_data_info):
        data_info = raw_data_info
        img_prefix = self.data_prefix.get('img_path', None)
        if img_prefix is not None:
            data_info['img_path'] = osp.join(
                img_prefix, data_info['img_path'])
        return data_info
```

使用自定义数据集类

在定义了数据集类后，就可以通过如下配置实例化 ToyDataset：

```
class LoadImage:

    def __call__(self, results):
        results['img'] = cv2.imread(results['img_path'])
        return results

class ParseImage:

    def __call__(self, results):
        results['img_shape'] = results['img'].shape
        return results

pipeline = [
    LoadImage(),
    ParseImage(),
]

toy_dataset = ToyDataset(
    data_root='data/',
    data_prefix=dict(img_path='train/'),
    ann_file='annotations/train.json',
    pipeline=pipeline)
```

同时可以使用数据集类提供的对外接口访问具体的样本信息：

```
toy_dataset.metainfo
# dict(classes=('cat', 'dog'))

toy_dataset.get_data_info(0)
# {
#     'img_path': "data/train/xxx/xxx_0.jpg",
#     'img_label': 0,
#     ...
# }

len(toy_dataset)
# 2

toy_dataset[0]
# {
```

(下页继续)

(续上页)

```
#     'img_path': "data/train/xxx/xxx_0.jpg",
#     'img_label': 0,
#     'img': a ndarray with shape (H, W, 3), which denotes the value of the image,
#     'img_shape': (H, W, 3) ,
#     ...
# }

# `get_subset` 接口不对原数据集类做修改，即完全复制一份新的
sub_toy_dataset = toy_dataset.get_subset(1)
len(toy_dataset), len(sub_toy_dataset)
# 2, 1

# `get_subset_` 接口会对原数据集类做修改，即 inplace 的方式
toy_dataset.get_subset_(1)
len(toy_dataset)
# 1
```

经过以上步骤，可以了解基于数据集基类如何自定义新的数据集类，以及如何使用自定义数据集类。

自定义视频的数据集类

在上面的例子中，标注文件的每个原始数据只包含一个训练/测试样本（通常是图像领域）。如果每个原始数据包含若干个训练/测试样本（通常是视频领域），则只需保证 `parse_data_info()` 的返回值为 `list[dict]` 即可：

```
from mmengine.dataset import BaseDataset


class ToyVideoDataset(BaseDataset):

    # raw_data_info 仍为一个字典，但它包含了多个样本
    def parse_data_info(self, raw_data_info):
        data_list = []

        ...

        for ... :

            data_info = dict()

            ...

            data_list.append(data_info)
```

(下页继续)

(续上页)

```
        return data_list
```

ToyVideoDataset 使用方法与 ToyDataset 类似，在此不做赘述。

13.2.2 对于不满足 OpenMMLab 2.0 数据集格式规范的标注文件

对于不满足 OpenMMLab 2.0 数据集格式规范的标注文件，有两种方式来使用数据集基类：

1. 将不满足规范的标注文件转换成满足规范的标注文件，再通过上述方式使用数据集基类。
2. 实现一个新的数据集类，继承自数据集基类，并且重载数据集基类的 load_data_list(self)：函数，处理不满足规范的标注文件，并保证返回值为 list[dict]，其中每个 dict 代表一个数据样本。

13.3 数据集基类的其它特性

数据集基类还包含以下特性：

13.3.1 懒加载（lazy init）

在数据集类实例化时，需要读取并解析标注文件，因此会消耗一定时间。然而在某些情况比如预测可视化时，往往只需要数据集类的元信息，可能并不需要读取与解析标注文件。为了节省这种情况下数据集类实例化的时间，数据集基类支持懒加载：

```
pipeline = [
    LoadImage(),
    ParseImage(),
]

toy_dataset = ToyDataset(
    data_root='data/',
    data_prefix=dict(img_path='train/'),
    ann_file='annotations/train.json',
    pipeline=pipeline,
    # 在这里传入 lazy_init 变量
    lazy_init=True)
```

当 lazy_init=True 时，ToyDataset 的初始化方法只执行了数据集基类的初始化流程中的 1、2、3 步骤，此时 toy_dataset 并未被完全初始化，因为 toy_dataset 并不会读取与解析标注文件，只会设置数据集类的元信息（metainfo）。

自然的，如果之后需要访问具体的数据信息，可以手动调用 `toy_dataset.full_init()` 接口来执行完整的初始化过程，在这个过程中数据标注文件将被读取与解析。调用 `get_data_info(idx)`，`__len__()`，`__getitem__(idx)`，`get_subset_(indices)`，`get_subset(indices)` 接口也会自动地调用 `full_init()` 接口来执行完整的初始化过程（仅在第一次调用时，之后调用不会重复地调用 `full_init()` 接口）：

```
# 完整初始化
toy_dataset.full_init()

# 初始化完毕，现在可以访问具体数据
len(toy_dataset)
# 2
toy_dataset[0]
# {
#     'img_path': "data/train/xxx/xxx_0.jpg",
#     'img_label': 0,
#     'img': a ndarray with shape (H, W, 3), which denotes the value the image,
#     'img_shape': (H, W, 3) ,
#     ...
# }
```

注意：

通过直接调用 `__getitem__()` 接口来执行完整初始化会带来一定风险：如果一个数据集类首先通过设置 `lazy_init=True` 未进行完全初始化，然后直接送入数据加载器（dataloader）中，在后续读取数据的过程中，不同的 worker 会同时读取与解析标注文件，虽然这样可能可以正常运行，但是会消耗大量的时间与内存。**因此，建议在需要访问具体数据之前，提前手动调用 `full_init()` 接口来执行完整的初始化过程**。

以上通过设置 `lazy_init=True` 未进行完全初始化，之后根据需求再进行完整初始化的方式，称为懒加载。

13.3.2 节省内存

在具体的读取数据过程中，数据加载器（dataloader）通常会起多个 worker 来预取数据，多个 worker 都拥有完整的数据集类备份，因此内存中会存在多份相同的 `data_list`，为了节省这部分内存消耗，数据集基类可以提前将 `data_list` 序列化存入内存中，使得多个 worker 可以共享同一份 `data_list`，以达到节省内存的目的。

数据集基类默认是将 `data_list` 序列化存入内存，也可以通过 `serialize_data` 变量（默认为 `True`）来控制是否提前将 `data_list` 序列化存入内存中：

```
pipeline = [
    LoadImage(),
    ParseImage(),
]
```

(下页继续)

(续上页)

```
toy_dataset = ToyDataset(
    data_root='data/',
    data_prefix=dict(img_path='train/'),
    ann_file='annotations/train.json',
    pipeline=pipeline,
    # 在这里传入 serialize_data 变量
    serialize_data=False)
```

上面例子不会提前将 data_list 序列化存入内存中，因此不建议在使用数据加载器开多个 worker 加载数据的情况下，使用这种方式实例化数据集类。

13.4 数据集基类包装

除了数据集基类，MMEngine 也提供了若干个数据集基类包装：ConcatDataset，RepeatDataset，ClassBalancedDataset。这些数据集基类包装同样也支持懒加载与拥有节省内存的特性。

13.4.1 ConcatDataset

MMEngine 提供了 ConcatDataset 包装来拼接多个数据集，使用方法如下：

```
from mmengine.dataset import ConcatDataset

pipeline = [
    LoadImage(),
    ParseImage(),
]

toy_dataset_1 = ToyDataset(
    data_root='data/',
    data_prefix=dict(img_path='train/'),
    ann_file='annotations/train.json',
    pipeline=pipeline)

toy_dataset_2 = ToyDataset(
    data_root='data/',
    data_prefix=dict(img_path='val/'),
    ann_file='annotations/val.json',
    pipeline=pipeline)

toy_dataset_12 = ConcatDataset(datasets=[toy_dataset_1, toy_dataset_2])
```

上述例子将数据集的 train 部分与 val 部分合成一个大的数据集。

13.4.2 RepeatDataset

MMEngine 提供了 RepeatDataset 包装来重复采样某个数据集若干次，使用方法如下：

```
from mmengine.dataset import RepeatDataset

pipeline = [
    LoadImage(),
    ParseImage(),
]

toy_dataset = ToyDataset(
    data_root='data/',
    data_prefix=dict(img_path='train/'),
    ann_file='annotations/train.json',
    pipeline=pipeline)

toy_dataset_repeat = RepeatDataset(dataset=toy_dataset, times=5)
```

上述例子将数据集的 train 部分重复采样了 5 次。

13.4.3 ClassBalancedDataset

MMEngine 提供了 ClassBalancedDataset 包装，来基于数据集中类别出现频率，重复采样相应样本。

注意：

ClassBalancedDataset 包装假设了被包装的数据集类支持 get_cat_ids(idx) 方法，get_cat_ids(idx) 方法返回一个列表，该列表包含了 idx 指定的 data_info 包含的样本类别，使用方法如下：

```
from mmengine.dataset import BaseDataset, ClassBalancedDataset

class ToyDataset(BaseDataset):

    def parse_data_info(self, raw_data_info):
        data_info = raw_data_info
        img_prefix = self.data_prefix.get('img_path', None)
        if img_prefix is not None:
            data_info['img_path'] = osp.join(
                img_prefix, data_info['img_path'])
```

(下页继续)

(续上页)

```
        return data_info

    # 必须支持的方法，需要返回样本的类别
    def get_cat_ids(self, idx):
        data_info = self.get_data_info(idx)
        return [int(data_info['img_label'])]

pipeline = [
    LoadImage(),
    ParseImage(),
]

toy_dataset = ToyDataset(
    data_root='data/',
    data_prefix=dict(img_path='train/'),
    ann_file='annotations/train.json',
    pipeline=pipeline)

toy_dataset_repeat = ClassBalancedDataset(dataset=toy_dataset, oversample_thr=1e-3)
```

上述例子将数据集的 `train` 部分以 `oversample_thr=1e-3` 重新采样，具体地，对于数据集中出现频率低于 `1e-3` 的类别，会重复采样该类别对应的样本，否则不重复采样，具体采样策略请参考 `ClassBalancedDataset` API 文档。

13.4.4 自定义数据集类包装

由于数据集基类实现了懒加载的功能，因此在自定义数据集类包装时，需要遵循一些规则，下面以一个例子的方式来展示如何自定义数据集类包装：

```
from mmengine.dataset import BaseDataset
from mmengine.registry import DATASETS


@DATASETS.register_module()
class ExampleDatasetWrapper:

    def __init__(self, dataset, lazy_init=False, ...):
        # 构建原数据集 (self.dataset)
        if isinstance(dataset, dict):
            self.dataset = DATASETS.build(dataset)
        elif isinstance(dataset, BaseDataset):
            self.dataset = dataset
```

(下页继续)

(续上页)

```
        else:
            raise TypeError(
                'elements in datasets sequence should be config or '
                f'`BaseDataset` instance, but got {type(dataset)}')
        # 记录原数据集的元信息
        self._metainfo = self.dataset.metainfo

        '''
        1. 在这里实现一些代码，来记录用于包装数据集的一些超参。
        '''

        self._fully_initialized = False
        if not lazy_init:
            self.full_init()

    def full_init(self):
        if self._fully_initialized:
            return

        # 将原数据集完全初始化
        self.dataset.full_init()

        '''
        2. 在这里实现一些代码，来包装原数据集。
        '''

        self._fully_initialized = True

    @force_full_init
    def _get_ori_dataset_idx(self, idx: int):

        '''
        3. 在这里实现一些代码，来将包装的索引 `idx` 映射到原数据集的索引 `ori_idx`。
        '''
        ori_idx = ...

        return ori_idx

    # 提供与 `self.dataset` 一样的对外接口。
    @force_full_init
    def get_data_info(self, idx):
        sample_idx = self._get_ori_dataset_idx(idx)
        return self.dataset.get_data_info(sample_idx)
```

(下页继续)

(续上页)

```
    # 提供与 `self.dataset` 一样的对外接口。
    def __getitem__(self, idx):
        if not self._fully_initialized:
            warnings.warn('Please call `full_init` method manually to '
                          'accelerate the speed.')
            self.full_init()

        sample_idx = self._get_ori_dataset_idx(idx)
        return self.dataset[sample_idx]

    # 提供与 `self.dataset` 一样的对外接口。
    @force_full_init
    def __len__(self):

        '''
        4. 在这里实现一些代码，来计算包装数据集之后的长度。
        '''
        len_wrapper = ...

        return len_wrapper

    # 提供与 `self.dataset` 一样的对外接口。
    @property
    def metainfo(self)
        return copy.deepcopy(self._metainfo)
```

CHAPTER 14

抽象数据接口

在模型的训练/测试过程中，组件之间往往有大量的数据需要传递，不同的算法需要传递的数据经常是不一样的，例如，训练单阶段检测器需要获得数据集的标注框（ground truth bounding boxes）和标签（ground truth box labels），训练 Mask R-CNN 时还需要实例掩码（instance masks）。训练这些模型时的代码如下所示

```
for img, img_metas, gt_bboxes, gt_labels in data_loader:
    loss = retinanet(img, img_metas, gt_bboxes, gt_labels)
```

```
for img, img_metas, gt_bboxes, gt_masks, gt_labels in data_loader:
    loss = mask_rcnn(img, img_metas, gt_bboxes, gt_masks, gt_labels)
```

可以发现，在不加封装的情况下，不同算法所需数据的不一致导致了不同算法模块之间接口的不一致，影响了算法库的拓展性，同时一个算法库内的模块为了保持兼容性往往在接口上存在冗余。上述弊端在算法库之间会体现地更加明显，导致在实现多任务（同时进行如语义分割、检测、关键点检测等多个任务）感知模型时模块难以复用，接口难以拓展。

为了解决上述问题，MMEngine 定义了一套抽象的数据接口来封装模型运行过程中的各种数据。假设将上述不同的数据封装进 `data_sample` ，不同算法的训练都可以被抽象和统一成如下代码

```
for img, data_sample in dataloader:
    loss = model(img, data_sample)
```

通过对各种数据提供统一的封装，抽象数据接口统一并简化了算法库中各个模块的接口，可以被用于算法库中 dataset，model，visualizer，和 evaluator 组件之间，或者 model 内各个模块之间的数据传递。抽象数据接口实现了基本的增/删/改/查功能，同时支持不同设备之间的迁移，支持类字典和张量的操作，可以充分满足算

法库对于这些数据的使用要求。基于 MMEngine 的算法库可以继承这套抽象数据接口并实现自己的抽象数据接口来适应不同算法中数据的特点与实际需要，在保持统一接口的同时提高了算法模块的拓展性。

在实际实现过程中，算法库中的各个组件所具备的数据接口，一般为如下两个种：

- 一个训练或测试样本 (例如一张图像) 的所有的标注信息和预测信息的集合，例如数据集的输出、模型以及可视化器的输入一般为单个训练或测试样本的所有信息。MMEngine 将其定义为数据样本 (DataSample)
- 单一类型的预测或标注，一般是算法模型中某个子模块的输出，例如二阶段检测中 RPN 的输出、语义分割模型的输出、关键点分支的输出，GAN 中生成器的输出等。MMengine 将其定义为数据元素 (XXXData)

下边首先介绍一下数据样本与数据元素的基类*BaseDataElement*。

14.1 数据基类 (BaseDataElement)

`BaseDataElement` 中存在两种类型的数据，一种是 `data` 类型，如标注框、框的标签、和实例掩码等；另一种是 `metainfo` 类型，包含数据的元信息以确保数据的完整性，如 `img_shape`, `img_id` 等数据所在图片的一些基本信息，方便可视化等情况下对数据进行恢复和使用。用户在创建 `BaseDataElement` 的过程中需要对这两类属性的数据进行显式地区分和声明。

为了能够更加方便地使用 `BaseDataElement`，`data` 和 `metainfo` 中的数据均为 `BaseDataElement` 的属性。我们可以通过访问类属性的方式直接访问 `data` 和 `metainfo` 中的数据。此外，`BaseDataElement` 还提供了很多方法，方便我们操作 `data` 内的数据：

- 增/删/改/查 `data` 中不同字段的数据
- 将 `data` 迁移至目标设备
- 支持像访问字典/张量一样访问 data 内的数据以充分满足算法库对于这些数据的使用要求。

14.1.1 1. 数据元素的创建

`BaseDataElement` 的 data 参数可以直接通过 `key=value` 的方式自由添加，metainfo 的字段需要显式通过关键字 `metainfo` 指定。

```
import torch
from mmengine.structures import BaseDataElement
# 可以声明一个空的 object
data_element = BaseDataElement()

bboxes = torch.rand((5, 4))  # 假定 bboxes 是一个 Nx4 维的 tensor, N 代表框的个数
scores = torch.rand((5,))  # 假定框的分数是一个 N 维的 tensor, N 代表框的个数
img_id = 0  # 图像的 ID
H = 800  # 图像的高度
W = 1333  # 图像的宽度
```

(下页继续)

(续上页)

```
# 直接设置 BaseDataElement 的 data 参数
data_element = BaseDataElement(bboxes=bboxes, scores=scores)

# 显式声明来设置 BaseDataElement 的参数 metainfo
data_element = BaseDataElement(
    bboxes=bboxes,
    scores=scores,
    metainfo=dict(img_id=img_id, img_shape=(H, W)))
```

14.1.2 2. new 与 clone 函数

用户可以使用 new() 函数通过已有的数据接口创建一个具有相同状态和数据的抽象数据接口。用户可以在创建新 BaseDataElement 时设置 metainfo 和 data，用于创建仅 data 或 metainfo 具有相同状态和数据的抽象接口。比如 new(metainfo=xx) 使得新的 BaseDataElement 与被 clone 的 BaseDataElement 包含相同的 data 内容，但 metainfo 为新设置的内容。也可以直接使用 clone() 来获得一份深拷贝，clone() 函数的行为与 PyTorch 中 Tensor 的 clone() 参数保持一致。

```
data_element = BaseDataElement(
    bboxes=torch.rand((5, 4)),
    scores=torch.rand((5,)),
    metainfo=dict(img_id=1, img_shape=(640, 640)))

# 可以在创建新 `BaseDataElement` 时设置 metainfo 和 data，使得新的 BaseDataElement 有相同未被
设置的数据
data_element1 = data_element.new(metainfo=dict(img_id=2, img_shape=(320, 320)))
print('bboxes is in data_element1:', 'bboxes' in data_element1) # True
print('bboxes in data_element1 is same as bbox in data_element', (data_element1.
↪bboxes == data_element.bboxes).all())
print('img_id in data_element1 is', data_element1.img_id == 2) # True

data_element2 = data_element.new(label=torch.rand(5,))
print('bboxes is not in data_element2', 'bboxes' not in data_element2) # True
print('img_id in data_element2 is same as img_id in data_element', data_element2.img_
↪id == data_element.img_id)
print('label in data_element2 is', 'label' in data_element2)

# 也可以通过 `clone` 构建一个新的 object，新的 object 会拥有和 data_element 相同的 data 和␣
↪metainfo 内容以及状态。
data_element2 = data_element1.clone()
```

```
bboxes is in data_element1: True
bboxes in data_element1 is same as bbox in data_element tensor(True)
img_id in data_element1 is True
bboxes is not in data_element2 True
img_id in data_element2 is same as img_id in data_element True
label in data_element2 is True
```

14.1.3 3. 属性的增加与查询

对增加属性而言，用户可以像增加类属性那样增加 data 内的属性；对 metainfo 而言，一般储存的为一些图像的元信息，一般情况下不会修改，如果需要增加，用户应当使用 set_metainfo 接口显示地修改。

对查询而言，用户可以可以通过 keys，values，和 items 来访问只存在于 data 中的键值，也可以通过 metainfo_keys，metainfo_values，和 metainfo_items 来访问只存在于 metainfo 中的键值。用户还能通过 all_keys，all_values，all_items 来访问 BaseDataElement 的所有的属性并且不区分他们的类型。

同时为了方便使用，用户可以像访问类属性一样访问 data 与 metainfo 内的数据，或着类字典方式通过 get() 接口访问数据。

注意：

1. BaseDataElement 不支持 metainfo 和 data 属性中有同名的字段，所以用户应当避免 metainfo 和 data 属性中设置相同的字段，否则 BaseDataElement 会报错。
2. 考虑到 InstanceData 和 PixelData 支持对数据进行切片操作，为了避免 [] 用法的不一致，同时减少同种需求的不同方法，BaseDataElement 不支持像字典那样访问和设置它的属性，所以类似 BaseDataElement[name] 的取值赋值操作是不被支持的。

```
data_element = BaseDataElement()
# 通过 `set_metainfo`设置 data_element 的 metainfo 字段,
# 同时 img_id 和 img_shape 成为 data_element 的属性
data_element.set_metainfo(dict(img_id=9, img_shape=(100, 100)))
# 查看 metainfo 的 key, value 和 item
print("metainfo'keys are", data_element.metainfo_keys())
print("metainfo'values are", data_element.metainfo_values())
for k, v in data_element.metainfo_items():
    print(f'{k}: {v}')

print("通过类属性查看 img_id 和 img_shape")
print('img_id:', data_element.img_id)
print('img_shape:', data_element.img_shape)
```

```
metainfo'keys are ['img_id', 'img_shape']
metainfo'values are [9, (100, 100)]
img_id: 9
img_shape: (100, 100)
通过类属性查看 img_id 和 img_shape
img_id: 9
img_shape: (100, 100)
```

```
# 通过类属性直接设置 BaseDataElement 中的 data 字段
data_element.scores = torch.rand((5,))
data_element.bboxes = torch.rand((5, 4))

print("data's key is:", data_element.keys())
print("data's value is:", data_element.values())
for k, v in data_element.items():
    print(f'{k}: {v}')

print("通过类属性查看 scores 和 bboxes")
print('scores:', data_element.scores)
print('bboxes:', data_element.bboxes)

print("通过 get() 查看 scores 和 bboxes")
print('scores:', data_element.get('scores', None))
print('bboxes:', data_element.get('bboxes', None))
print('fake:', data_element.get('fake', 'not exist'))
```

```
data's key is: ['scores', 'bboxes']
data's value is: [tensor([0.7937, 0.6307, 0.3682, 0.4425, 0.8515]), tensor([[0.9204,␣
↪0.2110, 0.2886, 0.7925],
        [0.7993, 0.8982, 0.5698, 0.4120],
        [0.7085, 0.7016, 0.3069, 0.3216],
        [0.0206, 0.5253, 0.1376, 0.9322],
        [0.2512, 0.7683, 0.3010, 0.2672]])]
scores: tensor([0.7937, 0.6307, 0.3682, 0.4425, 0.8515])
bboxes: tensor([[0.9204, 0.2110, 0.2886, 0.7925],
        [0.7993, 0.8982, 0.5698, 0.4120],
        [0.7085, 0.7016, 0.3069, 0.3216],
        [0.0206, 0.5253, 0.1376, 0.9322],
        [0.2512, 0.7683, 0.3010, 0.2672]])
通过类属性查看 scores 和 bboxes
scores: tensor([0.7937, 0.6307, 0.3682, 0.4425, 0.8515])
bboxes: tensor([[0.9204, 0.2110, 0.2886, 0.7925],
```

(下页继续)

(续上页)

```
        [0.7993, 0.8982, 0.5698, 0.4120],
        [0.7085, 0.7016, 0.3069, 0.3216],
        [0.0206, 0.5253, 0.1376, 0.9322],
        [0.2512, 0.7683, 0.3010, 0.2672]])
通过 get() 查看 scores 和 bboxes
scores: tensor([0.7937, 0.6307, 0.3682, 0.4425, 0.8515])
bboxes: tensor([[0.9204, 0.2110, 0.2886, 0.7925],
        [0.7993, 0.8982, 0.5698, 0.4120],
        [0.7085, 0.7016, 0.3069, 0.3216],
        [0.0206, 0.5253, 0.1376, 0.9322],
        [0.2512, 0.7683, 0.3010, 0.2672]])
fake: not exist
```

```
print("All key in data_element is:", data_element.all_keys())
print("The length of values in data_element is", len(data_element.all_values()))
for k, v in data_element.all_items():
    print(f'{k}: {v}')
```

```
All key in data_element is: ['img_id', 'img_shape', 'scores', 'bboxes']
The length of values in data_element is 4
img_id: 9
img_shape: (100, 100)
scores: tensor([0.7937, 0.6307, 0.3682, 0.4425, 0.8515])
bboxes: tensor([[0.9204, 0.2110, 0.2886, 0.7925],
        [0.7993, 0.8982, 0.5698, 0.4120],
        [0.7085, 0.7016, 0.3069, 0.3216],
        [0.0206, 0.5253, 0.1376, 0.9322],
        [0.2512, 0.7683, 0.3010, 0.2672]])
```

14.1.4 4. 属性的删改

用户可以像修改实例属性一样修改 BaseDataElement 的 data，对 metainfo 而言一般储存的为一些图像的元信息，一般情况下不会修改，如果需要修改，用户应当使用 set_metainfo 接口显示的修改。

同时为了操作的便捷性，对 data 和 metainfo 中的数据可以通过 del 直接删除，也支持 pop 在在访问属性后删除属性。

```
data_element = BaseDataElement(
    bboxes=torch.rand((6, 4)), scores=torch.rand((6,)),
    metainfo=dict(img_id=0, img_shape=(640, 640))
)
```

(下页继续)

(续上页)

```
for k, v in data_element.all_items():
    print(f'{k}: {v}')
```

```
img_id: 0
img_shape: (640, 640)
scores: tensor([0.8445, 0.6678, 0.8172, 0.9125, 0.7186, 0.5462])
bboxes: tensor([[0.5773, 0.0289, 0.4793, 0.7573],
        [0.8187, 0.8176, 0.3455, 0.3368],
        [0.6947, 0.5592, 0.7285, 0.0281],
        [0.7710, 0.9867, 0.7172, 0.5815],
        [0.3999, 0.9192, 0.7817, 0.2535],
        [0.2433, 0.0132, 0.1757, 0.6196]])
```

```
# 对 data 进行修改
data_element.bboxes = data_element.bboxes * 2
data_element.scores = data_element.scores * -1
for k, v in data_element.items():
    print(f'{k}: {v}')

# 删除 data 中的属性
del data_element.bboxes
for k, v in data_element.items():
    print(f'{k}: {v}')

data_element.pop('scores', None)
print('The keys in data is', data_element.keys())
```

```
scores: tensor([-0.8445, -0.6678, -0.8172, -0.9125, -0.7186, -0.5462])
bboxes: tensor([[1.1546, 0.0578, 0.9586, 1.5146],
        [1.6374, 1.6352, 0.6911, 0.6735],
        [1.3893, 1.1185, 1.4569, 0.0562],
        [1.5420, 1.9734, 1.4344, 1.1630],
        [0.7999, 1.8384, 1.5635, 0.5070],
        [0.4867, 0.0264, 0.3514, 1.2392]])
scores: tensor([-0.8445, -0.6678, -0.8172, -0.9125, -0.7186, -0.5462])
The keys in data is []
```

```
# 对 metainfo 进行修改
data_element.set_metainfo(dict(img_shape = (1280, 1280), img_id=10))
print(data_element.img_shape)  # (1280, 1280)
for k, v in data_element.metainfo_items():
    print(f'{k}: {v}')
```

(下页继续)

(续上页)

```
# 提供了便捷的属性删除和访问操作 pop
del data_element.img_shape
for k, v in data_element.metainfo_items():
    print(f'{k}: {v}')

data_element.pop('img_id')
print('The keys in metainfo is', data_element.metainfo_keys())
```

```
(1280, 1280)
img_id: 10
img_shape: (1280, 1280)
img_id: 10
The keys in metainfo is []
```

14.1.5 5. 类张量操作

用户可以像 torch.Tensor 那样对 `BaseDataElement` 的 data 进行状态转换，目前支持 `cuda`，`cpu`，`to`，`numpy` 等操作。其中，`to` 函数拥有和 `torch.Tensor.to()` 相同的接口，使得用户可以灵活地将被封装的 tensor 进行状态转换。**注意：**这些接口只会处理类型为 np.array，torch.Tensor，或者数字的序列，其他属性的数据（如字符串）会被跳过处理。

```
data_element = BaseDataElement(
    bboxes=torch.rand((6, 4)), scores=torch.rand((6,)),
    metainfo=dict(img_id=0, img_shape=(640, 640))
)
# 将所有 data 转移到 GPU 上
cuda_element_1 = data_element.cuda()
print('cuda_element_1 is on the device of', cuda_element_1.bboxes.device)  # cuda:0
cuda_element_2 = data_element.to('cuda:0')
print('cuda_element_1 is on the device of', cuda_element_2.bboxes.device)  # cuda:0

# 将所有 data 转移到 cpu 上
cpu_element_1 = cuda_element_1.cpu()
print('cpu_element_1 is on the device of', cpu_element_1.bboxes.device)  # cpu
cpu_element_2 = cuda_element_2.to('cpu')
print('cpu_element_2 is on the device of', cpu_element_2.bboxes.device)  # cpu

# 将所有 data 变成 FP16
fp16_instances = cuda_element_1.to(
    device=None, dtype=torch.float16, non_blocking=False, copy=False,
    memory_format=torch.preserve_format)
```

(下页继续)

(续上页)

```
print('The type of bboxes in fp16_instances is', fp16_instances.bboxes.dtype)  #␣
↪torch.float16

# 阻断所有 data 的梯度
cuda_element_3 = cuda_element_2.detach()
print('The data in cuda_element_3 requires grad: ', cuda_element_3.bboxes.requires_
↪grad)
# 转移 data 到 numpy array
np_instances = cpu_element_1.numpy()
print('The type of cpu_element_1 is convert to', type(np_instances.bboxes))
```

```
cuda_element_1 is on the device of cuda:0
cuda_element_1 is on the device of cuda:0
cpu_element_1 is on the device of cpu
cpu_element_2 is on the device of cpu
The type of bboxes in fp16_instances is torch.float16
The data in cuda_element_3 requires grad:  False
The type of cpu_element_1 is convert to <class 'numpy.ndarray'>
```

14.1.6 6. 属性的展示

BaseDataElement 还实现了 __repr__，因此，用户可以直接通过 print 函数看到其中的所有数据信息。同时，为了便捷开发者 debug，BaseDataElement 中的属性都会添加进 __dict__ 中，方便用户在 IDE 界面可以直观看到 BaseDataElement 中的内容。一个完整的属性展示如下

```
img_meta = dict(img_shape=(800, 1196, 3), pad_shape=(800, 1216, 3))
instance_data = BaseDataElement(metainfo=img_meta)
instance_data.det_labels = torch.LongTensor([0, 1, 2, 3])
instance_data.det_scores = torch.Tensor([0.01, 0.1, 0.2, 0.3])
print(instance_data)
```

```
<BaseDataElement(

    META INFORMATION
    pad_shape: (800, 1216, 3)
    img_shape: (800, 1196, 3)

    DATA FIELDS
    det_labels: tensor([0, 1, 2, 3])
    det_scores: tensor([0.0100, 0.1000, 0.2000, 0.3000])
) at 0x7f9f339f85b0>
```

14.2 数据元素 (xxxData)

MMEngine 将数据元素情况划分为三个类别:

- 实例数据 (InstanceData): 主要针对的是上层任务 (high-level) 中，对图像中所有实例相关的数据进行封装，比如检测框 (bounding boxes), 物体类别 (box labels), 实例掩码 (instance masks), 关键点 (key points), 文字边界 (polygons), 跟踪 id(tracking ids) 等. 所有实例相关的数据的**长度一致**，均为图像中实例的个数。
- 像素数据 (PixelData): 主要针对底层任务 (low-level) 以及需要感知像素级别标签的部分上层任务。像素数据对像素级相关的数据进行封装，比如语义分割中的分割图（segmentation map）, 光流任务中的光流图（flow map）, 全景分割中的全景分割图（panoptic seg map）；底层任务中生成的各种图像，比如超分辨图，去噪图，以及生成的各种风格图。这些数据的特点是都是三维或四维数组，最后两维度为数据的高度 (height) 和宽度 (width)，且具有相同的 height 和 width
- 标签数据 (LabelData): 主要标签级别的数据进行封装，比如图像分类，多分类中的类别，图像生成中生成图像的类别内容，或者文字识别中的文本等。

14.2.1 InstanceData

InstanceData 在 BaseDataElement 的基础上，对 data 存储的数据做了限制，即要求存储在 data 中的数据的长度一致。比如在目标检测中, 假设一张图像中有 N 个目标 (instance)，可以将图像的所有边界框 (bbox)，类别 (label) 等存储在 InstanceData 中, InstanceData 的 bbox 和 label 的长度相同。基于上述假定对 InstanceData 进行了扩展，包括:

- 对 InstanceData 中 data 所存储的数据进行了长度校验
- data 部分支持类字典访问和设置它的属性
- 支持基础索引，切片以及高级索引功能
- 支持具有**相同的 key** 但是不同 InstanceData 的拼接功能。这些扩展功能除了支持基础的数据结构，比如 torch.tensor, numpy.dnarray, list, str, tuple, 也可以是自定义的数据结构，只要自定义数据结构实现了 __len__, __getitem__ and cat.

数据校验

InstanceData 中 data 的数据长度要保持一致，如果传入不同长度的新数据，将会报错。

```
from mmengine.structures import InstanceData
import torch
import numpy as np

img_meta = dict(img_shape=(800, 1196, 3), pad_shape=(800, 1216, 3))
instance_data = InstanceData(metainfo=img_meta)
instance_data.det_labels = torch.LongTensor([2, 3])
```

(下页继续)

(续上页)

```
instance_data.det_scores = torch.Tensor([0.8, 0.7])
instance_data.bboxes = torch.rand((2, 4))
print('The length of instance_data is', len(instance_data))  # 2

instance_data.bboxes = torch.rand((3, 4))
```

```
The length of instance_data is 2
AssertionError: the length of values 3 is not consistent with the length of this␣
↪:obj:`InstanceData` 2
```

14.2.2 类字典访问和设置属性

InstanceData 支持类似字典的操作访问和设置其 **data** 属性。

```
img_meta = dict(img_shape=(800, 1196, 3), pad_shape=(800, 1216, 3))
instance_data = InstanceData(metainfo=img_meta)
instance_data["det_labels"] = torch.LongTensor([2, 3])
instance_data["det_scores"] = torch.Tensor([0.8, 0.7])
instance_data.bboxes = torch.rand((2, 4))
print(instance_data)
```

```
<InstanceData(

    META INFORMATION
    pad_shape: (800, 1216, 3)
    img_shape: (800, 1196, 3)

    DATA FIELDS
    det_labels: tensor([2, 3])
    det_scores: tensor([0.8000, 0.7000])
    bboxes: tensor([[0.6576, 0.5435, 0.5253, 0.8273],
                [0.4533, 0.6848, 0.7230, 0.9279]])
) at 0x7f9f339f8ca0>
```

索引与切片

InstanceData 支持 Python 中类似列表的索引与切片，同时也支持类似 numpy 的高级索引操作。

```
img_meta = dict(img_shape=(800, 1196, 3), pad_shape=(800, 1216, 3))
instance_data = InstanceData(metainfo=img_meta)
instance_data.det_labels = torch.LongTensor([2, 3])
instance_data.det_scores = torch.Tensor([0.8, 0.7])
instance_data.bboxes = torch.rand((2, 4))
print(instance_data)
```

```
<InstanceData(

    META INFORMATION
    pad_shape: (800, 1216, 3)
    img_shape: (800, 1196, 3)

    DATA FIELDS
    det_labels: tensor([2, 3])
    det_scores: tensor([0.8000, 0.7000])
    bboxes: tensor([[0.1872, 0.1669, 0.7563, 0.8777],
                [0.3421, 0.7104, 0.6000, 0.1518]])
) at 0x7f9f312b4dc0>
```

1. 索引

```
print(instance_data[1])
```

```
<InstanceData(

    META INFORMATION
    pad_shape: (800, 1216, 3)
    img_shape: (800, 1196, 3)

    DATA FIELDS
    det_labels: tensor([3])
    det_scores: tensor([0.7000])
    bboxes: tensor([[0.3421, 0.7104, 0.6000, 0.1518]])
) at 0x7f9f312b4610>
```

2. 切片

```
print(instance_data[0:1])
```

```
<InstanceData(

    META INFORMATION
    pad_shape: (800, 1216, 3)
    img_shape: (800, 1196, 3)

    DATA FIELDS
    det_labels: tensor([2])
    det_scores: tensor([0.8000])
    bboxes: tensor([[0.1872, 0.1669, 0.7563, 0.8777]])
) at 0x7f9f312b4e20>
```

3. 高级索引

- 列表索引

```
sorted_results = instance_data[instance_data.det_scores.sort().indices]
print(sorted_results)
```

```
<InstanceData(

    META INFORMATION
    pad_shape: (800, 1216, 3)
    img_shape: (800, 1196, 3)

    DATA FIELDS
    det_labels: tensor([3, 2])
    det_scores: tensor([0.7000, 0.8000])
    bboxes: tensor([[0.3421, 0.7104, 0.6000, 0.1518],
                [0.1872, 0.1669, 0.7563, 0.8777]])
) at 0x7f9f312b4a90>
```

- 布尔索引

```
filter_results = instance_data[instance_data.det_scores > 0.75]
print(filter_results)
```

```
<InstanceData(

    META INFORMATION
    pad_shape: (800, 1216, 3)
    img_shape: (800, 1196, 3)

    DATA FIELDS
```

(下页继续)

(续上页)

```
    det_labels: tensor([2])
    det_scores: tensor([0.8000])
    bboxes: tensor([[0.1872, 0.1669, 0.7563, 0.8777]])
) at 0x7fa061299dc0>
```

4. 结果为空

```
empty_results = instance_data[instance_data.det_scores > 1]
print(empty_results)
```

```
<InstanceData(

    META INFORMATION
    pad_shape: (800, 1216, 3)
    img_shape: (800, 1196, 3)

    DATA FIELDS
    det_labels: tensor([], dtype=torch.int64)
    det_scores: tensor([])
    bboxes: tensor([], size=(0, 4))
) at 0x7f9f439cccd0>
```

拼接 (cat)

用户可以将两个具有相同 key 的 `InstanceData` 拼接成一个 `InstanceData`。对于长度分别为 N 和 M 的两个 `InstanceData`，拼接后为长度 N + M 的新的 `InstanceData`

```
img_meta = dict(img_shape=(800, 1196, 3), pad_shape=(800, 1216, 3))
instance_data = InstanceData(metainfo=img_meta)
instance_data.det_labels = torch.LongTensor([2, 3])
instance_data.det_scores = torch.Tensor([0.8, 0.7])
instance_data.bboxes = torch.rand((2, 4))
print('The length of instance_data is', len(instance_data))
cat_results = InstanceData.cat([instance_data, instance_data])
print('The length of instance_data is', len(cat_results))
print(cat_results)
```

```
The length of instance_data is 2
The length of instance_data is 4
<InstanceData(

    META INFORMATION
```

(下页继续)

(续上页)

```
    pad_shape: (800, 1216, 3)
    img_shape: (800, 1196, 3)

    DATA FIELDS
    det_labels: tensor([2, 3, 2, 3])
    det_scores: tensor([0.8000, 0.7000, 0.8000, 0.7000])
    bboxes: tensor([[0.5341, 0.8962, 0.9043, 0.2824],
                [0.3864, 0.2215, 0.7610, 0.7060],
                [0.5341, 0.8962, 0.9043, 0.2824],
                [0.3864, 0.2215, 0.7610, 0.7060]])
) at 0x7fa061d4a9d0>
```

自定义数据结构

对于自定义结构如果想使用上述扩展要求需要实现 `__len__`, `__getitem__` 和 `cat` 三个接口.

```
import itertools

class TmpObject:
    def __init__(self, tmp) -> None:
        assert isinstance(tmp, list)
        self.tmp = tmp

    def __len__(self):
        return len(self.tmp)

    def __getitem__(self, item):
        if type(item) == int:
            if item >= len(self) or item < -len(self):  # type:ignore
                raise IndexError(f'Index {item} out of range!')
            else:
                # keep the dimension
                item = slice(item, None, len(self))
        return TmpObject(self.tmp[item])

    @staticmethod
    def cat(tmp_objs):
        assert all(isinstance(results, TmpObject) for results in tmp_objs)
        if len(tmp_objs) == 1:
            return tmp_objs[0]
        tmp_list = [tmp_obj.tmp for tmp_obj in tmp_objs]
        tmp_list = list(itertools.chain(*tmp_list))
        new_data = TmpObject(tmp_list)
```

(下页继续)

(续上页)

```
        return new_data

    def __repr__(self):
        return str(self.tmp)
```

```
img_meta = dict(img_shape=(800, 1196, 3), pad_shape=(800, 1216, 3))
instance_data = InstanceData(metainfo=img_meta)
instance_data.det_labels = torch.LongTensor([2, 3])
instance_data["det_scores"] = torch.Tensor([0.8, 0.7])
instance_data.bboxes = torch.rand((2, 4))
instance_data.polygons = TmpObject([[1, 2, 3, 4], [5, 6, 7, 8]])
print(instance_data)
```

```
<InstanceData(

    META INFORMATION
    pad_shape: (800, 1216, 3)
    img_shape: (800, 1196, 3)

    DATA FIELDS
    det_labels: tensor([2, 3])
    polygons: [[1, 2, 3, 4], [5, 6, 7, 8]]
    det_scores: tensor([0.8000, 0.7000])
    bboxes: tensor([[0.4207, 0.0778, 0.9959, 0.1967],
                [0.4679, 0.7934, 0.5372, 0.4655]])
) at 0x7fa061b5d2b0>
```

```
# 高级索引
print(instance_data[instance_data.det_scores > 0.75])
```

```
<InstanceData(

    META INFORMATION
    pad_shape: (800, 1216, 3)
    img_shape: (800, 1196, 3)

    DATA FIELDS
    bboxes: tensor([[0.4207, 0.0778, 0.9959, 0.1967]])
    det_labels: tensor([2])
    det_scores: tensor([0.8000])
    polygons: [[1, 2, 3, 4]]
) at 0x7f9f312716d0>
```

```
# 拼接
print(InstanceData.cat([instance_data, instance_data]))
```

```
<InstanceData(

    META INFORMATION
    pad_shape: (800, 1216, 3)
    img_shape: (800, 1196, 3)

    DATA FIELDS
    bboxes: tensor([[0.4207, 0.0778, 0.9959, 0.1967],
                [0.4679, 0.7934, 0.5372, 0.4655],
                [0.4207, 0.0778, 0.9959, 0.1967],
                [0.4679, 0.7934, 0.5372, 0.4655]])
    det_labels: tensor([2, 3, 2, 3])
    det_scores: tensor([0.8000, 0.7000, 0.8000, 0.7000])
    polygons: [[1, 2, 3, 4], [5, 6, 7, 8], [1, 2, 3, 4], [5, 6, 7, 8]]
) at 0x7f9f31271490>
```

14.2.3 PixelData

PixelData 在 BaseDataElement 的基础上，同样对对 data 中存储的数据做了限制:

- 所有 data 内的数据均为 3 维，并且顺序为 (通道，高，宽)
- 所有在 data 内的数据要有相同的长和宽基于上述假定对 PixelData 进行了扩展，包括:
- 对 PixelData 中 data 所存储的数据进行了尺寸的校验
- 支持对 data 部分的数据对实例进行空间维度的索引和切片。

14.2.4 数据校验

PixelData 会对传入到 data 的数据进行维度与长宽的校验。

```
from mmengine.structures import PixelData
import random
import torch
import numpy as np
metainfo = dict(
    img_id=random.randint(0, 100),
    img_shape=(random.randint(400, 600), random.randint(400, 600)))
image = np.random.randint(0, 255, (4, 20, 40))
featmap = torch.randint(0, 255, (10, 20, 40))
```

(下页继续)

(续上页)

```
pixel_data = PixelData(metainfo=metainfo,
                       image=image,
                       featmap=featmap)
print('The shape of pixel_data is', pixel_data.shape)
# set
pixel_data.map3 = torch.randint(0, 255, (20, 40))
print('The shape of pixel_data is', pixel_data.map3.shape)
```

```
The shape of pixel_data is (20, 40)
The shape of pixel_data is torch.Size([1, 20, 40])
```

```
pixel_data.map2 = torch.randint(0, 255, (3, 20, 30))
# AssertionError: the height and width of values (20, 30) is not consistent with the␣
↪length of this :obj:`PixelData` (20, 40)
```

```
AssertionError: the height and width of values (20, 30) is not consistent with the␣
↪length of this :obj:`PixelData` (20, 40)
```

```
pixel_data.map2 = torch.randint(0, 255, (1, 3, 20, 40))
# AssertionError: The dim of value must be 2 or 3, but got 4
```

```
AssertionError: The dim of value must be 2 or 3, but got 4
```

14.2.5 空间维度索引

PixelData 支持对 data 部分的数据对实例进行空间维度的索引和切片，只需传入长宽的索引即可。

```
metainfo = dict(
    img_id=random.randint(0, 100),
    img_shape=(random.randint(400, 600), random.randint(400, 600)))
image = np.random.randint(0, 255, (4, 20, 40))
featmap = torch.randint(0, 255, (10, 20, 40))
pixel_data = PixelData(metainfo=metainfo,
                       image=image,
                       featmap=featmap)
print('The shape of pixel_data is', pixel_data.shape)
```

```
The shape of pixel_data is (20, 40)
```

- 索引

```
index_data = pixel_data[10, 20]
print('The shape of index_data is', index_data.shape)
```

```
The shape of index_data is (1, 1)
```

- 切片

```
slice_data = pixel_data[10:20, 20:40]
print('The shape of slice_data is', slice_data.shape)
```

```
The shape of slice_data is (10, 20)
```

14.2.6 LabelData

LabelData 主要用来封装标签数据，如场景分类标签，文字识别标签等。LabelData 没有对 data 做任何限制，只提供了两个额外功能：onehot 与 index 的转换。

```
from mmengine.structures import LabelData
import torch

item = torch.tensor([1], dtype=torch.int64)
num_classes = 10
```

```
onehot = LabelData.label_to_onehot(label=item, num_classes=num_classes)
print(f'{num_classes} is convert to ', onehot)

index = LabelData.onehot_to_label(onehot=onehot)
print(f'{onehot} is convert to ', index)
```

```
10 is convert to  tensor([0, 1, 0, 0, 0, 0, 0, 0, 0, 0])
tensor([0, 1, 0, 0, 0, 0, 0, 0, 0, 0]) is convert to tensor([1])
```

14.3 数据样本 (DataSample)

数据样本作为不同模块最外层的接口，提供了 xxxDataSample 用于单任务中各模块之间统一格式的传递，同时为了各个模块从统一字段获取或写入信息，数据样本中的命名以及类型要进行约束和统一，保证各模块接口的统一性。OpenMMLab 中各个算法库的命名规范可以参考 OpenMMLab 中的命名规范。

14.3.1 下游库使用

以 MMDet 为例，说明下游库中数据样本的使用，以及数据样本字段的约束和命名。MMDet 中定义了 DetDataSample, 同时定义了 7 个字段，分别为：

- 标注信息
  - gt_instance(InstanceData): 实例标注信息，包括实例的类别、边界框等，类型约束为 InstanceData。
  - gt_panoptic_seg(PixelData): 全景分割的标注信息，类型约束为 PixelData。
  - gt_semantic_seg(PixelData): 语义分割的标注信息，类型约束为 PixelData。
- 预测结果
  - pred_instance(InstanceData): 实例预测结果，包括实例的类别、边界框等，类型约束为 InstanceData。
  - pred_panoptic_seg(PixelData): 全景分割的预测结果，类型约束为 PixelData。
  - pred_semantic_seg(PixelData): 语义分割的预测结果，类型约束为 PixelData。
- 中间结果
  - proposal(InstanceData): 主要为二阶段中 RPN 的预测结果，类型约束为 InstanceData。

```
from mmengine.structures import BaseDataElement
import torch

class DetDataSample(BaseDataElement):

    # 标注
    @property
    def gt_instances(self) -> InstanceData:
        return self._gt_instances

    @gt_instances.setter
    def gt_instances(self, value: InstanceData):
        self.set_field(value, '_gt_instances', dtype=InstanceData)

    @gt_instances.deleter
    def gt_instances(self):
        del self._gt_instances

    @property
    def gt_panoptic_seg(self) -> PixelData:
        return self._gt_panoptic_seg

    @gt_panoptic_seg.setter
```

(下页继续)

(续上页)

```
    def gt_panoptic_seg(self, value: PixelData):
        self.set_field(value, '_gt_panoptic_seg', dtype=PixelData)

    @gt_panoptic_seg.deleter
    def gt_panoptic_seg(self):
        del self._gt_panoptic_seg

    @property
    def gt_sem_seg(self) -> PixelData:
        return self._gt_sem_seg

    @gt_sem_seg.setter
    def gt_sem_seg(self, value: PixelData):
        self.set_field(value, '_gt_sem_seg', dtype=PixelData)

    @gt_sem_seg.deleter
    def gt_sem_seg(self):
        del self._gt_sem_seg

    # 预测
    @property
    def pred_instances(self) -> InstanceData:
        return self._pred_instances

    @pred_instances.setter
    def pred_instances(self, value: InstanceData):
        self.set_field(value, '_pred_instances', dtype=InstanceData)

    @pred_instances.deleter
    def pred_instances(self):
        del self._pred_instances

    @property
    def pred_panoptic_seg(self) -> PixelData:
        return self._pred_panoptic_seg

    @pred_panoptic_seg.setter
    def pred_panoptic_seg(self, value: PixelData):
        self.set_field(value, '_pred_panoptic_seg', dtype=PixelData)

    @pred_panoptic_seg.deleter
    def pred_panoptic_seg(self):
        del self._pred_panoptic_seg
```

(下页继续)

(续上页)

```
    # 中间结果
    @property
    def pred_sem_seg(self) -> PixelData:
        return self._pred_sem_seg

    @pred_sem_seg.setter
    def pred_sem_seg(self, value: PixelData):
        self.set_field(value, '_pred_sem_seg', dtype=PixelData)

    @pred_sem_seg.deleter
    def pred_sem_seg(self):
        del self._pred_sem_seg

    @property
    def proposals(self) -> InstanceData:
        return self._proposals

    @proposals.setter
    def proposals(self, value: InstanceData):
        self.set_field(value, '_proposals', dtype=InstanceData)

    @proposals.deleter
    def proposals(self):
        del self._proposals
```

14.3.2 类型约束

DetDataSample 的用法如下所示，在数据类型不符合要求的时候 (例如用 torch.Tensor 而非 InstanceData 定义 proposals 时)，DetDataSample 就会报错。

```
data_sample = DetDataSample()

data_sample.proposals = InstanceData(data=dict(bboxes=torch.rand((5,4))))
print(data_sample)
```

```
<DetDataSample(

    META INFORMATION

    DATA FIELDS
```

(下页继续)

(续上页)

```
    proposals: <InstanceData(

            META INFORMATION

            DATA FIELDS
            data:
                bboxes: tensor([[0.7513, 0.9275, 0.6169, 0.5581],
                            [0.6019, 0.6861, 0.7915, 0.0221],
                            [0.5977, 0.8987, 0.9541, 0.7877],
                            [0.0309, 0.1680, 0.1374, 0.0556],
                            [0.3842, 0.9965, 0.0747, 0.6546]])
        ) at 0x7f9f1c090310>
) at 0x7f9f1c090430>
```

```
data_sample.proposals = torch.rand((5, 4))
```

```
AssertionError: tensor([[0.4370, 0.1661, 0.0902, 0.8421],
        [0.4947, 0.1668, 0.0083, 0.1111],
        [0.2041, 0.8663, 0.0563, 0.3279],
        [0.7817, 0.1938, 0.2499, 0.6748],
        [0.4524, 0.8265, 0.4262, 0.2215]]) should be a <class 'mmengine.data.instance_
↪data.InstanceData'> but got <class 'torch.Tensor'>
```

14.4 接口的简化

下面以 MMDetection 为例更具体地说明 OpenMMLab 的算法库将如何迁移使用抽象数据接口，以简化模块和组件接口的。我们假定 MMDetection 和 MMEngine 中实现了 DetDataSample 和 InstanceData。

14.4.1 1. 组件接口的简化

检测器的外部接口可以得到显著的简化和统一。MMDet 2.X 中单阶段检测器和单阶段分割算法的接口如下。在训练过程中，`SingleStageDetector` 需要获取 `img`，`img_metas`，`gt_bboxes`，`gt_labels`，`gt_bboxes_ignore` 作为输入，但是 `SingleStageInstanceSegmentor` 还需要 `gt_masks`，导致 detector 的训练接口不一致，影响了代码的灵活性。

```
class SingleStageDetector(BaseDetector):
    ...

    def forward_train(self,
```

(下页继续)

(续上页)

```
                      img,
                      img_metas,
                      gt_bboxes,
                      gt_labels,
                      gt_bboxes_ignore=None):


class SingleStageInstanceSegmentor(BaseDetector):
    ...

    def forward_train(self,
                      img,
                      img_metas,
                      gt_masks,
                      gt_labels,
                      gt_bboxes=None,
                      gt_bboxes_ignore=None,
                      **kwargs):
```

在 MMDet 3.0 中，所有检测器的训练接口都可以使用 DetDataSample 统一简化为 `img` 和 `data_samples`，不同模块可以根据需要去访问 `data_samples` 封装的各种所需要的属性。

```
class SingleStageDetector(BaseDetector):
    ...

    def forward_train(self,
                      img,
                      data_samples):

class SingleStageInstanceSegmentor(BaseDetector):
    ...

    def forward_train(self,
                      img,
                      data_samples):
```

14.4.2 2. 模块接口的简化

MMDet 2.X 中 HungarianAssigner 和 MaskHungarianAssigner 分别用于在训练过程中将检测框和实例掩码和标注的实例进行匹配。他们内部的匹配逻辑实现是一样的，只是接口和损失函数的计算不同。但是，接口的不同使得 HungarianAssigner 中的代码无法被复用，MaskHungarianAssigner 中重写了很多冗余的逻辑。

```
class HungarianAssigner(BaseAssigner):

    def assign(self,
               bbox_pred,
               cls_pred,
               gt_bboxes,
               gt_labels,
               img_meta,
               gt_bboxes_ignore=None,
               eps=1e-7):

class MaskHungarianAssigner(BaseAssigner):

    def assign(self,
               cls_pred,
               mask_pred,
               gt_labels,
               gt_mask,
               img_meta,
               gt_bboxes_ignore=None,
               eps=1e-7):
```

InstanceData 可以封装实例的框、分数、和掩码，将 HungarianAssigner 的核心参数简化成 pred_instances，gt_instancess，和 gt_instances_ignore 使得 HungarianAssigner 和 MaskHungarianAssigner 可以合并成一个通用的 HungarianAssigner。

```
class HungarianAssigner(BaseAssigner):

    def assign(self,
               pred_instances,
               gt_instancess,
               gt_instances_ignore=None,
               eps=1e-7):
```

CHAPTER 15

可视化

可视化可以给深度学习的模型训练和测试过程提供直观解释。

MMEngine 提供了 `Visualizer` 可视化器用以可视化和存储模型训练和测试过程中的状态以及中间结果，具备如下功能：

- 支持基础绘图接口以及特征图可视化
- 支持本地，TensorBoard 以及 WandB 等多种后端，可以将训练状态例如 loss 、lr 或者性能评估指标以及可视化的结果写入指定的单一或多个后端
- 允许在代码库任意位置调用，对任意位置的特征，图像，状态等进行可视化和存储。

15.1 基础绘制接口

可视化器提供了常用对象的绘制接口，例如绘制**检测框**、**点**、**文本**、**线**、**圆**、**多边形和二值掩码**。这些基础 API 支持以下特性：

- 可以多次调用，实现叠加绘制需求
- 均支持多输入，除了要求文本输入的绘制接口外，其余接口同时支持 Tensor 以及 Numpy array 的输入

常见用法如下：

(1) 绘制检测框、掩码和文本等

```
import torch
import mmcv
```

(下页继续)

(续上页)

```
from mmengine.visualization import Visualizer

image = mmcv.imread('docs/en/_static/image/cat_dog.png', channel_order='rgb')
visualizer = Visualizer(image=image)
# 绘制单个检测框, xyxy 格式
visualizer.draw_bboxes(torch.tensor([72, 13, 179, 147]))
# 绘制多个检测框
visualizer.draw_bboxes(torch.tensor([[33, 120, 209, 220], [72, 13, 179, 147]]))
visualizer.show()
```

```
visualizer.set_image(image=image)
visualizer.draw_texts("cat and dog", torch.tensor([10, 20]))
visualizer.show()
```

你也可以通过通过各个绘制接口中提供的参数来定制绘制对象的颜色和宽度等等

```
visualizer.set_image(image=image)
visualizer.draw_bboxes(torch.tensor([72, 13, 179, 147]), edge_colors='r', line_
↪widths=3)
visualizer.draw_bboxes(torch.tensor([[33, 120, 209, 220]]),line_styles='--')
visualizer.show()
```

(2) 叠加显示

上述绘制接口可以多次调用，从而实现叠加显示需求

```
visualizer.set_image(image=image)
visualizer.draw_bboxes(torch.tensor([[33, 120, 209, 220], [72, 13, 179, 147]]))
visualizer.draw_texts("cat and dog",
                      torch.tensor([10, 20])).draw_circles(torch.tensor([40, 50]),␣
↪torch.tensor([20]))
visualizer.show()
```

15.2 特征图绘制

特征图可视化功能较多，目前只支持单张特征图的可视化，为了方便理解，将其对外接口梳理如下：

```
@staticmethod
def draw_featmap(featmap: torch.Tensor, # 输入格式要求为 CHW
                 overlaid_image: Optional[np.ndarray] = None, # 如果同时输入了 image 数据,
则特征图会叠加到 image 上绘制
                 channel_reduction: Optional[str] = 'squeeze_mean', # 多个通道压缩为单通道
的策略
```

(下页继续)

(续上页)

```
                topk: int = 10, # 可选择激活度最高的 topk 个特征图显示
                arrangement: Tuple[int, int] = (5, 2), # 多通道展开为多张图时候布局
                resize_shape: Optional[tuple] = None, # 可以指定 resize_shape 参数来缩放
特征图
                alpha: float = 0.5) -> np.ndarray: # 图片和特征图绘制的叠加比例
```

其功能可以归纳如下

- 输入的 Tensor 一般是包括多个通道的，channel_reduction 参数可以将多个通道压缩为单通道，然后和图片进行叠加显示
  - `squeeze_mean` 将输入的 C 维度采用 mean 函数压缩为一个通道，输出维度变成 (1, H, W)
  - `select_max` 从输入的 C 维度中先在空间维度 sum，维度变成 (C,)，然后选择值最大的通道
  - `None` 表示不需要压缩，此时可以通过 topk 参数可选择激活度最高的 topk 个特征图显示
- 在 channel_reduction 参数为 None 的情况下，topk 参数生效，其会按照激活度排序选择 topk 个通道，然后和图片进行叠加显示，并且此时会通过 arrangement 参数指定显示的布局
  - 如果 topk 不是 -1，则会按照激活度排序选择 topk 个通道显示
  - 如果 topk = -1，此时通道 C 必须是 1 或者 3 表示输入数据是图片，否则报错提示用户应该设置 `channel_reduction` 来压缩通道。
- 考虑到输入的特征图通常非常小，函数支持输入 `resize_shape` 参数，方便将特征图进行上采样后进行可视化。

常见用法如下：以预训练好的 ResNet18 模型为例，通过提取 layer4 层输出进行特征图可视化

(1) 将多通道特征图采用 `select_max` 参数压缩为单通道并显示

```python
import numpy as np
from torchvision.models import resnet18
from torchvision.transforms import Compose, Normalize, ToTensor

def preprocess_image(img, mean, std):
    preprocessing = Compose([
        ToTensor(),
        Normalize(mean=mean, std=std)
    ])
    return preprocessing(img.copy()).unsqueeze(0)

model = resnet18(pretrained=True)

def _forward(x):
    x = model.conv1(x)
    x = model.bn1(x)
```

(下页继续)

(续上页)

```
    x = model.relu(x)
    x = model.maxpool(x)

    x1 = model.layer1(x)
    x2 = model.layer2(x1)
    x3 = model.layer3(x2)
    x4 = model.layer4(x3)
    return x4

model.forward = _forward

image_norm = np.float32(image) / 255
input_tensor = preprocess_image(image_norm,
                                mean=[0.485, 0.456, 0.406],
                                std=[0.229, 0.224, 0.225])
feat = model(input_tensor)[0]

visualizer = Visualizer()
drawn_img = visualizer.draw_featmap(feat, channel_reduction='select_max')
visualizer.show(drawn_img)
```

由于输出的 feat 特征图尺寸为 7x7，直接可视化效果不佳，用户可以通过叠加输入图片或者 `resize_shape` 参数来缩放特征图。如果传入图片尺寸和特征图大小不一致，会强制将特征图采样到和输入图片相同空间尺寸

```
drawn_img = visualizer.draw_featmap(feat, image, channel_reduction='select_max')
visualizer.show(drawn_img)
```

(2) 利用 `topk=5` 参数选择多通道特征图中激活度最高的 5 个通道并采用 2x3 布局显示

```
drawn_img = visualizer.draw_featmap(feat, image, channel_reduction=None, topk=5,␣
↪arrangement=(2, 3))
visualizer.show(drawn_img)
```

用户可以通过 `arrangement` 参数选择自己想要的布局

```
drawn_img = visualizer.draw_featmap(feat, image, channel_reduction=None, topk=5,␣
↪arrangement=(4, 2))
visualizer.show(drawn_img)
```

15.3 基础存储接口

在绘制完成后，可以选择本地窗口显示，也可以存储到不同后端中，目前 MMEngine 内置了本地存储、Tensorboard 存储和 WandB 存储 3 个后端，且支持存储绘制后的图片、loss 等标量数据和配置文件。

(1) 存储绘制后的图片

假设存储后端为本地存储

```
visualizer = Visualizer(image=image, vis_backends=[dict(type='LocalVisBackend')],␣
↪save_dir='temp_dir')

visualizer.draw_bboxes(torch.tensor([[33, 120, 209, 220], [72, 13, 179, 147]]))
visualizer.draw_texts("cat and dog", torch.tensor([10, 20]))
visualizer.draw_circles(torch.tensor([40, 50]), torch.tensor([20]))

# 会生成 temp_dir/vis_data/vis_image/demo_0.png
visualizer.add_image('demo', visualizer.get_image())
```

其中生成的后缀 0 是用来区分不同 step 场景

```
# 会生成 temp_dir/vis_data/vis_image/demo_1.png
visualizer.add_image('demo', visualizer.get_image(), step=1)
# 会生成 temp_dir/vis_data/vis_image/demo_3.png
visualizer.add_image('demo', visualizer.get_image(), step=3)
```

如果想使用其他后端，则只需要修改配置文件即可

```
# TensorboardVisBackend
visualizer = Visualizer(image=image, vis_backends=[dict(type='TensorboardVisBackend
↪')], save_dir='temp_dir')
# 或者 WandbVisBackend
visualizer = Visualizer(image=image, vis_backends=[dict(type='WandbVisBackend')],␣
↪save_dir='temp_dir')
```

(2) 存储特征图

```
visualizer = Visualizer(vis_backends=[dict(type='LocalVisBackend')], save_dir='temp_
↪dir')
drawn_img = visualizer.draw_featmap(feat, image, channel_reduction=None, topk=5,␣
↪arrangement=(2, 3))
# 会生成 temp_dir/vis_data/vis_image/feat_0.png
visualizer.add_image('feat', drawn_img)
```

(3) 存储 loss 等标量数据

```
# 会生成 temp_dir/vis_data/scalars.json
# 保存 loss
visualizer.add_scalar('loss', 0.2, step=0)
visualizer.add_scalar('loss', 0.1, step=1)
# 保存 acc
visualizer.add_scalar('acc', 0.7, step=0)
visualizer.add_scalar('acc', 0.8, step=1)
```

也可以一次性保存多个标量数据

```
# 会将内容追加到 temp_dir/vis_data/scalars.json
visualizer.add_scalars({'loss': 0.3, 'acc': 0.8}, step=3)
```

(4) 保存配置文件

```
from mmengine import Config
cfg=Config.fromfile('tests/data/config/py_config/config.py')
# 会生成 temp_dir/vis_data/config.py
visualizer.add_config(cfg)
```

15.4 多后端存储

实际上，任何一个可视化器都可以配置任意多个存储后端，可视化器会循环调用配置好的多个存储后端，从而将结果保存到多后端中。

```
visualizer = Visualizer(image=image, vis_backends=[dict(type='TensorboardVisBackend'),
                                                    dict(type='LocalVisBackend')],
                        save_dir='temp_dir')
# 会生成 temp_dir/vis_data/events.out.tfevents.xxx 文件
visualizer.draw_bboxes(torch.tensor([[33, 120, 209, 220], [72, 13, 179, 147]]))
visualizer.draw_texts("cat and dog", torch.tensor([10, 20]))
visualizer.draw_circles(torch.tensor([40, 50]), torch.tensor([20]))

visualizer.add_image('demo', visualizer.get_image())
```

注意：如果多个存储后端中存在同一个类的多个后端，那么必须指定 name 字段，否则无法区分是哪个存储后端

```
visualizer = Visualizer(image=image, vis_backends=[dict(type='TensorboardVisBackend',␣
↪name='tb_1', save_dir='temp_dir_1'),
                                                    dict(type='TensorboardVisBackend',␣
↪name='tb_2', save_dir='temp_dir_2'),
                                                    dict(type='LocalVisBackend', name=
↪'local')],
```

(下页继续)

(续上页)

```
                        save_dir='temp_dir')
```

15.5 任意点位进行可视化

在深度学习过程中，会存在在某些代码位置插入可视化函数，并将其保存到不同后端的需求，这类需求主要用于可视化分析和调试阶段。MMEngine 设计的可视化器支持在任意点位获取同一个可视化器然后进行可视化的功能。用户只需要在初始化时候通过 `get_instance` 接口实例化可视化对象，此时该可视化对象即为全局可获取唯一对象，后续通过 `Visualizer.get_current_instance()` 即可在代码任意位置获取。

```
# 在程序初始化时候调用
visualizer1 = Visualizer.get_instance(name='vis', vis_backends=[dict(type=
↪'LocalVisBackend')])

# 在任何代码位置都可调用
visualizer2 = Visualizer.get_current_instance()
visualizer2.add_scalar('map', 0.7, step=0)

assert id(visualizer1) == id(visualizer2)
```

也可以通过字段配置方式全局初始化

```
from mmengine.registry import VISUALIZERS

visualizer_cfg=dict(
                type='Visualizer',
                name='vis_new',
                vis_backends=[dict(type='LocalVisBackend')])
VISUALIZERS.build(visualizer_cfg)
```

15.6 扩展存储后端和可视化器

(1) 调用特定存储后端

目前存储后端仅仅提供了保存配置、保存标量等基本功能，但是由于 WandB 和 Tensorboard 这类存储后端功能非常强大，用户可能会希望利用到这类存储后端的其他功能。因此，存储后端提供了 `experiment` 属性来方便用户获取后端对象，满足各类定制化功能。例如 WandB 提供了表格显示的 API 接口，用户可以通过 `experiment` 属性获取 WandB 对象，然后调用特定的 API 来将自定义数据保存为表格显示

```
visualizer = Visualizer(image=image, vis_backends=[dict(type='WandbVisBackend')],
                        save_dir='temp_dir')
```

(下页继续)

(续上页)

```
# 获取 wandb 对象
wandb = visualizer.get_backend('WandbVisBackend').experiment
# 追加表格数据
table = wandb.Table(columns=["step", "mAP"])
table.add_data(1, 0.2)
table.add_data(2, 0.5)
table.add_data(3, 0.9)
# 保存
wandb.log({"table": table})
```

(2) 扩展存储后端

用户可以方便快捷的扩展存储后端。只需要继承自 `BaseVisBackend` 并实现各类 `add_xx` 方法即可

```
from mmengine.registry import VISBACKENDS
from mmengine.visualization import BaseVisBackend

@VISBACKENDS.register_module()
class DemoVisBackend(BaseVisBackend):
    def add_image(self, **kwargs):
        pass

visualizer = Visualizer(vis_backends=[dict(type='DemoVisBackend')], save_dir='temp_dir
↪')
visualizer.add_image('demo',image)
```

(3) 扩展可视化器

同样的，用户可以通过继承 Visualizer 并实现想覆写的函数来方便快捷的扩展可视化器。大部分情况下，用户需要覆写 `add_datasample` 来进行拓展。数据中通常包括标注或模型预测的检测框和实例掩码，该接口为各个下游库绘制 datasample 数据的抽象接口。以 MMDetection 为例，datasample 数据中通常包括标注 bbox、标注 mask 、预测 bbox 或者预测 mask 等数据，MMDetection 会继承 Visualizer 并实现 `add_datasample` 接口，在该接口内部会针对检测任务相关数据进行可视化绘制，从而简化检测任务可视化需求。

```
from mmengine.registry import VISUALIZERS

@VISUALIZERS.register_module()
class DetLocalVisualizer(Visualizer):
    def add_datasample(self,
                       name,
                       image: np.ndarray,
                       data_sample: Optional['BaseDataElement'] = None,
                       draw_gt: bool = True,
```

(下页继续)

(续上页)

```
                        draw_pred: bool = True,
                        show: bool = False,
                        wait_time: int = 0,
                        step: int = 0) -> None:
        pass

visualizer_cfg = dict(
    type='DetLocalVisualizer', vis_backends=[dict(type='WandbVisBackend')], name=
↪'visualizer')

# 全局初始化
VISUALIZERS.build(visualizer_cfg)

# 任意代码位置
det_local_visualizer = Visualizer.get_current_instance()
det_local_visualizer.add_datasample('det', image, data_sample)
```

CHAPTER 16

初始化

基于 Pytorch 构建模型时，我们通常会选择 nn.Module 作为模型的基类，搭配使用 Pytorch 的初始化模块 torch.nn.init，完成模型的初始化。MMEngine 在此基础上抽象出基础模块（BaseModule），让我们能够通过传参或配置文件来选择模型的初始化方式。此外，MMEngine 还提供了一系列模块初始化函数，让我们能够更加方便灵活地初始化模型参数。

16.1 配置式初始化

为了能够更加灵活地初始化模型权重，MMEngine 抽象出了模块基类 BaseModule。模块基类继承自 nn.Module，在具备 nn.Module 基础功能的同时，还支持在构造时接受参数，以此来选择权重初始化方式。继承自 BaseModule 的模型可以在实例化阶段接受 init_cfg 参数，我们可以通过配置 init_cfg 为模型中任意组件灵活地选择初始化方式。目前我们可以在 init_cfg 中配置以下初始化器：

我们通过几个例子来理解如何在 init_cfg 里配置初始化器，来选择模型的初始化方式。

16.1.1 使用预训练权重初始化

假设我们定义了模型类 ToyNet，它继承自模块基类（BaseModule）。此时我们可以在 ToyNet 初始化时传入 init_cfg 参数来选择模型的初始化方式，实例化后再调用 init_weights 方法，完成权重的初始化。以加载预训练权重为例：

```
import torch
import torch.nn as nn
```

（下页继续）

(续上页)

```
from mmengine.model import BaseModule


class ToyNet(BaseModule):

    def __init__(self, init_cfg=None):
        super().__init__(init_cfg)
        self.conv1 = nn.Linear(1, 1)


# 保存预训练权重
toy_net = ToyNet()
torch.save(toy_net.state_dict(), './pretrained.pth')
pretrained = './pretrained.pth'

# 配置加载预训练权重的初始化方式
toy_net = ToyNet(init_cfg=dict(type='Pretrained', checkpoint=pretrained))
# 加载权重
toy_net.init_weights()
```

```
08/19 16:50:24 - mmengine - INFO - load model from: ./pretrained.pth
08/19 16:50:24 - mmengine - INFO - local loads checkpoint from path: ./pretrained.pth
```

当 init_cfg 是一个字典时，type 字段就表示一种初始化器，它需要被注册到 WEIGHT_INITIALIZERS 注册器。我们可以通过指定 init_cfg=dict(type='Pretrained', checkpoint='path/to/ckpt') 来加载预训练权重，其中 Pretrained 为 PretrainedInit 初始化器的缩写，这个映射名由 WEIGHT_INITIALIZERS 维护；checkpoint 是 PretrainedInit 的初始化参数，用于指定权重的加载路径，它可以是本地磁盘路径，也可以是 URL。

16.1.2 常用的初始化方式

和使用 PretrainedInit 初始化器类似，如果我们想对卷积做 Kaiming 初始化，需要令 init_cfg=dict(type='Kaiming', layer='Conv2d')。这样模型初始化时，就会以 Kaiming 初始化的方式来初始化类型为 Conv2d 的模块。

有时候我们可能需要用不同的初始化方式去初始化不同类型的模块，例如对卷积使用 Kaiming 初始化，对线性层使用 Xavier 初始化。此时我们可以使 init_cfg 成为一个列表，，其中的每一个元素都表示对某些层使用特定的初始化方式。

```
import torch.nn as nn
```

(下页继续)

(续上页)

```
from mmengine.model import BaseModule


class ToyNet(BaseModule):

    def __init__(self, init_cfg=None):
        super().__init__(init_cfg)
        self.linear = nn.Linear(1, 1)
        self.conv = nn.Conv2d(1, 1, 1)


# 对卷积做 Kaiming 初始化，线性层做 Xavier 初始化
toy_net = ToyNet(
    init_cfg=[
        dict(type='Kaiming', layer='Conv2d'),
        dict(type='Xavier', layer='Linear')
    ], )
toy_net.init_weights()
```

```
08/19 16:50:24 - mmengine - INFO -
linear.weight - torch.Size([1, 1]):
XavierInit: gain=1, distribution=normal, bias=0

08/19 16:50:24 - mmengine - INFO -
linear.bias - torch.Size([1]):
XavierInit: gain=1, distribution=normal, bias=0

08/19 16:50:24 - mmengine - INFO -
conv.weight - torch.Size([1, 1, 1, 1]):
KaimingInit: a=0, mode=fan_out, nonlinearity=relu, distribution =normal, bias=0

08/19 16:50:24 - mmengine - INFO -
conv.bias - torch.Size([1]):
KaimingInit: a=0, mode=fan_out, nonlinearity=relu, distribution =normal, bias=0
```

类似地，`layer` 参数也可以是一个列表，表示列表中的多种不同的 `layer` 均使用 `type` 指定的初始化方式

```
# 对卷积和线性层做 Kaiming 初始化
toy_net = ToyNet(init_cfg=[dict(type='Kaiming', layer=['Conv2d', 'Linear'])], )
toy_net.init_weights()
```

```
08/19 16:50:24 - mmengine - INFO -
linear.weight - torch.Size([1, 1]):
```

(下页继续)

(续上页)

```
KaimingInit: a=0, mode=fan_out, nonlinearity=relu, distribution =normal, bias=0

08/19 16:50:24 - mmengine - INFO -
linear.bias - torch.Size([1]):
KaimingInit: a=0, mode=fan_out, nonlinearity=relu, distribution =normal, bias=0

08/19 16:50:24 - mmengine - INFO -
conv.weight - torch.Size([1, 1, 1, 1]):
KaimingInit: a=0, mode=fan_out, nonlinearity=relu, distribution =normal, bias=0

08/19 16:50:24 - mmengine - INFO -
conv.bias - torch.Size([1]):
KaimingInit: a=0, mode=fan_out, nonlinearity=relu, distribution =normal, bias=0
```

更细粒度的初始化

有时同一类型的不同模块有不同初始化方式，例如现在有 `conv1` 和 `conv2` 两个模块，他们的类型均为 `Conv2d`。我们需要对 conv1 进行 `Kaiming` 初始化，conv2 进行 `Xavier` 初始化，则可以通过配置 `override` 参数来满足这样的需求：

```python
import torch.nn as nn

from mmengine.model import BaseModule


class ToyNet(BaseModule):

    def __init__(self, init_cfg=None):
        super().__init__(init_cfg)
        self.conv1 = nn.Conv2d(1, 1, 1)
        self.conv2 = nn.Conv2d(1, 1, 1)


# 对 conv1 做卷积初始化，对 从 conv2 做 Xavier 初始化
toy_net = ToyNet(
    init_cfg=[
        dict(
            type='Kaiming',
            layer=['Conv2d'],
            override=dict(name='conv2', type='Xavier')),
    ], )
toy_net.init_weights()
```

```
08/19 16:50:24 - mmengine - INFO -
conv1.weight - torch.Size([1, 1, 1, 1]):
KaimingInit: a=0, mode=fan_out, nonlinearity=relu, distribution =normal, bias=0

08/19 16:50:24 - mmengine - INFO -
conv1.bias - torch.Size([1]):
KaimingInit: a=0, mode=fan_out, nonlinearity=relu, distribution =normal, bias=0

08/19 16:50:24 - mmengine - INFO -
conv2.weight - torch.Size([1, 1, 1, 1]):
XavierInit: gain=1, distribution=normal, bias=0

08/19 16:50:24 - mmengine - INFO -
conv2.bias - torch.Size([1]):
KaimingInit: a=0, mode=fan_out, nonlinearity=relu, distribution =normal, bias=0
```

override 可以理解成一个嵌套的 init_cfg，他同样可以是 list 或者 dict，也需要通过 type 字段指定初始化方式。不同的是 override 必须指定 name，name 相当于 override 的作用域，如上例中，override 的作用域为 toy_net.conv2，我们会以 Xavier 初始化方式初始化 toy_net.conv2 下的所有参数，而不会影响作用域以外的模块。

16.1.3 自定义的初始化方式

尽管 init_cfg 能够控制各个模块的初始化方式，但是在不扩展 WEIGHT_INITIALIZERS 的情况下，我们是无法初始化一些自定义模块的，例如表格中提到的大多数初始化器，都需要对应的模块有 weight 和 bias 属性。对于这种情况，我们建议让自定义模块实现 init_weights 方法。模型调用 init_weights 时，会链式地调用所有子模块的 init_weights。

假设我们定义了以下模块：

- 继承自 nn.Module 的 ToyConv，实现了 init_weights 方法，让 custom_weight 初始化为 1，custom_bias 初始化为 0
- 继承自模块基类的模型 ToyNet，且含有 ToyConv 子模块。

我们在调用 ToyNet 的 init_weights 方法时，会链式的调用的子模块 ToyConv 的 init_weights 方法，实现自定义模块的初始化。

```
import torch
import torch.nn as nn

from mmengine.model import BaseModule


class ToyConv(nn.Module):
```

(下页继续)

(续上页)

```
    def __init__(self):
        super().__init__()
        self.custom_weight = nn.Parameter(torch.empty(1, 1, 1, 1))
        self.custom_bias = nn.Parameter(torch.empty(1))

    def init_weights(self):
        with torch.no_grad():
            self.custom_weight = self.custom_weight.fill_(1)
            self.custom_bias = self.custom_bias.fill_(0)


class ToyNet(BaseModule):

    def __init__(self, init_cfg=None):
        super().__init__(init_cfg)
        self.conv1 = nn.Conv2d(1, 1, 1)
        self.conv2 = nn.Conv2d(1, 1, 1)
        self.custom_conv = ToyConv()


toy_net = ToyNet(
    init_cfg=[
        dict(
            type='Kaiming',
            layer=['Conv2d'],
            override=dict(name='conv2', type='Xavier'))
    ])
# 链式调用 `ToyConv.init_weights()`，以自定义的方式初始化
toy_net.init_weights()
```

```
08/19 16:50:24 - mmengine - INFO -
conv1.weight - torch.Size([1, 1, 1, 1]):
KaimingInit: a=0, mode=fan_out, nonlinearity=relu, distribution =normal, bias=0

08/19 16:50:24 - mmengine - INFO -
conv1.bias - torch.Size([1]):
KaimingInit: a=0, mode=fan_out, nonlinearity=relu, distribution =normal, bias=0

08/19 16:50:24 - mmengine - INFO -
conv2.weight - torch.Size([1, 1, 1, 1]):
XavierInit: gain=1, distribution=normal, bias=0
```

(下页继续)

(续上页)

```
08/19 16:50:24 - mmengine - INFO -
conv2.bias - torch.Size([1]):
KaimingInit: a=0, mode=fan_out, nonlinearity=relu, distribution =normal, bias=0

08/19 16:50:24 - mmengine - INFO -
custom_conv.custom_weight - torch.Size([1, 1, 1, 1]):
Initialized by user-defined `init_weights` in ToyConv

08/19 16:50:24 - mmengine - INFO -
custom_conv.custom_bias - torch.Size([1]):
Initialized by user-defined `init_weights` in ToyConv
```

16.1.4 小结

最后我们对 `init_cfg` 和 `init_weights` 两种初始化方式做一些总结:

1. 配置 `init_cfg` 控制初始化

- 通常用于初始化一些比较底层的模块，例如卷积、线性层等。如果想通过 `init_cfg` 配置自定义模块的初始化方式，需要将相应的初始化器注册到 `WEIGHT_INITIALIZERS` 里。
- 动态初始化特性，初始化方式随 `init_cfg` 的值改变。

2. 实现 `init_weights` 方法

- 通常用于初始化自定义模块。相比于 `init_cfg` 的自定义初始化，实现 `init_weights` 方法更加简单，无需注册。然而，它的灵活性不及 `init_cfg`，无法动态地指定任意模块的初始化方式。

注解:

- init_weights 的优先级比 `init_cfg` 高
- 执行器会在 train() 函数中调用 init_weights。

16.2 函数式初始化

在自定义的初始化方式一节提到，我们可以在 `init_weights` 里实现自定义的参数初始化逻辑。为了能够更加方便地实现参数初始化，MMEngine 在 `torch.nn.init` 的基础上，提供了一系列**模块初始化函数**来初始化整个模块。例如我们对卷积层的权重（`weight`）进行正态分布初始化，卷积层的偏置（`bias`）进行常数初始化，基于 `torch.nn.init` 的实现如下:

```
from torch.nn.init import normal_, constant_
import torch.nn as nn

model = nn.Conv2d(1, 1, 1)
normal_(model.weight, mean=0, std=0.01)
constant_(model.bias, val=0)
```

```
Parameter containing:
tensor([0.], requires_grad=True)
```

上述流程实际上是卷积正态分布初始化的标准流程，因此 MMEngine 在此基础上做了进一步地简化，实现了一系列常用的**模块**初始化函数。相比 `torch.nn.init`，MMEngine 提供的初始化函数直接接受卷积模块，一行代码能实现同样的初始化逻辑：

```
from mmengine.model import normal_init

normal_init(model, mean=0, std=0.01, bias=0)
```

类似地，我们也可以用 Kaiming 初始化和 Xavier 初始化：

```
from mmengine.model import kaiming_init, xavier_init

kaiming_init(model)
xavier_init(model)
```

目前 MMEngine 提供了以下初始化函数：

CHAPTER 17

分布式通信原语

在分布式训练或测试的过程中，不同进程有时需要根据分布式的环境信息执行不同的代码逻辑，同时不同进程之间也经常会有相互通信的需求，对一些数据进行同步等操作。PyTorch 提供了一套基础的通信原语用于多进程之间张量的通信，基于这套原语，MMEngine 实现了更高层次的通信原语封装以满足更加丰富的需求。基于 MMEngine 的通信原语，算法库中的模块可以

1. 在使用通信原语封装时不显式区分分布式/非分布式环境
2. 进行除 Tensor 以外类型数据的多进程通信
3. 无需了解底层通信后端或框架

这些通信原语封装的接口和功能可以大致归类为如下三种，我们在后续章节中逐个介绍

1. 分布式初始化：`init_dist` 负责初始化执行器的分布式环境
2. 分布式信息获取与控制：包括 `get_world_size` 等函数获取当前的 `rank` 和 `world_size` 等信息
3. 分布式通信接口：包括如 `all_reduce` 等通信函数（collective functions）

17.1 分布式初始化

- *init_dist*：是分布式训练的启动函数，目前支持 pytorch，slurm，MPI 3 种分布式启动方式，同时允许设置通信的后端，默认使用 NCCL。

17.2 分布式信息获取与控制

分布式信息的获取与控制函数没有参数，这些函数兼容非分布式训练的情况，功能如下

- *get_world_size*：获取当前进程组的进程总数，非分布式情况下返回 1
- *get_rank*：获取当前进程对应的全局 rank 数，非分布式情况下返回 0
- *get_backend*：获取当前通信使用的后端，非分布式情况下返回 None
- *get_local_rank*：获取当前进程对应到当前机器的 rank 数，非分布式情况下返回 0
- *get_local_size*：获取当前进程所在机器的总进程数，非分布式情况下返回 0
- *get_dist_info*：获取当前任务的进程总数和当前进程对应到全局的 rank 数，非分布式情况下 word_size = 1，rank = 0
- *is_main_process*：判断是否为 0 号主进程，非分布式情况下返回 True
- *master_only*：函数装饰器，用于修饰只需要全局 0 号进程（rank 0 而不是 local rank 0）执行的函数
- *barrier*：同步所有进程到达相同位置

17.3 分布式通信函数

通信函数（Collective functions），主要用于进程间数据的通信，基于 PyTorch 原生的 all_reduce，all_gather，gather，broadcast 接口，MMEngine 提供了如下接口，兼容非分布式训练的情况，并支持更丰富数据类型的通信。

- *all_reduce*: 对进程间 tensor 进行 AllReduce 操作
- *all_gather*：对进程间 tensor 进行 AllGather 操作
- *gather*：将进程的 tensor 收集到一个目标 rank
- *broadcast*：对某个进程的 tensor 进行广播
- *sync_random_seed*：同步进程之间的随机种子
- *broadcast_object_list*：支持对任意可被 Pickle 序列化的 Python 对象列表进行广播，基于 broadcast 接口实现
- *all_reduce_dict*：对 dict 中的内容进行 all_reduce 操作，基于 broadcast 和 all_reduce 接口实现
- *all_gather_object*：基于 all_gather 实现对任意可以被 Pickle 序列化的 Python 对象进行 all_tather 操作
- *gather_object*：将 group 里每个 rank 中任意可被 Pickle 序列化的 Python 对象 gather 到指定的目标 rank
- *collect_results*：支持基于 CPU 通信或者 GPU 通信对不同进程间的列表数据进行收集

CHAPTER 18

记录日志

执行器（*Runner*）在运行过程中会产生很多日志，例如损失、迭代时间、学习率等。MMEngine 实现了一套灵活的日志系统让我们能够在配置执行器时，选择不同类型日志的统计方式；在代码的任意位置，新增需要被统计的日志。

18.1 灵活的日志统计方式

我们可以通过在构建执行器时候配置日志处理器，来灵活地选择日志统计方式。如果不为执行器配置日志处理器，则会按照日志处理器的默认参数构建实例，效果等价于：

```
log_processor = dict(window_size=10, by_epoch=True, custom_cfg=None, num_digits=4)
```

其输出的日志格式如下：

```
import torch
import torch.nn as nn
from torch.utils.data import DataLoader

from mmengine.runner import Runner
from mmengine.model import BaseModel

train_dataset = [(torch.ones(1, 1), torch.ones(1, 1))] * 50
train_dataloader = DataLoader(train_dataset, batch_size=2)
```

(下页继续)

(续上页)

```
class ToyModel(BaseModel):
    def __init__(self) -> None:
        super().__init__()
        self.linear = nn.Linear(1, 1)

    def forward(self, img, label, mode):
        feat = self.linear(img)
        loss1 = (feat - label).pow(2)
        loss2 = (feat - label).abs()
        return dict(loss1=loss1, loss2=loss2)

runner = Runner(
    model=ToyModel(),
    work_dir='tmp_dir',
    train_dataloader=train_dataloader,
    train_cfg=dict(by_epoch=True, max_epochs=1),
    optim_wrapper=dict(optimizer=dict(type='SGD', lr=0.01))
)
runner.train()
```

```
08/21 02:58:41 - mmengine - INFO - Epoch(train) [1][10/25]  lr: 1.0000e-02  eta:␣
↪0:00:00  time: 0.0019  data_time: 0.0004  loss1: 0.8381  loss2: 0.9007  loss: 1.7388
08/21 02:58:41 - mmengine - INFO - Epoch(train) [1][20/25]  lr: 1.0000e-02  eta:␣
↪0:00:00  time: 0.0029  data_time: 0.0010  loss1: 0.1978  loss2: 0.4312  loss: 0.6290
```

以训练阶段为例，日志处理器默认会按照以下方式统计执行器输出的日志：

- 日志前缀：
  - Epoch 模式（by_epoch=True）：Epoch(train) [{当前 epoch 次数}][{当前迭代次数}/{Dataloader 总长度}]
  - Iter 模式（by_epoch=False）：Iter(train) [{当前迭代次数}/{总迭代次数}]
- 学习率（lr）：统计最近一次迭代，参数更新的学习率
- 时间
  - 迭代时间（time）：最近 window_size（日志处理器参数）次迭代，处理一个 batch 数据（包括数据加载和模型前向推理）的平局时间
  - 数据时间（data_time）：最近 window_size 次迭代，加载一个 batch 数据的平局时间
  - 剩余时间（eta）：根据总迭代次数和历次迭代时间计算出来的总剩余时间，剩余时间随着迭代次数增加逐渐趋于稳定
- 损失：模型前向推理得到的各种字段的损失，默认统计最近 window_size 次迭代的平均损失。

默认情况下，window_size=10，日志处理器会统计最近 10 次迭代，损失、迭代时间、数据时间的均值。

默认情况下，所有日志的有效位数（num_digits 参数）为 4。

默认情况下，输出所有自定义日志最近一次迭代的值。

基于上述规则，代码示例中的日志处理器会输出 loss1 和 loss2 每 10 次迭代的均值。如果我们想统计 loss1 从第一次迭代开始至今的全局均值，可以这样配置：

```
runner = Runner(
    model=ToyModel(),
    work_dir='tmp_dir',
    train_dataloader=train_dataloader,
    train_cfg=dict(by_epoch=True, max_epochs=1),
    optim_wrapper=dict(optimizer=dict(type='SGD', lr=0.01)),
    log_processor=dict(  # 配置日志处理器
        custom_cfg=[
            dict(data_src='loss1',  # 原日志名：loss1
                 method_name='mean',  # 统计方法：均值统计
                 window_size='global')])  # 统计窗口：全局
)
runner.train()
```

```
08/21 02:58:49 - mmengine - INFO - Epoch(train) [1][10/25]  lr: 1.0000e-02  eta:␣
↪0:00:00  time: 0.0026  data_time: 0.0007  loss1: 0.7381  loss2: 0.8446  loss: 1.5827
08/21 02:58:49 - mmengine - INFO - Epoch(train) [1][20/25]  lr: 1.0000e-02  eta:␣
↪0:00:00  time: 0.0030  data_time: 0.0012  loss1: 0.4521  loss2: 0.3939  loss: 0.5600
```

注解： log_processor 默认输出 by_epoch=True 格式的日志。日志格式需要和 train_cfg 中的 by_epoch 参数保持一致，例如我们想按迭代次数输出日志，就需要另 log_processor 和 train_cfg 的 by_epoch=False。

其中 data_src 为原日志名，mean 为统计方法，global 为统计方法的参数。这样的话，日志中统计的 loss1 就是全局均值。我们可以在日志处理器中配置以下统计方法：

其中 window_size 的值可以是：

- 数字：表示统计窗口的大小
- global：统计全局的最大、最小和均值
- epoch：统计一个 epoch 内的最大、最小和均值

当然我们也可以选择自定义的统计方法，详细步骤见日志设计。

如果我们既想统计窗口为 10 的 loss1 的局部均值，又想统计 loss1 的全局均值，则需要额外指定 log_name：

```
runner = Runner(
    model=ToyModel(),
    work_dir='tmp_dir',
    train_dataloader=train_dataloader,
    train_cfg=dict(by_epoch=True, max_epochs=1),
    optim_wrapper=dict(optimizer=dict(type='SGD', lr=0.01)),
    log_processor=dict(
        custom_cfg=[
            # log_name 表示 loss1 重新统计后的日志名
            dict(data_src='loss1', log_name='loss1_global', method_name='mean',␣
↪window_size='global')])
)
runner.train()
```

```
08/21 18:39:32 - mmengine - INFO - Epoch(train) [1][10/25]  lr: 1.0000e-02  eta:␣
↪0:00:00  time: 0.0016  data_time: 0.0004  loss1: 0.1512  loss2: 0.3751  loss: 0.
↪5264  loss1_global: 0.1512
08/21 18:39:32 - mmengine - INFO - Epoch(train) [1][20/25]  lr: 1.0000e-02  eta:␣
↪0:00:00  time: 0.0051  data_time: 0.0036  loss1: 0.0113  loss2: 0.0856  loss: 0.
↪0970  loss1_global: 0.0813
```

类似地，我们也可以统计 loss1 的局部最大值和全局最大值：

```
runner = Runner(
    model=ToyModel(),
    work_dir='tmp_dir',
    train_dataloader=train_dataloader,
    train_cfg=dict(by_epoch=True, max_epochs=1),
    optim_wrapper=dict(optimizer=dict(type='SGD', lr=0.01)),
    log_processor=dict(custom_cfg=[
        # 统计 loss1 的局部最大值，统计窗口为 10，并在日志中重命名为 loss1_local_max
        dict(data_src='loss1',
             log_name='loss1_local_max',
             window_size=10,
             method_name='max'),
        # 统计 loss1 的全局最大值，并在日志中重命名为 loss1_local_max
        dict(
            data_src='loss1',
            log_name='loss1_global_max',
            method_name='max',
            window_size='global')
    ]))
runner.train()
```

```
08/21 03:17:26 - mmengine - INFO - Epoch(train) [1][10/25]  lr: 1.0000e-02  eta:␣
↪0:00:00  time: 0.0021  data_time: 0.0006  loss1: 1.8495  loss2: 1.3427  loss: 3.
↪1922  loss1_local_max: 2.8872  loss1_global_max: 2.8872
08/21 03:17:26 - mmengine - INFO - Epoch(train) [1][20/25]  lr: 1.0000e-02  eta:␣
↪0:00:00  time: 0.0024  data_time: 0.0010  loss1: 0.5464  loss2: 0.7251  loss: 1.
↪2715  loss1_local_max: 2.8872  loss1_global_max: 2.8872
```

更多配置规则见日志处理器文档

18.2 自定义统计内容

除了 MMEngine 默认的日志统计类型，如损失、迭代时间、学习率，用户也可以自行添加日志的统计内容。例如我们想统计损失的中间结果，可以这样做：

```
from mmengine.logging import MessageHub


class ToyModel(BaseModel):

    def __init__(self) -> None:
        super().__init__()
        self.linear = nn.Linear(1, 1)

    def forward(self, img, label, mode):
        feat = self.linear(img)
        loss_tmp = (feat - label).abs()
        loss = loss_tmp.pow(2)

        message_hub = MessageHub.get_current_instance()
        # 在日志中额外统计 `loss_tmp`
        message_hub.update_scalar('train/loss_tmp', loss_tmp.sum())
        return dict(loss=loss)


runner = Runner(
    model=ToyModel(),
    work_dir='tmp_dir',
    train_dataloader=train_dataloader,
    train_cfg=dict(by_epoch=True, max_epochs=1),
    optim_wrapper=dict(optimizer=dict(type='SGD', lr=0.01)),
    log_processor=dict(
        custom_cfg=[
        # 统计 loss_tmp 的局部均值
```

(下页继续)

(续上页)

```
            dict(
                data_src='loss_tmp',
                window_size=10,
                method_name='mean')
        ]
    )
)
runner.train()
```

```
08/21 03:40:31 - mmengine - INFO - Epoch(train) [1][10/25]  lr: 1.0000e-02  eta:␣
↪0:00:00  time: 0.0026  data_time: 0.0008  loss_tmp: 0.0097  loss: 0.0000
08/21 03:40:31 - mmengine - INFO - Epoch(train) [1][20/25]  lr: 1.0000e-02  eta:␣
↪0:00:00  time: 0.0028  data_time: 0.0013  loss_tmp: 0.0065  loss: 0.0000
```

通过调用消息枢纽的接口实现自定义日志的统计，具体步骤如下：

1. 调用 `get_current_instance` 接口获取执行器的消息枢纽。
2. 调用 `add_scalar` 接口更新日志内容，其中第一个参数为日志的名称，日志名称以 `train/`，`val/`，`test/` 前缀打头，用于区分训练状态，然后才是实际的日志名，如上例中的 `train/loss_tmp`，这样统计的日志中就会出现 `loss_tmp`。
3. 配置日志处理器，以均值的方式统计 `loss_tmp`。如果不配置，日志里显示 `loss_tmp` 最近一次更新的值。

18.3 输出调试日志

初始化执行器（Runner）时，将 `log_level` 设置成 `debug`。这样终端上就会额外输出日志等级为 `debug` 的日志

```
runner = Runner(
    model=ToyModel(),
    work_dir='tmp_dir',
    train_dataloader=train_dataloader,
    log_level='DEBUG',
    train_cfg=dict(by_epoch=True, max_epochs=1),
    optim_wrapper=dict(optimizer=dict(type='SGD', lr=0.01)))
runner.train()
```

```
08/21 18:16:22 - mmengine - DEBUG - Get class `LocalVisBackend` from "vis_backend"␣
↪registry in "mmengine"
08/21 18:16:22 - mmengine - DEBUG - An `LocalVisBackend` instance is built from␣
↪registry, its implementation can be found in mmengine.visualization.vis_backend
```

(下页继续)

(续上页)

```
08/21 18:16:22 - mmengine - DEBUG - Get class `RuntimeInfoHook` from "hook" registry␣
↪in "mmengine"
08/21 18:16:22 - mmengine - DEBUG - An `RuntimeInfoHook` instance is built from␣
↪registry, its implementation can be found in mmengine.hooks.runtime_info_hook
08/21 18:16:22 - mmengine - DEBUG - Get class `IterTimerHook` from "hook" registry in
↪"mmengine"
...
```

此外，分布式训练时，DEBUG 模式还会分进程存储日志。单机多卡，或者多机多卡但是共享存储的情况下，导出的分布式日志路径如下

```
#  共享存储
./tmp
├── tmp.log
├── tmp_rank1.log
├── tmp_rank2.log
├── tmp_rank3.log
├── tmp_rank4.log
├── tmp_rank5.log
├── tmp_rank6.log
└── tmp_rank7.log
...
└── tmp_rank63.log
```

多机多卡，独立存储的情况：

```
# 独立存储
# 设备 0:
work_dir/
└── exp_name_logs
    ├── exp_name.log
    ├── exp_name_rank1.log
    ├── exp_name_rank2.log
    ├── exp_name_rank3.log
    ...
    └── exp_name_rank7.log

# 设备 7:
work_dir/
└── exp_name_logs
    ├── exp_name_rank56.log
    ├── exp_name_rank57.log
    ├── exp_name_rank58.log
    ...
```

(下页继续)

(续上页)

```
    └── exp_name_rank63.log
```

如果想要更加深入的了解 MMEngine 的日志系统，可以参考日志系统设计。

CHAPTER 19

文件读写

MMEngine 实现了一套统一的文件读写接口，可以用同一个函数来处理不同的文件格式，如 json、yaml 和 pickle，并且可以方便地拓展其它的文件格式。除此之外，文件读写模块还支持从多种文件存储后端读写文件，包括本地磁盘、Petrel（内部使用）、Memcached、LMDB 和 HTTP。

19.1 读取和保存数据

MMEngine 提供了两个通用的接口用于读取和保存数据，目前支持的格式有 json、yaml 和 pickle。

19.1.1 从硬盘读取数据或者将数据保存至硬盘

```python
from mmengine import load, dump

# 从文件中读取数据
data = load('test.json')
data = load('test.yaml')
data = load('test.pkl')
# 从文件对象中读取数据
with open('test.json', 'r') as f:
    data = load(f, file_format='json')

# 将数据序列化为字符串
json_str = dump(data, file_format='json')
```

(下页继续)

(续上页)

```
# 将数据保存至文件 (根据文件名后缀反推文件类型)
dump(data, 'out.pkl')

# 将数据保存至文件对象
with open('test.yaml', 'w') as f:
    data = dump(data, f, file_format='yaml')
```

19.1.2 从其它文件存储后端读写文件

```
from mmengine import load, dump

# 从 s3 文件读取数据
data = load('s3://bucket-name/test.json')
data = load('s3://bucket-name/test.yaml')
data = load('s3://bucket-name/test.pkl')

# 将数据保存至 s3 文件 (根据文件名后缀反推文件类型)
dump(data, 's3://bucket-name/out.pkl')
```

我们提供了易于拓展的方式以支持更多的文件格式，我们只需要创建一个继承自 `BaseFileHandler` 的文件句柄类，句柄类至少需要重写三个方法。然后使用使用 `register_handler` 装饰器将句柄类注册为对应文件格式的读写句柄。

```
from mmengine import register_handler, BaseFileHandler

# 支持为文件句柄类注册多个文件格式
# @register_handler(['txt', 'log'])
@register_handler('txt')
class TxtHandler1(BaseFileHandler):

    def load_from_fileobj(self, file):
        return file.read()

    def dump_to_fileobj(self, obj, file):
        file.write(str(obj))

    def dump_to_str(self, obj, **kwargs):
        return str(obj)
```

以 `PickleHandler` 为例

```python
from mmengine import BaseFileHandler
import pickle

class PickleHandler(BaseFileHandler):

    def load_from_fileobj(self, file, **kwargs):
        return pickle.load(file, **kwargs)

    def load_from_path(self, filepath, **kwargs):
        return super(PickleHandler, self).load_from_path(
            filepath, mode='rb', **kwargs)

    def dump_to_str(self, obj, **kwargs):
        kwargs.setdefault('protocol', 2)
        return pickle.dumps(obj, **kwargs)

    def dump_to_fileobj(self, obj, file, **kwargs):
        kwargs.setdefault('protocol', 2)
        pickle.dump(obj, file, **kwargs)

    def dump_to_path(self, obj, filepath, **kwargs):
        super(PickleHandler, self).dump_to_path(
            obj, filepath, mode='wb', **kwargs)
```

19.2 读取文件并返回列表或字典

例如，a.txt 是文本文件，一共有 5 行内容。

```
a
b
c
d
e
```

19.2.1 从硬盘读取

使用 list_from_file 读取 a.txt

```python
from mmengine import list_from_file

print(list_from_file('a.txt'))
```

(下页继续)

(续上页)

```
# ['a', 'b', 'c', 'd', 'e']
print(list_from_file('a.txt', offset=2))
# ['c', 'd', 'e']
print(list_from_file('a.txt', max_num=2))
# ['a', 'b']
print(list_from_file('a.txt', prefix='/mnt/'))
# ['/mnt/a', '/mnt/b', '/mnt/c', '/mnt/d', '/mnt/e']
```

同样，b.txt 也是文本文件，一共有 3 行内容

```
1 cat
2 dog cow
3 panda
```

使用 dict_from_file 读取 b.txt

```
from mmengine import dict_from_file

print(dict_from_file('b.txt'))
# {'1': 'cat', '2': ['dog', 'cow'], '3': 'panda'}
print(dict_from_file('b.txt', key_type=int))
# {1: 'cat', 2: ['dog', 'cow'], 3: 'panda'}
```

19.2.2 从其他存储后端读取

使用 list_from_file 读取 s3://bucket-name/a.txt

```
from mmengine import list_from_file

print(list_from_file('s3://bucket-name/a.txt'))
# ['a', 'b', 'c', 'd', 'e']
print(list_from_file('s3://bucket-name/a.txt', offset=2))
# ['c', 'd', 'e']
print(list_from_file('s3://bucket-name/a.txt', max_num=2))
# ['a', 'b']
print(list_from_file('s3://bucket-name/a.txt', prefix='/mnt/'))
# ['/mnt/a', '/mnt/b', '/mnt/c', '/mnt/d', '/mnt/e']
```

使用 dict_from_file 读取 b.txt

```
from mmengine import dict_from_file

print(dict_from_file('s3://bucket-name/b.txt'))
```

(下页继续)

(续上页)

```
# {'1': 'cat', '2': ['dog', 'cow'], '3': 'panda'}
print(dict_from_file('s3://bucket-name/b.txt', key_type=int))
# {1: 'cat', 2: ['dog', 'cow'], 3: 'panda'}
```

19.3 读取和保存权重文件

通常情况下，我们可以通过下面的方式从磁盘或者网络远端读取权重文件。

```
import torch

filepath1 = '/path/of/your/checkpoint1.pth'
filepath2 = 'http://path/of/your/checkpoint3.pth'

# 从本地磁盘读取权重文件
checkpoint = torch.load(filepath1)
# 保存权重文件到本地磁盘
torch.save(checkpoint, filepath1)

# 从网络远端读取权重文件
checkpoint = torch.utils.model_zoo.load_url(filepath2)
```

在 mmengine 中，得益于多文件存储后端的支持，不同存储形式的权重文件读写可以通过 load_checkpoint 和 save_checkpoint 来统一实现。

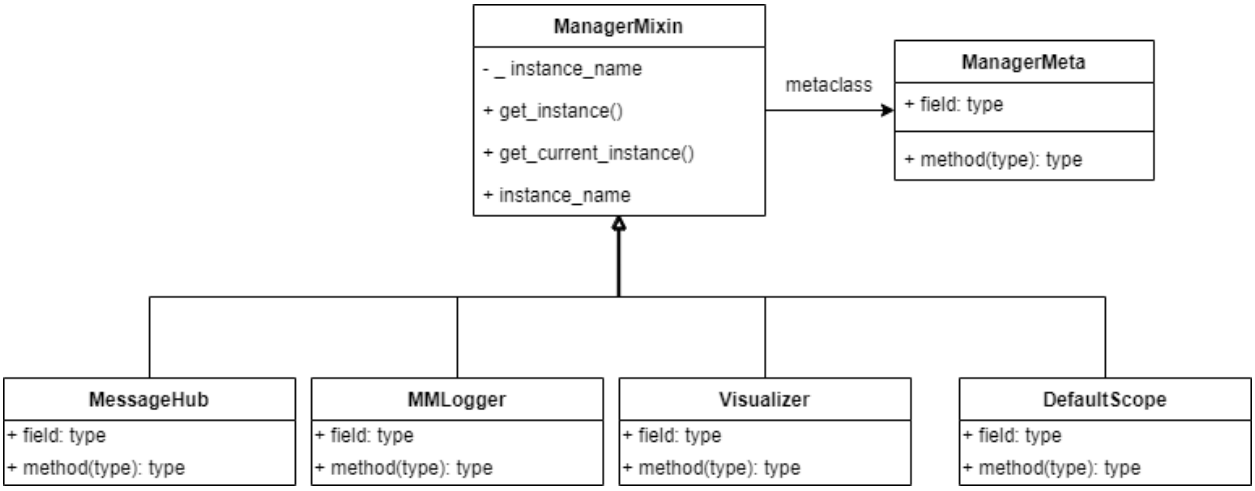
```
from mmengine import load_checkpoint, save_checkpoint

filepath1 = '/path/of/your/checkpoint1.pth'
filepath2 = 's3://bucket-name/path/of/your/checkpoint1.pth'
filepath3 = 'http://path/of/your/checkpoint3.pth'

# 从本地磁盘读取权重文件
checkpoint = load_checkpoint(filepath1)
# 保存权重文件到本地磁盘
save_checkpoint(checkpoint, filepath1)

# 从 s3 读取权重文件
checkpoint = load_checkpoint(filepath2)
# 保存权重文件到 s3
save_checkpoint(checkpoint, filepath2)

# 从网络远端读取权重文件
checkpoint = load_checkpoint(filepath3)
```

CHAPTER 20

辅助类

20.1 全局管理器（ManagerMixin）

Runner 在训练过程中，难免会使用全局变量来共享信息，例如我们会在 model 中获取全局的*logger* 来打印初始化信息；在 model 中获取全局的*Visualizer* 来可视化预测结果、特征图；在*Reigistry* 中获取全局的*DefaultScope* 来确定注册域。为了管理这些功能相似的模块，MMEngine 实现了管理器（ManagerMix）来统一全局变量的创建和获取方式。



20.1.1 接口介绍

- _instance_name：被创建的全局实例名
- get_instance(name='', **kwargs)：创建或者返回对应名字的的实例。
- get_current_instance()：返回最近被创建的实例。
- instance_name:：获取对应实例的 name。

20.1.2 使用方法

1. 定义有全局访问需求的类

```
from mmengine.utils import ManagerMixin


class GlobalClass(ManagerMixin):
    def __init__(self, name, value):
        super().__init__(name)
        self.value = value
```

注意全局类的构造函数必须带有 name 参数，并在构造函数中调用 `super().__init__(name)`，以确保后续能够根据 name 来获取对应的实例。

2. 在任意位置实例化该对象，以 Hook 为例（要确保访问该实例时，对象已经被创建）：

```
from mmengine import Hook

class CustomHook(Hook):
    def before_run(self, runner):
        GlobalClass.get_instance('mmengine', value=50)
        GlobalClass.get_instance(runner.experiment_name, value=100)
```

当我们调用子类的 `get_instance` 接口时，`ManagerMixin` 会根据名字来判断对应实例是否已经存在，进而创建/获取实例。如上例所示，当我们第一次调用 `GlobalClass.get_instance('mmengine', value=50)` 时，会创建一个名为 “mmengine” 的 `GlobalClass` 实例，其初始 value 为 50。为了方便后续介绍 `get_current_instance` 接口，这里我们创建了两个 `GlobalClass` 实例。

3. 在任意组件中访问该实例

```
import torch.nn as nn


class CustomModule(nn.Module):
    def forward(self, x):
```

(下页继续)

(续上页)

```
        value = GlobalClass.get_current_instance().value  # 最近一次被创建的实例 value 为␣
↪100 (步骤二中按顺序创建)
        value = GlobalClass.get_instance('mmengine').value  # 名为 mmengine 的实例␣
↪value 为 50
        # value = GlobalClass.get_instance('mmengine', 1000).value  # mmengine 已经被创
建，不能再接受额外参数
```

在同一进程里，我们可以在不同组件中访问 `GlobalClass` 实例。例如我们在 `CustomModule` 中，调用 `get_instance` 和 `get_current_instance` 接口来获取对应名字的实例和最近被创建的实例。需要注意的是，由于“mmengine”实例已经被创建，再次调用时不能再传入额外参数，否则会报错。

CHAPTER 21

恢复训练

恢复训练是指从之前某次训练保存下来的状态开始继续训练，这里的状态包括模型的权重、优化器和优化器参数调整策略的状态。

21.1 自动恢复训练

用户可以设置 Runner 的 resume 参数开启自动恢复训练的功能。在启动训练时，设置 Runner 的 resume 等于 True，Runner 会从 work_dir 中加载最新的 checkpoint。如果 work_dir 中有最新的 checkpoint（例如该训练在上一次训练时被中断），则会从该 checkpoint 恢复训练，否则（例如上一次训练还没来得及保存 checkpoint 或者启动了新的训练任务）会重新开始训练。下面是一个开启自动恢复训练的示例

```
runner = Runner(
    model=ResNet18(),
    work_dir='./work_dir',
    train_dataloader=train_dataloader_cfg,
    optim_wrapper=dict(optimizer=dict(type='SGD', lr=0.001, momentum=0.9)),
    train_cfg=dict(by_epoch=True, max_epochs=3),
    resume=True,
)
runner.train()
```

21.2 指定 checkpoint 路径

如果希望指定恢复训练的路径，除了设置 `resume=True`，还需要设置 `load_from` 参数。需要注意的是，如果只设置了 `load_from` 而没有设置 `resume=True`，则只会加载 checkpoint 中的权重并重新开始训练，而不是接着之前的状态继续训练。

```
runner = Runner(
    model=ResNet18(),
    work_dir='./work_dir',
    train_dataloader=train_dataloader_cfg,
    optim_wrapper=dict(optimizer=dict(type='SGD', lr=0.001, momentum=0.9)),
    train_cfg=dict(by_epoch=True, max_epochs=3),
    load_from='./work_dir/epoch_2.pth',
    resume=True,
)
runner.train()
```

CHAPTER 22

加速训练

22.1 分布式训练

MMEngine 支持 CPU、单卡、单机多卡以及多机多卡的训练。当环境中有多张显卡时，我们可以使用以下命令开启单机多卡或者多机多卡的方式从而缩短模型的训练时间。

- 单机多卡

假设当前机器有 8 张显卡，可以使用以下命令开启多卡训练

```
python -m torch.distributed.launch --nproc_per_node=8 examples/train.py --launcher␣
↪pytorch
```

如果需要指定显卡的编号，可以设置 CUDA_VISIBLE_DEVICES 环境变量，例如使用第 0 和第 3 张卡

```
CUDA_VISIBLE_DEVICES=0,3 python -m torch.distributed.launch --nproc_per_node=2␣
↪examples/train.py --launcher pytorch
```

- 多机多卡

假设有 2 台机器，每台机器有 8 张卡。

第一台机器运行以下命令

```
python -m torch.distributed.launch \
    --nnodes 8 \
    --node_rank 0 \
```

(下页继续)

(续上页)

```
    --master_addr 127.0.0.1 \
    --master_port 29500 \
    --nproc_per_node=8 \
    examples/train.py --launcher pytorch
```

第 2 台机器运行以下命令

```
python -m torch.distributed.launch \
    --nnodes 8 \
    --node_rank 1 \
    --master_addr 127.0.0.1 \
    --master_port 29500 \
    --nproc_per_node=8 \
    examples/train.py --launcher pytorch
```

如果在 slurm 集群运行 MMEngine，只需运行以下命令即可开启 2 机 16 卡的训练

```
srun -p mm_dev \
    --job-name=test \
    --gres=gpu:8 \
    --ntasks=16 \
    --ntasks-per-node=8 \
    --cpus-per-task=5 \
    --kill-on-bad-exit=1 \
    python examples/train.py --launcher="slurm"
```

22.2 混合精度训练

Nvidia 在 Volta 和 Turing 架构中引入 Tensor Core 单元，来支持 FP32 和 FP16 混合精度计算。开启自动混合精度训练后，部分算子的操作精度是 FP16，其余算子的操作精度是 FP32。这样在不改变模型、不降低模型训练精度的前提下，可以缩短训练时间，降低存储需求，因而能支持更大的 batch size、更大模型和尺寸更大的输入的训练。

PyTorch 从 1.6 开始官方支持 amp。如果你对自动混合精度的实现感兴趣，可以阅读 torch.cuda.amp: 自动混合精度详解。

MMEngine 提供自动混合精度的封装*AmpOptimWrapper*，只需在 `optim_wrapper` 设置 `type='AmpOptimWrapper'` 即可开启自动混合精度训练，无需对代码做其他修改。

```
runner = Runner(
    model=ResNet18(),
    work_dir='./work_dir',
    train_dataloader=train_dataloader_cfg,
```

(下页继续)

(续上页)

```
    optim_wrapper=dict(type='AmpOptimWrapper', optimizer=dict(type='SGD', lr=0.001,␣
↪momentum=0.9)),
    train_cfg=dict(by_epoch=True, max_epochs=3),
)
runner.train()
```

CHAPTER 23

节省显存

在深度学习训练推理过程中显存容量至关重要，其决定了模型是否能成功运行。常见的节省显存办法包括：

- 梯度累加

  梯度累加是指在每计算一个批次的梯度后，不进行清零而是进行梯度累加，当累加到一定的次数之后，再更新网络参数和梯度清零。通过这种参数延迟更新的手段，实现与采用大 batch 尺寸相近的效果，达到节省显存的目的。但是需要注意如果模型中包含 batch normalization 层，使用梯度累加会对性能有一定影响。

- 梯度检查点

  梯度检查点是一种以时间换空间的方法，通过减少保存的激活值来压缩模型占用空间，但是在计算梯度时必须重新计算没有存储的激活值。在 torch.utils.checkpoint 包中已经实现了对应功能。简要实现过程是：在前向阶段传递到 checkpoint 中的 forward 函数会以 `torch.no_grad` 模式运行，并且仅仅保存输入参数和 forward 函数，在反向阶段重新计算其 forward 输出值。

- 大模型训练技术

  最近的研究表明大型模型训练将有利于提高模型质量，但是训练如此大的模型需要巨大的资源，单卡显存已经越来越难以满足存放整个模型，因此诞生了大模型训练技术，典型的如 DeepSpeed ZeRO 和 FairScale 的完全分片数据并行（Fully Sharded Data Parallel, FSDP）技术，其允许在数据并行进程之间分片模型的参数、梯度和优化器状态，并同时仍然保持数据并行的简单性。

MMEngine 目前支持梯度累加和大模型训练 `FSDP` 技术。下面说明其用法。

23.1 梯度累加

配置写法如下所示：

```
optim_wrapper_cfg = dict(
    type='OptimWrapper',
    optimizer=dict(type='SGD', lr=0.001, momentum=0.9),
    # 累加 4 次参数更新一次
    accumulative_counts=4)
```

配合 Runner 使用的完整例子如下：

```
import torch
import torch.nn as nn
from torch.utils.data import DataLoader
from mmengine.runner import Runner
from mmengine.model import BaseModel

train_dataset = [(torch.ones(1, 1), torch.ones(1, 1))] * 50
train_dataloader = DataLoader(train_dataset, batch_size=2)


class ToyModel(BaseModel):
    def __init__(self) -> None:
        super().__init__()
        self.linear = nn.Linear(1, 1)

    def forward(self, img, label, mode):
        feat = self.linear(img)
        loss1 = (feat - label).pow(2)
        loss2 = (feat - label).abs()
        return dict(loss1=loss1, loss2=loss2)


runner = Runner(
    model=ToyModel(),
    work_dir='tmp_dir',
    train_dataloader=train_dataloader,
    train_cfg=dict(by_epoch=True, max_epochs=1),
    optim_wrapper=dict(optimizer=dict(type='SGD', lr=0.01),
                       accumulative_counts=4)
)
runner.train()
```

23.2 大模型训练

PyTorch 1.11 中已经原生支持了 FSDP 技术。配置写法如下所示：

```
# 位于 cfg 配置文件中
model_wrapper_cfg=dict(type='MMFullyShardedDataParallel', cpu_offload=True)
```

配合 Runner 使用的完整例子如下：

```
import torch
import torch.nn as nn
from torch.utils.data import DataLoader
from mmengine.runner import Runner
from mmengine.model import BaseModel

train_dataset = [(torch.ones(1, 1), torch.ones(1, 1))] * 50
train_dataloader = DataLoader(train_dataset, batch_size=2)


class ToyModel(BaseModel):
    def __init__(self) -> None:
        super().__init__()
        self.linear = nn.Linear(1, 1)

    def forward(self, img, label, mode):
        feat = self.linear(img)
        loss1 = (feat - label).pow(2)
        loss2 = (feat - label).abs()
        return dict(loss1=loss1, loss2=loss2)


runner = Runner(
    model=ToyModel(),
    work_dir='tmp_dir',
    train_dataloader=train_dataloader,
    train_cfg=dict(by_epoch=True, max_epochs=1),
    optim_wrapper=dict(optimizer=dict(type='SGD', lr=0.01)),
    cfg=dict(model_wrapper_cfg=dict(type='MMFullyShardedDataParallel', cpu_
↪offload=True))
)
runner.train()
```

注意必须在分布式训练环境中 FSDP 才能生效。

CHAPTER 24

跨库调用模块

通过使用 MMEngine 的注册器（*Registry*）和配置文件（*Config*），用户可以实现跨软件包的模块构建。例如，在 MMDetection 中使用 MMClassification 的 Backbone，或者在 MMRotate 中使用 MMDetection 的 Transform，或者在 MMTracking 中使用 MMDetection 的 Detector。一般来说，同类模块都可以进行跨库调用，只需要在配置文件的模块类型前加上软件包名的前缀即可。下面举几个常见的例子：

24.1 跨库调用 Backbone:

以在 MMDetection 中调用 MMClassification 的 ConvNeXt 为例，首先需要在配置中加入 `custom_imports` 字段将 MMClassification 的 Backbone 添加进注册器，然后只需要在 Backbone 的配置中的 `type` 加上 MMClassification 的软件包名 `mmcls` 作为前缀，即 `mmcls.ConvNeXt` 即可：

```
# 使用 custom_imports 将 mmcls 的 models 添加进注册器
custom_imports = dict(imports=['mmcls.models'], allow_failed_imports=False)

model = dict(
  type='MaskRCNN',
  data_preprocessor=dict(...),
  backbone=dict(
      type='mmcls.ConvNeXt',  # 添加 mmcls 前缀完成跨库调用
      arch='tiny',
      out_indices=[0, 1, 2, 3],
      drop_path_rate=0.4,
      layer_scale_init_value=1.0,
```

(下页继续)

(续上页)

```
      gap_before_final_norm=False,
      init_cfg=dict(
          type='Pretrained',
          checkpoint=
          'https://download.openmmlab.com/mmclassification/v0/convnext/downstream/
↪convnext-tiny_3rdparty_32xb128-noema_in1k_20220301-795e9634.pth',
          prefix='backbone.')),
  neck=dict(...),
  rpn_head=dict(...))
```

24.2 跨库调用 Transform:

与上文的跨库调用 Backbone 一样，使用 custom_imports 和添加前缀即可实现跨库调用：

```
# 使用 custom_imports 将 mmdet 的 transforms 添加进注册器
custom_imports = dict(imports=['mmdet.datasets.transforms'], allow_failed_
↪imports=False)

# 添加 mmdet 前缀完成跨库调用
train_pipeline=[
    dict(type='mmdet.LoadImageFromFile'),
    dict(type='mmdet.LoadAnnotations', with_bbox=True, box_type='qbox'),
    dict(type='ConvertBoxType', box_type_mapping=dict(gt_bboxes='rbox')),
    dict(type='mmdet.Resize', scale=(1024, 2014), keep_ratio=True),
    dict(type='mmdet.RandomFlip', prob=0.5),
    dict(type='mmdet.PackDetInputs')
]
```

24.3 跨库调用 Detector:

跨库调用算法是一个比较复杂的例子，一个算法会包含多个子模块，因此每个子模块也需要在 type 中增加前缀，以在 MMTracking 中调用 MMDetection 的 YOLOX 为例：

```
# 使用 custom_imports 将 mmdet 的 models 添加进注册器
custom_imports = dict(imports=['mmdet.models'], allow_failed_imports=False)
model = dict(
    type='mmdet.YOLOX',
    backbone=dict(type='mmdet.CSPDarknet', deepen_factor=1.33, widen_factor=1.25),
    neck=dict(
        type='mmdet.YOLOXPAFPN',
```

(下页继续)

(续上页)

```
        in_channels=[320, 640, 1280],
        out_channels=320,
        num_csp_blocks=4),
    bbox_head=dict(
        type='mmdet.YOLOXHead', num_classes=1, in_channels=320, feat_channels=320),
    train_cfg=dict(assigner=dict(type='mmdet.SimOTAAssigner', center_radius=2.5)))
```

为了避免给每个子模块手动增加前缀，配置文件中引入了 _scope_ 关键字，当某一模块的配置中添加了 _scope_ 关键字后，该模块配置文件下面的所有子模块配置都会从该关键字所对应的软件包内去构建：

```
# 使用 custom_imports 将 mmdet 的 models 添加进注册器
custom_imports = dict(imports=['mmdet.models'], allow_failed_imports=False)
model = dict(
    _scope_='mmdet',  # 使用 _scope_ 关键字，避免给所有子模块添加前缀
    type='YOLOX',
    backbone=dict(type='CSPDarknet', deepen_factor=1.33, widen_factor=1.25),
    neck=dict(
        type='YOLOXPAFPN',
        in_channels=[320, 640, 1280],
        out_channels=320,
        num_csp_blocks=4),
    bbox_head=dict(
        type='YOLOXHead', num_classes=1, in_channels=320, feat_channels=320),
    train_cfg=dict(assigner=dict(type='SimOTAAssigner', center_radius=2.5)))
```

以上这两种写法互相等价。

若希望了解更多关于注册器和配置文件的内容，请参考配置文件教程和注册器教程

CHAPTER 25

训练生成对抗网络

生成对抗网络 (Generative Adversarial Network, GAN) 可以用来生成图像视频等数据。这篇教程将带你一步步用 MMEngine 训练 GAN！

我们可以通过以下步骤来训练一个生成对抗网络。

25.1 构建数据加载器

25.1.1 构建数据集

接下来，我们为 MNIST 数据集构建一个数据集类 `MNISTDataset`，继承自数据集基类*BaseDataset*，并且重载数据集基类的 `load_data_list` 函数，保证返回值为 `list[dict]`，其中每个 `dict` 代表一个数据样本。更多关于 MMEngine 中数据集的用法，可以参考数据集教程。

```
import numpy as np
from mmcv.transforms import to_tensor
from torch.utils.data import random_split
from torchvision.datasets import MNIST

from mmengine.dataset import BaseDataset


class MNISTDataset(BaseDataset):

    def __init__(self, data_root, pipeline, test_mode=False):
        # 下载 MNIST 数据集
        if test_mode:
            mnist_full = MNIST(data_root, train=True, download=True)
            self.mnist_dataset, _ = random_split(mnist_full, [55000, 5000])
        else:
            self.mnist_dataset = MNIST(data_root, train=False, download=True)

        super().__init__(
            data_root=data_root, pipeline=pipeline, test_mode=test_mode)

    @staticmethod
    def totensor(img):
        if len(img.shape) < 3:
            img = np.expand_dims(img, -1)
        img = np.ascontiguousarray(img.transpose(2, 0, 1))
        return to_tensor(img)

    def load_data_list(self):
        return [
            dict(inputs=self.totensor(np.array(x[0]))) for x in self.mnist_dataset
        ]


dataset = MNISTDataset("./data", [])
```

使用 Runner 中的函数 build_dataloader 来构建数据加载器。

```
import os
import torch
from mmengine.runner import Runner

NUM_WORKERS = int(os.cpu_count() / 2)
```

(下页继续)

(续上页)

```
BATCH_SIZE = 256 if torch.cuda.is_available() else 64

train_dataloader = dict(
    batch_size=BATCH_SIZE,
    num_workers=NUM_WORKERS,
    persistent_workers=True,
    sampler=dict(type='DefaultSampler', shuffle=True),
    dataset=dataset)
train_dataloader = Runner.build_dataloader(train_dataloader)
```

25.2 构建生成器网络和判别器网络

下面的代码构建并实例化了一个生成器 (Generator) 和一个判别器 (Discriminator)。

```
import torch.nn as nn

class Generator(nn.Module):
    def __init__(self, noise_size, img_shape):
        super().__init__()
        self.img_shape = img_shape
        self.noise_size = noise_size

        def block(in_feat, out_feat, normalize=True):
            layers = [nn.Linear(in_feat, out_feat)]
            if normalize:
                layers.append(nn.BatchNorm1d(out_feat, 0.8))
            layers.append(nn.LeakyReLU(0.2, inplace=True))
            return layers

        self.model = nn.Sequential(
            *block(noise_size, 128, normalize=False),
            *block(128, 256),
            *block(256, 512),
            *block(512, 1024),
            nn.Linear(1024, int(np.prod(img_shape))),
            nn.Tanh(),
        )

    def forward(self, z):
        img = self.model(z)
        img = img.view(img.size(0), *self.img_shape)
        return img
```

```
class Discriminator(nn.Module):
    def __init__(self, img_shape):
        super().__init__()

        self.model = nn.Sequential(
            nn.Linear(int(np.prod(img_shape)), 512),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Linear(512, 256),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Linear(256, 1),
            nn.Sigmoid(),
        )

    def forward(self, img):
        img_flat = img.view(img.size(0), -1)
        validity = self.model(img_flat)

        return validity
```

```
generator = Generator(100, (1, 28, 28))
discriminator = Discriminator((1, 28, 28))
```

25.3 构建一个生成对抗网络模型

在使用 MMEngine 时，我们用*ImgDataPreprocessor* 来对数据进行归一化和颜色通道的转换。

```
from mmengine.model import ImgDataPreprocessor

data_preprocessor = ImgDataPreprocessor(mean=([127.5]), std=([127.5]))
```

下面的代码实现了基础 GAN 的算法。使用 MMEngine 实现算法类，需要继承*BaseModel* 基类，在 train_step 中实现训练过程。GAN 需要交替训练生成器和判别器，分别由 train_discriminator 和 train_generator 实现，并实现 disc_loss 和 gen_loss 计算判别器损失函数和生成器损失函数。关于 BaseModel 的更多信息，请参考模型教程.

```
import torch.nn.functional as F
from mmengine.model import BaseModel

class GAN(BaseModel):

    def __init__(self, generator, discriminator, noise_size,
                 data_preprocessor):
```

(下页继续)

(续上页)

```
        super().__init__(data_preprocessor=data_preprocessor)
        assert generator.noise_size == noise_size
        self.generator = generator
        self.discriminator = discriminator
        self.noise_size = noise_size

    def train_step(self, data, optim_wrapper):
        # 获取数据和数据预处理
        inputs_dict = self.data_preprocessor(data, True)
        # 训练判别器
        disc_optimizer_wrapper = optim_wrapper['discriminator']
        with disc_optimizer_wrapper.optim_context(self.discriminator):
            log_vars = self.train_discriminator(inputs_dict,
                                                disc_optimizer_wrapper)

        # 训练生成器
        set_requires_grad(self.discriminator, False)
        gen_optimizer_wrapper = optim_wrapper['generator']
        with gen_optimizer_wrapper.optim_context(self.generator):
            log_vars_gen = self.train_generator(inputs_dict,
                                                gen_optimizer_wrapper)

        set_requires_grad(self.discriminator, True)
        log_vars.update(log_vars_gen)

        return log_vars

    def forward(self, batch_inputs, data_samples=None, mode=None):
        return self.generator(batch_inputs)

    def disc_loss(self, disc_pred_fake, disc_pred_real):
        losses_dict = dict()
        losses_dict['loss_disc_fake'] = F.binary_cross_entropy(
            disc_pred_fake, 0. * torch.ones_like(disc_pred_fake))
        losses_dict['loss_disc_real'] = F.binary_cross_entropy(
            disc_pred_real, 1. * torch.ones_like(disc_pred_real))

        loss, log_var = self.parse_losses(losses_dict)
        return loss, log_var

    def gen_loss(self, disc_pred_fake):
        losses_dict = dict()
        losses_dict['loss_gen'] = F.binary_cross_entropy(
```

(下页继续)

(续上页)

```
            disc_pred_fake, 1. * torch.ones_like(disc_pred_fake))
        loss, log_var = self.parse_losses(losses_dict)
        return loss, log_var

    def train_discriminator(self, inputs, optimizer_wrapper):
        real_imgs = inputs['inputs']
        z = torch.randn(
            (real_imgs.shape[0], self.noise_size)).type_as(real_imgs)
        with torch.no_grad():
            fake_imgs = self.generator(z)

        disc_pred_fake = self.discriminator(fake_imgs)
        disc_pred_real = self.discriminator(real_imgs)

        parsed_losses, log_vars = self.disc_loss(disc_pred_fake,
                                                 disc_pred_real)
        optimizer_wrapper.update_params(parsed_losses)
        return log_vars

    def train_generator(self, inputs, optimizer_wrapper):
        real_imgs = inputs['inputs']
        z = torch.randn(real_imgs.shape[0], self.noise_size).type_as(real_imgs)

        fake_imgs = self.generator(z)

        disc_pred_fake = self.discriminator(fake_imgs)
        parsed_loss, log_vars = self.gen_loss(disc_pred_fake)

        optimizer_wrapper.update_params(parsed_loss)
        return log_vars
```

其中一个函数 set_requires_grad 用来锁定训练生成器时判别器的权重。

```
def set_requires_grad(nets, requires_grad=False):
    """Set requires_grad for all the networks.

    Args:
        nets (nn.Module | list[nn.Module]): A list of networks or a single
            network.
        requires_grad (bool): Whether the networks require gradients or not.
    """
    if not isinstance(nets, list):
        nets = [nets]
    for net in nets:
```

(下页继续)

(续上页)

```
        if net is not None:
            for param in net.parameters():
                param.requires_grad = requires_grad
```

```
model = GAN(generator, discriminator, 100, data_preprocessor)
```

25.4 构建优化器

MMEngine 使用*OptimWrapper* 来封装优化器，对于多个优化器的情况，使用*OptimWrapperDict* 对 OptimWrapper 再进行一次封装。关于优化器的更多信息，请参考优化器教程.

```
from mmengine.optim import OptimWrapper, OptimWrapperDict

opt_g = torch.optim.Adam(generator.parameters(), lr=0.0001, betas=(0.5, 0.999))
opt_g_wrapper = OptimWrapper(opt_g)

opt_d = torch.optim.Adam(
    discriminator.parameters(), lr=0.0001, betas=(0.5, 0.999))
opt_d_wrapper = OptimWrapper(opt_d)

opt_wrapper_dict = OptimWrapperDict(
    generator=opt_g_wrapper, discriminator=opt_d_wrapper)
```

25.5 使用执行器进行训练

下面的代码演示了如何使用 Runner 进行模型训练。关于 Runner 的更多信息，请参考执行器教程。

```
train_cfg = dict(by_epoch=True, max_epochs=220)
runner = Runner(
    model,
    work_dir='runs/gan/',
    train_dataloader=train_dataloader,
    train_cfg=train_cfg,
    optim_wrapper=opt_wrapper_dict)
runner.train()
```

到这里，我们就完成了一个 GAN 的训练，通过下面的代码可以查看刚才训练的 GAN 生成的结果。

```
z = torch.randn(64, 100).cuda()
img = model(z)

from torchvision.utils import save_image
save_image(img, "result.png", normalize=True)
```

如果你想了解更多如何使用 MMEngine 实现 GAN 和生成模型，我们强烈建议你使用同样基于 MMEngine 开发的生成框架 MMGen。

CHAPTER 26

钩子

钩子编程是一种编程模式，是指在程序的一个或者多个位置设置位点（挂载点），当程序运行至某个位点时，会自动调用运行时注册到位点的所有方法。钩子编程可以提高程序的灵活性和拓展性，用户将自定义的方法注册到位点便可被调用而无需修改程序中的代码。

26.1 钩子示例

下面是钩子的简单示例。

```
pre_hooks = [(print, 'hello')]
post_hooks = [(print, 'goodbye')]

def main():
    for func, arg in pre_hooks:
        func(arg)
    print('do something here')
    for func, arg in post_hooks:
        func(arg)

main()
```

下面是程序的输出：

```
hello
do something here
goodbye
```

可以看到，main 函数在两个位置调用钩子中的函数而无需做任何改动。

在 PyTorch 中，钩子的应用也随处可见，例如神经网络模块（nn.Module）中的钩子可以获得模块的前向输入输出以及反向的输入输出。以 register_forward_hook 方法为例，该方法往模块注册一个前向钩子，钩子可以获得模块的前向输入和输出。

下面是 register_forward_hook 用法的简单示例：

```
import torch
import torch.nn as nn

def forward_hook_fn(
    module,  # 被注册钩子的对象
    input,  # module 前向计算的输入
    output,  # module 前向计算的输出
):
    print(f'"forward_hook_fn" is invoked by {module.name}')
    print('weight:', module.weight.data)
    print('bias:', module.bias.data)
    print('input:', input)
    print('output:', output)

class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc = nn.Linear(3, 1)

    def forward(self, x):
        y = self.fc(x)
        return y

model = Model()
# 将 forward_hook_fn 注册到 model 每个子模块
for module in model.children():
    module.register_forward_hook(forward_hook_fn)

x = torch.Tensor([[0.0, 1.0, 2.0]])
y = model(x)
```

下面是程序的输出：

```
"forward_hook_fn" is invoked by Linear(in_features=3, out_features=1, bias=True)
weight: tensor([[-0.4077,  0.0119, -0.3606]])
bias: tensor([-0.2943])
input: (tensor([[0., 1., 2.]]),)
output: tensor([[-1.0036]], grad_fn=<AddmmBackward>)
```

可以看到注册到 Linear 模块的 `forward_hook_fn` 钩子被调用，在该钩子中打印了 Linear 模块的权重、偏置、模块的输入以及输出。更多关于 PyTorch 钩子的用法可以阅读 nn.Module。

26.2 MMEngine 中钩子的设计

在介绍 MMEngine 中钩子的设计之前，先简单介绍使用 PyTorch 实现模型训练的基本步骤（示例代码来自 PyTorch Tutorials）：

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import torchvision.transforms as transforms
from torch.utils.data import Dataset, DataLoader

class CustomDataset(Dataset):
    pass

class Net(nn.Module):
    pass

def main():
    transform = transforms.ToTensor()
    train_dataset = CustomDataset(transform=transform, ...)
    val_dataset = CustomDataset(transform=transform, ...)
    test_dataset = CustomDataset(transform=transform, ...)
    train_dataloader = DataLoader(train_dataset, ...)
    val_dataloader = DataLoader(val_dataset, ...)
    test_dataloader = DataLoader(test_dataset, ...)

    net = Net()
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)

    for i in range(max_epochs):
        for inputs, labels in train_dataloader:
            optimizer.zero_grad()
```

(下页继续)

(续上页)

```
            outputs = net(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

        with torch.no_grad():
            for inputs, labels in val_dataloader:
                outputs = net(inputs)
                loss = criterion(outputs, labels)

    with torch.no_grad():
        for inputs, labels in test_dataloader:
            outputs = net(inputs)
            accuracy = ...
```

上面的伪代码是训练模型的基本步骤。如果要在上面的代码中加入定制化的逻辑，我们需要不断修改和拓展 main 函数。为了提高 main 函数的灵活性和拓展性，我们可以在 main 方法中插入位点，并在对应位点实现调用 hook 的抽象逻辑。此时只需在这些位点插入 hook 来实现定制化逻辑，即可添加定制化功能，例如加载模型权重、更新模型参数等。

```
def main():
    ...
    call_hooks('before_run', hooks)  # 任务开始前执行的逻辑
    call_hooks('after_load_checkpoint', hooks)  # 加载权重后执行的逻辑
    call_hooks('before_train', hooks)  # 训练开始前执行的逻辑
    for i in range(max_epochs):
        call_hooks('before_train_epoch', hooks)  # 遍历训练数据集前执行的逻辑
        for inputs, labels in train_dataloader:
            call_hooks('before_train_iter', hooks)  # 模型前向计算前执行的逻辑
            outputs = net(inputs)
            loss = criterion(outputs, labels)
            call_hooks('after_train_iter', hooks)  # 模型前向计算后执行的逻辑
            loss.backward()
            optimizer.step()
        call_hooks('after_train_epoch', hooks)  # 遍历完训练数据集后执行的逻辑

        call_hooks('before_val_epoch', hooks)  # 遍历验证数据集前执行的逻辑
        with torch.no_grad():
            for inputs, labels in val_dataloader:
                call_hooks('before_val_iter', hooks)  # 模型前向计算前执行
                outputs = net(inputs)
                loss = criterion(outputs, labels)
                call_hooks('after_val_iter', hooks)  # 模型前向计算后执行
```

(下页继续)

(续上页)

```
        call_hooks('after_val_epoch', hooks)  # 遍历完验证数据集前执行

        call_hooks('before_save_checkpoint', hooks)  # 保存权重前执行的逻辑
    call_hooks('after_train', hooks)  # 训练结束后执行的逻辑

    call_hooks('before_test_epoch', hooks)  # 遍历测试数据集前执行的逻辑
    with torch.no_grad():
        for inputs, labels in test_dataloader:
            call_hooks('before_test_iter', hooks)  # 模型前向计算后执行的逻辑
            outputs = net(inputs)
            accuracy = ...
            call_hooks('after_test_iter', hooks)  # 遍历完成测试数据集后执行的逻辑
    call_hooks('after_test_epoch', hooks)  # 遍历完测试数据集后执行

    call_hooks('after_run', hooks)  # 任务结束后执行的逻辑
```

在 MMEngine 中，我们将训练过程抽象成执行器（Runner），执行器除了完成环境的初始化，另一个功能是在特定的位点调用钩子完成定制化逻辑。更多关于执行器的介绍请阅读执行器文档。

为了方便管理，MMEngine 将位点定义为方法并集成到钩子基类（*Hook*）中，我们只需继承钩子基类并根据需求在特定位点实现定制化逻辑，再将钩子注册到执行器中，便可自动调用钩子中相应位点的方法。

钩子中一共有 22 个位点：

- before_run
- after_run
- before_train
- after_train
- before_train_epoch
- after_train_epoch
- before_train_iter
- after_train_iter
- before_val
- after_val
- before_test_epoch
- after_test_epoch
- before_val_iter
- after_val_iter

- before_test
- after_test
- before_test_epoch
- after_test_epoch
- before_test_iter
- after_test_iter
- before_save_checkpoint
- after_load_checkpoint

你可能还想阅读钩子的用法或者钩子的 *API* 文档。

CHAPTER 27

执行器

深度学习算法的训练、验证和测试通常都拥有相似的流程，因此，MMEngine 抽象出了执行器来负责通用的算法模型的训练、测试、推理任务。用户一般可以直接使用 MMEngine 中的默认执行器，也可以对执行器进行修改以满足定制化需求。

在介绍执行器的设计之前，我们先举几个例子来帮助用户理解为什么需要执行器。下面是一段使用 PyTorch 进行模型训练的伪代码：

```
model = ResNet()
optimizer = SGD(model.parameters(), lr=0.01, momentum=0.9)
train_dataset = ImageNetDataset(...)
train_dataloader = DataLoader(train_dataset, ...)

for i in range(max_epochs):
    for data_batch in train_dataloader:
        optimizer.zero_grad()
        outputs = model(data_batch)
        loss = loss_func(outputs, data_batch)
        loss.backward()
        optimizer.step()
```

下面是一段使用 PyTorch 进行模型测试的伪代码：

```
model = ResNet()
model.load_state_dict(torch.load(CKPT_PATH))
model.eval()
```

(下页继续)

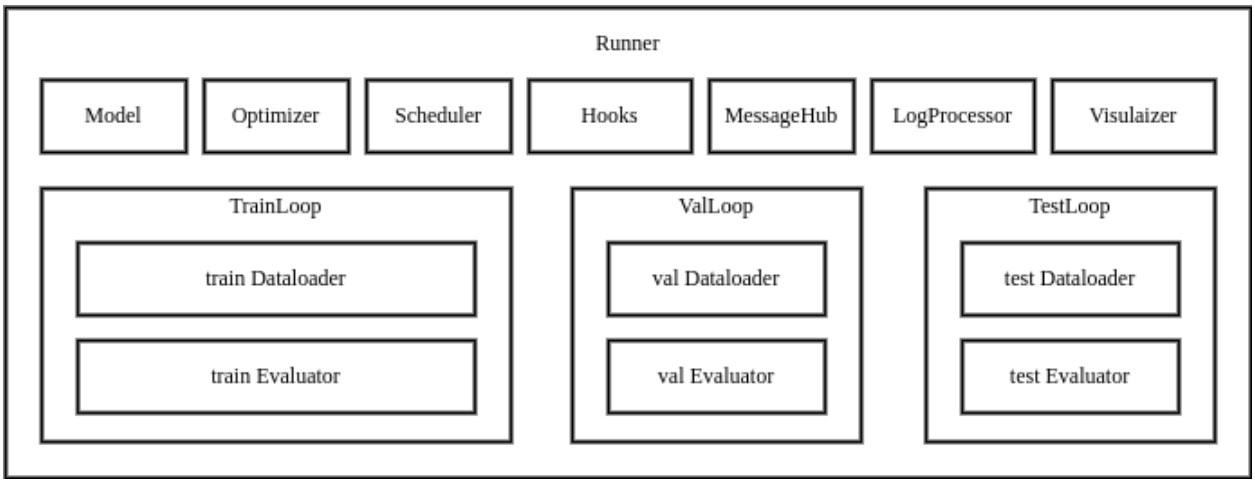
(续上页)

```
test_dataset = ImageNetDataset(...)
test_dataloader = DataLoader(test_dataset, ...)

for data_batch in test_dataloader:
    outputs = model(data_batch)
    acc = calculate_acc(outputs, data_batch)
```

下面是一段使用 PyTorch 进行模型推理的伪代码：

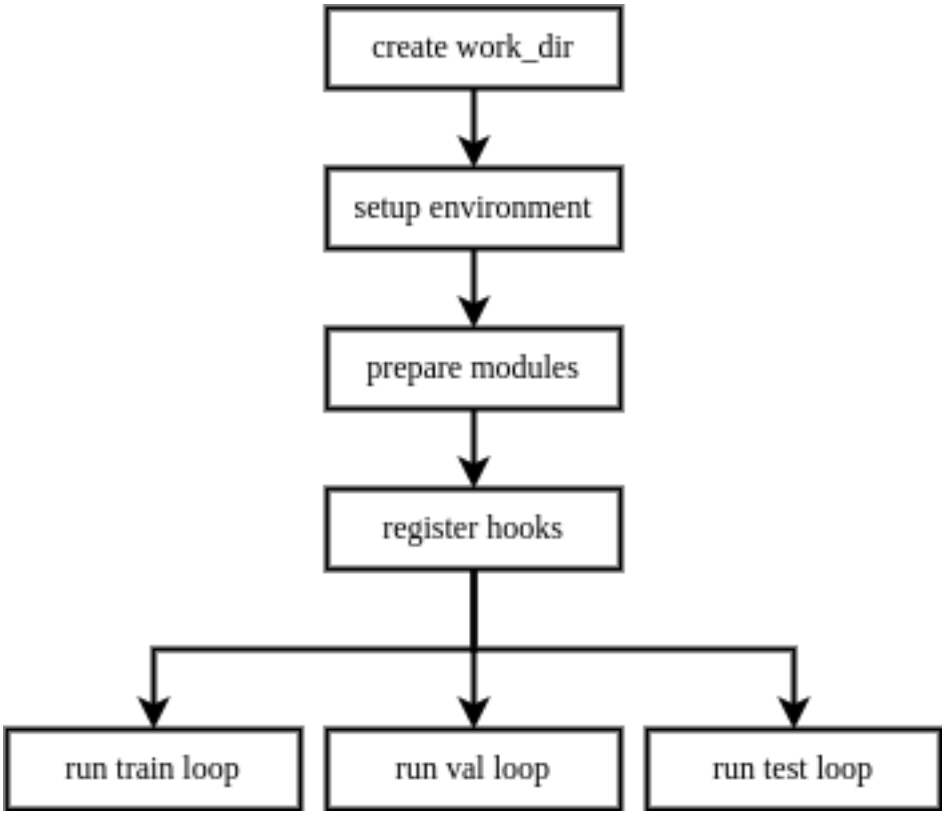
```
model = ResNet()
model.load_state_dict(torch.load(CKPT_PATH))
model.eval()

for img in imgs:
    prediction = model(img)
```

可以从上面的三段代码看出，这三个任务的执行流程都可以归纳为构建模型、读取数据、循环迭代等步骤。上述代码都是以图像分类为例，但不论是图像分类还是目标检测或是图像分割，都脱离不了这套范式。因此，我们将模型的训练、验证、测试的流程整合起来，形成了执行器。在执行器中，我们只需要准备好模型、数据等任务必须的模块或是这些模块的配置文件，执行器会自动完成任务流程的准备和执行。通过使用执行器以及 MMEngine 中丰富的功能模块，用户不再需要手动搭建训练测试的流程，也不再需要去处理分布式与非分布式训练的区别，可以专注于算法和模型本身。



MMEngine 的执行器内包含训练、测试、验证所需的各个模块，以及循环控制器（Loop）和钩子（*Hook*）。用户通过提供配置文件或已构建完成的模块，执行器将自动完成运行环境的配置，模块的构建和组合，最终通过循环控制器执行任务循环。执行器对外提供三个接口：`train`，`val`，`test`，当调用这三个接口时，便会运行对应的循环控制器，并在循环的运行过程中调用钩子模块各个位点的钩子函数。

当用户构建一个执行器并调用训练、验证、测试的接口时，执行器的执行流程如下：创建工作目录 -> 配置运行环境 -> 准备任务所需模块 -> 注册钩子 -> 运行循环

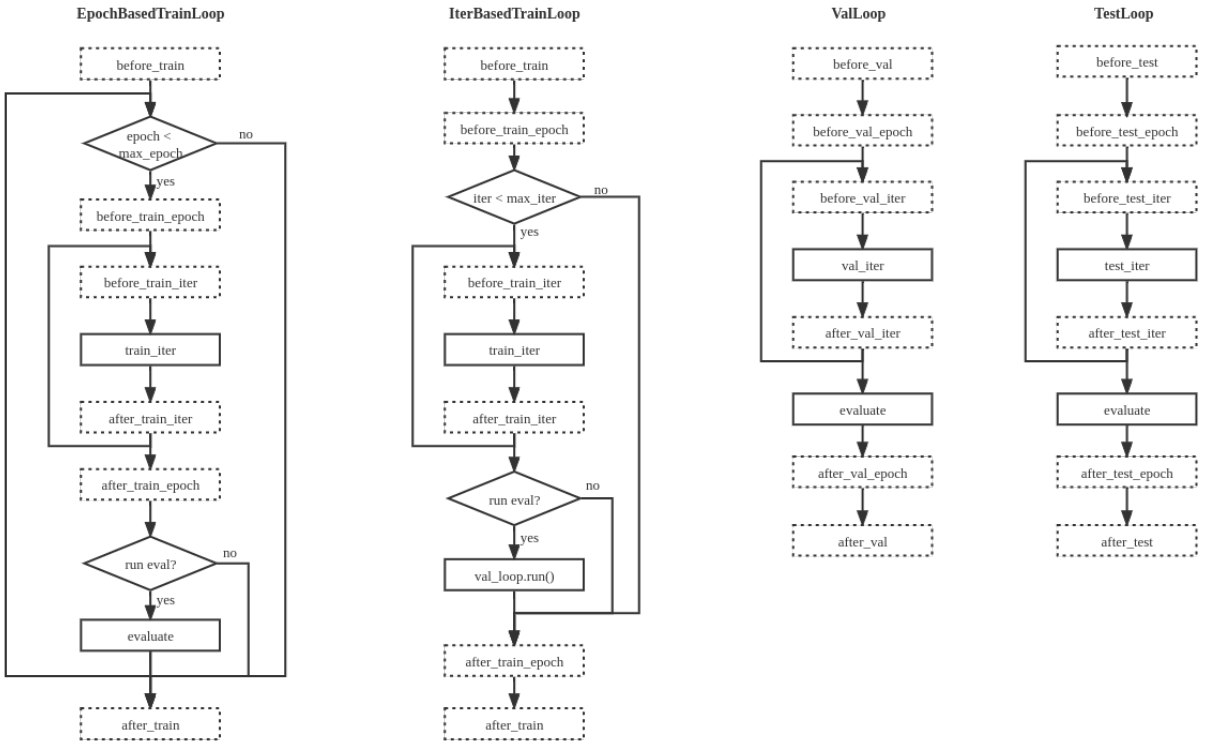


执行器具有延迟初始化（Lazy Initialization）的特性，在初始化执行器时，并不需要依赖训练、验证和测试的全量模块，只有当运行某个循环控制器时，才会检查所需模块是否构建。因此，若用户只需要执行训练、验证或测试中的某一项功能，只需提供对应的模块或模块的配置即可。

27.1 循环控制器

在 MMEngine 中，我们将任务的执行流程抽象成循环控制器（Loop），因为大部分的深度学习任务执行流程都可以归纳为模型在一组或多组数据上进行循环迭代。MMEngine 内提供了四种默认的循环控制器：

- EpochBasedTrainLoop 基于轮次的训练循环
- IterBasedTrainLoop 基于迭代次数的训练循环
- ValLoop 标准的验证循环
- TestLoop 标准的测试循环



MMEngine 中的默认执行器和循环控制器能够完成大部分的深度学习任务，但不可避免会存在无法满足的情况。有的用户希望能够对执行器进行更多自定义修改，因此，MMEngine 支持自定义模型的训练、验证以及测试的流程。

用户可以通过继承循环基类来实现自己的训练流程。循环基类需要提供两个输入：`runner` 执行器的实例和 `loader` 循环所需要迭代的迭代器。用户如果有自定义的需求，也可以增加更多的输入参数。MMEngine 中同样提供了 LOOPS 注册器对循环类进行管理，用户可以向注册器内注册自定义的循环模块，然后在配置文件的 `train_cfg`、`val_cfg`、`test_cfg` 中增加 `type` 字段来指定使用何种循环。用户可以在自定义的循环中实现任意的执行逻辑，也可以增加或删减钩子（hook）点位，但需要注意的是一旦钩子点位被修改，默认的钩子函数可能不会被执行，导致一些训练过程中默认发生的行为发生变化。因此，我们强烈建议用户按照本文档中定义的循环执行流程图以及钩子设计 去重载循环基类。

```
from mmengine.registry import LOOPS, HOOKS
from mmengine.runner import BaseLoop
from mmengine.hooks import Hook


# 自定义验证循环
@LOOPS.register_module()
class CustomValLoop(BaseLoop):
    def __init__(self, runner, dataloader, evaluator, dataloader2):
        super().__init__(runner, dataloader, evaluator)
        self.dataloader2 = runner.build_dataloader(dataloader2)
```

(下页继续)

(续上页)

```
    def run(self):
        self.runner.call_hooks('before_val_epoch')
        for idx, data_batch in enumerate(self.dataloader):
            self.runner.call_hooks(
                'before_val_iter', batch_idx=idx, data_batch=data_batch)
            outputs = self.run_iter(idx, data_batch)
            self.runner.call_hooks(
                'after_val_iter', batch_idx=idx, data_batch=data_batch,␣
↪outputs=outputs)
        metric = self.evaluator.evaluate()

        # 增加额外的验证循环
        for idx, data_batch in enumerate(self.dataloader2):
            # 增加额外的钩子点位
            self.runner.call_hooks(
                'before_valloader2_iter', batch_idx=idx, data_batch=data_batch)
            self.run_iter(idx, data_batch)
            # 增加额外的钩子点位
            self.runner.call_hooks(
                'after_valloader2_iter', batch_idx=idx, data_batch=data_batch,␣
↪outputs=outputs)
        metric2 = self.evaluator.evaluate()

        ...

        self.runner.call_hooks('after_val_epoch')


# 定义额外点位的钩子类
@HOOKS.register_module()
class CustomValHook(Hook):
    def before_valloader2_iter(self, batch_idx, data_batch):
        ...

    def after_valloader2_iter(self, batch_idx, data_batch, outputs):
        ...
```

上面的例子中实现了一个与默认验证循环不一样的自定义验证循环，它在两个不同的验证集上进行验证，同时对第二次验证增加了额外的钩子点位，并在最后对两个验证结果进行进一步的处理。在实现了自定义的循环类之后，只需要在配置文件的 `val_cfg` 内设置 `type='CustomValLoop'`，并添加额外的配置即可。

```
# 自定义验证循环
val_cfg = dict(type='CustomValLoop', dataloader2=dict(dataset=dict(type='ValDataset2
↪'), ...))
# 额外点位的钩子
custom_hooks = [dict(type='CustomValHook')]
```

27.2 自定义执行器

更进一步，如果默认执行器中依然有其他无法满足需求的部分，用户可以像自定义其他模块一样，通过继承重写的方式，实现自定义的执行器。执行器同样也可以通过注册器进行管理。具体实现流程与其他模块无异：继承 MMEngine 中的 Runner，重写需要修改的函数，添加进 RUNNERS 注册器中，最后在配置文件中指定 `runner_type` 即可。

```
from mmengine.registry import RUNNERS
from mmengine.runner import Runner

@RUNNERS.register_module()
class CustomRunner(Runner):

    def setup_env(self):
        ...
```

上述例子实现了一个自定义的执行器，并重写了 `setup_env` 函数，然后添加进了 RUNNERS 注册器中，完成了这些步骤之后，便可以在配置文件中设置 `runner_type='CustomRunner'` 来构建自定义的执行器。

你可能还想阅读执行器的教程或者执行器的 *API* 文档。

CHAPTER 28

模型精度评测

在模型验证和模型测试中，通常需要对模型精度做定量评测。在 MMEngine 中实现了评测指标和评测器来完成这一功能。

评测指标根据模型的输入数据和预测结果，完成特定指标下模型精度的计算。评测指标与数据集之间相互解耦，这使得用户可以任意组合所需的测试数据和评测指标。如 COCOMetric 可用于计算 COCO 数据集的 AP，AR 等评测指标，也可用于其他的目标检测数据集上。**评测器**是评测指标的上层模块，通常包含一个或多个评测指标。评测器的作用是在模型评测时完成必要的数据格式转换，并调用评测指标计算模型精度。评测器通常由执行器或测试脚本构建，分别用于在线评测和离线评测。

28.1 模型精度评测流程

通常，模型精度评测的过程如下图所示。

在线评测：测试数据通常会被划分为若干批次（batch）。通过一个循环，依次将每个批次的数据送入模型，得到对应的预测结果，并将测试数据和模型预测结果送入评测器。评测器会调用评测指标的 `process()` 方法对数据和预测结果进行处理。当循环结束后，评测器会调用评测指标的 `evaluate()` 方法，可计算得到对应指标的模型精度。

离线评测：与在线评测过程类似，区别是直接读取预先保存的模型预测结果来进行评测。评测器提供了 `offline_evaluate` 接口，用于在离线方式下调用评测指标来计算模型精度。为了避免同时处理大量数据导致内存溢出，离线评测时会将测试数据和预测结果分成若干个块（chunk）进行处理，类似在线评测中的批次。

28.2 增加自定义评测指标

在 OpenMMLab 的各个算法库中，已经实现了对应方向的常用评测指标。如 MMDetection 中提供了 COCO 评测指标，MMClassification 中提供了 Accuracy、F1Score 等评测指标等。

用户也可以增加自定义的评测指标。在实现自定义评测指标时，需要继承 MMEngine 中提供的评测指标基类*BaseMetric*，并实现对应的抽象方法。

28.2.1 评测指标基类

评测指标基类 `BaseMetric` 是一个抽象类，具有以下 2 个抽象方法：

- `process()`: 处理每个批次的测试数据和模型预测结果。处理结果应存放在 `self.results` 列表中，用于在处理完所有测试数据后计算评测指标。
- `compute_metrics()`: 计算评测指标，并将所评测指标存放在一个字典中返回。

其中，`compute_metrics()` 会在 `evaluate()` 方法中被调用；后者在计算评测指标前，会在分布式测试时收集和汇总不同 rank 的中间处理结果。

需要注意的是，`self.results` 中存放的具体类型取决于评测指标子类的实现。例如，当测试样本或模型输出数据量较大（如语义分割、图像生成等任务），不宜全部存放在内存中时，可以在 `self.results` 中存放每个批次计算得到的指标，并在 `compute_metrics()` 中汇总；或将每个批次的中间结果存储到临时文件中，并在 `self.results` 中存放临时文件路径，最后由 `compute_metrics()` 从文件中读取数据并计算指标。

28.2.2 自定义评测指标类

我们以实现分类正确率（Classification Accuracy）评测指标为例，说明自定义评测指标的方法。

首先，评测指标类应继承自 `BaseMetric`，并应加入注册器 `METRICS` (关于注册器的说明请参考相关文档)。

`process()` 方法有 2 个输入参数，分别是一个批次的测试数据样本 `data_batch` 和模型预测结果 `predictions`。我们从中分别取出样本类别标签和分类预测结果，并存放在 `self.results` 中。

`compute_metrics()` 方法有 1 个输入参数 `results`，里面存放了所有批次测试数据经过 `process()` 方法处理后得到的结果。从中取出样本类别标签和分类预测结果，即可计算得到分类正确率 `acc`。最终，将计算得到的评测指标以字典的形式返回。

此外，我们建议在子类中为类属性 `default_prefix` 赋值。如果在初始化参数（即 config 中）没有指定 `prefix`，则会自动使用 `default_prefix` 作为评测指标名的前缀。同时，应在 docstring 中说明该评测指标类的 `default_prefix` 值以及所有的返回指标名称。

具体的实现如下：

```python
from mmengine.evaluator import BaseMetric
from mmengine.registry import METRICS

import numpy as np


@METRICS.register_module()  # 将 Accuracy 类注册到 METRICS 注册器
class Accuracy(BaseMetric):
    """ Accuracy Evaluator

    Default prefix: ACC

    Metrics:
        - accuracy (float): classification accuracy
    """

    default_prefix = 'ACC'  # 设置 default_prefix

    def process(self, data_batch: Sequence[dict], predictions: Sequence[dict]):
        """Process one batch of data and predictions. The processed
        Results should be stored in `self.results`, which will be used
        to computed the metrics when all batches have been processed.

        Args:
            data_batch (Sequence[Tuple[Any, dict]]): A batch of data
                from the dataloader.
            predictions (Sequence[dict]): A batch of outputs from
                the model.
        """

        # 取出分类预测结果和类别标签
        result = {
            'pred': predictions['pred_label'],
            'gt': data_batch['data_sample']['gt_label']
        }

        # 将当前 batch 的结果存进 self.results
        self.results.append(result)

    def compute_metrics(self, results: List):
        """Compute the metrics from processed results.

        Args:
            results (dict): The processed results of each batch.
```

(下页继续)

(续上页)

```
        Returns:
            Dict: The computed metrics. The keys are the names of the metrics,
            and the values are corresponding results.
        """

        # 汇总所有样本的分类预测结果和类别标签
        preds = np.concatenate([res['pred'] for res in results])
        gts = np.concatenate([res['gt'] for res in results])

        # 计算分类正确率
        acc = (preds == gts).sum() / preds.size

        # 返回评测指标结果
        return {'accuracy': acc}
```

CHAPTER 29

可视化

29.1 1 总体设计

可视化可以给深度学习的模型训练和测试过程提供直观解释。在 OpenMMLab 算法库中，我们期望可视化功能的设计能满足以下需求：

- 提供丰富的开箱即用可视化功能，能够满足大部分计算机视觉可视化任务
- 高扩展性，可视化功能通常多样化，应该能够通过简单扩展实现定制需求
- 能够在训练和测试流程的任意点位进行可视化
- OpenMMLab 各个算法库具有统一可视化接口，利于用户理解和维护

基于上述需求，OpenMMLab 2.0 引入了可视化对象 Visualizer 和各个可视化存储后端 VisBackend 如 `LocalVisBackend`、`WandbVisBackend` 和 `TensorboardVisBackend` 等。此处的可视化不仅仅包括图片数据格式，还包括配置内容、标量和模型图等数据的可视化。

- 为了方便调用，Visualizer 提供的接口实现了绘制和存储的功能。可视化存储后端 VisBackend 作为 Visualizer 的内部属性，会在需要的时候被 Visualizer 调用，将数据存到不同的后端
- 考虑到绘制后会希望存储到多个后端，Visualizer 可以配置多个 VisBackend，当用户调用 Visualizer 的存储接口时候，Visualizer 内部会遍历的调用 VisBackend 存储接口

两者的 UML 关系图如下

29.2 2 可视化器 Visualizer

可视化对象 Visualizer 对外提供了所有接口。可以将其接口分成 3 大类，如下所示

(1) 绘制相关接口

- *draw_bboxes* 绘制单个或多个边界框
- *draw_points* 绘制单个或多个点
- *draw_texts* 绘制单个或多个文本框
- draw_lines 绘制单个或多个线段
- *draw_circles* 绘制单个或多个圆
- *draw_polygons* 绘制单个或多个多边形
- draw_binary_masks 绘制单个或多个二值掩码
- *draw_featmap* 绘制特征图，静态方法

上述接口除了 `draw_featmap` 外都可以链式调用，因为该方法调用后可能会导致图片尺寸发生改变。为了避免给用户带来困扰，`draw_featmap` 被设置为静态方法。

(2) 存储相关接口

- add_config 写配置到特定存储后端
- add_graph 写模型图到特定存储后端
- add_image 写图片到特定存储后端
- add_scalar 写标量到特定存储后端
- add_scalars 一次性写多个标量到特定存储后端
- add_datasample 各个下游库绘制 datasample 数据的抽象接口

以 add 前缀开头的接口表示存储接口。datasample 是 OpenMMLab 2.0 架构中设计的各个下游库统一的抽象数据接口，而 `add_datasample` 接口可以直接处理该数据格式，例如可视化预测结果、可视化 Dataset 或者 DataLoader 输出、可视化中间预测结果等等都可以直接调用下游库重写的 `add_datasample` 接口。所有下游库都必须要继承 Visualizer 并实现 `add_datasample` 接口。以 MMDetection 为例，应该继承并通过该接口实现目标检测中所有预置任务的可视化功能，例如目标检测、实例分割、全景分割任务结果的绘制和存储。

(3) 其余功能性接口

- *set_image* 设置原始图片数据，默认输入图片格式为 RGB
- *get_image* 获取绘制后的 Numpy 格式图片数据，默认输出格式为 RGB
- *show* 可视化
- *get_backend* 通过 name 获取特定存储后端
- *close* 关闭所有已经打开的资源，包括 VisBackend

关于其用法，可以参考 可视化器用户教程。

29.3 3 可视化存储后端 VisBackend

在绘制后可以将绘制后的数据存储到多个可视化存储后端中。为了统一接口调用，MMEngine 提供了统一的抽象类 `BaseVisBackend`，和一些常用的 VisBackend 如 `LocalVisBackend`、`WandbVisBackend` 和 `TensorboardVisBackend`。BaseVisBackend 定义了对外调用的接口规范，主要接口和属性如下：

- add_config 写配置到特定存储后端
- add_graph 写模型图到特定后端
- add_image 写图片到特定后端
- add_scalar 写标量到特定后端
- add_scalars 一次性写多个标量到特定后端
- close 关闭已经打开的资源
- experiment 写后端对象，例如 WandB 对象和 Tensorboard 对象

`BaseVisBackend` 定义了 5 个常见的写数据接口，考虑到某些写后端功能非常强大，例如 WandB，其具备写表格，写视频等等功能，针对这类需求用户可以直接获取 `experiment` 对象，然后调用写后端对象本身的 API 即可。而 `LocalVisBackend`、`WandbVisBackend` 和 `TensorboardVisBackend` 等都是继承自 `BaseVisBackend`，并根据自身特性实现了对应的存储功能。用户也可以继承 `BaseVisBackend` 从而扩展存储后端，实现自定义存储需求。关于其用法，可以参考*存储后端用户教程*。

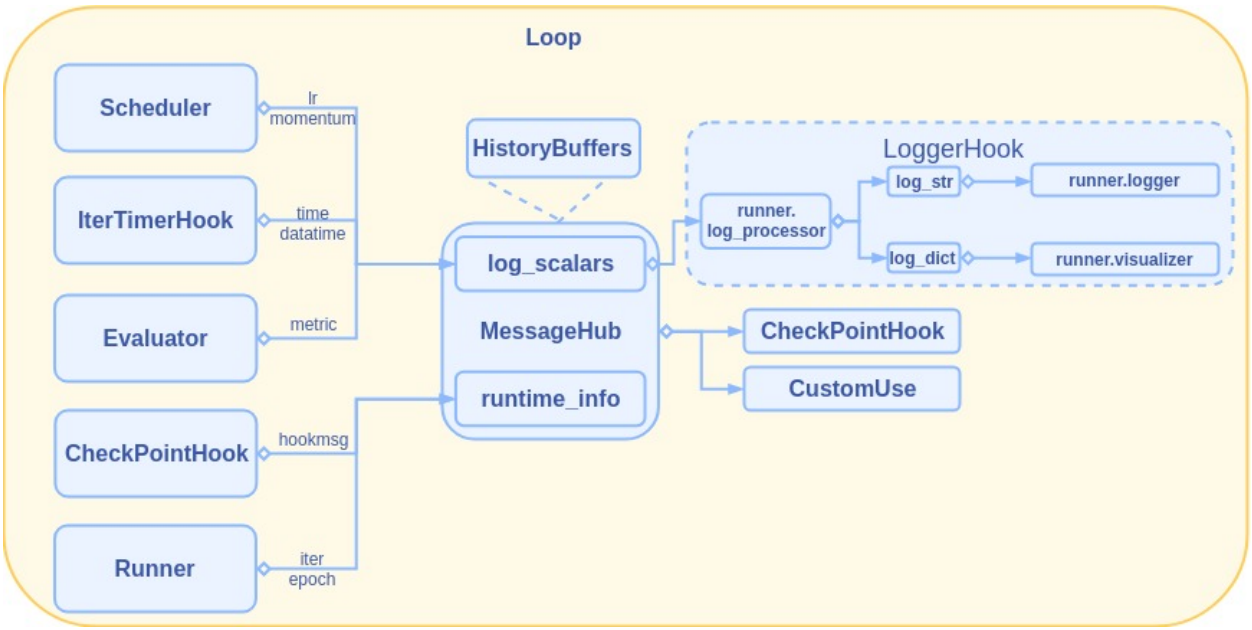
CHAPTER 30

日志系统

30.1 概述

执行器（*Runner*）在运行过程中会产生很多日志，例如加载的数据集信息、模型的初始化信息、训练过程中的学习率、损失等。为了让用户能够更加自由的获取这些日志信息，MMEngine 实现了消息枢纽（*MessageHub*）、历史缓冲区（*HistoryBuffer*）、日志处理器（*LogProcessor*）和*MMLogger* 来支持以下功能：

- 用户可以通过配置文件，根据个人偏好来选择日志统计方式，例如在终端输出整个训练过程中的平均损失而不是基于固定迭代次数平滑的损失
- 用户可以在任意组件中获取当前的训练状态，例如当前的迭代次数、训练轮次等
- 用户可以通过配置文件来控制是否保存分布式训练下的多进程日志



训练过程中的产生的损失、学习率等数据由历史缓冲区管理和封装，汇总后交给消息枢纽维护。日志处理器将消息枢纽中的数据进行格式化，最后通过记录器钩子（LoggerHook）展示到各种可视化后端。**一般情况下用户无需感知数据处理流程，可以直接通过配置日志处理器来选择日志的统计方式**。在介绍 MMEngine 的日志系统的设计之前，可以先阅读记录日志教程了解日志系统的基本用法。

30.2 历史缓冲区（HistoryBuffer）

MMEngine 实现了历史数据存储的抽象类历史缓冲区（HistoryBuffer），用于存储训练日志的历史轨迹，如模型损失、优化器学习率、迭代时间等。通常情况下，历史缓冲区作为内部类，配合消息枢纽（*MessageHub*）、记录器钩子（LoggerHook ）和日志处理器（*LogProcessor*）实现了训练日志的可配置化。

用户也可以单独使用历史缓冲区来管理训练日志，能够非常简单的使用不同方法来统计训练日志。我们先来介绍如何单独使用历史缓冲区，在消息枢纽一节再进一步介绍二者的联动。

30.2.1 历史缓冲区初始化

历史缓冲区的初始化可以接受 `log_history` 和 `count_history` 两个参数。`log_history` 表示日志的历史轨迹，例如前三次迭代的 loss 为 0.3，0.2，0.1。我们就可以记 `log_history=[0.3, 0.2, 0.3]`。`count_history` 是一个比较抽象的概念，如果按照迭代次数来算，0.3，0.2，0.1 分别是三次迭代的结果，那么我们可以记 `count_history=[1, 1, 1]`，其中“1”表示一次迭代。如果按照 batch 来算，例如每次迭代的 `batch_size` 为 8，那么 `count_history=[8, 8, 8]`。`count_history` 只会在统计均值时用到，用于控制返回均值的粒度。就拿上面那个例子来说，`count_history=[1, 1, 1]` 时会统计每次迭代的平均 loss，而 `count_history=[8, 8, 8]` 则会统计每张图片的平均 loss。

```
from mmengine.logging import HistoryBuffer

history_buffer = HistoryBuffer()  # 空初始化
log_history, count_history = history_buffer.data
# [] []
history_buffer = HistoryBuffer([1, 2, 3], [1, 2, 3])  # list 初始化
log_history, count_history = history_buffer.data
# [1 2 3] [1 2 3]
history_buffer = HistoryBuffer([1, 2, 3], [1, 2, 3], max_length=2)
# The length of history buffer(3) exceeds the max_length(2), the first few elements␣
↪will be ignored.
log_history, count_history = history_buffer.data  # 最大长度为 2, 只能存储 [2, 3]
# [2 3] [2 3]
```

我们可以通过 `history_buffer.data` 来返回日志的历史轨迹。此外，我们可以为历史缓冲区设置最大队列长度，当历史缓冲区的长度大于最大队列长度时，会自动丢弃最早更新的数据。

30.2.2 更新历史缓冲区

我们可以通过 `update` 接口来更新历史缓冲区。update 接受两个参数，第一个参数用于更新 `log_history`，第二个参数用于更新 `count_history`。

```
history_buffer = HistoryBuffer([1, 2, 3], [1, 1, 1])
history_buffer.update(4)  # 更新日志
log_history, count_history = history_buffer.data
# [1, 2, 3, 4] [1, 1, 1, 1]
history_buffer.update(5, 2)  # 更新日志
log_history, count_history = history_buffer.data
# [1, 2, 3, 4, 5] [1, 1, 1, 1, 2]
```

30.2.3 基本统计方法

历史缓冲区提供了基本的数据统计方法：

- `current()`：获取最新更新的数据。
- `mean(window_size=None)`：获取窗口内数据的均值，默认返回数据的全局均值
- `max(window_size=None)`：获取窗口内数据的最大值，默认返回全局最大值
- `min(window_size=None)`：获取窗口内数据的最小值，默认返回全局最小值

```
history_buffer = HistoryBuffer([1, 2, 3], [1, 1, 1])
history_buffer.min(2)
```

(下页继续)

(续上页)

```
# 2, 从 [2, 3] 中统计最小值
history_buffer.min()
# 返回全局最小值 1

history_buffer.max(2)
# 3, 从 [2, 3] 中统计最大值
history_buffer.min()
# 返回全局最大值 3
history_buffer.mean(2)
# 2.5, 从 [2, 3] 中统计均值, (2 + 3) / (1 + 1)
history_buffer.mean()  # (1 + 2 + 3) / (1 + 1 + 1)
# 返回全局均值 2
history_buffer = HistoryBuffer([1, 2, 3], [2, 2, 2])  # 当 count 不为 1 时
history_buffer.mean()  # (1 + 2 + 3) / (2 + 2 + 2)
# 返回均值 1
history_buffer = HistoryBuffer([1, 2, 3], [1, 1, 1])
history_buffer.update(4, 1)
history_buffer.current()
# 4
```

30.2.4 注册统计方法

为了保证历史缓冲区的可扩展性，用户可以通过 `register_statistics` 接口注册自定义的统计函数

```
from mmengine.logging import HistoryBuffer
import numpy as np


@HistoryBuffer.register_statistics
def weighted_mean(self, window_size, weight):
    assert len(weight) == window_size
    return (self._log_history[-window_size:] * np.array(weight)).sum() / \
            self._count_history[-window_size:]


history_buffer = HistoryBuffer([1, 2], [1, 1])
history_buffer.statistics('weighted_mean', 2, [2, 1])  # get (2 * 1 + 1 * 2) / (1 + 1)
```

用户可以通过 `statistics` 接口，传入方法名和对应参数来调用被注册的函数。

30.2.5 统计方法的统一入口

要想支持在配置文件中通过配置‘max’,’ min’等字段来选择日志的统计方式，那么 HistoryBuffer 就需要一个接口来接受‘min’,’ max’等统计方法字符串和相应参数，进而找到对应的统计方法，最后输出统计结果。`statistics(name, *args, **kwargs)` 接口就起到了这个作用。其中 name 是已注册的方法名（已经注册 `min`，`max` 等基本统计方法），`*arg` 和 `**kwarg` 用于接受对应方法的参数。

```
history_buffer = HistoryBuffer([1, 2, 3], [1, 1, 1])
history_buffer.statistics('mean')
# 2 返回全局均值
history_buffer.statistics('mean', 2)
# 2.5 返回 [2, 3] 的均值
history_buffer.statistics('mean', 2, 3)  # 错误! 传入了不匹配的参数
history_buffer.statistics('data')  # 错误!  data 方法未被注册，无法被调用
```

30.2.6 使用样例

用户可以独立使用历史缓冲区来记录日志，通过简单的接口调用就能得到期望的统计接口。

```
logs = dict(lr=HistoryBuffer(), loss=HistoryBuffer())  # 字典配合 HistoryBuffer 记录不同字段的日志
max_iter = 10
log_interval = 5
for iter in range(1, max_iter+1):
    lr = iter / max_iter * 0.1  # 线性学习率变化
    loss = 1 / iter  # loss
    logs['lr'].update(lr, 1)
    logs['loss'].update(loss, 1)
    if iter % log_interval == 0:
        latest_lr = logs['lr'].statistics('current')  # 通过字符串来选择统计方法
        mean_loss = logs['loss'].statistics('mean', log_interval)
        print(f'lr:   {latest_lr}\n'  # 返回最近一次更新的学习率。
              f'loss: {mean_loss}')  # 平滑最新更新的 log_interval 个数据。
# lr:   0.05
# loss: 0.45666666666666667
# lr:   0.1
# loss: 0.12912698412698415
```

MMEngine 利用历史缓冲区的特性，结合消息枢纽，实现了训练日志的高度可定制化。

30.3 消息枢纽（MessageHub）

历史缓冲区（HistoryBuffer）可以十分简单地完成单个日志的更新和统计，而在模型训练过程中，日志的种类繁多，并且来自于不同的组件，因此如何完成日志的分发和收集是需要考虑的问题。MMEngine 使用消息枢纽（MessageHub）来实现组件与组件、执行器与执行器之间的数据共享。消息枢纽继承自全局管理器（ManageMixin），支持跨模块访问。

消息枢纽存储了两种含义的数据：

- 历史缓冲区字典：消息枢纽会收集各个模块更新的训练日志，如损失、学习率、迭代时间，并将其更新至内部的历史缓冲区字典中。历史缓冲区字典经*消息处理器（LogProcessor）*处理后，会被输出到终端/保存到本地。如果用户需要记录自定义日志，可以往历史缓冲区字典中更新相应内容。
- 运行时信息字典：运行时信息字典用于存储迭代次数、训练轮次等运行时信息，方便 MMEngine 中所有组件共享这些信息。

注解： 当用户想在终端输出自定义日志，或者想跨模块共享一些自定义数据时，才会用到消息枢纽。

为了方便用户理解消息枢纽在训练过程中更新信息以及分发信息的流程，我们通过几个例子来介绍消息枢纽的使用方法，以及如何使用消息枢纽向终端输出自定义日志。

30.3.1 更新/获取训练日志

历史缓冲区以字典的形式存储在消息枢纽中。当我们第一次调用 `update_scalar` 时，会初始化对应字段的历史缓冲区，后续的每次更新等价于调用对应字段历史缓冲区的 `update` 方法。同样的我们可以通过 `get_scalar` 来获取对应字段的历史缓冲区，并按需计算统计值。如果想获取消息枢纽的全部日志，可以访问其 `log_scalars` 属性。

```
from mmengine import MessageHub

message_hub = MessageHub.get_instance('task')
message_hub.update_scalar('train/loss', 1, 1)
message_hub.get_scalar('train/loss').current()  # 1, 最近一次更新值为 1
message_hub.update_scalar('train/loss', 3, 1)
message_hub.get_scalar('train/loss').mean()  # 2, 均值为 (3 + 1) / (1 + 1)
message_hub.update_scalar('train/lr', 0.1, 1)

message_hub.update_scalars({'train/time': {'value': 0.1, 'count': 1},
                            'train/data_time': {'value': 0.1, 'count': 1}})

train_time = message_hub.get_scalar('train/time')  # 获取单个日志

log_dict = message_hub.log_scalars  # 返回存储全部 HistoryBuffer 的字典
```

(下页继续)

(续上页)

```
lr_buffer, loss_buffer, time_buffer, data_time_buffer = (
    log_dict['train/lr'], log_dict['train/loss'], log_dict['train/time'],
    log_dict['train/data_time'])
```

损失、学习率、迭代时间等训练日志在执行器和钩子中自动更新，无需用户维护。

注解： 消息枢纽的历史缓冲区字典对 key 没有特殊要求，但是 MMEngine 约定历史缓冲区字典的 key 要有 train/val/test 的前缀，只有带前缀的日志会被输出当终端。

30.3.2 更新/获取运行时信息

运行时信息以字典的形式存储在消息枢纽中，能够存储任意数据类型，每次更新都会被覆盖。

```
message_hub = MessageHub.get_instance('task')
message_hub.update_info('iter', 1)
message_hub.get_info('iter')  # 1
message_hub.update_info('iter', 2)
message_hub.get_info('iter')  # 2 覆盖上一次结果
```

30.3.3 消息枢纽的跨组件通讯

执行器运行过程中，各个组件会通过消息枢纽来分发、接受消息。*RuntimeInfoHook* 会汇总其他组件更新的学习率、损失等信息，将其导出到用户指定的输出端（Tensorboard，Wandb 等）。由于上述流程较为复杂，这里用一个简单示例来模拟日志钩子和其他组件通讯的过程。

```
from mmengine import MessageHub

class LogProcessor:
    # 汇总不同模块更新的消息，类似 LoggerHook
    def __init__(self, name):
        self.message_hub = MessageHub.get_instance(name)  # 获取 MessageHub

    def run(self):
        print(f"Learning rate is {self.message_hub.get_scalar('train/lr').current()}")
        print(f"loss is {self.message_hub.get_scalar('train/loss').current()}")
        print(f"meta is {self.message_hub.get_info('meta')}")


class LrUpdater:
    # 更新学习率
```

(下页继续)

(续上页)

```
    def __init__(self, name):
        self.message_hub = MessageHub.get_instance(name)  # 获取 MessageHub

    def run(self):
        self.message_hub.update_scalar('train/lr', 0.001)
        # 更新学习率，以 HistoryBuffer 形式存储


class MetaUpdater:
    # 更新元信息
    def __init__(self, name):
        self.message_hub = MessageHub.get_instance(name)

    def run(self):
        self.message_hub.update_info(
            'meta',
            dict(experiment='retinanet_r50_caffe_fpn_1x_coco.py',
                 repo='mmdetection'))     # 更新元信息，每次更新会覆盖上一次的信息


class LossUpdater:
    # 更新损失函数
    def __init__(self, name):
        self.message_hub = MessageHub.get_instance(name)

    def run(self):
        self.message_hub.update_scalar('train/loss', 0.1)

class ToyRunner:
    # 组合个各个模块
    def __init__(self, name):
        self.message_hub = MessageHub.get_instance(name)  # 创建 MessageHub
        self.log_processor = LogProcessor(name)
        self.updaters = [LossUpdater(name),
                         MetaUpdater(name),
                         LrUpdater(name)]

    def run(self):
        for updater in self.updaters:
            updater.run()
        self.log_processor.run()

if __name__ == '__main__':
```

(下页继续)

(续上页)

```
    task = ToyRunner('name')
    task.run()
    # Learning rate is 0.001
    # loss is 0.1
    # meta {'experiment': 'retinanet_r50_caffe_fpn_1x_coco.py', 'repo': 'mmdetection'}
```

30.3.4 添加自定义日志

我们可以在任意模块里更新消息枢纽的历史缓冲区字典，历史缓冲区字典中所有的字段经统计后最后显示到终端。

注解： 更新历史缓冲区字典时，需要保证更新的日志名带有 train，val，test 前缀，否则日志不会在终端显示。

```
class CustomModule:
    def __init__(self):
        self.message_hub = MessageHub.get_current_instance()

    def custom_method(self):
        self.message_hub.update_scalar('train/a', 100)
        self.message_hub.update_scalars({'train/b': 1, 'train/c': 2})
```

默认情况下，终端上额外显示 a、b、c 最后一次更新的结果。我们也可以通过配置日志处理器来切换自定义日志的统计方式。

30.4 日志处理器（LogProcessor）

用户可以通过配置日志处理器（LogProcessor）来控制日志的统计方法及其参数。默认配置下，日志处理器会统计最近一次更新的学习率、基于迭代次数平滑的损失和迭代时间。用户可以在日志处理器中配置已知字段的统计方式。

30.4.1 最简配置

```
log_processor = dict(
    window_size=10,
)
```

此时终端会输出每 10 次迭代的平均损失和平均迭代时间。假设此时终端的输出为

```
04/15 12:34:24 - mmengine - INFO - Iter [10/12]  , eta: 0:00:00, time: 0.003, data_
↪time: 0.002, loss: 0.13
```

30.4.2 自定义的统计方式

我们可以通过配置 custom_cfg 列表来选择日志的统计方式。custom_cfg 中的每一个元素需要包括以下信息：

- data_src：日志的数据源，用户通过指定 data_src 来选择需要被重新统计的日志，一份数据源可以有多种统计方式。默认的日志源包括模型输出的损失字典的 key、学习率（lr）和迭代时间（time/data_time），一切经消息枢纽的 update_scalar/update_scalars 更新的日志均为可以配置的数据源（需要去掉 train/、val/ 前缀）。（必填项）
- method_name：日志的统计方法，即历史缓冲区中的基本统计方法以及用户注册的自定义统计方法（必填项）
- log_name：日志被重新统计后的名字，如果不定义 log_name，新日志会覆盖旧日志（选填项）
- 其他参数：统计方法会用到的参数，其中 window_size 为特殊字段，可以为普通的整型、字符串 epoch 和字符串 global。LogProcessor 会实时解析这些参数，以返回基于 iteration、epoch 和全局平滑的统计结果（选填项）

1. 覆盖旧的统计方式

```
log_processor = dict(
    window_size=10,
    by_epoch=True,
    custom_cfg=[
        dict(data_src='loss',
             method_name='mean',
             window_size=100)])
```

此时会无视日志处理器的默认窗口 10，用更大的窗口 100 去统计 loss 的均值，并将原有结果覆盖。

```
04/15 12:34:24 - mmengine - INFO - Iter [10/12]  , eta: 0:00:00, time: 0.003, data_
↪time: 0.002, loss: 0.11
```

2. 新增统计方式，不覆盖

```
log_processor = dict(
    window_size=10,
    by_epoch=True,
    custom_cfg=[
        dict(data_src='loss',
             log_name='loss_min',
```

(下页继续)

(续上页)

```
            method_name='min',
            window_size=100)])
```

```
04/15 12:34:24 - mmengine - INFO - Iter [10/12]  , eta: 0:00:00, time: 0.003, data_
↪time: 0.002, loss: 0.11, loss_min: 0.08
```

30.5 MMLogger

为了能够导出层次分明、格式统一、且不受三方库日志系统干扰的日志，MMEngine 在 `logging` 模块的基础上实现了 `MMLogger`。`MMLogger` 继承自全局管理器（`ManagerMixin`），相比于 `logging.Logger`，`MMLogger` 能够在无法获取 `logger` 的名字（logger name）的情况下，拿到当前执行器的 `logger`。

30.5.1 创建 MMLogger

我们可以通过 `get_instance` 接口创建全局可获取的 `logger`，默认的日志格式如下

```
logger = MMLogger.get_instance('mmengine', log_level='INFO')
logger.info("this is a test")
# 04/15 14:01:11 - mmengine - INFO - this is a test
```

`logger` 除了输出消息外，还会额外输出时间戳、logger 的名字和日志等级。对于 ERROR 等级的日志，我们会用红色高亮日志等级，并额外输出错误日志的代码位置

```
logger = MMLogger.get_instance('mmengine', log_level='INFO')
logger.error('division by zero')
# 04/15 14:01:56 - mmengine - ERROR - /mnt/d/PythonCode/DeepLearning/OpenMMLab/
↪mmengine/a.py - <module> - 4 - division by zero
```

30.5.2 导出日志

调用 `get_instance` 时，如果指定了 log_file，会将日志记录的信息以文本格式导出到本地。

```
logger = MMLogger.get_instance('mmengine', log_file='tmp.log', log_level='INFO')
logger.info("this is a test")
# 04/15 14:01:11 - mmengine - INFO - this is a test
```

tmp/tmp.log:

```
04/15 14:01:11 - mmengine - INFO - this is a test
```

由于分布式情况下会创建多个日志文件，因此我们在预定的导出路径下，增加一级和导出文件同名的目录，用于存储所有进程的日志。上例中导出路径为 `tmp.log`，实际存储路径为 `tmp/tmp.log`。

30.5.3 分布式训练时导出日志

使用 pytorch 分布式训练时，我们可以通过配置 `distributed=True` 来导出分布式训练时各个进程的日志（默认关闭）。

```
logger = MMLogger.get_instance('mmengine', log_file='tmp.log', distributed=True, log_
↪level='INFO')
```

单机多卡，或者多机多卡但是共享存储的情况下，导出的分布式日志路径如下

```
#  共享存储
./tmp
├── tmp.log
├── tmp_rank1.log
├── tmp_rank2.log
├── tmp_rank3.log
├── tmp_rank4.log
├── tmp_rank5.log
├── tmp_rank6.log
└── tmp_rank7.log
...
└── tmp_rank63.log
```

多机多卡，独立存储的情况：

```
# 独立存储
# 设备 0:
work_dir/
└── exp_name_logs
    ├── exp_name.log
    ├── exp_name_rank1.log
    ├── exp_name_rank2.log
    ├── exp_name_rank3.log
    ...
    └── exp_name_rank7.log

# 设备 7:
work_dir/
└── exp_name_logs
    ├── exp_name_rank56.log
    ├── exp_name_rank57.log
```

（下页继续）

(续上页)

```
    ├── exp_name_rank58.log
    ...
    └── exp_name_rank63.log
```

CHAPTER 31

迁移 MMCV 执行器到 MMEngine

31.1 简介

随着支持的深度学习任务越来越多，用户的需求不断增加，我们对 MMCV 已有的执行器（Runner）的灵活性和通用性有了更高的要求。因此，MMEngine 在 MMCV 的基础上，实现了一个更加通用灵活的执行器以支持更多复杂的模型训练流程。MMEngine 中的执行器扩大了作用域，也承担了更多的功能；我们抽象出了训练循环控制器（EpochBasedTrainLoop/IterBasedTrainLoop）、验证循环控制器（*ValLoop*）和测试循环控制器（*TestLoop*）来方便用户灵活拓展模型的执行流程。

我们将首先介绍算法库的执行入口该如何从 MMCV 迁移到 MMEngine，以最大程度地简化和统一执行入口的代码。然后我们将详细介绍在 MMCV 和 MMEngine 中构造执行器及其内部组件进行训练的差异。在开始迁移前，我们建议用户先阅读执行器教程。

31.2 执行入口

以 MMDet 为例，我们首先展示基于 MMEngine 重构前后，配置文件和训练启动脚本的区别：

31.2.1 配置文件的迁移

```
checkpoint_config = dict(interval=1)
# yapf:disable
log_config = dict(
    interval=50,
    hooks=[
        dict(type='TextLoggerHook'),
        # dict(type='TensorboardLoggerHook')
    ])
# yapf:enable
custom_hooks = [dict(type='NumClassCheckHook')]

dist_params = dict(backend='nccl')
log_level = 'INFO'
load_from = None
resume_from = None
workflow = [('train', 1)]


opencv_num_threads = 0
mp_start_method = 'fork'
auto_scale_lr = dict(enable=False, base_batch_size=16)
```

```
default_scope = 'mmdet'

default_hooks = dict(
    timer=dict(type='IterTimerHook'),
    logger=dict(type='LoggerHook', interval=50),
    param_scheduler=dict(type='ParamSchedulerHook'),
    checkpoint=dict(type='CheckpointHook', interval=1),
    sampler_seed=dict(type='DistSamplerSeedHook'),
    visualization=dict(type='DetVisualizationHook'))

env_cfg = dict(
    cudnn_benchmark=False,
    mp_cfg=dict(mp_start_method='fork', opencv_num_threads=0),
    dist_cfg=dict(backend='nccl'),
)
```

(下页继续)

(续上页)

```
vis_backends = [dict(type='LocalVisBackend')]
visualizer = dict(
    type='DetLocalVisualizer', vis_backends=vis_backends, name='visualizer')
log_processor = dict(type='LogProcessor', window_size=50, by_epoch=True)

log_level = 'INFO'
load_from = None
resume = False
```

```
# optimizer
optimizer = dict(type='SGD', lr=0.02, momentum=0.9, weight_decay=0.0001)
optimizer_config = dict(grad_clip=None)
# learning policy
lr_config = dict(
    policy='step',
    warmup='linear',
    warmup_iters=500,
    warmup_ratio=0.001,
    step=[8, 11])
runner = dict(type='EpochBasedRunner', max_epochs=12)
```

```
# training schedule for 1x
train_cfg = dict(type='EpochBasedTrainLoop', max_epochs=12, val_interval=1)
val_cfg = dict(type='ValLoop')
test_cfg = dict(type='TestLoop')

# learning rate
param_scheduler = [
    dict(
        type='LinearLR', start_factor=0.001, by_epoch=False, begin=0, end=500),
    dict(
        type='MultiStepLR',
        begin=0,
        end=12,
        by_epoch=True,
        milestones=[8, 11],
        gamma=0.1)
]

# optimizer
optim_wrapper = dict(
    type='OptimWrapper',
```

(下页继续)

(续上页)

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

# Default setting for scaling LR automatically
#   - `enable` means enable scaling LR automatically
#       or not by default.
#   - `base_batch_size` = (8 GPUs) x (2 samples per GPU).
auto_scale_lr = dict(enable=False, base_batch_size=16)
```

```
# dataset settings
dataset_type = 'CocoDataset'
data_root = 'data/coco/'
img_norm_cfg = dict(
    mean=[123.675, 116.28, 103.53], std=[58.395, 57.12, 57.375], to_rgb=True)
train_pipeline = [
    dict(type='LoadImageFromFile'),
    dict(type='LoadAnnotations', with_bbox=True),
    dict(type='Resize', img_scale=(1333, 800), keep_ratio=True),
    dict(type='RandomFlip', flip_ratio=0.5),
    dict(type='Normalize', **img_norm_cfg),
    dict(type='Pad', size_divisor=32),
    dict(type='DefaultFormatBundle'),
    dict(type='Collect', keys=['img', 'gt_bboxes', 'gt_labels']),
]
test_pipeline = [
    dict(type='LoadImageFromFile'),
    dict(
        type='MultiScaleFlipAug',
        img_scale=(1333, 800),
        flip=False,
        transforms=[
            dict(type='Resize', keep_ratio=True),
            dict(type='RandomFlip'),
            dict(type='Normalize', **img_norm_cfg),
            dict(type='Pad', size_divisor=32),
            dict(type='ImageToTensor', keys=['img']),
            dict(type='Collect', keys=['img']),
        ])
]
data = dict(
    samples_per_gpu=2,
    workers_per_gpu=2,
    train=dict(
        type=dataset_type,
```

(下页继续)

(续上页)

```
        ann_file=data_root + 'annotations/instances_train2017.json',
        img_prefix=data_root + 'train2017/',
        pipeline=train_pipeline),
    val=dict(
        type=dataset_type,
        ann_file=data_root + 'annotations/instances_val2017.json',
        img_prefix=data_root + 'val2017/',
        pipeline=test_pipeline),
    test=dict(
        type=dataset_type,
        ann_file=data_root + 'annotations/instances_val2017.json',
        img_prefix=data_root + 'val2017/',
        pipeline=test_pipeline))
evaluation = dict(interval=1, metric='bbox')
```

```
# dataset settings
dataset_type = 'CocoDataset'
data_root = 'data/coco/'

file_client_args = dict(backend='disk')

train_pipeline = [
    dict(type='LoadImageFromFile', file_client_args=file_client_args),
    dict(type='LoadAnnotations', with_bbox=True),
    dict(type='Resize', scale=(1333, 800), keep_ratio=True),
    dict(type='RandomFlip', prob=0.5),
    dict(type='PackDetInputs')
]
test_pipeline = [
    dict(type='LoadImageFromFile', file_client_args=file_client_args),
    dict(type='Resize', scale=(1333, 800), keep_ratio=True),
    # If you don't have a gt annotation, delete the pipeline
    dict(type='LoadAnnotations', with_bbox=True),
    dict(
        type='PackDetInputs',
        meta_keys=('img_id', 'img_path', 'ori_shape', 'img_shape',
                   'scale_factor'))
]
train_dataloader = dict(
    batch_size=2,
    num_workers=2,
    persistent_workers=True,
    sampler=dict(type='DefaultSampler', shuffle=True),
```

(下页继续)

(续上页)

```
    batch_sampler=dict(type='AspectRatioBatchSampler'),
    dataset=dict(
        type=dataset_type,
        data_root=data_root,
        ann_file='annotations/instances_train2017.json',
        data_prefix=dict(img='train2017/'),
        filter_cfg=dict(filter_empty_gt=True, min_size=32),
        pipeline=train_pipeline))
val_dataloader = dict(
    batch_size=1,
    num_workers=2,
    persistent_workers=True,
    drop_last=False,
    sampler=dict(type='DefaultSampler', shuffle=False),
    dataset=dict(
        type=dataset_type,
        data_root=data_root,
        ann_file='annotations/instances_val2017.json',
        data_prefix=dict(img='val2017/'),
        test_mode=True,
        pipeline=test_pipeline))
test_dataloader = val_dataloader

val_evaluator = dict(
    type='CocoMetric',
    ann_file=data_root + 'annotations/instances_val2017.json',
    metric='bbox',
    format_only=False)
test_evaluator = val_evaluator
```

MMEngine 中的执行器提供了更多可自定义的部分，包括训练、验证、测试过程和数据加载器的配置，因此配置文件和之前相比会长一些。为了方便用户的理解和阅读，我们遵循所见即所得的原则，重新调整了各个组件配置的层次，使得大部分一级字段都对应着执行器中关键属性的配置，例如数据加载器、评测器、流程配置、钩子配置等。这些配置在 OpenMMLab 2.0 算法库中都有默认配置，因此用户很多时候无需关心其中的大部分参数。

31.2.2 启动脚本的迁移

相比于 MMCV 的执行器，MMEngine 的执行器可以承担更多的功能，例如构建 DataLoader，构建分布式模型等。因此我们需要在配置文件中指定更多的参数，例如 DataLoader 的 sampler 和 batch_sampler，而无需在训练的启动脚本里实现构建 DataLoader 相关的代码。以 MMDet 的训练启动脚本为例:

```
args = parse_args()

cfg = Config.fromfile(args.config)

# replace the ${key} with the value of cfg.key
cfg = replace_cfg_vals(cfg)

# update data root according to MMDET_DATASETS
update_data_root(cfg)

if args.cfg_options is not None:
    cfg.merge_from_dict(args.cfg_options)

if args.auto_scale_lr:
    if 'auto_scale_lr' in cfg and \
            'enable' in cfg.auto_scale_lr and \
            'base_batch_size' in cfg.auto_scale_lr:
        cfg.auto_scale_lr.enable = True
    else:
        warnings.warn('Can not find "auto_scale_lr" or '
                      '"auto_scale_lr.enable" or '
                      '"auto_scale_lr.base_batch_size" in your'
                      ' configuration file. Please update all the '
                      'configuration files to mmdet >= 2.24.1.')

# set multi-process settings
setup_multi_processes(cfg)

# set cudnn_benchmark
if cfg.get('cudnn_benchmark', False):
    torch.backends.cudnn.benchmark = True

# work_dir is determined in this priority: CLI > segment in file > filename
if args.work_dir is not None:
    # update configs according to CLI args if args.work_dir is not None
    cfg.work_dir = args.work_dir
elif cfg.get('work_dir', None) is None:
    # use config filename as default work_dir if cfg.work_dir is None
```

(下页继续)

(续上页)

```
    cfg.work_dir = osp.join('./work_dirs',
                            osp.splitext(osp.basename(args.config))[0])

if args.resume_from is not None:
    cfg.resume_from = args.resume_from
cfg.auto_resume = args.auto_resume
if args.gpus is not None:
    cfg.gpu_ids = range(1)
    warnings.warn('`--gpus` is deprecated because we only support '
                  'single GPU mode in non-distributed training. '
                  'Use `gpus=1` now.')
if args.gpu_ids is not None:
    cfg.gpu_ids = args.gpu_ids[0:1]
    warnings.warn('`--gpu-ids` is deprecated, please use `--gpu-id`. '
                  'Because we only support single GPU mode in '
                  'non-distributed training. Use the first GPU '
                  'in `gpu_ids` now.')
if args.gpus is None and args.gpu_ids is None:
    cfg.gpu_ids = [args.gpu_id]

# init distributed env first, since logger depends on the dist info.
if args.launcher == 'none':
    distributed = False
else:
    distributed = True
    init_dist(args.launcher, **cfg.dist_params)
    # re-set gpu_ids with distributed training mode
    _, world_size = get_dist_info()
    cfg.gpu_ids = range(world_size)

# create work_dir
mmcv.mkdir_or_exist(osp.abspath(cfg.work_dir))
# dump config
cfg.dump(osp.join(cfg.work_dir, osp.basename(args.config)))
# init the logger before other steps
timestamp = time.strftime('%Y%m%d_%H%M%S', time.localtime())
log_file = osp.join(cfg.work_dir, f'{timestamp}.log')
logger = get_root_logger(log_file=log_file, log_level=cfg.log_level)

# init the meta dict to record some important information such as
# environment info and seed, which will be logged
meta = dict()
# log env info
```

(下页继续)

(续上页)

```
env_info_dict = collect_env()
env_info = '\n'.join([(f'{k}: {v}') for k, v in env_info_dict.items()])
dash_line = '-' * 60 + '\n'
logger.info('Environment info:\n' + dash_line + env_info + '\n' +
            dash_line)
meta['env_info'] = env_info
meta['config'] = cfg.pretty_text
# log some basic info
logger.info(f'Distributed training: {distributed}')
logger.info(f'Config:\n{cfg.pretty_text}')

cfg.device = get_device()
# set random seeds
seed = init_random_seed(args.seed, device=cfg.device)
seed = seed + dist.get_rank() if args.diff_seed else seed
logger.info(f'Set random seed to {seed}, '
            f'deterministic: {args.deterministic}')
set_random_seed(seed, deterministic=args.deterministic)
cfg.seed = seed
meta['seed'] = seed
meta['exp_name'] = osp.basename(args.config)

model = build_detector(
    cfg.model,
    train_cfg=cfg.get('train_cfg'),
    test_cfg=cfg.get('test_cfg'))
model.init_weights()

datasets = []
train_detector(
    model,
    datasets,
    cfg,
    distributed=distributed,
    validate=(not args.no_validate),
    timestamp=timestamp,
    meta=meta)
```

```
args = parse_args()

# register all modules in mmdet into the registries
# do not init the default scope here because it will be init in the runner
register_all_modules(init_default_scope=False)
```

(下页继续)

(续上页)

```
# load config
cfg = Config.fromfile(args.config)
cfg.launcher = args.launcher
if args.cfg_options is not None:
    cfg.merge_from_dict(args.cfg_options)

# work_dir is determined in this priority: CLI > segment in file > filename
if args.work_dir is not None:
    # update configs according to CLI args if args.work_dir is not None
    cfg.work_dir = args.work_dir
elif cfg.get('work_dir', None) is None:
    # use config filename as default work_dir if cfg.work_dir is None
    cfg.work_dir = osp.join('./work_dirs',
                            osp.splitext(osp.basename(args.config))[0])

# enable automatic-mixed-precision training
if args.amp is True:
    optim_wrapper = cfg.optim_wrapper.type
    if optim_wrapper == 'AmpOptimWrapper':
        print_log(
            'AMP training is already enabled in your config.',
            logger='current',
            level=logging.WARNING)
    else:
        assert optim_wrapper == 'OptimWrapper', (
            '`--amp` is only supported when the optimizer wrapper type is '
            f'`OptimWrapper` but got {optim_wrapper}.')
        cfg.optim_wrapper.type = 'AmpOptimWrapper'
        cfg.optim_wrapper.loss_scale = 'dynamic'

# enable automatically scaling LR
if args.auto_scale_lr:
    if 'auto_scale_lr' in cfg and \
            'enable' in cfg.auto_scale_lr and \
            'base_batch_size' in cfg.auto_scale_lr:
        cfg.auto_scale_lr.enable = True
    else:
        raise RuntimeError('Can not find "auto_scale_lr" or '
                           '"auto_scale_lr.enable" or '
                           '"auto_scale_lr.base_batch_size" in your'
                           ' configuration file.')
```

(下页继续)

(续上页)

```
cfg.resume = args.resume

# build the runner from config
if 'runner_type' not in cfg:
    # build the default runner
    runner = Runner.from_cfg(cfg)
else:
    # build customized runner from the registry
    # if 'runner_type' is set in the cfg
    runner = RUNNERS.build(cfg)

# start training
runner.train()
```

```
def init_random_seed(...):
    ...

def set_random_seed(...):
    ...

# define function tools.
...


def train_detector(model,
                   dataset,
                   cfg,
                   distributed=False,
                   validate=False,
                   timestamp=None,
                   meta=None):

    cfg = compat_cfg(cfg)
    logger = get_root_logger(log_level=cfg.log_level)

    # put model on gpus
    if distributed:
        find_unused_parameters = cfg.get('find_unused_parameters', False)
        # Sets the `find_unused_parameters` parameter in
        # torch.nn.parallel.DistributedDataParallel
        model = build_ddp(
            model,
            cfg.device,
```

(下页继续)

(续上页)

```
            device_ids=[int(os.environ['LOCAL_RANK'])],
            broadcast_buffers=False,
            find_unused_parameters=find_unused_parameters)
    else:
        model = build_dp(model, cfg.device, device_ids=cfg.gpu_ids)

    # build optimizer
    auto_scale_lr(cfg, distributed, logger)
    optimizer = build_optimizer(model, cfg.optimizer)

    runner = build_runner(
        cfg.runner,
        default_args=dict(
            model=model,
            optimizer=optimizer,
            work_dir=cfg.work_dir,
            logger=logger,
            meta=meta))

    # an ugly workaround to make .log and .log.json filenames the same
    runner.timestamp = timestamp

    # fp16 setting
    fp16_cfg = cfg.get('fp16', None)
    if fp16_cfg is not None:
        optimizer_config = Fp16OptimizerHook(
            **cfg.optimizer_config, **fp16_cfg, distributed=distributed)
    elif distributed and 'type' not in cfg.optimizer_config:
        optimizer_config = OptimizerHook(**cfg.optimizer_config)
    else:
        optimizer_config = cfg.optimizer_config

    # register hooks
    runner.register_training_hooks(
        cfg.lr_config,
        optimizer_config,
        cfg.checkpoint_config,
        cfg.log_config,
        cfg.get('momentum_config', None),
        custom_hooks_config=cfg.get('custom_hooks', None))

    if distributed:
        if isinstance(runner, EpochBasedRunner):
```

(下页继续)

(续上页)

```
            runner.register_hook(DistSamplerSeedHook())

    # register eval hooks
    if validate:
        val_dataloader_default_args = dict(
            samples_per_gpu=1,
            workers_per_gpu=2,
            dist=distributed,
            shuffle=False,
            persistent_workers=False)

        val_dataloader_args = {
            **val_dataloader_default_args,
            **cfg.data.get('val_dataloader', {})
        }
        # Support batch_size > 1 in validation

        if val_dataloader_args['samples_per_gpu'] > 1:
            # Replace 'ImageToTensor' to 'DefaultFormatBundle'
            cfg.data.val.pipeline = replace_ImageToTensor(
                cfg.data.val.pipeline)
        val_dataset = build_dataset(cfg.data.val, dict(test_mode=True))

        val_dataloader = build_dataloader(val_dataset, **val_dataloader_args)
        eval_cfg = cfg.get('evaluation', {})
        eval_cfg['by_epoch'] = cfg.runner['type'] != 'IterBasedRunner'
        eval_hook = DistEvalHook if distributed else EvalHook
        # In this PR (https://github.com/open-mmlab/mmcv/pull/1193), the
        # priority of IterTimerHook has been modified from 'NORMAL' to 'LOW'.
        runner.register_hook(
            eval_hook(val_dataloader, **eval_cfg), priority='LOW')

    resume_from = None
    if cfg.resume_from is None and cfg.get('auto_resume'):
        resume_from = find_latest_checkpoint(cfg.work_dir)
    if resume_from is not None:
        cfg.resume_from = resume_from

    if cfg.resume_from:
        runner.resume(cfg.resume_from)
    elif cfg.load_from:
        runner.load_checkpoint(cfg.load_from)
    runner.run(data_loaders, cfg.workflow)
```

```
# `apis/train.py` is removed in `mmengine`
```

上表对比了基于 MMCV 执行器和 MMEngine 执行器 MMDet 启动脚本的区别。OpenMMLab 1.x 中的算法库都实现了一套 runner 的构建和训练流程，其中存在着大量的冗余代码。因此，MMEngine 的执行器在内部实现了很多流程化的代码以统一各个算法库的执行流程，例如初始化随机种子、初始化分布式环境、构建 `DataLoader` 等，使得下游算法库从此无需在训练启动脚本里实现相关代码，只需配置执行器的构造参数，就能够执行相应的流程。基于 MMEngine 执行器的启动脚本不仅简化了 `tools/train.py` 的代码，甚至可以直接删除 `apis/train.py`，极大程度的简化了训练启动脚本。同样的，我们在基于 MMEngine 开发自己的代码仓库时，可以通过配置执行器参数来设置随机种子、初始化分布式环境，无需自行实现相关代码。

31.3 迁移执行器（Runner）

本节主要介绍 MMCV 执行器和 MMEngine 执行器在训练、验证、测试流程上的区别。在使用 MMCV 执行器和 MMEngine 执行器训练、测试模型时，以下流程有着明显的不同：

1. 准备 *logger*
2. 设置随机种子
3. 初始化环境变量
4. 准备数据
5. 准备模型
6. 准备优化器
7. 准备钩子
8. 准备验证/测试模块
9. 构建执行器
10. 开始训练、开始测试
11. 迁移自定义训练流程

后续的教程中，我们会对每个流程的差异进行详细介绍。

31.3.1 准备 logger

MMCV 准备 logger

MMCV 需要在训练脚本里调用 `get_logger` 接口获得 logger，并用它输出、记录训练环境。

```
logger = get_logger(name='custom', log_file=log_file, log_level=cfg.log_level)
env_info_dict = collect_env()
env_info = '\n'.join([(f'{k}: {v}') for k, v in env_info_dict.items()])
```

(下页继续)

(续上页)

```
dash_line = '-' * 60 + '\n'
logger.info('Environment info:\n' + dash_line + env_info + '\n' +
            dash_line)
```

执行器构造时，也需要传入 logger。

```
runner = Runner(
    ...
    logger=logger
    ...)
```

MMEngine 准备 logger

在执行器构建时传入 logger 的日志等级，执行器构建时会自动创建 logger，并输出、记录训练环境。

```
log_level = 'INFO'
```

31.3.2 设置随机种子

MMCV 设置随机种子

在训练脚本中手动设置随机种子：

```
...
seed = init_random_seed(args.seed, device=cfg.device)
seed = seed + dist.get_rank() if args.diff_seed else seed
logger.info(f'Set random seed to {seed}, '
            f'deterministic: {args.deterministic}')
set_random_seed(seed, deterministic=args.deterministic)
...
```

MMEngine 设计随机种子

配置执行器的 randomness 参数，配置规则详见*执行器 api 文档*

OpenMMLab 系列算法库配置变更

```
seed = 1
deterministic=False
diff_seed=False
```

```
randomness=dict(seed=1,
                deterministic=True,
                diff_rank_seed=False)
```

在本教程中，我们将 randomness 配置为：

```
randomness = dict(seed=5)
```

31.3.3 初始化训练环境

MMCV 初始化训练环境

MMCV 需要在训练脚本中配置多进程启动方式、多进程通信后端等环境变量，并在执行器构建之前初始化分布式环境，对模型进行分布式封装：

```
...
setup_multi_processes(cfg)
init_dist(cfg.launcher, **cfg.dist_params)
model = MMDistributedDataParallel(
    model,
    device_ids=[int(os.environ['LOCAL_RANK'])],
    broadcast_buffers=False,
    find_unused_parameters=find_unused_parameters)
```

MMEngine 初始化训练环境

MMEngine 通过配置 env_cfg 来选择多进程启动方式和多进程通信后端，其默认值为 dict(dist_cfg=dict(backend='nccl'))，配置方式详见执行器 *api* 文档。

执行器构建时接受 launcher 参数，如果其值不为 'none'，执行器构建时会自动执行分布式初始化，模型分布式封装。换句话说，使用 MMEngine 的执行器时，我们无需在执行器外做分布式相关的操作，只需配置 launcher 参数，选择训练的启动方式即可。

OpenMMLab 系列算法库配置变更

```
launcher = 'pytorch'  # 开启分布式训练
dist_params = dict(backend='nccl')  # 选择多进程通信后端
```

```
launcher = 'pytorch'
env_cfg = dict(dist_cfg=dict(backend='nccl'))
```

在本教程中，我们将 env_cfg 配置为：

```
env_cfg = dict(dist_cfg=dict(backend='nccl'))
```

31.3.4 准备数据

MMCV 和 MMEngine 的执行器均可接受构建好的 DataLoader 实例。因此准备数据的流程没有差异：

```
import torchvision.transforms as transforms
from torch.utils.data import DataLoader
from torchvision.datasets import CIFAR10

transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])

train_dataset = CIFAR10(
    root='data', train=True, download=True, transform=transform)
train_dataloader = DataLoader(
    train_dataset, batch_size=128, shuffle=True, num_workers=2)

val_dataset = CIFAR10(
    root='data', train=False, download=True, transform=transform)
val_dataloader = DataLoader(
    val_dataset, batch_size=128, shuffle=False, num_workers=2)
```

OpenMMLab 系列算法库配置变更

```
data = dict(
    samples_per_gpu=2,  # 单卡 batch_size
    workers_per_gpu=2,  # Dataloader 采样进程数
    train=dict(
        type=dataset_type,
        ann_file=data_root + 'annotations/instances_train2017.json',
        img_prefix=data_root + 'train2017/',
        pipeline=train_pipeline),
    val=dict(
        type=dataset_type,
        ann_file=data_root + 'annotations/instances_val2017.json',
        img_prefix=data_root + 'val2017/',
        pipeline=test_pipeline),
    test=dict(
        type=dataset_type,
        ann_file=data_root + 'annotations/instances_val2017.json',
        img_prefix=data_root + 'val2017/',
        pipeline=test_pipeline))
```

```
train_dataloader = dict(
    batch_size=2, # samples_per_gpu -> batch_size,
    num_workers=2,
    # 遍历完 DataLoader 后，是否重启多进程采样
    persistent_workers=True,
    # 可配置的 sampler
    sampler=dict(type='DefaultSampler', shuffle=True),
    # 可配置的 batch_sampler
    batch_sampler=dict(type='AspectRatioBatchSampler'),
    dataset=dict(
        type=dataset_type,
        data_root=data_root,
        ann_file='annotations/instances_train2017.json',
        data_prefix=dict(img='train2017/'),
        filter_cfg=dict(filter_empty_gt=True, min_size=32),
        pipeline=train_pipeline))

val_dataloader = dict(
    batch_size=1, # 验证阶段的 batch_size
    num_workers=2,
    persistent_workers=True,
    drop_last=False, # 是否丢弃最后一个 batch
    sampler=dict(type='DefaultSampler', shuffle=False),
    dataset=dict(
        type=dataset_type,
        data_root=data_root,
        ann_file='annotations/instances_val2017.json',
        data_prefix=dict(img='val2017/'),
        test_mode=True,
        pipeline=test_pipeline))

test_dataloader = val_dataloader
```

相比于 MMCV 的算法库配置，MMEngine 的配置更加复杂，但是也更加灵活。`DataLoader` 的配置过程由 `Runner` 负责，无需各个算法库实现。

31.3.5 准备模型

详见迁移 MMCV 模型至 MMEngine

```
import torch.nn as nn
import torch.nn.functional as F
from mmengine.model import BaseModel


class Model(BaseModel):

    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)
        self.loss_fn = nn.CrossEntropyLoss()

    def forward(self, img, label, mode):
        feat = self.pool(F.relu(self.conv1(img)))
        feat = self.pool(F.relu(self.conv2(feat)))
        feat = feat.view(-1, 16 * 5 * 5)
        feat = F.relu(self.fc1(feat))
        feat = F.relu(self.fc2(feat))
        feat = self.fc3(feat)
        if mode == 'loss':
            loss = self.loss_fn(feat, label)
            return dict(loss=loss)
        else:
            return [feat.argmax(1)]

model = Model()
```

需要注意的是，分布式训练时，MMCV 的执行器需要接受分布式封装后的模型，而 `MMEngine` 接受分布式封装前的模型，在执行器实例化阶段对其段进行分布式封装。

31.3.6 准备优化器

MMCV 准备优化器

MMCV 执行器构造时，可以直接接受 Pytorch 优化器，如

```
optimizer = SGD(model.parameters(), lr=0.1, momentum=0.9)
```

对于复杂配置的优化器，MMCV 需要基于优化器构造器来构建优化器：

```
optimizer_cfg = dict(
    optimizer=dict(type='SGD', lr=0.01, weight_decay=0.0001),
    paramwise_cfg=dict(norm_decay_mult=0))

def build_optimizer_constructor(cfg):
    constructor_type = cfg.get('type')
    if constructor_type in OPTIMIZER_BUILDERS:
        return build_from_cfg(cfg, OPTIMIZER_BUILDERS)
    elif constructor_type in MMCV_OPTIMIZER_BUILDERS:
        return build_from_cfg(cfg, MMCV_OPTIMIZER_BUILDERS)
    else:
        raise KeyError(f'{constructor_type} is not registered '
                       'in the optimizer builder registry.')


def build_optimizer(model, cfg):
    optimizer_cfg = copy.deepcopy(cfg)
    constructor_type = optimizer_cfg.pop('constructor',
                                         'DefaultOptimizerConstructor')
    paramwise_cfg = optimizer_cfg.pop('paramwise_cfg', None)
    optim_constructor = build_optimizer_constructor(
        dict(
            type=constructor_type,
            optimizer_cfg=optimizer_cfg,
            paramwise_cfg=paramwise_cfg))
    optimizer = optim_constructor(model)
    return optimizer

optimizer = build_optimizer(model, optimizer_cfg)
```

MMEngine 准备优化器

构建 MMEngine 执行器时，需要接受 `optim_wrapper` 参数，即优化器封装实例或者优化器封装配置，对于复杂配置的优化器封装，MMEngine 同样只需要配置 `optim_wrapper`。`optim_wrapper` 的详细介绍见执行器 *api* 文档。

OpenMMLab 系列算法库配置变更

```
optimizer = dict(
    constructor='CustomConstructor',
    type='AdamW',  # 优化器配置为一级字段
    lr=0.0001,  # 优化器配置为一级字段
    betas=(0.9, 0.999),  # 优化器配置为一级字段
    weight_decay=0.05,  # 优化器配置为一级字段
    paramwise_cfg={  # constructor 的参数
        'decay_rate': 0.95,
        'decay_type': 'layer_wise',
        'num_layers': 6
    })
# MMEngine 还需要配置 `optim_config`
# 来构建优化器钩子，而 MMEngine 不需要
optimizer_config = dict(grad_clip=None)
```

```
optim_wrapper = dict(
    constructor='CustomConstructor',
    type='OptimWrapper',  # 指定优化器封装类型
    optimizer=dict(  # 将优化器配置集中在 optimizer 内
        type='AdamW',
        lr=0.0001,
        betas=(0.9, 0.999),
        weight_decay=0.05)
    paramwise_cfg={
        'decay_rate': 0.95,
        'decay_type': 'layer_wise',
        'num_layers': 6
    })
```

```
对于检测、分类一类的上层任务（High level）MMCV 需要配置 `optim_config` 来构建优化器钩子，而␣
↪MMEngine 不需要。
```

本教程使用的 optim_wrapper 如下：

```
from torch.optim import SGD

optimizer = SGD(model.parameters(), lr=0.1, momentum=0.9)
optim_wrapper = dict(optimizer=optimizer)
```

31.3.7 准备训练钩子

MMCV 准备训练钩子：

MMCV 常用训练钩子的配置如下：

```
# learning rate scheduler config
lr_config = dict(policy='step', step=[2, 3])
# configuration of optimizer
optimizer_config = dict(grad_clip=None)
# configuration of saving checkpoints periodically
checkpoint_config = dict(interval=1)
# save log periodically and multiple hooks can be used simultaneously
log_config = dict(interval=100, hooks=[dict(type='TextLoggerHook')])
# register hooks to runner and those hooks will be invoked automatically
runner.register_training_hooks(
    lr_config=lr_config,
    optimizer_config=optimizer_config,
    checkpoint_config=checkpoint_config,
    log_config=log_config)
```

其中：

- `lr_config` 用于配置 `LrUpdaterHook`
- `optimizer_config` 用于配置 `OptimizerHook`
- `checkpoint_config` 用于配置 `CheckPointHook`
- `log_config` 用于配置 `LoggerHook`

除了上面提到的 4 个 Hook，MMCV 执行器自带 `IterTimerHook`。MMCV 需要先实例化执行器，再注册训练钩子，而 `MMEngine` 则在实例化阶段配置钩子。

MMEngine 准备训练钩子

MMEngine 执行器将 MMCV 常用的训练钩子配置成默认钩子：

- *RuntimeInfoHook*
- *IterTimerHook*
- *DistSamplerSeedHook*
- LoggerHook
- *CheckpointHook*
- *ParamSchedulerHook*

对比上例中 MMCV 配置的训练钩子：

- `LrUpdaterHook` 对应 MMEngine 中的 `ParamSchedulerHook`，二者对应关系详见迁移 `scheduler` 文档
- MMEngine 在模型的 train_step 时更新参数，因此不需要配置优化器钩子（`OptimizerHook`）
- MMEngine 自带 `CheckPointHook`，可以使用默认配置
- MMEngine 自带 `LoggerHook`，可以使用默认配置

因此我们只需要配置执行器优化器参数调整策略（*param_scheduler*），就能达到和配置 `lr_config` 一样的效果。MMEngine 也支持注册自定义钩子，具体教程详见执行器教程 和迁移 `hook` 文档。

```
# MMCV 零散的配置训练钩子
# 配置 LrUpdaterHook，相当于 MMEngine 的参数调度器
lr_config = dict(
    policy='step',
    warmup='linear',
    warmup_iters=500,
    warmup_ratio=0.001,
    step=[8, 11])

# 配置 OptimizerHook，MMEngine 不需要
optimizer_config = dict(grad_clip=None)

# 配置 LoggerHook
log_config = dict(  # LoggerHook
    interval=50,
    hooks=[
        dict(type='TextLoggerHook'),
        # dict(type='TensorboardLoggerHook')
    ])

# 配置 CheckPointHook
checkpoint_config = dict(interval=1)  # CheckPointHook
```

```
# 配置参数调度器
param_scheduler = [
    dict(
        type='LinearLR', start_factor=0.001, by_epoch=False, begin=0, end=500),
    dict(
        type='MultiStepLR',
        begin=0,
        end=12,
        by_epoch=True,
        milestones=[8, 11],
        gamma=0.1)
```

（下页继续）

(续上页)

```
]

# MMEngine 集中配置默认钩子
default_hooks = dict(
    timer=dict(type='IterTimerHook'),
    logger=dict(type='LoggerHook', interval=50),
    param_scheduler=dict(type='ParamSchedulerHook'),
    checkpoint=dict(type='CheckpointHook', interval=1),
    sampler_seed=dict(type='DistSamplerSeedHook'),
    visualization=dict(type='DetVisualizationHook'))
```

注解： MMEngine 移除了 `OptimizerHook`，优化步骤在 model 中执行。

本教程使用的 param_scheduler 如下：

```
from math import gamma

param_scheduler = dict(type='MultiStepLR', milestones=[2, 3], gamma=0.1)
```

31.3.8 准备验证模块

MMCV 借助 `EvalHook` 实现验证流程，受限于篇幅，这里不做进一步展开。MMEngine 通过验证循环控制器（ValLoop）和评测器（Evaluator）实现执行流程，如果我们想基于自定义的评价指标完成验证流程，则需要定义一个 `Metric`，并将其注册至 `METRICS` 注册器：

```
import torch
from mmengine.evaluator import BaseMetric
from mmengine.registry import METRICS

@METRICS.register_module(force=True)
class ToyAccuracyMetric(BaseMetric):

    def process(self, label, pred) -> None:
        self.results.append((label[1], pred, len(label[1])))

    def compute_metrics(self, results: list) -> dict:
        num_sample = 0
        acc = 0
        for label, pred, batch_size in results:
            acc += (label == torch.stack(pred)).sum()
```

(下页继续)

(续上页)

```
            num_sample += batch_size
        return dict(Accuracy=acc / num_sample)
```

实现自定义 Metric 后，我们还需在执行器的构造参数中配置评测器和验证循环控制器，本教程中示例配置如下：

```
val_evaluator = dict(type='ToyAccuracyMetric')
val_cfg = dict(type='ValLoop')
```

```
eval_cfg = cfg.get('evaluation', {})
eval_cfg['by_epoch'] = cfg.runner['type'] != 'IterBasedRunner'
eval_hook = DistEvalHook if distributed else EvalHook  # 配置 EvalHook
runner.register_hook(
    eval_hook(val_dataloader, **eval_cfg), priority='LOW')  # 注册 EvalHook
```

```
val_dataloader = val_dataloader  # 配置验证数据
val_evaluator = dict(type='ToyAccuracyMetric')  # 配置评测器
val_cfg = dict(type='ValLoop')  # 配置验证循环控制器
```

31.3.9 构建执行器

MMCV 构建执行器

```
runner = EpochBasedRunner(
    model=model,
    optimizer=optimizer,
    work_dir=work_dir,
    logger=logger,
    max_epochs=4
)
```

MMEngine 构建执行器

MMEngine 执行器的作用域比 MMCV 更广，将设置随机种子、启动分布式训练等流程参数化。除了前几节提到的参数，上例中出现的 EpochBasedRunner，max_epochs，val_interval 现在由 train_cfg 配置：

- by_epoch: True 时相当于 MMCV 的 EpochBasedRunner，False 时相当于 IterBasedRunner。
- max_epoch/max_iters: 同 MMCV 执行器的配置
- val_interval: 同 EvalHook 的 interval 参数

train_cfg 实际上是训练循环控制器的构造参数，当 by_epoch=True 时，使用 EpochBasedTrainLoop。

```
from mmengine.runner import Runner

runner = Runner(
    model=model,  # 待优化的模型
    work_dir='./work_dir',  # 配置工作目录
    randomness=randomness,  # 配置随机种子
    env_cfg=env_cfg,  # 配置环境变量
    launcher='none',  # 分布式训练启动方式
    optim_wrapper=optim_wrapper,  # 配置优化器
    param_scheduler=param_scheduler,  # 配置学习率调度器
    train_dataloader=train_dataloader,  # 配置训练数据
    train_cfg=dict(by_epoch=True, max_epochs=4, val_interval=1),  # 配置训练循环控制器
    val_dataloader=val_dataloader,  # 配置验证数据
    val_evaluator=val_evaluator,  # 配置评测器
    val_cfg=val_cfg)  # 配置验证循环控制器
```

31.3.10 执行器加载检查点

MMCV 加载检查点:

在训练之前执行加载权重、恢复训练的流程。

```
if cfg.resume_from:
    runner.resume(cfg.resume_from)
elif cfg.load_from:
    runner.load_checkpoint(cfg.load_from)
```

MMEngine 加载检查点

```
runner = Runner(
    ...
    load_from='/path/to/checkpoint',
    resume=True
)
```

```
load_from = 'path/to/ckpt'
```

```
load_from = 'path/to/ckpt'
resume = False
```

```
resume_from = 'path/to/ckpt'
```

```
load_from = 'path/to/ckpt'
resume = True
```

31.3.11 执行器训练流程

MMCV 执行器训练流程：

在训练之前执行加载权重、恢复训练的流程。然后再执行 `runner.run`，并传入训练数据。

```
if cfg.resume_from:
    runner.resume(cfg.resume_from)
elif cfg.load_from:
    runner.load_checkpoint(cfg.load_from)
runner.run(data_loaders, cfg.workflow)
```

MMEngine 执行器训练流程

在执行器构建时配置加载权重、恢复训练参数

由于 MMEngine 的执行器在构造阶段就传入了训练数据，因此在调用 runner.train() 无需传入参数。

```
runner.train()
```

31.3.12 执行器测试流程

MMCV 的执行器没有测试功能，因此需要自行实现测试脚本。MMEngine 的执行器只需要在构建时配置 `test_dataloader`、`test_cfg` 和 `test_evaluator`，然后再调用 `runner.test()` 执行测试流程。

`work_dir` 和训练时一致，无需手动加载 checkpoint:

```
runner = Runner(
    model=model,
    work_dir='./work_dir',
    randomness=randomness,
    env_cfg=env_cfg,
    launcher='none',  # 不开启分布式训练
    optim_wrapper=optim_wrapper,
    train_dataloader=train_dataloader,
    train_cfg=dict(by_epoch=True, max_epochs=5, val_interval=1),
    val_dataloader=val_dataloader,
    val_evaluator=val_evaluator,
    val_cfg=val_cfg,
    test_dataloader=val_dataloader,  # 假设测试和验证使用相同的数据和评测器
    test_evaluator=val_evaluator,
```

(下页继续)

(续上页)

```
    test_cfg=dict(type='TestLoop'),
)
runner.test()
```

work_dir 和训练时不一致，需要额外指定 load_from:

```
runner = Runner(
    model=model,
    work_dir='./test_work_dir',
    load_from='./work_dir/epoch_5.pth',  # work_dir 不一致，指定 load_from，以加载指定的模型
    randomness=randomness,
    env_cfg=env_cfg,
    launcher='none',
    optim_wrapper=optim_wrapper,
    train_dataloader=train_dataloader,
    train_cfg=dict(by_epoch=True, max_epochs=5, val_interval=1),
    val_dataloader=val_dataloader,
    val_evaluator=val_evaluator,
    val_cfg=val_cfg,
    test_dataloader=val_dataloader,
    test_evaluator=val_evaluator,
    test_cfg=dict(type='TestLoop'),
)
runner.test()
```

31.4 迁移自定义执行流程

使用 MMCV 执行器时，我们可能会重载 `runner.train/runner.val` 或者 `runner.run_iter` 实现自定义的训练、测试流程。以重载 `runner.train` 为例，假设我们想对每个批次的图片训练两遍，我们可以这样重载 MMCV 的执行器：

```
class CustomRunner(EpochBasedRunner):
    def train(self, data_loader, **kwargs):
        self.model.train()
        self.mode = 'train'
        self.data_loader = data_loader
        self._max_iters = self._max_epochs * len(self.data_loader)
        self.call_hook('before_train_epoch')
        time.sleep(2)  # Prevent possible deadlock during epoch transition
        for i, data_batch in enumerate(self.data_loader):
            self.data_batch = data_batch
            self._inner_iter = i
```

(下页继续)

(续上页)

```
            for _ in range(2)
                self.call_hook('before_train_iter')
                self.run_iter(data_batch, train_mode=True, **kwargs)
                self.call_hook('after_train_iter')
            del self.data_batch
            self._iter += 1

        self.call_hook('after_train_epoch')
        self._epoch += 1
```

在 MMEngine 中，要实现上述功能，我们需要重载一个新的循环控制器

```
from mmengine.registry import LOOPS
from mmengine.runner import EpochBasedTrainLoop


@LOOPS.register_module()
class CustomEpochBasedTrainLoop(EpochBasedTrainLoop):
    def run_iter(self, idx, data_batch) -> None:
        for _ in range(2):
            super().run_iter(idx, data_batch)
```

在构建执行器时，指定 train_cfg 的 type 为 CustomEpochBasedTrainLoop。需要注意的是，by_epoch 和 type 不能同时配置，当配置 by_epoch 时，会推断训练循环控制器的类型为 EpochBasedTrainLoop。

```
runner = Runner(
    model=model,
    work_dir='./test_work_dir',
    randomness=randomness,
    env_cfg=env_cfg,
    launcher='none',
    optim_wrapper=dict(optimizer=dict(type='SGD', lr=0.001, momentum=0.9)),
    train_dataloader=train_dataloader,
    train_cfg=dict(
        type='CustomEpochBasedTrainLoop',
        max_epochs=5,
        val_interval=1),
    val_dataloader=val_dataloader,
    val_evaluator=val_evaluator,
    val_cfg=val_cfg,
    test_dataloader=val_dataloader,
    test_evaluator=val_evaluator,
    test_cfg=dict(type='TestLoop'),
)
```

(下页继续)

(续上页)

```
runner.train()
```

如果有更加复杂的执行器迁移需求，可以参考执行器教程 和执行器设计文档。

CHAPTER 32

迁移 MMCV 钩子到 MMEngine

32.1 简介

由于架构设计的更新和用户需求的不断增加，MMCV 的钩子（Hook）点位已经满足不了需求，因此在 MMEngine 中对钩子点位进行了重新设计以及对钩子的功能做了调整。在开始迁移前，阅读钩子的设计会很有帮助。

本文对比 MMCV v1.6.0 和 MMEngine v0.5.0 的钩子在功能、点位、用法和实现上的差异。

32.2 功能差异

32.3 点位差异

32.4 用法差异

在 MMCV 中，将钩子注册到执行器（Runner），需调用执行器的 `register_training_hooks` 方法往执行器注册钩子，而在 MMEngine 中，可以通过参数传递给执行器的初始化方法进行注册。

- MMCV

```
model = ResNet18()
optimizer = optim.SGD(model.parameters(), lr=0.001, momentum=0.9)
```

(下页继续)

(续上页)

```
lr_config = dict(policy='step', step=[2, 3])
optimizer_config = dict(grad_clip=None)
checkpoint_config = dict(interval=5)
log_config = dict(interval=100, hooks=[dict(type='TextLoggerHook')])
custom_hooks = [dict(type='NumClassCheckHook')]
runner = EpochBasedRunner(
    model=model,
    optimizer=optimizer,
    work_dir='./work_dir',
    max_epochs=3,
    xxx,
)
runner.register_training_hooks(
    lr_config=lr_config,
    optimizer_config=optimizer_config,
    checkpoint_config=checkpoint_config,
    log_config=log_config,
    custom_hooks_config=custom_hooks,
)
runner.run([trainloader], [('train', 1)])
```

- MMEngine

```
model=ResNet18()
optim_wrapper=dict(
    type='OptimizerWrapper',
    optimizer=dict(type='SGD', lr=0.001, momentum=0.9))
param_scheduler = dict(type='MultiStepLR', milestones=[2, 3]),
default_hooks = dict(
    logger=dict(type='LoggerHook'),
    param_scheduler=dict(type='ParamSchedulerHook'),
    checkpoint=dict(type='CheckpointHook', interval=5),
)
custom_hooks = [dict(type='NumClassCheckHook')]
runner = Runner(
    model=model,
    work_dir='./work_dir',
    optim_wrapper=optim_wrapper,
    param_scheduler=param_scheduler,
    train_cfg=dict(by_epoch=True, max_epochs=3),
    default_hooks=default_hooks,
    custom_hooks=custom_hooks,
    xxx,
)
```

(下页继续)

(续上页)

```
runner.train()
```

MMEngine 钩子的更多用法请参考钩子的用法。

32.5 实现差异

以 CheckpointHook 为例，MMEngine 的 CheckpointHook 相比 MMCV 的 CheckpointHook（新增保存最优权重的功能，在 MMCV 中，保存最优权重的功能由 EvalHook 提供），因此，它需要实现 after_val_epoch 点位。

- MMCV

```
class CheckpointHook(Hook):
    def before_run(self, runner):
        """ 初始化 out_dir 和 file_client 属性"""

    def after_train_epoch(self, runner):
        """ 同步 buffer 和保存权重，用于以 epoch 为单位训练的任务"""

    def after_train_iter(self, runner):
        """ 同步 buffer 和保存权重，用于以 iteration 为单位训练的任务"""
```

- MMEngine

```
class CheckpointHook(Hook):
    def before_run(self, runner):
        """ 初始化 out_dir 和 file_client 属性"""

    def after_train_epoch(self, runner):
        """ 同步 buffer 和保存权重，用于以 epoch 为单位训练的任务"""

    def after_train_iter(self, runner, batch_idx, data_batch, outputs):
        """ 同步 buffer 和保存权重，用于以 iteration 为单位训练的任务"""

    def after_val_epoch(self, runner, metrics):
        """ 根据 metrics 保存最优权重"""
```

CHAPTER 33

迁移 MMCV 模型到 MMEngine

33.1 简介

MMCV 早期支持的计算机视觉任务，例如目标检测、物体识别等，都采用了一种典型的模型参数优化流程，可以被归纳为以下四个步骤：

1. 计算损失
2. 计算梯度
3. 更新参数
4. 梯度清零

上述流程的一大特点就是调用位置统一（在训练迭代后调用）、执行步骤统一（依次执行步骤 1->2->3->4），非常契合钩子（*Hook*）的设计原则，因此这类任务通常会使用 `Hook` 来优化模型。MMCV 为此实现了一系列的 `Hook`，例如 `OptimizerHook`（单精度训练）、`Fp16OptimizerHook`（混合精度训练）和 `GradientCumulativeFp16OptimizerHook`（混合精度训练 + 梯度累加），为这类任务提供各种优化策略。

一些例如生成对抗网络（GAN），自监督（Self-supervision）等领域的算法一般有更加灵活的训练流程，这类流程并不满足调用位置统一、执行步骤统一的原则，难以使用 `Hook` 对参数进行优化。为了支持训练这类任务，MMCV 的执行器会在调用 `model.train_step` 时，额外传入 `optimizer` 参数，让模型在 `train_step` 里实现自定义的优化流程。这样虽然可以支持训练这类任务，但也会导致无法使用各种 `OptimizerHook`，需要算法在 `train_step` 中实现混合精度训练、梯度累加等训练策略。

为了统一深度学习任务的参数优化流程，MMEngine 设计了优化器封装，集成了混合精度训练、梯度累加等训练策略，各类深度学习任务一律在 `model.train_step` 里执行参数优化流程。

33.2 优化流程的迁移

33.2.1 统一的参数更新流程

考虑到目标检测、物体识别一类的深度学习任务参数优化的流程基本一致，我们可以通过继承模型基类来完成迁移。

基于 MMCV 执行器的模型

在介绍如何迁移模型之前，我们先来看一个基于 MMCV 执行器训练模型的最简示例：

```
import torch
import torch.nn as nn
from torch.optim import SGD
from torch.utils.data import DataLoader

from mmcv.runner import Runner
from mmcv.utils.logging import get_logger


train_dataset = [(torch.ones(1, 1), torch.ones(1, 1))] * 50
train_dataloader = DataLoader(train_dataset, batch_size=2)


class MMCVToyModel(nn.Module):
    def __init__(self) -> None:
        super().__init__()
        self.linear = nn.Linear(1, 1)

    def forward(self, img, label, return_loss=False):
        feat = self.linear(img)
        loss1 = (feat - label).pow(2)
        loss2 = (feat - label).abs()
        loss = (loss1 + loss2).sum()
        return dict(loss=loss,
                    num_samples=len(img),
                    log_vars=dict(
                        loss1=loss1.sum().item(),
                        loss2=loss2.sum().item()))

    def train_step(self, data, optimizer=None):
        return self(*data, return_loss=True)

    def val_step(self, data, optimizer=None):
        return self(*data, return_loss=False)
```

(下页继续)

(续上页)

```
model = MMCVToyModel()
optimizer = SGD(model.parameters(), lr=0.01)
logger = get_logger('demo')

lr_config = dict(policy='step', step=[2, 3])
optimizer_config = dict(grad_clip=None)
log_config = dict(interval=10, hooks=[dict(type='TextLoggerHook')])


runner = Runner(
    model=model,
    work_dir='tmp_dir',
    optimizer=optimizer,
    logger=logger,
    max_epochs=5)

runner.register_training_hooks(
    lr_config=lr_config,
    optimizer_config=optimizer_config,
    log_config=log_config)
runner.run([train_dataloader], [('train', 1)])
```

基于 MMCV 执行器训练模型时，我们必须实现 `train_step` 接口，并返回一个字典，字典需要包含以下三个字段：

- loss：传给 `OptimizerHook` 计算梯度
- num_samples：传给 `LogBuffer`，用于计算平滑后的损失
- log_vars：传给 `LogBuffer` 用于计算平滑后的损失

基于 MMEngine 执行器的模型

基于 MMEngine 的执行器，实现同样逻辑的代码：

```
import torch
import torch.nn as nn
from torch.utils.data import DataLoader

from mmengine.runner import Runner
from mmengine.model import BaseModel

train_dataset = [(torch.ones(1, 1), torch.ones(1, 1))] * 50
train_dataloader = DataLoader(train_dataset, batch_size=2)
```

(下页继续)

(续上页)

```
class MMEngineToyModel(BaseModel):

    def __init__(self) -> None:
        super().__init__()
        self.linear = nn.Linear(1, 1)

    def forward(self, img, label, mode):
        feat = self.linear(img)
        # 被 `train_step` 调用，返回用于更新参数的损失字典
        if mode == 'loss':
            loss1 = (feat - label).pow(2)
            loss2 = (feat - label).abs()
            return dict(loss1=loss1, loss2=loss2)
        # 被 `val_step` 调用，返回传给 `evaluator` 的预测结果
        elif mode == 'predict':
            return [_feat for _feat in feat]
        # tensor 模式，功能详见模型教程文档： tutorials/model.md
        else:
            pass


runner = Runner(
    model=MMEngineToyModel(),
    work_dir='tmp_dir',
    train_dataloader=train_dataloader,
    train_cfg=dict(by_epoch=True, max_epochs=5),
    optim_wrapper=dict(optimizer=dict(type='SGD', lr=0.01)))
runner.train()
```

MMEngine 实现了模型基类，模型基类在 `train_step` 里实现了 `OptimizerHook` 的优化流程。因此上例中，我们无需实现 `train_step`，运行时直接调用基类的 `train_step`。

```
class MMCVToyModel(nn.Module):

    def __init__(self) -> None:
        super().__init__()
        self.linear = nn.Linear(1, 1)

    def forward(self, img, label, return_loss=False):
        feat = self.linear(img)
        loss1 = (feat - label).pow(2)
```

(下页继续)

(续上页)

```
        loss2 = (feat - label).abs()
        loss = (loss1 + loss2).sum()
        return dict(loss=loss,
                    num_samples=len(img),
                    log_vars=dict(
                        loss1=loss1.sum().item(),
                        loss2=loss2.sum().item()))

    def train_step(self, data, optimizer=None):
        return self(*data, return_loss=True)

    def val_step(self, data, optimizer=None):
        return self(*data, return_loss=False)
```

```
class MMEngineToyModel(BaseModel):

    def __init__(self) -> None:
        super().__init__()
        self.linear = nn.Linear(1, 1)

    def forward(self, img, label, mode):
        if mode == 'loss':
            feat = self.linear(img)
            loss1 = (feat - label).pow(2)
            loss2 = (feat - label).abs()
            return dict(loss1=loss1, loss2=loss2)
        elif mode == 'predict':
            return [_feat for _feat in feat]
        else:
            # tensor 模式，功能详见模型教程文档： tutorials/model.md
            pass

    # 模型基类 `train_step` 等效代码
    # def train_step(self, data, optim_wrapper):
    #     data = self.data_preprocessor(data)
    #     loss_dict = self(*data, mode='loss')
    #     loss_dict['loss1'] = loss_dict['loss1'].sum()
    #     loss_dict['loss2'] = loss_dict['loss2'].sum()
    #     loss = (loss_dict['loss1'] + loss_dict['loss2']).sum()
    # 调用优化器封装更新模型参数
    #     optim_wrapper.update_params(loss)
    #     return loss_dict
```

关于等效代码中的数据处理器（*data_preprocessor*）和优化器封装（*optim_wrapper*）的说明，详见模型教程

和优化器封装教程。

模型具体差异如下：

- MMCVToyModel 继承自 nn.Module，而 MMEngineToyModel 继承自 BaseModel
- MMCVToyModel 必须实现 train_step，且必须返回损失字典，损失字典包含 loss 和 log_vars 和 num_samples 字段。MMEngineToyModel 继承自 BaseModel，只需要实现 forward 接口，并返回损失字典，损失字典的每一个值必须是可微的张量
- MMCVToyModel 和 MMEngineModel 的 forward 的接口需要匹配 train_step 中的调用方式，由于 MMEngineToyModel 直接调用基类的 train_step 方法，因此 forward 需要接受参数 mode，具体规则详见模型教程文档
- MMEngineModel 如果没有继承 BaseModel，必须实现 train_step 方法。

33.2.2 自定义的参数更新流程

以训练生成对抗网络为例，生成器和判别器的优化需要交替进行，且优化流程可能会随着迭代次数的增多发生变化，因此很难使用 OptimizerHook 来满足这种需求。在基于 MMCV 训练生成对抗网络时，通常会在模型的 train_step 接口中传入 optimizer，然后在 train_step 里实现自定义的参数更新逻辑。这种训练流程和 MMEngine 非常相似，只不过 MMEngine 在 train_step 接口中传入优化器封装，能够更加简单地优化模型。

参考训练生成对抗网络，如果用 MMCV 进行训练，GAN 的模型优化接口如下：

```
    def train_discriminator(
            self, inputs, data_sample,
            optimizer):
        real_imgs = inputs['inputs']
        z = torch.randn((real_imgs.shape[0], self.noise_size))
        with torch.no_grad():
            fake_imgs = self.generator(z)

        disc_pred_fake = self.discriminator(fake_imgs)
        disc_pred_real = self.discriminator(real_imgs)

        parsed_losses, log_vars = self.disc_loss(disc_pred_fake,
                                                 disc_pred_real)
        parsed_losses.backward()
        optimizer.step()
        optimizer.zero_grad()
        return log_vars

    def train_generator(self, inputs, data_sample, optimizer_wrapper):
        z = torch.randn(inputs['inputs'].shape[0], self.noise_size)
```

(下页继续)

(续上页)

```
        fake_imgs = self.generator(z)

        disc_pred_fake = self.discriminator(fake_imgs)
        parsed_loss, log_vars = self.gen_loss(disc_pred_fake)

        parsed_losses.backward()
        optimizer.step()
        optimizer.zero_grad()
        return log_vars
```

对比 MMEngine 的实现：

```
    def train_discriminator(
            self, inputs, data_sample,
            optimizer):
        real_imgs = inputs['inputs']
        z = torch.randn((real_imgs.shape[0], self.noise_size))
        with torch.no_grad():
            fake_imgs = self.generator(z)

        disc_pred_fake = self.discriminator(fake_imgs)
        disc_pred_real = self.discriminator(real_imgs)

        parsed_losses, log_vars = self.disc_loss(disc_pred_fake,
                                                 disc_pred_real)
        parsed_losses.backward()
        optimizer.step()
        optimizer.zero_grad()
        return log_vars

    def train_generator(self, inputs, data_sample, optimizer_wrapper):
        z = torch.randn(inputs['inputs'].shape[0], self.noise_size)

        fake_imgs = self.generator(z)

        disc_pred_fake = self.discriminator(fake_imgs)
        parsed_loss, log_vars = self.gen_loss(disc_pred_fake)

        parsed_losses.backward()
        optimizer.step()
        optimizer.zero_grad()
        return log_vars
```

```
    def train_discriminator(
            self, inputs, data_sample,
            optimizer_wrapper):
        real_imgs = inputs['inputs']
        z = torch.randn((real_imgs.shape[0], self.noise_size))
        with torch.no_grad():
            fake_imgs = self.generator(z)

        disc_pred_fake = self.discriminator(fake_imgs)
        disc_pred_real = self.discriminator(real_imgs)

        parsed_losses, log_vars = self.disc_loss(disc_pred_fake,
                                                 disc_pred_real)
        optimizer_wrapper.update_params(parsed_losses)
        return log_vars



    def train_generator(self, inputs, data_sample, optimizer_wrapper):
        z = torch.randn(inputs['inputs'].shape[0], self.noise_size)

        fake_imgs = self.generator(z)

        disc_pred_fake = self.discriminator(fake_imgs)
        parsed_loss, log_vars = self.gen_loss(disc_pred_fake)

        optimizer_wrapper.update_params(parsed_loss)
        return log_vars
```

二者的区别主要在于优化器的使用方式。此外，`train_step` 接口返回值的差异和上一节提到的一致。

33.3 验证/测试流程的迁移

基于 MMCV 执行器实现的模型通常不需要为验证、测试流程提供独立的 `val_step`、`test_step`（测试流程由 `EvalHook` 实现，这里不做展开）。基于 MMEngine 执行器实现的模型则有所不同，*ValLoop*、*TestLoop* 会分别调用模型的 `val_step` 和 `test_step` 接口，输出会进一步传给*Evaluator.process*。因此模型的 `val_step` 和 `test_step` 接口输出需要和 `Evaluator.process` 的入参（第一个参数）对齐，即返回列表（推荐，也可以是其他可迭代类型）类型的结果。列表中的每一个元素代表一个批次（batch）数据中每个样本的预测结果。模型的 `test_step` 和 `val_step` 会调 `forward` 接口（详见模型教程文档），因此在上一节的模型示例中，模型 `forward` 的 `predict` 模式会将 `feat` 切片后，以列表的形式返回预测结果。

```
class MMEngineToyModel(BaseModel):


    ...
    def forward(self, img, label, mode):
        if mode == 'loss':
            ...
        elif mode == 'predict':
            # 把一个 batch 的预测结果切片成列表，每个元素代表一个样本的预测结果
            return [_feat for _feat in feat]
        else:
            ...
            # tensor 模式，功能详见模型教程文档： tutorials/model.md
```

33.4 迁移分布式训练

MMCV 需要在执行器构建之前，使用 `MMDistributedDataParallel` 对模型进行分布式封装。MMEngine 实现了*MMDistributedDataParallel* 和*MMSeparateDistributedDataParallel* 两种分布式模型封装，供不同类型的任务选择。执行器会在构建时对模型进行分布式封装。

1. **常用训练流程**

   对于简介中提到的常用优化流程的训练任务，即一次参数更新可以被拆解成梯度计算、参数优化、梯度清零的任务，使用 Runner 默认的 `MMDistributedDataParallel` 即可满足需求，无需为 runner 其他额外参数。

   ```
   model = MMDistributedDataParallel(
       model,
       device_ids=[int(os.environ['LOCAL_RANK'])],
       broadcast_buffers=False,
       find_unused_parameters=find_unused_parameters)
   ...
   runner = Runner(model=model, ...)
   ```

   ```
   runner = Runner(
       model=model,
       launcher='pytorch', # 开启分布式训练
       ..., # 其他参数
   )
   ```

2. **分模块优化的学习任务**

同样以训练生成对抗网络为例，生成对抗网络有两个需要分别优化的子模块，即生成器和判别器。因此需要使用 MMSeparateDistributedDataParallel 对模型进行封装。我们需要在构建执行器时指定：

```
cfg = dict(model_wrapper_cfg=dict(type='MMSeparateDistributedDataParallel'))
runner = Runner(
    model=model,
    ..., # 其他配置
    launcher='pytorch',
    cfg=cfg # 模型封装配置
)
```

即可进行分布式训练。

3. **单模块优化、自定义流程的深度学习任务**

有时候我们需要用自定义的优化流程来优化单个模块，这时候我们就不能复用模型基类的 train_step，而需要重新实现，例如我们想用同一批图片对模型优化两次，第一次开启批数据增强，第二次关闭：

```
class CustomModel(BaseModel):

    def train_step(self, data, optim_wrapper):
        data = self.data_preprocessor(data, training=True)  # 开启批数据增强
        loss = self(data, mode='loss')
        optim_wrapper.update_params(loss)
        data = self.data_preprocessor(data, training=False)  # 关闭批数据增强
        loss = self(data, mode='loss')
        optim_wrapper.update_params(loss)
```

要想启用分布式训练，我们就需要重载 MMSeparateDistributedDataParallel，并在 train_step 中实现和 CustomModel.train_step 相同的流程（test_step、val_step 同理）。

```
class CustomDistributedDataParallel(MMSeparateDistributedDataParallel):

    def train_step(self, data, optim_wrapper):
        data = self.data_preprocessor(data, training=True)  # 开启批数据增强
        loss = self(data, mode='loss')
        optim_wrapper.update_params(loss)
        data = self.data_preprocessor(data, training=False)  # 关闭批数据增强
        loss = self(data, mode='loss')
        optim_wrapper.update_params(loss)
```

最后在构建 runner 时指定：

```
# 指定封装类型为 `CustomDistributedDataParallel`，并基于默认参数封装模型。
cfg = dict(model_wrapper_cfg=dict(type='CustomDistributedDataParallel'))
runner = Runner(
    model=model,
    ..., # 其他配置
    launcher='pytorch',
    cfg=cfg # 模型封装配置
)
```

CHAPTER 34

迁移 MMCV 参数调度器到 MMEngine

MMCV 1.x 版本使用 LrUpdaterHook 和 MomentumUpdaterHook 来调整学习率和动量。但随着深度学习算法训练方式的不断发展，使用 Hook 修改学习率已经难以满足更加丰富的自定义需求，因此 MMEngine 提供了参数调度器（ParamScheduler）。一方面，参数调度器的接口与 PyTroch 的学习率调度器（LRScheduler）对齐，另一方面，参数调度器提供了更丰富的功能，详细请参考参数调度器使用指南。

34.1 学习率调度器（LrUpdater）迁移

MMEngine 中使用 LRScheduler 替代 LrUpdaterHook，配置文件中的字段从原本的 `lr_config` 修改为 `param_scheduler`。MMCV 中的学习率配置与 MMEngine 中的参数调度器配置对应关系如下：

34.1.1 学习率预热（Warmup）迁移

由于 MMEngine 中的学习率调度器在实现时增加了 begin 和 end 参数，指定了调度器的生效区间，所以可以通过调度器组合的方式实现学习率预热。MMCV 中有 3 种学习率预热方式，分别是‘constant’，‘linear’，‘exp’，在 MMEngine 中对应的配置应修改为:

常数预热（‘constant’）

```
lr_config = dict(
    warmup='constant',
    warmup_ratio=0.1,
    warmup_iters=500,
    warmup_by_epoch=False
)
```

```
param_scheduler = [
    dict(type='ConstantLR',
         factor=0.1,
         begin=0,
         end=500,
         by_epoch=False),
    dict(...) # 主学习率调度器配置
]
```

线性预热（‘linear’）

```
lr_config = dict(
    warmup='linear',
    warmup_ratio=0.1,
    warmup_iters=500,
    warmup_by_epoch=False
)
```

```
param_scheduler = [
    dict(type='LinearLR',
         start_factor=0.1,
         begin=0,
         end=500,
         by_epoch=False),
    dict(...) # 主学习率调度器配置
]
```

指数预热（‘exp’）

```
lr_config = dict(
    warmup='exp',
    warmup_ratio=0.1,
    warmup_iters=500,
    warmup_by_epoch=False
)
```

```
param_scheduler = [
    dict(type='ExponentialLR',
         gamma=0.1,
         begin=0,
         end=500,
         by_epoch=False),
    dict(...) # 主学习率调度器配置
]
```

34.1.2 ‘fixed’学习率（FixedLrUpdaterHook）迁移

```
lr_config = dict(policy='fixed')
```

```
param_scheduler = [
    dict(type='ConstantLR', factor=1)
]
```

34.1.3 ‘step’学习率（StepLrUpdaterHook）迁移

```
lr_config = dict(
    policy='step',
    step=[8, 11],
    gamma=0.1,
    by_epoch=True
)
```

```
param_scheduler = [
    dict(type='MultiStepLR',
         milestone=[8, 11],
         gamma=0.1,
         by_epoch=True)
]
```

34.1.4 ‘poly’学习率（PolyLrUpdaterHook）迁移

```
lr_config = dict(
    policy='poly',
    power=0.7,
    min_lr=0.001,
    by_epoch=True
)
```

```
param_scheduler = [
    dict(type='PolyLR',
         power=0.7,
         eta_min=0.001,
         begin=0,
         end=num_epochs,
         by_epoch=True)
]
```

34.1.5 ‘exp’学习率（ExpLrUpdaterHook）迁移

```
lr_config = dict(
    policy='exp',
    power=0.5,
    by_epoch=True
)
```

```
param_scheduler = [
    dict(type='ExponentialLR',
         gamma=0.5,
         begin=0,
         end=num_epochs,
         by_epoch=True)
]
```

34.1.6 ‘CosineAnnealing’学习率（CosineAnnealingLrUpdaterHook）迁移

```
lr_config = dict(
    policy='CosineAnnealing',
    min_lr=0.5,
    by_epoch=True
)
```

```
param_scheduler = [
    dict(type='CosineAnnealingLR',
         eta_min=0.5,
         T_max=num_epochs,
         begin=0,
         end=num_epochs,
         by_epoch=True)
]
```

34.1.7 ‘FlatCosineAnnealing’学习率（FlatCosineAnnealingLrUpdaterHook）迁移

像 FlatCosineAnnealing 这种由多个学习率策略拼接而成的学习率，原本需要重写 Hook 来实现，而在 MMEngine 中只需将两个参数调度器组合即可

```
lr_config = dict(
    policy='FlatCosineAnnealing',
    start_percent=0.5,
    min_lr=0.005,
    by_epoch=True
)
```

```
param_scheduler = [
    dict(type='ConstantLR', factor=1, begin=0, end=num_epochs * 0.75)
    dict(type='CosineAnnealingLR',
         eta_min=0.005,
         begin=num_epochs * 0.75,
         end=num_epochs,
         T_max=num_epochs * 0.25,
         by_epoch=True)
]
```

34.1.8 ‘CosineRestart’学习率（CosineRestartLrUpdaterHook）迁移

```
lr_config = dict(policy='CosineRestart',
                 periods=[5, 10, 15],
                 restart_weights=[1, 0.7, 0.3],
                 min_lr=0.001,
                 by_epoch=True)
```

```
param_scheduler = [
    dict(type='CosineRestartLR',
         periods=[5, 10, 15],
         restart_weights=[1, 0.7, 0.3],
         eta_min=0.001,
         by_epoch=True)
]
```

34.1.9 ‘OneCycle’学习率（OneCycleLrUpdaterHook）迁移

```
lr_config = dict(policy='OneCycle',
                 max_lr=0.02,
                 total_steps=90000,
                 pct_start=0.3,
                 anneal_strategy='cos',
                 div_factor=25,
                 final_div_factor=1e4,
                 three_phase=True,
                 by_epoch=False)
```

```
param_scheduler = [
    dict(type='OneCycleLR',
         eta_max=0.02,
         total_steps=90000,
         pct_start=0.3,
         anneal_strategy='cos',
         div_factor=25,
         final_div_factor=1e4,
         three_phase=True,
         by_epoch=False)
]
```

需要注意的是 `by_epoch` 参数 MMCV 默认是 `False`, MMEngine 默认是 `True`

34.1.10 ‘LinearAnnealing’学习率（LinearAnnealingLrUpdaterHook）迁移

```
lr_config = dict(
    policy='LinearAnnealing',
    min_lr_ratio=0.01,
    by_epoch=True
)
```

```
param_scheduler = [
    dict(type='LinearLR',
         start_factor=1,
         end_factor=0.01,
         begin=0,
         end=num_epochs,
         by_epoch=True)
]
```

34.2 动量调度器（MomentumUpdater）迁移

MMCV 使用 momentum_config 字段和 MomentumUpdateHook 调整动量。MMEngine 中动量同样由参数调度器控制。用户可以简单将学习率调度器后的 LR 修改为 Momentum，即可使用同样的策略来调整动量。动量调度器只需要和学习率调度器一样添加进 param_scheduler 列表中即可。举一个简单的例子：

```
lr_config = dict(...)
momentum_config = dict(
    policy='CosineAnnealing',
    min_momentum=0.1,
    by_epoch=True
)
```

```
param_scheduler = [
    # 学习率调度器配置
    dict(...),
    # 动量调度器配置
    dict(type='CosineAnnealingMomentum',
         eta_min=0.1,
         T_max=num_epochs,
         begin=0,
         end=num_epochs,
         by_epoch=True)
]
```

34.3 参数更新频率相关配置迁移

如果在使用 epoch-based 训练循环且配置文件中按 epoch 设置生效区间（begin，end）或周期（T_max）等变量的同时希望参数率按 iteration 更新，在 MMCV 中需要将 by_epoch 设置为 False。而在 MMEngine 中需要注意，配置中的 by_epoch 仍需设置为 True，通过在配置中添加 convert_to_iter_based=True 来构建按 iteration 更新的参数调度器，关于此配置详见参数调度器教程。以迁移 CosineAnnealing 为例：

```
lr_config = dict(
    policy='CosineAnnealing',
    min_lr=0.5,
    by_epoch=False
)
```

```
param_scheduler = [
    dict(
        type='CosineAnnealingLR',
        eta_min=0.5,
        T_max=num_epochs,
        by_epoch=True,  # 注意，by_epoch 需要设置为 True
        convert_to_iter_based=True  # 转换为按 iter 更新参数
    )
]
```

你可能还想阅读参数调度器的教程或者参数调度器的 API 文档。

CHAPTER 35

数据变换类的迁移

35.1 简介

在 TorchVision 的数据变换类接口约定中，数据变换类需要实现 `__call__` 方法，而在 OpenMMLab 1.0 的接口约定中，进一步要求 `__call__` 方法的输出应当是一个字典，在各种数据变换中对这个字典进行增删查改。在 OpenMMLab 2.0 中，为了提升后续的可扩展性，我们将原先的 `__call__` 方法迁移为 `transform` 方法，并要求数据变换类应当继承 `mmcv.transforms.BaseTransfrom`。具体如何实现一个数据变换类，可以参见文档。

由于在此次更新中，我们将部分共用的数据变换类统一迁移至 MMCV 中，因此本文的将会以 MMClassification v0.23.2、MMDetection v2.25.1 和 MMCV v2.0.0rc0 为例，对比这些数据变换类在新旧版本中功能、用法和实现上的差异。

35.2 功能差异

35.3 实现差异

以 `RandomFlip` 为例，MMCV 的 RandomFlip 相比旧版 MMDetection 的 RandomFlip，需要继承 `BaseTransfrom`，将功能实现放在 `transforms` 方法，并将生成随机结果的部分放在单独的方法中，用 `cache_randomness` 包装。有关随机方法的包装相关功能，参见相关文档。

- MMDetection (旧)

```
class RandomFlip:
    def __call__(self, results):
        """ 调用时进行随机翻转"""
        ...
        # 随机选择翻转方向
        cur_dir = np.random.choice(direction_list, p=flip_ratio_list)
        ...
        return results
```

- MMCV

```
class RandomFlip(BaseTransfrom):
    def transform(self, results):
        """ 调用时进行随机翻转"""
        ...
        cur_dir = self._random_direction()
        ...
        return results

    @cache_randomness
    def _random_direction(self):
        """ 随机选择翻转方向"""
        ...
        return np.random.choice(direction_list, p=flip_ratio_list)
```

CHAPTER 36

mmengine.registry

Registry	A registry to map strings to classes or functions.
DefaultScope	Scope of current task used to reset the current registry, which can be accessed globally.

36.1 Registry

class mmengine.registry.**Registry**(*name*, *build_func=None*, *parent=None*, *scope=None*)

A registry to map strings to classes or functions.

Registered object could be built from registry. Meanwhile, registered functions could be called from registry.

参数

- **name** (*str*) –Registry name.
- **build_func** (*callable, optional*) –A function to construct instance from Registry. *build_from_cfg()* is used if neither `parent` or `build_func` is specified. If `parent` is specified and `build_func` is not given, `build_func` will be inherited from `parent`. Defaults to None.
- **parent** (*Registry*, optional) –Parent registry. The class registered in children registry could be built from parent. Defaults to None.
- **scope** (*str, optional*) –The scope of registry. It is the key to search for children reg-

istry. If not specified, scope will be the name of the package where class is defined, e.g. mmdet, mmcls, mmseg. Defaults to None.

实际案例

```
>>> # define a registry
>>> MODELS = Registry('models')
>>> # registry the `ResNet` to `MODELS`
>>> @MODELS.register_module()
>>> class ResNet:
>>>     pass
>>> # build model from `MODELS`
>>> resnet = MODELS.build(dict(type='ResNet'))
>>> @MODELS.register_module()
>>> def resnet50():
>>>     pass
>>> resnet = MODELS.build(dict(type='resnet50'))
```

```
>>> # hierarchical registry
>>> DETECTORS = Registry('detectors', parent=MODELS, scope='det')
>>> @DETECTORS.register_module()
>>> class FasterRCNN:
>>>     pass
>>> fasterrcnn = DETECTORS.build(dict(type='FasterRCNN'))
```

More advanced usages can be found at https://mmengine.readthedocs.io/en/latest/tutorials/registry.html.

build(*cfg*, **args*, ***kwargs*)

Build an instance.

Build an instance by calling `build_func`.

参数 **cfg** (*dict*) –Config dict needs to be built.

返回 The constructed object.

返回类型 Any

实际案例

```
>>> from mmengine import Registry
>>> MODELS = Registry('models')
>>> @MODELS.register_module()
>>> class ResNet:
>>>     def __init__(self, depth, stages=4):
>>>         self.depth = depth
>>>         self.stages = stages
>>> cfg = dict(type='ResNet', depth=50)
>>> model = MODELS.build(cfg)
```

get(*key*)

Get the registry record.

The method will first parse `key` and check whether it contains a scope name. The logic to search for `key`:

- `key` does not contain a scope name, i.e., it is purely a module name like "ResNet" : *`get()`* will search for `ResNet` from the current registry to its parent or ancestors until finding it.
- `key` contains a scope name and it is equal to the scope of the current registry (e.g., "mmcls"), e.g., "mmcls.ResNet" : *`get()`* will only search for `ResNet` in the current registry.
- `key` contains a scope name and it is not equal to the scope of the current registry (e.g., "mmdet"), e.g., "mmcls.FCNet" : If the scope exists in its children, *`get()`* will get "FCNet" from them. If not, *`get()`* will first get the root registry and root registry call its own *`get()`* method.

参数 **key** (*str*) –Name of the registered item, e.g., the class name in string format.

返回 Return the corresponding class if `key` exists, otherwise return None.

返回类型 Type or None

实际案例

```
>>> # define a registry
>>> MODELS = Registry('models')
>>> # register `ResNet` to `MODELS`
>>> @MODELS.register_module()
>>> class ResNet:
>>>     pass
>>> resnet_cls = MODELS.get('ResNet')
```

```
>>> # hierarchical registry
>>> DETECTORS = Registry('detector', parent=MODELS, scope='det')
>>> # `ResNet` does not exist in `DETECTORS` but `get` method
```

(下页继续)

(续上页)

```
>>> # will try to search from its parenet or ancestors
>>> resnet_cls = DETECTORS.get('ResNet')
>>> CLASSIFIER = Registry('classifier', parent=MODELS, scope='cls')
>>> @CLASSIFIER.register_module()
>>> class MobileNet:
>>>     pass
>>> # `get` from its sibling registries
>>> mobilenet_cls = DETECTORS.get('cls.MobileNet')
```

static infer_scope()

Infer the scope of registry.

The name of the package where registry is defined will be returned.

返回 The inferred scope name.

返回类型 str

实际案例

```
>>> # in mmdet/models/backbone/resnet.py
>>> MODELS = Registry('models')
>>> @MODELS.register_module()
>>> class ResNet:
>>>     pass
>>> # The scope of ``ResNet`` will be ``mmdet``.
```

register_module(*name=None*, *force=False*, *module=None*)

Register a module.

A record will be added to `self._module_dict`, whose key is the class name or the specified name, and value is the class itself. It can be used as a decorator or a normal function.

参数

- **name** (*str or list of str, optional*) –The module name to be registered. If not specified, the class name will be used.
- **force** (*bool*) –Whether to override an existing class with the same name. Default to False.
- **module** (*type, optional*) –Module class or function to be registered. Defaults to None.

返回类型 Union[type, collections.abc.Callable]

实际案例

```
>>> backbones = Registry('backbone')
>>> # as a decorator
>>> @backbones.register_module()
>>> class ResNet:
>>>     pass
>>> backbones = Registry('backbone')
>>> @backbones.register_module(name='mnet')
>>> class MobileNet:
>>>     pass
```

```
>>> # as a normal function
>>> class ResNet:
>>>     pass
>>> backbones.register_module(module=ResNet)
```

static split_scope_key(*key*)

Split scope and key.

The first scope will be split from key.

返回 The former element is the first scope of the key, which can be `None`. The latter is the remaining key.

返回类型 tuple[str | None, str]

参数 **key** (*str*) –

实际案例

```
>>> Registry.split_scope_key('mmdet.ResNet')
'mmdet', 'ResNet'
>>> Registry.split_scope_key('ResNet')
None, 'ResNet'
```

switch_scope_and_registry(*scope*)

Temporarily switch default scope to the target scope, and get the corresponding registry.

If the registry of the corresponding scope exists, yield the registry, otherwise yield the current itself.

参数 **scope** (*str*) –The target scope.

返回类型 Generator

实际案例

```
>>> from mmengine.registry import Registry, DefaultScope, MODELS
>>> import time
>>> # External Registry
>>> MMDET_MODELS = Registry('mmdet_model', scope='mmdet',
>>>     parent=MODELS)
>>> MMCLS_MODELS = Registry('mmcls_model', scope='mmcls',
>>>     parent=MODELS)
>>> # Local Registry
>>> CUSTOM_MODELS = Registry('custom_model', scope='custom',
>>>     parent=MODELS)
>>>
>>> # Initiate DefaultScope
>>> DefaultScope.get_instance(f'scope_{time.time()}',
>>>     scope_name='custom')
>>> # Check default scope
>>> DefaultScope.get_current_instance().scope_name
custom
>>> # Switch to mmcls scope and get `MMCLS_MODELS` registry.
>>> with CUSTOM_MODELS.switch_scope_and_registry(scope='mmcls') as registry:
>>>     DefaultScope.get_current_instance().scope_name
mmcls
>>>     registry.scope
mmcls
>>> # Nested switch scope
>>> with CUSTOM_MODELS.switch_scope_and_registry(scope='mmdet') as mmdet_
↪registry:
>>>     DefaultScope.get_current_instance().scope_name
mmdet
>>>     mmdet_registry.scope
mmdet
>>>     with CUSTOM_MODELS.switch_scope_and_registry(scope='mmcls') as mmcls_
↪registry:
>>>         DefaultScope.get_current_instance().scope_name
mmcls
>>>         mmcls_registry.scope
mmcls
>>>
>>> # Check switch back to original scope.
>>> DefaultScope.get_current_instance().scope_name
custom
```

36.2 DefaultScope

class mmengine.registry.**DefaultScope**(*name*, *scope_name*)

Scope of current task used to reset the current registry, which can be accessed globally.

Consider the case of resetting the current `Resgitry` by"default_scope" in the internal module which cannot access runner directly, it is difficult to get the `default_scope` defined in `Runner`. However, if `Runner` created `DefaultScope` instance by given `default_scope`, the internal module can get `default_scope` by `DefaultScope.get_current_instance` everywhere.

参数

- **name** (*str*) –Name of default scope for global access.
- **scope_name** (*str*) –Scope of current task.

实际案例

```
>>> from mmengine.model import MODELS
>>> # Define default scope in runner.
>>> DefaultScope.get_instance('task', scope_name='mmdet')
>>> # Get default scope globally.
>>> scope_name = DefaultScope.get_instance('task').scope_name
```

classmethod get_current_instance()

Get latest created default scope.

Since default_scope is an optional argument for `Registry.build`. `get_current_instance` should return `None` if there is no `DefaultScope` created.

实际案例

```
>>> default_scope = DefaultScope.get_current_instance()
>>> # There is no `DefaultScope` created yet,
>>> # `get_current_instance` return `None`.
>>> default_scope = DefaultScope.get_instance(
>>>     'instance_name', scope_name='mmengine')
>>> default_scope.scope_name
mmengine
>>> default_scope = DefaultScope.get_current_instance()
>>> default_scope.scope_name
mmengine
```

返回 Return None If there has not been `DefaultScope` instance created yet, otherwise return the latest created DefaultScope instance.

返回类型 Optional[*DefaultScope*]

classmethod overwrite_default_scope(*scope_name*)

overwrite the current default scope with *scope_name*

参数 **scope_name**(*Optional[str]*) –

返回类型 Generator

property scope_name: str

Returns: str: Get current scope.

build_from_cfg	Build a module from config dict when it is a class configuration, or call a function from config dict when it is a function configuration.
build_model_from_cfg	Build a PyTorch model from config dict(s).
build_runner_from_cfg	Build a Runner object.
build_scheduler_from_cfg	Builds a `ParamScheduler` instance from config.
count_registered_modules	Scan all modules in MMEngine's root and child registries and dump to json.
traverse_registry_tree	Traverse the whole registry tree from any given node, and collect information of all registered modules in this registry tree.

36.3 mmengine.registry.build_from_cfg

mmengine.registry.**build_from_cfg**(*cfg*, *registry*, *default_args=None*)

Build a module from config dict when it is a class configuration, or call a function from config dict when it is a function configuration.

If the global variable default scope (*DefaultScope*) exists, `build()` will firstly get the responding registry and then call its own `build()`.

At least one of the `cfg` and `default_args` contains the key "type", which should be either str or class. If they all contain it, the key in `cfg` will be used because `cfg` has a high priority than `default_args` that means if a key exists in both of them, the value of the key will be `cfg[key]`. They will be merged first and the key "type" will be popped up and the remaining keys will be used as initialization arguments.

实际案例

```
>>> from mmengine import Registry, build_from_cfg
>>> MODELS = Registry('models')
>>> @MODELS.register_module()
>>> class ResNet:
>>>     def __init__(self, depth, stages=4):
>>>         self.depth = depth
>>>         self.stages = stages
>>> cfg = dict(type='ResNet', depth=50)
>>> model = build_from_cfg(cfg, MODELS)
>>> # Returns an instantiated object
>>> @MODELS.register_module()
>>> def resnet50():
>>>     pass
>>> resnet = build_from_cfg(dict(type='resnet50'), MODELS)
>>> # Return a result of the calling function
```

参数

- **cfg** (*dict or ConfigDict or Config*) –Config dict. It should at least contain the key "type".
- **registry** (*Registry*) –The registry to search the type from.
- **default_args** (*dict or ConfigDict or Config, optional*) –Default initialization arguments. Defaults to None.

返回 The constructed object.

返回类型 object

36.4 mmengine.registry.build_model_from_cfg

mmengine.registry.**build_model_from_cfg**(*cfg*, *registry*, *default_args=None*)

Build a PyTorch model from config dict(s). Different from `build_from_cfg`, if cfg is a list, a `nn.Sequential` will be built.

参数

- **cfg** (*dict, list[dict]*) –The config of modules, which is either a config dict or a list of config dicts. If cfg is a list, the built modules will be wrapped with `nn.Sequential`.
- **registry** (*Registry*) –A registry the module belongs to.
- **default_args** (*dict, optional*) –Default arguments to build the module. Defaults to None.

返回 A built nn.Module.

返回类型 nn.Module

36.5 mmengine.registry.build_runner_from_cfg

mmengine.registry.**build_runner_from_cfg**(*cfg*, *registry*)

Build a Runner object. .. rubric:: 实际案例

```
>>> from mmengine.registry import Registry, build_runner_from_cfg
>>> RUNNERS = Registry('runners', build_func=build_runner_from_cfg)
>>> @RUNNERS.register_module()
>>> class CustomRunner(Runner):
>>>     def setup_env(env_cfg):
>>>         pass
>>> cfg = dict(runner_type='CustomRunner', ...)
>>> custom_runner = RUNNERS.build(cfg)
```

参数

- **cfg** (*dict or ConfigDict or Config*) –Config dict. If "runner_type" key exists, it will be used to build a custom runner. Otherwise, it will be used to build a default runner.
- **registry** (*Registry*) –The registry to search the type from.

返回 The constructed runner object.

返回类型 object

36.6 mmengine.registry.build_scheduler_from_cfg

mmengine.registry.**build_scheduler_from_cfg**(*cfg*, *registry*, *default_args=None*)

Builds a `ParamScheduler` instance from config.

`ParamScheduler` supports building instance by its constructor or method `build_iter_from_epoch`. Therefore, its registry needs a build function to handle both cases.

参数

- **cfg** (*dict or ConfigDict or Config*) –Config dictionary. If it contains the key `convert_to_iter_based`, instance will be built by method `convert_to_iter_based`, otherwise instance will be built by its constructor.
- **registry** (*Registry*) –The `PARAM_SCHEDULERS` registry.

- **default_args** (*dict or ConfigDict or Config, optional*) –Default initialization arguments. It must contain key `optimizer`. If `convert_to_iter_based` is defined in `cfg`, it must additionally contain key `epoch_length`. Defaults to None.

返回 The constructed `ParamScheduler`.

返回类型 object

36.7 mmengine.registry.count_registered_modules

`mmengine.registry.`**count_registered_modules**(*save_path=None*, *verbose=True*)

Scan all modules in MMEngine's root and child registries and dump to json.

参数

- **save_path** (*str, optional*) –Path to save the json file.
- **verbose** (*bool*) –Whether to print log. Defaults to True.

返回 Statistic results of all registered modules.

返回类型 dict

36.8 mmengine.registry.traverse_registry_tree

`mmengine.registry.`**traverse_registry_tree**(*registry*, *verbose=True*)

Traverse the whole registry tree from any given node, and collect information of all registered modules in this registry tree.

参数

- **registry** (`Registry`) –a registry node in the registry tree.
- **verbose** (*bool*) –Whether to print log. Default: True

返回 Statistic results of all modules in each node of the registry tree.

返回类型 list

CHAPTER 37

mmengine.config

Config	A facility for config and config files.
ConfigDict	A dictionary for config which has the same interface as python's built- in dictionary and can be used as a normal dictionary.
DictAction	argparse action to split an argument into KEY=VALUE form on the first = and append to a dictionary.

37.1 Config

class mmengine.config.**Config**(*cfg_dict=None*, *cfg_text=None*, *filename=None*)

A facility for config and config files.

It supports common file formats as configs: python/json/yaml. `Config.fromfile` can parse a dictionary from a config file, then build a `Config` instance with the dictionary. The interface is the same as a dict object and also allows access config values as attributes.

参数

- **cfg_dict** (*dict, optional*) –A config dictionary. Defaults to None.
- **cfg_text** (*str, optional*) –Text of config. Defaults to None.
- **filename** (*str or Path, optional*) –Name of config file. Defaults to None.

实际案例

```
>>> cfg = Config(dict(a=1, b=dict(b1=[0, 1])))
>>> cfg.a
1
>>> cfg.b
{'b1': [0, 1]}
>>> cfg.b.b1
[0, 1]
>>> cfg = Config.fromfile('tests/data/config/a.py')
>>> cfg.filename
"/home/username/projects/mmengine/tests/data/config/a.py"
>>> cfg.item4
'test'
>>> cfg
"Config [path: /home/username/projects/mmengine/tests/data/config/a.py]
:"
"{'item1': [1, 2], 'item2': {'a': 0}, 'item3': True, 'item4': 'test'}"
```

static auto_argparser(*description=None*)

Generate argparser from config file automatically (experimental)

dump(*file=None*)

Dump config to file or return config text.

参数

- **file** (*str or Path, optional*) –If not specified, then the object
- **dumped to a str** (*is*) –
- **to a file specified by the filename.** (*otherwise*) –
- **to None.** (*Defaults*) –

返回 Config text.

返回类型 str or None

property filename: str

get file name of config.

static fromfile(*filename*, *use_predefined_variables=True*, *import_custom_modules=True*)

Build a Config instance from config file.

参数

- **filename** (*str or Path*) –Name of config file.
- **use_predefined_variables** (*bool, optional*) –Whether to use predefined variables. Defaults to True.

- **import_custom_modules** (*bool, optional*) –Whether to support importing custom modules in config. Defaults to True.

返回 Config instance built from config file.

返回类型 *Config*

static fromstring(*cfg_str*, *file_format*)

Build a Config instance from config text.

参数

- **cfg_str** (*str*) –Config text.
- **file_format** (*str*) –Config file format corresponding to the config str. Only py/yml/yaml/json type are supported now!

返回 Config object generated from `cfg_str`.

返回类型 *Config*

merge_from_dict(*options*, *allow_list_keys=True*)

Merge list into cfg_dict.

Merge the dict parsed by MultipleKVAction into this cfg.

参数

- **options** (*dict*) –dict of configs to merge from.
- **allow_list_keys** (*bool*) –If True, int string keys (e.g. '0', '1') are allowed in `options` and will replace the element of the corresponding index in the config if the config is a list. Defaults to True.

返回类型 None

实际案例

```
>>> from mmengine import Config
>>> #  Merge dictionary element
>>> options = {'model.backbone.depth': 50, 'model.backbone.with_cp': True}
>>> cfg = Config(dict(model=dict(backbone=dict(type='ResNet'))))
>>> cfg.merge_from_dict(options)
>>> cfg._cfg_dict
{'model': {'backbone': {'type': 'ResNet', 'depth': 50, 'with_cp': True}}}
>>> # Merge list element
>>> cfg = Config(
>>>     dict(pipeline=[dict(type='LoadImage'),
>>>                    dict(type='LoadAnnotations')]))
>>> options = dict(pipeline={'0': dict(type='SelfLoadImage')})
```

(下页继续)

(续上页)

```
>>> cfg.merge_from_dict(options, allow_list_keys=True)
>>> cfg._cfg_dict
{'pipeline': [{'type': 'SelfLoadImage'}, {'type': 'LoadAnnotations'}]}
```

property pretty_text: str

get formatted python config text.

property text: str

get config text.

37.2 ConfigDict

class mmengine.config.**ConfigDict**(**args*, ***kwargs*)

A dictionary for config which has the same interface as python's built- in dictionary and can be used as a normal dictionary.

The Config class would transform the nested fields (dictionary-like fields) in config file into `ConfigDict`.

37.3 DictAction

class mmengine.config.**DictAction**(*option_strings*, *dest*, *nargs=None*, *const=None*, *default=None*, *type=None*, *choices=None*, *required=False*, *help=None*, *metavar=None*)

argparse action to split an argument into KEY=VALUE form on the first = and append to a dictionary. List options can be passed as comma separated values, i.e 'KEY=V1,V2,V3', or with explicit brackets, i.e. 'KEY=[V1,V2,V3]'. It also support nested brackets to build list/tuple values. e.g. 'KEY=[(V1,V2),(V3,V4)]'

CHAPTER 38

mmengine.runner

mmengine.runner

38.1 Runner

`Runner`	A training helper for PyTorch.

38.1.1 Runner

class mmengine.runner.**Runner**(*model*, *work_dir*, *train_dataloader=None*, *val_dataloader=None*, *test_dataloader=None*, *train_cfg=None*, *val_cfg=None*, *test_cfg=None*, *auto_scale_lr=None*, *optim_wrapper=None*, *param_scheduler=None*, *val_evaluator=None*, *test_evaluator=None*, *default_hooks=None*, *custom_hooks=None*, *data_preprocessor=None*, *load_from=None*, *resume=False*, *launcher='none'*, *env_cfg={'dist_cfg': {'backend': 'nccl'}}*, *log_processor=None*, *log_level='INFO'*, *visualizer=None*, *default_scope='mmengine'*, *randomness={'seed': None}*, *experiment_name=None*, *cfg=None*)

A training helper for PyTorch.

Runner object can be built from config by `runner = Runner.from_cfg(cfg)` where the `cfg` usually contains training, validation, and test-related configurations to build corresponding components. We usually use the same config to launch training, testing, and validation tasks. However, only some of these components are necessary at the same time, e.g., testing a model does not need training or validation-related components.

To avoid repeatedly modifying config, the construction of `Runner` adopts lazy initialization to only initialize components when they are going to be used. Therefore, the model is always initialized at the beginning, and training, validation, and, testing related components are only initialized when calling `runner.train()`, `runner.val()`, and `runner.test()`, respectively.

参数

- **model** (`torch.nn.Module` or dict) –The model to be run. It can be a dict used for build a model.
- **work_dir** (*str*) –The working directory to save checkpoints. The logs will be saved in the subdirectory of *work_dir* named *timestamp*.
- **train_dataloader** (*Dataloader or dict, optional*) –A dataloader object or a dict to build a dataloader. If `None` is given, it means skipping training steps. Defaults to None. See *build_dataloader()* for more details.
- **val_dataloader** (*Dataloader or dict, optional*) –A dataloader object or a dict to build a dataloader. If `None` is given, it means skipping validation steps. Defaults to None. See *build_dataloader()* for more details.
- **test_dataloader** (*Dataloader or dict, optional*) –A dataloader object or a dict to build a dataloader. If `None` is given, it means skipping test steps. Defaults to None. See *build_dataloader()* for more details.
- **train_cfg** (*dict, optional*) –A dict to build a training loop. If it does not provide "type" key, it should contain "by_epoch" to decide which type of training loop *EpochBasedTrainLoop* or *IterBasedTrainLoop* should be used. If `train_cfg` specified, *train_dataloader* should also be specified. Defaults to None. See

build_train_loop() for more details.

- **val_cfg** (*dict, optional*) –A dict to build a validation loop. If it does not provide "type" key, *ValLoop* will be used by default. If `val_cfg` specified, *val_dataloader* should also be specified. If `ValLoop` is built with *fp16=True*', `runner.val()` will be performed under fp16 precision. Defaults to None. See *build_val_loop()* for more details.
- **test_cfg** (*dict, optional*) –A dict to build a test loop. If it does not provide "type" key, *TestLoop* will be used by default. If `test_cfg` specified, *test_dataloader* should also be specified. If `ValLoop` is built with *fp16=True*', `runner.val()` will be performed under fp16 precision. Defaults to None. See *build_test_loop()* for more details.
- **auto_scale_lr** (*dict, Optional*) –Config to scale the learning rate automatically. It includes `base_batch_size` and `enable`. `base_batch_size` is the batch size that the optimizer lr is based on. `enable` is the switch to turn on and off the feature.
- **optim_wrapper** (`OptimWrapper` *or dict, optional*) –Computing gradient of model parameters. If specified, *train_dataloader* should also be specified. If automatic mixed precision or gradient accmulation training is required. The type of `optim_wrapper` should be AmpOptimizerWrapper. See *build_optim_wrapper()* for examples. Defaults to None.
- **param_scheduler** (*_ParamScheduler or dict or list, optional*) –Parameter scheduler for updating optimizer parameters. If specified, `optimizer` should also be specified. Defaults to None. See *build_param_scheduler()* for examples.
- **val_evaluator** (`Evaluator` *or dict or list, optional*) –A evaluator object used for computing metrics for validation. It can be a dict or a list of dict to build a evaluator. If specified, *val_dataloader* should also be specified. Defaults to None.
- **test_evaluator** (`Evaluator` *or dict or list, optional*) –A evaluator object used for computing metrics for test steps. It can be a dict or a list of dict to build a evaluator. If specified, *test_dataloader* should also be specified. Defaults to None.
- **default_hooks** (*dict[str, dict] or dict[str, Hook], optional*) –Hooks to execute default actions like updating model parameters and saving checkpoints. Default hooks are `OptimizerHook`, `IterTimerHook`, `LoggerHook`, `ParamSchedulerHook` and `CheckpointHook`. Defaults to None. See *register_default_hooks()* for more details.
- **custom_hooks** (*list[dict] or list[Hook], optional*) –Hooks to execute custom actions like visualizing images processed by pipeline. Defaults to None.
- **data_preprocessor** (*dict, optional*) –The pre-process config of `BaseDataPreprocessor`. If the `model` argument is a dict and doesn' t contain

the key `data_preprocessor`, set the argument as the `data_preprocessor` of the `model` dict. Defaults to None.

- **load_from** (*str, optional*) –The checkpoint file to load from. Defaults to None.
- **resume** (*bool*) –Whether to resume training. Defaults to False. If `resume` is True and `load_from` is None, automatically to find latest checkpoint from `work_dir`. If not found, resuming does nothing.
- **launcher** (*str*) –Way to launcher multi-process. Supported launchers are 'pytorch', 'mpi', 'slurm' and 'none'. If 'none' is provided, non-distributed environment will be launched.
- **env_cfg** (*dict*) –A dict used for setting environment. Defaults to dict(dist_cfg=dict(backend='nccl')).
- **log_processor** (*dict, optional*) –A processor to format logs. Defaults to None.
- **log_level** (*int or str*) –The log level of MMLogger handlers. Defaults to 'INFO'.
- **visualizer** (*Visualizer or dict, optional*) –A Visualizer object or a dict build Visualizer object. Defaults to None. If not specified, default config will be used.
- **default_scope** (*str*) –Used to reset registries location. Defaults to "mmengine".
- **randomness** (*dict*) –Some settings to make the experiment as reproducible as possible like seed and deterministic. Defaults to `dict(seed=None)`. If seed is None, a random number will be generated and it will be broadcasted to all other processes if in distributed environment. If `cudnn_benchmarch` is `True` in `env_cfg` but `deterministic` is `True` in `randomness`, the value of `torch.backends.cudnn.benchmark` will be `False` finally.
- **experiment_name** (*str, optional*) –Name of current experiment. If not specified, timestamp will be used as `experiment_name`. Defaults to None.
- **cfg** (dict or Configdict or `Config`, optional) –Full config. Defaults to None.

实际案例

```
>>> from mmengine.runner import Runner
>>> cfg = dict(
>>>     model=dict(type='ToyModel'),
>>>     work_dir='path/of/work_dir',
>>>     train_dataloader=dict(
>>>     dataset=dict(type='ToyDataset'),
>>>     sampler=dict(type='DefaultSampler', shuffle=True),
>>>     batch_size=1,
```

(下页继续)

(续上页)

```
>>>     num_workers=0),
>>>     val_dataloader=dict(
>>>         dataset=dict(type='ToyDataset'),
>>>         sampler=dict(type='DefaultSampler', shuffle=False),
>>>        batch_size=1,
>>>        num_workers=0),
>>>     test_dataloader=dict(
>>>         dataset=dict(type='ToyDataset'),
>>>         sampler=dict(type='DefaultSampler', shuffle=False),
>>>         batch_size=1,
>>>         num_workers=0),
>>>     auto_scale_lr=dict(base_batch_size=16, enable=False),
>>>     optim_wrapper=dict(type='OptimizerWrapper', optimizer=dict(
>>>         type='SGD', lr=0.01)),
>>>     param_scheduler=dict(type='MultiStepLR', milestones=[1, 2]),
>>>     val_evaluator=dict(type='ToyEvaluator'),
>>>     test_evaluator=dict(type='ToyEvaluator'),
>>>     train_cfg=dict(by_epoch=True, max_epochs=3, val_interval=1),
>>>     val_cfg=dict(),
>>>     test_cfg=dict(),
>>>     custom_hooks=[],
>>>     default_hooks=dict(
>>>         timer=dict(type='IterTimerHook'),
>>>         checkpoint=dict(type='CheckpointHook', interval=1),
>>>         logger=dict(type='LoggerHook'),
>>>         optimizer=dict(type='OptimizerHook', grad_clip=False),
>>>         param_scheduler=dict(type='ParamSchedulerHook')),
>>>     launcher='none',
>>>     env_cfg=dict(dist_cfg=dict(backend='nccl')),
>>>     log_processor=dict(window_size=20),
>>>     visualizer=dict(type='Visualizer',
>>>     vis_backends=[dict(type='LocalVisBackend',
>>>                        save_dir='temp_dir')])
>>>    )
>>> runner = Runner.from_cfg(cfg)
>>> runner.train()
>>> runner.test()
```

static build_dataloader(*dataloader*, *seed=None*, *diff_rank_seed=False*)

Build dataloader.

The method builds three components:

- Dataset

- Sampler
- Dataloader

An example of `dataloader`:

```
dataloader = dict(
    dataset=dict(type='ToyDataset'),
    sampler=dict(type='DefaultSampler', shuffle=True),
    batch_size=1,
    num_workers=9
)
```

参数

- **dataloader** (*DataLoader or dict*) –A Dataloader object or a dict to build Dataloader object. If `dataloader` is a Dataloader object, just returns itself.
- **seed** (*int, optional*) –Random seed. Defaults to None.
- **diff_rank_seed** (*bool*) –Whether or not set different seeds to different ranks. If True, the seed passed to sampler is set to None, in order to synchronize the seeds used in samplers across different ranks.

返回 DataLoader build from `dataloader_cfg`.

返回类型 Dataloader

build_evaluator(*evaluator*)

Build evaluator.

Examples of `evaluator`:

```
# evaluator could be a built Evaluator instance
evaluator = Evaluator(metrics=[ToyMetric()])

# evaluator can also be a list of dict
evaluator = [
    dict(type='ToyMetric1'),
    dict(type='ToyEvaluator2')
]

# evaluator can also be a list of built metric
evaluator = [ToyMetric1(), ToyMetric2()]

# evaluator can also be a dict with key metrics
evaluator = dict(metrics=ToyMetric())
```

(下页继续)

(续上页)

```
# metric is a list
evaluator = dict(metrics=[ToyMetric()])
```

参数 **evaluator** (*Evaluator or dict or list*) –An Evaluator object or a config dict or list of config dict used to build an Evaluator.

返回 Evaluator build from `evaluator`.

返回类型 *Evaluator*

build_log_processor(*log_processor*)

Build test log_processor.

Examples of `log_processor`:

LogProcessor will be used log_processor = dict()

custom log_processor log_processor = dict(type='CustomLogProcessor')

参数

- **log_processor** (*LogProcessor or dict*) –A log processor or a dict
- **build log processor. If log_processor is a log processor** (*to*) –
- **object** –
- **returns itself.** (*just*) –

返回 Log processor object build from `log_processor_cfg`.

返回类型 *LogProcessor*

build_logger(*log_level='INFO'*, *log_file=None*, ***kwargs*)

Build a global asscessable MMLogger.

参数

- **log_level** (*int or str*) –The log level of MMLogger handlers. Defaults to 'INFO'.
- **log_file** (*str, optional*) –Path of filename to save log. Defaults to None.
- ****kwargs** –Remaining parameters passed to `MMLogger`.

返回 A MMLogger object build from `logger`.

返回类型 *MMLogger*

build_message_hub(*message_hub=None*)

Build a global asscessable MessageHub.

参数 **message_hub** (*dict, optional*) –A dict to build MessageHub object. If not specified, default config will be used to build MessageHub object. Defaults to None.

返回 A MessageHub object build from `message_hub`.

返回类型 *MessageHub*

build_model(*model*)

Build model.

If `model` is a dict, it will be used to build a nn.Module object. Else, if `model` is a nn.Module object it will be returned directly.

An example of `model`:

```
model = dict(type='ResNet')
```

参数 **model** (*nn.Module or dict*) –A `nn.Module` object or a dict to build nn.Module object. If `model` is a nn.Module object, just returns itself.

返回类型 torch.nn.modules.module.Module

注解: The returned model must implement `train_step`, `test_step` if `runner.train` or `runner.test` will be called. If `runner.val` will be called or `val_cfg` is configured, model must implement *val_step*.

返回 Model build from `model`.

返回类型 nn.Module

参数 **model** (*Union[torch.nn.modules.module.Module, Dict]*) –

build_optim_wrapper(*optim_wrapper*)

Build optimizer wrapper.

If `optim_wrapper` is a config dict for only one optimizer, the keys must contain `optimizer`, and `type` is optional. It will build a `OptimWrapper` by default.

If `optim_wrapper` is a config dict for multiple optimizers, i.e., it has multiple keys and each key is for an optimizer wrapper. The constructor must be specified since `DefaultOptimizerConstructor` cannot handle the building of training with multiple optimizers.

If `optim_wrapper` is a dict of pre-built optimizer wrappers, i.e., each value of `optim_wrapper` represents an `OptimWrapper` instance. `build_optim_wrapper` will directly build the `OptimWrapperDict` instance from `optim_wrapper`.

参数 **optim_wrapper** (*OptimWrapper or dict*) –An OptimWrapper object or a dict to build OptimWrapper objects. If `optim_wrapper` is an OptimWrapper, just return an `OptimizeWrapper` instance.

返回类型 Union[*mmengine.optim.optimizer.optimizer_wrapper.OptimWrapper*, *mmengine.optim.optimizer.optimizer_wrapper_dict.OptimWrapperDict*]

注解: For single optimizer training, if *optim_wrapper* is a config dict, *type* is optional(defaults to `OptimWrapper`) and it must contain *optimizer* to build the corresponding optimizer.

实际案例

```
>>> # build an optimizer
>>> optim_wrapper_cfg = dict(type='OptimWrapper', optimizer=dict(
...     type='SGD', lr=0.01))
>>> # optim_wrapper_cfg = dict(optimizer=dict(type='SGD', lr=0.01))
>>> # is also valid.
>>> optim_wrapper = runner.build_optim_wrapper(optim_wrapper_cfg)
>>> optim_wrapper
Type: OptimWrapper
accumulative_counts: 1
optimizer:
SGD (
Parameter Group 0
    dampening: 0
    lr: 0.01
    momentum: 0
    nesterov: False
    weight_decay: 0
)
>>> # build optimizer without `type`
>>> optim_wrapper_cfg = dict(optimizer=dict(type='SGD', lr=0.01))
>>> optim_wrapper = runner.build_optim_wrapper(optim_wrapper_cfg)
>>> optim_wrapper
Type: OptimWrapper
accumulative_counts: 1
optimizer:
SGD (
Parameter Group 0
    dampening: 0
    lr: 0.01
    maximize: False
```

(下页继续)

(续上页)

```
    momentum: 0
    nesterov: False
    weight_decay: 0
)
>>> # build multiple optimizers
>>> optim_wrapper_cfg = dict(
...     generator=dict(type='OptimWrapper', optimizer=dict(
...         type='SGD', lr=0.01)),
...     discriminator=dict(type='OptimWrapper', optimizer=dict(
...         type='Adam', lr=0.001))
...     # need to customize a multiple optimizer constructor
...     constructor='CustomMultiOptimizerConstructor',
...)
>>> optim_wrapper = runner.optim_wrapper(optim_wrapper_cfg)
>>> optim_wrapper
name: generator
Type: OptimWrapper
accumulative_counts: 1
optimizer:
SGD (
Parameter Group 0
    dampening: 0
    lr: 0.1
    momentum: 0
    nesterov: False
    weight_decay: 0
)
name: discriminator
Type: OptimWrapper
accumulative_counts: 1
optimizer:
'discriminator': Adam (
Parameter Group 0
    dampening: 0
    lr: 0.02
    momentum: 0
    nesterov: False
    weight_decay: 0
)
```

重要： If you need to build multiple optimizers, you should implement a MultiOptimWrapperConstructor which gets parameters passed to corresponding optimizers and compose the `OptimWrapperDict`. More

details about how to customize OptimizerConstructor can be found at optimizer-docs.

返回 Optimizer wrapper build from `optimizer_cfg`.

返回类型 *OptimWrapper*

参数 **optim_wrapper** (*Union[torch.optim.optimizer.Optimizer, mmengine.optim.optimizer.optimizer_wrapper.OptimWrapper, Dict]*) –

build_param_scheduler(*scheduler*)

Build parameter schedulers.

`build_param_scheduler` should be called after `build_optim_wrapper` because the building logic will change according to the number of optimizers built by the runner. The cases are as below:

- Single optimizer: When only one optimizer is built and used in the runner, `build_param_scheduler` will return a list of parameter schedulers.
- Multiple optimizers: When two or more optimizers are built and used in runner, `build_param_scheduler` will return a dict containing the same keys with multiple optimizers and each value is a list of parameter schedulers. Note that, if you want different optimizers to use different parameter shedulers to update optimizer's hyper-parameters, the input parameter `scheduler` also needs to be a dict and its key are consistent with multiple optimizers. Otherwise, the same parameter schedulers will be used to update optimizer's hyper-parameters.

参数 **scheduler** (*_ParamScheduler or dict or list*) –A Param Scheduler object or a dict or list of dict to build parameter schedulers.

返回类型 Union[List[*mmengine.optim.scheduler.param_scheduler._ParamScheduler*], Dict[str, List[*mmengine.optim.scheduler.param_scheduler._ParamScheduler*]]]

实际案例

```
>>> # build one scheduler
>>> optim_cfg = dict(dict(type='SGD', lr=0.01))
>>> runner.optim_wrapper = runner.build_optim_wrapper(
>>>     optim_cfg)
>>> scheduler_cfg = dict(type='MultiStepLR', milestones=[1, 2])
>>> schedulers = runner.build_param_scheduler(scheduler_cfg)
>>> schedulers
[<mmengine.optim.scheduler.lr_scheduler.MultiStepLR at 0x7f70f6966290>]  #␣
↪noqa: E501
```

```
>>> # build multiple schedulers
>>> scheduler_cfg = [
...     dict(type='MultiStepLR', milestones=[1, 2]),
...     dict(type='StepLR', step_size=1)
... ]
>>> schedulers = runner.build_param_scheduler(scheduler_cfg)
>>> schedulers
[<mmengine.optim.scheduler.lr_scheduler.MultiStepLR at 0x7f70f60dd3d0>,  #␣
↪noqa: E501
<mmengine.optim.scheduler.lr_scheduler.StepLR at 0x7f70f6eb6150>]
```

Above examples only provide the case of one optimizer and one scheduler or multiple shedulers. If you want to know how to set parameter scheduler when using multiple optimizers, you can find more examples optimizer-docs.

返回 List of parameter schedulers or a dictionary contains list of parameter schedulers build from `scheduler`.

返回类型 list[*_ParamScheduler*] or dict[str, list[*_ParamScheduler*]]

参数 **scheduler** (*Union[mmengine.optim.scheduler.param_scheduler._ParamScheduler, Dict, List]*) –

build_test_loop(*loop*)

Build test loop.

Examples of `loop`:

```
# `TestLoop` will be used
loop = dict()

# custom test loop
loop = dict(type='CustomTestLoop')
```

参数 **loop** (*BaseLoop or dict*) –A test loop or a dict to build test loop. If `loop` is a test loop object, just returns itself.

返回 Test loop object build from `loop_cfg`.

返回类型 *BaseLoop*

build_train_loop(*loop*)

Build training loop.

Examples of `loop`:

```
# `EpochBasedTrainLoop` will be used
loop = dict(by_epoch=True, max_epochs=3)

# `IterBasedTrainLoop` will be used
loop = dict(by_epoch=False, max_epochs=3)

# custom training loop
loop = dict(type='CustomTrainLoop', max_epochs=3)
```

参数 **loop** (*BaseLoop or dict*) –A training loop or a dict to build training loop. If `loop` is a training loop object, just returns itself.

返回 Training loop object build from `loop`.

返回类型 *BaseLoop*

build_val_loop(*loop*)

Build validation loop.

Examples of `loop`:

ValLoop will be used loop = dict()

custom validation loop loop = dict(type=’CustomValLoop’)

参数 **loop** (*BaseLoop or dict*) –A validation loop or a dict to build validation loop. If `loop` is a validation loop object, just returns itself.

返回 Validation loop object build from `loop`.

返回类型 *BaseLoop*

build_visualizer(*visualizer=None*)

Build a global asscessable Visualizer.

参数 **visualizer** (*Visualizer or dict, optional*) –A Visualizer object or a dict to build Visualizer object. If `visualizer` is a Visualizer object, just returns itself. If not specified, default config will be used to build Visualizer object. Defaults to None.

返回 A Visualizer object build from `visualizer`.

返回类型 *Visualizer*

call_hook(*fn_name*, ***kwargs*)

Call all hooks.

参数

- **fn_name** (*str*) –The function name in each hook to be called, such as “before_train_epoch”.

- ****kwargs** –Keyword arguments passed to hook.

返回类型 None

property deterministic

Whether cudnn to select deterministic algorithms.

Type int

property distributed

Whether current environment is distributed.

Type bool

dump_config()

Dump config to *work_dir*.

返回类型 None

property epoch

Current epoch.

Type int

property experiment_name

Name of experiment.

Type str

classmethod from_cfg(*cfg*)

Build a runner from config.

参数 **cfg** (*ConfigType*) –A config used for building runner. Keys of `cfg` can see `__init__()`.

返回 A runner build from `cfg`.

返回类型 *Runner*

property hooks

A list of registered hooks.

Type list[`Hook`]

property iter

Current iteration.

Type int

property launcher

Way to launcher multi processes.

Type str

load_checkpoint(*filename*, *map_location='cpu'*, *strict=False*, *revise_keys=[('^module.', '')]*)

Load checkpoint from given `filename`.

参数

- **filename** (*str*) –Accept local filepath, URL, `torchvision://xxx`, `open-mmlab://xxx`.
- **map_location** (*str or callable*) –A string or a callable function to specifying how to remap storage locations. Defaults to 'cpu'.
- **strict** (*bool*) –strict (bool): Whether to allow different params for the model and checkpoint.
- **revise_keys** (*list*) –A list of customized keywords to modify the state_dict in checkpoint. Each item is a (pattern, replacement) pair of the regular expression operations. Default: strip the prefix 'module.' by [(r'^module.', '')].

load_or_resume()

load or resume checkpoint.

返回类型 None

property max_epochs

Total epochs to train model.

Type int

property max_iters

Total iterations to train model.

Type int

property model_name

Name of the model, usually the module class name.

Type str

property rank

Rank of current process.

Type int

register_custom_hooks(*hooks*)

Register custom hooks into hook list.

参数 **hooks** (*list[Hook | dict]*) –List of hooks or configs to be registered.

返回类型 None

register_default_hooks(*hooks=None*)

Register default hooks into hook list.

hooks will be registered into runner to execute some default actions like updating model parameters or saving checkpoints.

Default hooks and their priorities:

Hooks	Priority
RuntimeInfoHook	VERY_HIGH (10)
IterTimerHook	NORMAL (50)
DistSamplerSeedHook	NORMAL (50)
LoggerHook	BELOW_NORMAL (60)
ParamSchedulerHook	LOW (70)
CheckpointHook	VERY_LOW (90)

If hooks is None, above hooks will be registered by default:

```
default_hooks = dict(
    runtime_info=dict(type='RuntimeInfoHook'),
    timer=dict(type='IterTimerHook'),
    sampler_seed=dict(type='DistSamplerSeedHook'),
    logger=dict(type='LoggerHook'),
    param_scheduler=dict(type='ParamSchedulerHook'),
    checkpoint=dict(type='CheckpointHook', interval=1),
)
```

If not None, hooks will be merged into default_hooks. If there are None value in default_hooks, the corresponding item will be popped from default_hooks:

```
hooks = dict(timer=None)
```

The final registered default hooks will be RuntimeInfoHook, DistSamplerSeedHook, LoggerHook, ParamSchedulerHook and CheckpointHook.

参数 **hooks** (*dict[str, Hook or dict], optional*) –Default hooks or configs to be registered.

返回类型 None

register_hook(*hook*, *priority=None*)

Register a hook into the hook list.

The hook will be inserted into a priority queue, with the specified priority (See *Priority* for details of priorities). For hooks with the same priority, they will be triggered in the same order as they are registered.

Priority of hook will be decided with the following priority:

- priority argument. If priority is given, it will be priority of hook.

- If `hook` argument is a dict and `priority` in it, the priority will be the value of `hook['priority']`.
- If `hook` argument is a dict but `priority` not in it or `hook` is an instance of `hook`, the priority will be `hook.priority`.

参数

- **hook** (`Hook` or dict) –The hook to be registered.
- **priority** (int or str or *`Priority`*, optional) –Hook priority. Lower value means higher priority.

返回类型 None

register_hooks(*default_hooks=None*, *custom_hooks=None*)

Register default hooks and custom hooks into hook list.

参数

- **default_hooks** (*`dict[str, dict] or dict[str, Hook], optional`*) –Hooks to execute default actions like updating model parameters and saving checkpoints. Defaults to None.
- **custom_hooks** (*`list[dict] or list[Hook], optional`*) –Hooks to execute custom actions like visualizing images processed by pipeline. Defaults to None.

返回类型 None

resume(*filename*, *resume_optimizer=True*, *resume_param_scheduler=True*, *map_location='default'*)

Resume model from checkpoint.

参数

- **filename** (*`str`*) –Accept local filepath, URL, `torchvision://xxx`, `open-mmlab://xxx`.
- **resume_optimizer** (*`bool`*) –Whether to resume optimizer state. Defaults to True.
- **resume_param_scheduler** (*`bool`*) –Whether to resume param scheduler state. Defaults to True.
- **map_location** (*`str or callable`*) –A string or a callable function to specifying how to remap storage locations. Defaults to ‘default’.

返回类型 None

save_checkpoint(*out_dir*, *filename*, *file_client_args=None*, *save_optimizer=True*, *save_param_scheduler=True*, *meta=None*, *by_epoch=True*, *backend_args=None*)

Save checkpoints.

`CheckpointHook` invokes this method to save checkpoints periodically.

参数

- **out_dir** (*str*) –The directory that checkpoints are saved.
- **filename** (*str*) –The checkpoint filename.
- **file_client_args** (*dict, optional*) –Arguments to instantiate a FileClient. See *mmengine.fileio.FileClient* for details. Defaults to None. It will be deprecated in future. Please use *backend_args* instead.
- **save_optimizer** (*bool*) –Whether to save the optimizer to the checkpoint. Defaults to True.
- **save_param_scheduler** (*bool*) –Whether to save the param_scheduler to the checkpoint. Defaults to True.
- **meta** (*dict, optional*) –The meta information to be saved in the checkpoint. Defaults to None.
- **by_epoch** (*bool*) –Whether the scheduled momentum is updated by epochs. Defaults to True.
- **backend_args** (*dict, optional*) –Arguments to instantiate the preifx of uri corresponding backend. Defaults to None. New in v0.2.0.

scale_lr(*optim_wrapper*, *auto_scale_lr=None*)

Automatically scaling learning rate in training according to the ratio of `base_batch_size` in `autoscalelr_cfg` and real batch size.

It scales the learning rate linearly according to the paper.

注解: `scale_lr` must be called after building optimizer wrappers and before building parameter schedulers.

参数

- **optim_wrapper** (`OptimWrapper`) –An OptimWrapper object whose parameter groups' learning rate need to be scaled.
- **auto_scale_lr** (*Dict, Optional*) –Config to scale the learning rate automatically. It includes `base_batch_size` and `enable`. `base_batch_size` is the batch size that the optimizer lr is based on. `enable` is the switch to turn on and off the feature.

返回类型 None

property seed

A number to set random modules.

Type int

set_randomness(*seed*, *diff_rank_seed=False*, *deterministic=False*)

Set random seed to guarantee reproducible results.

参数

- **seed** (*int*) –A number to set random modules.
- **diff_rank_seed** (*bool*) –Whether or not set different seeds according to global rank. Defaults to False.
- **deterministic** (*bool*) –Whether to set the deterministic option for CUDNN backend, i.e., set *torch.backends.cudnn.deterministic* to True and *torch.backends.cudnn.benchmark* to False. Defaults to False. See https://pytorch.org/docs/stable/notes/randomness.html for more details.

返回类型 None

setup_env(*env_cfg*)

Setup environment.

An example of `env_cfg`:

```
env_cfg = dict(
    cudnn_benchmark=True,
    mp_cfg=dict(
        mp_start_method='fork',
        opencv_num_threads=0
    ),
    dist_cfg=dict(backend='nccl'),
    resource_limit=4096
)
```

参数 **env_cfg** (*dict*) –Config for setting environment.

返回类型 None

test()

Launch test.

返回 A dict of metrics on testing set.

返回类型 dict

property test_dataloader

The data loader for testing.

property test_evaluator

An evaluator for testing.

Type `Evaluator`

property test_loop

A loop to run testing.

Type *BaseLoop*

property timestamp

Timestamp when creating experiment.

Type str

train()

Launch training.

返回 The model after training.

返回类型 nn.Module

property train_dataloader

The data loader for training.

property train_loop

A loop to run training.

Type *BaseLoop*

val()

Launch validation.

返回 A dict of metrics on validation set.

返回类型 dict

property val_begin

The epoch/iteration to start running validation during training.

Type int

property val_dataloader

The data loader for validation.

property val_evaluator

An evaluator for validation.

Type Evaluator

property val_interval

Interval to run validation during training.

Type int

property val_loop

A loop to run validation.

Type *BaseLoop*

property work_dir

The working directory to save checkpoints and logs.

Type str

property world_size

Number of processes participating in the job.

Type int

wrap_model(*model_wrapper_cfg*, *model*)

Wrap the model to :obj:`MMDistributedDataParallel` or other custom distributed data-parallel module wrappers.

An example of `model_wrapper_cfg`:

```
model_wrapper_cfg = dict(
    broadcast_buffers=False,
    find_unused_parameters=False
)
```

参数

- **model_wrapper_cfg** (*dict, optional*) –Config to wrap model. If not specified, `DistributedDataParallel` will be used in distributed environment. Defaults to None.
- **model** (*nn.Module*) –Model to be wrapped.

返回 nn.Module or subclass of `DistributedDataParallel`.

返回类型 nn.Module or DistributedDataParallel

38.2 Loop

BaseLoop	Base loop class.
EpochBasedTrainLoop	Loop for epoch-based training.
IterBasedTrainLoop	Loop for iter-based training.
ValLoop	Loop for validation.
TestLoop	Loop for test.

38.2.1 BaseLoop

class mmengine.runner.**BaseLoop**(*runner*, *dataloader*)

Base loop class.

All subclasses inherited from `BaseLoop` should overwrite the *`run()`* method.

参数

- **runner** (Runner) –A reference of runner.
- **dataloader** (*Dataloader or dict*) –An iterator to generate one batch of dataset each iteration.

返回类型 None

abstract run()

Execute loop.

返回类型 Any

38.2.2 EpochBasedTrainLoop

class mmengine.runner.**EpochBasedTrainLoop**(*runner*, *dataloader*, *max_epochs*, *val_begin=1*, *val_interval=1*, *dynamic_intervals=None*)

Loop for epoch-based training.

参数

- **runner** (Runner) –A reference of runner.
- **dataloader** (*Dataloader or dict*) –A dataloader object or a dict to build a dataloader.
- **max_epochs** (*int*) –Total training epochs.
- **val_begin** (*int*) –The epoch that begins validating. Defaults to 1.
- **val_interval** (*int*) –Validation interval. Defaults to 1.
- **dynamic_intervals** (*List[Tuple[int, int]], optional*) –The first element in the tuple is a milestone and the second element is a interval. The interval is used after the corresponding milestone. Defaults to None.

返回类型 None

property epoch

Current epoch.

Type int

property iter

Current iteration.

Type int

property max_epochs

Total epochs to train model.

Type int

property max_iters

Total iterations to train model.

Type int

run()

Launch training.

返回类型 torch.nn.modules.module.Module

run_epoch()

Iterate one epoch.

返回类型 None

run_iter(*idx*, *data_batch*)

Iterate one min-batch.

参数 **data_batch** (*Sequence[dict]*) –Batch of data from dataloader.

返回类型 None

38.2.3 IterBasedTrainLoop

class mmengine.runner.**IterBasedTrainLoop**(*runner*, *dataloader*, *max_iters*, *val_begin=1*, *val_interval=1000*, *dynamic_intervals=None*)

Loop for iter-based training.

参数

- **runner** (Runner) –A reference of runner.
- **dataloader** (*Dataloader or dict*) –A dataloader object or a dict to build a dataloader.
- **max_iters** (*int*) –Total training iterations.
- **val_begin** (*int*) –The iteration that begins validating. Defaults to 1.
- **val_interval** (*int*) –Validation interval. Defaults to 1000.

- **dynamic_intervals** (*List[Tuple[int, int]], optional*) –The first element in the tuple is a milestone and the second element is a interval. The interval is used after the corresponding milestone. Defaults to None.

返回类型 None

property epoch

Current epoch.

Type int

property iter

Current iteration.

Type int

property max_epochs

Total epochs to train model.

Type int

property max_iters

Total iterations to train model.

Type int

run()

Launch training.

返回类型 None

run_iter(*data_batch*)

Iterate one mini-batch.

参数 **data_batch** (*Sequence[dict]*) –Batch of data from dataloader.

返回类型 None

38.2.4 ValLoop

class mmengine.runner.**ValLoop**(*runner*, *dataloader*, *evaluator*, *fp16=False*)

Loop for validation.

参数

- **runner** (Runner) –A reference of runner.
- **dataloader** (*Dataloader or dict*) –A dataloader object or a dict to build a dataloader.
- **evaluator** (*Evaluator or dict or list*) –Used for computing metrics.
- **fp16** (*bool*) –Whether to enable fp16 validation. Defaults to False.

返回类型 None

run()

Launch validation.

返回类型 dict

run_iter(*idx*, *data_batch*)

Iterate one mini-batch.

参数 **data_batch** (*Sequence[dict]*) –Batch of data from dataloader.

38.2.5 TestLoop

class mmengine.runner.**TestLoop**(*runner*, *dataloader*, *evaluator*, *fp16=False*)

Loop for test.

参数

- **runner** (Runner) –A reference of runner.
- **dataloader** (*Dataloader or dict*) –A dataloader object or a dict to build a dataloader.
- **evaluator** (*Evaluator or dict or list*) –Used for computing metrics.
- **fp16** (*bool*) –Whether to enable fp16 testing. Defaults to False.

run()

Launch test.

返回类型 dict

run_iter(*idx*, *data_batch*)

Iterate one mini-batch.

参数 **data_batch** (*Sequence[dict]*) –Batch of data from dataloader.

返回类型 None

38.3 Checkpoints

CheckpointLoader	A general checkpoint loader to manage all schemes.

38.3.1 CheckpointLoader

class mmengine.runner.**CheckpointLoader**

A general checkpoint loader to manage all schemes.

classmethod load_checkpoint(*filename*, *map_location=None*, *logger=None*)

load checkpoint through URL scheme path.

参数

- **filename** (*str*) –checkpoint file name with given prefix
- **map_location** (*str, optional*) –Same as `torch.load()`. Default: None
- **logger** (`logging.Logger`, optional) –The logger for message. Default: None

返回 The loaded checkpoint.

返回类型 dict or OrderedDict

classmethod register_scheme(*prefixes*, *loader=None*, *force=False*)

Register a loader to CheckpointLoader.

This method can be used as a normal class method or a decorator.

参数

- **prefixes** (*str or list[str] or tuple[str]*) –
- **prefix of the registered loader.** (*The*) –
- **loader** (*function, optional*) –The loader function to be registered. When this method is used as a decorator, loader is None. Defaults to None.
- **force** (*bool, optional*) –Whether to override the loader if the prefix has already been registered. Defaults to False.

find_latest_checkpoint	Find the latest checkpoint from the given path.
get_deprecated_model_names	
get_external_models	
get_mmcls_models	
get_state_dict	Returns a dictionary containing a whole state of the module.
get_torchvision_models	
load_checkpoint	Load checkpoint from a file or URI.

下页继续

表 4 – 续上页

load_state_dict	Load state_dict to a module.
save_checkpoint	Save checkpoint to file.
weights_to_cpu	Copy a model state_dict to cpu.

38.3.2 mmengine.runner.find_latest_checkpoint

mmengine.runner.**find_latest_checkpoint**(*path*)

Find the latest checkpoint from the given path.

Refer to https://github.com/facebookresearch/fvcore/blob/main/fvcore/common/checkpoint.py # noqa: E501

参数 **path** (*str*) –The path to find checkpoints.

返回 File path of the latest checkpoint.

返回类型 str or None

38.3.3 mmengine.runner.get_deprecated_model_names

mmengine.runner.**get_deprecated_model_names**()

38.3.4 mmengine.runner.get_external_models

mmengine.runner.**get_external_models**()

38.3.5 mmengine.runner.get_mmcls_models

mmengine.runner.**get_mmcls_models**()

38.3.6 mmengine.runner.get_state_dict

mmengine.runner.**get_state_dict**(*module*, *destination=None*, *prefix=''*, *keep_vars=False*)

Returns a dictionary containing a whole state of the module.

Both parameters and persistent buffers (e.g. running averages) are included. Keys are corresponding parameter and buffer names. This method is modified from `torch.nn.Module.state_dict()` to recursively check parallel module in case that the model has a complicated structure, e.g., nn.Module(nn.Module(DDP)).

参数

- **module** (*nn.Module*) –The module to generate state_dict.
- **destination** (*OrderedDict*) –Returned dict for the state of the module.

- **prefix** (*str*) –Prefix of the key.
- **keep_vars** (*bool*) –Whether to keep the variable property of the parameters. Default: False.

返回 A dictionary containing a whole state of the module.

返回类型 dict

38.3.7 mmengine.runner.get_torchvision_models

mmengine.runner.**get_torchvision_models**()

38.3.8 mmengine.runner.load_checkpoint

mmengine.runner.**load_checkpoint**(*model*, *filename*, *map_location=None*, *strict=False*, *logger=None*, *revise_keys=[(r'^module\.', '')]*)

Load checkpoint from a file or URI.

参数

- **model** (*Module*) –Module to load checkpoint.
- **filename** (*str*) –Accept local filepath, URL, `torchvision://xxx`, `open-mmlab://xxx`. Please refer to `docs/model_zoo.md` for details.
- **map_location** (*str*) –Same as `torch.load()`.
- **strict** (*bool*) –Whether to allow different params for the model and checkpoint.
- **logger** (`logging.Logger` or None) –The logger for error message.
- **revise_keys** (*list*) –A list of customized keywords to modify the state_dict in checkpoint. Each item is a (pattern, replacement) pair of the regular expression operations. Default: strip the prefix 'module.' by [(r'^module.', '')].

返回 The loaded checkpoint.

返回类型 dict or OrderedDict

38.3.9 mmengine.runner.load_state_dict

mmengine.runner.**load_state_dict**(*module*, *state_dict*, *strict=False*, *logger=None*)

Load state_dict to a module.

This method is modified from `torch.nn.Module.load_state_dict()`. Default value for `strict` is set to `False` and the message for param mismatch will be shown even if strict is False.

参数

- **module** (*Module*) –Module that receives the state_dict.
- **state_dict** (*OrderedDict*) –Weights.
- **strict** (*bool*) –whether to strictly enforce that the keys in `state_dict` match the keys returned by this module's `state_dict()` function. Default: `False`.
- **logger** (`logging.Logger`, optional) –Logger to log the error message. If not specified, print function will be used.

38.3.10 mmengine.runner.save_checkpoint

mmengine.runner.**save_checkpoint**(*checkpoint*, *filename*, *file_client_args=None*, *backend_args=None*)

Save checkpoint to file.

参数

- **checkpoint** (*dict*) –Module whose params are to be saved.
- **filename** (*str*) –Checkpoint filename.
- **file_client_args** (*dict, optional*) –Arguments to instantiate a FileClient. See *mmengine.fileio.FileClient* for details. Defaults to None. It will be deprecated in future. Please use *backend_args* instead.
- **backend_args** (*dict, optional*) –Arguments to instantiate the preifx of uri corresponding backend. Defaults to None. New in v0.2.0.

38.3.11 mmengine.runner.weights_to_cpu

mmengine.runner.**weights_to_cpu**(*state_dict*)

Copy a model state_dict to cpu.

参数 **state_dict** (*OrderedDict*) –Model weights on GPU.

返回 Model weights on GPU.

返回类型 OrderedDict

38.4 AMP

autocast	A wrapper of `torch.autocast` and `toch.cuda.amp.autocast`.

38.4.1 mmengine.runner.autocast

mmengine.runner.**autocast**(*device_type=None*, *dtype=None*, *enabled=True*, *cache_enabled=None*)

A wrapper of `torch.autocast` and `toch.cuda.amp.autocast`.

Pytorch 1.5.0 provide `torch.cuda.amp.autocast` for running in mixed precision , and update it to `torch.autocast` in 1.10.0. Both interfaces have different arguments, and `torch.autocast` support running with cpu additionally.

This function provides a unified interface by wrapping `torch.autocast` and `torch.cuda.amp.autocast`, which resolves the compatibility issues that `torch.cuda.amp.autocast` does not support running mixed precision with cpu, and both contexts have different arguments. We suggest users using this function in the code to achieve maximized compatibility of different PyTorch versions.

注解： `autocast` requires pytorch version >= 1.5.0. If pytorch version <= 1.10.0 and cuda is not available, it will raise an error with `enabled=True`, since `torch.cuda.amp.autocast` only support cuda mode.

实际案例

```
>>> # case1: 1.10 > Pytorch version >= 1.5.0
>>> with autocast():
>>>     # run in mixed precision context
>>>     pass
>>> with autocast(device_type='cpu')::
>>>     # raise error, torch.cuda.amp.autocast only support cuda mode.
>>>     pass
>>> # case2: Pytorch version >= 1.10.0
>>> with autocast():
>>>     # default cuda mixed precision context
>>>     pass
>>> with autocast(device_type='cpu'):
>>>     # cpu mixed precision context
>>>     pass
>>> with autocast(
>>>      device_type='cuda', enabled=True, cache_enabled=True):
>>>     # enable precision context with more specific arguments.
>>>     pass
```

参数

- **device_type** (*str, required*) –Whether to use ‘cuda’ or ‘cpu’ device.
- **enabled** (*bool*) –Whether autocasting should be enabled in the region. Defaults to True

- **dtype** (*torch_dtype, optional*) –Whether to use `torch.float16` or `torch.bfloat16`.
- **cache_enabled** (*bool, optional*) –Whether the weight cache inside autocast should be enabled.

38.5 Miscellaneous

LogProcessor	A log processor used to format log information collected from `runner.message_hub.log_scalars`.
Priority	Hook priority levels.

38.5.1 LogProcessor

class mmengine.runner.**LogProcessor**(*window_size=10*, *by_epoch=True*, *custom_cfg=None*, *num_digits=4*)

A log processor used to format log information collected from `runner.message_hub.log_scalars`.

`LogProcessor` instance is built by runner and will format `runner.message_hub.log_scalars` to `tag` and `log_str`, which can directly used by `LoggerHook` and `MMLogger`. Besides, the argument `custom_cfg` of constructor can control the statistics method of logs.

参数

- **window_size** (*int*) –default smooth interval Defaults to 10.
- **by_epoch** (*bool*) –Whether to format logs with epoch stype. Defaults to True.
- **custom_cfg** (*list[dict], optional*) –Contains multiple log config dict, in which key means the data source name of log and value means the statistic method and corresponding arguments used to count the data source. Defaults to None.
  - If custom_cfg is None, all logs will be formatted via default methods, such as smoothing loss by default window_size. If custom_cfg is defined as a list of config dict, for example: [dict(data_src=loss, method='mean', log_name='global_loss', window_size='global')]. It means the log item `loss` will be counted as global mean and additionally logged as `global_loss` (defined by `log_name`). If `log_name` is not defined in config dict, the original logged key will be overwritten.
  - The original log item cannot be overwritten twice. Here is an error example: [dict(data_src=loss, method='mean', window_size='global'), dict(data_src=loss, method='mean', window_size='epoch')]. Both log config dict in custom_cfg do not have `log_name` key, which means the loss item will be overwritten twice.

  - For those statistic methods with the `window_size` argument, if `by_epoch` is set to False, `windows_size` should not be *epoch* to statistics log value by epoch.

- **num_digits** (*int*) –The number of significant digit shown in the logging message.

实际案例

```
>>> # `log_name` is defined, `loss_large_window` will be an additional
>>> # record.
>>> log_processor = dict(
>>>     window_size=10,
>>>     by_epoch=True,
>>>     custom_cfg=[dict(data_src='loss',
>>>                      log_name='loss_large_window',
>>>                      method_name='mean',
>>>                      window_size=100)])
>>> # `log_name` is not defined. `loss` will be overwritten.
>>> log_processor = dict(
>>>     window_size=10,
>>>     by_epoch=True,
>>>     custom_cfg=[dict(data_src='loss',
>>>                      method_name='mean',
>>>                      window_size=100)])
>>> # Record loss with different statistics methods.
>>> log_processor = dict(
>>>     window_size=10,
>>>     by_epoch=True,
>>>     custom_cfg=[dict(data_src='loss',
>>>                      log_name='loss_large_window',
>>>                      method_name='mean',
>>>                      window_size=100),
>>>                 dict(data_src='loss',
>>>                      method_name='mean',
>>>                      window_size=100)])
>>> # Overwrite loss item twice will raise an error.
>>> log_processor = dict(
>>>     window_size=10,
>>>     by_epoch=True,
>>>     custom_cfg=[dict(data_src='loss',
>>>                      method_name='mean',
>>>                      window_size=100),
>>>                 dict(data_src='loss',
>>>                      method_name='max',
>>>                      window_size=100)])
```

(下页继续)

(续上页)

```
AssertionError
```

get_log_after_epoch(*runner*, *batch_idx*, *mode*)

Format log string after validation or testing epoch.

参数

- **runner** (Runner) –The runner of validation/testing phase.
- **batch_idx** (*int*) –The index of the current batch in the current loop.
- **mode** (*str*) –Current mode of runner.

返回 Formatted log dict/string which will be recorded by `runner.message_hub` and `runner.visualizer`.

返回类型 Tuple(dict, str)

get_log_after_iter(*runner*, *batch_idx*, *mode*)

Format log string after training, validation or testing epoch.

参数

- **runner** (Runner) –The runner of training phase.
- **batch_idx** (*int*) –The index of the current batch in the current loop.
- **mode** (*str*) –Current mode of runner, train, test or val.

返回 Formatted log dict/string which will be recorded by `runner.message_hub` and `runner.visualizer`.

返回类型 Tuple(dict, str)

38.5.2 Priority

class mmengine.runner.**Priority**(*value*)

Hook priority levels.

Level	Value
HIGHEST	0
VERY_HIGH	10
HIGH	30
ABOVE_NORMAL	40
NORMAL	50
BELOW_NORMAL	60
LOW	70
VERY_LOW	90
LOWEST	100

get_priority	Get priority value.

38.5.3 mmengine.runner.get_priority

mmengine.runner.**get_priority**(*priority*)

Get priority value.

参数 **priority** (int or str or *Priority*) –Priority.

返回 The priority value.

返回类型 int

CHAPTER 39

mmengine.hooks

Hook	Base hook class.
CheckpointHook	Save checkpoints periodically.
EMAHook	A Hook to apply Exponential Moving Average (EMA) on the model during training.
LoggerHook	Collect logs from different components of `Runner` and write them to terminal, JSON file, tensorboard and wandb .etc.
NaiveVisualizationHook	Show or Write the predicted results during the process of testing.
ParamSchedulerHook	A hook to update some hyper-parameters in optimizer, e.g., learning rate and momentum.
RuntimeInfoHook	A hook that updates runtime information into message hub.
DistSamplerSeedHook	Data-loading sampler for distributed training.
IterTimerHook	A hook that logs the time spent during iteration.
SyncBuffersHook	Synchronize model buffers such as running_mean and running_var in BN at the end of each epoch.
EmptyCacheHook	Releases all unoccupied cached GPU memory during the process of training.

39.1 Hook

class mmengine.hooks.**Hook**

Base hook class.

All hooks should inherit from this class.

after_load_checkpoint(*runner*, *checkpoint*)

All subclasses should override this method, if they need any operations after loading the checkpoint.

参数

- **runner** (Runner) –The runner of the training, validation or testing process.
- **checkpoint** (*dict*) –Model’s checkpoint.

返回类型 None

after_run(*runner*)

All subclasses should override this method, if they need any operations before the training validation or testing process.

参数 **runner** (Runner) –The runner of the training, validation or testing process.

返回类型 None

after_test(*runner*)

All subclasses should override this method, if they need any operations after testing.

参数 **runner** (Runner) –The runner of the testing process.

返回类型 None

after_test_epoch(*runner*, *metrics=None*)

All subclasses should override this method, if they need any operations after each test epoch.

参数

- **runner** (Runner) –The runner of the testing process.
- **metrics** (*Dict[str, float], optional*) –Evaluation results of all metrics on test dataset. The keys are the names of the metrics, and the values are corresponding results.

返回类型 None

after_test_iter(*runner*, *batch_idx*, *data_batch=None*, *outputs=None*)

All subclasses should override this method, if they need any operations after each test iteration.

参数

- **runner** (Runner) –The runner of the training process.
- **batch_idx** (*int*) –The index of the current batch in the test loop.

- **data_batch** (*dict or tuple or list, optional*) –Data from dataloader.
- **outputs** (*Sequence, optional*) –Outputs from model.

返回类型 None

after_train(*runner*)

All subclasses should override this method, if they need any operations after train.

参数 **runner** (Runner) –The runner of the training process.

返回类型 None

after_train_epoch(*runner*)

All subclasses should override this method, if they need any operations after each training epoch.

参数 **runner** (Runner) –The runner of the training process.

返回类型 None

after_train_iter(*runner*, *batch_idx*, *data_batch=None*, *outputs=None*)

All subclasses should override this method, if they need any operations after each training iteration.

参数

- **runner** (Runner) –The runner of the training process.
- **batch_idx** (*int*) –The index of the current batch in the train loop.
- **data_batch** (*dict tuple or list, optional*) –Data from dataloader.
- **outputs** (*dict, optional*) –Outputs from model.

返回类型 None

after_val(*runner*)

All subclasses should override this method, if they need any operations after validation.

参数 **runner** (Runner) –The runner of the validation process.

返回类型 None

after_val_epoch(*runner*, *metrics=None*)

All subclasses should override this method, if they need any operations after each validation epoch.

参数

- **runner** (Runner) –The runner of the validation process.
- **metrics** (*Dict[str, float], optional*) –Evaluation results of all metrics on validation dataset. The keys are the names of the metrics, and the values are corresponding results.

返回类型 None

after_val_iter(*runner*, *batch_idx*, *data_batch=None*, *outputs=None*)

All subclasses should override this method, if they need any operations after each validation iteration.

参数

- **runner** (Runner) –The runner of the validation process.
- **batch_idx** (*int*) –The index of the current batch in the val loop.
- **data_batch** (*dict or tuple or list, optional*) –Data from dataloader.
- **outputs** (*Sequence, optional*) –Outputs from model.

返回类型 None

before_run(*runner*)

All subclasses should override this method, if they need any operations before the training validation or testing process.

参数 **runner** (Runner) –The runner of the training, validation or testing process.

返回类型 None

before_save_checkpoint(*runner*, *checkpoint*)

All subclasses should override this method, if they need any operations before saving the checkpoint.

参数

- **runner** (Runner) –The runner of the training, validation or testing process.
- **checkpoint** (*dict*) –Model’s checkpoint.

返回类型 None

before_test(*runner*)

All subclasses should override this method, if they need any operations before testing.

参数 **runner** (Runner) –The runner of the testing process.

返回类型 None

before_test_epoch(*runner*)

All subclasses should override this method, if they need any operations before each test epoch.

参数 **runner** (Runner) –The runner of the testing process.

返回类型 None

before_test_iter(*runner*, *batch_idx*, *data_batch=None*)

All subclasses should override this method, if they need any operations before each test iteration.

参数

- **runner** (Runner) –The runner of the testing process.
- **batch_idx** (*int*) –The index of the current batch in the test loop.

- **data_batch** (*dict or tuple or list, optional*) –Data from dataloader. Defaults to None.

返回类型 None

before_train(*runner*)

All subclasses should override this method, if they need any operations before train.

参数 **runner** (Runner) –The runner of the training process.

返回类型 None

before_train_epoch(*runner*)

All subclasses should override this method, if they need any operations before each training epoch.

参数 **runner** (Runner) –The runner of the training process.

返回类型 None

before_train_iter(*runner*, *batch_idx*, *data_batch=None*)

All subclasses should override this method, if they need any operations before each training iteration.

参数

- **runner** (Runner) –The runner of the training process.
- **batch_idx** (*int*) –The index of the current batch in the train loop.
- **data_batch** (*dict or tuple or list, optional*) –Data from dataloader.

返回类型 None

before_val(*runner*)

All subclasses should override this method, if they need any operations before validation.

参数 **runner** (Runner) –The runner of the validation process.

返回类型 None

before_val_epoch(*runner*)

All subclasses should override this method, if they need any operations before each validation epoch.

参数 **runner** (Runner) –The runner of the validation process.

返回类型 None

before_val_iter(*runner*, *batch_idx*, *data_batch=None*)

All subclasses should override this method, if they need any operations before each validation iteration.

参数

- **runner** (Runner) –The runner of the validation process.
- **batch_idx** (*int*) –The index of the current batch in the val loop.
- **data_batch** (*dict, optional*) –Data from dataloader. Defaults to None.

返回类型 None

end_of_epoch(*dataloader*, *batch_idx*)

Check whether the current iteration reaches the last iteration of the dataloader.

参数

- **dataloader** (*Dataloader*) –The dataloader of the training, validation or testing process.
- **batch_idx** (*int*) –The index of the current batch in the loop.

返回 Whether reaches the end of current epoch or not.

返回类型 bool

every_n_epochs(*runner*, *n*)

Test whether current epoch can be evenly divided by n.

参数

- **runner** (*Runner*) –The runner of the training, validation or testing process.
- **n** (*int*) –Whether current epoch can be evenly divided by n.

返回 Whether current epoch can be evenly divided by n.

返回类型 bool

every_n_inner_iters(*batch_idx*, *n*)

Test whether current inner iteration can be evenly divided by n.

参数

- **batch_idx** (*int*) –Current batch index of the training, validation or testing loop.
- **n** (*int*) –Whether current inner iteration can be evenly divided by n.

返回 Whether current inner iteration can be evenly divided by n.

返回类型 bool

every_n_train_iters(*runner*, *n*)

Test whether current training iteration can be evenly divided by n.

参数

- **runner** (*Runner*) –The runner of the training, validation or testing process.
- **n** (*int*) –Whether current iteration can be evenly divided by n.

返回 Return True if the current iteration can be evenly divided by n, otherwise False.

返回类型 bool

is_last_train_epoch(*runner*)

Test whether current epoch is the last train epoch.

参数 **runner** (Runner) –The runner of the training process.

返回 Whether reaches the end of training epoch.

返回类型 bool

is_last_train_iter(*runner*)

Test whether current iteration is the last train iteration.

参数 **runner** (Runner) –The runner of the training process.

返回 Whether current iteration is the last train iteration.

返回类型 bool

39.2 CheckpointHook

class mmengine.hooks.**CheckpointHook**(*interval=-1*, *by_epoch=True*, *save_optimizer=True*, *save_param_scheduler=True*, *out_dir=None*, *max_keep_ckpts=-1*, *save_last=True*, *save_best=None*, *rule=None*, *greater_keys=None*, *less_keys=None*, *file_client_args=None*, *filename_tmpl=None*, *backend_args=None*, ***kwargs*)

Save checkpoints periodically.

参数

- **interval** (*int*) –The saving period. If `by_epoch=True`, interval indicates epochs, otherwise it indicates iterations. Defaults to -1, which means "never".
- **by_epoch** (*bool*) –Saving checkpoints by epoch or by iteration. Defaults to True.
- **save_optimizer** (*bool*) –Whether to save optimizer state_dict in the checkpoint. It is usually used for resuming experiments. Defaults to True.
- **save_param_scheduler** (*bool*) –Whether to save param_scheduler state_dict in the checkpoint. It is usually used for resuming experiments. Defaults to True.
- **out_dir** (*str, Path, Optional*) –The root directory to save checkpoints. If not specified, `runner.work_dir` will be used by default. If specified, the `out_dir` will be the concatenation of `out_dir` and the last level directory of `runner.work_dir`. For example, if the input `our_dir` is `./tmp` and `runner.work_dir` is `./work_dir/cur_exp`, then the ckpt will be saved in `./tmp/cur_exp`. Defaults to None.
- **max_keep_ckpts** (*int*) –The maximum checkpoints to keep. In some cases we want only the latest few checkpoints and would like to delete old ones to save the disk space. Defaults to -1, which means unlimited.
- **save_last** (*bool*) –Whether to force the last checkpoint to be saved regardless of interval. Defaults to True.

- **save_best** (*str, List[str], optional*) –If a metric is specified, it would measure the best checkpoint during evaluation. If a list of metrics is passed, it would measure a group of best checkpoints corresponding to the passed metrics. The information about best checkpoint(s) would be saved in `runner.message_hub` to keep best score value and best checkpoint path, which will be also loaded when resuming checkpoint. Options are the evaluation metrics on the test dataset. e.g., `bbox_mAP`, `segm_mAP` for bbox detection and instance segmentation. `AR@100` for proposal recall. If `save_best` is `auto`, the first key of the returned `OrderedDict` result will be used. Defaults to None.
- **rule** (*str, List[str], optional*) –Comparison rule for best score. If set to None, it will infer a reasonable rule. Keys such as 'acc', 'top' .etc will be inferred by 'greater' rule. Keys contain 'loss' will be inferred by 'less' rule. If `save_best` is a list of metrics and `rule` is a str, all metrics in `save_best` will share the comparison rule. If `save_best` and `rule` are both lists, their length must be the same, and metrics in `save_best` will use the corresponding comparison rule in `rule`. Options are 'greater', 'less', None and list which contains 'greater' and 'less'. Defaults to None.
- **greater_keys** (*List[str], optional*) –Metric keys that will be inferred by 'greater' comparison rule. If `None`, _default_greater_keys will be used. Defaults to None.
- **less_keys** (*List[str], optional*) –Metric keys that will be inferred by 'less' comparison rule. If `None`, _default_less_keys will be used. Defaults to None.
- **file_client_args** (*dict, optional*) –Arguments to instantiate a FileClient. See *mmengine.fileio.FileClient* for details. Defaults to None. It will be deprecated in future. Please use `backend_args` instead.
- **filename_tmpl** (*str, optional*) –String template to indicate checkpoint name. If specified, must contain one and only one "{}", which will be replaced with `epoch + 1` if `by_epoch=True` else `iteration + 1`. Defaults to None, which means "epoch_{}.pth" or "iter_{}.pth" accordingly.
- **backend_args** (*dict, optional*) –Arguments to instantiate the preifx of uri corresponding backend. Defaults to None. New in v0.2.0.

返回类型 None

实际案例

```
>>> # Save best based on single metric
>>> CheckpointHook(interval=2, by_epoch=True, save_best='acc',
>>>                rule='less')
>>> # Save best based on multi metrics with the same comparison rule
>>> CheckpointHook(interval=2, by_epoch=True,
>>>                save_best=['acc', 'mIoU'], rule='greater')
```

(下页继续)

(续上页)

```
>>> # Save best based on multi metrics with different comparison rule
>>> CheckpointHook(interval=2, by_epoch=True,
>>>                save_best=['FID', 'IS'], rule=['less', 'greater'])
```

after_train_epoch(*runner*)

Save the checkpoint and synchronize buffers after each epoch.

参数 **runner** (Runner) –The runner of the training process.

返回类型 None

after_train_iter(*runner*, *batch_idx*, *data_batch=None*, *outputs=typing.Union[dict, NoneType]*)

Save the checkpoint and synchronize buffers after each iteration.

参数

- **runner** (Runner) –The runner of the training process.
- **batch_idx** (*int*) –The index of the current batch in the train loop.
- **data_batch** (*dict or tuple or list, optional*) –Data from dataloader.
- **outputs** (*dict, optional*) –Outputs from model.

返回类型 None

after_val_epoch(*runner*, *metrics*)

Save the checkpoint and synchronize buffers after each evaluation epoch.

参数

- **runner** (Runner) –The runner of the training process.
- **metrics** (*dict*) –Evaluation results of all metrics

before_train(*runner*)

Finish all operations, related to checkpoint.

This function will get the appropriate file client, and the directory to save these checkpoints of the model.

参数 **runner** (Runner) –The runner of the training process.

返回类型 None

39.3 EMAHook

class mmengine.hooks.**EMAHook**(*ema_type='ExponentialMovingAverage'*, *strict_load=True*, *begin_iter=0*, *begin_epoch=0*, ***kwargs*)

A Hook to apply Exponential Moving Average (EMA) on the model during training.

注解:

- EMAHook takes priority over CheckpointHook.
- The original model parameters are actually saved in ema field after train.
- `begin_iter` and `begin_epoch` cannot be set at the same time.

参数

- **ema_type** (*str*) –The type of EMA strategy to use. You can find the supported strategies in `mmengine.model.averaged_model`. Defaults to 'ExponentialMovingAverage'.
- **strict_load** (*bool*) –Whether to strictly enforce that the keys of `state_dict` in checkpoint match the keys returned by `self.module.state_dict`. Defaults to True.
- **begin_iter** (*int*) –The number of iteration to enable `EMAHook`. Defaults to 0.
- **begin_epoch** (*int*) –The number of epoch to enable `EMAHook`. Defaults to 0.
- ****kwargs** –Keyword arguments passed to subclasses of `BaseAveragedModel`

after_load_checkpoint(*runner*, *checkpoint*)

Resume ema parameters from checkpoint.

参数

- **runner** (Runner) –The runner of the testing process.
- **checkpoint** (*dict*) –

返回类型 None

after_test_epoch(*runner*, *metrics=None*)

We recover source model's parameter from ema model after test.

参数

- **runner** (Runner) –The runner of the testing process.
- **metrics** (*Dict[str, float], optional*) –Evaluation results of all metrics on test dataset. The keys are the names of the metrics, and the values are corresponding results.

返回类型 None

after_train_iter(*runner*, *batch_idx*, *data_batch=None*, *outputs=None*)

Update ema parameter.

参数

- **runner** (Runner) –The runner of the training process.
- **batch_idx** (*int*) –The index of the current batch in the train loop.
- **data_batch** (*Sequence[dict], optional*) –Data from dataloader. Defaults to None.
- **outputs** (*dict, optional*) –Outputs from model. Defaults to None.

返回类型 None

after_val_epoch(*runner*, *metrics=None*)

We recover source model's parameter from ema model after validation.

参数

- **runner** (Runner) –The runner of the validation process.
- **metrics** (*Dict[str, float], optional*) –Evaluation results of all metrics on validation dataset. The keys are the names of the metrics, and the values are corresponding results.

返回类型 None

before_run(*runner*)

Create an ema copy of the model.

参数 **runner** (Runner) –The runner of the training process.

返回类型 None

before_save_checkpoint(*runner*, *checkpoint*)

Save ema parameters to checkpoint.

参数

- **runner** (Runner) –The runner of the testing process.
- **checkpoint** (*dict*) –

返回类型 None

before_test_epoch(*runner*)

We load parameter values from ema model to source model before test.

参数 **runner** (Runner) –The runner of the training process.

返回类型 None

before_train(*runner*)

Check the begin_epoch/iter is smaller than max_epochs/iters.

参数 **runner** (Runner) –The runner of the training process.

返回类型 None

before_val_epoch(*runner*)

We load parameter values from ema model to source model before validation.

参数 **runner** (Runner) –The runner of the training process.

返回类型 None

39.4 LoggerHook

class mmengine.hooks.**LoggerHook**(*interval=10*, *ignore_last=True*, *interval_exp_name=1000*, *out_dir=None*, *out_suffix=('.json', '.log', '.py', 'yaml')*, *keep_local=True*, *file_client_args=None*, *log_metric_by_epoch=True*, *backend_args=None*)

Collect logs from different components of Runner and write them to terminal, JSON file, tensorboard and wandb .etc.

LoggerHook is used to record logs formatted by LogProcessor during training/validation/testing phase. It is used to control following behaviors:

- The frequency of logs update in terminal, local, tensorboad wandb.etc.
- The frequency of show experiment information in terminal.
- The work directory to save logs.

参数

- **interval** (*int*) –Logging interval (every k iterations). Defaults to 10.
- **ignore_last** (*bool*) –Ignore the log of last iterations in each epoch if the number of remaining iterations is less than interval. Defaults to True.
- **interval_exp_name** (*int*) –Logging interval for experiment name. This feature is to help users conveniently get the experiment information from screen or log file. Defaults to 1000.
- **out_dir** (*str or Path, optional*) –The root directory to save checkpoints. If not specified, runner.work_dir will be used by default. If specified, the out_dir will be the concatenation of out_dir and the last level directory of runner.work_dir. For example, if the input our_dir is ./tmp and runner.work_dir is ./work_dir/cur_exp, then the log will be saved in ./tmp/cur_exp. Defaults to None.
- **out_suffix** (*Tuple[str] or str*) –Those files in runner._log_dir ending with out_suffix will be copied to out_dir. Defaults to ('.json', '.log', '.py').

- **keep_local** (*bool*) –Whether to keep local logs in the local machine when `out_dir` is specified. If False, the local log will be removed. Defaults to True.
- **file_client_args** (*dict, optional*) –Arguments to instantiate a FileClient. See *mmengine.fileio.FileClient* for details. Defaults to None. It will be deprecated in future. Please use *backend_args* instead.
- **log_metric_by_epoch** (*bool*) –Whether to output metric in validation step by epoch. It can be true when running in epoch based runner. If set to True, *after_val_epoch* will set *step* to self.epoch in *runner.visualizer.add_scalars*. Otherwise *step* will be self.iter. Default to True.
- **backend_args** (*dict, optional*) –Arguments to instantiate the preifx of uri corresponding backend. Defaults to None. New in v0.2.0.

实际案例

```
>>> # The simplest LoggerHook config.
>>> logger_hook_cfg = dict(interval=20)
```

after_run(*runner*)

Copy logs to `self.out_dir` if `self.out_dir is not None`

参数 **runner** (Runner) –The runner of the training/testing/validation process.

返回类型 None

after_test_epoch(*runner*, *metrics=None*)

All subclasses should override this method, if they need any operations after each test epoch.

参数

- **runner** (Runner) –The runner of the testing process.
- **metrics** (*Dict[str, float], optional*) –Evaluation results of all metrics on test dataset. The keys are the names of the metrics, and the values are corresponding results.

返回类型 None

after_test_iter(*runner*, *batch_idx*, *data_batch=None*, *outputs=None*)

Record logs after testing iteration.

参数

- **runner** (Runner) –The runner of the testing process.
- **batch_idx** (*int*) –The index of the current batch in the test loop.
- **data_batch** (*dict or tuple or list, optional*) –Data from dataloader.
- **outputs** (*sequence, optional*) –Outputs from model.

返回类型 None

after_train_iter(*runner*, *batch_idx*, *data_batch=None*, *outputs=None*)

Record logs after training iteration.

参数

- **runner** (Runner) –The runner of the training process.
- **batch_idx** (*int*) –The index of the current batch in the train loop.
- **data_batch** (*dict tuple or list, optional*) –Data from dataloader.
- **outputs** (*dict, optional*) –Outputs from model.

返回类型 None

after_val_epoch(*runner*, *metrics=None*)

All subclasses should override this method, if they need any operations after each validation epoch.

参数

- **runner** (Runner) –The runner of the validation process.
- **metrics** (*Dict[str, float], optional*) –Evaluation results of all metrics on validation dataset. The keys are the names of the metrics, and the values are corresponding results.

返回类型 None

after_val_iter(*runner*, *batch_idx*, *data_batch=None*, *outputs=None*)

Record logs after validation iteration.

参数

- **runner** (Runner) –The runner of the validation process.
- **batch_idx** (*int*) –The index of the current batch in the validation loop.
- **data_batch** (*dict or tuple or list, optional*) –Data from dataloader. Defaults to None.
- **outputs** (*sequence, optional*) –Outputs from model.

返回类型 None

before_run(*runner*)

Infer `self.file_client` from `self.out_dir`. Initialize the `self.start_iter` and record the meta information.

参数 **runner** (Runner) –The runner of the training process.

返回类型 None

39.5 NaiveVisualizationHook

class mmengine.hooks.**NaiveVisualizationHook**(*interval=1*, *draw_gt=True*, *draw_pred=True*)

Show or Write the predicted results during the process of testing.

参数

- **interval** (*int*) –Visualization interval. Defaults to 1.
- **draw_gt** (*bool*) –Whether to draw the ground truth. Default to True.
- **draw_pred** (*bool*) –Whether to draw the predicted result. Default to True.

after_test_iter(*runner*, *batch_idx*, *data_batch=None*, *outputs=None*)

Show or Write the predicted results.

参数

- **runner** (Runner) –The runner of the training process.
- **batch_idx** (*int*) –The index of the current batch in the test loop.
- **data_batch** (*dict or tuple or list, optional*) –Data from dataloader.
- **outputs** (*Sequence, optional*) –Outputs from model.

返回类型 None

39.6 ParamSchedulerHook

class mmengine.hooks.**ParamSchedulerHook**

A hook to update some hyper-parameters in optimizer, e.g., learning rate and momentum.

after_train_epoch(*runner*)

Call step function for each scheduler after each epoch.

参数 **runner** (Runner) –The runner of the training process.

返回类型 None

after_train_iter(*runner*, *batch_idx*, *data_batch=None*, *outputs=None*)

Call step function for each scheduler after each iteration.

参数

- **runner** (Runner) –The runner of the training process.
- **batch_idx** (*int*) –The index of the current batch in the train loop.
- **data_batch** (*dict or tuple or list, optional*) –Data from dataloader. In order to keep this interface consistent with other hooks, we keep `data_batch` here.

- **outputs** (*dict, optional*) –Outputs from model. In order to keep this interface consistent with other hooks, we keep `data_batch` here.

返回类型 None

39.7 RuntimeInfoHook

class mmengine.hooks.**RuntimeInfoHook**

A hook that updates runtime information into message hub.

E.g. `epoch`, `iter`, `max_epochs`, and `max_iters` for the training state. Components that cannot access the runner can get runtime information through the message hub.

after_test_epoch(*runner*, *metrics=None*)

All subclasses should override this method, if they need any operations after each test epoch.

参数

- **runner** (Runner) –The runner of the testing process.
- **metrics** (*Dict[str, float], optional*) –Evaluation results of all metrics on test dataset. The keys are the names of the metrics, and the values are corresponding results.

返回类型 None

after_train_iter(*runner*, *batch_idx*, *data_batch=None*, *outputs=None*)

Update `log_vars` in model outputs every iteration.

参数

- **runner** (Runner) –The runner of the training process.
- **batch_idx** (*int*) –The index of the current batch in the train loop.
- **data_batch** (*Sequence[dict], optional*) –Data from dataloader. Defaults to None.
- **outputs** (*dict, optional*) –Outputs from model. Defaults to None.

返回类型 None

after_val_epoch(*runner*, *metrics=None*)

All subclasses should override this method, if they need any operations after each validation epoch.

参数

- **runner** (Runner) –The runner of the validation process.
- **metrics** (*Dict[str, float], optional*) –Evaluation results of all metrics on validation dataset. The keys are the names of the metrics, and the values are corresponding results.

返回类型 None

before_run(*runner*)

Update metainfo.

参数 **runner** (Runner) –The runner of the training process.

返回类型 None

before_train(*runner*)

Update resumed training state.

参数 **runner** (Runner) –The runner of the training process.

返回类型 None

before_train_epoch(*runner*)

Update current epoch information before every epoch.

参数 **runner** (Runner) –The runner of the training process.

返回类型 None

before_train_iter(*runner*, *batch_idx*, *data_batch=None*)

Update current iter and learning rate information before every iteration.

参数

- **runner** (Runner) –The runner of the training process.
- **batch_idx** (*int*) –The index of the current batch in the train loop.
- **data_batch** (*Sequence[dict], optional*) –Data from dataloader. Defaults to None.

返回类型 None

39.8 DistSamplerSeedHook

class mmengine.hooks.**DistSamplerSeedHook**

Data-loading sampler for distributed training.

When distributed training, it is only useful in conjunction with `EpochBasedRunner`, while `IterBasedRunner` achieves the same purpose with `IterLoader`.

before_train_epoch(*runner*)

Set the seed for sampler and batch_sampler.

参数 **runner** (Runner) –The runner of the training process.

返回类型 None

39.9 IterTimerHook

class mmengine.hooks.**IterTimerHook**

A hook that logs the time spent during iteration.

E.g. `data_time` for loading data and `time` for a model train step.

before_train(*runner*)

Synchronize the number of iterations with the runner after resuming from checkpoints.

参数 **runner** –The runner of the training, validation or testing process.

返回类型 None

39.10 SyncBuffersHook

class mmengine.hooks.**SyncBuffersHook**

Synchronize model buffers such as running_mean and running_var in BN at the end of each epoch.

返回类型 None

after_train_epoch(*runner*)

All-reduce model buffers at the end of each epoch.

参数 **runner** (Runner) –The runner of the training process.

返回类型 None

39.11 EmptyCacheHook

class mmengine.hooks.**EmptyCacheHook**(*before_epoch=False*, *after_epoch=True*, *after_iter=False*)

Releases all unoccupied cached GPU memory during the process of training.

参数

- **before_epoch** (*bool*) –Whether to release cache before an epoch. Defaults to False.
- **after_epoch** (*bool*) –Whether to release cache after an epoch. Defaults to True.
- **after_iter** (*bool*) –Whether to release cache after an iteration. Defaults to False.

返回类型 None

CHAPTER 40

mmengine.model

mmengine.model

40.1 Module

`BaseModule`	Base module for all modules in openmmlab.
`ModuleDict`	ModuleDict in openmmlab.
`ModuleList`	ModuleList in openmmlab.
`Sequential`	Sequential module in openmmlab.

40.1.1 BaseModule

class mmengine.model.**BaseModule**(*init_cfg=None*)

Base module for all modules in openmmlab. `BaseModule` is a wrapper of `torch.nn.Module` with additional functionality of parameter initialization. Compared with `torch.nn.Module`, `BaseModule` mainly adds three attributes.

- `init_cfg`: the config to control the initialization.
- `init_weights`: The function of parameter initialization and recording initialization information.
- `_params_init_info`: Used to track the parameter initialization information. This attribute only exists during executing the `init_weights`.

参数 **init_cfg** (*dict, optional*) –Initialization config dict.

init_weights()

Initialize the weights.

40.1.2 ModuleDict

class mmengine.model.**ModuleDict**(*modules=None*, *init_cfg=None*)

ModuleDict in openmmlab.

Ensures that all modules in `ModuleDict` have a different initialization strategy than the outer model

参数

- **modules** (*dict, optional*) –A mapping (dictionary) of (string: module) or an iterable of key-value pairs of type (string, module).
- **init_cfg** (*dict, optional*) –Initialization config dict.

40.1.3 ModuleList

class mmengine.model.**ModuleList**(*modules=None*, *init_cfg=None*)

ModuleList in openmmlab.

Ensures that all modules in `ModuleList` have a different initialization strategy than the outer model

参数

- **modules** (*iterable, optional*) –An iterable of modules to add.
- **init_cfg** (*dict, optional*) –Initialization config dict.

40.1.4 Sequential

class mmengine.model.**Sequential**(**args*, *init_cfg=None*)

Sequential module in openmmlab.

Ensures that all modules in `Sequential` have a different initialization strategy than the outer model

参数 **init_cfg**(*dict, optional*) –Initialization config dict.

40.2 Model

BaseModel	Base class for all algorithmic models.
BaseDataPreprocessor	Base data pre-processor used for copying data to the target device.
ImgDataPreprocessor	Image pre-processor for normalization and bgr to rgb conversion.
BaseTTAModel	Base model for inference with test-time augmentation.

40.2.1 BaseModel

class mmengine.model.**BaseModel**(*data_preprocessor=None*, *init_cfg=None*)

Base class for all algorithmic models.

BaseModel implements the basic functions of the algorithmic model, such as weights initialize, batch inputs preprocess(see more information in *BaseDataPreprocessor*), parse losses, and update model parameters.

Subclasses inherit from BaseModel only need to implement the forward method, which implements the logic to calculate loss and predictions, then can be trained in the runner.

实际案例

```
>>> @MODELS.register_module()
>>> class ToyModel(BaseModel):
>>>
>>>     def __init__(self):
>>>         super().__init__()
>>>         self.backbone = nn.Sequential()
>>>         self.backbone.add_module('conv1', nn.Conv2d(3, 6, 5))
>>>         self.backbone.add_module('pool', nn.MaxPool2d(2, 2))
>>>         self.backbone.add_module('conv2', nn.Conv2d(6, 16, 5))
>>>         self.backbone.add_module('fc1', nn.Linear(16 * 5 * 5, 120))
```

(下页继续)

(续上页)

```
>>>         self.backbone.add_module('fc2', nn.Linear(120, 84))
>>>         self.backbone.add_module('fc3', nn.Linear(84, 10))
>>>
>>>         self.criterion = nn.CrossEntropyLoss()
>>>
>>>     def forward(self, batch_inputs, data_samples, mode='tensor'):
>>>         data_samples = torch.stack(data_samples)
>>>         if mode == 'tensor':
>>>             return self.backbone(batch_inputs)
>>>         elif mode == 'predict':
>>>             feats = self.backbone(batch_inputs)
>>>             predictions = torch.argmax(feats, 1)
>>>             return predictions
>>>         elif mode == 'loss':
>>>             feats = self.backbone(batch_inputs)
>>>             loss = self.criterion(feats, data_samples)
>>>             return dict(loss=loss)
```

参数

- **data_preprocessor** (*dict, optional*) –The pre-process config of *BaseDataPreprocessor*.
- **init_cfg** (*dict, optional*) –The weight initialized config for *BaseModule*.

data_preprocessor

Used for pre-processing data sampled by dataloader to the format accepted by *forward()*.

Type *BaseDataPreprocessor*

init_cfg

Initialization config dict.

Type dict, optional

cpu(**args*, ***kwargs*)

Overrides this method to call *BaseDataPreprocessor.cpu()* additionally.

返回 The model itself.

返回类型 nn.Module

cuda(*device=None*)

Overrides this method to call *BaseDataPreprocessor.cuda()* additionally.

返回 The model itself.

返回类型 nn.Module

参数 **device** (*Optional[Union[int, str, torch.device]]*) –

abstract forward (*inputs*, *data_samples=None*, *mode='tensor'*)

Returns losses or predictions of training, validation, testing, and simple inference process.

`forward` method of BaseModel is an abstract method, its subclasses must implement this method.

Accepts `batch_inputs` and `data_sample` processed by *data_preprocessor*, and returns results according to mode arguments.

During non-distributed training, validation, and testing process, `forward` will be called by `BaseModel.train_step`, `BaseModel.val_step` and `BaseModel.val_step` directly.

During distributed data parallel training process, `MMSeparateDistributedDataParallel.train_step` will first call `DistributedDataParallel.forward` to enable automatic gradient synchronization, and then call `forward` to get training loss.

参数

- **inputs** (*torch.Tensor*) –batch input tensor collated by *data_preprocessor*.
- **data_samples** (*list, optional*) –data samples collated by *data_preprocessor*.
- **mode** (*str*) –mode should be one of `loss`, `predict` and `tensor`
  - `loss`: Called by `train_step` and return loss `dict` used for logging
  - `predict`: Called by `val_step` and `test_step` and return list of ‘results used for computing metric.
  - `tensor`: Called by custom use to get `Tensor` type results.

返回

- If `mode == loss`, return a `dict` of loss tensor used for backward and logging.
- If `mode == predict`, return a `list` of inference results.
- If `mode == tensor`, return a tensor or `tuple` of tensor or ‘‘dict of tensor for custom use.

返回类型 dict or list

parse_losses (*losses*)

Parses the raw outputs (losses) of the network.

参数 **losses** (*dict*) –Raw output of the network, which usually contain losses and other necessary information.

返回 There are two elements. The first is the loss tensor passed to optim_wrapper which may be a weighted sum of all losses, and the second is log_vars which will be sent to the logger.

返回类型 tuple[Tensor, dict]

test_step(*data*)

BaseModel implements test_step the same as val_step.

参数 **data** (*dict or tuple or list*) –Data sampled from dataset.

返回 The predictions of given data.

返回类型 list

to(*device=None*, **args*, ***kwargs*)

Overrides this method to call *BaseDataPreprocessor.to()* additionally.

参数 **device** (*int, str or torch.device, optional*) –the desired device of the parameters and buffers in this module.

返回 The model itself.

返回类型 nn.Module

train_step(*data*, *optim_wrapper*)

Implements the default model training process including preprocessing, model forward propagation, loss calculation, optimization, and back-propagation.

During non-distributed training. If subclasses do not override the *train_step()*, EpochBasedTrainLoop or IterBasedTrainLoop will call this method to update model parameters. The default parameter update process is as follows:

1. Calls self.data_processor(data, training=False) to collect batch_inputs and corresponding data_samples(labels).
2. Calls self(batch_inputs, data_samples, mode='loss') to get raw loss
3. Calls self.parse_losses to get parsed_losses tensor used to backward and dict of loss tensor used to log messages.
4. Calls optim_wrapper.update_params(loss) to update model.

参数

- **data** (*dict or tuple or list*) –Data sampled from dataset.
- **optim_wrapper** (OptimWrapper) –OptimWrapper instance used to update model parameters.

返回 A dict of tensor for logging.

返回类型 Dict[str, torch.Tensor]

val_step(*data*)

Gets the predictions of given data.

Calls self.data_preprocessor(data, False) and self(inputs, data_sample, mode='predict') in order. Return the predictions which will be passed to evaluator.

参数 **data** (*dict or tuple or list*) –Data sampled from dataset.

返回 The predictions of given data.

返回类型 list

40.2.2 BaseDataPreprocessor

class mmengine.model.**BaseDataPreprocessor**

Base data pre-processor used for copying data to the target device.

Subclasses inherit from `BaseDataPreprocessor` could override the forward method to implement custom data pre-processing, such as batch-resize, MixUp, or CutMix.

注解: Data dictionary returned by dataloader must be a dict and at least contain the `inputs` key.

---

cast_data(*data*)

Copying data to the target device.

参数 **data** (*dict*) –Data returned by `DataLoader`.

返回 Inputs and data sample at target device.

返回类型 CollatedResult

cpu(**args*, ***kwargs*)

Overrides this method to set the `device`

返回 The model itself.

返回类型 nn.Module

cuda(**args*, ***kwargs*)

Overrides this method to set the `device`

返回 The model itself.

返回类型 nn.Module

forward(*data*, *training=False*)

Preprocesses the data into the model input format.

After the data pre-processing of *cast_data()*, `forward` will stack the input tensor list to a batch tensor at the first dimension.

参数

- **data** (*dict*) –Data returned by dataloader
- **training** (*bool*) –Whether to enable training time augmentation.

返回 Data in the same format as the model input.

返回类型 dict or list

to(*device*, **args*, ***kwargs*)

Overrides this method to set the `device`

参数 **device** (*int or torch.device, optional*) –The desired device of the parameters and buffers in this module.

返回 The model itself.

返回类型 nn.Module

40.2.3 ImgDataPreprocessor

class mmengine.model.**ImgDataPreprocessor**(*mean=None*, *std=None*, *pad_size_divisor=1*, *pad_value=0*, *bgr_to_rgb=False*, *rgb_to_bgr=False*)

Image pre-processor for normalization and bgr to rgb conversion.

Accepts the data sampled by the dataloader, and preprocesses it into the format of the model input. `ImgDataPreprocessor` provides the basic data pre-processing as follows

- Collates and moves data to the target device.
- Converts inputs from bgr to rgb if the shape of input is (3, H, W).
- Normalizes image with defined std and mean.
- Pads inputs to the maximum size of current batch with defined `pad_value`. The padding size can be divisible by a defined `pad_size_divisor`
- Stack inputs to batch_inputs.

For `ImgDataPreprocessor`, the dimension of the single inputs must be (3, H, W).

注解: `ImgDataPreprocessor` and its subclass is built in the constructor of `BaseDataset`.

参数

- **mean** (*Sequence[float or int], optional*) –The pixel mean of image channels. If `bgr_to_rgb=True` it means the mean value of R, G, B channels. If the length of *mean* is 1, it means all channels have the same mean value, or the input is a gray image. If it is not specified, images will not be normalized. Defaults None.
- **std** (*Sequence[float or int], optional*) –The pixel standard deviation of image channels. If `bgr_to_rgb=True` it means the standard deviation of R, G, B channels. If the length of *std* is 1, it means all channels have the same standard deviation, or the input is a gray image. If it is not specified, images will not be normalized. Defaults None.

- **pad_size_divisor** (*int*) –The size of padded image should be divisible by `pad_size_divisor`. Defaults to 1.
- **pad_value** (*float or int*) –The padded pixel value. Defaults to 0.
- **bgr_to_rgb** (*bool*) –whether to convert image from BGR to RGB. Defaults to False.
- **rgb_to_bgr** (*bool*) –whether to convert image from RGB to RGB. Defaults to False.

注解： if images do not need to be normalized, *std* and *mean* should be both set to None, otherwise both of them should be set to a tuple of corresponding values.

---

forward(*data*, *training=False*)

Performs normalization、padding and bgr2rgb conversion based on `BaseDataPreprocessor`.

参数

- **data** (*dict*) –Data sampled from dataset. If the collate function of DataLoader is `pseudo_collate`, data will be a list of dict. If collate function is `default_collate`, data will be a tuple with batch input tensor and list of data samples.
- **training** (*bool*) –Whether to enable training time augmentation. If subclasses override this method, they can perform different preprocessing strategies for training and testing based on the value of `training`.

返回 Data in the same format as the model input.

返回类型 dict or list

40.2.4 BaseTTAModel

class mmengine.model.**BaseTTAModel**(*module*)

Base model for inference with test-time augmentation.

`BaseTTAModel` is a wrapper for inference given multi-batch data. It implements the *`test_step()`* for multi-batch data inference. `multi-batch` data means data processed by different augmentation from the same batch.

During test time augmentation, the data processed by `mmcv.transforms.TestTimeAug`, and then collated by `pseudo_collate` will have the following format:

```
result = dict(
    inputs=[
        [image1_aug1, image2_aug1],
        [image1_aug2, image2_aug2]
    ],
    data_samples=[
```

(下页继续)

(续上页)

```
        [data_sample1_aug1, data_sample2_aug1],
        [data_sample1_aug2, data_sample2_aug2],
    ]
)
```

`image{i}_aug{j}` means the i-th image of the batch, which is augmented by the j-th augmentation.

`BaseTTAModel` will collate the data to:

```
data1 = dict(
    inputs=[image1_aug1, image2_aug1],
    data_samples=[data_sample1_aug1, data_sample2_aug1]
)

data2 = dict(
    inputs=[image1_aug2, image2_aug2],
    data_samples=[data_sample1_aug2, data_sample2_aug2]
)
```

`data1` and `data2` will be passed to model, and the results will be merged by *`merge_preds()`*.

注解: *`merge_preds()`* is an abstract method, all subclasses should implement it.

参数 module (*dict or nn.Module*) –Tested model.

abstract merge_preds(*data_samples_list*)

Merge predictions of enhanced data to one prediction.

参数 data_samples_list (*EnhancedBatchDataSamples*) –List of predictions of all enhanced data.

返回 Merged prediction.

返回类型 List[*BaseDataElement*]

test_step(*data*)

Get predictions of each enhanced data, a multiple predictions.

参数 data (*DataBatch*) –Enhanced data batch sampled from dataloader.

返回 Merged prediction.

返回类型 MergedDataSamples

40.3 EMA

BaseAveragedModel	A base class for averaging model weights.
ExponentialMovingAverage	Implements the exponential moving average (EMA) of the model.
MomentumAnnealingEMA	Exponential moving average (EMA) with momentum annealing strategy.
StochasticWeightAverage	Implements the stochastic weight averaging (SWA) of the model.

40.3.1 BaseAveragedModel

class mmengine.model.**BaseAveragedModel**(*model*, *interval=1*, *device=None*, *update_buffers=False*)

A base class for averaging model weights.

Weight averaging, such as SWA and EMA, is a widely used technique for training neural networks. This class implements the averaging process for a model. All subclasses must implement the *avg_func* method. This class creates a copy of the provided module `model` on the `device` and allows computing running averages of the parameters of the `model`.

The code is referenced from: https://github.com/pytorch/pytorch/blob/master/torch/optim/swa_utils.py.

Different from the *AveragedModel* in PyTorch, we use in-place operation to improve the parameter updating speed, which is about 5 times faster than the non-in-place version.

In mmengine, we provide two ways to use the model averaging:

1. Use the model averaging module in hook: We provide an *mmengine.hooks.EMAHook* to apply the model averaging during training. Add `custom_hooks=[dict(type='EMAHook')]` to the config or the runner.
2. Use the model averaging module directly in the algorithm. Take the ema teacher in semi-supervise as an example:

```
>>> from mmengine.model import ExponentialMovingAverage
>>> student = ResNet(depth=50)
>>> # use ema model as teacher
>>> ema_teacher = ExponentialMovingAverage(student)
```

参数

- **model** (*nn.Module*) –The model to be averaged.
- **interval** (*int*) –Interval between two updates. Defaults to 1.

- **device** (*torch.device, optional*) –If provided, the averaged model will be stored on the `device`. Defaults to None.
- **update_buffers** (*bool*) –if True, it will compute running averages for both the parameters and the buffers of the model. Defaults to False.

返回类型 None

abstract avg_func(*averaged_param*, *source_param*, *steps*)

Use in-place operation to compute the average of the parameters. All subclasses must implement this method.

参数

- **averaged_param** (*Tensor*) –The averaged parameters.
- **source_param** (*Tensor*) –The source parameters.
- **steps** (*int*) –The number of times the parameters have been updated.

返回类型 None

forward(**args*, ***kwargs*)

Forward method of the averaged model.

update_parameters(*model*)

Update the parameters of the model. This method will execute the `avg_func` to compute the new parameters and update the model' s parameters.

参数 **model** (*nn.Module*) –The model whose parameters will be averaged.

返回类型 None

40.3.2 ExponentialMovingAverage

class mmengine.model.**ExponentialMovingAverage**(*model*, *momentum=0.0002*, *interval=1*, *device=None*, *update_buffers=False*)

Implements the exponential moving average (EMA) of the model.

All parameters are updated by the formula as below:

$$Xema_{t+1} = (1 - momentum) * Xema_t + momentum * X_t$$

参数

- **model** (*nn.Module*) –The model to be averaged.
- **momentum** (*float*) –The momentum used for updating ema parameter. Defaults to 0.0002. Ema' s parameter are updated with the formula $averaged\_param = (1 - momentum) * averaged\_param + momentum * source\_param$.

- **interval** (*int*) –Interval between two updates. Defaults to 1.
- **device** (*torch.device, optional*) –If provided, the averaged model will be stored on the `device`. Defaults to None.
- **update_buffers** (*bool*) –if True, it will compute running averages for both the parameters and the buffers of the model. Defaults to False.

返回类型 None

avg_func(*averaged_param*, *source_param*, *steps*)

Compute the moving average of the parameters using exponential moving average.

参数

- **averaged_param** (*Tensor*) –The averaged parameters.
- **source_param** (*Tensor*) –The source parameters.
- **steps** (*int*) –The number of times the parameters have been updated.

返回类型 None

40.3.3 MomentumAnnealingEMA

class mmengine.model.**MomentumAnnealingEMA**(*model*, *momentum=0.0002*, *gamma=100*, *interval=1*, *device=None*, *update_buffers=False*)

Exponential moving average (EMA) with momentum annealing strategy.

参数

- **model** (*nn.Module*) –The model to be averaged.
- **momentum** (*float*) –The momentum used for updating ema parameter. Defaults to 0.0002. Ema's parameter are updated with the formula $averaged\_param = (1 - momentum) * averaged\_param + momentum * source\_param$.
- **gamma** (*int*) –Use a larger momentum early in training and gradually annealing to a smaller value to update the ema model smoothly. The momentum is calculated as max(momentum, gamma / (gamma + steps)) Defaults to 100.
- **interval** (*int*) –Interval between two updates. Defaults to 1.
- **device** (*torch.device, optional*) –If provided, the averaged model will be stored on the `device`. Defaults to None.
- **update_buffers** (*bool*) –if True, it will compute running averages for both the parameters and the buffers of the model. Defaults to False.

返回类型 None

avg_func(*averaged_param*, *source_param*, *steps*)

Compute the moving average of the parameters using the linear momentum strategy.

参数

- **averaged_param** (*Tensor*) –The averaged parameters.
- **source_param** (*Tensor*) –The source parameters.
- **steps** (*int*) –The number of times the parameters have been updated.

返回类型 None

40.3.4 StochasticWeightAverage

class mmengine.model.**StochasticWeightAverage**(*model*, *interval=1*, *device=None*, *update_buffers=False*)

Implements the stochastic weight averaging (SWA) of the model.

Stochastic Weight Averaging was proposed in Averaging Weights Leads to Wider Optima and Better Generalization, UAI 2018. by Pavel Izmailov, Dmitrii Podoprikhin, Timur Garipov, Dmitry Vetrov and Andrew Gordon Wilson.

参数

- **model** (*torch.nn.modules.module.Module*) –
- **interval** (*int*) –
- **device** (*Optional[torch.device]*) –
- **update_buffers** (*bool*) –

返回类型 None

avg_func(*averaged_param*, *source_param*, *steps*)

Compute the average of the parameters using stochastic weight average.

参数

- **averaged_param** (*Tensor*) –The averaged parameters.
- **source_param** (*Tensor*) –The source parameters.
- **steps** (*int*) –The number of times the parameters have been updated.

返回类型 None

40.4 Model Wrapper

MMDistributedDataParallel	A distributed model wrapper used for training,testing and validation in loop.
MMSeparateDistributedDataParallel	A DistributedDataParallel wrapper for models in MMGeneration.
MMFullyShardedDataParallel	A wrapper for sharding Module parameters across data parallel workers.

40.4.1 MMDistributedDataParallel

class mmengine.model.**MMDistributedDataParallel**(*module*, *detect_anomalous_params=False*, ***kwargs*)

A distributed model wrapper used for training,testing and validation in loop.

Different from DistributedDataParallel, MMDistributedDataParallel implements three methods *train_step()*, *val_step()* and *test_step()*, which will be called by `train_loop`, `val_loop` and `test_loop`.

- `train_step`: Called by `runner.train_loop`, and implement default model forward, gradient back propagation, parameter updating logic. To take advantage of DistributedDataParallel's automatic gradient synchronization, `train_step` calls `DistributedDataParallel.forward` to calculate the losses, and call other methods of *BaseModel* to pre-process data and parse losses. Finally, update model parameters by `OptimWrapper` and return the loss dictionary used for logging.
- `val_step`: Called by `runner.val_loop` and get the inference results. Since there is no gradient synchronization requirement, this procedure is equivalent to `BaseModel.val_step`
- `test_step`: Called by `runner.test_loop`, equivalent `val_step`.

参数

- **detect_anomalous_params** (*bool*) –This option is only used for debugging which will slow down the training speed. Detect anomalous parameters that are not included in the computational graph with *loss* as the root. There are two cases
  - Parameters were not used during forward pass.
  - Parameters were not used to produce loss.

  Defaults to False.
- ****kwargs** –keyword arguments passed to `DistributedDataParallel`.
  - device_ids (List[int] or torch.device, optional): CUDA devices for module.

- output_device (int or torch.device, optional): Device location of output for single-device CUDA modules.
- dim (int): Defaults to 0.
- broadcast_buffers (bool): Flag that enables syncing (broadcasting) buffers of the module at beginning of the `forward` function. Defaults to True
- find_unused_parameters (bool): Whether to find parameters of module, which are not in the forward graph. Defaults to False.
- process_group (ProcessGroup, optional): The process group to be used for distributed data all-reduction.
- bucket_cap_mb (int): bucket size in MegaBytes (MB). Defaults to 25.
- check_reduction (bool): This argument is deprecated. Defaults to False.
- gradient_as_bucket_view (bool): Defaults to False.
- static_graph (bool): Defaults to False.

See more information about arguments in `torch.nn.parallel.DistributedDataParallel`.

注解: If model has multiple submodules and each module has separate optimization strategies, *`MMSeparateDistributedDataParallel`* should be used to wrap the model.

注解: If model itself has custom optimization strategy, rather than simply forward model and update model. A custom model wrapper inherit from `MMDistributedDataParallel` should be defined and override the `train_step` method.

---

test_step(*data*)

Gets the predictions of module during testing process.

参数 **data** (*dict or tuple or list*) –Data sampled from dataset.

返回 The predictions of given data.

返回类型 list

train_step(*data*, *optim_wrapper*)

Interface for model forward, backward and parameters updating during training process.

`train_step()` will perform the following steps in order:

- If `module` defines the preprocess method, call `module.preprocess` to pre-processing data.
- Call `module.forward(**data)` and get losses.

- Parse losses.
- Call `optim_wrapper.optimizer_step` to update parameters.
- Return log messages of losses.

参数

- **data** (*dict or tuple or list*) –Data sampled from dataset.
- **optim_wrapper** (OptimWrapper) –A wrapper of optimizer to update parameters.

返回 A `dict` of tensor for logging.

返回类型 Dict[str, torch.Tensor]

val_step(*data*)

Gets the prediction of module during validation process.

参数 **data** (*dict or tuple or list*) –Data sampled from dataset.

返回 The predictions of given data.

返回类型 list

40.4.2 MMSeparateDistributedDataParallel

class mmengine.model.**MMSeparateDistributedDataParallel**(*module*, *broadcast_buffers=False*, *find_unused_parameters=False*, ***kwargs*)

A DistributedDataParallel wrapper for models in MMGeneration.

In MMedting and MMGeneration there is a need to wrap different modules in the models with separate DistributedDataParallel. Otherwise, it will cause errors for GAN training. For example, the GAN model, usually has two submodules: generator and discriminator. If we wrap both of them in one standard DistributedDataParallel, it will cause errors during training, because when we update the parameters of the generator (or discriminator), the parameters of the discriminator (or generator) is not updated, which is not allowed for DistributedDataParallel. So we design this wrapper to separately wrap DistributedDataParallel for generator and discriminator. In this wrapper, we perform two operations:

1. Wraps each module in the models with separate MMDistributedDataParallel. Note that only modules with parameters will be wrapped.
2. Calls `train_step`, `val_step` and `test_step` of submodules to get losses and predictions.

参数

- **module** (*nn.Module*) –model contain multiple submodules which have separately updating strategy.

- **broadcast_buffers** (*bool*) –Same as that in `torch.nn.parallel.distributed.DistributedDataParallel`. Defaults to False.
- **find_unused_parameters** (*bool*) –Same as that in `torch.nn.parallel.distributed.DistributedDataParallel`. Traverse the autograd graph of all tensors contained in returned value of the wrapped module' s forward function. Defaults to False.
- ****kwargs** –Keyword arguments passed to `MMDistributedDataParallel`.
  - device_ids (List[int] or torch.device, optional): CUDA devices for module.
  - output_device (int or torch.device, optional): Device location of output for single-device CUDA modules.
  - dim (int): Defaults to 0.
  - process_group (ProcessGroup, optional): The process group to be used for distributed data all-reduction.
  - bucket_cap_mb (int): bucket size in MegaBytes (MB). Defaults to 25.
  - check_reduction (bool): This argument is deprecated. Defaults to False.
  - gradient_as_bucket_view (bool): Defaults to False.
  - static_graph (bool): Defaults to False.

See more information about arguments in `torch.nn.parallel.DistributedDataParallel`.

no_sync()

Enables `no_sync` context of all sub `MMDistributedDataParallel` modules.

test_step(*data*)

Gets the predictions of module during testing process.

参数 **data** (*dict or tuple or list*) –Data sampled from dataset.

返回 The predictions of given data.

返回类型 list

train(*mode=True*)

Sets the module in training mode.

In order to make the ddp wrapper inheritance hierarchy more uniform, `MMSeparateDistributedDataParallel` inherits from `DistributedDataParallel`, but will not call its constructor. Since the attributes of `DistributedDataParallel` have not been initialized, call the `train` method of `DistributedDataParallel` will raise an error if pytorch version <= 1.9. Therefore, override this method to call the `train` method of submodules.

参数 **mode** (*bool*) –whether to set training mode (`True`) or evaluation mode (`False`). Defaults to `True`.

返回 self.

返回类型 Module

train_step(*data*, *optim_wrapper*)

Interface for model forward, backward and parameters updating during training process.

参数

- **data** (*dict or tuple or list*) –Data sampled from dataset.
- **optim_wrapper** (OptimWrapperDict) –A wrapper of optimizer to update parameters.

返回 A dict of tensor for logging.

返回类型 Dict[str, torch.Tensor]

val_step(*data*)

Gets the prediction of module during validation process.

参数 **data** (*dict or tuple or list*) –Data sampled from dataset.

返回 The predictions of given data.

返回类型 list

40.4.3 MMFullyShardedDataParallel

class mmengine.model.**MMFullyShardedDataParallel**(*module*, *process_group=None*, *cpu_offload=None*, *fsdp_auto_wrap_policy=None*, *backward_prefetch=None*, ***kwargs*)

A wrapper for sharding Module parameters across data parallel workers.

Different from FullyShardedDataParallel, MMFullyShardedDataParallel implements three methods *train_step()*, *val_step()* and *test_step()*, which will be called by `train_loop`, `val_loop` and `test_loop`.

- `train_step`: Called by `runner.train_loop`, and implement default model forward, gradient back propagation, parameter updating logic.
- `val_step`: Called by `runner.val_loop` and get the inference results. Specially, since MMFullyShardedDataParallel will wrap model recursively, it may cause some problem if one just use `BaseModel.val_step` to implement `val_step` here. To avoid that, `val_step` will call methods of *BaseModel* to pre-process data first, and use `FullyShardedDataParallel.forward` to get result.
- `test_step`: Called by `runner.test_loop` and get the inference results. Its logic is equivalent to `val_loop`.

参数

- **module** (*nn.Module*) –module to be wrapped with FSDP.
- **process_group** (*Optional[ProcessGroup]*) –process group for sharding.
- **cpu_offload** (*Optional[Union[bool,CPUOffload]]*) –CPU offloading config. Different from FullyShardedDataParallel,Since it can be set by users' pre-defined config in MMEngine,its type is expected to be *None*, *bool* or *CPUOffload*.

  Currently, only parameter and gradient CPU offload is supported. It can be enabled via passing in `cpu_offload=CPUOffload(offload_params=True)`. Note that this currently implicitly enables gradient offloading to CPU in order for params and grads to be on same device to work with optimizer. This API is subject to change. Default is `None` in which case there will be no offloading.

- **fsdp_auto_wrap_policy** (*Optional[Union[str, Callable]]*) –(Optional[Union[str,Callable]]): Specifying a policy to recursively wrap layers with FSDP. Different from FullyShardedDataParallel, Since it can be set by users' pre-defined config in MMEngine, its type is expected to be *None*, *str* or *Callable*. If it's *str*, then MMFullyShardedDataParallel will try to get specified method in `FSDP_WRAP_POLICIES` registry,and this method will be passed to FullyShardedDataParallel to finally initialize model.

  Note that this policy currently will only apply to child modules of the passed in module. The remainder modules are always wrapped in the returned FSDP root instance. `default_auto_wrap_policy` written in `torch.distributed.fsdp.wrap` is an example of `fsdp_auto_wrap_policy` callable, this policy wraps layers with parameter sizes larger than 100M. Users can supply the customized `fsdp_auto_wrap_policy` callable that should accept following arguments: `module: nn.Module`, `recurse: bool`, `unwrapped_params: int`, extra customized arguments could be added to the customized `fsdp_auto_wrap_policy` callable as well.

  Example:

```
>>> def custom_auto_wrap_policy(
>>>     module: nn.Module,
>>>     recurse: bool,
>>>     unwrapped_params: int,
>>>     # These are customizable for this policy function.
>>>     min_num_params: int = int(1e8),
>>> ) -> bool:
>>>     return unwrapped_params >= min_num_params
```

- **backward_prefetch** (*Optional[Union[str, torch.distributed.fsdp.fully_sharded_data_parallel.BackwardPrefetch]]*) –(Optional[Union[str,BackwardPrefetch]]): Different from FullyShardedDataParallel, Since it will be set by users' pre-defined config in MMEngine,its type is expected to be *None*, *str* or

BackwardPrefetch.

This is an experimental feature that is subject to change in the the near future. It allows users to enable two different backward_prefetch algorithms to help backward communication and computation overlapping. Pros and cons of each algorithm is explained in class `BackwardPrefetch`.

- **`**kwargs`** –Keyword arguments passed to `FullyShardedDataParallel`.

`test_step`(*data*)

Gets the predictions of module during testing process.

参数 **`data`** (*dict*) –Data sampled by dataloader.

返回 The predictions of given data.

返回类型 List[*BaseDataElement*]

`train_step`(*data*, *optim_wrapper*)

Interface for model forward, backward and parameters updating during training process.

`train_step()` will perform the following steps in order:

- **If `module` defines the preprocess method,** call `module.preprocess` to pre-processing data.
- Call `module.forward(**data)` and get losses.
- Parse losses.
- Call `optim_wrapper.optimizer_step` to update parameters.
- Return log messages of losses.

参数

- **`data`** (*dict*) –Data sampled by dataloader.
- **`optim_wrapper`** (`OptimWrapper`) –A wrapper of optimizer to update parameters.

返回 A `dict` of tensor for logging.

返回类型 Dict[str, torch.Tensor]

`val_step`(*data*)

Gets the prediction of module during validation process.

参数 **`data`** (*dict*) –Data sampled by dataloader.

返回 The predictions of given data.

返回类型 List[*BaseDataElement*] or dict

is_model_wrapper	Check if a module is a model wrapper.

40.4.4 is_model_wrapper

class mmengine.model.**is_model_wrapper**(*model*, *registry=Registry(name=model_wrapper, items={'DistributedDataParallel': <class 'torch.nn.parallel.distributed.DistributedDataParallel'>, 'DataParallel': <class 'torch.nn.parallel.data_parallel.DataParallel'>, 'MMDistributedDataParallel': <class 'mmengine.model.wrappers.distributed.MMDistributedDataParallel'>, 'MMSeparateDistributedDataParallel': <class 'mmengine.model.wrappers.seperate_distributed.MMSeparateDistributedDataPara 'MMFullyShardedDataParallel': <class 'mmengine.model.wrappers.fully_sharded_distributed.MMFullyShardedDataPara*)

Check if a module is a model wrapper.

The following 4 model in MMEngine (and their subclasses) are regarded as model wrappers: DataParallel, DistributedDataParallel, MMDataParallel, MMDistributedDataParallel. You may add you own model wrapper by registering it to mmengine.registry.MODEL_WRAPPERS.

参数

- **model** (*nn.Module*) –The model to be checked.
- **registry** (Registry) –The parent registry to search for model wrappers.

返回 True if the input model is a model wrapper.

返回类型 bool

40.5 Weight Initialization

BaseInit	
Caffe2XavierInit	
ConstantInit	Initialize module parameters with constant values.

下页继续

表 6 – 续上页

KaimingInit	Initialize module parameters with the values according to the method described in 'Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification - He, K.
NormalInit	Initialize module parameters with the values drawn from the normal distribution $\mathcal{N}(\text{mean}, \text{std}^2)$.
PretrainedInit	Initialize module by loading a pretrained model.
TruncNormalInit	Initialize module parameters with the values drawn from the normal distribution $\mathcal{N}(\text{mean}, \text{std}^2)$ with values outside $[a, b]$.
UniformInit	Initialize module parameters with values drawn from the uniform distribution $\mathcal{U}(a, b)$.
XavierInit	Initialize module parameters with values according to the method described in 'Understanding the difficulty of training deep feedforward neural networks - Glorot, X.

40.5.1 BaseInit

class mmengine.model.**BaseInit**(*, *bias=0*, *bias_prob=None*, *layer=None*)

40.5.2 Caffe2XavierInit

class mmengine.model.**Caffe2XavierInit**(***kwargs*)

40.5.3 ConstantInit

class mmengine.model.**ConstantInit**(*val*, ***kwargs*)

Initialize module parameters with constant values.

参数

- **val** (*int | float*) –the value to fill the weights in the module with
- **bias** (*int | float*) –the value to fill the bias. Defaults to 0.
- **bias_prob** (*float, optional*) –the probability for bias initialization. Defaults to None.
- **layer** (*str | list[str], optional*) –the layer will be initialized. Defaults to None.

40.5.4 KaimingInit

class mmengine.model.**KaimingInit**(*a=0*, *mode='fan_out'*, *nonlinearity='relu'*, *distribution='normal'*, ***kwargs*)

Initialize module parameters with the values according to the method described in Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification - He, K. et al. (2015).

参数

- **a** (*int | float*) –the negative slope of the rectifier used after this layer (only used with `'leaky_relu'`). Defaults to 0.
- **mode** (*str*) –either `'fan_in'` or `'fan_out'`. Choosing `'fan_in'` preserves the magnitude of the variance of the weights in the forward pass. Choosing `'fan_out'` preserves the magnitudes in the backwards pass. Defaults to `'fan_out'`.
- **nonlinearity** (*str*) –the non-linear function (*nn.functional* name), recommended to use only with `'relu'` or `'leaky_relu'` . Defaults to 'relu'.
- **bias** (*int | float*) –the value to fill the bias. Defaults to 0.
- **bias_prob** (*float, optional*) –the probability for bias initialization. Defaults to None.
- **distribution** (*str*) –distribution either be `'normal'` or `'uniform'`. Defaults to `'normal'`.
- **layer** (*str | list[str], optional*) –the layer will be initialized. Defaults to None.

40.5.5 NormalInit

class mmengine.model.**NormalInit**(*mean=0*, *std=1*, ***kwargs*)

Initialize module parameters with the values drawn from the normal distribution $\mathcal{N}(\text{mean}, \text{std}^2)$.

参数

- **mean** (*int | float*) –the mean of the normal distribution. Defaults to 0.
- **std** (*int | float*) –the standard deviation of the normal distribution. Defaults to 1.
- **bias** (*int | float*) –the value to fill the bias. Defaults to 0.
- **bias_prob** (*float, optional*) –the probability for bias initialization. Defaults to None.
- **layer** (*str | list[str], optional*) –the layer will be initialized. Defaults to None.

40.5.6 PretrainedInit

class mmengine.model.**PretrainedInit**(*checkpoint*, *prefix=None*, *map_location=None*)

Initialize module by loading a pretrained model.

参数

- **checkpoint** (*str*) –the checkpoint file of the pretrained model should be load.
- **prefix** (*str, optional*) –the prefix of a sub-module in the pretrained model. it is for loading a part of the pretrained model to initialize. For example, if we would like to only load the backbone of a detector model, we can set `prefix='backbone.'`. Defaults to None.
- **map_location** (*str*) –map tensors into proper locations.

40.5.7 TruncNormalInit

class mmengine.model.**TruncNormalInit**(*mean=0*, *std=1*, *a=- 2*, *b=2*, ***kwargs*)

Initialize module parameters with the values drawn from the normal distribution $\mathcal{N}(\text{mean}, \text{std}^2)$ with values outside $[a, b]$.

参数

- **mean** (*float*) –the mean of the normal distribution. Defaults to 0.
- **std** (*float*) –the standard deviation of the normal distribution. Defaults to 1.
- **a** (*float*) –The minimum cutoff value.
- **b** (*float*) –The maximum cutoff value.
- **bias** (*float*) –the value to fill the bias. Defaults to 0.
- **bias_prob** (*float, optional*) –the probability for bias initialization. Defaults to None.
- **layer** (*str | list[str], optional*) –the layer will be initialized. Defaults to None.

返回类型 None

40.5.8 UniformInit

class mmengine.model.**UniformInit**(*a=0*, *b=1*, ***kwargs*)

Initialize module parameters with values drawn from the uniform distribution $\mathcal{U}(a, b)$.

参数

- **a** (*int | float*) –the lower bound of the uniform distribution. Defaults to 0.

- **b** (*int | float*) –the upper bound of the uniform distribution. Defaults to 1.
- **bias** (*int | float*) –the value to fill the bias. Defaults to 0.
- **bias_prob** (*float, optional*) –the probability for bias initialization. Defaults to None.
- **layer** (*str | list[str], optional*) –the layer will be initialized. Defaults to None.

40.5.9 XavierInit

class mmengine.model.**XavierInit**(*gain=1*, *distribution='normal'*, ***kwargs*)

Initialize module parameters with values according to the method described in Understanding the difficulty of training deep feedforward neural networks - Glorot, X. & Bengio, Y. (2010).

参数

- **gain** (*int | float*) –an optional scaling factor. Defaults to 1.
- **bias** (*int | float*) –the value to fill the bias. Defaults to 0.
- **bias_prob** (*float, optional*) –the probability for bias initialization. Defaults to None.
- **distribution** (*str*) –distribution either be `'normal'` or `'uniform'`. Defaults to `'normal'`.
- **layer** (*str | list[str], optional*) –the layer will be initialized. Defaults to None.

bias_init_with_prob	initialize conv/fc bias value according to a given probability value.
caffe2_xavier_init	
constant_init	
initialize	Initialize a module.
kaiming_init	
normal_init	
trunc_normal_init	

下页继续

表 7 – 续上页

uniform_init	
update_init_info	Update the *_params_init_info* in the module if the value of parameters are changed.
xavier_init	

40.5.10 mmengine.model.bias_init_with_prob

mmengine.model.**bias_init_with_prob**(*prior_prob*)

initialize conv/fc bias value according to a given probability value.

40.5.11 mmengine.model.caffe2_xavier_init

mmengine.model.**caffe2_xavier_init**(*module*, *bias=0*)

40.5.12 mmengine.model.constant_init

mmengine.model.**constant_init**(*module*, *val*, *bias=0*)

40.5.13 mmengine.model.initialize

mmengine.model.**initialize**(*module*, *init_cfg*)

Initialize a module.

参数

- **module** (torch.nn.Module) –the module will be initialized.
- **init_cfg** (*dict | list[dict]*) –initialization configuration dict to define initializer. OpenMMLab has implemented 6 initializers including Constant, Xavier, Normal, Uniform, Kaiming, and Pretrained.

示例

```
>>> module = nn.Linear(2, 3, bias=True)
>>> init_cfg = dict(type='Constant', layer='Linear', val =1 , bias =2)
>>> initialize(module, init_cfg)
>>> module = nn.Sequential(nn.Conv1d(3, 1, 3), nn.Linear(1,2))
>>> # define key ``'layer'`` for initializing layer with different
>>> # configuration
>>> init_cfg = [dict(type='Constant', layer='Conv1d', val=1),
        dict(type='Constant', layer='Linear', val=2)]
>>> initialize(module, init_cfg)
>>> # define key``'override'`` to initialize some specific part in
>>> # module
>>> class FooNet(nn.Module):
>>>     def __init__(self):
>>>         super().__init__()
>>>         self.feat = nn.Conv2d(3, 16, 3)
>>>         self.reg = nn.Conv2d(16, 10, 3)
>>>         self.cls = nn.Conv2d(16, 5, 3)
>>> model = FooNet()
>>> init_cfg = dict(type='Constant', val=1, bias=2, layer='Conv2d',
>>>     override=dict(type='Constant', name='reg', val=3, bias=4))
>>> initialize(model, init_cfg)
>>> model = ResNet(depth=50)
>>> # Initialize weights with the pretrained model.
>>> init_cfg = dict(type='Pretrained',
        checkpoint='torchvision://resnet50')
>>> initialize(model, init_cfg)
>>> # Initialize weights of a sub-module with the specific part of
>>> # a pretrained model by using "prefix".
>>> url = 'http://download.openmmlab.com/mmdetection/v2.0/retinanet/'\
>>>     'retinanet_r50_fpn_1x_coco/'\
>>>     'retinanet_r50_fpn_1x_coco_20200130-c2398f9e.pth'
>>> init_cfg = dict(type='Pretrained',
        checkpoint=url, prefix='backbone.')
```

40.5.14 mmengine.model.kaiming_init

mmengine.model.**kaiming_init**(*module*, *a=0*, *mode='fan_out'*, *nonlinearity='relu'*, *bias=0*, *distribution='normal'*)

40.5.15 mmengine.model.normal_init

mmengine.model.**normal_init**(*module*, *mean=0*, *std=1*, *bias=0*)

40.5.16 mmengine.model.trunc_normal_init

mmengine.model.**trunc_normal_init**(*module*, *mean=0*, *std=1*, *a=- 2*, *b=2*, *bias=0*)

参数

- **module** (*torch.nn.modules.module.Module*) –
- **mean** (*float*) –
- **std** (*float*) –
- **a** (*float*) –
- **b** (*float*) –
- **bias** (*float*) –

返回类型 None

40.5.17 mmengine.model.uniform_init

mmengine.model.**uniform_init**(*module*, *a=0*, *b=1*, *bias=0*)

40.5.18 mmengine.model.update_init_info

mmengine.model.**update_init_info**(*module*, *init_info*)

Update the *_params_init_info* in the module if the value of parameters are changed.

参数

- **(obj** (*module*) –*nn.Module*): The module of PyTorch with a user-defined attribute *_params_init_info* which records the initialization information.
- **init_info** (*str*) –The string that describes the initialization.

40.5.19 mmengine.model.xavier_init

mmengine.model.**xavier_init**(*module*, *gain=1*, *bias=0*, *distribution='normal'*)

40.6 Utils

detect_anomalous_params	

merge_dict	Merge all dictionaries into one dictionary.
stack_batch	Stack multiple tensors to form a batch and pad the tensor to the max shape use the right bottom padding mode in these images.
revert_sync_batchnorm	Helper function to convert all *SyncBatchNorm* (SyncBN) and *mmcv.ops.sync_bn.SyncBatchNorm'(MMSyncBN) layers in the model to 'BatchNormXd* layers.
convert_sync_batchnorm	Helper function to convert all *BatchNorm* layers in the model to *SyncBatchNorm* (SyncBN) or 'mmcv.ops.sync_bn.SyncBatchNorm'(MMSyncBN) layers. Adapted from <https://pytorch.org/docs/stable/generated/torch.nn.SyncBatchNorm.html#torch.nn.SyncBatchNorm.convert_sync_batchnorm>_.

40.6.1 mmengine.model.detect_anomalous_params

mmengine.model.**detect_anomalous_params**(*loss*, *model*)

参数 **loss** (*torch.Tensor*) –

返回类型 None

40.6.2 mmengine.model.merge_dict

mmengine.model.**merge_dict**(**args*)

Merge all dictionaries into one dictionary.

If pytorch version >= 1.8, `merge_dict` will be wrapped by `torch.fx.wrap`, which will make `torch.fx.symbolic_trace` skip trace `merge_dict`.

注解: If a function needs to be traced by `torch.fx.symbolic_trace`, but inevitably needs to use `update`

method of `dict``(``update` is not traceable). It should use `merge_dict` to replace `xxx.update`.

参数 ***args** –dictionary needs to be merged.

返回 Merged dict from args

返回类型 dict

40.6.3 mmengine.model.stack_batch

`mmengine.model.`**`stack_batch`**(*tensor_list*, *pad_size_divisor=1*, *pad_value=0*)

Stack multiple tensors to form a batch and pad the tensor to the max shape use the right bottom padding mode in these images. If `pad_size_divisor > 0`, add padding to ensure the shape of each dim is divisible by `pad_size_divisor`.

参数

- **tensor_list** (*List[Tensor]*) –A list of tensors with the same dim.
- **pad_size_divisor** (*int*) –If `pad_size_divisor > 0`, add padding to ensure the shape of each dim is divisible by `pad_size_divisor`. This depends on the model, and many models need to be divisible by 32. Defaults to 1
- **pad_value** (*int, float*) –The padding value. Defaults to 0.

返回 The n dim tensor.

返回类型 Tensor

40.6.4 mmengine.model.revert_sync_batchnorm

`mmengine.model.`**`revert_sync_batchnorm`**(*module*)

Helper function to convert all *SyncBatchNorm* (SyncBN) and *mmcv.ops.sync_bn.SyncBatchNorm*(*MMSyncBN) layers in the model to *BatchNormXd* layers.

Adapted from @kapily’s work: (https://github.com/pytorch/pytorch/issues/41081#issuecomment-783961547)

参数 **module** (*nn.Module*) –The module containing *SyncBatchNorm* layers.

返回 The converted module with *BatchNormXd* layers.

返回类型 module_output

40.6.5 mmengine.model.convert_sync_batchnorm

mmengine.model.**convert_sync_batchnorm**(*module*, *implementation='torch'*)

Helper function to convert all *BatchNorm* layers in the model to *SyncBatchNorm* (SyncBN) or ‘mmcv.ops.sync_bn.SyncBatchNorm‘(MMSyncBN) layers. Adapted from <https://pytorch.org/docs/stable/generated/torch.nn.SyncBatchNorm.html#torch.nn.SyncBatchNorm.convert_sync_batchnorm>_.

参数

- **module** (*nn.Module*) –The module containing *SyncBatchNorm* layers.
- **implementation** (*str*) –The type of *SyncBatchNorm* to convert to.
  - ’torch’: convert to *torch.nn.modules.batchnorm.SyncBatchNorm*.
  - ’mmcv’: convert to *mmcv.ops.sync_bn.SyncBatchNorm*.

返回 The converted module with *SyncBatchNorm* layers.

返回类型 nn.Module

CHAPTER 41

mmengine.optim

mmengine.optim

41.1 Optimizer

AmpOptimWrapper	A subclass of *OptimWrapper* that supports automatic mixed precision training based on torch.cuda.amp.
OptimWrapper	Optimizer wrapper provides a common interface for updating parameters.
OptimWrapperDict	A dictionary container of *OptimWrapper*.
DefaultOptimWrapperConstructor	Default constructor for optimizers.

41.1.1 AmpOptimWrapper

class mmengine.optim.**AmpOptimWrapper**(*loss_scale='dynamic'*, ***kwargs*)

A subclass of *OptimWrapper* that supports automatic mixed precision training based on torch.cuda.amp.

AmpOptimWrapper provides a unified interface with OptimWrapper, so AmpOptimWrapper can be used in the same way as OptimWrapper.

> **警告:** AmpOptimWrapper requires PyTorch >= 1.6.

参数

- **loss_scale** (*float or str or dict*) –The initial configuration of *torch.cuda.amp.GradScaler*. See more specific arguments introduction at PyTorch AMP # noqa: E501 Defaults to dynamic.
  - " dynamic" : Initialize GradScale without any arguments.
  - float: Initialize GradScaler with init_scale.
  - dict: Initialize GradScaler with more detail configuration.
- ****kwargs** –Keyword arguments passed to OptimWrapper.

注解: If you use IterBasedRunner and enable gradient accumulation, the original *max_iters* should be multiplied by accumulative_counts.

---

backward(*loss*, ***kwargs*)

Perform gradient back propagation with loss_scaler.

参数

- **loss** (*torch.Tensor*) –The loss of current iteration.
- **kwargs** –Keyword arguments passed to torch.Tensor.backward()

load_state_dict(*state_dict*)

Load and parse the state dictionary of optimizer and loss_scaler.

If state_dict contains "loss_scaler." , the loss_scaler will load the corresponding keys. Otherwise, only the optimizer will load the state dictionary.

参数 **state_dict** (*dict*) –The state dict of optimizer and loss_scaler

optim_context(*model*)

Enables the context for mixed precision training, and enables the context for disabling gradient synchronization during gradient accumulation context.

参数 **model** (*nn.Module*) –The training model.

state_dict()

Get the state dictionary of `optimizer` and `loss_scaler`.

Based on the state dictionary of the optimizer, the returned state dictionary will add a key named "loss_scaler" .

返回 The merged state dict of `loss_scaler` and `optimizer`.

返回类型 dict

step(***kwargs*)

Update parameters with `loss_scaler`.

参数 **kwargs** –Keyword arguments passed to `torch.optim.Optimizer.step()`.

41.1.2 OptimWrapper

class mmengine.optim.**OptimWrapper**(*optimizer*, *accumulative_counts=1*, *clip_grad=None*)

Optimizer wrapper provides a common interface for updating parameters.

Optimizer wrapper provides a unified interface for single precision training and automatic mixed precision training with different hardware. OptimWrapper encapsulates optimizer to provide simplified interfaces for commonly used training techniques such as gradient accumulative and grad clips. `OptimWrapper` implements the basic logic of gradient accumulation and gradient clipping based on `torch.optim.Optimizer`. The subclasses only need to override some methods to implement the mixed precision training. See more information in *AmpOptimWrapper*.

参数

- **optimizer** (*Optimizer*) –Optimizer used to update model parameters.
- **accumulative_counts** (*int*) –The number of iterations to accumulate gradients. The parameters will be updated per `accumulative_counts`.
- **clip_grad** (*dict, optional*) –If `clip_grad` is not None, it will be the arguments of `torch.nn.utils.clip_grad`.

注解: If `accumulative_counts` is larger than 1, perform *update_params()* under the context of `optim_context` could avoid unnecessary gradient synchronization.

注解: If you use `IterBasedRunner` and enable gradient accumulation, the original *max_iters* should be multiplied by `accumulative_counts`.

注解： The subclass should ensure that once *update_params()* is called, `_inner_count += 1` is automatically performed.

实际案例

```
>>> # Config sample of OptimWrapper.
>>> optim_wrapper_cfg = dict(
>>>     type='OptimWrapper',
>>>     _accumulative_counts=1,
>>>     clip_grad=dict(max_norm=0.2))
>>> # Use OptimWrapper to update model.
>>> import torch.nn as nn
>>> import torch
>>> from torch.optim import SGD
>>> from torch.utils.data import DataLoader
>>> from mmengine.optim import OptimWrapper
>>>
>>> model = nn.Linear(1, 1)
>>> dataset = torch.randn(10, 1, 1)
>>> dataloader = DataLoader(dataset)
>>> optimizer = SGD(model.parameters(), lr=0.1)
>>> optim_wrapper = OptimWrapper(optimizer)
>>>
>>> for data in dataloader:
>>>     loss = model(data)
>>>     optim_wrapper.update_params(loss)
>>> # Enable gradient accumulation
>>> optim_wrapper_cfg = dict(
>>>     type='OptimWrapper',
>>>     _accumulative_counts=3,
>>>     clip_grad=dict(max_norm=0.2))
>>> ddp_model = DistributedDataParallel(model)
>>> optimizer = SGD(ddp_model.parameters(), lr=0.1)
>>> optim_wrapper = OptimWrapper(optimizer)
>>> optim_wrapper.initialize_count_status(0, len(dataloader))
>>> # If model is a subclass instance of DistributedDataParallel,
>>> # `optim_context` context manager can avoid unnecessary gradient
>>> #  synchronize.
>>> for iter, data in enumerate(dataloader):
>>>     with optim_wrapper.optim_context(ddp_model):
>>>         loss = model(data)
>>>     optim_wrapper.update_params(loss)
```

backward(*loss*, ***kwargs*)

Perform gradient back propagation.

Provide unified `backward` interface compatible with automatic mixed precision training. Subclass can overload this method to implement the required logic. For example, `torch.cuda.amp` require some extra operation on GradScaler during backward process.

注解： If subclasses inherit from `OptimWrapper` override `backward`, `_inner_count +=1` must be implemented.

参数

- **loss** (*torch.Tensor*) –The loss of current iteration.
- **kwargs** –Keyword arguments passed to `torch.Tensor.backward()`.

返回类型 None

property defaults: dict

A wrapper of `Optimizer.defaults`.

Make OptimizeWrapper compatible with *_ParamScheduler*.

返回 the `param_groups` of `optimizer`.

返回类型 dict

get_lr()

Get the learning rate of the optimizer.

Provide unified interface to get learning rate of optimizer.

返回 Learning rate of the optimizer.

返回类型 Dict[str, List[float]]

get_momentum()

Get the momentum of the optimizer.

Provide unified interface to get momentum of optimizer.

返回 Momentum of the optimizer.

返回类型 Dict[str, List[float]]

initialize_count_status(*model*, *init_counts*, *max_counts*)

Initialize gradient accumulation related attributes.

`OptimWrapper` can be used without calling `initialize_iter_status`. However, Consider the case of `len( dataloader) == 10`, and the `accumulative_iter == 3`. Since 10 is not divisible by 3, the last iteration will not trigger `optimizer.step()`, resulting in one less parameter updating.

参数

- **model** (*nn.Module*) –Training model
- **init_counts** (*int*) –The initial value of the inner count.
- **max_counts** (*int*) –The maximum value of the inner count.

返回类型 None

property inner_count

Get the number of updating parameters of optimizer wrapper.

load_state_dict(*state_dict*)

A wrapper of `Optimizer.load_state_dict`. load the state dict of `optimizer`.

Provide unified `load_state_dict` interface compatible with automatic mixed precision training. Subclass can overload this method to implement the required logic. For example, the state dictionary of GradScaler should be loaded when training with `torch.cuda.amp`.

参数 **state_dict** (*dict*) –The state dictionary of `optimizer`.

返回类型 None

optim_context(*model*)

A Context for gradient accumulation and automatic mix precision training.

If subclasses need to enable the context for mix precision training, e.g., `:class:`AmpOptimWrapper`, the corresponding context should be enabled in *optim_context*. Since `OptimWrapper` uses default fp32 training, `optim_context` will only enable the context for blocking the unnecessary gradient synchronization during gradient accumulation

If model is an instance with `no_sync` method (which means blocking the gradient synchronization) and `self._accumulative_counts != 1`. The model will not automatically synchronize gradients if `cur_iter` is divisible by `self._accumulative_counts`. Otherwise, this method will enable an empty context.

参数 **model** (*nn.Module*) –The training model.

property param_groups: List[dict]

A wrapper of `Optimizer.param_groups`.

Make OptimizeWrapper compatible with *_ParamScheduler*.

返回 the `param_groups` of `optimizer`.

返回类型 dict

scale_loss(*loss*)

Get scaled loss according to `_accumulative_counts`, `_inner_count` and max_counts.

参数 **loss** (*torch.Tensor*) –Original loss calculated by model.

返回 Scaled loss.

返回类型 loss (torch.Tensor)

should_sync()

Decide whether the automatic gradient synchronization should be allowed at the current iteration.

It takes effect when gradient accumulation is used to skip synchronization at the iterations where the parameter is not updated.

Since `should_sync` is called by *optim_context()*, and it is called before *backward()* which means `self._inner_count += 1` has not happened yet. Therefore, `self._inner_count += 1` should be performed manually here.

返回 Whether to block the automatic gradient synchronization.

返回类型 bool

should_update()

Decide whether the parameters should be updated at the current iteration.

Called by *update_params()* and check whether the optimizer wrapper should update parameters at current iteration.

返回 Whether to update parameters.

返回类型 bool

state_dict()

A wrapper of `Optimizer.state_dict`.

Provide unified `state_dict` interface compatible with automatic mixed precision training. Subclass can overload this method to implement the required logic. For example, the state dictionary of GradScaler should be saved when training with `torch.cuda.amp`.

返回 The state dictionary of `optimizer`.

返回类型 dict

step(***kwargs*)

A wrapper of `Optimizer.step`.

Provide unified `step` interface compatible with automatic mixed precision training. Subclass can overload this method to implement the required logic. For example, `torch.cuda.amp` require some extra operation on `GradScaler` during step process.

Clip grad if `clip_grad_kwargs` is not None, and then update parameters.

参数 **kwargs** –Keyword arguments passed to `torch.optim.Optimizer.step()`.

返回类型 None

update_params(*loss*)

Update parameters in `optimizer`.

参数 **loss** (*torch.Tensor*) –A tensor for back propagation.

返回类型 None

zero_grad(***kwargs*)

A wrapper of `Optimizer.zero_grad`.

Provide unified `zero_grad` interface compatible with automatic mixed precision training. Subclass can overload this method to implement the required logic.

参数 **kwargs** –Keyword arguments passed to `torch.optim.Optimizer.zero_grad()`.

返回类型 None

41.1.3 OptimWrapperDict

class mmengine.optim.**OptimWrapperDict**(***optim_wrapper_dict*)

A dictionary container of *OptimWrapper*.

If runner is training with multiple optimizers, all optimizer wrappers should be managed by *OptimWrapperDict* which is built by `CustomOptimWrapperConstructor`. `OptimWrapperDict` will load and save the state dictionary of all optimizer wrappers.

Consider the semantic ambiguity of calling :meth:`update_params`, *backward()* of all optimizer wrappers, `OptimWrapperDict` will not implement these methods.

实际案例

```
>>> import torch.nn as nn
>>> from torch.optim import SGD
>>> from mmengine.optim import OptimWrapperDict, OptimWrapper
>>> model1 = nn.Linear(1, 1)
>>> model2 = nn.Linear(1, 1)
>>> optim_wrapper1 = OptimWrapper(SGD(model1.parameters(), lr=0.1))
>>> optim_wrapper2 = OptimWrapper(SGD(model2.parameters(), lr=0.1))
>>> optim_wrapper_dict = OptimWrapperDict(model1=optim_wrapper1,
>>>                                       model2=optim_wrapper2)
```

注解: The optimizer wrapper contained in `OptimWrapperDict` can be accessed in the same way as *dict*.

参数

- ****optim_wrappers** –A dictionary of `OptimWrapper` instance.
- **optim_wrapper_dict** (`mmengine.optim.optimizer.optimizer_wrapper.OptimWrapper`) –

backward(*loss*, ***kwargs*)

Since OptimWrapperDict doesn't know which optimizer wrapper's backward method should be called (`loss_scaler` maybe different in different :obj:AmpOptimWrapper), this method is not implemented.

The optimizer wrapper of OptimWrapperDict should be accessed and call its 'backward.

参数 **loss** (*torch.Tensor*) –

返回类型 None

get_lr()

Get the learning rate of all optimizers.

返回 Learning rate of all optimizers.

返回类型 Dict[str, List[float]]

get_momentum()

Get the momentum of all optimizers.

返回 momentum of all optimizers.

返回类型 Dict[str, List[float]]

initialize_count_status(*model*, *cur_iter*, *max_iters*)

Do nothing but provide unified interface for *OptimWrapper*

Since `OptimWrapperDict` does not know the correspondence between model and optimizer wrapper. `initialize_iter_status` will do nothing and each optimizer wrapper should call `initialize_iter_status` separately.

参数 **model** (*torch.nn.modules.module.Module*) –

返回类型 None

items()

A generator to get the name and corresponding *OptimWrapper*

返回类型 Iterator[Tuple[str, *mmengine.optim.optimizer.optimizer_wrapper.OptimWrapper*]]

keys()

A generator to get the name of *OptimWrapper*

返回类型 Iterator[str]

load_state_dict(*state_dict*)

Load the state dictionary from the `state_dict`.

参数 **state_dict** (*dict*) –Each key-value pair in *state_dict* represents the name and the state dictionary of corresponding *OptimWrapper*.

返回类型 None

optim_context(*model*)

optim_context should be called by each optimizer separately.

参数 **model** (*torch.nn.modules.module.Module*) –

property param_groups

Returns the parameter groups of each OptimWrapper.

state_dict()

Get the state dictionary of all optimizer wrappers.

返回 Each key-value pair in the dictionary represents the name and state dictionary of corresponding *OptimWrapper*.

返回类型 dict

step(***kwargs*)

Since the backward method is not implemented, the step should not be implemented either.

返回类型 None

update_params(*loss*)

Update all optimizer wrappers would lead to a duplicate backward errors, and OptimWrapperDict does not know which optimizer wrapper should be updated.

Therefore, this method is not implemented. The optimizer wrapper of OptimWrapperDict should be accessed and call its ‘update_params.

参数 **loss** (*torch.Tensor*) –

返回类型 None

values()

A generator to get *OptimWrapper*

返回类型 Iterator[*mmengine.optim.optimizer.optimizer_wrapper.OptimWrapper*]

zero_grad(***kwargs*)

Set the gradients of all optimizer wrappers to zero.

返回类型 None

41.1.4 DefaultOptimWrapperConstructor

class mmengine.optim.**DefaultOptimWrapperConstructor**(*optim_wrapper_cfg*, *paramwise_cfg=None*)

Default constructor for optimizers.

By default, each parameter share the same optimizer settings, and we provide an argument paramwise_cfg to specify parameter-wise settings. It is a dict and may contain the following fields:

- custom_keys (dict): Specified parameters-wise settings by keys. If one of the keys in custom_keys is a substring of the name of one parameter, then the setting of the parameter will be specified by custom_keys[key] and other setting like bias_lr_mult etc. will be ignored. It should be noted that the aforementioned key is the longest key that is a substring of the name of the parameter. If there are multiple matched keys with the same length, then the key with lower alphabet order will be chosen. custom_keys[key] should be a dict and may contain fields lr_mult and decay_mult. See Example 2 below.
- bias_lr_mult (float): It will be multiplied to the learning rate for all bias parameters (except for those in normalization layers and offset layers of DCN).
- bias_decay_mult (float): It will be multiplied to the weight decay for all bias parameters (except for those in normalization layers, depthwise conv layers, offset layers of DCN).
- norm_decay_mult (float): It will be multiplied to the weight decay for all weight and bias parameters of normalization layers.
- dwconv_decay_mult (float): It will be multiplied to the weight decay for all weight and bias parameters of depthwise conv layers.
- dcn_offset_lr_mult (float): It will be multiplied to the learning rate for parameters of offset layer in the deformable convs of a model.
- bypass_duplicate (bool): If true, the duplicate parameters would not be added into optimizer. Default: False.

注解: 1. If the option dcn_offset_lr_mult is used, the constructor will override the effect of bias_lr_mult in the bias of offset layer. So be careful when using both bias_lr_mult and dcn_offset_lr_mult. If you wish to apply both of them to the offset layer in deformable convs, set dcn_offset_lr_mult to the original dcn_offset_lr_mult * bias_lr_mult.

2. If the option dcn_offset_lr_mult is used, the constructor will apply it to all the DCN layers in the model. So be careful when the model contains multiple DCN layers in places other than backbone.

参数

- **optim_wrapper_cfg** (*dict*) –The config dict of the optimizer wrapper. Positional fields are
  - type: class name of the OptimizerWrapper
  - optimizer: The configuration of optimizer.

  Optional fields are
  - any arguments of the corresponding optimizer wrapper type, e.g., accumulative_counts, clip_grad, etc.

- **positional fields of optimizer are** (*The*) –
  - *type*: class name of the optimizer.
- **fields are** (*Optional*) –
  - any arguments of the corresponding optimizer type, e.g., lr, weight_decay, momentum, etc.
- **paramwise_cfg** (*dict, optional*) –Parameter-wise options.

Example 1:

```
>>> model = torch.nn.modules.Conv1d(1, 1, 1)
>>> optim_wrapper_cfg = dict(
>>>     dict(type='OptimWrapper', optimizer=dict(type='SGD', lr=0.01,
>>>         momentum=0.9, weight_decay=0.0001))
>>> paramwise_cfg = dict(norm_decay_mult=0.)
>>> optim_wrapper_builder = DefaultOptimWrapperConstructor(
>>>     optim_wrapper_cfg, paramwise_cfg)
>>> optim_wrapper = optim_wrapper_builder(model)
```

Example 2:

```
>>> # assume model have attribute model.backbone and model.cls_head
>>> optim_wrapper_cfg = dict(type='OptimWrapper', optimizer=dict(
>>>     type='SGD', lr=0.01, weight_decay=0.95))
>>> paramwise_cfg = dict(custom_keys={
>>>     '.backbone': dict(lr_mult=0.1, decay_mult=0.9)})
>>> optim_wrapper_builder = DefaultOptimWrapperConstructor(
>>>     optim_wrapper_cfg, paramwise_cfg)
>>> optim_wrapper = optim_wrapper_builder(model)
>>> # Then the `lr` and `weight_decay` for model.backbone is
>>> # (0.01 * 0.1, 0.95 * 0.9). `lr` and `weight_decay` for
>>> # model.cls_head is (0.01, 0.95).
```

add_params (*params*, *module*, *prefix=''*, *is_dcn_module=None*)

Add all parameters of module to the params list.

The parameters of the given module will be added to the list of param groups, with specific rules defined by paramwise_cfg.

参数

- **params** (*list[dict]*) –A list of param groups, it will be modified in place.
- **module** (*nn.Module*) –The module to be added.
- **prefix** (*str*) –The prefix of the module

- **is_dcn_module** (*int | float | None*) –If the current module is a submodule of DCN, *is_dcn_module* will be passed to control conv_offset layer's learning rate. Defaults to None.

返回类型 None

build_optim_wrapper	Build function of OptimWrapper.

41.1.5 mmengine.optim.build_optim_wrapper

mmengine.optim.**build_optim_wrapper**(*model*, *cfg*)

Build function of OptimWrapper.

If `constructor` is set in the `cfg`, this method will build an optimizer wrapper constructor, and use optimizer wrapper constructor to build the optimizer wrapper. If `constructor` is not set, the `DefaultOptimWrapperConstructor` will be used by default.

参数

- **model** (*nn.Module*) –Model to be optimized.
- **cfg** (*dict*) –Config of optimizer wrapper, optimizer constructor and optimizer.

返回 The built optimizer wrapper.

返回类型 *OptimWrapper*

41.2 Scheduler

_ParamScheduler	Base class for parameter schedulers.
ConstantLR	Decays the learning rate value of each parameter group by a small constant factor until the number of epoch reaches a pre-defined milestone: `end`.
ConstantMomentum	Decays the momentum value of each parameter group by a small constant factor until the number of epoch reaches a pre-defined milestone: `end`.
ConstantParamScheduler	Decays the parameter value of each parameter group by a small constant factor until the number of epoch reaches a pre-defined milestone: `end`.
CosineAnnealingLR	Set the learning rate of each parameter group using a cosine annealing schedule, where η_{max} is set to the initial value and T_{cur} is the number of epochs since the last restart in SGDR:

下页继续

表 3 – 续上页

CosineAnnealingMomentum	Set the momentum of each parameter group using a cosine annealing schedule, where η_{max} is set to the initial value and T_{cur} is the number of epochs since the last restart in SGDR:
CosineAnnealingParamScheduler	Set the parameter value of each parameter group using a cosine annealing schedule, where η_{max} is set to the initial value and T_{cur} is the number of epochs since the last restart in SGDR:
ExponentialLR	Decays the learning rate of each parameter group by gamma every epoch.
ExponentialMomentum	Decays the momentum of each parameter group by gamma every epoch.
ExponentialParamScheduler	Decays the parameter value of each parameter group by gamma every epoch.
LinearLR	Decays the learning rate of each parameter group by linearly changing small multiplicative factor until the number of epoch reaches a pre-defined milestone: `end`.
LinearMomentum	Decays the momentum of each parameter group by linearly changing small multiplicative factor until the number of epoch reaches a pre-defined milestone: `end`.
LinearParamScheduler	Decays the parameter value of each parameter group by linearly changing small multiplicative factor until the number of epoch reaches a pre-defined milestone: `end`.
MultiStepLR	Decays the specified learning rate in each parameter group by gamma once the number of epoch reaches one of the milestones.
MultiStepMomentum	Decays the specified momentum in each parameter group by gamma once the number of epoch reaches one of the milestones.
MultiStepParamScheduler	Decays the specified parameter in each parameter group by gamma once the number of epoch reaches one of the milestones.
OneCycleLR	Sets the learning rate of each parameter group according to the 1cycle learning rate policy.
OneCycleParamScheduler	Sets the parameters of each parameter group according to the 1cycle learning rate policy.
PolyLR	Decays the learning rate of each parameter group in a polynomial decay scheme.

下页继续

表 3 – 续上页

PolyMomentum	Decays the momentum of each parameter group in a polynomial decay scheme.
PolyParamScheduler	Decays the parameter value of each parameter group in a polynomial decay scheme.
StepLR	Decays the learning rate of each parameter group by gamma every step_size epochs.
StepMomentum	Decays the momentum of each parameter group by gamma every step_size epochs.
StepParamScheduler	Decays the parameter value of each parameter group by gamma every step_size epochs.

41.2.1 _ParamScheduler

class mmengine.optim.**_ParamScheduler**(*optimizer*, *param_name*, *begin=0*, *end=1000000000*, *last_step=- 1*, *by_epoch=True*, *verbose=False*)

Base class for parameter schedulers.

It should be inherited by all schedulers that schedule parameters in the optimizer's `param_groups`. All subclasses should overwrite the `_get_value()` according to their own schedule strategy. The implementation is motivated by https://github.com/pytorch/pytorch/blob/master/torch/optim/lr_scheduler.py.

参数

- **optimizer** (*OptimWrapper or Optimizer*) –Wrapped optimizer.
- **param_name** (*str*) –Name of the parameter to be adjusted, such as `lr`, `momentum`.
- **begin** (*int*) –Step at which to start updating the parameters. Defaults to 0.
- **end** (*int*) –Step at which to stop updating the parameters. Defaults to INF.
- **last_step** (*int*) –The index of last step. Used for resuming without state dict. Default value `-1` means the `step` function is never be called before. Defaults to -1.
- **by_epoch** (*bool*) –Whether the scheduled parameters are updated by epochs. Defaults to True.
- **verbose** (*bool*) –Whether to print the value for each update. Defaults to False.

get_last_value()

Return the last computed value by current scheduler.

返回 A list of the last computed value of the optimizer's `param_group`.

返回类型 list

load_state_dict(*state_dict*)

Loads the schedulers state.

参数 **state_dict** (*dict*) –scheduler state. Should be an object returned from a call to *state_dict()*.

print_value(*is_verbose*, *group*, *value*)

Display the current parameter value.

参数

- **is_verbose** (*bool*) –Whether to print the value.
- **group** (*int*) –The index of the current `param_group`.
- **value** (*float*) –The parameter value.

state_dict()

Returns the state of the scheduler as a `dict`.

It contains an entry for every variable in self.__dict__ which is not the optimizer.

返回 scheduler state.

返回类型 dict

step()

Adjusts the parameter value of each parameter group based on the specified schedule.

41.2.2 ConstantLR

class mmengine.optim.**ConstantLR**(*optimizer*, **args*, ***kwargs*)

Decays the learning rate value of each parameter group by a small constant factor until the number of epoch reaches a pre-defined milestone: `end`. Notice that such decay can happen simultaneously with other changes to the learning rate value from outside this scheduler.

参数

- **optimizer** (*Optimizer or OptimWrapper*) –Wrapped optimizer.
- **factor** (*float*) –The number we multiply learning rate until the milestone. Defaults to 1./3.
- **begin** (*int*) –Step at which to start updating the learning rate. Defaults to 0.
- **end** (*int*) –Step at which to stop updating the learning rate. Defaults to INF.
- **last_step** (*int*) –The index of last step. Used for resume without state dict. Defaults to -1.
- **by_epoch** (*bool*) –Whether the scheduled learning rate is updated by epochs. Defaults to True.
- **verbose** (*bool*) –Whether to print the learning rate for each update. Defaults to False.

41.2.3 ConstantMomentum

class mmengine.optim.**ConstantMomentum**(*optimizer*, **args*, ***kwargs*)

Decays the momentum value of each parameter group by a small constant factor until the number of epoch reaches a pre-defined milestone: `end`. Notice that such decay can happen simultaneously with other changes to the momentum value from outside this scheduler.

参数

- **optimizer** (*Optimizer or OptimWrapper*) –optimizer or Wrapped optimizer.
- **factor** (*float*) –The number we multiply momentum until the milestone. Defaults to 1./3.
- **begin** (*int*) –Step at which to start updating the momentum. Defaults to 0.
- **end** (*int*) –Step at which to stop updating the momentum. Defaults to INF.
- **last_step** (*int*) –The index of last step. Used for resume without state dict. Defaults to -1.
- **by_epoch** (*bool*) –Whether the scheduled momentum is updated by epochs. Defaults to True.
- **verbose** (*bool*) –Whether to print the momentum for each update. Defaults to False.

41.2.4 ConstantParamScheduler

class mmengine.optim.**ConstantParamScheduler**(*optimizer*, *param_name*, *factor=0.3333333333333333*, *begin=0*, *end=1000000000*, *last_step=- 1*, *by_epoch=True*, *verbose=False*)

Decays the parameter value of each parameter group by a small constant factor until the number of epoch reaches a pre-defined milestone: `end`. Notice that such decay can happen simultaneously with other changes to the parameter value from outside this scheduler.

参数

- **optimizer** (*Optimizer or OptimWrapper*) –optimizer or Wrapped optimizer.
- **param_name** (*str*) –Name of the parameter to be adjusted, such as `lr`, `momentum`.
- **factor** (*float*) –The number we multiply parameter value until the milestone. Defaults to 1./3.
- **begin** (*int*) –Step at which to start updating the parameters. Defaults to 0.
- **end** (*int*) –Step at which to stop updating the parameters. Defaults to INF.
- **last_step** (*int*) –The index of last step. Used for resume without state dict. Defaults to -1.

- **by_epoch** (*bool*) –Whether the scheduled parameters are updated by epochs. Defaults to True.
- **verbose** (*bool*) –Whether to print the value for each update. Defaults to False.

classmethod build_iter_from_epoch(**args*, *begin=0*, *end=1000000000*, *by_epoch=True*, *epoch_length=None*, ***kwargs*)

Build an iter-based instance of this scheduler from an epoch-based config.

41.2.5 CosineAnnealingLR

class mmengine.optim.**CosineAnnealingLR**(*optimizer*, **args*, ***kwargs*)

Set the learning rate of each parameter group using a cosine annealing schedule, where η_{max} is set to the initial value and T_{cur} is the number of epochs since the last restart in SGDR:

$$\eta_t = \eta_{min} + \frac{1}{2}(\eta_{max} - \eta_{min})\left(1 + \cos\left(\frac{T_{cur}}{T_{max}}\pi\right)\right), \quad T_{cur} \neq (2k+1)T_{max};$$
$$\eta_{t+1} = \eta_t + \frac{1}{2}(\eta_{max} - \eta_{min})\left(1 - \cos\left(\frac{1}{T_{max}}\pi\right)\right), \quad T_{cur} = (2k+1)T_{max}.$$

Notice that because the schedule is defined recursively, the learning rate can be simultaneously modified outside this scheduler by other operators. If the learning rate is set solely by this scheduler, the learning rate at each step becomes:

$$\eta_t = \eta_{min} + \frac{1}{2}(\eta_{max} - \eta_{min})\left(1 + \cos\left(\frac{T_{cur}}{T_{max}}\pi\right)\right)$$

It has been proposed in SGDR: Stochastic Gradient Descent with Warm Restarts. Note that this only implements the cosine annealing part of SGDR, and not the restarts.

参数

- **optimizer** (*Optimizer or OptimWrapper*) –Wrapped optimizer.
- **T_max** (*int*) –Maximum number of iterations.
- **eta_min** (*float*) –Minimum learning rate. Defaults to 0.
- **begin** (*int*) –Step at which to start updating the learning rate. Defaults to 0.
- **end** (*int*) –Step at which to stop updating the learning rate. Defaults to INF.
- **last_step** (*int*) –The index of last step. Used for resume without state dict. Defaults to -1.
- **by_epoch** (*bool*) –Whether the scheduled learning rate is updated by epochs. Defaults to True.
- **verbose** (*bool*) –Whether to print the learning rate for each update. Defaults to False.

41.2.6 CosineAnnealingMomentum

class mmengine.optim.**CosineAnnealingMomentum**(*optimizer*, **args*, ***kwargs*)

Set the momentum of each parameter group using a cosine annealing schedule, where η_{max} is set to the initial value and T_{cur} is the number of epochs since the last restart in SGDR:

$$\eta_t = \eta_{min} + \frac{1}{2}(\eta_{max} - \eta_{min})\left(1 + \cos\left(\frac{T_{cur}}{T_{max}}\pi\right)\right), \quad T_{cur} \neq (2k+1)T_{max};$$
$$\eta_{t+1} = \eta_t + \frac{1}{2}(\eta_{max} - \eta_{min})\left(1 - \cos\left(\frac{1}{T_{max}}\pi\right)\right), \qquad T_{cur} = (2k+1)T_{max}.$$

Notice that because the schedule is defined recursively, the momentum can be simultaneously modified outside this scheduler by other operators. If the momentum is set solely by this scheduler, the momentum at each step becomes:

$$\eta_t = \eta_{min} + \frac{1}{2}(\eta_{max} - \eta_{min})\left(1 + \cos\left(\frac{T_{cur}}{T_{max}}\pi\right)\right)$$

It has been proposed in SGDR: Stochastic Gradient Descent with Warm Restarts. Note that this only implements the cosine annealing part of SGDR, and not the restarts.

参数

- **optimizer** (*Optimizer or OptimWrapper*) –optimizer or Wrapped optimizer.
- **T_max** (*int*) –Maximum number of iterations.
- **eta_min** (*float*) –Minimum momentum value. Defaults to 0.
- **begin** (*int*) –Step at which to start updating the momentum. Defaults to 0.
- **end** (*int*) –Step at which to stop updating the momentum. Defaults to INF.
- **last_step** (*int*) –The index of last step. Used for resume without state dict. Defaults to -1.
- **by_epoch** (*bool*) –Whether the scheduled momentum is updated by epochs. Defaults to True.
- **verbose** (*bool*) –Whether to print the momentum for each update. Defaults to False.

41.2.7 CosineAnnealingParamScheduler

class mmengine.optim.**CosineAnnealingParamScheduler**(*optimizer*, *param_name*, *T_max=None*, *eta_min=0.0*, *begin=0*, *end=1000000000*, *last_step=- 1*, *by_epoch=True*, *verbose=False*)

Set the parameter value of each parameter group using a cosine annealing schedule, where η_{max} is set to the initial value and T_{cur} is the number of epochs since the last restart in SGDR:

$$\eta_t = \eta_{min} + \frac{1}{2}(\eta_{max} - \eta_{min})\left(1 + \cos\left(\frac{T_{cur}}{T_{max}}\pi\right)\right), \quad T_{cur} \neq (2k+1)T_{max};$$
$$\eta_{t+1} = \eta_t + \frac{1}{2}(\eta_{max} - \eta_{min})\left(1 - \cos\left(\frac{1}{T_{max}}\pi\right)\right), \qquad T_{cur} = (2k+1)T_{max}.$$

Notice that because the schedule is defined recursively, the parameter value can be simultaneously modified outside this scheduler by other operators. If the parameter value is set solely by this scheduler, the parameter value at each step becomes:

$$\eta_t = \eta_{min} + \frac{1}{2}(\eta_{max} - \eta_{min})\left(1 + \cos\left(\frac{T_{cur}}{T_{max}}\pi\right)\right)$$

It has been proposed in SGDR: Stochastic Gradient Descent with Warm Restarts. Note that this only implements the cosine annealing part of SGDR, and not the restarts.

参数

- **optimizer** (*Optimizer or OptimWrapper*) –optimizer or Wrapped optimizer.
- **param_name** (*str*) –Name of the parameter to be adjusted, such as `lr`, `momentum`.
- **T_max** (*int, optional*) –Maximum number of iterations. If not specified, use `end - begin`. Defaults to None.
- **eta_min** (*float*) –Minimum parameter value. Defaults to 0.
- **begin** (*int*) –Step at which to start updating the parameters. Defaults to 0.
- **end** (*int*) –Step at which to stop updating the parameters. Defaults to INF.
- **last_step** (*int*) –The index of last step. Used for resume without state dict. Defaults to -1.
- **by_epoch** (*bool*) –Whether the scheduled parameters are updated by epochs. Defaults to True.
- **verbose** (*bool*) –Whether to print the value for each update. Defaults to False.

classmethod build_iter_from_epoch(**args*, *T_max*, *begin=0*, *end=1000000000*, *by_epoch=True*, *epoch_length=None*, ***kwargs*)

Build an iter-based instance of this scheduler from an epoch-based config.

41.2.8 ExponentialLR

class mmengine.optim.**ExponentialLR**(*optimizer*, **args*, ***kwargs*)

Decays the learning rate of each parameter group by gamma every epoch.

参数

- **optimizer** (*Optimizer or OptimWrapper*) –Wrapped optimizer.
- **gamma** (*float*) –Multiplicative factor of learning rate decay.
- **begin** (*int*) –Step at which to start updating the learning rate. Defaults to 0.
- **end** (*int*) –Step at which to stop updating the learning rate. Defaults to INF.
- **last_step** (*int*) –The index of last step. Used for resume without state dict. Defaults to -1.

- **by_epoch** (*bool*) –Whether the scheduled learning rate is updated by epochs. Defaults to True.
- **verbose** (*bool*) –Whether to print the learning rate for each update. Defaults to False.

41.2.9 ExponentialMomentum

class mmengine.optim.**ExponentialMomentum**(*optimizer*, **args*, ***kwargs*)

Decays the momentum of each parameter group by gamma every epoch.

参数

- **optimizer** (*Optimizer or OptimWrapper*) –optimizer or Wrapped optimizer.
- **gamma** (*float*) –Multiplicative factor of momentum value decay.
- **begin** (*int*) –Step at which to start updating the momentum. Defaults to 0.
- **end** (*int*) –Step at which to stop updating the momentum. Defaults to INF.
- **last_step** (*int*) –The index of last step. Used for resume without state dict. Defaults to -1.
- **by_epoch** (*bool*) –Whether the scheduled momentum is updated by epochs. Defaults to True.
- **verbose** (*bool*) –Whether to print the momentum for each update. Defaults to False.

41.2.10 ExponentialParamScheduler

class mmengine.optim.**ExponentialParamScheduler**(*optimizer*, *param_name*, *gamma*, *begin=0*, *end=1000000000*, *last_step=- 1*, *by_epoch=True*, *verbose=False*)

Decays the parameter value of each parameter group by gamma every epoch.

参数

- **optimizer** (*Optimizer or OptimWrapper*) –optimizer or Wrapped optimizer.
- **param_name** (*str*) –Name of the parameter to be adjusted, such as `lr`, `momentum`.
- **gamma** (*float*) –Multiplicative factor of parameter value decay.
- **begin** (*int*) –Step at which to start updating the parameters. Defaults to 0.
- **end** (*int*) –Step at which to stop updating the parameters. Defaults to INF.
- **last_step** (*int*) –The index of last step. Used for resume without state dict. Defaults to -1.

- **by_epoch** (*bool*) –Whether the scheduled parameters are updated by epochs. Defaults to True.
- **verbose** (*bool*) –Whether to print the value for each update. Defaults to False.

classmethod build_iter_from_epoch(**args*, *begin=0*, *end=1000000000*, *by_epoch=True*, *epoch_length=None*, ***kwargs*)

Build an iter-based instance of this scheduler from an epoch-based config.

41.2.11 LinearLR

class mmengine.optim.**LinearLR**(*optimizer*, **args*, ***kwargs*)

Decays the learning rate of each parameter group by linearly changing small multiplicative factor until the number of epoch reaches a pre-defined milestone: `end`.

Notice that such decay can happen simultaneously with other changes to the learning rate from outside this scheduler. :param optimizer: Wrapped optimizer. :type optimizer: Optimizer or OptimWrapper :param start_factor: The number we multiply learning rate in the

first epoch. The multiplication factor changes towards end_factor in the following epochs. Defaults to 1./3.

参数

- **end_factor** (*float*) –The number we multiply learning rate at the end of linear changing process. Defaults to 1.0.
- **begin** (*int*) –Step at which to start updating the learning rate. Defaults to 0.
- **end** (*int*) –Step at which to stop updating the learning rate. Defaults to INF.
- **last_step** (*int*) –The index of last step. Used for resume without state dict. Defaults to -1.
- **by_epoch** (*bool*) –Whether the scheduled learning rate is updated by epochs. Defaults to True.
- **verbose** (*bool*) –Whether to print the learning rate for each update. Defaults to False.

41.2.12 LinearMomentum

class mmengine.optim.**LinearMomentum**(*optimizer*, **args*, ***kwargs*)

Decays the momentum of each parameter group by linearly changing small multiplicative factor until the number of epoch reaches a pre-defined milestone: `end`.

Notice that such decay can happen simultaneously with other changes to the momentum from outside this scheduler. :param optimizer: optimizer or Wrapped

optimizer.

参数

- **start_factor** (*float*) –The number we multiply momentum in the first epoch. The multiplication factor changes towards end_factor in the following epochs. Defaults to 1./3.
- **end_factor** (*float*) –The number we multiply momentum at the end of linear changing process. Defaults to 1.0.
- **begin** (*int*) –Step at which to start updating the momentum. Defaults to 0.
- **end** (*int*) –Step at which to stop updating the momentum. Defaults to INF.
- **last_step** (*int*) –The index of last step. Used for resume without state dict. Defaults to -1.
- **by_epoch** (*bool*) –Whether the scheduled momentum is updated by epochs. Defaults to True.
- **verbose** (*bool*) –Whether to print the momentum for each update. Defaults to False.

41.2.13 LinearParamScheduler

class mmengine.optim.**LinearParamScheduler**(*optimizer*, *param_name*, *start_factor=0.3333333333333333*, *end_factor=1.0*, *begin=0*, *end=1000000000*, *last_step=- 1*, *by_epoch=True*, *verbose=False*)

Decays the parameter value of each parameter group by linearly changing small multiplicative factor until the number of epoch reaches a pre-defined milestone: `end`.

Notice that such decay can happen simultaneously with other changes to the parameter value from outside this scheduler.

参数

- **optimizer** (*Optimizer or OptimWrapper*) –optimizer or Wrapped optimizer.
- **param_name** (*str*) –Name of the parameter to be adjusted, such as `lr`, `momentum`.
- **start_factor** (*float*) –The number we multiply parameter value in the first epoch. The multiplication factor changes towards end_factor in the following epochs. Defaults to 1./3.
- **end_factor** (*float*) –The number we multiply parameter value at the end of linear changing process. Defaults to 1.0.
- **begin** (*int*) –Step at which to start updating the parameters. Defaults to 0.
- **end** (*int*) –Step at which to stop updating the parameters. Defaults to INF.

- **last_step** (*int*) –The index of last step. Used for resume without state dict. Defaults to -1.
- **by_epoch** (*bool*) –Whether the scheduled parameters are updated by epochs. Defaults to True.
- **verbose** (*bool*) –Whether to print the value for each update. Defaults to False.

classmethod build_iter_from_epoch(**args*, *begin=0*, *end=1000000000*, *by_epoch=True*, *epoch_length=None*, ***kwargs*)

Build an iter-based instance of this scheduler from an epoch-based config.

41.2.14 MultiStepLR

class mmengine.optim.**MultiStepLR**(*optimizer*, **args*, ***kwargs*)

Decays the specified learning rate in each parameter group by gamma once the number of epoch reaches one of the milestones. Notice that such decay can happen simultaneously with other changes to the learning rate from outside this scheduler.

参数

- **optimizer** (*Optimizer or OptimWrapper*) –Wrapped optimizer.
- **milestones** (*list*) –List of epoch indices. Must be increasing.
- **gamma** (*float*) –Multiplicative factor of learning rate decay. Defaults to 0.1.
- **begin** (*int*) –Step at which to start updating the learning rate. Defaults to 0.
- **end** (*int*) –Step at which to stop updating the learning rate. Defaults to INF.
- **last_step** (*int*) –The index of last step. Used for resume without state dict. Defaults to -1.
- **by_epoch** (*bool*) –Whether the scheduled learning rate is updated by epochs. Defaults to True.
- **verbose** (*bool*) –Whether to print the learning rate for each update. Defaults to False.

41.2.15 MultiStepMomentum

class mmengine.optim.**MultiStepMomentum**(*optimizer*, **args*, ***kwargs*)

Decays the specified momentum in each parameter group by gamma once the number of epoch reaches one of the milestones. Notice that such decay can happen simultaneously with other changes to the momentum from outside this scheduler.

参数

- **optimizer** (*Optimizer or OptimWrapper*) –optimizer or Wrapped optimizer.

- **milestones** (*list*) –List of epoch indices. Must be increasing.
- **gamma** (*float*) –Multiplicative factor of momentum value decay. Defaults to 0.1.
- **begin** (*int*) –Step at which to start updating the momentum. Defaults to 0.
- **end** (*int*) –Step at which to stop updating the momentum. Defaults to INF.
- **last_step** (*int*) –The index of last step. Used for resume without state dict. Defaults to -1.
- **by_epoch** (*bool*) –Whether the scheduled momentum is updated by epochs. Defaults to True.
- **verbose** (*bool*) –Whether to print the momentum for each update. Defaults to False.

41.2.16 MultiStepParamScheduler

class mmengine.optim.**MultiStepParamScheduler**(*optimizer*, *param_name*, *milestones*, *gamma=0.1*, *last_step=- 1*, *begin=0*, *end=1000000000*, *by_epoch=True*, *verbose=False*)

Decays the specified parameter in each parameter group by gamma once the number of epoch reaches one of the milestones. Notice that such decay can happen simultaneously with other changes to the parameter from outside this scheduler.

参数

- **optimizer** (*OptimWrapper or Optimizer*) –Wrapped optimizer.
- **param_name** (*str*) –Name of the parameter to be adjusted, such as `lr`, `momentum`.
- **milestones** (*list*) –List of epoch indices. Must be increasing.
- **gamma** (*float*) –Multiplicative factor of parameter value decay. Defaults to 0.1.
- **begin** (*int*) –Step at which to start updating the parameters. Defaults to 0.
- **end** (*int*) –Step at which to stop updating the parameters. Defaults to INF.
- **last_step** (*int*) –The index of last step. Used for resume without state dict. Defaults to -1.
- **by_epoch** (*bool*) –Whether the scheduled parameters are updated by epochs. Defaults to True.
- **verbose** (*bool*) –Whether to print the value for each update. Defaults to False.

classmethod build_iter_from_epoch(**args*, *milestones*, *begin=0*, *end=1000000000*, *by_epoch=True*, *epoch_length=None*, ***kwargs*)

Build an iter-based instance of this scheduler from an epoch-based config.

41.2.17 OneCycleLR

class mmengine.optim.**OneCycleLR**(*optimizer*, **args*, ***kwargs*)

Sets the learning rate of each parameter group according to the 1cycle learning rate policy. The 1cycle policy anneals the learning rate from an initial learning rate to some maximum learning rate and then from that maximum learning rate to some minimum learning rate much lower than the initial learning rate. This policy was initially described in the paper Super-Convergence: Very Fast Training of Neural Networks Using Large Learning Rates.

The 1cycle learning rate policy changes the learning rate after every batch. *step* should be called after a batch has been used for training.

This scheduler is not chainable.

Note also that the total number of steps in the cycle can be determined in one of two ways (listed in order of precedence):

1. A value for total_steps is explicitly provided.
2. A number of epochs (epochs) and a number of steps per epoch (steps_per_epoch) are provided. In this case, the number of total steps is inferred by total_steps = epochs * steps_per_epoch

You must either provide a value for total_steps or provide a value for both epochs and steps_per_epoch.

The default behaviour of this scheduler follows the fastai implementation of 1cycle, which claims that "unpublished work has shown even better results by using only two phases". To mimic the behaviour of the original paper instead, set `three_phase=True`.

参数

- **optimizer** (*Optimizer*) –Wrapped optimizer.
- **eta_max** (*float or list*) –Upper parameter value boundaries in the cycle for each parameter group.
- **total_steps** (*int*) –The total number of steps in the cycle. Note that if a value is not provided here, then it must be inferred by providing a value for epochs and steps_per_epoch. Default to None.
- **pct_start** (*float*) –The percentage of the cycle (in number of steps) spent increasing the learning rate. Default to 0.3
- **anneal_strategy** (*str*) –{'cos', 'linear'} Specifies the annealing strategy: "cos" for cosine annealing, "linear" for linear annealing. Default to 'cos'
- **div_factor** (*float*) –Determines the initial learning rate via initial_param = eta_max/div_factor Default to 25
- **final_div_factor** (*float*) –Determines the minimum learning rate via eta_min = initial_param/final_div_factor Default to 1e4

- **three_phase** (*bool*) –If `True`, use a third phase of the schedule to annihilate the learning rate according to 'final_div_factor' instead of modifying the second phase (the first two phases will be symmetrical about the step indicated by 'pct_start').
- **last_step** (*int*) –The index of last step. Used for resume without state dict. Defaults to -1.
- **by_epoch** (*bool*) –Whether the scheduled parameters are updated by epochs. Defaults to True.
- **verbose** (*bool*) –Whether to print the value for each update. Defaults to False.

41.2.18 OneCycleParamScheduler

class mmengine.optim.**OneCycleParamScheduler**(*optimizer*, *param_name*, *eta_max=0*, *total_steps=None*, *pct_start=0.3*, *anneal_strategy='cos'*, *div_factor=25.0*, *final_div_factor=10000.0*, *three_phase=False*, *begin=0*, *end=1000000000*, *last_step=- 1*, *by_epoch=True*, *verbose=False*)

Sets the parameters of each parameter group according to the 1cycle learning rate policy. The 1cycle policy anneals the learning rate from an initial learning rate to some maximum learning rate and then from that maximum learning rate to some minimum learning rate much lower than the initial learning rate. This policy was initially described in the paper Super-Convergence: Very Fast Training of Neural Networks Using Large Learning Rates.

The 1cycle learning rate policy changes the learning rate after every batch. *step* should be called after a batch has been used for training.

This scheduler is not chainable.

Note also that the total number of steps in the cycle can be determined in one of two ways (listed in order of precedence):

1. A value for total_steps is explicitly provided.
2. If total_steps is not defined, begin and end of the ParamSchedul will works for it. In this case, the number of total steps is inferred by total_steps = end - begin

The default behaviour of this scheduler follows the fastai implementation of 1cycle, which claims that "unpublished work has shown even better results by using only two phases". To mimic the behaviour of the original paper instead, set `three_phase=True`.

参数

- **optimizer** (*Optimizer*) –Wrapped optimizer.
- **param_name** (*str*) –Name of the parameter to be adjusted, such as `lr`, `momentum`.

- **eta_max** (*float or list*) –Upper parameter value boundaries in the cycle for each parameter group.
- **total_steps** (*int*) –The total number of steps in the cycle. Note that if a value is not provided here, then it will be equal to `end - begin`. Default to None
- **pct_start** (*float*) –The percentage of the cycle (in number of steps) spent increasing the learning rate. Default to 0.3
- **anneal_strategy** (*str*) –{'cos', 'linear'} Specifies the annealing strategy: "cos" for cosine annealing, "linear" for linear annealing. Default to 'cos'
- **div_factor** (*float*) –Determines the initial learning rate via initial_param = eta_max/div_factor Default to 25
- **final_div_factor** (*float*) –Determines the minimum learning rate via eta_min = initial_param/final_div_factor Default to 1e4
- **three_phase** (*bool*) –If `True`, use a third phase of the schedule to annihilate the learning rate according to 'final_div_factor' instead of modifying the second phase (the first two phases will be symmetrical about the step indicated by 'pct_start').
- **last_step** (*int*) –The index of last step. Used for resume without state dict. Defaults to -1.
- **by_epoch** (*bool*) –Whether the scheduled parameters are updated by epochs. Defaults to True.
- **verbose** (*bool*) –Whether to print the value for each update. Defaults to False.
- **begin** (*int*) –
- **end** (*int*) –

classmethod build_iter_from_epoch(**args*, *begin=0*, *end=1000000000*, *total_steps=None*, *by_epoch=True*, *epoch_length=None*, ***kwargs*)

Build an iter-based instance of this scheduler from an epoch-based config.

41.2.19 PolyLR

class mmengine.optim.**PolyLR**(*optimizer*, **args*, ***kwargs*)

Decays the learning rate of each parameter group in a polynomial decay scheme.

Notice that such decay can happen simultaneously with other changes to the parameter value from outside this scheduler.

参数

- **optimizer** (*Optimizer or OptimWrapper*) –Wrapped optimizer.
- **eta_min** (*float*) –Minimum learning rate at the end of scheduling. Defaults to 0.

- **power** (*float*) –The power of the polynomial. Defaults to 1.0.
- **begin** (*int*) –Step at which to start updating the parameters. Defaults to 0.
- **end** (*int*) –Step at which to stop updating the parameters. Defaults to INF.
- **last_step** (*int*) –The index of last step. Used for resume without state dict. Defaults to -1.
- **by_epoch** (*bool*) –Whether the scheduled parameters are updated by epochs. Defaults to True.
- **verbose** (*bool*) –Whether to print the value for each update. Defaults to False.

41.2.20 PolyMomentum

class mmengine.optim.**PolyMomentum**(*optimizer*, **args*, ***kwargs*)

Decays the momentum of each parameter group in a polynomial decay scheme.

Notice that such decay can happen simultaneously with other changes to the parameter value from outside this scheduler.

参数

- **optimizer** (*Optimizer or OptimWrapper*) –optimizer or Wrapped optimizer.
- **eta_min** (*float*) –Minimum momentum at the end of scheduling. Defaults to 0.
- **power** (*float*) –The power of the polynomial. Defaults to 1.0.
- **begin** (*int*) –Step at which to start updating the parameters. Defaults to 0.
- **end** (*int*) –Step at which to stop updating the parameters. Defaults to INF.
- **last_step** (*int*) –The index of last step. Used for resume without state dict. Defaults to -1.
- **by_epoch** (*bool*) –Whether the scheduled parameters are updated by epochs. Defaults to True.
- **verbose** (*bool*) –Whether to print the value for each update. Defaults to False.

41.2.21 PolyParamScheduler

class mmengine.optim.**PolyParamScheduler**(*optimizer*, *param_name*, *eta_min=0*, *power=1.0*, *begin=0*, *end=1000000000*, *last_step=- 1*, *by_epoch=True*, *verbose=False*)

Decays the parameter value of each parameter group in a polynomial decay scheme.

Notice that such decay can happen simultaneously with other changes to the parameter value from outside this scheduler.

参数

- **optimizer** (*Optimizer or OptimWrapper*) –optimizer or Wrapped optimizer.
- **param_name** (*str*) –Name of the parameter to be adjusted, such as `lr`, `momentum`.
- **eta_min** (*float*) –Minimum parameter value at the end of scheduling. Defaults to 0.
- **power** (*float*) –The power of the polynomial. Defaults to 1.0.
- **begin** (*int*) –Step at which to start updating the parameters. Defaults to 0.
- **end** (*int*) –Step at which to stop updating the parameters. Defaults to INF.
- **last_step** (*int*) –The index of last step. Used for resume without state dict. Defaults to -1.
- **by_epoch** (*bool*) –Whether the scheduled parameters are updated by epochs. Defaults to True.
- **verbose** (*bool*) –Whether to print the value for each update. Defaults to False.

classmethod build_iter_from_epoch(**args*, *begin=0*, *end=1000000000*, *by_epoch=True*, *epoch_length=None*, ***kwargs*)

Build an iter-based instance of this scheduler from an epoch-based config.

41.2.22 StepLR

class mmengine.optim.**StepLR**(*optimizer*, **args*, ***kwargs*)

Decays the learning rate of each parameter group by gamma every step_size epochs. Notice that such decay can happen simultaneously with other changes to the learning rate from outside this scheduler.

参数

- **optimizer** (*Optimizer or OptimWrapper*) –Wrapped optimizer.
- **step_size** (*int*) –Period of learning rate decay.
- **gamma** (*float*) –Multiplicative factor of learning rate decay. Defaults to 0.1.
- **begin** (*int*) –Step at which to start updating the learning rate. Defaults to 0.
- **end** (*int*) –Step at which to stop updating the learning rate. Defaults to INF.
- **last_step** (*int*) –The index of last step. Used for resume without state dict. Defaults to -1.
- **by_epoch** (*bool*) –Whether the scheduled learning rate is updated by epochs. Defaults to True.
- **verbose** (*bool*) –Whether to print the learning rate for each update. Defaults to False.

41.2.23 StepMomentum

class mmengine.optim.**StepMomentum**(*optimizer*, **args*, ***kwargs*)

Decays the momentum of each parameter group by gamma every step_size epochs. Notice that such decay can happen simultaneously with other changes to the momentum from outside this scheduler.

参数

- **optimizer** (*Optimizer or OptimWrapper*) –optimizer or Wrapped optimizer.
- **step_size** (*int*) –Period of momentum value decay.
- **gamma** (*float*) –Multiplicative factor of momentum value decay. Defaults to 0.1.
- **begin** (*int*) –Step at which to start updating the momentum. Defaults to 0.
- **end** (*int*) –Step at which to stop updating the momentum. Defaults to INF.
- **last_step** (*int*) –The index of last step. Used for resume without state dict. Defaults to -1.
- **by_epoch** (*bool*) –Whether the scheduled momentum is updated by epochs. Defaults to True.
- **verbose** (*bool*) –Whether to print the momentum for each update. Defaults to False.

41.2.24 StepParamScheduler

class mmengine.optim.**StepParamScheduler**(*optimizer*, *param_name*, *step_size*, *gamma=0.1*, *begin=0*, *end=1000000000*, *last_step=- 1*, *by_epoch=True*, *verbose=False*)

Decays the parameter value of each parameter group by gamma every step_size epochs. Notice that such decay can happen simultaneously with other changes to the parameter value from outside this scheduler.

参数

- **optimizer** (*OptimWrapper or Optimizer*) –Wrapped optimizer.
- **param_name** (*str*) –Name of the parameter to be adjusted, such as `lr`, `momentum`.
- **step_size** (*int*) –Period of parameter value decay.
- **gamma** (*float*) –Multiplicative factor of parameter value decay. Defaults to 0.1.
- **begin** (*int*) –Step at which to start updating the parameters. Defaults to 0.
- **end** (*int*) –Step at which to stop updating the parameters. Defaults to INF.
- **last_step** (*int*) –The index of last step. Used for resume without state dict. Defaults to -1.

- **by_epoch** (*bool*) –Whether the scheduled parameters are updated by epochs. Defaults to True.
- **verbose** (*bool*) –Whether to print the value for each update. Defaults to False.

classmethod build_iter_from_epoch(**args*, *step_size*, *begin=0*, *end=1000000000*, *by_epoch=True*, *epoch_length=None*, ***kwargs*)

Build an iter-based instance of this scheduler from an epoch-based config.

CHAPTER 42

mmengine.evaluator

mmengine.evaluator

42.1 Evaluator

Evaluator	Wrapper class to compose multiple *BaseMetric* instances.

42.1.1 Evaluator

class mmengine.evaluator.**Evaluator**(*metrics*)

Wrapper class to compose multiple *BaseMetric* instances.

参数 **metrics**(*dict or BaseMetric or Sequence*) –The config of metrics.

property dataset_meta: Optional[dict]

Meta info of the dataset.

Type Optional[dict]

evaluate(*size*)

Invoke `evaluate` method of each metric and collect the metrics dictionary.

参数 **size** (*int*) –Length of the entire validation dataset. When batch size > 1, the dataloader may pad some data samples to make sure all ranks have the same length of dataset slice. The `collect_results` function will drop the padded data based on this size.

返回 Evaluation results of all metrics. The keys are the names of the metrics, and the values are corresponding results.

返回类型 dict

offline_evaluate(*data_samples*, *data=None*, *chunk_size=1*)

Offline evaluate the dumped predictions on the given data .

参数

- **data_samples** (*Sequence*) –All predictions and ground truth of the model and the validation set.
- **data** (*Sequence, optional*) –All data of the validation set.
- **chunk_size** (*int*) –The number of data samples and predictions to be processed in a batch.

process(*data_samples*, *data_batch=None*)

Convert `BaseDataSample` to dict and invoke process method of each metric.

参数

- **data_samples** (*Sequence[BaseDataElement]*) –predictions of the model, and the ground truth of the validation set.
- **data_batch** (*Any, optional*) –A batch of data from the dataloader.

42.2 Metric

BaseMetric	Base class for a metric.
DumpResults	Dump model predictions to a pickle file for offline evaluation.

42.2.1 BaseMetric

class mmengine.evaluator.**BaseMetric**(*collect_device='cpu'*, *prefix=None*)

Base class for a metric.

The metric first processes each batch of data_samples and predictions, and appends the processed results to the results list. Then it collects all results together from all ranks if distributed training is used. Finally, it computes the metrics of the entire dataset.

A subclass of class:*BaseMetric* should assign a meaningful value to the class attribute *default_prefix*. See the argument *prefix* for details.

参数

- **collect_device** (*str*) –Device name used for collecting results from different ranks during distributed training. Must be 'cpu' or 'gpu'. Defaults to 'cpu'.
- **prefix** (*str, optional*) –The prefix that will be added in the metric names to disambiguate homonymous metrics of different evaluators. If prefix is not provided in the argument, self.default_prefix will be used instead. Default: None

返回类型 None

abstract compute_metrics(*results*)

Compute the metrics from processed results.

参数 **results** (*list*) –The processed results of each batch.

返回 The computed metrics. The keys are the names of the metrics, and the values are corresponding results.

返回类型 dict

property dataset_meta: Optional[dict]

Meta info of the dataset.

Type Optional[dict]

evaluate(*size*)

Evaluate the model performance of the whole dataset after processing all batches.

参数 **size** (*int*) –Length of the entire validation dataset. When batch size > 1, the dataloader may pad some data samples to make sure all ranks have the same length of dataset slice. The `collect_results` function will drop the padded data based on this size.

返回 Evaluation metrics dict on the val dataset. The keys are the names of the metrics, and the values are corresponding results.

返回类型 dict

abstract process(*data_batch*, *data_samples*)

Process one batch of data samples and predictions. The processed results should be stored in `self.results`, which will be used to compute the metrics when all batches have been processed.

参数

- **data_batch** (*Any*) –A batch of data from the dataloader.
- **data_samples** (*Sequence[dict]*) –A batch of outputs from the model.

返回类型 None

42.2.2 DumpResults

class mmengine.evaluator.**DumpResults**(*out_file_path*, *collect_device='cpu'*)

Dump model predictions to a pickle file for offline evaluation.

参数

- **out_file_path** (*str*) –Path of the dumped file. Must end with '.pkl' or '.pickle'.
- **collect_device** (*str*) –Device name used for collecting results from different ranks during distributed training. Must be 'cpu' or 'gpu'. Defaults to 'cpu'.

返回类型 None

compute_metrics(*results*)

dump the prediction results to a pickle file.

参数 **results** (*list*) –

返回类型 dict

process(*data_batch*, *predictions*)

transfer tensors in predictions to CPU.

参数

- **data_batch** (*Any*) –
- **predictions** (*Sequence[dict]*) –

返回类型 None

42.3 Utils

get_metric_value	Get the metric value specified by an indicator, which can be either a metric name or a full name with evaluator prefix.

42.3.1 mmengine.evaluator.get_metric_value

mmengine.evaluator.**get_metric_value**(*indicator*, *metrics*)

Get the metric value specified by an indicator, which can be either a metric name or a full name with evaluator prefix.

参数

- **indicator** (*str*) –The metric indicator, which can be the metric name (e.g. 'AP') or the full name with prefix (e.g. 'COCO/AP')
- **metrics** (*dict*) –The evaluation results output by the evaluator

返回 The specified metric value

返回类型 Any

CHAPTER 43

mmengine.structures

BaseDataElement	A base data interface that supports Tensor-like and dict-like operations.
InstanceData	Data structure for instance-level annotations or predictions.
LabelData	Data structure for label-level annotations or predictions.
PixelData	Data structure for pixel-level annotations or predictions.

43.1 BaseDataElement

class mmengine.structures.**BaseDataElement**(*, *metainfo=None*, ***kwargs*)

A base data interface that supports Tensor-like and dict-like operations.

A typical data elements refer to predicted results or ground truth labels on a task, such as predicted bboxes, instance masks, semantic segmentation masks, etc. Because groundtruth labels and predicted results often have similar properties (for example, the predicted bboxes and the groundtruth bboxes), MMEngine uses the same abstract data interface to encapsulate predicted results and groundtruth labels, and it is recommended to use different name conventions to distinguish them, such as using `gt_instances` and `pred_instances` to distinguish between labels and predicted results. Additionally, we distinguish data elements at instance level, pixel level, and label level. Each of these types has its own characteristics. Therefore, MMEngine defines the base class `BaseDataElement`, and implement `InstanceData`, `PixelData`, and `LabelData` inheriting from `BaseDataElement` to represent different types of ground truth labels or predictions.

Another common data element is sample data. A sample data consists of input data (such as an image) and its annotations and predictions. In general, an image can have multiple types of annotations and/or predictions at the same time (for example, both pixel-level semantic segmentation annotations and instance-level detection bboxes annotations). All labels and predictions of a training sample are often passed between Dataset, Model, Visualizer, and Evaluator components. In order to simplify the interface between components, we can treat them as a large data element and encapsulate them. Such data elements are generally called XXDataSample in the OpenMMLab. Therefore, Similar to *nn.Module*, the *BaseDataElement* allows *BaseDataElement* as its attribute. Such a class generally encapsulates all the data of a sample in the algorithm library, and its attributes generally are various types of data elements. For example, MMDetection is assigned by the BaseDataElement to encapsulate all the data elements of the sample labeling and prediction of a sample in the algorithm library.

The attributes in `BaseDataElement` are divided into two parts, the `metainfo` and the `data` respectively.

- `metainfo`: Usually contains the information about the image such as filename, image_shape, pad_shape, etc. The attributes can be accessed or modified by dict-like or object-like operations, such as .``(for data access and modification) , ``in, del, pop(str), get(str), metainfo_keys(), metainfo_values(), metainfo_items(), "set_metainfo()"(for set or change key-value pairs in metainfo).
- `data`: Annotations or model predictions are stored. The attributes can be accessed or modified by dict-like or object-like operations, such as . , in, del, pop(str) get(str), keys(), values(), items(). Users can also apply tensor-like methods to all obj:`torch.Tensor` in the `data_fileds`, such as . cuda(), .cpu(), .numpy(), , .to() to_tensor(), .detach().

  参数

  - **metainfo** (*dict, optional*) –A dict contains the meta information of single image. such as `dict(img_shape=(512, 512, 3), scale_factor=(1, 1, 1, 1))`. Defaults to None.
  - **kwargs** (*dict, optional*) –A dict contains annotations of single image or model predictions. Defaults to None.

  返回类型 None

实际案例

```
>>> from mmengine.structures import BaseDataElement
>>> gt_instances = BaseDataElement()
>>> bboxes = torch.rand((5, 4))
>>> scores = torch.rand((5,))
>>> img_id = 0
>>> img_shape = (800, 1333)
>>> gt_instances = BaseDataElement(
...     metainfo=dict(img_id=img_id, img_shape=img_shape),
```

(下页继续)

(续上页)

```
...     bboxes=bboxes, scores=scores)
>>> gt_instances = BaseDataElement(
...                     metainfo=dict(img_id=img_id,
...                                   img_shape=(H, W)))
```

```
>>> # new
>>> gt_instances1 = gt_instance.new(
...                      metainfo=dict(img_id=1, img_shape=(640, 640)),
...                      bboxes=torch.rand((5, 4)),
...                      scores=torch.rand((5,)))
>>> gt_instances2 = gt_instances1.new()
```

```
>>> # add and process property
>>> gt_instances = BaseDataElement()
>>> gt_instances.set_metainfo(dict(img_id=9, img_shape=(100, 100))
>>> assert 'img_shape' in gt_instances.metainfo_keys()
>>> assert 'img_shape' in gt_instances
>>> assert 'img_shape' not in gt_instances.keys()
>>> assert 'img_shape' in gt_instances.all_keys()
>>> print(gt_instances.img_shape)
>>> gt_instances.scores = torch.rand((5,))
>>> assert 'scores' in gt_instances.keys()
>>> assert 'scores' in gt_instances
>>> assert 'scores' in gt_instances.all_keys()
>>> assert 'scores' not in gt_instances.metainfo_keys()
>>> print(gt_instances.scores)
>>> gt_instances.bboxes = torch.rand((5, 4))
>>> assert 'bboxes' in gt_instances.keys()
>>> assert 'bboxes' in gt_instances
>>> assert 'bboxes' in gt_instances.all_keys()
>>> assert 'bboxes' not in gt_instances.metainfo_keys()
>>> print(gt_instances.bboxes)
```

```
>>> # delete and change property
>>> gt_instances = BaseDataElement(
...  metainfo=dict(img_id=0, img_shape=(640, 640)),
...  bboxes=torch.rand((6, 4)), scores=torch.rand((6,)))
>>> gt_instances.img_shape = (1280, 1280)
>>> gt_instances.img_shape  # (1280, 1280)
>>> gt_instances.bboxes = gt_instances.bboxes * 2
>>> gt_instances.get('img_shape', None)  # (640, 640)
>>> gt_instances.get('bboxes', None)    # 6x4 tensor
```

(下页继续)

(续上页)

```
>>> del gt_instances.img_shape
>>> del gt_instances.bboxes
>>> assert 'img_shape' not in gt_instances
>>> assert 'bboxes' not in gt_instances
>>> gt_instances.pop('img_shape', None)  # None
>>> gt_instances.pop('bboxes', None)  # None
```

```
>>> # Tensor-like
>>> cuda_instances = gt_instances.cuda()
>>> cuda_instances = gt_instancess.to('cuda:0')
>>> cpu_instances = cuda_instances.cpu()
>>> cpu_instances = cuda_instances.to('cpu')
>>> fp16_instances = cuda_instances.to(
...  device=None, dtype=torch.float16, non_blocking=False, copy=False,
...  memory_format=torch.preserve_format)
>>> cpu_instances = cuda_instances.detach()
>>> np_instances = cpu_instances.numpy()
```

```
>>> # print
>>> metainfo = dict(img_shape=(800, 1196, 3))
>>> gt_instances = BaseDataElement(
>>>     metainfo=metainfo, det_labels=torch.LongTensor([0, 1, 2, 3]))
>>> sample = BaseDataElement(metainfo=metainfo,
...                          gt_instances=gt_instances)
>>> print(sample)
<BaseDataElement(
    META INFORMATION
    img_shape: (800, 1196, 3)
    DATA FIELDS
    gt_instances: <BaseDataElement(
            META INFORMATION
            img_shape: (800, 1196, 3)
            DATA FIELDS
            det_labels: tensor([0, 1, 2, 3])
        ) at 0x7f0ec5eadc70>
) at 0x7f0fea49e130>
```

```
>>> # inheritance
>>> class DetDataSample(BaseDataElement):
...     @property
...     def proposals(self):
...         return self._proposals
```

(下页继续)

(续上页)

```
...     @proposals.setter
...     def proposals(self, value):
...         self.set_field(value, '_proposals', dtype=BaseDataElement)
...     @proposals.deleter
...     def proposals(self):
...         del self._proposals
...     @property
...     def gt_instances(self):
...         return self._gt_instances
...     @gt_instances.setter
...     def gt_instances(self, value):
...         self.set_field(value, '_gt_instances',
...                        dtype=BaseDataElement)
...     @gt_instances.deleter
...     def gt_instances(self):
...         del self._gt_instances
...     @property
...     def pred_instances(self):
...         return self._pred_instances
...     @pred_instances.setter
...     def pred_instances(self, value):
...         self.set_field(value,'_pred_instances',
...                        dtype=BaseDataElement)
...     @pred_instances.deleter
...     def pred_instances(self):
...         del self._pred_instances
>>> det_sample = DetDataSample()
>>> proposals = BaseDataElement(bboxes=torch.rand((5, 4)))
>>> det_sample.proposals = proposals
>>> assert 'proposals' in det_sample
>>> assert det_sample.proposals == proposals
>>> del det_sample.proposals
>>> assert 'proposals' not in det_sample
>>> with self.assertRaises(AssertionError):
...     det_sample.proposals = torch.rand((5, 4))
```

all_items()

返回 an iterator object whose element is (key, value) tuple pairs for `metainfo` and `data`.

返回类型 iterator

all_keys()

返回 Contains all keys in metainfo and data.

返回类型 list

all_values()

返回 Contains all values in metainfo and data.

返回类型 list

clone()

Deep copy the current data element.

返回 the copy of current data element.

返回类型 *BaseDataElement*

cpu()

Convert all tensors to CPU in data.

返回类型 *mmengine.structures.base_data_element.BaseDataElement*

cuda()

Convert all tensors to GPU in data.

返回类型 *mmengine.structures.base_data_element.BaseDataElement*

detach()

Detach all tensors in data.

返回类型 *mmengine.structures.base_data_element.BaseDataElement*

get(*key*, *default=None*)

get property in data and metainfo as the same as python.

返回类型 Any

items()

返回 an iterator object whose element is (key, value) tuple pairs for `data`.

返回类型 iterator

keys()

返回 Contains all keys in data_fields.

返回类型 list

property metainfo: dict

A dict contains metainfo of current data element.

Type dict

metainfo_items()

返回 an iterator object whose element is (key, value) tuple pairs for `metainfo`.

返回类型 iterator

metainfo_keys()

返回 Contains all keys in metainfo_fields.

返回类型 list

metainfo_values()

返回 Contains all values in metainfo.

返回类型 list

new(*, *metainfo=None*, ***kwargs*)

Return a new data element with same type. If `metainfo` and `data` are None, the new data element will have same metainfo and data. If metainfo or data is not None, the new result will overwrite it with the input value.

参数

- **metainfo** (*dict, optional*) –A dict contains the meta information of image, such as `img_shape`, `scale_factor`, etc. Defaults to None.
- **kwargs** (*dict*) –A dict contains annotations of image or model predictions.

返回 a new data element with same type.

返回类型 *BaseDataElement*

numpy()

Convert all tensor to np.narray in data.

返回类型 *mmengine.structures.base_data_element.BaseDataElement*

pop(**args*)

pop property in data and metainfo as the same as python.

返回类型 Any

set_data(*data*)

Set or change key-value pairs in `data_field` by parameter `data`.

参数 **data** (*dict*) –A dict contains annotations of image or model predictions.

返回类型 None

set_field(*value*, *name*, *dtype=None*, *field_type='data'*)

Special method for set union field, used as property.setter functions.

参数

- **value** (*Any*) –
- **name** (*str*) –
- **dtype** (*Optional[Union[Type, Tuple[Type, ...]]]*) –
- **field_type** (*str*) –

返回类型 None

set_metainfo(*metainfo*)

Set or change key-value pairs in `metainfo_field` by parameter `metainfo`.

参数 **metainfo** (*dict*) –A dict contains the meta information of image, such as `img_shape`, `scale_factor`, etc.

返回类型 None

to(**args*, ***kwargs*)

Apply same name function to all tensors in data_fields.

返回类型 *mmengine.structures.base_data_element.BaseDataElement*

to_dict()

Convert BaseDataElement to dict.

返回类型 dict

to_tensor()

Convert all np.narray to tensor in data.

返回类型 *mmengine.structures.base_data_element.BaseDataElement*

update(*instance*)

The update() method updates the BaseDataElement with the elements from another BaseDataElement object.

参数

- **instance** (`BaseDataElement`) –Another BaseDataElement object for
- **the current object.** (*update*) –

返回类型 None

values()

返回 Contains all values in data.

返回类型 list

43.2 InstanceData

class mmengine.structures.**InstanceData**(*, *metainfo=None*, ***kwargs*)

Data structure for instance-level annotations or predictions.

Subclass of *BaseDataElement*. All value in *data_fields* should have the same length. This design refer to https://github.com/facebookresearch/detectron2/blob/master/detectron2/structures/instances.py # noqa E501 InstanceData also support extra functions: `index`, `slice` and `cat` for data field. The type of value in data field can be base data structure such as *torch.tensor*, *numpy.ndarray*, *list*, *str*, *tuple*, and can be customized data structure that has `__len__`, `__getitem__` and `cat` attributes.

实际案例

```
>>> # custom data structure
>>> class TmpObject:
...     def __init__(self, tmp) -> None:
...         assert isinstance(tmp, list)
...         self.tmp = tmp
...     def __len__(self):
...         return len(self.tmp)
...     def __getitem__(self, item):
...         if type(item) == int:
...             if item >= len(self) or item < -len(self):  # type:ignore
...                 raise IndexError(f'Index {item} out of range!')
...             else:
...                 # keep the dimension
...                 item = slice(item, None, len(self))
...         return TmpObject(self.tmp[item])
...     @staticmethod
...     def cat(tmp_objs):
...         assert all(isinstance(results, TmpObject) for results in tmp_objs)
...         if len(tmp_objs) == 1:
...             return tmp_objs[0]
...         tmp_list = [tmp_obj.tmp for tmp_obj in tmp_objs]
...         tmp_list = list(itertools.chain(*tmp_list))
...         new_data = TmpObject(tmp_list)
...         return new_data
...     def __repr__(self):
...         return str(self.tmp)
>>> from mmengine.structures import InstanceData
>>> import numpy as np
>>> img_meta = dict(img_shape=(800, 1196, 3), pad_shape=(800, 1216, 3))
>>> instance_data = InstanceData(metainfo=img_meta)
```

(下页继续)

(续上页)

```
>>> 'img_shape' in instance_data
True
>>> instance_data.det_labels = torch.LongTensor([2, 3])
>>> instance_data["det_scores"] = torch.Tensor([0.8, 0.7])
>>> instance_data.bboxes = torch.rand((2, 4))
>>> instance_data.polygons = TmpObject([[1, 2, 3, 4], [5, 6, 7, 8]])
>>> len(instance_data)
2
>>> print(instance_data)
<InstanceData(
    META INFORMATION
    pad_shape: (800, 1196, 3)
    img_shape: (800, 1216, 3)
    DATA FIELDS
    det_labels: tensor([2, 3])
    det_scores: tensor([0.8, 0.7000])
    bboxes: tensor([[0.4997, 0.7707, 0.0595, 0.4188],
        [0.8101, 0.3105, 0.5123, 0.6263]])
    polygons: [[1, 2, 3, 4], [5, 6, 7, 8]]
    ) at 0x7fb492de6280>
>>> sorted_results = instance_data[instance_data.det_scores.sort().indices]
>>> sorted_results.det_scores
tensor([0.7000, 0.8000])
>>> print(instance_data[instance_data.det_scores > 0.75])
<InstanceData(
    META INFORMATION
    pad_shape: (800, 1216, 3)
    img_shape: (800, 1196, 3)
    DATA FIELDS
    det_labels: tensor([2])
    masks: [[11, 21, 31, 41]]
    det_scores: tensor([0.8000])
    bboxes: tensor([[0.9308, 0.4000, 0.6077, 0.5554]])
    polygons: [[1, 2, 3, 4]]
) at 0x7f64ecf0ec40>
>>> print(instance_data[instance_data.det_scores > 1])
<InstanceData(
    META INFORMATION
    pad_shape: (800, 1216, 3)
    img_shape: (800, 1196, 3)
    DATA FIELDS
    det_labels: tensor([], dtype=torch.int64)
    masks: []
```

(下页继续)

(续上页)

```
    det_scores: tensor([])
    bboxes: tensor([], size=(0, 4))
    polygons: [[]]
) at 0x7f660a6a7f70>
>>> print(instance_data.cat([instance_data, instance_data]))
<InstanceData(
    META INFORMATION
    img_shape: (800, 1196, 3)
    pad_shape: (800, 1216, 3)
    DATA FIELDS
    det_labels: tensor([2, 3, 2, 3])
    bboxes: tensor([[0.7404, 0.6332, 0.1684, 0.9961],
                [0.2837, 0.8112, 0.5416, 0.2810],
                [0.7404, 0.6332, 0.1684, 0.9961],
                [0.2837, 0.8112, 0.5416, 0.2810]])
    data:
    polygons: [[1, 2, 3, 4], [5, 6, 7, 8],
               [1, 2, 3, 4], [5, 6, 7, 8]]
    det_scores: tensor([0.8000, 0.7000, 0.8000, 0.7000])
    masks: [[11, 21, 31, 41], [51, 61, 71, 81],
            [11, 21, 31, 41], [51, 61, 71, 81]]
) at 0x7f203542feb0>
```

参数 **metainfo**(*Optional[dict]*) –

返回类型 None

static cat(*instances_list*)

Concat the instances of all *InstanceData* in the list.

Note: To ensure that cat returns as expected, make sure that all elements in the list must have exactly the same keys.

参数 **instances_list** (list[*InstanceData*]) –A list of *InstanceData*.

返回 *InstanceData*

返回类型 obj

43.3 LabelData

class mmengine.structures.**LabelData**(*, *metainfo=None*, ***kwargs*)

Data structure for label-level annotations or predictions.

参数 **metainfo** (*Optional[dict]*) –

返回类型 None

static label_to_onehot(*label*, *num_classes*)

Convert the label-format input to one-hot.

参数

- **label** (*torch.Tensor*) –The label-format input. The format of item must be label-format.
- **num_classes** (*int*) –The number of classes.

返回 The converted results.

返回类型 torch.Tensor

static onehot_to_label(*onehot*)

Convert the one-hot input to label.

参数 **onehot** (*torch.Tensor, optional*) –The one-hot input. The format of input must be one-hot.

返回 The converted results.

返回类型 torch.Tensor

43.4 PixelData

class mmengine.structures.**PixelData**(*, *metainfo=None*, ***kwargs*)

Data structure for pixel-level annotations or predictions.

All data items in `data_fields` of `PixelData` meet the following requirements:

- They all have 3 dimensions in orders of channel, height, and width.
- They should have the same height and width.

实际案例

```
>>> metainfo = dict(
...     img_id=random.randint(0, 100),
...     img_shape=(random.randint(400, 600), random.randint(400, 600)))
>>> image = np.random.randint(0, 255, (4, 20, 40))
>>> featmap = torch.randint(0, 255, (10, 20, 40))
>>> pixel_data = PixelData(metainfo=metainfo,
...                        image=image,
...                        featmap=featmap)
>>> print(pixel_data)
>>> (20, 40)
```

```
>>> # slice
>>> slice_data = pixel_data[10:20, 20:40]
>>> assert slice_data.shape == (10, 10)
>>> slice_data = pixel_data[10, 20]
>>> assert slice_data.shape == (1, 1)
```

```
>>> # set
>>> pixel_data.map3 = torch.randint(0, 255, (20, 40))
>>> assert tuple(pixel_data.map3.shape) == (1, 20, 40)
>>> with self.assertRaises(AssertionError):
...     # The dimension must be 3 or 2
...     pixel_data.map2 = torch.randint(0, 255, (1, 3, 20, 40))
```

参数 **metainfo**(*Optional[dict]*) –

返回类型 None

property shape

The shape of pixel data.

CHAPTER 44

mmengine.dataset

mmengine.dataset

44.1 Dataset

`BaseDataset`	BaseDataset for open source projects in OpenMMLab.
`Compose`	Compose multiple transforms sequentially.

44.1.1 BaseDataset

class mmengine.dataset.**BaseDataset**(*ann_file=''*, *metainfo=None*, *data_root=''*, *data_prefix={'img_path': ''}*, *filter_cfg=None*, *indices=None*, *serialize_data=True*, *pipeline=[]*, *test_mode=False*, *lazy_init=False*, *max_refetch=1000*)

BaseDataset for open source projects in OpenMMLab.

The annotation format is shown as follows.

```
{
    "metainfo":
    {
      "dataset_type": "test_dataset",
      "task_name": "test_task"
    },
    "data_list":
    [
      {
        "img_path": "test_img.jpg",
        "height": 604,
        "width": 640,
        "instances":
        [
          {
            "bbox": [0, 0, 10, 20],
            "bbox_label": 1,
            "mask": [[0,0],[0,10],[10,20],[20,0]],
            "extra_anns": [1,2,3]
          },
          {
            "bbox": [10, 10, 110, 120],
            "bbox_label": 2,
            "mask": [[10,10],[10,110],[110,120],[120,10]],
            "extra_anns": [4,5,6]
          }
        ]
      },
    ]
}
```

参数

- **ann_file** (*str*) –Annotation file path. Defaults to ''.
- **metainfo** (*dict, optional*) –Meta information for dataset, such as class information.

Defaults to None.

- **data_root** (*str*) –The root directory for `data_prefix` and `ann_file`. Defaults to ''.
- **data_prefix** (*dict*) –Prefix for training data. Defaults to dict(img_path='').
- **filter_cfg** (*dict, optional*) –Config for filter data. Defaults to None.
- **indices** (*int or Sequence[int], optional*) –Support using first few data in annotation file to facilitate training/testing on a smaller dataset. Defaults to None which means using all `data_infos`.
- **serialize_data** (*bool, optional*) –Whether to hold memory using serialized objects, when enabled, data loader workers can use shared RAM from master process instead of making a copy. Defaults to True.
- **pipeline** (*list, optional*) –Processing pipeline. Defaults to [].
- **test_mode** (*bool, optional*) –`test_mode=True` means in test phase. Defaults to False.
- **lazy_init** (*bool, optional*) –Whether to load annotation during instantiation. In some cases, such as visualization, only the meta information of the dataset is needed, which is not necessary to load annotation file. `Basedataset` can skip load annotations to save time by set `lazy_init=False`. Defaults to False.
- **max_refetch** (*int, optional*) –If `Basedataset.prepare_data` get a None img. The maximum extra number of cycles to get a valid image. Defaults to 1000.

注解: BaseDataset collects meta information from *annotation file* (the lowest priority), "BaseDataset.METAINFO"(medium) and *metainfo parameter* (highest) passed to constructors. The lower priority meta information will be overwritten by higher one.

注解: Dataset wrapper such as `ConcatDataset`, `RepeatDataset` .etc. should not inherit from `BaseDataset` since `get_subset` and `get_subset_` could produce ambiguous meaning sub-dataset which conflicts with original dataset.

实际案例

Assume the annotation file is given above. »> class CustomDataset(BaseDataset): »> METAINFO: dict = dict(task_name='custom_task', »> dataset_type='custom_type') »> metainfo=dict(task_name='custom_task_name') »> custom_dataset = CustomDataset(»> 'path/to/ann_file', »> metainfo=metainfo) »> # meta information of annotation file will be overwritten by »> # *CustomDataset.METAINFO*. The merged meta information will »> # further be overwritten by argument *metainfo*. »> custom_dataset.metainfo { 'task_name': custom_task_name, dataset_type: custom_type}

filter_data()

Filter annotations according to filter_cfg. Defaults return all `data_list`.

If some `data_list` could be filtered according to specific logic, the subclass should override this method.

返回 Filtered results.

返回类型 list[int]

full_init()

Load annotation file and set `BaseDataset._fully_initialized` to True.

If `lazy_init=False`, `full_init` will be called during the instantiation and `self._fully_initialized` will be set to True. If `obj._fully_initialized=False`, the class method decorated by `force_full_init` will call `full_init` automatically.

Several steps to initialize annotation:

- load_data_list: Load annotations from annotation file.
- filter data information: Filter annotations according to filter_cfg.
- slice_data: Slice dataset according to `self._indices`
- serialize_data: Serialize `self.data_list` if

`self.serialize_data` is True.

get_cat_ids(*idx*)

Get category ids by index. Dataset wrapped by ClassBalancedDataset must implement this method.

The `ClassBalancedDataset` requires a subclass which implements this method.

参数 **idx** (*int*) –The index of data.

返回 All categories in the image of specified index.

返回类型 list[int]

get_data_info(*idx*)

Get annotation by index and automatically call `full_init` if the dataset has not been fully initialized.

参数 **idx** (*int*) –The index of data.

返回 The idx-th annotation of the dataset.

返回类型 dict

get_subset(*indices*)

Return a subset of dataset.

This method will return a subset of original dataset. If type of indices is int, `get_subset_` will return a subdataset which contains the first or last few data information according to indices is positive or negative. If type of indices is a sequence of int, the subdataset will extract the information according to the index given in indices.

实际案例

```
>>> dataset = BaseDataset('path/to/ann_file')
>>> len(dataset)
100
>>> subdataset = dataset.get_subset(90)
>>> len(sub_dataset)
90
>>> # if type of indices is list, extract the corresponding
>>> # index data information
>>> subdataset = dataset.get_subset([0, 1, 2, 3, 4, 5, 6, 7,
>>>                                  8, 9])
>>> len(sub_dataset)
10
>>> subdataset = dataset.get_subset(-3)
>>> len(subdataset) # Get the latest few data information.
3
```

参数 **indices** (*int or Sequence[int]*) –If type of indices is int, indices represents the first or last few data of dataset according to indices is positive or negative. If type of indices is Sequence, indices represents the target data information index of dataset.

返回 A subset of dataset.

返回类型 *BaseDataset*

get_subset_(*indices*)

The in-place version of "get_subset " to convert dataset to a subset of original dataset.

This method will convert the original dataset to a subset of dataset. If type of indices is int, `get_subset_` will return a subdataset which contains the first or last few data information according to indices is positive or negative. If type of indices is a sequence of int, the subdataset will extract the data information according to the index given in indices.

实际案例

```
>>> dataset = BaseDataset('path/to/ann_file')
>>> len(dataset)
100
>>> dataset.get_subset_(90)
>>> len(dataset)
90
>>> # if type of indices is sequence, extract the corresponding
>>> # index data information
>>> dataset.get_subset_([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> len(dataset)
10
>>> dataset.get_subset_(-3)
>>> len(dataset) # Get the latest few data information.
3
```

参数 **indices** (*int or Sequence[int]*) –If type of indices is int, indices represents the first or last few data of dataset according to indices is positive or negative. If type of indices is Sequence, indices represents the target data information index of dataset.

返回类型 None

load_data_list()

Load annotations from an annotation file named as `self.ann_file`

If the annotation file does not follow OpenMMLab 2.0 format dataset . The subclass must override this method for load annotations. The meta information of annotation file will be overwritten `METAINFO` and `metainfo` argument of constructor.

返回 A list of annotation.

返回类型 list[dict]

property metainfo: dict

Get meta information of dataset.

返回 meta information collected from `BaseDataset.METAINFO`, annotation file and metainfo argument during instantiation.

返回类型 dict

parse_data_info(*raw_data_info*)

Parse raw annotation to target format.

This method should return dict or list of dict. Each dict or list contains the data information of a training sample. If the protocol of the sample annotations is changed, this function can be overridden to update the parsing logic while keeping compatibility.

参数 **raw_data_info** (*dict*) –Raw data information load from `ann_file`

返回 Parsed annotation.

返回类型 list or list[dict]

prepare_data(*idx*)

Get data processed by `self.pipeline`.

参数 **idx** (*int*) –The index of `data_info`.

返回 Depends on `self.pipeline`.

返回类型 Any

44.1.2 Compose

class mmengine.dataset.**Compose**(*transforms*)

Compose multiple transforms sequentially.

参数 **transforms** (*Sequence[dict, callable], optional*) –Sequence of transform object or config dict to be composed.

44.2 Dataset Wrapper

ClassBalancedDataset	A wrapper of class balanced dataset.
ConcatDataset	A wrapper of concatenated dataset.
RepeatDataset	A wrapper of repeated dataset.

44.2.1 ClassBalancedDataset

class mmengine.dataset.**ClassBalancedDataset**(*dataset*, *oversample_thr*, *lazy_init=False*)

A wrapper of class balanced dataset.

Suitable for training on class imbalanced datasets like LVIS. Following the sampling strategy in the paper, in each epoch, an image may appear multiple times based on its "repeat factor". The repeat factor for an image is a function of the frequency the rarest category labeled in that image. The "frequency of category c" in [0, 1] is defined by the fraction of images in the training set (without repeats) in which category c appears. The dataset needs to instantiate *get_cat_ids()* to support ClassBalancedDataset.

The repeat factor is computed as followed.

1. For each category c, compute the fraction # of images that contain it: $f(c)$
2. For each category c, compute the category-level repeat factor: $r(c) = max(1, sqrt(t/f(c)))$

3. For each image I, compute the image-level repeat factor: $r(I) = max_{cinI} r(c)$

注解: `ClassBalancedDataset` should not inherit from `BaseDataset` since `get_subset` and `get_subset_` could produce ambiguous meaning sub-dataset which conflicts with original dataset. If you want to use a sub-dataset of `ClassBalancedDataset`, you should set `indices` arguments for wrapped dataset which inherit from `BaseDataset`.

参数

- **dataset** (*BaseDataset or dict*) –The dataset to be repeated.
- **oversample_thr** (*float*) –frequency threshold below which data is repeated. For categories with `f_c >= oversample_thr`, there is no oversampling. For categories with `f_c < oversample_thr`, the degree of oversampling following the square-root inverse frequency heuristic above.
- **lazy_init** (*bool, optional*) –whether to load annotation during instantiation. Defaults to False

full_init()

Loop to `full_init` each dataset.

get_cat_ids(*idx*)

Get category ids of class balanced dataset by index.

参数 **idx** (*int*) –Index of data.

返回 All categories in the image of specified index.

返回类型 List[int]

get_data_info(*idx*)

Get annotation by index.

参数 **idx** (*int*) –Global index of `ConcatDataset`.

返回 The idx-th annotation of the dataset.

返回类型 dict

get_subset(*indices*)

Not supported in `ClassBalancedDataset` for the ambiguous meaning of sub-dataset.

参数 **indices** (*Union[List[int], int]*) –

返回类型 *mmengine.dataset.base_dataset.BaseDataset*

get_subset_(*indices*)

Not supported in `ClassBalancedDataset` for the ambiguous meaning of sub-dataset.

参数 **indices** (*Union[List[int], int]*) –

返回类型 None

property metainfo: dict

Get the meta information of the repeated dataset.

返回 The meta information of repeated dataset.

返回类型 dict

44.2.2 ConcatDataset

class mmengine.dataset.**ConcatDataset**(*datasets*, *lazy_init=False*)

A wrapper of concatenated dataset.

Same as `torch.utils.data.dataset.ConcatDataset` and support lazy_init.

注解: `ConcatDataset` should not inherit from `BaseDataset` since `get_subset` and `get_subset_` could produce ambiguous meaning sub-dataset which conflicts with original dataset. If you want to use a sub-dataset of `ConcatDataset`, you should set `indices` arguments for wrapped dataset which inherit from `BaseDataset`.

参数

- **datasets** (*Sequence[BaseDataset] or Sequence[dict]*) –A list of datasets which will be concatenated.
- **lazy_init** (*bool, optional*) –Whether to load annotation during instantiation. Defaults to False.

full_init()

Loop to `full_init` each dataset.

get_data_info(*idx*)

Get annotation by index.

参数 **idx** (*int*) –Global index of `ConcatDataset`.

返回 The idx-th annotation of the datasets.

返回类型 dict

get_subset(*indices*)

Not supported in `ConcatDataset` for the ambiguous meaning of sub- dataset.

参数 **indices** (*Union[List[int], int]*) –

返回类型 *mmengine.dataset.base_dataset.BaseDataset*

get_subset_(*indices*)

Not supported in `ConcatDataset` for the ambiguous meaning of sub- dataset.

参数 **indices** (*Union[List[int], int]*) –

返回类型 None

property metainfo: dict

Get the meta information of the first dataset in `self.datasets`.

返回 Meta information of first dataset.

返回类型 dict

44.2.3 RepeatDataset

class mmengine.dataset.**RepeatDataset**(*dataset*, *times*, *lazy_init=False*)

A wrapper of repeated dataset.

The length of repeated dataset will be *times* larger than the original dataset. This is useful when the data loading time is long but the dataset is small. Using RepeatDataset can reduce the data loading time between epochs.

注解: `RepeatDataset` should not inherit from `BaseDataset` since `get_subset` and `get_subset_` could produce ambiguous meaning sub-dataset which conflicts with original dataset. If you want to use a sub-dataset of `RepeatDataset`, you should set `indices` arguments for wrapped dataset which inherit from `BaseDataset`.

参数

- **dataset** (`BaseDataset` *or* *dict*) –The dataset to be repeated.
- **times** (*int*) –Repeat times.
- **lazy_init** (*bool*) –Whether to load annotation during instantiation. Defaults to False.

full_init()

Loop to `full_init` each dataset.

get_data_info(*idx*)

Get annotation by index.

参数 **idx** (*int*) –Global index of `ConcatDataset`.

返回 The idx-th annotation of the datasets.

返回类型 dict

get_subset(*indices*)

Not supported in `RepeatDataset` for the ambiguous meaning of sub- dataset.

参数 **indices** (*Union[List[int], int]*) –

返回类型 *mmengine.dataset.base_dataset.BaseDataset*

get_subset_(*indices*)

Not supported in `RepeatDataset` for the ambiguous meaning of sub- dataset.

参数 **indices** (*Union[List[int], int]*) –

返回类型 None

property metainfo: dict

Get the meta information of the repeated dataset.

返回 The meta information of repeated dataset.

返回类型 dict

44.3 Sampler

DefaultSampler	The default data sampler for both distributed and non-distributed environment.
InfiniteSampler	It ' s designed for iteration-based runner and yields a mini-batch indices each time.

44.3.1 DefaultSampler

class mmengine.dataset.**DefaultSampler**(*dataset*, *shuffle=True*, *seed=None*, *round_up=True*)

The default data sampler for both distributed and non-distributed environment.

It has several differences from the PyTorch `DistributedSampler` as below:

1. This sampler supports non-distributed environment.
2. The round up behaviors are a little different.
   - If `round_up=True`, this sampler will add extra samples to make the number of samples is evenly divisible by the world size. And this behavior is the same as the `DistributedSampler` with `drop_last=False`.
   - If `round_up=False`, this sampler won' t remove or add any samples while the `DistributedSampler` with `drop_last=True` will remove tail samples.

参数

- **dataset** (*Sized*) –The dataset.
- **shuffle** (*bool*) –Whether shuffle the dataset or not. Defaults to True.

- **seed** (*int, optional*) –Random seed used to shuffle the sampler if `shuffle=True`. This number should be identical across all processes in the distributed group. Defaults to None.
- **round_up** (*bool*) –Whether to add extra samples to make the number of samples evenly divisible by the world size. Defaults to True.

set_epoch(*epoch*)

Sets the epoch for this sampler.

When `shuffle=True`, this ensures all replicas use a different random ordering for each epoch. Otherwise, the next iteration of this sampler will yield the same ordering.

参数 **epoch** (*int*) –Epoch number.

返回类型 None

44.3.2 InfiniteSampler

class mmengine.dataset.**InfiniteSampler**(*dataset*, *shuffle=True*, *seed=None*)

It’s designed for iteration-based runner and yields a mini-batch indices each time.

The implementation logic is referred to https://github.com/facebookresearch/detectron2/blob/main/detectron2/data/samplers/distributed_sampler.py

参数

- **dataset** (*Sized*) –The dataset.
- **shuffle** (*bool*) –Whether shuffle the dataset or not. Defaults to True.
- **seed** (*int, optional*) –Random seed. If None, set a random seed. Defaults to None.

set_epoch(*epoch*)

Not supported in iteration-based runner.

参数 **epoch** (*int*) –

返回类型 None

44.4 Utils

default_collate	Convert list of data sampled from dataset into a batch of data, of which type consistent with the type of each data_itement in `data_batch`.

下页继续

表 4 – 续上页

pseudo_collate	Convert list of data sampled from dataset into a batch of data, of which type consistent with the type of each data_itement in data_batch.
worker_init_fn	This function will be called on each worker subprocess after seeding and before data loading.

44.4.1 mmengine.dataset.default_collate

mmengine.dataset.**default_collate**(*data_batch*)

Convert list of data sampled from dataset into a batch of data, of which type consistent with the type of each data_itement in data_batch.

Different from *pseudo_collate()*, default_collate will stack tensor contained in data_batch into a batched tensor with the first dimension batch size, and then move input tensor to the target device.

Different from default_collate in pytorch, default_collate will not process BaseDataElement.

This code is referenced from: Pytorch default_collate. # noqa: E501

注解: default_collate only accept input tensor with the same shape.

参数 **data_batch** (*Sequence*) –Data sampled from dataset.

返回 Data in the same format as the data_itement of data_batch, of which tensors have been stacked, and ndarray, int, float have been converted to tensors.

返回类型 Any

44.4.2 mmengine.dataset.pseudo_collate

mmengine.dataset.**pseudo_collate**(*data_batch*)

Convert list of data sampled from dataset into a batch of data, of which type consistent with the type of each data_itement in data_batch.

The default behavior of dataloader is to merge a list of samples to form a mini-batch of Tensor(s). However, in MMEngine, pseudo_collate will not stack tensors to batch tensors, and convert int, float, ndarray to tensors.

This code is referenced from: Pytorch default_collate. # noqa: E501 :param data_batch: Batch of data from dataloader. :type data_batch: Sequence

返回 Transversed Data in the same format as the data_itement of data_batch.

返回类型 Any

参数 **data_batch** (*Sequence*) –

44.4.3 mmengine.dataset.worker_init_fn

mmengine.dataset.**worker_init_fn**(*worker_id*, *num_workers*, *rank*, *seed*)

This function will be called on each worker subprocess after seeding and before data loading.

参数

- **worker_id** (*int*) –Worker id in [0, num_workers - 1].
- **num_workers** (*int*) –How many subprocesses to use for data loading.
- **rank** (*int*) –Rank of process in distributed environment. If in non-distributed environment, it is a constant number *0*.
- **seed** (*int*) –Random seed.

返回类型 None

CHAPTER 45

mmengine.device

get_device	Returns the currently existing device type.
get_max_cuda_memory	Returns the maximum GPU memory occupied by tensors in megabytes (MB) for a given device.
is_cuda_available	Returns True if cuda devices exist.
is_mlu_available	Returns True if Cambricon PyTorch and mlu devices exist.
is_mps_available	Return True if mps devices exist.

45.1 mmengine.device.get_device

mmengine.device.**get_device**()

Returns the currently existing device type.

返回 cuda | mlu | mps | cpu.

返回类型 str

45.2 mmengine.device.get_max_cuda_memory

mmengine.device.**get_max_cuda_memory**(*device=None*)

Returns the maximum GPU memory occupied by tensors in megabytes (MB) for a given device. By default, this returns the peak allocated memory since the beginning of this program.

参数 **device** (*torch.device, optional*) –selected device. Returns statistic for the current device, given by `current_device()`, if `device` is None. Defaults to None.

返回 The maximum GPU memory occupied by tensors in megabytes for a given device.

返回类型 int

45.3 mmengine.device.is_cuda_available

mmengine.device.**is_cuda_available**()

Returns True if cuda devices exist.

返回类型 bool

45.4 mmengine.device.is_mlu_available

mmengine.device.**is_mlu_available**()

Returns True if Cambricon PyTorch and mlu devices exist.

返回类型 bool

45.5 mmengine.device.is_mps_available

mmengine.device.**is_mps_available**()

Return True if mps devices exist.

It’s specialized for mac m1 chips and require torch version 1.12 or higher.

返回类型 bool

CHAPTER 46

mmengine.hub

get_config	Get config from external package.
get_model	Get built model from external package.

46.1 mmengine.hub.get_config

mmengine.hub.**get_config**(*cfg_path*, *pretrained=False*)

Get config from external package.

参数

- **cfg_path** (*str*) –External relative config path.
- **pretrained** (*bool*) –Whether to save pretrained model path. If `pretrained==True`, the url of pretrained model can be accessed by `cfg.model_path`. Defaults to False.

返回类型 *mmengine.config.config.Config*

实际案例

```
>>> cfg = get_config('mmdet::faster_rcnn/faster_rcnn_r50_fpn_1x_coco.py',␣
↪pretrained=True)
>>> # Equivalent to
>>> # cfg = Config.fromfile('/path/to/faster_rcnn_r50_fpn_1x_coco.py')
>>> cfg.model_path
https://download.openmmlab.com/mmdetection/v2.0/faster_rcnn/faster_rcnn_r50_fpn_
↪1x_coco/faster_rcnn_r50_fpn_1x_coco_20200130-047c8118.pth
```

返回 A *Config* parsed from external package.

返回类型 *Config*

参数

- **cfg_path** (*str*) –
- **pretrained** (*bool*) –

46.2 mmengine.hub.get_model

mmengine.hub.**get_model**(*cfg_path*, *pretrained=False*, ***kwargs*)

Get built model from external package.

参数

- **cfg_path** (*str*) –External relative config path with prefix ‘package::’ and without suffix.
- **pretrained** (*bool*) –Whether to load pretrained model. Defaults to False.
- **kwargs** (*dict*) –Default arguments to build model.

实际案例

```
>>> model = get_model('mmdet::faster_rcnn/faster-rcnn_r50_fpn_1x_coco.py',␣
↪pretrained=True)
>>> type(model)
<class 'mmdet.models.detectors.faster_rcnn.FasterRCNN'>
```

返回 Built model.

返回类型 nn.Module

参数

- **cfg_path** (*str*) –

- **pretrained** (*bool*) –

CHAPTER 47

mmengine.logging

MMLogger	Formatted logger used to record messages.
MessageHub	Message hub for component interaction.
HistoryBuffer	Unified storage format for different log types.

47.1 MMLogger

class mmengine.logging.**MMLogger**(*name*, *logger_name='mmengine'*, *log_file=None*, *log_level='INFO'*, *file_mode='w'*, *distributed=False*)

Formatted logger used to record messages.

`MMLogger` can create formatted logger to log message with different log levels and get instance in the same way as `ManagerMixin`. `MMLogger` has the following features:

- Distributed log storage, `MMLogger` can choose whether to save log of different ranks according to *log_file*.
- Message with different log levels will have different colors and format when displayed on terminal.

注解:

- The *name* of logger and the `instance_name` of `MMLogger` could be different. We can only get `MMLogger` instance by `MMLogger.get_instance` but not `logging.getLogger`. This feature ensures `MMLogger` will not be incluenced by third-party logging config.
- Different from `logging.Logger`, `MMLogger` will not log warning or error message without `Handler`.

实际案例

```
>>> logger = MMLogger.get_instance(name='MMLogger',
>>>                                logger_name='Logger')
>>> # Although logger has name attribute just like `logging.Logger`
>>> # We cannot get logger instance by `logging.getLogger`.
>>> assert logger.name == 'Logger'
>>> assert logger.instance_name = 'MMLogger'
>>> assert id(logger) != id(logging.getLogger('Logger'))
>>> # Get logger that do not store logs.
>>> logger1 = MMLogger.get_instance('logger1')
>>> # Get logger only save rank0 logs.
>>> logger2 = MMLogger.get_instance('logger2', log_file='out.log')
>>> # Get logger only save multiple ranks logs.
>>> logger3 = MMLogger.get_instance('logger3', log_file='out.log',
>>>                                 distributed=True)
```

参数

- **name** (*str*) –Global instance name.
- **logger_name** (*str*) –`name` attribute of `Logging.Logger` instance. If *logger_name* is not defined, defaults to ‘mmengine’.
- **log_file** (*str, optional*) –The log filename. If specified, a `FileHandler` will be added to the logger. Defaults to None.
- **log_level** (*str*) –The log level of the handler and logger. Defaults to “NOTSET”.
- **file_mode** (*str*) –The file mode used to open log file. Defaults to ‘w’.
- **distributed** (*bool*) –Whether to save distributed logs, Defaults to false.

callHandlers(*record*)

Pass a record to all relevant handlers.

Override `callHandlers` method in `logging.Logger` to avoid multiple warning messages in DDP mode. Loop through all handlers of the logger instance and its parents in the logger hierarchy. If no handler was found, the record will not be output.

参数 **record** (*LogRecord*) –A `LogRecord` instance contains logged message.

返回类型 None

classmethod get_current_instance()

Get latest created `MMLogger` instance.

MMLogger can call *get_current_instance()* before any instance has been created, and return a logger with the instance name "mmengine".

返回 Configured logger instance.

返回类型 *MMLogger*

setLevel(*level*)

Set the logging level of this logger.

If `logging.Logger.selLevel` is called, all `logging.Logger` instances managed by `logging.Manager` will clear the cache. Since `MMLogger` is not managed by `logging.Manager` anymore, `MMLogger` should override this method to clear caches of all `MMLogger` instance which is managed by `ManagerMixin`.

level must be an int or a str.

47.2 MessageHub

class mmengine.logging.**MessageHub**(*name*, *log_scalars=None*, *runtime_info=None*, *resumed_keys=None*)

Message hub for component interaction. MessageHub is created and accessed in the same way as ManagerMixin.

`MessageHub` will record log information and runtime information. The log information refers to the learning rate, loss, etc. of the model during training phase, which will be stored as `HistoryBuffer`. The runtime information refers to the iter times, meta information of runner etc., which will be overwritten by next update.

参数

- **name** (*str*) –Name of message hub used to get corresponding instance globally.
- **log_scalars** (*OrderedDict, optional*) –Each key-value pair in the dictionary is the name of the log information such as "loss", "lr", "metric" and their corresponding values. The type of value must be HistoryBuffer. Defaults to None.
- **runtime_info** (*OrderedDict, optional*) –Each key-value pair in the dictionary is the name of the runtime information and their corresponding values. Defaults to None.
- **resumed_keys** (*OrderedDict, optional*) –Each key-value pair in the dictionary decides whether the key in `_log_scalars` and `_runtime_info` will be serialized.

注解： Key in `_resumed_keys` belongs to `_log_scalars` or `_runtime_info`. The corresponding value cannot be set repeatedly.

实际案例

```
>>> # create empty `MessageHub`.
>>> message_hub1 = MessageHub()
>>> log_scalars = OrderedDict(loss=HistoryBuffer())
>>> runtime_info = OrderedDict(task='task')
>>> resumed_keys = dict(loss=True)
>>> # create `MessageHub` from data.
>>> message_hub2 = MessageHub(
>>>     name='name',
>>>     log_scalars=log_scalars,
>>>     runtime_info=runtime_info,
>>>     resumed_keys=resumed_keys)
```

classmethod get_current_instance()

Get latest created `MessageHub` instance.

MessageHub can call *get_current_instance()* before any instance has been created, and return a message hub with the instance name "mmengine".

返回 Empty `MessageHub` instance.

返回类型 *MessageHub*

get_info(*key*)

Get runtime information by key.

参数 **key** (*str*) – Key of runtime information.

返回 A copy of corresponding runtime information if the key exists.

返回类型 Any

get_scalar(*key*)

Get `HistoryBuffer` instance by key.

注解: Considering the large memory footprint of history buffers in the post-training, *get_scalar()* will not return a reference of history buffer rather than a copy.

参数 **key** (*str*) – Key of `HistoryBuffer`.

返回 Corresponding `HistoryBuffer` instance if the key exists.

返回类型 *HistoryBuffer*

load_state_dict(*state_dict*)

Loads log scalars, runtime information and resumed keys from `state_dict` or `message_hub`.

If `state_dict` is a dictionary returned by *`state_dict()`*, it will only make copies of data which should be resumed from the source `message_hub`.

If `state_dict` is a `message_hub` instance, it will make copies of all data from the source message_hub. We suggest to load data from `dict` rather than a `MessageHub` instance.

参数 **state_dict** (*dict or MessageHub*) –A dictionary contains key `log_scalars` `runtime_info` and `resumed_keys`, or a MessageHub instance.

返回类型 None

property log_scalars: collections.OrderedDict

Get all `HistoryBuffer` instances.

注解：Considering the large memory footprint of history buffers in the post-training, *`get_scalar()`* will return a reference of history buffer rather than a copy.

返回 All `HistoryBuffer` instances.

返回类型 OrderedDict

property runtime_info: collections.OrderedDict

Get all runtime information.

返回 A copy of all runtime information.

返回类型 OrderedDict

state_dict()

Returns a dictionary containing log scalars, runtime information and resumed keys, which should be resumed.

The returned `state_dict` can be loaded by *`load_state_dict()`*.

返回 A dictionary contains `log_scalars`, `runtime_info` and `resumed_keys`.

返回类型 dict

update_info(*key*, *value*, *resumed=True*)

Update runtime information.

The key corresponding runtime information will be overwritten each time calling `update_info`.

注解：The `resumed` argument needs to be consistent for the same `key`.

实际案例

```
>>> message_hub = MessageHub()
>>> message_hub.update_info('iter', 100)
```

参数

- **key** (*str*) –Key of runtime information.
- **value** (*Any*) –Value of runtime information.
- **resumed** (*bool*) –Whether the corresponding `HistoryBuffer` could be resumed.

返回类型 None

update_info_dict(*info_dict*, *resumed=True*)

Update runtime information with dictionary.

The key corresponding runtime information will be overwritten each time calling `update_info`.

注解： The `resumed` argument needs to be consistent for the same `info_dict`.

实际案例

```
>>> message_hub = MessageHub()
>>> message_hub.update_info({'iter': 100})
```

参数

- **info_dict** (*str*) –Runtime information dictionary.
- **resumed** (*bool*) –Whether the corresponding `HistoryBuffer` could be resumed.

返回类型 None

update_scalar(*key*, *value*, *count=1*, *resumed=True*)

Update :attr:_log_scalars.

Update `HistoryBuffer` in `_log_scalars`. If corresponding key `HistoryBuffer` has been created, `value` and `count` is the argument of `HistoryBuffer.update`, Otherwise, `update_scalar` will create an `HistoryBuffer` with value and count via the constructor of `HistoryBuffer`.

实际案例

```
>>> message_hub = MessageHub
>>> # create loss `HistoryBuffer` with value=1, count=1
>>> message_hub.update_scalar('loss', 1)
>>> # update loss `HistoryBuffer` with value
>>> message_hub.update_scalar('loss', 3)
>>> message_hub.update_scalar('loss', 3, resumed=False)
AssertionError: loss used to be true, but got false now. resumed
keys cannot be modified repeatedly'
```

注解： The `resumed` argument needs to be consistent for the same `key`.

参数

- **key** (*str*) –Key of `HistoryBuffer`.
- **value** (*torch.Tensor or np.ndarray or int or float*) –Value of log.
- **count** (*torch.Tensor or np.ndarray or int or float*) –Accumulation times of log, defaults to 1. *count* will be used in smooth statistics.
- **resumed** (*str*) –Whether the corresponding `HistoryBuffer` could be resumed. Defaults to True.

返回类型 None

update_scalars(*log_dict*, *resumed=True*)

Update `_log_scalars` with a dict.

`update_scalars` iterates through each pair of log_dict key-value, and calls `update_scalar`. If type of value is dict, the value should be `dict(value=xxx) or dict(value=xxx, count=xxx)`. Item in `log_dict` has the same resume option.

注解： The `resumed` argument needs to be consistent for the same `log_dict`.

参数

- **log_dict** (*str*) –Used for batch updating `_log_scalars`.
- **resumed** (*bool*) –Whether all `HistoryBuffer` referred in log_dict should be resumed. Defaults to True.

返回类型 None

实际案例

```
>>> message_hub = MessageHub.get_instance('mmengine')
>>> log_dict = dict(a=1, b=2, c=3)
>>> message_hub.update_scalars(log_dict)
>>> # The default count of  `a`, `b` and `c` is 1.
>>> log_dict = dict(a=1, b=2, c=dict(value=1, count=2))
>>> message_hub.update_scalars(log_dict)
>>> # The count of `c` is 2.
```

47.3 HistoryBuffer

class mmengine.logging.**HistoryBuffer**(*log_history=[]*, *count_history=[]*, *max_length=1000000*)

Unified storage format for different log types.

`HistoryBuffer` records the history of log for further statistics.

实际案例

```
>>> history_buffer = HistoryBuffer()
>>> # Update history_buffer.
>>> history_buffer.update(1)
>>> history_buffer.update(2)
>>> history_buffer.min()  # minimum of (1, 2)
1
>>> history_buffer.max()  # maximum of (1, 2)
2
>>> history_buffer.mean()  # mean of (1, 2)
1.5
>>> history_buffer.statistics('mean')  # access method by string.
1.5
```

参数

- **log_history** (*Sequence*) –History logs. Defaults to [].
- **count_history** (*Sequence*) –Counts of history logs. Defaults to [].
- **max_length** (*int*) –The max length of history logs. Defaults to 1000000.

current()

Return the recently updated values in log histories.

返回 Recently updated values in log histories.

返回类型 np.ndarray

property data: Tuple[numpy.ndarray, numpy.ndarray]

Get the `_log_history` and `_count_history`.

返回 History logs and the counts of the history logs.

返回类型 Tuple[np.ndarray, np.ndarray]

max(*window_size=None*)

Return the maximum value of the latest `window_size` values in log histories.

If `window_size is None` or `window_size > len(self._log_history)`, return the global maximum value of history logs.

参数 **window_size**(*int, optional*) –Size of statistics window.

返回 The maximum value within the window.

返回类型 np.ndarray

mean(*window_size=None*)

Return the mean of the latest `window_size` values in log histories.

If `window_size is None` or `window_size > len(self._log_history)`, return the global mean value of history logs.

参数 **window_size**(*int, optional*) –Size of statistics window.

返回 Mean value within the window.

返回类型 np.ndarray

min(*window_size=None*)

Return the minimum value of the latest `window_size` values in log histories.

If `window_size is None` or `window_size > len(self._log_history)`, return the global minimum value of history logs.

参数 **window_size**(*int, optional*) –Size of statistics window.

返回 The minimum value within the window.

返回类型 np.ndarray

classmethod register_statistics(*method*)

Register custom statistics method to `_statistics_methods`.

The registered method can be called by `history_buffer.statistics` with corresponding method name and arguments.

实际案例

```
>>> @HistoryBuffer.register_statistics
>>> def weighted_mean(self, window_size, weight):
>>>     assert len(weight) == window_size
>>>     return (self._log_history[-window_size:] *
>>>             np.array(weight)).sum() /             >>>             self._
↪count_history[-window_size:]
```

```
>>> log_buffer = HistoryBuffer([1, 2], [1, 1])
>>> log_buffer.statistics('weighted_mean', 2, [2, 1])
2
```

参数 **method** (*Callable*) –Custom statistics method.

返回 Original custom statistics method.

返回类型 Callable

statistics(*method_name*, **arg*, ***kwargs*)

Access statistics method by name.

参数 **method_name** (*str*) –Name of method.

返回 Depends on corresponding method.

返回类型 Any

update(*log_val*, *count=1*)

update the log history.

If the length of the buffer exceeds `self._max_length`, the oldest element will be removed from the buffer.

参数

- **log_val** (*int or float*) –The value of log.
- **count** (*int*) –The accumulation times of log, defaults to 1.
- **will be used in smooth statistics.** (*count*) –

返回类型 None

print_log	Print a log message.

47.4 mmengine.logging.print_log

mmengine.logging.**print_log**(*msg*, *logger=None*, *level=20*)

Print a log message.

参数

- **msg** (*str*) –The message to be logged.
- **logger** (*Logger or str, optional*) –If the type of logger is
- **logging.Logger** –Some special loggers are:
  - ” silent” : No message will be printed.
  - ” current” : Use latest created logger to log message.
  - other str: Instance name of logger. The corresponding logger will log message if it has been created, otherwise `print_log` will raise a *ValueError*.
  - None: The *print()* method will be used to print log messages.
- **directly use logger to log messages.** (*we*) –Some special loggers are:
  - ” silent” : No message will be printed.
  - ” current” : Use latest created logger to log message.
  - other str: Instance name of logger. The corresponding logger will log message if it has been created, otherwise `print_log` will raise a *ValueError*.
  - None: The *print()* method will be used to print log messages.
- **level** (*int*) –Logging level. Only available when *logger* is a Logger object, “current”, or a created logger instance name.

返回类型 None

CHAPTER 48

mmengine.visualization

mmengine.visualization

48.1 Visualizer

Visualizer	MMEngine provides a Visualizer class that uses the `Matplotlib` library as the backend.

48.1.1 Visualizer

class mmengine.visualization.**Visualizer**(*name='visualizer'*, *image=None*, *vis_backends=None*, *save_dir=None*, *fig_save_cfg={'frameon': False}*, *fig_show_cfg={'frameon': False}*)

MMEngine provides a Visualizer class that uses the `Matplotlib` library as the backend. It has the following functions:

- Basic drawing methods

  - draw_bboxes: draw single or multiple bounding boxes
  - draw_texts: draw single or multiple text boxes
  - draw_points: draw single or multiple points
  - draw_lines: draw single or multiple line segments
  - draw_circles: draw single or multiple circles
  - draw_polygons: draw single or multiple polygons
  - draw_binary_masks: draw single or multiple binary masks
  - draw_featmap: draw feature map

- Basic visualizer backend methods
  - add_configs: write config to all vis storage backends
  - add_graph: write model graph to all vis storage backends
  - add_image: write image to all vis storage backends
  - add_scalar: write scalar to all vis storage backends
  - add_scalars: write scalars to all vis storage backends
  - add_datasample: write datasample to all vis storage backends. The abstract drawing interface used by the user

- Basic info methods
  - set_image: sets the original image data
  - get_image: get the image data in Numpy format after drawing
  - show: visualization
  - close: close all resources that have been opened
  - get_backend: get the specified vis backend

All the basic drawing methods support chain calls, which is convenient for overlaydrawing and display. Each downstream algorithm library can inherit `Visualizer` and implement the add_datasample logic. For example, `DetLocalVisualizer` in MMDetection inherits from `Visualizer` and implements functions, such as visual detection boxes, instance masks, and semantic segmentation maps in the add_datasample interface.

参数

- **name** (*str*) –Name of the instance. Defaults to ‘visualizer’.
- **image** (*np.ndarray, optional*) –the origin image to draw. The format should be RGB. Defaults to None.
- **vis_backends** (*list, optional*) –Visual backend config list. Default to None.

- **save_dir** (*str, optional*) –Save file dir for all storage backends. If it is None, the backend storage will not save any data.
- **fig_save_cfg** (*dict*) –Keyword parameters of figure for saving. Defaults to empty dict.
- **fig_show_cfg** (*dict*) –Keyword parameters of figure for showing. Defaults to empty dict.

返回类型 None

实际案例

```
>>> # Basic info methods
>>> vis = Visualizer()
>>> vis.set_image(image)
>>> vis.get_image()
>>> vis.show()
```

```
>>> # Basic drawing methods
>>> vis = Visualizer(image=image)
>>> vis.draw_bboxes(np.array([0, 0, 1, 1]), edge_colors='g')
>>> vis.draw_bboxes(bbox=np.array([[1, 1, 2, 2], [2, 2, 3, 3]]),
>>>                     edge_colors=['g', 'r'])
>>> vis.draw_lines(x_datas=np.array([1, 3]),
>>>                y_datas=np.array([1, 3]),
>>>                colors='r', line_widths=1)
>>> vis.draw_lines(x_datas=np.array([[1, 3], [2, 4]]),
>>>                y_datas=np.array([[1, 3], [2, 4]]),
>>>                colors=['r', 'r'], line_widths=[1, 2])
>>> vis.draw_texts(text='MMEngine',
>>>               position=np.array([2, 2]),
>>>               colors='b')
>>> vis.draw_texts(text=['MMEngine','OpenMMLab'],
>>>                position=np.array([[2, 2], [5, 5]]),
>>>                colors=['b', 'b'])
>>> vis.draw_circles(circle_coord=np.array([2, 2]), radius=np.array[1])
>>> vis.draw_circles(circle_coord=np.array([[2, 2], [3, 5]),
>>>                  radius=np.array[1, 2], colors=['g', 'r'])
>>> vis.draw_polygons(np.array([0, 0, 1, 0, 1, 1, 0, 1]),
>>>                    edge_colors='g')
>>> vis.draw_polygons(bbox=[np.array([0, 0, 1, 0, 1, 1, 0, 1],
>>>                         np.array([2, 2, 3, 2, 3, 3, 2, 3]],
>>>                   edge_colors=['g', 'r'])
>>> vis.draw_binary_masks(binary_mask, alpha=0.6)
>>> heatmap = vis.draw_featmap(featmap, img,
```

(下页继续)

(续上页)

```
>>>                               channel_reduction='select_max')
>>> heatmap = vis.draw_featmap(featmap, img, channel_reduction=None,
>>>                               topk=8, arrangement=(4, 2))
>>> heatmap = vis.draw_featmap(featmap, img, channel_reduction=None,
>>>                               topk=-1)
```

```
>>> # chain calls
>>> vis.draw_bboxes().draw_texts().draw_circle().draw_binary_masks()
```

```
>>> # Backend related methods
>>> vis = Visualizer(vis_backends=[dict(type='LocalVisBackend')],
>>>                                  save_dir='temp_dir')
>>> cfg = Config(dict(a=1, b=dict(b1=[0, 1])))
>>> vis.add_config(cfg)
>>> image=np.random.randint(0, 256, size=(10, 10, 3)).astype(np.uint8)
>>> vis.add_image('image',image)
>>> vis.add_scaler('mAP', 0.6)
>>> vis.add_scalars({'loss': 0.1,'acc':0.8})
```

```
>>> # inherit
>>> class DetLocalVisualizer(Visualizer):
>>>        def add_datasample(self,
>>>                           name,
>>>                           image: np.ndarray,
>>>                           gt_sample:
>>>                               Optional['BaseDataElement'] = None,
>>>                           pred_sample:
>>>                               Optional['BaseDataElement'] = None,
>>>                           draw_gt: bool = True,
>>>                           draw_pred: bool = True,
>>>                           show: bool = False,
>>>                           wait_time: int = 0,
>>>                           step: int = 0) -> None:
>>>            pass
```

add_config(*config*, ***kwargs*)

Record the config.

参数 **config**(Config) –The Config object.

add_datasample(*name*, *image*, *data_sample=None*, *draw_gt=True*, *draw_pred=True*, *show=False*, *wait_time=0*, *step=0*)

Draw datasample.

参数

- **image** (*numpy.ndarray*) –
- **data_sample** (*Optional[mmengine.structures.base_data_element.BaseDataElement]*) –
- **draw_gt** (*bool*) –
- **draw_pred** (*bool*) –
- **show** (*bool*) –
- **wait_time** (*int*) –
- **step** (*int*) –

返回类型 None

add_graph(*model*, *data_batch*, ***kwargs*)

Record the model graph.

参数

- **model** (*torch.nn.Module*) –Model to draw.
- **data_batch** (*Sequence[dict]*) –Batch of data from dataloader.

返回类型 None

add_image(*name*, *image*, *step=0*)

Record the image.

参数

- **name** (*str*) –The image identifier.
- **image** (*np.ndarray, optional*) –The image to be saved. The format should be RGB. Default to None.
- **step** (*int*) –Global step value to record. Default to 0.

返回类型 None

add_scalar(*name*, *value*, *step=0*, ***kwargs*)

Record the scalar data.

参数

- **name** (*str*) –The scalar identifier.
- **value** (*float, int*) –Value to save.
- **step** (*int*) –Global step value to record. Default to 0.

返回类型 None

add_scalars(*scalar_dict*, *step=0*, *file_path=None*, ***kwargs*)

Record the scalars' data.

参数

- **scalar_dict** (*dict*) –Key-value pair storing the tag and corresponding values.
- **step** (*int*) –Global step value to record. Default to 0.
- **file_path** (*str, optional*) –The scalar's data will be saved to the *file_path* file at the same time if the *file_path* parameter is specified. Default to None.

返回类型 None

close()

close an opened object.

返回类型 None

property dataset_meta: Optional[dict]

Meta info of the dataset.

Type Optional[dict]

draw_bboxes(*bboxes*, *edge_colors='g'*, *line_styles='-'*, *line_widths=2*, *face_colors='none'*, *alpha=0.8*)

Draw single or multiple bboxes.

参数

- **bboxes** (*Union[np.ndarray, torch.Tensor]*) –The bboxes to draw with the format of(x1,y1,x2,y2).
- **edge_colors** (*Union[str, tuple, List[str], List[tuple]]*) –The colors of bboxes. `colors` can have the same length with lines or just single value. If `colors` is single value, all the lines will have the same colors. Refer to *matplotlib. colors* for full list of formats that are accepted. Defaults to 'g'.
- **line_styles** (*Union[str, List[str]]*) –The linestyle of lines. `line_styles` can have the same length with texts or just single value. If `line_styles` is single value, all the lines will have the same linestyle. Reference to https://matplotlib.org/stable/api/collections_api.html?highlight=collection#matplotlib.collections.AsteriskPolygonCollection.set_linestyle for more details. Defaults to '-'.
- **line_widths** (*Union[Union[int, float], List[Union[int, float]]]*) –The linewidth of lines. `line_widths` can have the same length with lines or just single value. If `line_widths` is single value, all the lines will have the same linewidth. Defaults to 2.
- **face_colors** (*Union[str, tuple, List[str], List[tuple]]*) –The face colors. Default to None.
- **alpha** (*Union[int, float]*) –The transparency of bboxes. Defaults to 0.8.

返回类型 *mmengine.visualization.visualizer.Visualizer*

draw_binary_masks(*binary_masks*, *colors='g'*, *alphas=0.8*)

Draw single or multiple binary masks.

参数

- **binary_masks** (*np.ndarray, torch.Tensor*) –The binary_masks to draw with of shape (N, H, W), where H is the image height and W is the image width. Each value in the array is either a 0 or 1 value of uint8 type.
- **colors** (*np.ndarray*) –The colors which binary_masks will convert to. `colors` can have the same length with binary_masks or just single value. If `colors` is single value, all the binary_masks will convert to the same colors. The colors format is RGB. Defaults to np.array([0, 255, 0]).
- **alphas** (*Union[int, List[int]]*) –The transparency of masks. Defaults to 0.8.

返回类型 *mmengine.visualization.visualizer.Visualizer*

draw_circles(*center*, *radius*, *edge_colors='g'*, *line_styles='-'*, *line_widths=2*, *face_colors='none'*, *alpha=0.8*)

Draw single or multiple circles.

参数

- **center** (*Union[np.ndarray, torch.Tensor]*) –The x coordinate of each line' start and end points.
- **radius** (*Union[np.ndarray, torch.Tensor]*) –The y coordinate of each line' start and end points.
- **edge_colors** (*Union[str, tuple, List[str], List[tuple]]*) –The colors of circles. `colors` can have the same length with lines or just single value. If `colors` is single value, all the lines will have the same colors. Reference to https://matplotlib.org/stable/gallery/color/named_colors.html for more details. Defaults to 'g.
- **line_styles** (*Union[str, List[str]]*) –The linestyle of lines. `line_styles` can have the same length with texts or just single value. If `line_styles` is single value, all the lines will have the same linestyle. Reference to https://matplotlib.org/stable/api/collections_api.html?highlight=collection#matplotlib.collections.AsteriskPolygonCollection.set_linestyle for more details. Defaults to '-'.
- **line_widths** (*Union[Union[int, float], List[Union[int, float]]]*) –The linewidth of lines. `line_widths` can have the same length with lines or just single value. If `line_widths` is single value, all the lines will have the same linewidth. Defaults to 2.
- **face_colors** (*Union[str, tuple, List[str], List[tuple]]*) –The face colors. Default to None.

- **alpha** (*Union[int, float]*) –The transparency of circles. Defaults to 0.8.

返回类型 *mmengine.visualization.visualizer.Visualizer*

static draw_featmap(*featmap*, *overlaid_image=None*, *channel_reduction='squeeze_mean'*, *topk=20*, *arrangement=(4, 5)*, *resize_shape=None*, *alpha=0.5*)

Draw featmap.

- If *overlaid_image* is not None, the final output image will be the

weighted sum of img and featmap.

- If *resize_shape* is specified, *featmap* and *overlaid_image*

are interpolated.

- If *resize_shape* is None and *overlaid_image* is not None,

the feature map will be interpolated to the spatial size of the image in the case where the spatial dimensions of *overlaid_image* and *featmap* are different.

- If *channel_reduction* is "squeeze_mean" and "select_max",

it will compress featmap to single channel image and weighted sum to *overlaid_image*.

- if *channel_reduction* is None

  - If topk <= 0, featmap is assert to be one or three

  channel and treated as image and will be weighted sum to `overlaid_image`. - If topk > 0, it will select topk channel to show by the sum of each channel. At the same time, you can specify the *arrangement* to set the window layout.

  参数

  - **featmap** (*torch.Tensor*) –The featmap to draw which format is (C, H, W).
  - **overlaid_image** (*np.ndarray, optional*) –The overlaid image. Default to None.
  - **channel_reduction** (*str, optional*) –Reduce multiple channels to a single channel. The optional value is 'squeeze_mean' or 'select_max'. Defaults to 'squeeze_mean'.
  - **topk** (*int*) –If channel_reduction is not None and topk > 0, it will select topk channel to show by the sum of each channel. if topk <= 0, tensor_chw is assert to be one or three. Defaults to 20.
  - **arrangement** (*Tuple[int, int]*) –The arrangement of featmap when channel_reduction is not None and topk > 0. Defaults to (4, 5).
  - **resize_shape** (*tuple, optional*) –The shape to scale the feature map. Default to None.

- **alpha** (*Union[int, List[int]]*) –The transparency of featmap. Defaults to 0.5.

返回 RGB image.

返回类型 np.ndarray

draw_lines(*x_datas*, *y_datas*, *colors='g'*, *line_styles='-'*, *line_widths=2*)

Draw single or multiple line segments.

参数

- **x_datas** (*Union[np.ndarray, torch.Tensor]*) –The x coordinate of each line' start and end points.
- **y_datas** (*Union[np.ndarray, torch.Tensor]*) –The y coordinate of each line' start and end points.
- **colors** (*Union[str, tuple, List[str], List[tuple]]*) –The colors of lines. `colors` can have the same length with lines or just single value. If `colors` is single value, all the lines will have the same colors. Reference to https://matplotlib.org/stable/gallery/color/named_colors.html for more details. Defaults to 'g'.
- **line_styles** (*Union[str, List[str]]*) –The linestyle of lines. `line_styles` can have the same length with texts or just single value. If `line_styles` is single value, all the lines will have the same linestyle. Reference to https://matplotlib.org/stable/api/collections_api.html?highlight=collection#matplotlib.collections.AsteriskPolygonCollection.set_linestyle for more details. Defaults to '-'.
- **line_widths** (*Union[Union[int, float], List[Union[int, float]]]*) –The linewidth of lines. `line_widths` can have the same length with lines or just single value. If `line_widths` is single value, all the lines will have the same linewidth. Defaults to 2.

返回类型 *mmengine.visualization.visualizer.Visualizer*

draw_points(*positions*, *colors='g'*, *marker=None*, *sizes=None*)

Draw single or multiple points.

参数

- **positions** (*Union[np.ndarray, torch.Tensor]*) –Positions to draw.
- **colors** (*Union[str, tuple, List[str], List[tuple]]*) –The colors of points. `colors` can have the same length with points or just single value. If `colors` is single value, all the points will have the same colors. Reference to https://matplotlib.org/stable/gallery/color/named_colors.html for more details. Defaults to 'g.
- **marker** (*str, optional*) –The marker style. See `matplotlib.markers` for more information about marker styles. Default to None.

- **sizes** (*Optional[Union[np.ndarray, torch.Tensor]]*) –The marker size. Default to None.

draw_polygons(*polygons*, *edge_colors='g'*, *line_styles='-'*, *line_widths=2*, *face_colors='none'*, *alpha=0.8*)

Draw single or multiple bboxes.

参数

- **(Union[Union[np.ndarray** (*polygons*) –List[Union[np.ndarray, torch.Tensor]]]): The polygons to draw with the format of (x1,y1,x2,y2,…,xn,yn).
- **torch.Tensor]** –List[Union[np.ndarray, torch.Tensor]]]): The polygons to draw with the format of (x1,y1,x2,y2,…,xn,yn).
- **polygons** (*Union[numpy.ndarray, torch.Tensor, List[Union[numpy.ndarray, torch.Tensor]]]*) –
- **edge_colors** (*Union[str, tuple, List[str], List[tuple]]*) –
- **line_styles** (*Union[str, List[str]]*) –
- **line_widths** (*Union[int, float, List[Union[int, float]]]*) –
- **face_colors** (*Union[str, tuple, List[str], List[tuple]]*) –
- **alpha** (*Union[int, float]*) –

返回类型 *mmengine.visualization.visualizer.Visualizer*

:param [List[Union[np.ndarray, torch.Tensor]]]): The polygons to draw] with the format of (x1,y1,x2,y2,…,xn,yn).

参数

- **edge_colors** (*Union[str, tuple, List[str], List[tuple]]*) –The colors of polygons. `colors` can have the same length with lines or just single value. If `colors` is single value, all the lines will have the same colors. Refer to *matplotlib.colors* for full list of formats that are accepted. Defaults to ‘g.
- **line_styles** (*Union[str, List[str]]*) –The linestyle of lines. `line_styles` can have the same length with texts or just single value. If `line_styles` is single value, all the lines will have the same linestyle. Reference to https://matplotlib.org/stable/api/collections_api.html?highlight=collection#matplotlib.collections.AsteriskPolygonCollection.set_linestyle for more details. Defaults to ‘- ‘.
- **line_widths** (*Union[Union[int, float], List[Union[int, float]]]*) –The linewidth of lines. `line_widths` can have the same length with lines or just single value. If `line_widths` is single value, all the lines will have the same linewidth. Defaults to 2.

- **face_colors** (*Union[str, tuple, List[str], List[tuple]]*) –The face colors. Default to None.
- **alpha** (*Union[int, float]*) –The transparency of polygons. Defaults to 0.8.
- **polygons** (*Union[numpy.ndarray, torch.Tensor, List[Union[numpy.ndarray, torch.Tensor]]]*) –

返回类型 *mmengine.visualization.visualizer.Visualizer*

draw_texts(*texts*, *positions*, *font_sizes=None*, *colors='g'*, *vertical_alignments='top'*, *horizontal_alignments='left'*, *font_families='sans-serif'*, *bboxes=None*)

Draw single or multiple text boxes.

参数

- **texts** (*Union[str, List[str]]*) –Texts to draw.
- **positions** (*Union[np.ndarray, torch.Tensor]*) –The position to draw the texts, which should have the same length with texts and each dim contain x and y.
- **font_sizes** (*Union[int, List[int]], optional*) –The font size of texts. `font_sizes` can have the same length with texts or just single value. If `font_sizes` is single value, all the texts will have the same font size. Defaults to None.
- **colors** (*Union[str, tuple, List[str], List[tuple]]*) –The colors of texts. `colors` can have the same length with texts or just single value. If `colors` is single value, all the texts will have the same colors. Reference to https://matplotlib.org/stable/gallery/color/named_colors.html for more details. Defaults to 'g.
- **vertical_alignments** (*Union[str, List[str]]*) –The verticalalignment of texts. verticalalignment controls whether the y positional argument for the text indicates the bottom, center or top side of the text bounding box. `vertical_alignments` can have the same length with texts or just single value. If `vertical_alignments` is single value, all the texts will have the same verticalalignment. verticalalignment can be 'center' or 'top', 'bottom' or 'baseline'. Defaults to 'top'.
- **horizontal_alignments** (*Union[str, List[str]]*) –The horizontalalignment of texts. Horizontalalignment controls whether the x positional argument for the text indicates the left, center or right side of the text bounding box. `horizontal_alignments` can have the same length with texts or just single value. If `horizontal_alignments` is single value, all the texts will have the same horizontalalignment. Horizontalalignment can be 'center','right' or 'left'. Defaults to 'left'.
- **font_families** (*Union[str, List[str]]*) –The font family of texts. `font_families` can have the same length with texts or just single value. If `font_families` is single value, all the texts will have the same font family. font_familiy can be 'serif', 'sans-serif', 'cursive', 'fantasy'

or ‘monospace’. Defaults to ‘sans-serif’.

- **bboxes** (*Union[dict, List[dict]], optional*) –The bounding box of the texts. If bboxes is None, there are no bounding box around texts. `bboxes` can have the same length with texts or just single value. If `bboxes` is single value, all the texts will have the same bbox. Reference to https://matplotlib.org/stable/api/_as_gen/matplotlib.patches.FancyBboxPatch.html#matplotlib.patches.FancyBboxPatch for more details. Defaults to None.

返回类型 *mmengine.visualization.visualizer.Visualizer*

get_backend(*name*)

get vis backend by name.

参数 **name** (*str*) –The name of vis backend

返回 The vis backend.

返回类型 *BaseVisBackend*

get_image()

Get the drawn image. The format is RGB.

返回 the drawn image which channel is RGB.

返回类型 np.ndarray

classmethod get_instance(*name*, ***kwargs*)

Make subclass can get latest created instance by `Visualizer.get_current_instance()`.

Downstream codebase may need to get the latest created instance without knowing the specific Visualizer type. For example, mmdetection builds visualizer in runner and some component which cannot access runner wants to get latest created visualizer. In this case, the component does not know which type of visualizer has been built and cannot get target instance. Therefore, *Visualizer* overrides the *get_instance()* and its subclass will register the created instance to `_instance_dict` additionally. `get_current_instance()` will return the latest created subclass instance.

实际案例

```
>>> class DetLocalVisualizer(Visualizer):
>>>     def __init__(self, name):
>>>         super().__init__(name)
>>>
>>> visualizer1 = DetLocalVisualizer.get_instance('name1')
>>> visualizer2 = Visualizer.get_current_instance()
>>> visualizer3 = DetLocalVisualizer.get_current_instance()
>>> assert id(visualizer1) == id(visualizer2) == id(visualizer3)
```

参数 **name** (*str*) –Name of instance.

返回 Corresponding name instance.

返回类型 object

set_image(*image*)

Set the image to draw.

参数 **image** (*np.ndarray*) –The image to draw.

返回类型 None

show(*drawn_img=None*, *win_name='image'*, *wait_time=0*, *continue_key=' '*)

Show the drawn image.

参数

- **drawn_img** (*np.ndarray, optional*) –The image to show. If drawn_img is None, it will show the image got by Visualizer. Defaults to None.
- **win_name** (*str*) –The image title. Defaults to 'image'.
- **wait_time** (*int*) –Delay in milliseconds. 0 is the special value that means "forever". Defaults to 0.
- **continue_key** (*str*) –The key for users to continue. Defaults to the space key.

返回类型 None

48.2 visualization Backend

BaseVisBackend	Base class for visualization backend.
LocalVisBackend	Local visualization backend class.
TensorboardVisBackend	Tensorboard visualization backend class.
WandbVisBackend	Wandb visualization backend class.

48.2.1 BaseVisBackend

class mmengine.visualization.**BaseVisBackend**(*save_dir*)

Base class for visualization backend.

All backends must inherit `BaseVisBackend` and implement the required functions.

参数 **save_dir** (*str, optional*) –The root directory to save the files produced by the backend.

add_config(*config*, ***kwargs*)

Record the config.

参数 **config** (Config) –The Config object

返回类型 None

add_graph(*model*, *data_batch*, ***kwargs*)

Record the model graph.

参数

- **model** (*torch.nn.Module*) –Model to draw.
- **data_batch** (*Sequence[dict]*) –Batch of data from dataloader.

返回类型 None

add_image(*name*, *image*, *step=0*, ***kwargs*)

Record the image.

参数

- **name** (*str*) –The image identifier.
- **image** (*np.ndarray*) –The image to be saved. The format should be RGB. Default to None.
- **step** (*int*) –Global step value to record. Default to 0.

返回类型 None

add_scalar(*name*, *value*, *step=0*, ***kwargs*)

Record the scalar.

参数

- **name** (*str*) –The scalar identifier.
- **value** (*int, float*) –Value to save.
- **step** (*int*) –Global step value to record. Default to 0.

返回类型 None

add_scalars(*scalar_dict*, *step=0*, *file_path=None*, ***kwargs*)

Record the scalars' data.

参数

- **scalar_dict** (*dict*) –Key-value pair storing the tag and corresponding values.
- **step** (*int*) –Global step value to record. Default to 0.
- **file_path** (*str, optional*) –The scalar's data will be saved to the *file_path* file at the same time if the *file_path* parameter is specified. Default to None.

返回类型 None

close()

close an opened object.

返回类型 None

abstract property experiment: Any

Return the experiment object associated with this visualization backend.

The experiment attribute can get the visualization backend, such as wandb, tensorboard. If you want to write other data, such as writing a table, you can directly get the visualization backend through experiment.

48.2.2 LocalVisBackend

class mmengine.visualization.**LocalVisBackend**(*save_dir*, *img_save_dir='vis_image'*, *config_save_file='config.py'*, *scalar_save_file='scalars.json'*)

Local visualization backend class.

It can write image, config, scalars, etc. to the local hard disk. You can get the drawing backend through the experiment property for custom drawing.

实际案例

```
>>> from mmengine.visualization import LocalVisBackend
>>> import numpy as np
>>> local_vis_backend = LocalVisBackend(save_dir='temp_dir')
>>> img = np.random.randint(0, 256, size=(10, 10, 3))
>>> local_vis_backend.add_image('img', img)
>>> local_vis_backend.add_scalar('mAP', 0.6)
>>> local_vis_backend.add_scalars({'loss': [1, 2, 3], 'acc': 0.8})
>>> cfg = Config(dict(a=1, b=dict(b1=[0, 1])))
>>> local_vis_backend.add_config(cfg)
```

参数

- **save_dir** (*str, optional*) –The root directory to save the files produced by the visualizer. If it is none, it means no data is stored.
- **img_save_dir** (*str*) –The directory to save images. Default to ‘vis_image’.
- **config_save_file** (*str*) –The file name to save config. Default to ‘config.py’.
- **scalar_save_file** (*str*) –The file name to save scalar values. Default to ‘scalars.json’.

add_config(*config*, ***kwargs*)

Record the config to disk.

参数 **config** (Config) –The Config object

返回类型 None

add_image(*name*, *image*, *step=0*, ***kwargs*)

Record the image to disk.

参数

- **name** (*str*) –The image identifier.
- **image** (*np.ndarray*) –The image to be saved. The format should be RGB. Default to None.
- **step** (*int*) –Global step value to record. Default to 0.

返回类型 None

add_scalar(*name*, *value*, *step=0*, ***kwargs*)

Record the scalar data to disk.

参数

- **name** (*str*) –The scalar identifier.
- **value** (*int, float, torch.Tensor, np.ndarray*) –Value to save.
- **step** (*int*) –Global step value to record. Default to 0.

返回类型 None

add_scalars(*scalar_dict*, *step=0*, *file_path=None*, ***kwargs*)

Record the scalars to disk.

The scalar dict will be written to the default and specified files if `file_path` is specified.

参数

- **scalar_dict** (*dict*) –Key-value pair storing the tag and corresponding values. The value must be dumped into json format.
- **step** (*int*) –Global step value to record. Default to 0.
- **file_path** (*str, optional*) –The scalar's data will be saved to the `file_path` file at the same time if the `file_path` parameter is specified. Default to None.

返回类型 None

property experiment: mmengine.visualization.vis_backend.LocalVisBackend

Return the experiment object associated with this visualization backend.

48.2.3 TensorboardVisBackend

class mmengine.visualization.**TensorboardVisBackend**(*save_dir*)

Tensorboard visualization backend class.

It can write images, config, scalars, etc. to a tensorboard file.

实际案例

```
>>> from mmengine.visualization import TensorboardVisBackend
>>> import numpy as np
>>> tensorboard_vis_backend =          >>>     TensorboardVisBackend(save_dir=
↪'temp_dir')
>>> img=np.random.randint(0, 256, size=(10, 10, 3))
>>> tensorboard_vis_backend.add_image('img', img)
>>> tensorboard_vis_backend.add_scaler('mAP', 0.6)
>>> tensorboard_vis_backend.add_scalars({'loss': 0.1,'acc':0.8})
>>> cfg = Config(dict(a=1, b=dict(b1=[0, 1])))
>>> tensorboard_vis_backend.add_config(cfg)
```

参数 **save_dir** (*str*) –The root directory to save the files produced by the backend.

add_config(*config*, ***kwargs*)

Record the config to tensorboard.

参数 **config** (Config) –The Config object

返回类型 None

add_image(*name*, *image*, *step=0*, ***kwargs*)

Record the image to tensorboard.

参数

- **name** (*str*) –The image identifier.
- **image** (*np.ndarray*) –The image to be saved. The format should be RGB.
- **step** (*int*) –Global step value to record. Default to 0.

返回类型 None

add_scalar(*name*, *value*, *step=0*, ***kwargs*)

Record the scalar data to tensorboard.

参数

- **name** (*str*) –The scalar identifier.
- **value** (*int*, *float*, *torch.Tensor*, *np.ndarray*) –Value to save.

- **step** (*int*) –Global step value to record. Default to 0.

返回类型 None

add_scalars(*scalar_dict*, *step=0*, *file_path=None*, ***kwargs*)

Record the scalar’ s data to tensorboard.

参数

- **scalar_dict** (*dict*) –Key-value pair storing the tag and corresponding values.
- **step** (*int*) –Global step value to record. Default to 0.
- **file_path** (*str, optional*) –Useless parameter. Just for interface unification. Default to None.

返回类型 None

close()

close an opened tensorboard object.

property experiment

Return Tensorboard object.

48.2.4 WandbVisBackend

class mmengine.visualization.**WandbVisBackend**(*save_dir*, *init_kwargs=None*, *define_metric_cfg=None*, *commit=True*)

Wandb visualization backend class.

实际案例

```
>>> from mmengine.visualization import WandbVisBackend
>>> import numpy as np
>>> wandb_vis_backend = WandbVisBackend()
>>> img=np.random.randint(0, 256, size=(10, 10, 3))
>>> wandb_vis_backend.add_image('img', img)
>>> wandb_vis_backend.add_scaler('mAP', 0.6)
>>> wandb_vis_backend.add_scalars({'loss': [1, 2, 3],'acc': 0.8})
>>> cfg = Config(dict(a=1, b=dict(b1=[0, 1])))
>>> wandb_vis_backend.add_config(cfg)
```

参数

- **save_dir** (*str, optional*) –The root directory to save the files produced by the visualizer.

- **init_kwargs** (*dict, optional*) –wandb initialization input parameters. Default to None.
- **define_metric_cfg** (*dict, optional*) –A dict of metrics and summary for wandb.define_metric. The key is metric and the value is summary. When `define_metric_cfg={'coco/bbox_mAP': 'max'}`, The maximum value of"coco/bbox_mAP" is logged on wandb UI. See wandb docs for details. Default: None
- **commit** (*Optional[bool]*) –(bool, optional) Save the metrics dict to the wandb server and increment the step. If false *wandb.log* just updates the current metrics dict with the row argument and metrics won't be saved until *wandb.log* is called with *commit=True*. Default to True.

add_config(*config*, ***kwargs*)

Record the config to wandb.

参数 **config** (Config) –The Config object

返回类型 None

add_image(*name*, *image*, *step=0*, ***kwargs*)

Record the image to wandb.

参数

- **name** (*str*) –The image identifier.
- **image** (*np.ndarray*) –The image to be saved. The format should be RGB.
- **step** (*int*) –Useless parameter. Wandb does not need this parameter. Default to 0.

返回类型 None

add_scalar(*name*, *value*, *step=0*, ***kwargs*)

Record the scalar data to wandb.

参数

- **name** (*str*) –The scalar identifier.
- **value** (*int, float, torch.Tensor, np.ndarray*) –Value to save.
- **step** (*int*) –Useless parameter. Wandb does not need this parameter. Default to 0.

返回类型 None

add_scalars(*scalar_dict*, *step=0*, *file_path=None*, ***kwargs*)

Record the scalar's data to wandb.

参数

- **scalar_dict** (*dict*) –Key-value pair storing the tag and corresponding values.
- **step** (*int*) –Useless parameter. Wandb does not need this parameter. Default to 0.

- **file_path** (*str, optional*) –Useless parameter. Just for interface unification. Default to None.

返回类型 None

close()

close an opened wandb object.

返回类型 None

property experiment

Return wandb object.

The experiment attribute can get the wandb backend, If you want to write other data, such as writing a table, you can directly get the wandb backend through experiment.

CHAPTER 49

mmengine.fileio

mmengine.fileio

49.1 File Backend

`BaseStorageBackend`	Abstract class of storage backends.
`FileClient`	A general file client to access files in different backends.
`HardDiskBackend`	Raw hard disks storage backend.
`LocalBackend`	Raw local storage backend.
`HTTPBackend`	HTTP and HTTPS storage bachend.
`LmdbBackend`	Lmdb storage backend.
`MemcachedBackend`	Memcached storage backend.
`PetrelBackend`	Petrel storage backend (for internal usage).

49.1.1 BaseStorageBackend

class mmengine.fileio.**BaseStorageBackend**

Abstract class of storage backends.

All backends need to implement two apis: *get()* and *get_text()*.

- *get()* reads the file as a byte stream.
- *get_text()* reads the file as texts.

49.1.2 FileClient

class mmengine.fileio.**FileClient**(*backend=None*, *prefix=None*, ***kwargs*)

A general file client to access files in different backends.

The client loads a file or text in a specified backend from its path and returns it as a binary or text file. There are two ways to choose a backend, the name of backend and the prefix of path. Although both of them can be used to choose a storage backend, `backend` has a higher priority that is if they are all set, the storage backend will be chosen by the backend argument. If they are all *None*, the disk backend will be chosen. Note that It can also register other backend accessor with a given name, prefixes, and backend class. In addition, We use the singleton pattern to avoid repeated object creation. If the arguments are the same, the same object will be returned.

> **警告:** *FileClient* will be deprecated in future. Please use io functions in https://mmengine.readthedocs.io/en/latest/api/fileio.html#file-io

参数

- **backend** (*str, optional*) –The storage backend type. Options are "disk", "memcached", "lmdb", "http" and "petrel". Default: None.
- **prefix** (*str, optional*) –The prefix of the registered storage backend. Options are "s3", "http", "https". Default: None.

实际案例

```
>>> # only set backend
>>> file_client = FileClient(backend='petrel')
>>> # only set prefix
>>> file_client = FileClient(prefix='s3')
>>> # set both backend and prefix but use backend to choose client
>>> file_client = FileClient(backend='petrel', prefix='s3')
>>> # if the arguments are the same, the same object is returned
```

(下页继续)

(续上页)

```
>>> file_client1 = FileClient(backend='petrel')
>>> file_client1 is file_client
True
```

client

The backend object.

Type *BaseStorageBackend*

exists(*filepath*)

Check whether a file path exists.

参数 **filepath**(*str or Path*) –Path to be checked whether exists.

返回 Return `True` if `filepath` exists, `False` otherwise.

返回类型 bool

get(*filepath*)

Read data from a given `filepath` with 'rb' mode.

注解: There are two types of return values for `get`, one is `bytes` and the other is `memoryview`. The advantage of using memoryview is that you can avoid copying, and if you want to convert it to `bytes`, you can use `.tobytes()`.

参数 **filepath**(*str or Path*) –Path to read data.

返回 Expected bytes object or a memory view of the bytes object.

返回类型 bytes | memoryview

get_local_path(*filepath*)

Download data from `filepath` and write the data to local path.

`get_local_path` is decorated by `contxtlib.contextmanager()`. It can be called with `with` statement, and when exists from the `with` statement, the temporary path will be released.

注解: If the `filepath` is a local path, just return itself.

警告: `get_local_path` is an experimental interface that may change in the future.

参数 **filepath**(*str or Path*) –Path to be read data.

返回类型 Generator[Union[str, pathlib.Path], None, None]

实际案例

```
>>> file_client = FileClient(prefix='s3')
>>> with file_client.get_local_path('s3://bucket/abc.jpg') as path:
...     # do something here
```

生成器 *Iterable[str]* –Only yield one path.

参数 **filepath** (*Union[str, pathlib.Path]*) –

返回类型 Generator[Union[str, pathlib.Path], None, None]

get_text (*filepath*, *encoding='utf-8'*)

Read data from a given `filepath` with 'r' mode.

参数

- **filepath** (*str or Path*) –Path to read data.
- **encoding** (*str*) –The encoding format used to open the `filepath`. Default: 'utf-8'.

返回 Expected text reading from `filepath`.

返回类型 str

classmethod infer_client (*file_client_args=None*, *uri=None*)

Infer a suitable file client based on the URI and arguments.

参数

- **file_client_args** (*dict, optional*) –Arguments to instantiate a FileClient. Default: None.
- **uri** (*str | Path, optional*) –Uri to be parsed that contains the file prefix. Default: None.

返回类型 *mmengine.fileio.file_client.FileClient*

实际案例

```
>>> uri = 's3://path/of/your/file'
>>> file_client = FileClient.infer_client(uri=uri)
>>> file_client_args = {'backend': 'petrel'}
>>> file_client = FileClient.infer_client(file_client_args)
```

返回 Instantiated FileClient object.

返回类型 *FileClient*

参数

- **file_client_args** (*Optional[dict]*) –
- **uri** (*Optional[Union[str, pathlib.Path]]*) –

isdir (*filepath*)

Check whether a file path is a directory.

参数 **filepath** (*str or Path*) –Path to be checked whether it is a directory.

返回 Return `True` if `filepath` points to a directory, `False` otherwise.

返回类型 bool

isfile (*filepath*)

Check whether a file path is a file.

参数 **filepath** (*str or Path*) –Path to be checked whether it is a file.

返回 Return `True` if `filepath` points to a file, `False` otherwise.

返回类型 bool

join_path (*filepath*, **filepaths*)

Concatenate all file paths.

Join one or more filepath components intelligently. The return value is the concatenation of filepath and any members of *filepaths.

参数

- **filepath** (*str or Path*) –Path to be concatenated.
- **filepaths** (*Union[str, pathlib.Path]*) –

返回 The result of concatenation.

返回类型 str

list_dir_or_file (*dir_path*, *list_dir=True*, *list_file=True*, *suffix=None*, *recursive=False*)

Scan a directory to find the interested directories or files in arbitrary order.

注解: *list_dir_or_file()* returns the path relative to `dir_path`.

参数

- **dir_path** (*str | Path*) –Path of the directory.
- **list_dir** (*bool*) –List the directories. Default: True.
- **list_file** (*bool*) –List the path of files. Default: True.

- **suffix** (*str or tuple[str], optional*) –File suffix that we are interested in. Default: None.
- **recursive** (*bool*) –If set to True, recursively scan the directory. Default: False.

生成器 *Iterable[str]* –A relative path to `dir_path`.

返回类型 Iterator[str]

static parse_uri_prefix(*uri*)

Parse the prefix of a uri.

参数 **uri** (*str | Path*) –Uri to be parsed that contains the file prefix.

返回类型 Optional[str]

实际案例

```
>>> FileClient.parse_uri_prefix('s3://path/of/your/file')
's3'
```

返回 Return the prefix of uri if the uri contains '://' else `None`.

返回类型 str | None

参数 **uri** (*Union[str, pathlib.Path]*) –

put(*obj*, *filepath*)

Write data to a given `filepath` with 'wb' mode.

注解: `put` should create a directory if the directory of `filepath` does not exist.

参数

- **obj** (*bytes*) –Data to be written.
- **filepath** (*str or Path*) –Path to write data.

返回类型 None

put_text(*obj*, *filepath*)

Write data to a given `filepath` with 'w' mode.

注解: `put_text` should create a directory if the directory of `filepath` does not exist.

参数

- **obj** (*str*) –Data to be written.
- **filepath** (*str or Path*) –Path to write data.
- **encoding** (*str, optional*) –The encoding format used to open the *filepath*. Default: 'utf-8'.

返回类型 None

classmethod register_backend(*name*, *backend=None*, *force=False*, *prefixes=None*)

Register a backend to FileClient.

This method can be used as a normal class method or a decorator.

```
class NewBackend(BaseStorageBackend):

    def get(self, filepath):
        return filepath

    def get_text(self, filepath):
        return filepath

FileClient.register_backend('new', NewBackend)
```

or

```
@FileClient.register_backend('new')
class NewBackend(BaseStorageBackend):

    def get(self, filepath):
        return filepath

    def get_text(self, filepath):
        return filepath
```

参数

- **name** (*str*) –The name of the registered backend.
- **backend** (*class, optional*) –The backend class to be registered, which must be a subclass of *BaseStorageBackend*. When this method is used as a decorator, backend is None. Defaults to None.
- **force** (*bool, optional*) –Whether to override the backend if the name has already been registered. Defaults to False.
- **prefixes** (*str or list[str] or tuple[str], optional*) –The prefixes of the registered storage backend. Default: None. *New in version 1.3.15.*

remove(*filepath*)

Remove a file.

参数 **filepath** (*str, Path*) –Path to be removed.

返回类型 None

49.1.3 HardDiskBackend

class mmengine.fileio.**HardDiskBackend**

Raw hard disks storage backend.

返回类型 None

49.1.4 LocalBackend

class mmengine.fileio.**LocalBackend**

Raw local storage backend.

copy_if_symlink_fails(*src*, *dst*)

Create a symbolic link pointing to src named dst.

If failed to create a symbolic link pointing to src, directly copy src to dst instead.

参数

- **src** (*str or Path*) –Create a symbolic link pointing to src.
- **dst** (*str or Path*) –Create a symbolic link named dst.

返回 Return True if successfully create a symbolic link pointing to src. Otherwise, return False.

返回类型 bool

实际案例

```
>>> backend = LocalBackend()
>>> src = '/path/of/file'
>>> dst = '/path1/of/file1'
>>> backend.copy_if_symlink_fails(src, dst)
True
>>> src = '/path/of/dir'
>>> dst = '/path1/of/dir1'
>>> backend.copy_if_symlink_fails(src, dst)
True
```

copyfile(*src*, *dst*)

Copy a file src to dst and return the destination file.

src and dst should have the same prefix. If dst specifies a directory, the file will be copied into dst using the base filename from src. If dst specifies a file that already exists, it will be replaced.

参数

- **src** (*str or Path*) –A file to be copied.
- **dst** (*str or Path*) –Copy file to dst.

返回 The destination file.

返回类型 str

引发 **SameFileError** –If src and dst are the same file, a SameFileError will be raised.

实际案例

```
>>> backend = LocalBackend()
>>> # dst is a file
>>> src = '/path/of/file'
>>> dst = '/path1/of/file1'
>>> # src will be copied to '/path1/of/file1'
>>> backend.copyfile(src, dst)
'/path1/of/file1'
```

```
>>> # dst is a directory
>>> dst = '/path1/of/dir'
>>> # src will be copied to '/path1/of/dir/file'
>>> backend.copyfile(src, dst)
'/path1/of/dir/file'
```

copyfile_from_local(*src*, *dst*)

Copy a local file src to dst and return the destination file. Same as *copyfile()*.

参数

- **src** (*str or Path*) –A local file to be copied.
- **dst** (*str or Path*) –Copy file to dst.

返回 If dst specifies a directory, the file will be copied into dst using the base filename from src.

返回类型 str

引发 **SameFileError** –If src and dst are the same file, a SameFileError will be raised.

实际案例

```
>>> backend = LocalBackend()
>>> # dst is a file
>>> src = '/path/of/file'
>>> dst = '/path1/of/file1'
>>> # src will be copied to '/path1/of/file1'
>>> backend.copyfile_from_local(src, dst)
'/path1/of/file1'
```

```
>>> # dst is a directory
>>> dst = '/path1/of/dir'
>>> # src will be copied to
>>> backend.copyfile_from_local(src, dst)
'/path1/of/dir/file'
```

copyfile_to_local(*src*, *dst*)

Copy the file src to local dst and return the destination file. Same as *copyfile()*.

If dst specifies a directory, the file will be copied into dst using the base filename from src. If dst specifies a file that already exists, it will be replaced.

参数

- **src** (*str or Path*) –A file to be copied.
- **dst** (*str or Path*) –Copy file to to local dst.

返回 If dst specifies a directory, the file will be copied into dst using the base filename from src.

返回类型 str

实际案例

```
>>> backend = LocalBackend()
>>> # dst is a file
>>> src = '/path/of/file'
>>> dst = '/path1/of/file1'
>>> # src will be copied to '/path1/of/file1'
>>> backend.copyfile_to_local(src, dst)
'/path1/of/file1'
```

```
>>> # dst is a directory
>>> dst = '/path1/of/dir'
>>> # src will be copied to
```

(下页继续)

(续上页)

```
>>> backend.copyfile_to_local(src, dst)
'/path1/of/dir/file'
```

copytree(*src*, *dst*)

Recursively copy an entire directory tree rooted at src to a directory named dst and return the destination directory.

src and dst should have the same prefix and dst must not already exist.

TODO: Whether to support dirs_exist_ok parameter.

参数

- **src** (*str or Path*) –A directory to be copied.
- **dst** (*str or Path*) –Copy directory to dst.

返回 The destination directory.

返回类型 str

引发 **FileExistsError** –If dst had already existed, a FileExistsError will be raised.

实际案例

```
>>> backend = LocalBackend()
>>> src = '/path/of/dir1'
>>> dst = '/path/of/dir2'
>>> backend.copytree(src, dst)
'/path/of/dir2'
```

copytree_from_local(*src*, *dst*)

Recursively copy an entire directory tree rooted at src to a directory named dst and return the destination directory. Same as *copytree()*.

参数

- **src** (*str or Path*) –A local directory to be copied.
- **dst** (*str or Path*) –Copy directory to dst.

返回 The destination directory.

返回类型 str

实际案例

```
>>> backend = LocalBackend()
>>> src = '/path/of/dir1'
>>> dst = '/path/of/dir2'
>>> backend.copytree_from_local(src, dst)
'/path/of/dir2'
```

copytree_to_local(*src*, *dst*)

Recursively copy an entire directory tree rooted at src to a local directory named dst and return the destination directory.

参数

- **src** (*str or Path*) –A directory to be copied.
- **dst** (*str or Path*) –Copy directory to local dst.
- **backend_args** (*dict, optional*) –Arguments to instantiate the preifx of uri corresponding backend. Defaults to None.

返回 The destination directory.

返回类型 str

实际案例

```
>>> backend = LocalBackend()
>>> src = '/path/of/dir1'
>>> dst = '/path/of/dir2'
>>> backend.copytree_from_local(src, dst)
'/path/of/dir2'
```

exists(*filepath*)

Check whether a file path exists.

参数 **filepath** (*str or Path*) –Path to be checked whether exists.

返回 Return `True` if `filepath` exists, `False` otherwise.

返回类型 bool

实际案例

```
>>> backend = LocalBackend()
>>> filepath = '/path/of/file'
>>> backend.exists(filepath)
True
```

get(*filepath*)

Read bytes from a given `filepath` with 'rb' mode.

参数 **filepath** (*str or Path*) –Path to read data.

返回 Expected bytes object.

返回类型 bytes

实际案例

```
>>> backend = LocalBackend()
>>> filepath = '/path/of/file'
>>> backend.get(filepath)
b'hello world'
```

get_local_path(*filepath*)

Only for unified API and do nothing.

参数

- **filepath** (*str or Path*) –Path to be read data.
- **backend_args** (*dict, optional*) –Arguments to instantiate the corresponding backend. Defaults to None.

返回类型 Generator[Union[str, pathlib.Path], None, None]

实际案例

```
>>> backend = LocalBackend()
>>> with backend.get_local_path('s3://bucket/abc.jpg') as path:
...     # do something here
```

get_text(*filepath*, *encoding='utf-8'*)

Read text from a given `filepath` with 'r' mode.

参数

- **filepath** (*str or Path*) –Path to read data.

- **encoding** (*str*) –The encoding format used to open the `filepath`. Defaults to 'utf-8'.

返回 Expected text reading from `filepath`.

返回类型 str

实际案例

```
>>> backend = LocalBackend()
>>> filepath = '/path/of/file'
>>> backend.get_text(filepath)
'hello world'
```

isdir(*filepath*)

Check whether a file path is a directory.

参数 **filepath** (*str or Path*) –Path to be checked whether it is a directory.

返回 Return `True` if `filepath` points to a directory, `False` otherwise.

返回类型 bool

实际案例

```
>>> backend = LocalBackend()
>>> filepath = '/path/of/dir'
>>> backend.isdir(filepath)
True
```

isfile(*filepath*)

Check whether a file path is a file.

参数 **filepath** (*str or Path*) –Path to be checked whether it is a file.

返回 Return `True` if `filepath` points to a file, `False` otherwise.

返回类型 bool

实际案例

```
>>> backend = LocalBackend()
>>> filepath = '/path/of/file'
>>> backend.isfile(filepath)
True
```

join_path(*filepath*, **filepaths*)

Concatenate all file paths.

Join one or more filepath components intelligently. The return value is the concatenation of filepath and any members of *filepaths.

参数

- **filepath** (*str or Path*) –Path to be concatenated.
- **filepaths** (*Union[str, pathlib.Path]*) –

返回 The result of concatenation.

返回类型 str

实际案例

```
>>> backend = LocalBackend()
>>> filepath1 = '/path/of/dir1'
>>> filepath2 = 'dir2'
>>> filepath3 = 'path/of/file'
>>> backend.join_path(filepath1, filepath2, filepath3)
'/path/of/dir/dir2/path/of/file'
```

list_dir_or_file(*dir_path*, *list_dir=True*, *list_file=True*, *suffix=None*, *recursive=False*)

Scan a directory to find the interested directories or files in arbitrary order.

注解: *list_dir_or_file()* returns the path relative to `dir_path`.

参数

- **dir_path** (*str or Path*) –Path of the directory.
- **list_dir** (*bool*) –List the directories. Defaults to True.
- **list_file** (*bool*) –List the path of files. Defaults to True.
- **suffix** (*str or tuple[str], optional*) –File suffix that we are interested in. Defaults to None.

- **recursive** (*bool*) –If set to True, recursively scan the directory. Defaults to False.

生成器 *Iterable[str]* –A relative path to `dir_path`.

返回类型 Iterator[str]

实际案例

```
>>> backend = LocalBackend()
>>> dir_path = '/path/of/dir'
>>> # list those files and directories in current directory
>>> for file_path in backend.list_dir_or_file(dir_path):
...     print(file_path)
>>> # only list files
>>> for file_path in backend.list_dir_or_file(dir_path, list_dir=False):
...     print(file_path)
>>> # only list directories
>>> for file_path in backend.list_dir_or_file(dir_path, list_file=False):
...     print(file_path)
>>> # only list files ending with specified suffixes
>>> for file_path in backend.list_dir_or_file(dir_path, suffix='.txt'):
...     print(file_path)
>>> # list all files and directory recursively
>>> for file_path in backend.list_dir_or_file(dir_path, recursive=True):
...     print(file_path)
```

put(*obj*, *filepath*)

Write bytes to a given `filepath` with ‘wb’ mode.

注解： `put` will create a directory if the directory of `filepath` does not exist.

参数

- **obj** (*bytes*) –Data to be written.
- **filepath** (*str or Path*) –Path to write data.

返回类型 None

实际案例

```
>>> backend = LocalBackend()
>>> filepath = '/path/of/file'
>>> backend.put(b'hello world', filepath)
```

put_text(*obj*, *filepath*, *encoding='utf-8'*)

Write text to a given `filepath` with ‘w’ mode.

注解: `put_text` will create a directory if the directory of `filepath` does not exist.

参数

- **obj** (*str*) –Data to be written.
- **filepath** (*str or Path*) –Path to write data.
- **encoding** (*str*) –The encoding format used to open the `filepath`. Defaults to ‘utf-8’ .

返回类型 None

实际案例

```
>>> backend = LocalBackend()
>>> filepath = '/path/of/file'
>>> backend.put_text('hello world', filepath)
```

remove(*filepath*)

Remove a file.

参数 **filepath** (*str or Path*) –Path to be removed.

引发

- **IsADirectoryError** –If filepath is a directory, an IsADirectoryError will be raised.
- **FileNotFoundError** –If filepath does not exist, an FileNotFoundError will be raised.

返回类型 None

实际案例

```
>>> backend = LocalBackend()
>>> filepath = '/path/of/file'
>>> backend.remove(filepath)
```

rmtree(*dir_path*)

Recursively delete a directory tree.

参数 **dir_path** (*str or Path*) –A directory to be removed.

返回类型 None

实际案例

```
>>> dir_path = '/path/of/dir'
>>> backend.rmtree(dir_path)
```

49.1.5 HTTPBackend

class mmengine.fileio.**HTTPBackend**

HTTP and HTTPS storage bachend.

get(*filepath*)

Read bytes from a given `filepath`.

参数 **filepath** (*str*) –Path to read data.

返回 Expected bytes object.

返回类型 bytes

实际案例

```
>>> backend = HTTPBackend()
>>> backend.get('http://path/of/file')
b'hello world'
```

get_local_path(*filepath*)

Download a file from `filepath` to a local temporary directory, and return the temporary path.

`get_local_path` is decorated by `contxtlib.contextmanager()`. It can be called with `with` statement, and when exists from the `with` statement, the temporary path will be released.

参数 **filepath** (*str*) –Download a file from `filepath`.

生成器 *Iterable[str]* –Only yield one temporary path.

返回类型 Generator[Union[str, pathlib.Path], None, None]

实际案例

```
>>> backend = HTTPBackend()
>>> # After existing from the ``with`` clause,
>>> # the path will be removed
>>> with backend.get_local_path('http://path/of/file') as path:
...     # do something here
```

get_text(*filepath*, *encoding='utf-8'*)

Read text from a given `filepath`.

参数

- **filepath** (*str*) –Path to read data.
- **encoding** (*str*) –The encoding format used to open the `filepath`. Defaults to 'utf-8'.

返回 Expected text reading from `filepath`.

返回类型 str

实际案例

```
>>> backend = HTTPBackend()
>>> backend.get_text('http://path/of/file')
'hello world'
```

49.1.6 LmdbBackend

class mmengine.fileio.**LmdbBackend**(*db_path*, *readonly=True*, *lock=False*, *readahead=False*, ***kwargs*)

Lmdb storage backend.

参数

- **db_path** (*str*) –Lmdb database path.
- **readonly** (*bool*) –Lmdb environment parameter. If True, disallow any write operations. Defaults to True.
- **lock** (*bool*) –Lmdb environment parameter. If False, when concurrent access occurs, do not lock the database. Defaults to False.

- **readahead** (*bool*) –Lmdb environment parameter. If False, disable the OS filesystem readahead mechanism, which may improve random read performance when a database is larger than RAM. Defaults to False.
- ****kwargs** –Keyword arguments passed to *lmdb.open*.

db_path

Lmdb database path.

Type str

get(*filepath*)

Get values according to the filepath.

参数 **filepath** (*str or Path*) –Here, filepath is the lmdb key.

返回 Expected bytes object.

返回类型 bytes

实际案例

```
>>> backend = LmdbBackend('path/to/lmdb')
>>> backend.get('key')
b'hello world'
```

49.1.7 MemcachedBackend

class mmengine.fileio.**MemcachedBackend**(*server_list_cfg*, *client_cfg*, *sys_path=None*)

Memcached storage backend.

server_list_cfg

Config file for memcached server list.

Type str

client_cfg

Config file for memcached client.

Type str

sys_path

Additional path to be appended to *sys.path*. Defaults to None.

Type str, optional

get(*filepath*)

Get values according to the filepath.

参数 **filepath** (*str or Path*) –Path to read data.

返回 Expected bytes object.

返回类型 bytes

实际案例

```
>>> server_list_cfg = '/path/of/server_list.conf'
>>> client_cfg = '/path/of/mc.conf'
>>> backend = MemcachedBackend(server_list_cfg, client_cfg)
>>> backend.get('/path/of/file')
b'hello world'
```

49.1.8 PetrelBackend

class mmengine.fileio.**PetrelBackend**(*path_mapping=None*, *enable_mc=True*)

Petrel storage backend (for internal usage).

PetrelBackend supports reading and writing data to multiple clusters. If the file path contains the cluster name, PetrelBackend will read data from specified cluster or write data to it. Otherwise, PetrelBackend will access the default cluster.

参数

- **path_mapping** (*dict, optional*) –Path mapping dict from local path to Petrel path. When `path_mapping={'src': 'dst'}`, `src` in `filepath` will be replaced by `dst`. Defaults to None.
- **enable_mc** (*bool, optional*) –Whether to enable memcached support. Defaults to True.

实际案例

```
>>> backend = PetrelBackend()
>>> filepath1 = 'petrel://path/of/file'
>>> filepath2 = 'cluster-name:petrel://path/of/file'
>>> backend.get(filepath1)  # get data from default cluster
>>> client.get(filepath2)  # get data from 'cluster-name' cluster
```

copy_if_symlink_fails(*src*, *dst*)

Create a symbolic link pointing to src named dst.

Directly copy src to dst because PetrelBacekend does not support create a symbolic link.

参数

- **src** (*str or Path*) –A file or directory to be copied.

- **dst** (*str or Path*) –Copy a file or directory to dst.
- **backend_args** (*dict, optional*) –Arguments to instantiate the preifx of uri corresponding backend. Defaults to None.

返回 Return False because PetrelBackend does not support create a symbolic link.

返回类型 bool

实际案例

```
>>> backend = PetrelBackend()
>>> src = 'petrel://path/of/file'
>>> dst = 'petrel://path/of/your/file'
>>> backend.copy_if_symlink_fails(src, dst)
False
>>> src = 'petrel://path/of/dir'
>>> dst = 'petrel://path/of/your/dir'
>>> backend.copy_if_symlink_fails(src, dst)
False
```

copyfile(*src*, *dst*)

Copy a file src to dst and return the destination file.

src and dst should have the same prefix. If dst specifies a directory, the file will be copied into dst using the base filename from src. If dst specifies a file that already exists, it will be replaced.

参数

- **src** (*str or Path*) –A file to be copied.
- **dst** (*str or Path*) –Copy file to dst.

返回 The destination file.

返回类型 str

引发 **SameFileError** –If src and dst are the same file, a SameFileError will be raised.

实际案例

```
>>> backend = PetrelBackend()
>>> # dst is a file
>>> src = 'petrel://path/of/file'
>>> dst = 'petrel://path/of/file1'
>>> backend.copyfile(src, dst)
'petrel://path/of/file1'
```

```
>>> # dst is a directory
>>> dst = 'petrel://path/of/dir'
>>> backend.copyfile(src, dst)
'petrel://path/of/dir/file'
```

copyfile_from_local(*src*, *dst*)

Upload a local file src to dst and return the destination file.

参数

- **src** (*str or Path*) –A local file to be copied.
- **dst** (*str or Path*) –Copy file to dst.
- **backend_args** (*dict, optional*) –Arguments to instantiate the preifx of uri corresponding backend. Defaults to None.

返回 If dst specifies a directory, the file will be copied into dst using the base filename from src.

返回类型 str

实际案例

```
>>> backend = PetrelBackend()
>>> # dst is a file
>>> src = 'path/of/your/file'
>>> dst = 'petrel://path/of/file1'
>>> backend.copyfile_from_local(src, dst)
'petrel://path/of/file1'
```

```
>>> # dst is a directory
>>> dst = 'petrel://path/of/dir'
>>> backend.copyfile_from_local(src, dst)
'petrel://path/of/dir/file'
```

copyfile_to_local(*src*, *dst*)

Copy the file src to local dst and return the destination file.

If dst specifies a directory, the file will be copied into dst using the base filename from src. If dst specifies a file that already exists, it will be replaced.

参数

- **src** (*str or Path*) –A file to be copied.
- **dst** (*str or Path*) –Copy file to to local dst.

返回 If dst specifies a directory, the file will be copied into dst using the base filename from src.

返回类型 str

实际案例

```
>>> backend = PetrelBackend()
>>> # dst is a file
>>> src = 'petrel://path/of/file'
>>> dst = 'path/of/your/file'
>>> backend.copyfile_to_local(src, dst)
'path/of/your/file'
```

```
>>> # dst is a directory
>>> dst = 'path/of/your/dir'
>>> backend.copyfile_to_local(src, dst)
'path/of/your/dir/file'
```

copytree(*src*, *dst*)

Recursively copy an entire directory tree rooted at src to a directory named dst and return the destination directory.

src and dst should have the same prefix.

参数

- **src** (*str or Path*) –A directory to be copied.
- **dst** (*str or Path*) –Copy directory to dst.
- **backend_args** (*dict, optional*) –Arguments to instantiate the preifx of uri corresponding backend. Defaults to None.

返回 The destination directory.

返回类型 str

引发 **FileExistsError** –If dst had already existed, a FileExistsError will be raised.

实际案例

```
>>> backend = PetrelBackend()
>>> src = 'petrel://path/of/dir'
>>> dst = 'petrel://path/of/dir1'
>>> backend.copytree(src, dst)
'petrel://path/of/dir1'
```

copytree_from_local(*src*, *dst*)

Recursively copy an entire directory tree rooted at src to a directory named dst and return the destination directory.

参数

- **src** (*str or Path*) –A local directory to be copied.
- **dst** (*str or Path*) –Copy directory to dst.

返回 The destination directory.

返回类型 str

引发 **FileExistsError** –If dst had already existed, a FileExistsError will be raised.

实际案例

```
>>> backend = PetrelBackend()
>>> src = 'path/of/your/dir'
>>> dst = 'petrel://path/of/dir1'
>>> backend.copytree_from_local(src, dst)
'petrel://path/of/dir1'
```

copytree_to_local(*src*, *dst*)

Recursively copy an entire directory tree rooted at src to a local directory named dst and return the destination directory.

参数

- **src** (*str or Path*) –A directory to be copied.
- **dst** (*str or Path*) –Copy directory to local dst.
- **backend_args** (*dict, optional*) –Arguments to instantiate the preifx of uri corresponding backend. Defaults to None.

返回 The destination directory.

返回类型 str

实际案例

```
>>> backend = PetrelBackend()
>>> src = 'petrel://path/of/dir'
>>> dst = 'path/of/your/dir'
>>> backend.copytree_to_local(src, dst)
'path/of/your/dir'
```

exists(*filepath*)

Check whether a file path exists.

参数 **filepath** (*str or Path*) –Path to be checked whether exists.

返回 Return `True` if `filepath` exists, `False` otherwise.

返回类型 bool

实际案例

```
>>> backend = PetrelBackend()
>>> filepath = 'petrel://path/of/file'
>>> backend.exists(filepath)
True
```

generate_presigned_url(*url*, *client_method='get_object'*, *expires_in=3600*)

Generate the presigned url of video stream which can be passed to mmcv.VideoReader. Now only work on Petrel backend.

注解: Now only work on Petrel backend.

参数

- **url** (*str*) –Url of video stream.
- **client_method** (*str*) –Method of client, 'get_object' or 'put_object'. Default: 'get_object'.
- **expires_in** (*int*) –expires, in seconds. Default: 3600.

返回 Generated presigned url.

返回类型 str

get(*filepath*)

Read bytes from a given `filepath` with 'rb' mode.

参数 **filepath** (*str or Path*) –Path to read data.

返回 Return bytes read from filepath.

返回类型 bytes

实际案例

```
>>> backend = PetrelBackend()
>>> filepath = 'petrel://path/of/file'
>>> backend.get(filepath)
b'hello world'
```

get_local_path(*filepath*)

Download a file from `filepath` to a local temporary directory, and return the temporary path.

`get_local_path` is decorated by `contxtlib.contextmanager()`. It can be called with `with` statement, and when exists from the `with` statement, the temporary path will be released.

参数 **filepath** (*str or Path*) –Download a file from `filepath`.

生成器 *Iterable[str]* –Only yield one temporary path.

返回类型 Generator[Union[str, pathlib.Path], None, None]

实际案例

```
>>> backend = PetrelBackend()
>>> # After existing from the ``with`` clause,
>>> # the path will be removed
>>> filepath = 'petrel://path/of/file'
>>> with backend.get_local_path(filepath) as path:
...     # do something here
```

get_text(*filepath*, *encoding='utf-8'*)

Read text from a given `filepath` with ‘r’ mode.

参数

- **filepath** (*str or Path*) –Path to read data.
- **encoding** (*str*) –The encoding format used to open the `filepath`. Defaults to ‘utf-8’.

返回 Expected text reading from `filepath`.

返回类型 str

实际案例

```
>>> backend = PetrelBackend()
>>> filepath = 'petrel://path/of/file'
>>> backend.get_text(filepath)
'hello world'
```

isdir(*filepath*)

Check whether a file path is a directory.

参数 **filepath** (*str or Path*) –Path to be checked whether it is a directory.

返回 Return `True` if `filepath` points to a directory, `False` otherwise.

返回类型 bool

实际案例

```
>>> backend = PetrelBackend()
>>> filepath = 'petrel://path/of/dir'
>>> backend.isdir(filepath)
True
```

isfile(*filepath*)

Check whether a file path is a file.

参数 **filepath** (*str or Path*) –Path to be checked whether it is a file.

返回 Return `True` if `filepath` points to a file, `False` otherwise.

返回类型 bool

实际案例

```
>>> backend = PetrelBackend()
>>> filepath = 'petrel://path/of/file'
>>> backend.isfile(filepath)
True
```

join_path(*filepath*, **filepaths*)

Concatenate all file paths.

Join one or more filepath components intelligently. The return value is the concatenation of filepath and any members of *filepaths.

参数

- **filepath** (*str or Path*) –Path to be concatenated.
- **filepaths** (*Union[str, pathlib.Path]*) –

返回 The result after concatenation.

返回类型 str

实际案例

```
>>> backend = PetrelBackend()
>>> filepath = 'petrel://path/of/file'
>>> backend.join_path(filepath, 'another/path')
'petrel://path/of/file/another/path'
>>> backend.join_path(filepath, '/another/path')
'petrel://path/of/file/another/path'
```

list_dir_or_file(*dir_path*, *list_dir=True*, *list_file=True*, *suffix=None*, *recursive=False*)

Scan a directory to find the interested directories or files in arbitrary order.

注解： Petrel has no concept of directories but it simulates the directory hierarchy in the filesystem through public prefixes. In addition, if the returned path ends with '/', it means the path is a public prefix which is a logical directory.

注解： *list_dir_or_file()* returns the path relative to `dir_path`. In addition, the returned path of directory will not contains the suffix '/' which is consistent with other backends.

参数

- **dir_path** (*str | Path*) –Path of the directory.
- **list_dir** (*bool*) –List the directories. Defaults to True.
- **list_file** (*bool*) –List the path of files. Defaults to True.
- **suffix** (*str or tuple[str], optional*) –File suffix that we are interested in. Defaults to None.
- **recursive** (*bool*) –If set to True, recursively scan the directory. Defaults to False.

生成器 *Iterable[str]* –A relative path to `dir_path`.

返回类型 Iterator[str]

实际案例

```
>>> backend = PetrelBackend()
>>> dir_path = 'petrel://path/of/dir'
>>> # list those files and directories in current directory
>>> for file_path in backend.list_dir_or_file(dir_path):
...     print(file_path)
>>> # only list files
>>> for file_path in backend.list_dir_or_file(dir_path, list_dir=False):
...     print(file_path)
>>> # only list directories
>>> for file_path in backend.list_dir_or_file(dir_path, list_file=False):
...     print(file_path)
>>> # only list files ending with specified suffixes
>>> for file_path in backend.list_dir_or_file(dir_path, suffix='.txt'):
...     print(file_path)
>>> # list all files and directory recursively
>>> for file_path in backend.list_dir_or_file(dir_path, recursive=True):
...     print(file_path)
```

put(*obj*, *filepath*)

Write bytes to a given `filepath`.

参数

- **obj** (*bytes*) –Data to be saved.
- **filepath** (*str or Path*) –Path to write data.

返回类型 None

实际案例

```
>>> backend = PetrelBackend()
>>> filepath = 'petrel://path/of/file'
>>> backend.put(b'hello world', filepath)
```

put_text(*obj*, *filepath*, *encoding='utf-8'*)

Write text to a given `filepath`.

参数

- **obj** (*str*) –Data to be written.
- **filepath** (*str or Path*) –Path to write data.
- **encoding** (*str*) –The encoding format used to encode the `obj`. Defaults to 'utf-8'.

返回类型 None

实际案例

```
>>> backend = PetrelBackend()
>>> filepath = 'petrel://path/of/file'
>>> backend.put_text('hello world', filepath)
```

remove(*filepath*)

Remove a file.

参数 **filepath** (*str or Path*) –Path to be removed.

引发

- **FileNotFoundError** –If filepath does not exist, an FileNotFoundError will be raised.
- **IsADirectoryError** –If filepath is a directory, an IsADirectoryError will be raised.

返回类型 None

实际案例

```
>>> backend = PetrelBackend()
>>> filepath = 'petrel://path/of/file'
>>> backend.remove(filepath)
```

rmtree(*dir_path*)

Recursively delete a directory tree.

参数 **dir_path** (*str or Path*) –A directory to be removed.

返回类型 None

实际案例

```
>>> backend = PetrelBackend()
>>> dir_path = 'petrel://path/of/dir'
>>> backend.rmtree(dir_path)
```

register_backend	Register a backend.

49.1.9 mmengine.fileio.register_backend

mmengine.fileio.**register_backend**(*name*, *backend=None*, *force=False*, *prefixes=None*)

Register a backend.

参数

- **name** (*str*) –The name of the registered backend.
- **backend** (*class, optional*) –The backend class to be registered, which must be a subclass of *BaseStorageBackend*. When this method is used as a decorator, backend is None. Defaults to None.
- **force** (*bool*) –Whether to override the backend if the name has already been registered. Defaults to False.
- **prefixes** (*str or list[str] or tuple[str], optional*) –The prefix of the registered storage backend. Defaults to None.

This method can be used as a normal method or a decorator.

实际案例

```
>>> class NewBackend(BaseStorageBackend):
...     def get(self, filepath):
...         return filepath
...
...     def get_text(self, filepath):
...         return filepath
>>> register_backend('new', NewBackend)
```

```
>>> @register_backend('new')
... class NewBackend(BaseStorageBackend):
...     def get(self, filepath):
...         return filepath
...
...     def get_text(self, filepath):
...         return filepath
```

49.2 File Handler

BaseFileHandler
JsonHandler
PickleHandler
YamlHandler

49.2.1 BaseFileHandler

class mmengine.fileio.**BaseFileHandler**

49.2.2 JsonHandler

class mmengine.fileio.**JsonHandler**

49.2.3 PickleHandler

class mmengine.fileio.**PickleHandler**

49.2.4 YamlHandler

class mmengine.fileio.**YamlHandler**

register_handler

49.2.5 mmengine.fileio.register_handler

mmengine.fileio.**register_handler**(*file_formats*, ***kwargs*)

49.3 File IO

dump	Dump data to json/yaml/pickle strings or files.
load	Load data from json/yaml/pickle files.
copy_if_symlink_fails	Create a symbolic link pointing to src named dst.
copyfile	Copy a file src to dst and return the destination file.
copyfile_from_local	Copy a local file src to dst and return the destination file.
copyfile_to_local	Copy the file src to local dst and return the destination file.
copytree	Recursively copy an entire directory tree rooted at src to a directory named dst and return the destination directory.
copytree_from_local	Recursively copy an entire directory tree rooted at src to a directory named dst and return the destination directory.
copytree_to_local	Recursively copy an entire directory tree rooted at src to a local directory named dst and return the destination directory.
exists	Check whether a file path exists.
generate_presigned_url	Generate the presigned url of video stream which can be passed to mmcv.VideoReader.
get	Read bytes from a given `filepath` with 'rb' mode.
get_file_backend	Return a file backend based on the prefix of uri or backend_args.
get_local_path	Download data from `filepath` and write the data to local path.
get_text	Read text from a given `filepath` with 'r' mode.
isdir	Check whether a file path is a directory.
isfile	Check whether a file path is a file.
join_path	Concatenate all file paths.
list_dir_or_file	Scan a directory to find the interested directories or files in arbitrary order.
put	Write bytes to a given `filepath` with 'wb' mode.
put_text	Write text to a given `filepath` with 'w' mode.
remove	Remove a file.
rmtree	Recursively delete a directory tree.

49.3.1 mmengine.fileio.dump

mmengine.fileio.**dump**(*obj*, *file=None*, *file_format=None*, *file_client_args=None*, *backend_args=None*, ***kwargs*)

Dump data to json/yaml/pickle strings or files.

This method provides a unified api for dumping data as strings or to files, and also supports custom arguments for each file format.

dump supports dumping data as strings or to files which is saved to different backends.

参数

- **obj** (*any*) –The python object to be dumped.
- **file** (str or Path or file-like object, optional) –If not specified, then the object is dumped to a str, otherwise to a file specified by the filename or file-like object.
- **file_format** (*str, optional*) –Same as *load()*.
- **file_client_args** (*dict, optional*) –Arguments to instantiate a FileClient. See *mmengine.fileio.FileClient* for details. Defaults to None. It will be deprecated in future. Please use backend_args instead.
- **backend_args** (*dict, optional*) –Arguments to instantiate the preifx of uri corresponding backend. Defaults to None. New in v0.2.0.

实际案例

```
>>> dump('hello world', '/path/of/your/file')  # disk
>>> dump('hello world', 's3://path/of/your/file')  # ceph or petrel
```

返回 True for success, False otherwise.

返回类型 bool

49.3.2 mmengine.fileio.load

mmengine.fileio.**load**(*file*, *file_format=None*, *file_client_args=None*, *backend_args=None*, ***kwargs*)

Load data from json/yaml/pickle files.

This method provides a unified api for loading data from serialized files.

load supports loading data from serialized files those can be storaged in different backends.

参数

- **file** (str or Path or file-like object) –Filename or a file-like object.

- **file_format** (*str, optional*) –If not specified, the file format will be inferred from the file extension, otherwise use the specified one. Currently supported formats include "json", "yaml/yml" and "pickle/pkl".
- **file_client_args** (*dict, optional*) –Arguments to instantiate a FileClient. See *mmengine.fileio.FileClient* for details. Defaults to None. It will be deprecated in future. Please use `backend_args` instead.
- **backend_args** (*dict, optional*) –Arguments to instantiate the preifx of uri corresponding backend. Defaults to None. New in v0.2.0.

实际案例

```
>>> load('/path/of/your/file')  # file is storaged in disk
>>> load('https://path/of/your/file')  # file is storaged in Internet
>>> load('s3://path/of/your/file')  # file is storaged in petrel
```

返回 The content from the file.

49.3.3 mmengine.fileio.copy_if_symlink_fails

mmengine.fileio.**copy_if_symlink_fails**(*src*, *dst*, *backend_args=None*)

Create a symbolic link pointing to src named dst.

If failed to create a symbolic link pointing to src, directory copy src to dst instead.

参数

- **src** (*str or Path*) –Create a symbolic link pointing to src.
- **dst** (*str or Path*) –Create a symbolic link named dst.
- **backend_args** (*dict, optional*) –Arguments to instantiate the corresponding backend. Defaults to None.

返回 Return True if successfully create a symbolic link pointing to src. Otherwise, return False.

返回类型 bool

实际案例

```
>>> src = '/path/of/file'
>>> dst = '/path1/of/file1'
>>> copy_if_symlink_fails(src, dst)
True
>>> src = '/path/of/dir'
>>> dst = '/path1/of/dir1'
>>> copy_if_symlink_fails(src, dst)
True
```

49.3.4 mmengine.fileio.copyfile

mmengine.fileio.**copyfile**(*src*, *dst*, *backend_args=None*)

Copy a file src to dst and return the destination file.

src and dst should have the same prefix. If dst specifies a directory, the file will be copied into dst using the base filename from src. If dst specifies a file that already exists, it will be replaced.

参数

- **src** (*str or Path*) –A file to be copied.
- **dst** (*str or Path*) –Copy file to dst.
- **backend_args** (*dict, optional*) –Arguments to instantiate the corresponding backend. Defaults to None.

返回 The destination file.

返回类型 str

引发 **SameFileError** –If src and dst are the same file, a SameFileError will be raised.

实际案例

```
>>> # dst is a file
>>> src = '/path/of/file'
>>> dst = '/path1/of/file1'
>>> # src will be copied to '/path1/of/file1'
>>> copyfile(src, dst)
'/path1/of/file1'
```

```
>>> # dst is a directory
>>> dst = '/path1/of/dir'
```

(下页继续)

(续上页)

```
>>> # src will be copied to '/path1/of/dir/file'
>>> copyfile(src, dst)
'/path1/of/dir/file'
```

49.3.5 mmengine.fileio.copyfile_from_local

mmengine.fileio.**copyfile_from_local**(*src*, *dst*, *backend_args=None*)

Copy a local file src to dst and return the destination file.

注解： If the backend is the instance of LocalBackend, it does the same thing with *copyfile()*.

参数

- **src** (*str or Path*) –A local file to be copied.
- **dst** (*str or Path*) –Copy file to dst.
- **backend_args** (*dict, optional*) –Arguments to instantiate the corresponding backend. Defaults to None.

返回 If dst specifies a directory, the file will be copied into dst using the base filename from src.

返回类型 str

实际案例

```
>>> # dst is a file
>>> src = '/path/of/file'
>>> dst = 's3://openmmlab/mmengine/file1'
>>> # src will be copied to 's3://openmmlab/mmengine/file1'
>>> copyfile_from_local(src, dst)
s3://openmmlab/mmengine/file1
```

```
>>> # dst is a directory
>>> dst = 's3://openmmlab/mmengine'
>>> # src will be copied to 's3://openmmlab/mmengine/file''
>>> copyfile_from_local(src, dst)
's3://openmmlab/mmengine/file'
```

49.3.6 mmengine.fileio.copyfile_to_local

mmengine.fileio.**copyfile_to_local**(*src*, *dst*, *backend_args=None*)

Copy the file src to local dst and return the destination file.

If dst specifies a directory, the file will be copied into dst using the base filename from src. If dst specifies a file that already exists, it will be replaced.

注解： If the backend is the instance of LocalBackend, it does the same thing with *copyfile()*.

参数

- **src** (*str or Path*) –A file to be copied.
- **dst** (*str or Path*) –Copy file to to local dst.
- **backend_args** (*dict, optional*) –Arguments to instantiate the corresponding backend. Defaults to None.

返回 If dst specifies a directory, the file will be copied into dst using the base filename from src.

返回类型 str

实际案例

```
>>> # dst is a file
>>> src = 's3://openmmlab/mmengine/file'
>>> dst = '/path/of/file'
>>> # src will be copied to '/path/of/file'
>>> copyfile_to_local(src, dst)
'/path/of/file'
```

```
>>> # dst is a directory
>>> dst = '/path/of/dir'
>>> # src will be copied to '/path/of/dir/file'
>>> copyfile_to_local(src, dst)
'/path/of/dir/file'
```

49.3.7 mmengine.fileio.copytree

mmengine.fileio.**copytree**(*src*, *dst*, *backend_args=None*)

Recursively copy an entire directory tree rooted at src to a directory named dst and return the destination directory.

src and dst should have the same prefix and dst must not already exist.

参数

- **src** (*str or Path*) –A directory to be copied.
- **dst** (*str or Path*) –Copy directory to dst.
- **backend_args** (*dict, optional*) –Arguments to instantiate the corresponding backend. Defaults to None.

返回 The destination directory.

返回类型 str

引发 **FileExistsError** –If dst had already existed, a FileExistsError will be raised.

实际案例

```
>>> src = '/path/of/dir1'
>>> dst = '/path/of/dir2'
>>> copytree(src, dst)
'/path/of/dir2'
```

49.3.8 mmengine.fileio.copytree_from_local

mmengine.fileio.**copytree_from_local**(*src*, *dst*, *backend_args=None*)

Recursively copy an entire directory tree rooted at src to a directory named dst and return the destination directory.

注解: If the backend is the instance of LocalBackend, it does the same thing with *copytree()*.

参数

- **src** (*str or Path*) –A local directory to be copied.
- **dst** (*str or Path*) –Copy directory to dst.
- **backend_args** (*dict, optional*) –Arguments to instantiate the corresponding backend. Defaults to None.

返回 The destination directory.

返回类型 str

实际案例

```
>>> src = '/path/of/dir'
>>> dst = 's3://openmmlab/mmengine/dir'
>>> copyfile_from_local(src, dst)
's3://openmmlab/mmengine/dir'
```

49.3.9 mmengine.fileio.copytree_to_local

mmengine.fileio.**copytree_to_local**(*src*, *dst*, *backend_args=None*)

Recursively copy an entire directory tree rooted at src to a local directory named dst and return the destination directory.

注解： If the backend is the instance of LocalBackend, it does the same thing with *copytree()*.

参数

- **src** (*str or Path*) –A directory to be copied.
- **dst** (*str or Path*) –Copy directory to local dst.
- **backend_args** (*dict, optional*) –Arguments to instantiate the corresponding backend. Defaults to None.

返回 The destination directory.

返回类型 str

实际案例

```
>>> src = 's3://openmmlab/mmengine/dir'
>>> dst = '/path/of/dir'
>>> copytree_to_local(src, dst)
'/path/of/dir'
```

49.3.10 mmengine.fileio.exists

mmengine.fileio.**exists**(*filepath*, *backend_args=None*)

Check whether a file path exists.

参数

- **filepath** (*str or Path*) –Path to be checked whether exists.
- **backend_args** (*dict, optional*) –Arguments to instantiate the corresponding backend. Defaults to None.

返回 Return `True` if `filepath` exists, `False` otherwise.

返回类型 bool

实际案例

```
>>> filepath = '/path/of/file'
>>> exists(filepath)
True
```

49.3.11 mmengine.fileio.generate_presigned_url

mmengine.fileio.**generate_presigned_url**(*url*, *client_method='get_object'*, *expires_in=3600*, *backend_args=None*)

Generate the presigned url of video stream which can be passed to mmcv.VideoReader. Now only work on Petrel backend.

注解: Now only work on Petrel backend.

参数

- **url** (*str*) –Url of video stream.
- **client_method** (*str*) –Method of client, 'get_object' or 'put_object'. Default: 'get_object'.
- **expires_in** (*int*) –expires, in seconds. Default: 3600.
- **backend_args** (*dict, optional*) –Arguments to instantiate the corresponding backend. Defaults to None.

返回 Generated presigned url.

返回类型 str

49.3.12 mmengine.fileio.get

mmengine.fileio.**get**(*filepath*, *backend_args=None*)

Read bytes from a given `filepath` with 'rb' mode.

参数

- **filepath** (*str or Path*) –Path to read data.
- **backend_args** (*dict, optional*) –Arguments to instantiate the corresponding backend. Defaults to None.

返回 Expected bytes object.

返回类型 bytes

实际案例

```
>>> filepath = '/path/of/file'
>>> get(filepath)
b'hello world'
```

49.3.13 mmengine.fileio.get_file_backend

mmengine.fileio.**get_file_backend**(*uri=None*, *, *backend_args=None*, *enable_singleton=False*)

Return a file backend based on the prefix of uri or backend_args.

参数

- **uri** (*str or Path*) –Uri to be parsed that contains the file prefix.
- **backend_args** (*dict, optional*) –Arguments to instantiate the corresponding backend. Defaults to None.
- **enable_singleton** (*bool*) –Whether to enable the singleton pattern. If it is True, the backend created will be reused if the signature is same with the previous one. Defaults to False.

返回 Instantiated Backend object.

返回类型 *BaseStorageBackend*

实际案例

```
>>> # get file backend based on the prefix of uri
>>> uri = 's3://path/of/your/file'
>>> backend = get_file_backend(uri)
>>> # get file backend based on the backend_args
>>> backend = get_file_backend(backend_args={'backend': 'petrel'})
>>> # backend name has a higher priority if 'backend' in backend_args
>>> backend = get_file_backend(uri, backend_args={'backend': 'petrel'})
```

49.3.14 mmengine.fileio.get_local_path

mmengine.fileio.**get_local_path**(*filepath*, *backend_args=None*)

Download data from `filepath` and write the data to local path.

`get_local_path` is decorated by `contxtlib.contextmanager()`. It can be called with `with` statement, and when exists from the `with` statement, the temporary path will be released.

注解： If the `filepath` is a local path, just return itself and it will not be released (removed).

参数

- **filepath** (*str or Path*) –Path to be read data.
- **backend_args** (*dict, optional*) –Arguments to instantiate the corresponding backend. Defaults to None.

生成器 *Iterable[str]* –Only yield one path.

返回类型 Generator[Union[str, pathlib.Path], None, None]

实际案例

```
>>> with get_local_path('s3://bucket/abc.jpg') as path:
...     # do something here
```

49.3.15 mmengine.fileio.get_text

mmengine.fileio.**get_text**(*filepath*, *encoding='utf-8'*, *backend_args=None*)

Read text from a given `filepath` with ‘r’ mode.

参数

- **filepath** (*str or Path*) –Path to read data.
- **encoding** (*str*) –The encoding format used to open the `filepath`. Defaults to ‘utf-8’.
- **backend_args** (*dict, optional*) –Arguments to instantiate the corresponding backend. Defaults to None.

返回 Expected text reading from `filepath`.

返回类型 str

实际案例

```
>>> filepath = '/path/of/file'
>>> get_text(filepath)
'hello world'
```

49.3.16 mmengine.fileio.isdir

mmengine.fileio.**isdir**(*filepath*, *backend_args=None*)

Check whether a file path is a directory.

参数

- **filepath** (*str or Path*) –Path to be checked whether it is a directory.
- **backend_args** (*dict, optional*) –Arguments to instantiate the corresponding backend. Defaults to None.

返回 Return `True` if `filepath` points to a directory, `False` otherwise.

返回类型 bool

实际案例

```
>>> filepath = '/path/of/dir'
>>> isdir(filepath)
True
```

49.3.17 mmengine.fileio.isfile

mmengine.fileio.**isfile**(*filepath*, *backend_args=None*)

Check whether a file path is a file.

参数

- **filepath** (*str or Path*) –Path to be checked whether it is a file.
- **backend_args** (*dict, optional*) –Arguments to instantiate the corresponding backend. Defaults to None.

返回 Return `True` if `filepath` points to a file, `False` otherwise.

返回类型 bool

实际案例

```
>>> filepath = '/path/of/file'
>>> isfile(filepath)
True
```

49.3.18 mmengine.fileio.join_path

mmengine.fileio.**join_path**(*filepath*, **filepaths*, *backend_args=None*)

Concatenate all file paths.

Join one or more filepath components intelligently. The return value is the concatenation of filepath and any members of *filepaths.

参数

- **filepath** (*str or Path*) –Path to be concatenated.
- ***filepaths** (*str or Path*) –Other paths to be concatenated.
- **backend_args** (*dict, optional*) –Arguments to instantiate the corresponding backend. Defaults to None.
- **filepaths** (*Union[str, pathlib.Path]*) –

返回 The result of concatenation.

返回类型 str

实际案例

```
>>> filepath1 = '/path/of/dir1'
>>> filepath2 = 'dir2'
>>> filepath3 = 'path/of/file'
>>> join_path(filepath1, filepath2, filepath3)
'/path/of/dir/dir2/path/of/file'
```

49.3.19 mmengine.fileio.list_dir_or_file

mmengine.fileio.**list_dir_or_file**(*dir_path*, *list_dir=True*, *list_file=True*, *suffix=None*, *recursive=False*, *backend_args=None*)

Scan a directory to find the interested directories or files in arbitrary order.

注解: *list_dir_or_file()* returns the path relative to dir_path.

参数

- **dir_path** (*str or Path*) –Path of the directory.
- **list_dir** (*bool*) –List the directories. Defaults to True.
- **list_file** (*bool*) –List the path of files. Defaults to True.
- **suffix** (*str or tuple[str], optional*) –File suffix that we are interested in. Defaults to None.
- **recursive** (*bool*) –If set to True, recursively scan the directory. Defaults to False.
- **backend_args** (*dict, optional*) –Arguments to instantiate the corresponding backend. Defaults to None.

生成器 *Iterable[str]* –A relative path to dir_path.

返回类型 Iterator[str]

实际案例

```
>>> dir_path = '/path/of/dir'
>>> for file_path in list_dir_or_file(dir_path):
...     print(file_path)
>>> # list those files and directories in current directory
>>> for file_path in list_dir_or_file(dir_path):
...     print(file_path)
>>> # only list files
>>> for file_path in list_dir_or_file(dir_path, list_dir=False):
...     print(file_path)
>>> # only list directories
>>> for file_path in list_dir_or_file(dir_path, list_file=False):
...     print(file_path)
>>> # only list files ending with specified suffixes
>>> for file_path in list_dir_or_file(dir_path, suffix='.txt'):
...     print(file_path)
>>> # list all files and directory recursively
>>> for file_path in list_dir_or_file(dir_path, recursive=True):
...     print(file_path)
```

49.3.20 mmengine.fileio.put

mmengine.fileio.**put**(*obj*, *filepath*, *backend_args=None*)

Write bytes to a given `filepath` with ‘wb’ mode.

注解： `put` should create a directory if the directory of `filepath` does not exist.

参数

- **obj** (*bytes*) –Data to be written.
- **filepath** (*str or Path*) –Path to write data.
- **backend_args** (*dict, optional*) –Arguments to instantiate the corresponding backend. Defaults to None.

返回类型 None

实际案例

```
>>> filepath = '/path/of/file'
>>> put(b'hello world', filepath)
```

49.3.21 mmengine.fileio.put_text

mmengine.fileio.**put_text**(*obj*, *filepath*, *backend_args=None*)

Write text to a given `filepath` with ‘w’ mode.

注解: `put_text` should create a directory if the directory of `filepath` does not exist.

参数

- **obj** (*str*) –Data to be written.
- **filepath** (*str or Path*) –Path to write data.
- **encoding** (*str, optional*) –The encoding format used to open the `filepath`. Defaults to ‘utf-8’.
- **backend_args** (*dict, optional*) –Arguments to instantiate the corresponding backend. Defaults to None.

返回类型 None

实际案例

```
>>> filepath = '/path/of/file'
>>> put_text('hello world', filepath)
```

49.3.22 mmengine.fileio.remove

mmengine.fileio.**remove**(*filepath*, *backend_args=None*)

Remove a file.

参数

- **filepath** (*str, Path*) –Path to be removed.
- **backend_args** (*dict, optional*) –Arguments to instantiate the corresponding backend. Defaults to None.

引发

- **FileNotFoundError** –If filepath does not exist, an FileNotFoundError will be raised.
- **IsADirectoryError** –If filepath is a directory, an IsADirectoryError will be raised.

返回类型 None

实际案例

```
>>> filepath = '/path/of/file'
>>> remove(filepath)
```

49.3.23 mmengine.fileio.rmtree

mmengine.fileio.**rmtree**(*dir_path*, *backend_args=None*)

Recursively delete a directory tree.

参数

- **dir_path** (*str or Path*) –A directory to be removed.
- **backend_args** (*dict, optional*) –Arguments to instantiate the corresponding backend. Defaults to None.

返回类型 None

实际案例

```
>>> dir_path = '/path/of/dir'
>>> rmtree(dir_path)
```

49.4 Parse File

dict_from_file	Load a text file and parse the content as a dict.
list_from_file	Load a text file and parse the content as a list of strings.

49.4.1 mmengine.fileio.dict_from_file

mmengine.fileio.**dict_from_file**(*filename*, *key_type=<class 'str'>*, *encoding='utf-8'*, *file_client_args=None*, *backend_args=None*)

Load a text file and parse the content as a dict.

Each line of the text file will be two or more columns split by whitespaces or tabs. The first column will be parsed as dict keys, and the following columns will be parsed as dict values.

`dict_from_file` supports loading a text file which can be storaged in different backends and parsing the content as a dict.

参数

- **filename** (*str*) –Filename.
- **key_type** (*type*) –Type of the dict keys. str is user by default and type conversion will be performed if specified.
- **encoding** (*str*) –Encoding used to open the file. Defaults to utf-8.
- **file_client_args** (*dict, optional*) –Arguments to instantiate a FileClient. See *mmengine.fileio.FileClient* for details. Defaults to None. It will be deprecated in future. Please use `backend_args` instead.
- **backend_args** (*dict, optional*) –Arguments to instantiate the preifx of uri corresponding backend. Defaults to None. New in v0.2.0.

实际案例

```
>>> dict_from_file('/path/of/your/file')  # disk
{'key1': 'value1', 'key2': 'value2'}
>>> dict_from_file('s3://path/of/your/file')  # ceph or petrel
{'key1': 'value1', 'key2': 'value2'}
```

返回 The parsed contents.

返回类型 dict

49.4.2 mmengine.fileio.list_from_file

mmengine.fileio.**list_from_file**(*filename*, *prefix=''*, *offset=0*, *max_num=0*, *encoding='utf-8'*, *file_client_args=None*, *backend_args=None*)

Load a text file and parse the content as a list of strings.

`list_from_file` supports loading a text file which can be storaged in different backends and parsing the content as a list for strings.

参数

- **filename** (*str*) –Filename.
- **prefix** (*str*) –The prefix to be inserted to the beginning of each item.
- **offset** (*int*) –The offset of lines.
- **max_num** (*int*) –The maximum number of lines to be read, zeros and negatives mean no limitation.
- **encoding** (*str*) –Encoding used to open the file. Defaults to utf-8.
- **file_client_args** (*dict, optional*) –Arguments to instantiate a FileClient. See *mmengine.fileio.FileClient* for details. Defaults to None. It will be deprecated in future. Please use `backend_args` instead.
- **backend_args** (*dict, optional*) –Arguments to instantiate the preifx of uri corresponding backend. Defaults to None. New in v0.2.0.

实际案例

```
>>> list_from_file('/path/of/your/file')  # disk
['hello', 'world']
>>> list_from_file('s3://path/of/your/file')  # ceph or petrel
['hello', 'world']
```

返回 A list of strings.

返回类型 list[str]

CHAPTER 50

mmengine.dist

mmengine.dist

50.1 dist

gather	Gather data from the whole group to `dst` process.
gather_object	Gathers picklable objects from the whole group in a single process.
all_gather	Gather data from the whole group in a list.
all_gather_object	Gather picklable objects from the whole group into a list.
all_reduce	Reduces the tensor data across all machines in such a way that all get the final result.
all_reduce_dict	Reduces the dict across all machines in such a way that all get the final result.
all_reduce_params	All-reduce parameters.
broadcast	Broadcast the data from `src` process to the whole group.
sync_random_seed	Synchronize a random seed to all processes.

下页继续

表 1 – 续上页

broadcast_object_list	Broadcasts picklable objects in `object_list` to the whole group.
collect_results	Collected results in distributed environments.
collect_results_cpu	Collect results under cpu mode.
collect_results_gpu	Collect results under gpu mode.

50.1.1 mmengine.dist.gather

`mmengine.dist.`**`gather`**(*data*, *dst=0*, *group=None*)

Gather data from the whole group to `dst` process.

注解： Calling `gather` in non-distributed environment dose nothing and just returns a list containing `data` itself.

注解： `NCCL` backend does not support `gather`.

注解： Unlike PyTorch `torch.distributed.gather`, *`gather()`* in MMEngine does not pass in an empty list `gather_list` and returns the `gather_list` directly, which is more convenient. The difference between their interfaces is as below:

- MMEngine: gather(data, dst, group) -> gather_list
- PyTorch: gather(data, gather_list, dst, group) -> None

参数

- **`data`** (*`Tensor`*) –Tensor to be gathered. CUDA tensor is not supported.
- **`dst`** (*`int`*) –Destination rank. Defaults to 0.
- **`group`** (*`ProcessGroup, optional`*) –The process group to work on. If None, the default process group will be used. Defaults to None.

返回 `dst` process will get a list of tensor gathering from the whole group. Other process will get a empty list. If in non-distributed environment, just return a list containing `data` itself.

返回类型 list[Tensor]

实际案例

```
>>> import torch
>>> import mmengine.dist as dist
```

```
>>> # non-distributed environment
>>> data = torch.arange(2, dtype=torch.int64)
>>> data
tensor([0, 1])
>>> output = dist.gather(data)
>>> output
[tensor([0, 1])]
```

```
>>> # distributed environment
>>> # We have 2 process groups, 2 ranks.
>>> data = torch.arange(2, dtype=torch.int64) + 1 + 2 * rank
>>> data
tensor([1, 2]) # Rank 0
tensor([3, 4]) # Rank 1
>>> output = dist.gather(data)
>>> output
[tensor([1, 2]), tensor([3, 4])]  # Rank 0
[]  # Rank 1
```

50.1.2 mmengine.dist.gather_object

mmengine.dist.**gather_object**(*data*, *dst=0*, *group=None*)

Gathers picklable objects from the whole group in a single process. Similar to *gather()*, but Python objects can be passed in. Note that the object must be picklable in order to be gathered.

注解: NCCL backend does not support gather_object.

注解: Unlike PyTorch torch.distributed.gather_object, *gather_object()* in MMEngine does not pass in an empty list gather_list and returns the gather_list directly, which is more convenient. The difference between their interfaces is as below:

- MMEngine: gather_object(data, dst, group) -> gather_list
- PyTorch: gather_object(data, gather_list, data, group) -> None

---

参数

- **data** (*Any*) –Input object. Must be picklable.
- **dst** (*int*) –Destination rank. Defaults to 0.
- **group** (*Optional[torch.distributed.distributed_c10d.ProcessGroup]*) –(ProcessGroup, optional): The process group to work on. If None, the default process group will be used. Defaults to None.

返回 list[Any]. On the `dst` rank, return `gather_list` which contains the output of the collective.

返回类型 Optional[List[Any]]

实际案例

```
>>> import torch
>>> import mmengine.dist as dist
```

```
>>> # non-distributed environment
>>> data = ['foo', 12, {1: 2}]  # any picklable object
>>> gather_objects = dist.gather_object(data[dist.get_rank()])
>>> output
['foo']
```

```
>>> # distributed environment
>>> # We have 3 process groups, 3 ranks.
>>> dist.gather_object(gather_objects[dist.get_rank()], dst=0)
>>> output
['foo', 12, {1: 2}]  # Rank 0
None  # Rank 1
None  # Rank 2
```

50.1.3 mmengine.dist.all_gather

`mmengine.dist.`**`all_gather`**(*data*, *group=None*)

Gather data from the whole group in a list.

注解: Calling `all_gather` in non-distributed environment does nothing and just returns a list containing `data` itself.

注解: Unlike PyTorch `torch.distributed.all_gather`, *all_gather()* in MMEngine does not

pass in an empty list `gather_list` and returns the `gather_list` directly, which is more convenient. The difference between their interfaces is as below:

- MMEngine: all_gather(data, group) -> gather_list
- PyTorch: all_gather(gather_list, data, group) -> None

参数

- **data** (*Tensor*) –Tensor to be gathered.
- **group** (*ProcessGroup, optional*) –The process group to work on. If None, the default process group will be used. Defaults to None.

返回 Return a list containing data from the whole group if in distributed environment, otherwise a list only containing `data` itself.

返回类型 list[Tensor]

实际案例

```
>>> import torch
>>> import mmengine.dist as dist
```

```
>>> # non-distributed environment
>>> data = torch.arange(2, dtype=torch.int64)
>>> data
tensor([0, 1])
>>> output = dist.all_gather(data)
>>> output
[tensor([0, 1])]
```

```
>>> # distributed environment
>>> # We have 2 process groups, 2 ranks.
>>> data = torch.arange(2, dtype=torch.int64) + 1 + 2 * rank
>>> data
tensor([1, 2])  # Rank 0
tensor([3, 4])  # Rank 1
>>> output = dist.all_gather(data)
>>> output
[tensor([1, 2]), tensor([3, 4])]  # Rank 0
[tensor([1, 2]), tensor([3, 4])]  # Rank 1
```

50.1.4 mmengine.dist.all_gather_object

mmengine.dist.**all_gather_object**(*data*, *group=None*)

Gather picklable objects from the whole group into a list. Similar to *all_gather()*, but Python objects can be passed in. Note that the object must be picklable in order to be gathered.

注解： Calling `all_gather_object` in non-distributed environment does nothing and just returns a list containing `data` itself.

注解： Unlike PyTorch `torch.distributed.all_gather_object`, *all_gather_object()* in MMEngine does not pass in an empty list `gather_list` and returns the `gather_list` directly, which is more convenient. The difference between their interfaces is as below:

- MMEngine: all_gather_object(data, group) -> gather_list
- PyTorch: all_gather_object(gather_list, data, group) -> None

---

参数

- **data** (*Any*) –Pickable Python object to be broadcast from current process.
- **group** (*ProcessGroup, optional*) –The process group to work on. If None, the default process group will be used. Defaults to None.

返回 Return a list containing data from the whole group if in distributed environment, otherwise a list only containing `data` itself.

返回类型 list[Tensor]

注解： For NCCL-based process groups, internal tensor representations of objects must be moved to the GPU device before communication starts. In this case, the used device is given by `torch.cuda.current_device()` and it is the user’s responsibility to ensure that this is correctly set so that each rank has an individual GPU, via `torch.cuda.set_device()`.

实际案例

```
>>> import torch
>>> import mmengine.dist as dist
```

```
>>> # non-distributed environment
>>> data = ['foo', 12, {1: 2}]  # any picklable object
>>> gather_objects = dist.all_gather_object(data[dist.get_rank()])
>>> output
['foo']
```

```
>>> # distributed environment
>>> # We have 3 process groups, 3 ranks.
>>> output = dist.all_gather_object(data[dist.get_rank()])
>>> output
['foo', 12, {1: 2}]  # Rank 0
['foo', 12, {1: 2}]  # Rank 1
['foo', 12, {1: 2}]  # Rank 2
```

50.1.5 mmengine.dist.all_reduce

mmengine.dist.**all_reduce**(*data*, *op='sum'*, *group=None*)

Reduces the tensor data across all machines in such a way that all get the final result.

After the call `data` is going to be bitwise identical in all processes.

注解： Calling `all_reduce` in non-distributed environment does nothing.

参数

- **data** (*Tensor*) –Input and output of the collective. The function operates in-place.
- **op** (*str*) –Operation to reduce data. Defaults to 'sum'. Optional values are 'sum', 'mean' and 'produce', 'min', 'max', 'band', 'bor' and 'bxor'.
- **group** (*ProcessGroup, optional*) –The process group to work on. If None, the default process group will be used. Defaults to None.

返回类型 None

实际案例

```
>>> import torch
>>> import mmengine.dist as dist
```

```
>>> # non-distributed environment
>>> data = torch.arange(2, dtype=torch.int64)
>>> dist.all_reduce(data)
>>> data
tensor([0, 1])
```

```
>>> # distributed environment
>>> # We have 2 process groups, 2 ranks.
>>> data = torch.arange(2, dtype=torch.int64) + 1 + 2 * rank
>>> data
tensor([1, 2]) # Rank 0
tensor([3, 4]) # Rank 1
>>> dist.all_reduce(data, op=dist.ReduceOp.SUM)
>>> data
tensor([4, 6]) # Rank 0
tensor([4, 6]) # Rank 1
```

50.1.6 mmengine.dist.all_reduce_dict

mmengine.dist.**all_reduce_dict**(*data*, *op='sum'*, *group=None*)

Reduces the dict across all machines in such a way that all get the final result.

The code is modified from https://github.com/Megvii- BaseDetection/YOLOX/blob/main/yolox/utils/allreduce_norm.py.

参数

- **data** (*dict[str, Tensor]*) –Data to be reduced.
- **op** (*str*) –Operation to reduce data. Defaults to 'sum'. Optional values are 'sum', 'mean' and 'produce', 'min', 'max', 'band', 'bor' and 'bxor'.
- **group** (*ProcessGroup, optional*) –The process group to work on. If None, the default process group will be used. Defaults to None.

返回类型 None

实际案例

```
>>> import torch
>>> import mmengine.dist as dist
```

```
>>> # non-distributed environment
>>> data = {
        'key1': torch.arange(2, dtype=torch.int64),
        'key2': torch.arange(3, dtype=torch.int64)
    }
>>> dist.all_reduce_dict(data)
>>> data
    {'key1': tensor([0, 1]), 'key2': tensor([0, 1, 2])}
```

```
>>> # distributed environment
>>> # We have 2 process groups, 2 ranks.
>>> data = {
        'key1': torch.arange(2, dtype=torch.int64),
        'key2': torch.arange(3, dtype=torch.int64)
    }
>>> dist.all_reduce_dict(data)
>>> data
{'key1': tensor([0, 2]), 'key2': tensor([0, 2, 4])}  # Rank 0
{'key1': tensor([0, 2]), 'key2': tensor([0, 2, 4])}  # Rank 1
```

50.1.7 mmengine.dist.all_reduce_params

mmengine.dist.**all_reduce_params**(*params*, *coalesce=True*, *bucket_size_mb=- 1*, *op='sum'*, *group=None*)

All-reduce parameters.

参数

- **params** (*List or Generator[torch.Tensor, None, None]*) –List of parameters or buffers of a model.
- **coalesce** (*bool, optional*) –Whether to reduce parameters as a whole. Defaults to True.
- **bucket_size_mb** (*int, optional*) –Size of bucket, the unit is MB. Defaults to -1.
- **op** (*str*) –Operation to reduce data. Defaults to 'sum'. Optional values are 'sum', 'mean' and 'produce', 'min', 'max', 'band', 'bor' and 'bxor'.
- **group** (*ProcessGroup, optional*) –The process group to work on. If None, the default process group will be used. Defaults to None.

返回类型 None

实际案例

```
>>> import torch
>>> import mmengine.dist as dist
```

```
>>> # non-distributed environment
>>> data = [torch.arange(2), torch.arange(3)]
>>> dist.all_reduce_params(data)
>>> data
    [tensor([0, 1]), tensor([0, 1, 2])]
```

```
>>> # distributed environment
>>> # We have 2 process groups, 2 ranks.
>>> if dist.get_rank() == 0:
...     data = [torch.tensor([1, 2]), torch.tensor([3, 4])]
... else:
...     data = [torch.tensor([2, 3]), torch.tensor([4, 5])]
```

```
>>> dist.all_reduce_params(data)
>>> data
    [torch.tensor([3, 5]), torch.tensor([7, 9])]
```

50.1.8 mmengine.dist.broadcast

mmengine.dist.**broadcast**(*data*, *src=0*, *group=None*)

Broadcast the data from `src` process to the whole group.

`data` must have the same number of elements in all processes participating in the collective.

注解： Calling `broadcast` in non-distributed environment does nothing.

参数

- **data** (*Tensor*) –Data to be sent if `src` is the rank of current process, and data to be used to save received data otherwise.
- **src** (*int*) –Source rank. Defaults to 0.
- **group** (*ProcessGroup, optional*) –The process group to work on. If None, the default process group will be used. Defaults to None.

返回类型 None

实际案例

```
>>> import torch
>>> import mmengine.dist as dist
```

```
>>> # non-distributed environment
>>> data = torch.arange(2, dtype=torch.int64)
>>> data
tensor([0, 1])
>>> dist.broadcast(data)
>>> data
tensor([0, 1])
```

```
>>> # distributed environment
>>> # We have 2 process groups, 2 ranks.
>>> data = torch.arange(2, dtype=torch.int64) + 1 + 2 * rank
>>> data
tensor([1, 2]) # Rank 0
tensor([3, 4]) # Rank 1
>>> dist.broadcast(data)
>>> data
tensor([1, 2]) # Rank 0
tensor([1, 2]) # Rank 1
```

50.1.9 mmengine.dist.sync_random_seed

mmengine.dist.**sync_random_seed**(*group=None*)

Synchronize a random seed to all processes.

In distributed sampling, different ranks should sample non-overlapped data in the dataset. Therefore, this function is used to make sure that each rank shuffles the data indices in the same order based on the same seed. Then different ranks could use different indices to select non-overlapped data from the same data list.

参数 **group** (*ProcessGroup, optional*) –The process group to work on. If None, the default process group will be used. Defaults to None.

返回 Random seed.

返回类型 int

实际案例

```
>>> import torch
>>> import mmengine.dist as dist
```

```
>>> # non-distributed environment
>>> seed = dist.sync_random_seed()
>>> seed  # which a random number
587791752
```

```
>>> distributed environment
>>> # We have 2 process groups, 2 ranks.
>>> seed = dist.sync_random_seed()
>>> seed
587791752  # Rank 0
587791752  # Rank 1
```

50.1.10 mmengine.dist.broadcast_object_list

mmengine.dist.**broadcast_object_list**(*data*, *src=0*, *group=None*)

Broadcasts picklable objects in `object_list` to the whole group. Similar to *broadcast()*, but Python objects can be passed in. Note that all objects in `object_list` must be picklable in order to be broadcasted.

注解： Calling `broadcast_object_list` in non-distributed environment does nothing.

参数

- **data** (*List[Any]*) –List of input objects to broadcast. Each object must be picklable. Only objects on the `src` rank will be broadcast, but each rank must provide lists of equal sizes.
- **src** (*int*) –Source rank from which to broadcast `object_list`.
- **group** (*Optional[torch.distributed.distributed_c10d.ProcessGroup]*) –(ProcessGroup, optional): The process group to work on. If None, the default process group will be used. Default is `None`.
- **device** (`torch.device`, optional) –If not None, the objects are serialized and converted to tensors which are moved to the `device` before broadcasting. Default is `None`.

返回类型 None

注解： For NCCL-based process groups, internal tensor representations of objects must be moved to the GPU de-

vice before communication starts. In this case, the used device is given by `torch.cuda.current_device()` and it is the user's responsibility to ensure that this is correctly set so that each rank has an individual GPU, via `torch.cuda.set_device()`.

实际案例

```
>>> import torch
>>> import mmengine.dist as dist
```

```
>>> # non-distributed environment
>>> data = ['foo', 12, {1: 2}]
>>> dist.broadcast_object_list(data)
>>> data
['foo', 12, {1: 2}]
```

```
>>> # distributed environment
>>> # We have 2 process groups, 2 ranks.
>>> if dist.get_rank() == 0:
>>>     # Assumes world_size of 3.
>>>     data = ["foo", 12, {1: 2}]  # any picklable object
>>> else:
>>>     data = [None, None, None]
>>> dist.broadcast_object_list(data)
>>> data
["foo", 12, {1: 2}]  # Rank 0
["foo", 12, {1: 2}]  # Rank 1
```

50.1.11 mmengine.dist.collect_results

mmengine.dist.**collect_results**(*results*, *size*, *device='cpu'*, *tmpdir=None*)

Collected results in distributed environments.

参数

- **results** (*list[object]*) –Result list containing result parts to be collected. Each item of `result_part` should be a picklable object.
- **size** (*int*) –Size of the results, commonly equal to length of the results.
- **device** (*str*) –Device name. Optional values are 'cpu' and 'gpu'.
- **tmpdir** (*str | None*) –Temporal directory for collected results to store. If set to None, it will create a temporal directory for it. `tmpdir` should be None when device is 'gpu'. Defaults to None.

返回 The collected results.

返回类型 list or None

实际案例

```
>>> # distributed environment
>>> # We have 2 process groups, 2 ranks.
>>> import mmengine.dist as dist
>>> if dist.get_rank() == 0:
        data = ['foo', {1: 2}]
    else:
        data = [24, {'a': 'b'}]
>>> size = 4
>>> output = dist.collect_results(data, size, device='cpu')
>>> output
['foo', 24, {1: 2}, {'a': 'b'}]  # rank 0
None  # rank 1
```

50.1.12 mmengine.dist.collect_results_cpu

mmengine.dist.**collect_results_cpu**(*result_part*, *size*, *tmpdir=None*)

Collect results under cpu mode.

On cpu mode, this function will save the results on different gpus to `tmpdir` and collect them by the rank 0 worker.

参数

- **result_part** (*list*) –Result list containing result parts to be collected. Each item of `result_part` should be a picklable object.
- **size** (*int*) –Size of the results, commonly equal to length of the results.
- **tmpdir** (*str | None*) –Temporal directory for collected results to store. If set to None, it will create a random temporal directory for it. Defaults to None.

返回 The collected results.

返回类型 list or None

实际案例

```
>>> # distributed environment
>>> # We have 2 process groups, 2 ranks.
>>> import mmengine.dist as dist
>>> if dist.get_rank() == 0:
        data = ['foo', {1: 2}]
    else:
        data = [24, {'a': 'b'}]
>>> size = 4
>>> output = dist.collect_results_cpu(data, size)
>>> output
['foo', 24, {1: 2}, {'a': 'b'}]  # rank 0
None  # rank 1
```

50.1.13 mmengine.dist.collect_results_gpu

mmengine.dist.**collect_results_gpu**(*result_part*, *size*)

Collect results under gpu mode.

On gpu mode, this function will encode results to gpu tensors and use gpu communication for results collection.

参数

- **result_part** (*list[object]*) –Result list containing result parts to be collected. Each item of result_part should be a picklable object.
- **size** (*int*) –Size of the results, commonly equal to length of the results.

返回 The collected results.

返回类型 list or None

实际案例

```
>>> # distributed environment
>>> # We have 2 process groups, 2 ranks.
>>> import mmengine.dist as dist
>>> if dist.get_rank() == 0:
        data = ['foo', {1: 2}]
    else:
        data = [24, {'a': 'b'}]
>>> size = 4
>>> output = dist.collect_results_gpu(data, size)
>>> output
```

(下页继续)

(续上页)

```
['foo', 24, {1: 2}, {'a': 'b'}]  # rank 0
None  # rank 1
```

50.2 utils

get_dist_info	Get distributed information of the given process group.
init_dist	Initialize distributed environment.
init_local_group	Setup the local process group.
get_backend	Return the backend of the given process group.
get_world_size	Return the number of the given process group.
get_rank	Return the rank of the given process group.
get_local_size	Return the number of the current node.
get_local_rank	Return the rank of current process in the current node.
is_main_process	Whether the current rank of the given process group is equal to 0.
master_only	Decorate those methods which should be executed in master process.
barrier	Synchronize all processes from the given process group.
is_distributed	Return True if distributed environment has been initialized.
get_local_group	Return local process group.
get_default_group	Return default process group.
get_data_device	Return the device of `data`.
get_comm_device	Return the device for communication among groups.
cast_data_device	Recursively convert Tensor in `data` to `device`.

50.2.1 mmengine.dist.get_dist_info

mmengine.dist.**get_dist_info**(*group=None*)

Get distributed information of the given process group.

注解： Calling `get_dist_info` in non-distributed environment will return (0, 1).

参数 **group** (*ProcessGroup, optional*) –The process group to work on. If None, the default process group will be used. Defaults to None.

返回 Return a tuple containing the `rank` and `world_size`.

返回类型 tuple[int, int]

50.2.2 mmengine.dist.init_dist

mmengine.dist.**init_dist**(*launcher*, *backend='nccl'*, ***kwargs*)

Initialize distributed environment.

参数

- **launcher** (*str*) –Way to launcher multi processes. Supported launchers are ‘pytorch’, ‘mpi’ and ‘slurm’.
- **backend** (*str*) –Communication Backends. Supported backends are ‘nccl’, ‘gloo’ and ‘mpi’. Defaults to ‘nccl’.
- ****kwargs** –keyword arguments are passed to `init_process_group`.

返回类型 None

50.2.3 mmengine.dist.init_local_group

mmengine.dist.**init_local_group**(*node_rank*, *num_gpus_per_node*)

Setup the local process group.

Setup a process group which only includes processes that on the same machine as the current process.

The code is modified from https://github.com/facebookresearch/detectron2/blob/main/detectron2/engine/launch.py

参数

- **node_rank** (*int*) –Rank of machines used for training.
- **num_gpus_per_node** (*int*) –Number of gpus used for training in a single machine.

50.2.4 mmengine.dist.get_backend

mmengine.dist.**get_backend**(*group=None*)

Return the backend of the given process group.

注解: Calling `get_backend` in non-distributed environment will return None.

参数 **group** (*ProcessGroup, optional*) –The process group to work on. The default is the general main process group. If another specific group is specified, the calling process must be part of `group`. Defaults to None.

返回 Return the backend of the given process group as a lower case string if in distributed environment, otherwise None.

返回类型 str or None

50.2.5 mmengine.dist.get_world_size

mmengine.dist.**get_world_size**(*group=None*)

Return the number of the given process group.

注解: Calling `get_world_size` in non-distributed environment will return 1.

参数 **group** (*ProcessGroup, optional*) –The process group to work on. If None, the default process group will be used. Defaults to None.

返回 Return the number of processes of the given process group if in distributed environment, otherwise 1.

返回类型 int

50.2.6 mmengine.dist.get_rank

mmengine.dist.**get_rank**(*group=None*)

Return the rank of the given process group.

Rank is a unique identifier assigned to each process within a distributed process group. They are always consecutive integers ranging from 0 to `world_size`.

注解: Calling `get_rank` in non-distributed environment will return 0.

参数 **group** (*ProcessGroup, optional*) –The process group to work on. If None, the default process group will be used. Defaults to None.

返回 Return the rank of the process group if in distributed environment, otherwise 0.

返回类型 int

50.2.7 mmengine.dist.get_local_size

mmengine.dist.**get_local_size**()

Return the number of the current node.

返回 Return the number of processes in the current node if in distributed environment, otherwise 1.

返回类型 int

50.2.8 mmengine.dist.get_local_rank

mmengine.dist.**get_local_rank**()

Return the rank of current process in the current node.

返回 Return the rank of current process in the current node if in distributed environment, otherwise 0

返回类型 int

50.2.9 mmengine.dist.is_main_process

mmengine.dist.**is_main_process**(*group=None*)

Whether the current rank of the given process group is equal to 0.

参数 **group** (*ProcessGroup, optional*) –The process group to work on. If None, the default process group will be used. Defaults to None.

返回 Return True if the current rank of the given process group is equal to 0, otherwise False.

返回类型 bool

50.2.10 mmengine.dist.master_only

mmengine.dist.**master_only**(*func*)

Decorate those methods which should be executed in master process.

参数 **func** (*callable*) –Function to be decorated.

返回 Return decorated function.

返回类型 callable

50.2.11 mmengine.dist.barrier

mmengine.dist.**barrier**(*group=None*)

Synchronize all processes from the given process group.

This collective blocks processes until the whole group enters this function.

注解： Calling `barrier` in non-distributed environment will do nothing.

参数 **group** (*ProcessGroup, optional*) –The process group to work on. If None, the default process group will be used. Defaults to None.

返回类型 None

50.2.12 mmengine.dist.is_distributed

mmengine.dist.**is_distributed**()

Return True if distributed environment has been initialized.

返回类型 bool

50.2.13 mmengine.dist.get_local_group

mmengine.dist.**get_local_group**()

Return local process group.

返回类型 Optional[torch.distributed.distributed_c10d.ProcessGroup]

50.2.14 mmengine.dist.get_default_group

mmengine.dist.**get_default_group**()

Return default process group.

返回类型 Optional[torch.distributed.distributed_c10d.ProcessGroup]

50.2.15 mmengine.dist.get_data_device

mmengine.dist.**get_data_device**(*data*)

Return the device of `data`.

If `data` is a sequence of Tensor, all items in `data` should have a same device type.

If `data` is a dict whose values are Tensor, all values should have a same device type.

参数 **data** (*Tensor or Sequence or dict*) –Inputs to be inferred the device.

返回 The device of `data`.

返回类型 torch.device

实际案例

```
>>> import torch
>>> from mmengine.dist import cast_data_device
>>> # data is a Tensor
>>> data = torch.tensor([0, 1])
>>> get_data_device(data)
device(type='cpu')
>>> # data is a list of Tensor
>>> data = [torch.tensor([0, 1]), torch.tensor([2, 3])]
>>> get_data_device(data)
device(type='cpu')
>>> # data is a dict
>>> data = {'key1': torch.tensor([0, 1]), 'key2': torch.tensor([0, 1])}
>>> get_data_device(data)
device(type='cpu')
```

50.2.16 mmengine.dist.get_comm_device

mmengine.dist.**get_comm_device**(*group=None*)

Return the device for communication among groups.

参数 **group** (*ProcessGroup, optional*) –The process group to work on.

返回 The device of backend.

返回类型 torch.device

50.2.17 mmengine.dist.cast_data_device

mmengine.dist.**cast_data_device**(*data*, *device*, *out=None*)

Recursively convert Tensor in `data` to `device`.

If `data` has already on the `device`, it will not be casted again.

参数

- **data** (*Tensor or list or dict*) –Inputs to be casted.
- **device** (*torch.device*) –Destination device type.
- **out** (*Tensor or list or dict, optional*) –If `out` is specified, its value will be equal to `data`. Defaults to None.

返回 `data` was casted to `device`.

返回类型 Tensor or list or dict

CHAPTER 51

mmengine.utils

mmengine.utils

51.1 Manager

`ManagerMeta`	The metaclass for global accessible class.
`ManagerMixin`	`ManagerMixin` is the base class for classes that have global access requirements.

51.1.1 ManagerMeta

class mmengine.utils.**ManagerMeta**(**args*)

The metaclass for global accessible class.

The subclasses inheriting from `ManagerMeta` will manage their own `_instance_dict` and root instances. The constructors of subclasses must contain the `name` argument.

实际案例

```
>>> class SubClass1(metaclass=ManagerMeta):
>>>     def __init__(self, *args, **kwargs):
>>>         pass
AssertionError: <class '__main__.SubClass1'>.__init__ must have the
name argument.
>>> class SubClass2(metaclass=ManagerMeta):
>>>     def __init__(self, name):
>>>         pass
>>> # valid format.
```

51.1.2 ManagerMixin

class mmengine.utils.**ManagerMixin**(*name=''*, ***kwargs*)

`ManagerMixin` is the base class for classes that have global access requirements.

The subclasses inheriting from `ManagerMixin` can get their global instances.

实际案例

```
>>> class GlobalAccessible(ManagerMixin):
>>>     def __init__(self, name=''):
>>>         super().__init__(name)
>>>
>>> GlobalAccessible.get_instance('name')
>>> instance_1 = GlobalAccessible.get_instance('name')
>>> instance_2 = GlobalAccessible.get_instance('name')
>>> assert id(instance_1) == id(instance_2)
```

参数 **name** (*str*) –Name of the instance. Defaults to ''.

classmethod **check_instance_created**(*name*)

Check whether the name corresponding instance exists.

参数 **name** (*str*) –Name of instance.

返回 Whether the name corresponding instance exists.

返回类型 bool

classmethod get_current_instance()

Get latest created instance.

Before calling `get_current_instance`, The subclass must have called `get_instance(xxx)` at least once.

Examples

```
>>> instance = GlobalAccessible.get_current_instance()
AssertionError: At least one of name and current needs to be set
>>> instance = GlobalAccessible.get_instance('name1')
>>> instance.instance_name
name1
>>> instance = GlobalAccessible.get_current_instance()
>>> instance.instance_name
name1
```

返回 Latest created instance.

返回类型 object

classmethod get_instance(*name*, ***kwargs*)

Get subclass instance by name if the name exists.

If corresponding name instance has not been created, `get_instance` will create an instance, otherwise `get_instance` will return the corresponding instance.

Examples

```
>>> instance1 = GlobalAccessible.get_instance('name1')
>>> # Create name1 instance.
>>> instance.instance_name
name1
>>> instance2 = GlobalAccessible.get_instance('name1')
>>> # Get name1 instance.
>>> assert id(instance1) == id(instance2)
```

参数 **name** (*str*) –Name of instance. Defaults to ‘’.

返回 Corresponding name instance, the latest instance, or root instance.

返回类型 object

property instance_name: str

Get the name of instance.

返回 Name of instance.

返回类型 str

51.2 Path

check_file_exist	
fopen	
is_abs	Check if path is an absolute path in different backends.
is_filepath	
mkdir_or_exist	
scandir	Scan a directory to find the interested files.
symlink	

51.2.1 mmengine.utils.check_file_exist

mmengine.utils.**check_file_exist**(*filename*, *msg_tmpl='file "{}" does not exist'*)

51.2.2 mmengine.utils.fopen

mmengine.utils.**fopen**(*filepath*, **args*, ***kwargs*)

51.2.3 mmengine.utils.is_abs

mmengine.utils.**is_abs**(*path*)

Check if path is an absolute path in different backends.

参数 **path** (*str*) –path of directory or file.

返回 whether path is an absolute path.

返回类型 bool

51.2.4 mmengine.utils.is_filepath

mmengine.utils.**is_filepath**(*x*)

51.2.5 mmengine.utils.mkdir_or_exist

mmengine.utils.**mkdir_or_exist**(*dir_name*, *mode=511*)

51.2.6 mmengine.utils.scandir

mmengine.utils.**scandir**(*dir_path*, *suffix=None*, *recursive=False*, *case_sensitive=True*)

Scan a directory to find the interested files.

参数

- **dir_path** (str | Path) –Path of the directory.
- **suffix** (*str | tuple(str), optional*) –File suffix that we are interested in. Default: None.
- **recursive** (*bool, optional*) –If set to True, recursively scan the directory. Default: False.
- **case_sensitive** (*bool, optional*) –If set to False, ignore the case of suffix. Default: True.

返回 A generator for all the interested files with relative paths.

51.2.7 mmengine.utils.symlink

mmengine.utils.**symlink**(*src*, *dst*, *overwrite=True*, ***kwargs*)

51.3 Package

call_command	
install_package	
get_installed_path	Get installed path of package.
is_installed	Check package whether installed.

51.3.1 mmengine.utils.call_command

mmengine.utils.**call_command**(*cmd*)

参数 **cmd** (*list*) –

返回类型 None

51.3.2 mmengine.utils.install_package

mmengine.utils.**install_package**(*package*)

参数 **package** (*str*) –

51.3.3 mmengine.utils.get_installed_path

mmengine.utils.**get_installed_path**(*package*)

Get installed path of package.

参数 **package** (*str*) –Name of package.

返回类型 str

示例

```
>>> get_installed_path('mmcls')
>>> '.../lib/python3.7/site-packages/mmcls'
```

51.3.4 mmengine.utils.is_installed

mmengine.utils.**is_installed**(*package*)

Check package whether installed.

参数 **package** (*str*) –Name of package to be checked.

返回类型 bool

51.4 Version

digit_version	Convert a version string into a tuple of integers.
get_git_hash	Get the git hash of the current repo.

51.4.1 mmengine.utils.digit_version

mmengine.utils.**digit_version**(*version_str*, *length=4*)

Convert a version string into a tuple of integers.

This method is usually used for comparing two versions. For pre-release versions: alpha < beta < rc.

参数

- **version_str** (*str*) –The version string.
- **length** (*int*) –The maximum number of version levels. Default: 4.

返回 The version info in digits (integers).

返回类型 tuple[int]

51.4.2 mmengine.utils.get_git_hash

mmengine.utils.**get_git_hash**(*fallback='unknown'*, *digits=None*)

Get the git hash of the current repo.

参数

- **fallback** (*str, optional*) –The fallback string when git hash is unavailable. Defaults to ‘unknown’.
- **digits** (*int, optional*) –kept digits of the hash. Defaults to None, meaning all digits are kept.

返回 Git commit hash.

返回类型 str

51.5 Progress Bar

ProgressBar	A progress bar which can print the progress.

51.5.1 ProgressBar

class mmengine.utils.**ProgressBar**(*task_num=0*, *bar_width=50*, *start=True*, *file=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='UTF-8'>*)

A progress bar which can print the progress.

track_iter_progress	Track the progress of tasks iteration or enumeration with a progress bar.
track_parallel_progress	Track the progress of parallel task execution with a progress bar.
track_progress	Track the progress of tasks execution with a progress bar.

51.5.2 mmengine.utils.track_iter_progress

mmengine.utils.**track_iter_progress**(*tasks*, *bar_width=50*, *file=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='UTF-8'>*)

Track the progress of tasks iteration or enumeration with a progress bar.

Tasks are yielded with a simple for-loop.

参数

- **tasks** (*list or tuple[Iterable, int]*) –A list of tasks or (tasks, total num).
- **bar_width** (*int*) –Width of progress bar.

生成器 *list* –The task results.

51.5.3 mmengine.utils.track_parallel_progress

mmengine.utils.**track_parallel_progress**(*func*, *tasks*, *nproc*, *initializer=None*, *initargs=None*, *bar_width=50*, *chunksize=1*, *skip_first=False*, *keep_order=True*, *file=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='UTF-8'>*)

Track the progress of parallel task execution with a progress bar.

The built-in multiprocessing module is used for process pools and tasks are done with Pool.map() or Pool.imap_unordered().

参数

- **func** (*callable*) –The function to be applied to each task.
- **tasks** (*list or tuple[Iterable, int]*) –A list of tasks or (tasks, total num).
- **nproc** (*int*) –Process (worker) number.
- **initializer** (*None or callable*) –Refer to `multiprocessing.Pool` for details.
- **initargs** (*None or tuple*) –Refer to `multiprocessing.Pool` for details.
- **chunksize** (*int*) –Refer to `multiprocessing.Pool` for details.
- **bar_width** (*int*) –Width of progress bar.
- **skip_first** (*bool*) –Whether to skip the first sample for each worker when estimating fps, since the initialization step may takes longer.
- **keep_order** (*bool*) –If True, `Pool.imap()` is used, otherwise `Pool.imap_unordered()` is used.

返回 The task results.

返回类型 list

51.5.4 mmengine.utils.track_progress

`mmengine.utils.`**`track_progress`**(*func*, *tasks*, *bar_width=50*, *file=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='UTF-8'>*, ***kwargs*)

Track the progress of tasks execution with a progress bar.

Tasks are done with a simple for-loop.

参数

- **func** (*callable*) –The function to be applied to each task.
- **tasks** (*list or tuple[Iterable, int]*) –A list of tasks or (tasks, total num).
- **bar_width** (*int*) –Width of progress bar.

返回 The task results.

返回类型 list

51.6 Miscellaneous

Timer	A flexible Timer class.
TimerError	

51.6.1 Timer

class mmengine.utils.**Timer**(*start=True*, *print_tmpl=None*)

A flexible Timer class.

实际案例

```
>>> import time
>>> import mmcv
>>> with mmcv.Timer():
>>>     # simulate a code block that will run for 1s
>>>     time.sleep(1)
1.000
>>> with mmcv.Timer(print_tmpl='it takes {:.1f} seconds'):
>>>     # simulate a code block that will run for 1s
>>>     time.sleep(1)
it takes 1.0 seconds
>>> timer = mmcv.Timer()
>>> time.sleep(0.5)
>>> print(timer.since_start())
0.500
>>> time.sleep(0.5)
>>> print(timer.since_last_check())
0.500
>>> print(timer.since_start())
1.000
```

property is_running

indicate whether the timer is running

Type bool

since_last_check()

Time since the last checking.

Either *since_start()* or *since_last_check()* is a checking operation.

返回 Time in seconds.

返回类型 float

since_start()

Total time since the timer is started.

返回 Time in seconds.

返回类型 float

start()

Start the timer.

51.6.2 TimerError

class mmengine.utils.**TimerError**(*message*)

is_list_of	Check whether it is a list of some type.
is_tuple_of	Check whether it is a tuple of some type.
is_seq_of	Check whether it is a sequence of some type.
is_str	Whether the input is an string instance.
iter_cast	Cast elements of an iterable object into some type.
list_cast	Cast elements of an iterable object into a list of some type.
tuple_cast	Cast elements of an iterable object into a tuple of some type.
concat_list	Concatenate a list of list into a single list.
slice_list	Slice a list into several sub lists by a list of given length.
to_1tuple	
to_2tuple	
to_3tuple	
to_4tuple	
to_ntuple	
check_prerequisites	A decorator factory to check if prerequisites are satisfied.
deprecated_api_warning	A decorator to check if some arguments are deprecate and try to replace deprecate src_arg_name to dst_arg_name.
has_method	Check whether the object has a method.

下页继续

表 8 – 续上页

is_method_overridden	Check if a method of base class is overridden in derived class.
import_modules_from_strings	Import modules from the given list of strings.
requires_executable	A decorator to check if some executable files are installed.
requires_package	A decorator to check if some python packages are installed.
check_time	Add check points in a single line.

51.6.3 mmengine.utils.is_list_of

mmengine.utils.**is_list_of**(*seq*, *expected_type*)

Check whether it is a list of some type.

A partial method of *is_seq_of()*.

51.6.4 mmengine.utils.is_tuple_of

mmengine.utils.**is_tuple_of**(*seq*, *expected_type*)

Check whether it is a tuple of some type.

A partial method of *is_seq_of()*.

51.6.5 mmengine.utils.is_seq_of

mmengine.utils.**is_seq_of**(*seq*, *expected_type*, *seq_type=None*)

Check whether it is a sequence of some type.

参数

- **seq** (*Sequence*) –The sequence to be checked.
- **expected_type** (*type or tuple*) –Expected type of sequence items.
- **seq_type** (*type, optional*) –Expected sequence type. Defaults to None.

返回 Return True if `seq` is valid else False.

返回类型 bool

实际案例

```
>>> from mmengine.utils import is_seq_of
>>> seq = ['a', 'b', 'c']
>>> is_seq_of(seq, str)
True
>>> is_seq_of(seq, int)
False
```

51.6.6 mmengine.utils.is_str

mmengine.utils.**is_str**(*x*)

Whether the input is an string instance.

Note: This method is deprecated since python 2 is no longer supported.

51.6.7 mmengine.utils.iter_cast

mmengine.utils.**iter_cast**(*inputs*, *dst_type*, *return_type=None*)

Cast elements of an iterable object into some type.

参数

- **inputs** (*Iterable*) –The input object.
- **dst_type** (*type*) –Destination type.
- **return_type** (*type, optional*) –If specified, the output object will be converted to this type, otherwise an iterator.

返回 The converted object.

返回类型 iterator or specified type

51.6.8 mmengine.utils.list_cast

mmengine.utils.**list_cast**(*inputs*, *dst_type*)

Cast elements of an iterable object into a list of some type.

A partial method of *iter_cast()*.

51.6.9 mmengine.utils.tuple_cast

mmengine.utils.**tuple_cast**(*inputs*, *dst_type*)

Cast elements of an iterable object into a tuple of some type.

A partial method of *iter_cast()*.

51.6.10 mmengine.utils.concat_list

mmengine.utils.**concat_list**(*in_list*)

Concatenate a list of list into a single list.

参数 **in_list** (*list*) –The list of list to be merged.

返回 The concatenated flat list.

返回类型 list

51.6.11 mmengine.utils.slice_list

mmengine.utils.**slice_list**(*in_list*, *lens*)

Slice a list into several sub lists by a list of given length.

参数

- **in_list** (*list*) –The list to be sliced.
- **lens** (*int or list*) –The expected length of each out list.

返回 A list of sliced list.

返回类型 list

51.6.12 mmengine.utils.to_1tuple

mmengine.utils.**to_1tuple**(*x*)

51.6.13 mmengine.utils.to_2tuple

mmengine.utils.**to_2tuple**(*x*)

51.6.14 mmengine.utils.to_3tuple

mmengine.utils.**to_3tuple**(*x*)

51.6.15 mmengine.utils.to_4tuple

mmengine.utils.**to_4tuple**(*x*)

51.6.16 mmengine.utils.to_ntuple

mmengine.utils.**to_ntuple**(*n*)

51.6.17 mmengine.utils.check_prerequisites

mmengine.utils.**check_prerequisites**(*prerequisites*, *checker*, *msg_tmpl='Prerequisites "{}" are required in method "{}" but not found, please install them first.'*)

A decorator factory to check if prerequisites are satisfied.

参数

- **prerequisites** (*str of list[str]*) –Prerequisites to be checked.
- **checker** (*callable*) –The checker method that returns True if a prerequisite is meet, False otherwise.
- **msg_tmpl** (*str*) –The message template with two variables.

返回 A specific decorator.

返回类型 decorator

51.6.18 mmengine.utils.deprecated_api_warning

mmengine.utils.**deprecated_api_warning**(*name_dict*, *cls_name=None*)

A decorator to check if some arguments are deprecate and try to replace deprecate src_arg_name to dst_arg_name.

参数

- **name_dict** (*dict*) –key (str): Deprecate argument names. val (str): Expected argument names.
- **cls_name** (*Optional[str]*) –

返回 New function.

返回类型 func

51.6.19 mmengine.utils.has_method

mmengine.utils.**has_method**(*obj*, *method*)

Check whether the object has a method.

参数

- **method** (*str*) –The method name to check.
- **obj** (*object*) –The object to check.

返回 True if the object has the method else False.

返回类型 bool

51.6.20 mmengine.utils.is_method_overridden

mmengine.utils.**is_method_overridden**(*method*, *base_class*, *derived_class*)

Check if a method of base class is overridden in derived class.

参数

- **method** (*str*) –the method name to check.
- **base_class** (*type*) –the class of the base class.
- **derived_class** (*type | Any*) –the class or instance of the derived class.

返回类型 bool

51.6.21 mmengine.utils.import_modules_from_strings

mmengine.utils.**import_modules_from_strings**(*imports*, *allow_failed_imports=False*)

Import modules from the given list of strings.

参数

- **imports** (*list | str | None*) –The given module names to be imported.
- **allow_failed_imports** (*bool*) –If True, the failed imports will return None. Otherwise, an ImportError is raise. Default: False.

返回 The imported modules.

返回类型 list[module] | module | None

实际案例

```
>>> osp, sys = import_modules_from_strings(
...     ['os.path', 'sys'])
>>> import os.path as osp_
>>> import sys as sys_
>>> assert osp == osp_
>>> assert sys == sys_
```

51.6.22 mmengine.utils.requires_executable

mmengine.utils.**requires_executable**(*prerequisites*)

A decorator to check if some executable files are installed.

示例

```
>>> @requires_executable('ffmpeg')
>>> func(arg1, args):
>>>     print(1)
1
```

51.6.23 mmengine.utils.requires_package

mmengine.utils.**requires_package**(*prerequisites*)

A decorator to check if some python packages are installed.

示例

```
>>> @requires_package('numpy')
>>> func(arg1, args):
>>>     return numpy.zeros(1)
array([0.])
>>> @requires_package(['numpy', 'non_package'])
>>> func(arg1, args):
>>>     return numpy.zeros(1)
ImportError
```

51.6.24 mmengine.utils.check_time

mmengine.utils.**check_time**(*timer_id*)

Add check points in a single line.

This method is suitable for running a task on a list of items. A timer will be registered when the method is called for the first time.

实际案例

```
>>> import time
>>> import mmcv
>>> for i in range(1, 6):
>>>     # simulate a code block
>>>     time.sleep(i)
>>>     mmcv.check_time('task1')
2.000
3.000
4.000
5.000
```

参数 **str** –Timer identifier.

CHAPTER 52

mmengine.utils.dl_utils

TimeCounter	A tool that counts the average running time of a function or a method.

52.1 TimeCounter

class mmengine.utils.dl_utils.**TimeCounter**(*log_interval=1*, *warmup_interval=1*, *with_sync=True*, *tag=None*, *logger=None*)

A tool that counts the average running time of a function or a method. Users can use it as a decorator or context manager to calculate the average running time of code blocks.

参数

- **log_interval** (*int*) –The interval of logging. Defaults to 1.
- **warmup_interval** (*int*) –The interval of warmup. Defaults to 1.
- **with_sync** (*bool*) –Whether to synchronize cuda. Defaults to True.
- **tag** (*str, optional*) –Function tag. Used to distinguish between different functions or methods being called. Defaults to None.
- **logger** (MMLogger, *optional*) –Formatted logger used to record messages. Defaults to None.

实际案例

```
>>> import time
>>> from mmengine.utils.dl_utils import TimeCounter
>>> @TimeCounter()
... def fun1():
...     time.sleep(0.1)
... fun1()
[fun1]-time per run averaged in the past 1 runs: 100.0 ms
```

```
>>> @@TimeCounter(log_interval=2, tag='fun')
... def fun2():
...    time.sleep(0.2)
>>> for _ in range(3):
...    fun2()
[fun]-time per run averaged in the past 2 runs: 200.0 ms
```

```
>>> with TimeCounter(tag='fun3'):
...      time.sleep(0.3)
[fun3]-time per run averaged in the past 1 runs: 300.0 ms
```

print_time(*elapsed*)

print times per count.

参数 elapsed(*Union[int, float]*) –

返回类型 None

collect_env	Collect the information of the running environments.
load_url	Loads the Torch serialized object at the given URL.
has_batch_norm	Detect whether model has a BatchNormalization layer.
is_norm	Check if a layer is a normalization layer.
mmcv_full_available	Check whether mmcv-full is installed.
tensor2imgs	Convert tensor to 3-channel images or 1-channel gray images.
TORCH_VERSION	A string with magic powers to compare to both Version and iterables! Prior to 1.10.0 torch.__version__ was stored as a str and so many did comparisons against torch.__version__ as if it were a str.
set_multi_processing	Set multi-processing related environment.
torch_meshgrid	A wrapper of torch.meshgrid to compat different PyTorch versions.

下页继续

表 2 – 续上页

is_jit_tracing

52.2 mmengine.utils.dl_utils.collect_env

mmengine.utils.dl_utils.**collect_env**()

Collect the information of the running environments.

返回

The environment information. The following fields are contained.

- sys.platform: The variable of `sys.platform`.
- Python: Python version.
- CUDA available: Bool, indicating if CUDA is available.
- GPU devices: Device type of each GPU.
- CUDA_HOME (optional): The env var `CUDA_HOME`.
- NVCC (optional): NVCC version.
- GCC: GCC version, "n/a" if GCC is not installed.
- MSVC: Microsoft Virtual C++ Compiler version, Windows only.
- PyTorch: PyTorch version.
- PyTorch compiling details: The output of `torch.__config__.show()`.
- TorchVision (optional): TorchVision version.
- OpenCV (optional): OpenCV version.
- MMENGINE: MMENGINE version.

返回类型 dict

52.3 mmengine.utils.dl_utils.load_url

mmengine.utils.dl_utils.**load_url**(*url*, *model_dir=None*, *map_location=None*, *progress=True*, *check_hash=False*, *file_name=None*)

Loads the Torch serialized object at the given URL.

If downloaded file is a zip file, it will be automatically decompressed.

If the object is already present in *model_dir*, it’s deserialized and returned. The default value of `model_dir` is `<hub_dir>/checkpoints` where `hub_dir` is the directory returned by `get_dir()`.

参数

- **url** (*str*) –URL of the object to download
- **model_dir** (*str, optional*) –directory in which to save the object
- **map_location** (*optional*) –a function or a dict specifying how to remap storage locations (see torch.load)
- **progress** (*bool, optional*) –whether or not to display a progress bar to stderr. Default: True
- **check_hash** (*bool, optional*) –If True, the filename part of the URL should follow the naming convention `filename-<sha256>.ext` where `<sha256>` is the first eight or more digits of the SHA256 hash of the contents of the file. The hash is used to ensure unique names and to verify the contents of the file. Default: False
- **file_name** (*str, optional*) –name for the downloaded file. Filename from `url` will be used if not set.

返回类型 Dict[str, Any]

示例

```
>>> state_dict = torch.hub.load_state_dict_from_url('https://s3.amazonaws.com/
↪pytorch/models/resnet18-5c106cde.pth')
```

52.4 mmengine.utils.dl_utils.has_batch_norm

mmengine.utils.dl_utils.**has_batch_norm**(*model*)

Detect whether model has a BatchNormalization layer.

参数 **model** (*nn.Module*) –training model.

返回 whether model has a BatchNormalization layer

返回类型 bool

52.5 mmengine.utils.dl_utils.is_norm

mmengine.utils.dl_utils.**is_norm**(*layer*, *exclude=None*)

Check if a layer is a normalization layer.

参数

- **layer** (*nn.Module*) –The layer to be checked.
- **exclude** (*type, tuple[type], optional*) –Types to be excluded.

返回 Whether the layer is a norm layer.

返回类型 bool

52.6 mmengine.utils.dl_utils.mmcv_full_available

mmengine.utils.dl_utils.**mmcv_full_available**()

Check whether mmcv-full is installed.

返回 True if mmcv-full is installed else False.

返回类型 bool

52.7 mmengine.utils.dl_utils.tensor2imgs

mmengine.utils.dl_utils.**tensor2imgs**(*tensor*, *mean=None*, *std=None*, *to_bgr=True*)

Convert tensor to 3-channel images or 1-channel gray images.

参数

- **tensor** (*torch.Tensor*) –Tensor that contains multiple images, shape (N, C, H, W). C can be either 3 or 1. If C is 3, the format should be RGB.
- **mean** (*tuple[float], optional*) –Mean of images. If None, (0, 0, 0) will be used for tensor with 3-channel, while (0,) for tensor with 1-channel. Defaults to None.
- **std** (*tuple[float], optional*) –Standard deviation of images. If None, (1, 1, 1) will be used for tensor with 3-channel, while (1,) for tensor with 1-channel. Defaults to None.
- **to_bgr** (*bool*) –For the tensor with 3 channel, convert its format to BGR. For the tensor with 1 channel, it must be False. Defaults to True.

返回 A list that contains multiple images.

返回类型 list[np.ndarray]

52.8 mmengine.utils.dl_utils.TORCH_VERSION

mmengine.utils.dl_utils.**TORCH_VERSION = '1.13.1+cu117'**

A string with magic powers to compare to both Version and iterables! Prior to 1.10.0 torch.__version__ was stored as a str and so many did comparisons against torch.__version__ as if it were a str. In order to not break them we have TorchVersion which masquerades as a str while also having the ability to compare against both packaging.version.Version as well as tuples of values, eg. (1, 2, 1) .. rubric:: 实际案例

Comparing a TorchVersion object to a Version object TorchVersion('1.10.0a') > Version('1.10.0a')

Comparing a TorchVersion object to a Tuple object TorchVersion('1.10.0a') > (1, 2) # 1.2 TorchVersion('1.10.0a') > (1, 2, 1) # 1.2.1

Comparing a TorchVersion object against a string TorchVersion('1.10.0a') > '1.2' TorchVersion('1.10.0a') > '1.2.1'

52.9 mmengine.utils.dl_utils.set_multi_processing

mmengine.utils.dl_utils.**set_multi_processing**(*mp_start_method='fork'*, *opencv_num_threads=0*, *distributed=False*)

Set multi-processing related environment.

参数

- **mp_start_method** (*str*) –Set the method which should be used to start child processes. Defaults to 'fork'.
- **opencv_num_threads** (*int*) –Number of threads for opencv. Defaults to 0.
- **distributed** (*bool*) –True if distributed environment. Defaults to False.

返回类型 None

52.10 mmengine.utils.dl_utils.torch_meshgrid

mmengine.utils.dl_utils.**torch_meshgrid**(**tensors*)

A wrapper of torch.meshgrid to compat different PyTorch versions.

Since PyTorch 1.10.0a0, torch.meshgrid supports the arguments `indexing`. So we implement a wrapper here to avoid warning when using high-version PyTorch and avoid compatibility issues when using previous versions of PyTorch.

参数 **tensors** (*List[Tensor]*) –List of scalars or 1 dimensional tensors.

返回 Sequence of meshgrid tensors.

返回类型 Sequence[Tensor]

52.11 mmengine.utils.dl_utils.is_jit_tracing

mmengine.utils.dl_utils.**is_jit_tracing**()

返回类型 bool

CHAPTER 53

English

CHAPTER 54

简体中文

CHAPTER 55

Indices and tables

- genindex
- modindex
- search

索引

符号

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X

Y

Z