mmengine

Release 0.4.0

mmengine contributors

Feb 07, 2023

GET STARTED

You can switch between Chinese and English documents in the lower-left corner of the layout.

CHAPTER

ONE

INTRODUCTION

Coming soon. Please refer to chinese documentation.

CHAPTER

TWO

INSTALLATION

2.1 Prerequisites

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

2.2 Prepare the Environment

1. Use conda and activate the environment:

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

2. Install PyTorch

   Before installing `MMEngine`, please make sure that PyTorch has been successfully installed in the environment. You can refer to PyTorch official installation documentation. Verify the installation with the following command:

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

2.3 Install MMEngine

2.3.1 Install with mim

mim is a package management tool for OpenMMLab projects, which can be used to install the OpenMMLab project easily.

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

2.3.2 Install with pip

```
pip install mmengine
```

2.3.3 Use docker images

1. Build the image

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

   More information can be referred from mmengine/docker.

2. Run the image

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

Build from source

```
# if cloning speed is too slow, you can switch the source to https://gitee.com/open-
↪mmlab/mmengine.git
git clone https://github.com/open-mmlab/mmengine.git
cd mmengine
pip install -e . -v
```

2.3.4 Verify the Installation

To verify if `MMEngine` and the necessary environment are successfully installed, we can run this command:

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

CHAPTER

THREE

15 MINUTES TO GET STARTED WITH MMENGINE

In this tutorial, we'll take training a ResNet-50 model on CIFAR-10 dataset as an example. We will build a complete and configurable pipeline for both training and validation in only 80 lines of code with `MMEgnine`. The whole process includes the following steps:

1. *Build a Model*
2. *Build a Dataset and DataLoader*
3. *Build a Evaluation Metrics*
4. *Build a Runner and Run the Task*

3.1 Build a Model

First, we need to build a **model**. In MMEngine, the model should inherit from `BaseModel`. Aside from parameters representing inputs from the dataset, its `forward` method needs to accept an extra argument called `mode`:

- for training, the value of `mode` is "loss," and the `forward` method should return a `dict` containing the key "loss".
- for validation, the value of `mode` is "predict", and the forward method should return results containing both predictions and labels.

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


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
```

3.2 Build a Dataset and DataLoader

Next, we need to create **Dataset** and **DataLoader** for training and validation. For basic training and validation, we can simply use built-in datasets supported in TorchVision.

```
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 Build a Evaluation Metrics

To validate and test the model, we need to define a **Metric** called accuracy to evaluate the model. This metric needs inherit from `BaseMetric` and implements the `process` and `compute_metrics` methods where the `process` method accepts the output of the dataset and other outputs when `mode="predict"`. The output data at this scenario is a batch of data. After processing this batch of data, we save the information to `self.results` property. `compute_metrics` accepts a `results` parameter. The input `results` of `compute_metrics` is all the information saved in `process` (In the case of a distributed environment, `results` are the information collected from all `process` in all the processes). Use these information to calculate and return a `dict` that holds the results of the evaluation metrics

```
from mmengine.evaluator import BaseMetric

class Accuracy(BaseMetric):
    def process(self, data_batch, data_samples):
        score, gt = data_samples
        # save the middle result of a batch to `self.results`
        self.results.append({
            'batch_size': len(gt),
```

(continues on next page)

(continued from previous page)

```
            '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 the dict containing the eval results
        # the key is the name of the metric name
        return dict(accuracy=100 * total_correct / total_size)
```

3.4 Build a Runner and Run the Task

Now we can build a **Runner** with previously defined `Model`, `DataLoader`, and `Metrics`, and some other configs shown as follows:

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

runner = Runner(
    # the model used for training and validation.
    # Needs to meet specific interface requirements
    model=MMResNet50(),
    # working directory which saves training logs and weight files
    work_dir='./work_dir',
    # train dataloader needs to meet the PyTorch data loader protocol
    train_dataloader=train_dataloader,
    # optimize wrapper for optimization with additional features like
    # AMP, gradtient accumulation, etc
    optim_wrapper=dict(optimizer=dict(type=SGD, lr=0.001, momentum=0.9)),
    # trainging coinfs for specifying training epoches, verification intervals, etc
    train_cfg=dict(by_epoch=True, max_epochs=5, val_interval=1),
    # validation dataloaer also needs to meet the PyTorch data loader protocol
    val_dataloader=val_dataloader,
    # validation configs for specifying additional parameters required for validation
    val_cfg=dict(),
    # validation evaluator. The default one is used here
    val_evaluator=dict(type=Accuracy),
)

runner.train()
```

Finally, let's put all the codes above together into a complete script that uses the `MMEngine` executor for training and validation:

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

(continues on next page)

(continued from previous page)

```
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
        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,
```

(continues on next page)

(continued from previous page)

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

Training log would be similar to this:

```
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
```

(continues on next page)

(continued from previous page)

```
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
```

The corresponding implementation of PyTorch and MMEngine:

In addition to these basic components, you can also use **executor** to easily combine and configure various training techniques, such as enabling mixed-precision training and gradient accumulation (see *OptimWrapper*), configuring the learning rate decay curve (see *Metrics & Evaluator*), and etc.

CHAPTER

FOUR

RESUME TRAINING

Resuming training means continuing training from the state saved from some previous training, where the state includes the model's weights, the state of the optimizer and the state of parameter scheduler.

4.1 Automatically resume training

Users can set the `resume` parameter of *Runner* to enable automatic training resumption. When `resume` is set to `True`, the Runner will try to resume from the latest checkpoint in `work_dir` automatically. If there is a latest checkpoint in `work_dir` (e.g. the training was interrupted during the last training), the training will be resumed from that checkpoint, otherwise (e.g. the last training did not have time to save the checkpoint or a new training task is started) the training will restart. Here is an example of how to enable automatic training resumption.

```
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()
```

4.2 Specify the checkpoint path

If you want to specify the path to resume training, you need to set `load_from` in addition to `resume=True`. Note that if only `load_from` is set without `resume=True`, then only the weights in the checkpoint will be loaded and training will be restarted, instead of continuing with the previous state.

```
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

FIVE

SPEED UP TRAINING

5.1 Distributed Training

MMEngine supports training models with CPU, single GPU, multiple GPUs in single machine and multiple machines. When multiple GPUs are available in the environment, we can use the following command to enable multiple GPUs in single machine or multiple machines to shorten the training time of the model.

- multiple GPUs in single machine

  Assuming the current machine has 8 GPUs, you can enable multiple GPUs training with the following command:

```
python -m torch.distributed.launch --nproc_per_node=8 examples/train.py --launcher␣
↪pytorch
```

  If you need to specify the GPU index, you can set the `CUDA_VISIBLE_DEVICES` environment variable, e.g. use the 0th and 3rd GPU.

```
CUDA_VISIBLE_DEVICES=0,3 python -m torch.distributed.launch --nproc_per_node=2␣
↪examples/train.py --launcher pytorch
```

- multiple machines

  Assume that there are 2 machines connected with ethernet, you can simply run following commands.

  On the first machine:

```
python -m torch.distributed.launch \
    --nnodes 8 \
    --node_rank 0 \
    --master_addr 127.0.0.1 \
    --master_port 29500 \
    --nproc_per_node=8 \
    examples/train.py --launcher pytorch
```

  On the second machine:

```
python -m torch.distributed.launch \
    --nnodes 8 \
    --node_rank 1 \
    --master_addr 127.0.0.1 \
    --master_port 29500 \
    --nproc_per_node=8 \
```

If you are running MMEngine in a slurm cluster, simply run the following command to enable training for 2 machines and 16 GPUs.

```
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"
```

5.2 Mixed Precision Training

Nvidia introduced the Tensor Core unit into the Volta and Turing architectures to support FP32 and FP16 mixed precision computing. With automatic mixed precision training enabled, some operators operate at FP16 and the rest operate at FP32, which reduces training time and storage requirements without changing the model or degrading its training precision, thus supporting training with larger batch sizes, larger models, and larger input sizes.

PyTorch officially supports amp from 1.6. If you are interested in the implementation of automatic mixing precision, you can refer to Mixed Precision Training.

MMEngine provides the wrapper *AmpOptimWrapper* for auto-mixing precision training, just set `type='AmpOptimWrapper'` in `optim_wrapper` to enable auto-mixing precision training, no other code changes are needed.

```
runner = Runner(
    model=ResNet18(),
    work_dir='./work_dir',
    train_dataloader=train_dataloader_cfg,
    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

SIX

SAVE MEMORY ON GPU

Memory capacity is critical in deep learning training and inference and determines whether the model can run successfully. Common memory saving approaches include:

- Gradient Accumulation

  Gradient accumulation is the mechanism that runs at a configured number of steps accumulating the gradients instead of updating parameters, after which the network parameters are updated and the gradients are cleared. With this technique of delayed parameter update, the result is similar to those scenarios using a large batch size, while the memory of activation can be saved. However, it should be noted that if the model contains a batch normalization layer, using gradient accumulation will impact performance.

- Gradient Checkpointing

  Gradient checkpointing is a time-for-space method that compresses the model by reducing the number of saved activations, however, the unstored activations must be recomputed when calculating the gradient. The corresponding functionality has been implemented in the `torch.utils.checkpoint` package. The implementation can be briefly concluded as that, in the forward phase, the forward function passed to the checkpoint runs in `torch.no_grad` mode and saves only the input and the output of the forward function. Then recalculates its intermediate activations in the backward phase.

- Large Model Training Techniques

  Recent research has shown that training a large model would be helpful to improve performance, but training a model at such a scale requires huge resources, and it is hard to store the entire model in the memory of a single graphics card. Therefore large model training techniques, typically such as DeepSpeed ZeRO and the Fully Shared Data Parallel (FSDP) technique introduced in FairScale are introduced. These techniques allow slicing the parameters, gradients, and optimizer states among the parallel processes, while still maintaining the simplicity of the data parallelism.

MMEngine now supports gradient accumulation and large model training FSDP techniques, and the usages are described as follows.

6.1 Gradient Accumulation

The configuration can be written in this way:

```
optim_wrapper_cfg = dict(
    type='OptimWrapper',
    optimizer=dict(type='SGD', lr=0.001, momentum=0.9),
    # update every four times
    accumulative_counts=4)
```

The full example working with `Runner` is as follows.

```
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()
```

6.2 Large Model Training

FSDP is officially supported from PyTorch 1.11. The config can be written in this way:

```
# located in cfg file
model_wrapper_cfg=dict(type='MMFullyShardedDataParallel', cpu_offload=True)
```

The full example working with Runner is as follows.

```
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)
```

(continues on next page)

(continued from previous page)

```
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()
```

Please be noted that FSDP works only in distributed training environments.

CHAPTER

SEVEN

TRAIN A GAN

Coming soon. Please refer to chinese documentation.

CHAPTER

EIGHT

RUNNER

Welcome to the tutorial of runner, the core of MMEngine's user interface!

The runner, as an "integrator" in MMEngine, covers all aspects of the framework and shoulders the responsibility of organizing and scheduling nearly all modules. Therefore, the code logic in it has to take into account various situations, making it relatively hard to understand. But **don't worry**! In this tutorial, we will leave out some messy details and have a quick overview of commonly used APIs, functionalities, and examples. Hopefully, this should provide you with a clear and easy-to-understand user interface. After reading through this tutorial, you will be able to:

- Master the common usage and configuration of the runner
- Learn the best practice - writing config files - of the runner
- Know about the basic dataflow and execution order
- Feel by yourself the advantages of using runner (perhaps)

8.1 Example codes of the runner

To build your training pipeline with a runner, there are typically two ways to get started:

- Refer to runner's *API documentation* for argument-by-argument configuration
- Make your custom modifications based on some existing configurations, such as *Getting started in 15 minutes* and downstream repositories like MMDet

Pros and cons lie in both approaches. For the former one, beginners may be lost in a vast number of configurable arguments. For the latter one, beginners may find it hard to get a good reference, since neither an over-simplified nor an over-detailed reference is conducive to them.

We argue that the key to learning runner is using it as a memo. You should remember its most commonly used arguments and only focus on those less used when in need, since default values usually work fine. In the following, we will provide a beginner-friendly example to illustrate the most commonly used arguments of the runner, along with advanced guidelines for those less used.

8.1.1 A beginer-friendly example

Hint: In this tutorial, we hope you can focus more on overall architecture instead of implementation details. This "top-down" way of thinking is exactly what we advocate. Don't worry, you will definitely have plenty of opportunities and guidance afterward to focus on modules you want to improve.

```python
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import Dataset

from mmengine.model import BaseModel
from mmengine.evaluator import BaseMetric
from mmengine.registry import MODELS, DATASETS, METRICS


@MODELS.register_module()
class MyAwesomeModel(BaseModel):
    def __init__(self, layers=4, activation='relu') -> None:
        super().__init__()
        if activation == 'relu':
            act_type = nn.ReLU
        elif activation == 'silu':
            act_type = nn.SiLU
        elif activation == 'none':
            act_type = nn.Identity
        else:
            raise NotImplementedError
        sequence = [nn.Linear(2, 64), act_type()]
        for _ in range(layers-1):
            sequence.extend([nn.Linear(64, 64), act_type()])
        self.mlp = nn.Sequential(*sequence)
        self.classifier = nn.Linear(64, 2)

    def forward(self, data, labels, mode):
        x = self.mlp(data)
        x = self.classifier(x)
        if mode == 'tensor':
            return x
        elif mode == 'predict':
            return F.softmax(x, dim=1), labels
        elif mode == 'loss':
            return {'loss': F.cross_entropy(x, labels)}


@DATASETS.register_module()
class MyDataset(Dataset):
    def __init__(self, is_train, size):
        self.is_train = is_train
        if self.is_train:
            torch.manual_seed(0)
```

(continues on next page)

(continued from previous page)

```
            self.labels = torch.randint(0, 2, (size,))
        else:
            torch.manual_seed(3407)
            self.labels = torch.randint(0, 2, (size,))
        r = 3 * (self.labels+1) + torch.randn(self.labels.shape)
        theta = torch.rand(self.labels.shape) * 2 * torch.pi
        self.data = torch.vstack([r*torch.cos(theta), r*torch.sin(theta)]).T

    def __getitem__(self, index):
        return self.data[index], self.labels[index]

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


@METRICS.register_module()
class Accuracy(BaseMetric):
    def __init__(self):
        super().__init__()

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

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

```
from torch.utils.data import DataLoader, default_collate
from torch.optim import Adam
from mmengine.runner import Runner


runner = Runner(
    # your model
    model=MyAwesomeModel(
        layers=2,
        activation='relu'),
    # work directory for saving checkpoints and logs
    work_dir='exp/my_awesome_model',

    # training data
    train_dataloader=DataLoader(
        dataset=MyDataset(
            is_train=True,
            size=10000),
        shuffle=True,
        collate_fn=default_collate,
```

(continues on next page)

(continued from previous page)

```
        batch_size=64,
        pin_memory=True,
        num_workers=2),
    # training configurations
    train_cfg=dict(
        by_epoch=True,   # display in epoch number instead of iterations
        max_epochs=10,
        val_begin=2,     # start validation from the 2nd epoch
        val_interval=1), # do validation every 1 epoch

    # OptimizerWrapper, new concept in MMEngine for richer optimization options
    # Default value works fine for most cases. You may check our documentations
    # for more details, e.g. 'AmpOptimWrapper' for enabling mixed precision
    # training.
    optim_wrapper=dict(
        optimizer=dict(
            type=Adam,
            lr=0.001)),
    # ParamScheduler to adjust learning rates or momentums during training
    param_scheduler=dict(
        type='MultiStepLR',
        by_epoch=True,
        milestones=[4, 8],
        gamma=0.1),

    # validation data
    val_dataloader=DataLoader(
        dataset=MyDataset(
            is_train=False,
            size=1000),
        shuffle=False,
        collate_fn=default_collate,
        batch_size=1000,
        pin_memory=True,
        num_workers=2),
    # validation configurations, usually leave it an empty dict
    val_cfg=dict(),
    # evaluation metrics and evaluator
    val_evaluator=dict(type=Accuracy),

    # following are advanced configurations, try to default when not in need
    # hooks are advanced usage, try to default when not in need
    default_hooks=dict(
        # the most commonly used hook for modifying checkpoint saving interval
        checkpoint=dict(type='CheckpointHook', interval=1)),

    # `luancher` and `env_cfg` responsible for distributed environment
    launcher='none',
    env_cfg=dict(
        cudnn_benchmark=False,   # whether enable cudnn_benchmark
        backend='nccl',   # distributed communication backend
        mp_cfg=dict(mp_start_method='fork')),  # multiprocessing configs
```

(continues on next page)

(continued from previous page)

```
    log_level='INFO',

    # load model weights from given path. None for no loading.
    load_from=None
    # resume training from the given path
    resume=False
)

# start training your model
runner.train()
```

8.1.2 Explanations on example codes

Really a long piece of code, isn't it! However, if you read through the above example, you may have already understood the training process in general even without knowing any implementation details, thanks to the compactness and readability of runner codes (probably). This is what MMEngine expects: a structured, modular, and standardized training process that allows for more reliable reproductions and clearer comparisons.

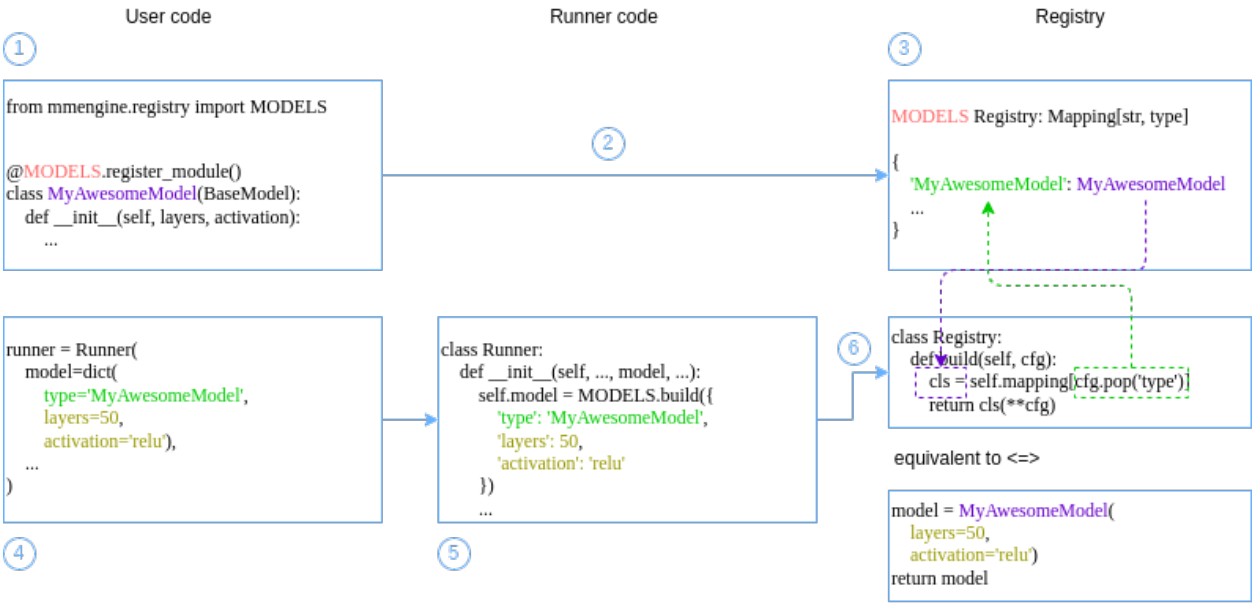
The above example may lead you to the following confusion:

Don't worry. As we mentioned before, **use runner as a memo**. The runner covers all aspects just to ensure you won't miss something important. You don't actually need to configure everything. The simple example in *15 minutes* still works fine, and it can be even more simplified by removing `val_evaluator`, `val_dataloader`, and `val_cfg` without breaking down. All configurable arguments are driven by your demands. Those not in your focus usually work fine by default.

Well, this is related to MMEngine's style. In MMEngine, we provide 2 different styles of runner construction: a) manual construction and b) construction via registry. If you are confused, the following example will give a good illustration:

```
from mmengine.model import BaseModel
from mmengine.runner import Runner
from mmengine.registry import MODELS # root registry for your custom model

@MODELS.register_module() # decorator for registration
class MyAwesomeModel(BaseModel): # your custom model
    def __init__(self, layers=18, activation='silu'):
        ...

# An example of manual construction
runner = Runner(
    model=dict(
        type='MyAwesomeModel',
        layers=50,
        activation='relu'),
    ...
)

# An example of construction via registry
model = MyAwesomeModel(layers=18, activation='relu')
runner = Runner(
    model=model,
    ...
)
```

Similar to the above example, most arguments in the runner accept both 2 types of inputs. They are conceptually equivalent. The difference is, in the former style, the module (passed in as a `dict`) will be built **in the runner when actually needed**, while in the latter style, the module has been built before being passed to the runner. The following figure illustrates the core idea of registry: it maintains the mapping between a module's **build method** and its **registry name**. If you want to learn more about the full usage of the registry, you are recommended to read *Registry* tutorial.



You might still be confused after the explanation. Why should we let the Runner build modules from dicts? What are the benefits? If you have such questions, then we are proud to answer: "Absolutely - no benefits!" In fact, module construction via registry only works to its best advantage when combined with a configuration file. It is still far from the best practice to write as the above example. We provide it here just to make sure you can read and get used to this writing style, which may facilitate your understanding of the actual best practice we will soon talk about - the configuration file. Stay tuned!

If you as a beginner do not immediately understand, it doesn't matter too much, because **manual construction** is still a good choice, especially for small-scale development and trial-and-error due to its being IDE friendly. However, you are still expected to read and get used to the writing style via registry, so that you can avoid being unnecessarily confused and puzzled in subsequent tutorials.

You will find extensive instructions and examples in those tutorials of the corresponding modules. You can also find all possible arguments in *Runner's API documentation*. If neither of the above resolves your query, you are always encouraged to start a topic in our discussion forum. It also helps us improve documentations.

Downstream repositories in OpenMMLab have widely adopted the writing style of config files. In the following chapter, we will show the usage of config files, the best practice of the runner in MMEngine, based on the above example with a slight variation.

8.2 Best practice of the Runner - config files

MMEngine provides a powerful config file system that supports Python syntax. You can **almost seamlessly** (which we will illustrate below) convert from the previous sample code to a config file. Here is an example:

```
# Save the following codes in example_config.py
# Almost copied from the above example, with some commas removed
model = dict(type='MyAwesomeModel',
    layers=2,
    activation='relu')
work_dir = 'exp/my_awesome_model'

train_dataloader = dict(
    dataset=dict(type='MyDataset',
        is_train=True,
        size=10000),
    sampler=dict(
        type='DefaultSampler',
        shuffle=True),
    collate_fn=dict(type='default_collate'),
    batch_size=64,
    pin_memory=True,
    num_workers=2)
train_cfg = dict(
    by_epoch=True,
    max_epochs=10,
    val_begin=2,
    val_interval=1)
optim_wrapper = dict(
    optimizer=dict(
        type='Adam',
        lr=0.001))
param_scheduler = dict(
    type='MultiStepLR',
    by_epoch=True,
    milestones=[4, 8],
    gamma=0.1)

val_dataloader = dict(
    dataset=dict(type='MyDataset',
        is_train=False,
        size=1000),
    sampler=dict(
        type='DefaultSampler',
        shuffle=False),
    collate_fn=dict(type='default_collate'),
    batch_size=1000,
    pin_memory=True,
    num_workers=2)
val_cfg = dict()
val_evaluator = dict(type='Accuracy')

default_hooks = dict(
```

(continues on next page)

(continued from previous page)

```
    checkpoint=dict(type='CheckpointHook', interval=1))
launcher = 'none'
env_cfg = dict(
    cudnn_benchmark=False,
    backend='nccl',
    mp_cfg=dict(mp_start_method='fork'))
log_level = 'INFO'
load_from = None
resume = False
```

Given the above config file, we can simply load configurations and run the training pipeline in a few lines of codes as follows:

```
from mmengine.config import Config
from mmengine.runner import Runner
config = Config.fromfile('example_config.py')
runner = Runner.from_cfg(config)
runner.train()
```

Note: Although it supports Python syntax, a valid config file needs to meet the condition that all variables must be Python built-in types such as `str`, `dict` and `int`. Therefore, the config system is highly dependent on the registry mechanism to enable construction from built-in types to other types such as `nn.Module`.

Note: When using config files, you typically don't need to manually register every module. For instance, all optimizers in `torch.optim` including `Adam` and `SGD` have already been registered in `mmengine.optim`. The rule of thumb is, try to directly access modules provided by PyTorch, and only start to register them manually after error occurs.

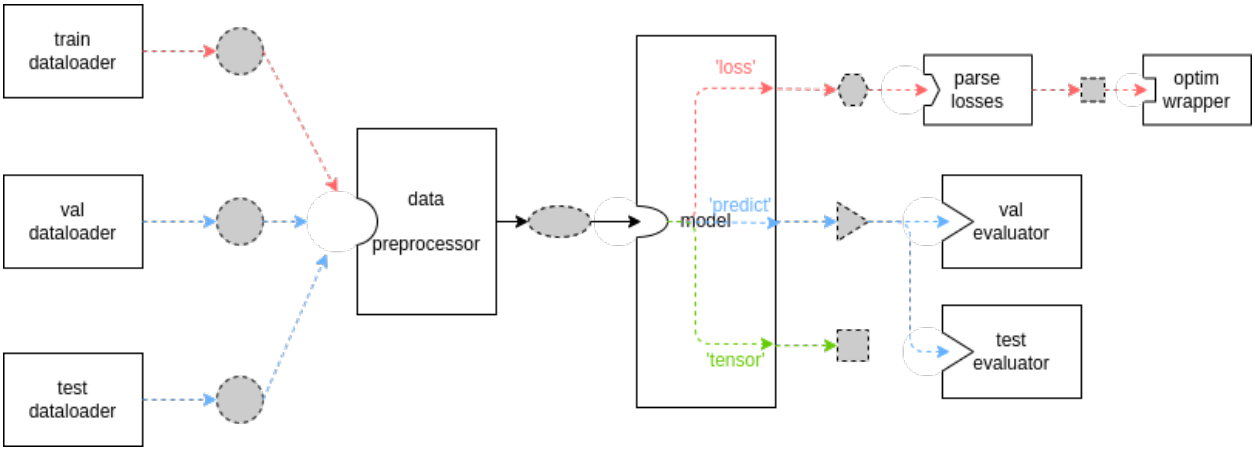
Note: When using config files, the implementations of your custom modules may be stored in separate files and thus not registered properly, which will lead to errors in the build process. You may find solutions in Registry tutorial by searching for `custom_imports`.

Writing config files of the runner has been widely adopted in downstream repositories in OpenMMLab projects. It has been a de facto convention and best practice. The config files are far more featured than illustrated above. You can refer to *Config tutorial* for more advanced features including keywords inheriting and overriding.

8.3 Basic dataflow

Hint: In this chapter, we'll dive deeper into the runner to illustrate dataflow and data format convention between modules managed by the runner. It may be relatively abstract and dry if you haven't built a training pipeline with MMEngine. Therefore, you are free to skip for now and read it in conjunction with practice in the future when in need.

Now let's dive **slightly deeper** into the runner, and illustrate the dataflow and data format convention under the hood (or, under the engine)!



The diagram above illustrates the **basic** dataflow of the runner, where the dashed border, gray filled shapes represent different data formats, while solid boxes represent modules/methods. Due to the great flexibility and extensibility of MMEngine, you can always inherit some key base classes and override their methods, so the above diagram doesn't always hold. It only holds when you are not customizing your own `Runner` or `TrainLoop`, and you are not overriding `train_step`, `val_step` or `test_step` method in your custom model. Actually, this is common for most tasks like detection and segmentation, as referred to *Model tutorial*.

Unfortunately, this is not possible. Although we did heavy type annotations in MMEngine, Python is still a highly dynamic programming language, and deep learning as a data-centric system needs to be flexible enough to deal with a wide range of complex data sources. You always have full freedom to decide when you need (and sometimes must) break type conventions. Therefore, when you are customizing your module (e.g. `val_evaluator`), you need to make sure its input is compatible with upstream (e.g. `model`) output and its output can be parsed by downstream. MMEngine puts the flexibility of handling data in the hands of the user, and thus also requires the user to ensure compatibility of dataflow, which, in fact, is not that difficult once you get started.

The uniformity of data formats has always been a problem in deep learning. We are trying to improve it in MMEngine in our own way. If you are interested, you can refer to *BaseDataset* and *BaseDataElement* - but please note that they are mainly geared towards advanced users.

For the basic dataflow shown in the diagram above, the data transfer between the above three modules can be represented by the following pseudo-code:

```
# training
for data_batch in train_dataloader:
    data_batch = data_preprocessor(data_batch)
    if isinstance(data_batch, dict):
        losses = model.forward(**data_batch, mode='loss')
    elif isinstance(data_batch, (list, tuple)):
        losses = model.forward(*data_batch, mode='loss')
    else:
        raise TypeError()

# validation
for data_batch in val_dataloader:
    data_batch = data_preprocessor(data_batch)
    if isinstance(data_batch, dict):
        outputs = model.forward(**data_batch, mode='predict')
    elif isinstance(data_batch, (list, tuple)):
        outputs = model.forward(**data_batch, mode='predict')
    else:
```

(continues on next page)

(continued from previous page)

```
        raise TypeError()
    evaluator.process(data_samples=outputs, data_batch=data_batch)
metrics = evaluator.evaluate(len(val_dataloader.dataset))
```

The key points of the above pseudo-code is:

- Outputs of data_preprocessor are passed to model **after unpacking**
- The `data_samples` argument of the evaluator receives the prediction results of the model, while the `data_batch` argument receives the raw data coming from dataloader

Though drawn separately in the diagram, data_preprocessor is a part of the model and thus can be found in *Model tutorial* in DataPreprocessor chapter.

In most cases, data_preprocessor needs no special attention or manual configuration. The default data_preprocessor will only do data transfer between host and GPU devices. However, if your model has incompatible inputs format with dataloader's output, you can also customize you own data_preprocessor for data formatting.

Image pre-processing such as crop and resize is more recommended in *data transforms module*, but for batch-related data transforms (e.g. batch-resize), it can be implemented here.

As described in *get started in 15 minutes*, you need to implement 3 data paths in your custom model's `forward` function to suit different pipelines for training, validation and testing. This is further discussed in *Model tutorial*.

Currently model outputs in "tensor" mode has not been officially used in runner. The "tensor" mode can output some intermediate results and thus facilitating debugging process.

The behavior of default `train_step`, `val_step` and `test_step` covers the dataflow from data_preprocessor to model outputs and optim_wrapper. The rest of the diagram will not be spoiled.

8.4 Why use the runner? (Optional reading)

Hint: Contents in this chapter will not teach you how to use the runner and MMEngine. If you are being pushed by your employer/advisor/DDL to work out a result in a few hours, it may not help you and you can feel free to skip it. However, we highly recommend taking time to read through this chapter, since it will help you better understand the aim and style of MMEngine.

Congratulations for reading through the runner tutorial, a long, long but kind of interesting (hope so) tutorial! Please believe that all of these - this tutorial, the runner, MMEngine - are intended to **make things easier for you**.

The runner is the "manager" of all modules in MMEngine. In the runner, all the distinct modules - whether visible ones like model and dataset, or obscure ones like logging, distributed environment and random seed - are getting organized and scheduled. The runner deals with the complex relationship between different modules and provides you with a clear, easy-to-understand and configurable interface. The benefits of this design are:

1. You can modify or add your codes without spoiling your whole codebase. For example, you may start with single GPU training and you can always add a few lines of configuration codes to enable multi GPUs or even multi nodes training.
2. You can continuously benefit from new features without worrying about backward compatibility. Mixed precision training, visualization, state of the art distributed training methods, various device backends... We will continue to absorb the best suggestions and cutting-edge technologies from the community while ensuring backward compatibility, and provide them to you in a clear interface.
3. You can focus on your own awesome ideas without being bothered by other annoying and irrelevant details. The default values will handle most cases.

So, MMEngine and the runner will truly make things easier for you. With only a little effort on migration, your code and experiments will evolve with MMEngine. With a little more effort, the config file system allows you to manage your data, model, and experiments more efficiently. Convenience and reliability are the aims we strive for.

The blue one, or the red one - are you prepared to use MMEngine?

8.5 Suggestions on next steps

If you want to:

Refer to *Model tutorial*

Refer to *Dataset and DataLoader tutorial*

Refer to *Evaluation tutorial*

Refer to *OptimWrapper tutorial*

Refer to *Parameter Scheduler tutorial*

- "Common Usage" section to the left contains more example codes
- "Advanced tutorials" to the left consists of more contents for experienced developers to make more flexible extensions to the training pipeline
- *Hook* provides some flexible modifications without spoiling your codes
- If none of the above solves your problem, you are always welcome to start a topic in our discussion forum!

CHAPTER

NINE

DATASET AND DATALOADER

Hint: If you have never been exposed to PyTorch's Dataset and DataLoader classes, you are recommended to read through PyTorch official tutorial to get familiar with some basic concepts.

Datasets and DataLoaders are necessary components in MMEngine's training pipeline. They are conceptually derived from and consistent with PyTorch. Typically, a dataset defines the quantity, parsing, and pre-processing of the data, while a dataloader iteratively loads data according to settings such as `batch_size`, `shuffle`, `num_workers`, etc. Datasets are encapsulated with dataloaders and they together constitute the data source.

In this tutorial, we will step through their usage in MMEngine runner from the outside (dataloader) to the inside (dataset) and give some practical examples. After reading through this tutorial, you will be able to:

- Master the configuration of dataloaders in MMEngine
- Learn to use existing datasets (e.g. those from `torchvision`) from config files
- Know about building and using your own dataset

9.1 Details on dataloader

Dataloaders can be configured in MMEngine's `Runner` with 3 arguments:

- `train_dataloader`: Used in `Runner.train()` to provide training data for models
- `val_dataloader`: Used in `Runner.val()` or in `Runner.train()` at regular intervals for model evaluation
- `test_dataloader`: Used in `Runner.test()` for the final test

MMEngine has full support for PyTorch native `DataLoader` objects. Therefore, you can simply pass your valid, already built dataloaders to the runner, as shown in *getting started in 15 minutes*. Meanwhile, thanks to the *Registry Mechanism* of MMEngine, those arguments also accept `dicts` as inputs, as illustrated in the following example (referred to as example 1). The keys in the dictionary correspond to arguments in DataLoader's init function.

```
runner = Runner(
    train_dataloader=dict(
        batch_size=32,
        sampler=dict(
            type='DefaultSampler',
            shuffle=True),
        dataset=torchvision.datasets.CIFAR10(...),
        collate_fn=dict(type='default_collate')
    )
)
```

When passed to the runner in the form of a dict, the dataloader will be lazily built in the runner when actually needed.

Note: For more configurable arguments of the `DataLoader`, please refer to PyTorch API documentation

Note: If you are interested in the details of the building procedure, you may refer to *build_dataloader*

You may find example 1 differs from that in *getting started in 15 minutes* in some arguments. Indeed, due to some obscure conventions in MMEngine, you can't seamlessly switch it to a dict by simply replacing `DataLoader` with `dict`. We will discuss the differences between our convention and PyTorch's in the following sections, in case you run into trouble when using config files.

9.1.1 sampler and shuffle

One obvious difference is that we add a `sampler` argument to the dict. This is because we **require `sampler` to be explicitly specified** when using a dict as a dataloader. Meanwhile, `shuffle` is also removed from `DataLoader` arguments, because it conflicts with `sampler` in PyTorch, as referred to in PyTorch DataLoader API documentation.

Note: In fact, `shuffle` is just a notation for convenience in PyTorch implementation. If `shuffle` is set to `True`, the dataloader will automatically switch to `RandomSampler`

With a `sampler` argument, codes in example 1 is **nearly** equivalent to code block below

```
from mmengine.dataset import DefaultSampler

dataset = torchvision.datasets.CIFAR10(...)
sampler = DefaultSampler(dataset, shuffle=True)

runner = Runner(
    train_dataloader=DataLoader(
        batch_size=32,
        sampler=sampler,
        dataset=dataset,
        collate_fn=default_collate
    )
)
```

Warning: The equivalence of the above codes holds only if: 1) you are training with a single process, and 2) no `randomness` argument is passed to the runner. This is due to the fact that `sampler` should be built after distributed environment setup to be correct. The runner will guarantee the correct order and proper random seed by applying lazy initialization techniques, which is only possible for dict inputs. Instead, when building a sampler manually, it requires extra work and is highly error-prone. Therefore, the code block above is just for illustration and definitely not recommended. We **strongly suggest passing `sampler` as a `dict`** to avoid potential problems.

9.1.2 DefaultSampler

The above example may make you wonder what a `DefaultSampler` is, why use it and whether there are other options. In fact, `DefaultSampler` is a built-in sampler in MMEngine which eliminates the gap between distributed and non-distributed training and thus enabling a seamless conversion between them. If you have the experience of using `DistributedDataParallel` in PyTorch, you may be impressed by having to change the `sampler` argument to make it correct. However, in MMEngine, you don't need to bother with this `DefaultSampler`.

`DefaultSampler` accepts the following arguments:

- `shuffle`: Set to `True` to load data in the dataset in random order
- `seed`: Random seed used to shuffle the dataset. Typically it doesn't require manual configuration here because the runner will handle it with `randomness` configuration
- `round_up`: When set this to `True`, this is the same behavior as setting `drop_last=False` in PyTorch `DataLoader`. You should take care of it when doing migration from PyTorch.

Note: For more details about `DefaultSampler`, please refer to *its API docs*

`DefaultSampler` handles most of the cases. We ensure that error-prone details such as random seeds are handled properly when you are using it in a runner. This prevents you from getting into troubles with distributed training. Apart from `DefaultSampler`, you may also be interested in *InfiniteSampler* for iteration-based training pipelines. If you have more advanced demands, you may want to refer to the codes of these two built-in samplers to implement your own one and register it to `DATA_SAMPLERS` registry.

```
@DATA_SAMPLERS.register_module()
class MySampler(Sampler):
    pass

runner = Runner(
    train_dataloader=dict(
        sampler=dict(type='MySampler'),
        ...
    )
)
```

9.1.3 The obscure collate_fn

Among the arguments of PyTorch `DataLoader`, `collate_fn` is often ignored by users, but in MMEngine you must pay special attention to it. When you pass the dataloader argument as a dict, MMEngine will use the built-in *pseudo_collate* by default, which is significantly different from that, default_collate, in PyTorch. Therefore, when doing a migration from PyTorch, you have to explicitly specify the `collate_fn` in config files to be consistent in behavior.

Note: MMEngine uses `pseudo_collate` as default value is mainly due to historical compatibility reasons. You don't have to look deeply into it. You can just know about it and avoid potential errors.

MMEngine provides 2 built-in `collate_fn`:

- `pseudo_collate`: Default value in MMEngine. It won't concatenate data through `batch` index. Detailed explanations can be found in *pseudo_collate API doc*

- `default_collate`: It behaves almost identically to PyTorch's `default_collate`. It will transfer data into `Tensor` and concatenate them through `batch` index. More details and slight differences from PyTorch can be found in *default_collate API doc*

If you want to use a custom `collate_fn`, you can register it to `COLLATE_FUNCTIONS` registry.

```
@COLLATE_FUNCTIONS.register_module()
def my_collate_func(data_batch: Sequence) -> Any:
    pass

runner = Runner(
    train_dataloader=dict(
        ...
        collate_fn=dict(type='my_collate_func')
    )
)
```

9.2 Details on dataset

Typically, datasets define the quantity, parsing, and pre-processing of the data. It is encapsulated in dataloader, allowing the latter to load data in batches. Since we fully support PyTorch `DataLoader`, the dataset is also compatible. Meanwhile, thanks to the registry mechanism, when a dataloader is given as a dict, its `dataset` argument can also be given as a dict, which enables lazy initialization in the runner. This mechanism allows for writing config files.

9.2.1 Use torchvision datasets

`torchvision` provides various open datasets. They can be directly used in MMEngine as shown in *getting started in 15 minutes*, where a `CIFAR10` dataset is used together with torchvision's built-in data transforms.

However, if you want to use the dataset in config files, registration is needed. What's more, if you also require data transforms in torchvision, some more registrations are required. The following example illustrates how to do it.

```
import torchvision.transforms as tvt
from mmengine.registry import DATASETS, TRANSFORMS
from mmengine.dataset.base_dataset import Compose

# register CIFAR10 dataset in torchvision
# data transforms should also be built here
@DATASETS.register_module(name='Cifar10', force=False)
def build_torchvision_cifar10(transform=None, **kwargs):
    if isinstance(transform, dict):
        transform = [transform]
    if isinstance(transform, (list, tuple)):
        transform = Compose(transform)
    return torchvision.datasets.CIFAR10(**kwargs, transform=transform)

# register data transforms in torchvision
DATA_TRANSFORMS.register_module('RandomCrop', module=tvt.RandomCrop)
DATA_TRANSFORMS.register_module('RandomHorizontalFlip', module=tvt.RandomHorizontalFlip)
DATA_TRANSFORMS.register_module('ToTensor', module=tvt.ToTensor)
DATA_TRANSFORMS.register_module('Normalize', module=tvt.Normalize)
```

(continues on next page)

(continued from previous page)

```
# specify in runner
runner = Runner(
    train_dataloader=dict(
        batch_size=32,
        sampler=dict(
            type='DefaultSampler',
            shuffle=True),
        dataset=dict(type='Cifar10',
            root='data/cifar10',
            train=True,
            download=True,
            transform=[
                dict(type='RandomCrop', size=32, padding=4),
                dict(type='RandomHorizontalFlip'),
                dict(type='ToTensor'),
                dict(type='Normalize', **norm_cfg)])
    )
)
```

Note: The above example makes extensive use of the registry mechanism and borrows the *Compose* module from MMEngine. If you urge to use torchvision dataset in your config files, you can refer to it and make some slight modifications. However, we recommend you borrow datasets from downstream repos such as MMDet, MMCls, etc. This may give you a better experience.

9.2.2 Customize your dataset

You are free to customize your own datasets, as you would with PyTorch. You can also copy existing datasets from your previous PyTorch projects. If you want to learn how to customize your dataset, please refer to PyTorch official tutorials

9.2.3 Use MMEngine BaseDataset

Apart from directly using PyTorch native `Dataset` class, you can also use MMEngine's built-in class `BaseDataset` to customize your own one, as referred to *BaseDataset tutorial*. It makes some conventions on the format of annotation files, which makes the data interface more unified and multi-task training more convenient. Meanwhile, `BaseDataset` can easily cooperate with built-in *data transforms* in MMEngine, which releases you from writing one from scratch.

Currently, `BaseDataset` has been widely used in downstream repos of OpenMMLab 2.0 projects.

CHAPTER

TEN

MODEL

10.1 Runner and model

As mentioned in basic dataflow, the dataflow between DataLoader, model and evaluator follows some rules. Don't remember clearly? Let's review it:

```
# Training process
for data_batch in train_dataloader:
    data_batch = model.data_preprocessor(data_batch, training=True)
    if isinstance(data_batch, dict):
        losses = model(**data_batch, mode='loss')
    elif isinstance(data_batch, (list, tuple)):
        losses = model(*data_batch, mode='loss')
    else:
        raise TypeError()
# Validation process
for data_batch in val_dataloader:
    data_batch = model.data_preprocessor(data_batch, training=False)
    if isinstance(data_batch, dict):
        outputs = model(**data_batch, mode='predict')
    elif isinstance(data_batch, (list, tuple)):
        outputs = model(**data_batch, mode='predict')
    else:
        raise TypeError()
    evaluator.process(data_samples=outputs, data_batch=data_batch)
metrics = evaluator.evaluate(len(val_dataloader.dataset))
```

In *runner tutorial*, we simply mentioned the relationship between DataLoader, model and evaluator, and introduced the concept of `data_preprocessor`. You may have a certain understanding of the model. However, during the running of Runner, the situation is far more complex than the above pseudo-code.

In order to focus your attention on the algorithm itself, and ignore the complex relationship between the model, DataLoader and evaluator, we designed *BaseModel*. In most cases, the only thing you need to do is to make your model inherit from `BaseModel`, and implement the `forward` as required to perform the training, testing, and validation process.

Before continuing reading the model tutorial, let's throw out two questions that we hope you will find the answers after reading the model tutorial:

1. When do we update the parameters of model? and how to update the parameters by a custom optimization process?
2. Why is the concept of data_preprocessor necessary? What functions can it perform?

10.2 Interface introduction

Usually, we should define a model to implement the body of the algorithm. In MMEngine, model will be managed by Runner, and need to implement some interfaces, such as `train_step`, `val_step`, and `test_step`. For high-level tasks like detection, classification, and segmentation, the interfaces mentioned above commonly implement a standard workflow. For example, `train_step` will calculate the loss and update the parameters of the model, and `val_step`/`test_step` will calculate the metrics and return the predictions. Therefore, MMEnine abstracts the *BaseModel* to implement the common workflow.

Benefits from the `BaseModel`, we only need to make the model inherit from `BaseModel`, and implement the `forward` function to perform the training, testing, and validation process.

Note: BaseModel inherits from *BaseModule*which can be used to initialize the model parameters dynamically.

forward: The arguments of `forward` need to match with the data given by DataLoader. If the DataLoader samples a tuple `data`, `forward` needs to accept the value of unpacked `*data`. If DataLoader returns a dict `data`, `forward` needs to accept the key-value of unpacked `**data`. `forward` also accepts `mode` parameter, which is used to control the running branch:

- `mode='loss'`: `loss` mode is enabled in training process, and `forward` returns a differentiable loss `dict`. Each key-value pair in loss `dict` will be used to log the training status and optimize the parameters of model. This branch will be called by `train_step`
- `mode='predict'`: `predict` mode is enabled in validation/testing process, and `forward` will return predictions, which matches with arguments of *process*. Repositories of OpenMMLab have a more strict rules. The predictions must be a list and each element of it must be a *BaseDataElement*. This branch will be called by `val_step`
- `mode='tensor'`: In `tensor` and `predict` modes, `forward` will return the predictions. The difference is that `forward` will return a `tensor` or a container or `tensor` which has not been processed by a series of post-process methods, such as non-maximum suppression (NMS). You can customize your post-process method after getting the result of `tensor` mode.

train_step: Get the loss `dict` by calling `forward` with `loss` mode. `BaseModel` implements a standard optimization process as follows:

```
def train_step(self, data, optim_wrapper):
    # See details in the next section
    data = self.data_preprocessor(data, training=True)
    # `loss` mode, return a loss dict. Actually train_step accepts
    #  both tuple  dict input, and unpack it with ** or *
    loss = self(**data, mode='loss')
    # Parse the loss dict and return the parsed losses for optimization
    # and log_vars for logging
    parsed_losses, log_vars = self.parse_losses()
    optim_wrapper.update_params(parsed_losses)  #
    return log_vars
```

val_step: Get the predictions by calling `forward` with `predict` mode.

```
def val_step(self, data, optim_wrapper):
    data = self.data_preprocessor(data, training=False)
    outputs = self(**data, mode='predict')
    return outputs
```

test_step: There is no difference between `val_step` and `test_step` in `BaseModel`. But we can customize it in the subclasses, for example, you can get validation loss in `val_step`.

Understand the interfaces of `BaseModel`, now we are able to come up with a more complete pseudo-code:

```
# training
for data_batch in train_dataloader:
    loss_dict = model.train_step(data_batch)
# validation
for data_batch in val_dataloader:
    preds = model.test_step(data_batch)
    evaluator.process(data_samples=outputs, data_batch=data_batch)
metrics = evaluator.evaluate(len(val_dataloader.dataset))
```

Great!, ignoring `Hook`, the pseudo-code above almost implements the main logic in *loop*! Let's go back to *15 minutes to get started with MMEngine*, we may truly understand what `MMResNet` has done:

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

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

    # train_step, val_step and test_step have been implemented in BaseModel.
    # We list the equivalent code here for better understanding
    def train_step(self, data, optim_wrapper):
        data = self.data_preprocessor(data)
        loss = self(*data, mode='loss')
        parsed_losses, log_vars = self.parse_losses()
        optim_wrapper.update_params(parsed_losses)
        return log_vars

    def val_step(self, data, optim_wrapper):
        data = self.data_preprocessor(data)
        outputs = self(*data, mode='predict')
        return outputs

    def test_step(self, data, optim_wrapper):
        data = self.data_preprocessor(data)
        outputs = self(*data, mode='predict')
        return outputs
```

Now, you may have a deeper understanding of dataflow, and can answer the first question in *Runner and model*.

`BaseModel.train_step` implements the standard optimization, and if we want to customize a new optimization

process, we can override it in the subclass. However, it is important to note that we need to make sure that `train_step` returns a loss dict.

10.3 DataPreprocessor

If your computer is equipped with a GPU (or other hardware that can accelerate training, such as MPS, IPU, etc.), when you run the *15 minutes tutorial*, you will see that the program is running on the GPU, but, when does `MMEngine` move the data and model from the CPU to the GPU?

In fact, the Runner will move the model to the specified device during the construction, while the data will be moved to the specified device at the `self.data_preprocessor(data)` mentioned in the code snippet of the previous section. The moved data will be further passed to the model.

Makes sense but it's weird, isn't it? At this point you may be wondering:

1. `MMResNet50` does not define `data_preprocessor`, but why it can still access `data_preprocessor` and move data to GPU?
2. Why `BaseModel` does not move data by `data = data.to(device)`, but needs the `DataPreprocessor` to move data?

The answer to the first question is that: `MMResNet50` inherit from `BaseModel`, and `super().__init__` will build a default `data_preprocessor` for it. The equivalent implementation of the default one is like this:

```
class BaseDataPreprocessor(nn.Module):
    def forward(self, data, training=True):  # ignore the training parameter here
        # suppose data given by CIFAR10 is a tuple. Actually
        # BaseDataPreprocessor could move various type of data
        # to target device.
        return tuple(_data.cuda() for _data in data)
```

`BaseDataPreprocessor` will move the data to the specified device.

Before answering the second question, let's think about a few more questions

1. Where should we perform normalization? *transform* or `Model`?

   It sounds reasonable to put it in transform to take advantage of Dataloader's multi-process acceleration, and in the model to move it to GPU to use GPU resources to accelerate normalization. However, while we are debating whether CPU normalization is faster than GPU normalization, the time of data moving from CPU to GPU is much longer than the former.

   In fact, for less computationally intensive operations like normalization, it takes much less time than data transferring, which has a higher priority for being optimized. If I could move the data to the specified device while it is still in `uint8` and before it is normalized (the size of normalized `float` data is 4 times larger than that of unit8), it would reduce the bandwidth and greatly improve the efficiency of data transferring. This "lagged" normalization behavior is one of the main reasons why we designed the `DataPreprocessor`. The data preprocessor moves the data first and then normalizes it.

2. How we implement the data augmentation like MixUp and Mosaic?

   Although it seems that MixUp and Mosaic are just special data transformations that should be implemented in transform. However, considering that these two transformations involve **fusing multiple images into one**, it would be very difficult to implement them in transform since the current paradigm of transform is to do various enhancements on **one** image. It would be hard to read additional images from dataset because the dataset is not accessible in the transform. However, if we implement Mosaic or Mixup based on the `batch_data` sampled from Dataloader, everything becomes easy. We can access multiple images at the same time, and we can easily perform the image fusion operation.

```python
class MixUpDataPreprocessor(nn.Module):
    def __init__(self, num_class, alpha):
        self.alpha = alpha

    def forward(self, data, training=True):
        data = tuple(_data.cuda() for _data in data)
        # Only perform MixUp in training mode
        if not training:
            return data

        label = F.one_hot(label)  # label to OneHot
        batch_size = len(label)
        index = torch.randperm(batch_size)  # Get the index of fused image
        img, label = data
        lam = np.random.beta(self.alpha, self.alpha)  # Fusion factor

        # MixUp
        img = lam * img + (1 - lam) * img[index, :]
        label = lam * batch_scores + (1 - lam) * batch_scores[index, :]
        # Since the returned label is onehot encoded, the `forward` of the
        # model should also be adjusted.
        return tuple(img, label)
```

Therefore, besides data transferring and normalization, another major function of `data_preprocessor` is BatchAugmentation. The modularity of the data preprocessor also helps us to achieve a free combination between algorithms and data augmentation.

3. What should we do if the data sampled from the DataLoader does not match the model input, should I modify the DataLoader or the model interface?

   The answer is: neither is appropriate. The ideal solution is to do the adaptation without breaking the existing interface between the model and the DataLoader. `DataPreprocessor` could also handle this, you can customize your `DataPreprocessor` to convert the incoming to the target type.

By now, You must understand the rationale of the data preprocessor and can confidently answer the two questions posed at the beginning of the tutorial! But you may still wonder what is the `optim_wrapper` passed to `train_step`, and how do the predictions returned by `test_step` and `val_step` relate to the evaluator. You will find more introduction in the *evaluation tutorial* and the *optimizer wrapper tutorial*.

CHAPTER

ELEVEN

EVALUATION

Coming soon. Please refer to chinese documentation.

CHAPTER

TWELVE

OPTIMWRAPPER

In previous tutorials of *runner* and *model*, we have more or less mentioned the concept of `OptimWrapper`, but we have not introduced why we need it and what are the advantages of `OptimWrapper` compared to Pytorch's native optimizer. In this tutorial, we will help you understand the advantages and demonstrate how to use the wrapper.

As its name suggests, `OptimWrapper` is a high-level abstraction of PyTorch's native optimizer, which provides a unified set of interfaces while adding more functionality. `OptimWrapper` supports different training strategies, including mixed precision training, gradient accumulation, and gradient clipping. We can choose the appropriate training strategy according to our needs. `OptimWrapper` also defines a standard process for parameter updating based on which users can switch between different training strategies for the same set of code.

12.1 OptimWrapper vs Optimizer

Now we use both the native optimizer of PyTorch and the OptimWrapper in MMEngine to perform single-precision training, mixed-precision training, and gradient accumulation to show the difference in implementations.

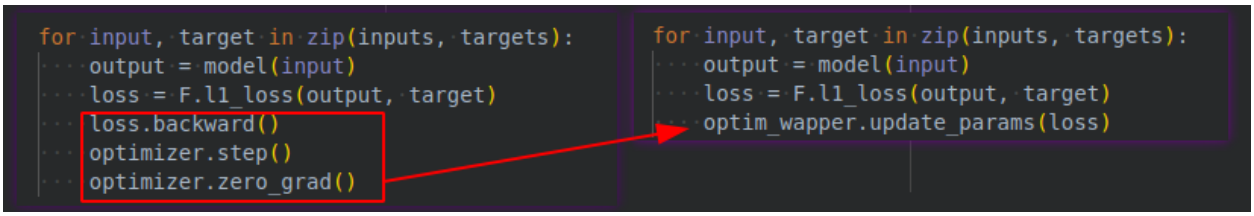
12.1.1 Model training

1.1 Single-precision training with SGD in PyTorch

```
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)
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 Single-precision training with OptimWrapper in MMEngine

```python
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)
```



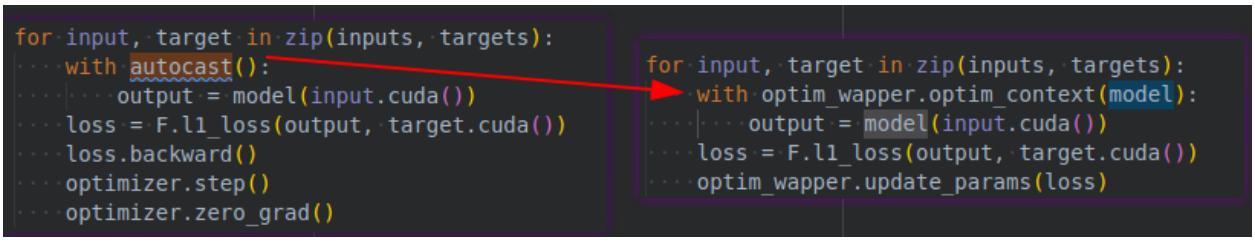
The `OptimWrapper.update_params` achieves the standard process for gradient computation, parameter updating, and gradient zeroing, which can be used to update the model parameters directly.

2.1 Mixed-precision training with SGD in PyTorch

```python
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 Mixed-precision training with OptimWrapper in MMEngine

```python
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)
```



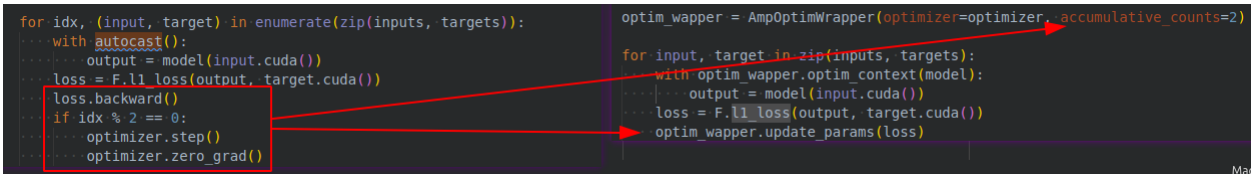
To enable mixed precision training, users need to use `AmpOptimWrapper.optim_context` which is similar to the `autocast` for enabling the context for mixed precision training. In addition, `AmpOptimWrapper.optim_context` can accelerate the gradient accumulation during the distributed training, which will be introduced in the next example.

3.1 Mixed-precision training and gradient accumulation with SGD in PyTorch

```
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 Mixed-precision training and gradient accumulation with OptimWrapper in MMEngine

```
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)
```



We only need to configure the `accumulative_counts` parameter and call the `update_params` interface to achieve the gradient accumulation function. Besides, in the distributed training scenario, if we configure the gradient accumulation with `optim_context` context enabled, we can avoid unnecessary gradient synchronization during the gradient accumulation step.

The OptimWrapper also provides a more fine-grained interface for users to customize with their own parameter update logics.

- `backward`: Accept a `loss` dictionary, and compute the gradient of parameters.
- `step`: Same as `optimizer.step`, and update the parameters.
- `zero_grad`: Same as `optimizer.zero_grad`, and zero the gradient of parameters

We can use the above interface to implement the same logic of parameters updating as the Pytorch optimizer.

```
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()
```

We can also configure a gradient clipping strategy for the OptimWrapper.

```
# based on torch.nn.utils.clip_grad_norm_ method
optim_wrapper = AmpOptimWrapper(
    optimizer=optimizer, clip_grad=dict(max_norm=1))

# based on torch.nn.utils.clip_grad_value_ method
optim_wrapper = AmpOptimWrapper(
    optimizer=optimizer, clip_grad=dict(clip_value=0.2))
```

12.1.2 Get learning rate/momentum

The OptimWrapper provides the `get_lr` and `get_momentum` for the convenience of getting the learning rate and momentum of the first parameter group in the optimizer.

```
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'])  # 0.01
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]}
```

12.1.3 Export/load state dicts

Similar to the optimizer, the OptimWrapper provides the `state_dict` and `load_state_dict` interfaces for exporting and loading the optimizer states. For the `AmpOptimWrapper`, it can export mixed-precision training parameters as well.

```
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_wrapper = OptimWrapper(optimizer=optimizer)
amp_optim_wrapper = AmpOptimWrapper(optimizer=optimizer)

# export state dicts
optim_state_dict = optim_wrapper.state_dict()
amp_optim_state_dict = amp_optim_wrapper.state_dict()
```

(continues on next page)

(continued from previous page)

```
print(optim_state_dict)
print(amp_optim_state_dict)
optim_wrapper_new = OptimWrapper(optimizer=optimizer)
amp_optim_wrapper_new = AmpOptimWrapper(optimizer=optimizer)

# load state dicts
amp_optim_wrapper_new.load_state_dict(amp_optim_state_dict)
optim_wrapper_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}}
```

12.1.4 Use multiple optimizers

Considering that algorithms like GANs usually need to use multiple optimizers to train the generator and the discriminator, MMEngine provides a container class called `OptimWrapperDict` to manage them. `OptimWrapperDict` stores the sub-OptimWrapper in the form of `dict`, and can be accessed and traversed just like a `dict`.

Unlike regular OptimWrapper, `OptimWrapperDict` does not provide methods such as `update_prarms`, `optim_context`, `backward`, `step`, etc. Therefore, it cannot be used directly to train models. We suggest implementing the logic of parameter updating by accessing the sub-OptimWarpper in `OptimWrapperDict` directly.

Users may wonder why not just use `dict` to manage multiple optimizers since `OptimWrapperDict` does not have training capabilities. Actually, the core function of `OptimWrapperDict` is to support exporting or loading the state dictionary of all sub-OptimWrapper and to support getting learning rates and momentums as well. Without `OptimWrapperDict`, MMEngine needs to do a lot of `if-else` in OptimWrapper to get the states of the `OptimWrappers`.

```
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]}
```

As shown in the above example, `OptimWrapperDict` exports learning rates and momentums for all OptimWrappers easily, and `OptimWrapperDict` can export and load all the state dicts in a similar way.

12.1.5 Configure the OptimWapper in Runner

We first need to configure the `optimizer` for the OptimWrapper. MMEngine automatically adds all optimizers in PyTorch to the `OPTIMIZERS` registry, and users can specify the optimizers they need in the form of a `dict`. All supported optimizers in PyTorch are listed here.

Now we take setting up a SGD OptimWrapper as an example.

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

Here we have set up an OptimWrapper with a SGD optimizer with the learning rate and momentum parameters as specified. Since OptimWrapper is designed for standard single precision training, we can also omit the `type` field in the configuration:

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

To enable mixed-precision training and gradient accumulation, we change `type` to `AmpOptimWrapper` and specify the `accumulative_counts` parameter.

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

Note: If you are new to reading the MMEngine tutorial and are not familiar with concepts such as *configs* and *registries*, it is recommended to skip the following advanced tutorials for now and read other documents first. Of course, if you already have a good understanding of this prerequisite knowledge, we highly recommend reading the advanced part which covers:

1. How to customize the learning rate, decay coefficient, and other parameters of the model parameters in the configuration of OptimWrapper.
2. how to customize the construction policy of the optimizer.

Apart from the pre-requisite knowledge of the configs and the registries, it is recommended to have a thorough understanding of the native construction of PyTorch optimizer before starting the advanced tutorials.

12.2 Advanced usages

PyTorch's optimizer allows different hyperparameters to be set for each parameter in the model, such as using different learning rates for the backbone and head for a classification model.

```
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}],
```

(continues on next page)

(continued from previous page)

```
    lr=0.01,
    momentum=0.9)
```

In the above example, we set a learning rate of 0.01 for the backbone, while another learning rate of 1e-3 for the head. Users can pass a list of dictionaries containing the different parts of the model's parameters and their corresponding hyperparameters to the optimizer, allowing for fine-grained adjustment of the model optimization.

In MMEngine, the optimizer wrapper constructor allows users to set hyperparameters in different parts of the model directly by setting the `paramwise_cfg` in the configuration file rather than by modifying the code of building the optimizer.

12.2.1 Set different hyperparamters for different types of parameters

The default optimizer wrapper constructor in MMEngine supports setting different hyperparameters for different types of parameters in the model. For example, we can set `norm_decay_mult=0` for `paramwise_cfg` to set the weight decay factor to 0 for the weight and bias of the normalization layer to implement the trick of not decaying the weight of the normalization layer as mentioned in the Bag of Tricks.

Here, we set the weight decay coefficient in all normalization layers (`head.bn`) in `ToyModel` to 0 as follows.

```
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
```

In addition to configuring the weight decay, `paramwise_cfg` of MMEngine's default optimizer wrapper constructor supports the following hyperparameters as well.

`lr_mult`: Learning rate for all parameters.

`decay_mult`: Decay coefficient for all parameters.

`bias_lr_mult`: Learning rate coefficient of the bias (excluding bias of normalization layer and offset of the deformable convolution).

`bias_decay_mult`: Weight decay coefficient of the bias (excluding bias of normalization layer and offset of the deformable convolution).

`norm_decay_mult`: Weight decay coefficient for weights and bias of the normalization layer.

`flat_decay_mult`: Weight decay coefficient of the one-dimension parameters.

`dwconv_decay_mult`: Decay coefficient of the depth-wise convolution.

`bypass_duplicate`: Whether to skip duplicate parameters, default to `False`.

`dcn_offset_lr_mult`: Learning rate of the deformable convolution.

12.2.2 Set different hyperparamters for different model modules

In addition, as shown in the PyTorch code above, in MMEngine we can also set different hyperparameters for any module in the model by setting `custom_keys` in `paramwise_cfg`.

If we want to set the learning rate and the decay coefficient to 0 for `backbone.layer0`, and set the learning rate to 0.001 for the rest of the modules in the `backbone`. At the same time, we want to keep all the learning rate to 0.001 for the `head` module. We can do it in this way:

```
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
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
```

(continues on next page)

(continued from previous page)

```
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
```

The state dictionary of the above model can be printed as the following:

```
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
```

Each field in `custom_keys` is defined as follows.

1. `'backbone': dict(lr_mult=1)`: Set the learning rate of the parameter whose name is prefixed with `backbone` to 1.
2. `'backbone.layer0': dict(lr_mult=0, decay_mult=0)`: Set the learning rate of the parameter with the prefix `backbone.layer0` to 0 and the decay coefficient to 0. This configuration has a higher priority than the first one.
3. `'head': dict(lr_mult=0.1)`: Set the learning rate of the parameter whose name is prefixed with `head` to 0.1.

12.2.3 Customize optimizer construction policies

Like other modules in MMEngine, the optimizer wrapper constructor is also managed by the registry. We can customize the hyperparameter policies by implementing custom optimizer wrapper constructors.

For example, we can implement an optimizer wrapper constructor called `LayerDecayOptimWrapperConstructor` that automatically set decreasing learning rates for layers of different depths of the model.

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

(continues on next page)

(continued from previous page)

```
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):

    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
```

When `add_params` is called for the first time, the `params` argument is an empty `list` and the `module` is the `ToyModel` instance. Please refer to the *Optimizer Wrapper Constructor Documentation* for detailed explanations on overloading.

Similarly, if we want to construct multiple optimizers, we also need to implement a custom constructor.

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

12.2.4 Adjust hyperparameters during training

The hyperparameters in the optimizer can only be set to a fixed value at the time it is constructed, and you cannot adjust parameters such as the learning rate during training by just using the optimizer wrapper. In MMEngine, we have implemented a parameter scheduler that allows the tuning of parameters during training. For the usage of the parameter scheduler, please refer to the *Parameter Scheduler*

CHAPTER

THIRTEEN

PARAMETER SCHEDULER

During neural network training, optimization hyperparameters (e.g. learning rate) are usually adjusted along with the training process. One of the simplest and most common learning rate adjustment strategies is multi-step learning rate decay, which reduces the learning rate to a fraction at regular intervals. PyTorch provides LRScheduler to implement various learning rate adjustment strategies. In MMEngine, we have extended it and implemented a more general *ParamScheduler*. It can adjust optimization hyperparameters such as learning rate and momentum. It also supports the combination of multiple schedulers to create more complex scheduling strategies.

13.1 Usage

We first introduce how to use PyTorch's `torch.optim.lr_scheduler` to adjust learning rate.

Here is an example which refers from PyTorch official documentation:

Initialize an ExponentialLR object, and call the `step` method after each training epoch.

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

`mmengine.optim.scheduler` supports most of PyTorch's learning rate schedulers such as `ExponentialLR`, `LinearLR`, `StepLR`, `MultiStepLR`, etc. Please refer to parameter scheduler API documentation for all of the supported schedulers.

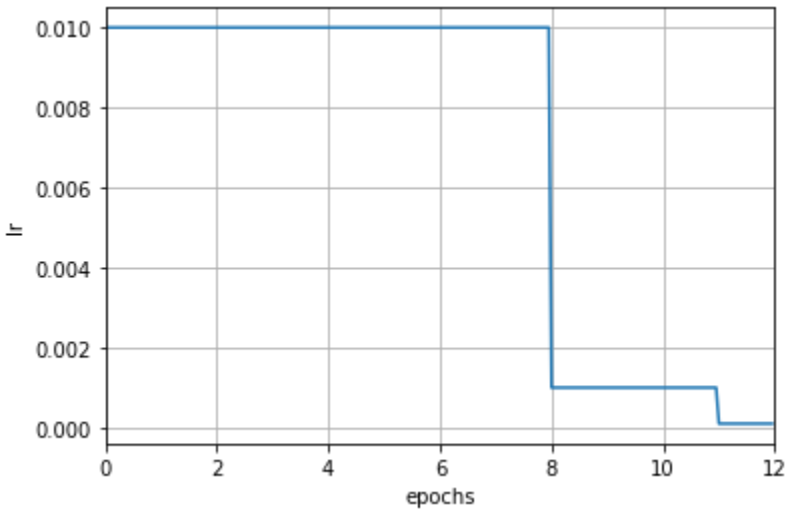
MMEngine also supports adjusting momentum with parameter schedulers. To use momentum schedulers, replace `LR` in the class name to `Momentum`, such as `ExponentialMomentumLinearMomentum`. Further, we implement the general parameter scheduler ParamScheduler, which is used to adjust the specified hyperparameters in the optimizer, such as weight_decay, etc. This feature makes it easier to apply some complex hyperparameter tuning strategies.

Different from the above example, MMEngine usually does not need to manually implement the training loop and call `optimizer.step()`. The runner will automatically manage the training progress and control the execution of the parameter scheduler through `ParamSchedulerHook`.

13.1.1 Use a single LRScheduler

If only one scheduler needs to be used for the entire training process, there is no difference with PyTorch's learning rate scheduler.

```
# build the scheduler manually
from torch.optim import SGD
from mmengine.runner import Runner
from mmengine.optim.scheduler import MultiStepLR

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

runner = Runner(
    model=model,
    optim_wrapper=dict(
        optimizer=optimizer),
    param_scheduler=param_scheduler,
    ...
    )
```



If using the runner with the registry and config file, we can specify the scheduler by setting the `param_scheduler` field in the config. The runner will automatically build a parameter scheduler based on this field:

```
# build the scheduler with config file
param_scheduler = dict(type='MultiStepLR', by_epoch=True, milestones=[8, 11], gamma=0.1)
```

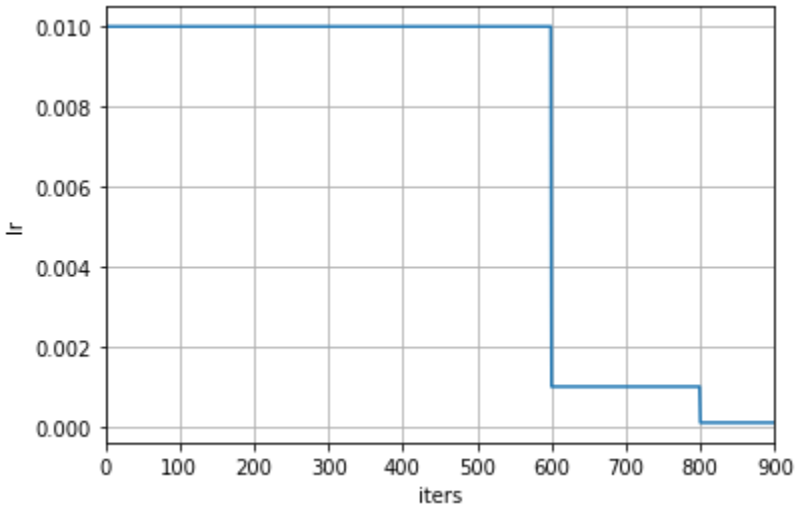
Note that the parameter `by_epoch` is added here, which controls the frequency of learning rate adjustment. When set to True, it means adjusting by epoch. When set to False, it means adjusting by iteration. The default value is True.

In the above example, it means to adjust according to epochs. At this time, the unit of the parameters is epoch. For example, [8, 11] in `milestones` means that the learning rate will be multiplied by 0.1 at the end of the 8 and 11 epoch.

When the frequency is modified, the meaning of the count-related settings of the scheduler will be changed accordingly. When `by_epoch=True`, the numbers in milestones indicate at which epoch the learning rate decay is performed, and when `by_epoch=False` it indicates at which iteration the learning rate decay is performed.

Here is an example of adjusting by iterations: At the end of the 600th and 800th iterations, the learning rate will be multiplied by 0.1 times.

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



If users want to use the iteration-based frequency while filling the scheduler config settings by epoch, MMEngine's scheduler also provides an automatic conversion method. Users can call the `build_iter_from_epoch` method and provide the number of iterations for each training epoch to construct a scheduler object updated by iterations:

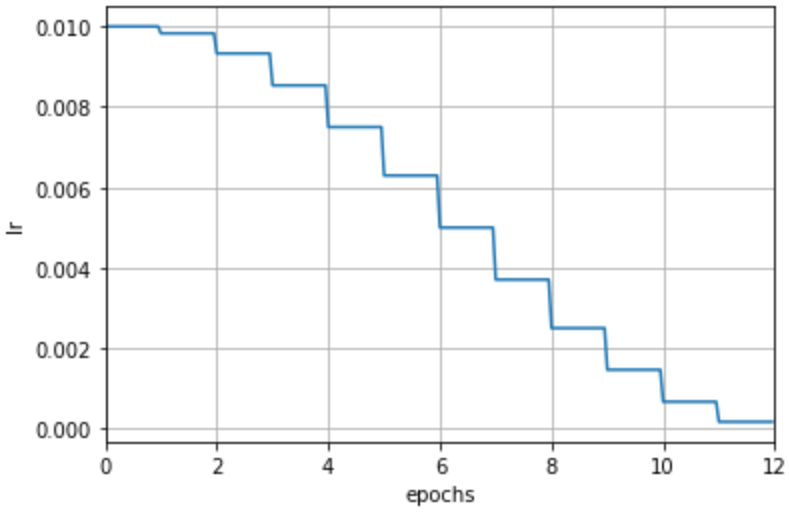
```
epoch_length = len(train_dataloader)
param_scheduler = MultiStepLR.build_iter_from_epoch(optimizer, milestones=[8, 11],␣
↪gamma=0.1, epoch_length=epoch_length)
```

If using config to build a scheduler, just add `convert_to_iter_based=True` to the field. The runner will automatically call `build_iter_from_epoch` to convert the epoch-based config to an iteration-based scheduler object:

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

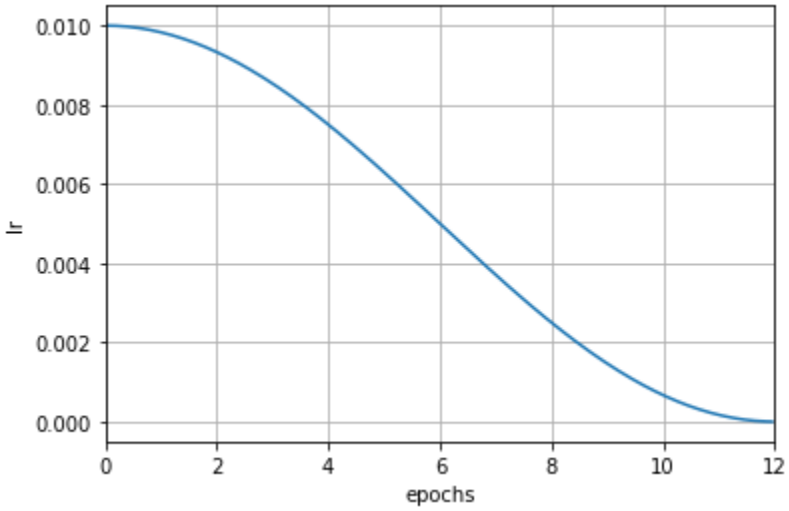
Below is a Cosine Annealing learning rate scheduler that is updated by epoch, where the learning rate is only modified after each epoch:

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



After automatically conversion, the learning rate is updated by iteration. As you can see from the graph below, the learning rate changes more smoothly.

```
param_scheduler = dict(type='CosineAnnealingLR', by_epoch=True, T_max=12, convert_to_
↪iter_based=True)
```



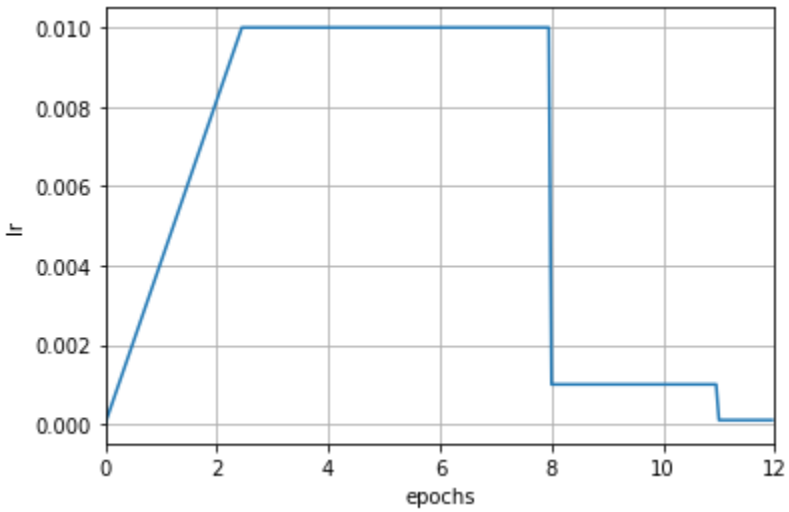
13.1.2 Combine multiple LRSchedulers (e.g. learning rate warm-up)

In the training process of some algorithms, the learning rate is not adjusted according to a certain scheduling strategy from beginning to end. The most common example is learning rate warm-up.

For example, in the first few iterations, a linear strategy is used to increase the learning rate from a small value to normal, and then another strategy is applied.

MMEngine supports combining multiple schedulers together. Just modify the `param_scheduler` field in the config file to a list of scheduler config, and the ParamSchedulerHook can automatically process the scheduler list. The following example implements learning rate warm-up.

```
param_scheduler = [
    # Linear learning rate warm-up scheduler
    dict(type='LinearLR',
         start_factor=0.001,
         by_epoch=False,  # Updated by iterations
         begin=0,
         end=50),  # Warm up for the first 50 iterations
    # The main LRScheduler
    dict(type='MultiStepLR',
         by_epoch=True,  # Updated by epochs
         milestones=[8, 11],
         gamma=0.1)
]
```



Note that the `begin` and `end` parameters are added here. These two parameters specify the **valid interval** of the scheduler. The valid interval usually only needs to be set when multiple schedulers are combined, and can be ignored when using a single scheduler. When the `begin` and `end` parameters are specified, it means that the scheduler only takes effect in the [begin, end) interval, and the unit is determined by the `by_epoch` parameter.

In the above example, the `by_epoch` of `LinearLR` in the warm-up phase is False, which means that the scheduler only takes effect in the first 50 iterations. After more than 50 iterations, the scheduler will no longer take effect, and the second scheduler, which is `MultiStepLR`, will control the learning rate. When combining different schedulers, the `by_epoch` parameter does not have to be the same for each scheduler.

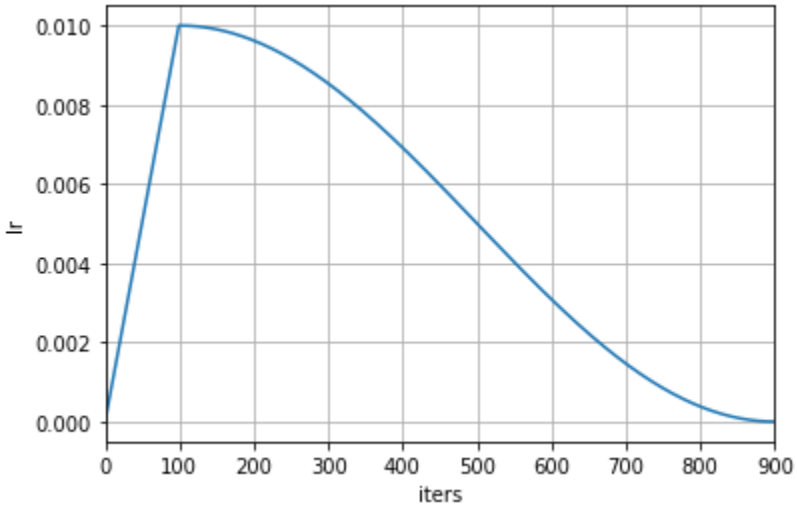
Here is another example

```
param_scheduler = [
    # Use a linear warm-up at [0, 100) iterations
    dict(type='LinearLR',
         start_factor=0.001,
         by_epoch=False,
         begin=0,
         end=100),
    # Use a cosine learning rate at [100, 900) iterations
    dict(type='CosineAnnealingLR',
         T_max=800,
         by_epoch=False,
```

(continues on next page)

(continued from previous page)

```
        begin=100,
        end=900)
]
```



The above example uses a linear learning rate warm-up for the first 100 iterations, and then uses a cosine annealing learning rate scheduler with a period of 800 from the 100th to the 900th iteration.

Users can combine any number of schedulers. If the valid intervals of two schedulers are not connected to each other which leads to an interval that is not covered, the learning rate of this interval remains unchanged. If the valid intervals of the two schedulers overlap, the adjustment of the learning rate will be triggered in the order of the scheduler config (similar with `ChainedScheduler`).

We recommend using different learning rate scheduling strategies in different stages of training to avoid overlapping of the valid intervals. Be careful If you really need to stack two schedulers overlapped. We recommend using learning rate visualization tool to visualize the learning rate after stacking, to avoid the adjustment not as expected.

13.2 How to adjust other hyperparameters

13.2.1 Momentum

Like learning rate, momentum is a schedulable hyperparameter in the optimizer's parameter group. The momentum scheduler is used in exactly the same way as the learning rate scheduler. Just add the momentum scheduler config to the list in the `param_scheduler` field.

Example:

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

(continues on next page)

(continued from previous page)

```
        end=1000)
]
```

13.2.2 Generic parameter scheduler

MMEngine also provides a set of generic parameter schedulers for scheduling other hyperparameters in the `param_groups` of the optimizer. Change `LR` in the class name of the learning rate scheduler to `Param`, such as `LinearParamScheduler`. Users can schedule the specific hyperparameters by setting the `param_name` variable of the scheduler.

Here is an example

```
param_scheduler = [
    dict(type='LinearParamScheduler',
         param_name='lr',  # adjust the 'lr' in `optimizer.param_groups`
         start_factor=0.001,
         by_epoch=False,
         begin=0,
         end=1000)
]
```

By setting the `param_name` to `'lr'`, this parameter scheduler is equivalent to `LinearLRScheduler`.

In addition to learning rate and momentum, users can also schedule other parameters in `optimizer.param_groups`. The schedulable parameters depend on the optimizer used. For example, when using the SGD optimizer with `weight_decay`, the `weight_decay` can be adjusted as follows:

```
param_scheduler = [
    dict(type='LinearParamScheduler',
         param_name='weight_decay',  # adjust 'weight_decay' in `optimizer.param_groups`
         start_factor=0.001,
         by_epoch=False,
         begin=0,
         end=1000)
]
```

CHAPTER

FOURTEEN

HOOK

Hook programming is a programming pattern in which a mount point is set in one or more locations of a program. When the program runs to a mount point, all methods registered to it at runtime are automatically called. Hook programming can increase the flexibility and extensibility of the program, since users can register custom methods to the mount point to be called without modifying the code in the program.

14.1 Built-in Hooks

MMEngine encapsules many ultilities as built-in hooks. These hooks are divided into two categories, namely default hooks and custom hooks. The former refers to those registered with the *Runner* by default, while the latter refers to those registered by the user on demand.

Each hook has a corresponding priority. At each mount point, hooks with higher priority are called earlier by the `Runner`. When sharing the same priority, the hooks are called in their registration order. The priority list is as follows.

- HIGHEST (0)
- VERY_HIGH (10)
- HIGH (30)
- ABOVE_NORMAL (40)
- NORMAL (50)
- BELOW_NORMAL (60)
- LOW (70)
- VERY_LOW (90)
- LOWEST (100)

default hooks

Name	Function	Priority
RuntimeInfoHook	update runtime information into message hub	VERY_HIGH (10)
IterTimerHook	Update the time spent during iteration into message hub	NORMAL (50)
DistSamplerSeedHook	Ensure distributed Sampler shuffle is active	NORMAL (50)
LoggerHook	Collect logs from different components of `Runner` and write them to terminal, JSON file, tensorboard and wandb .etc	BELOW_NORMAL (60)
ParamSchedulerHook	update some hyper-parameters of optimizer	LOW (70)
CheckpointHook	Save checkpoints periodically	VERY_LOW (90)

custom hooks

Name	Function	Priority
EMAHook	apply Exponential Moving Average (EMA) on the model during training	NORMAL (50)
EmptyCacheHook	Releases all unoccupied cached GPU memory during the process of training	NORMAL (50)
SyncBuffersHook	Synchronize model buffers at the end of each epoch	NORMAL (50)

Note: It is not recommended to modify the priority of the default hooks, as hooks with lower priority may depend on hooks with higher priority. For example, `CheckpointHook` needs to have a lower priority than ParamSchedulerHook so that the saved optimizer state is correct. Also, the priority of custom hooks defaults to `NORMAL (50)`.

The two types of hooks are set differently in the Runner, with the configuration of default hooks being passed to the `default_hooks` parameter of the Runner and the configuration of custom hooks being passed to the `custom_hooks` parameter, as follows.

```
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='EmptyCacheHook')]
runner = Runner(default_hooks=default_hooks, custom_hooks=custom_hooks, ...)
runner.train()
```

14.1.1 CheckpointHook

CheckpointHook saves the checkpoints at a given interval. In the case of distributed training, only the master process will save the checkpoints. The main features of `CheckpointHook` is as follows.

- Save checkpoints by interval, and support saving them by epoch or iteration
- Save the most recent checkpoints
- Save the best checkpoints
- Specify the path to save the checkpoints

For more features, please read the *CheckpointHook API documentation*.

The four features mentioned above are described below.

- Save checkpoints by interval, and support saving them by epoch or iteration

  Suppose we train a total of 20 epochs and want to save the checkpoints every 5 epochs, the following configuration will help us achieve this requirement.

```
# the default value of by_epoch is True
default_hooks = dict(checkpoint=dict(type='CheckpointHook', interval=5, by_
↪epoch=True))
```

  If you want to save checkpoints by iteration, you can set `by_epoch` to False and `interval=5` to save them every 5 iterations.

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

- Save the most recent checkpoints

  If you only want to keep a certain number of checkpoints, you can set the `max_keep_ckpts` parameter. When the number of checkpoints saved exceeds `max_keep_ckpts`, the previous checkpoints will be deleted.

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

  The above config shows that if a total of 20 epochs are trained, the model will be saved at epochs 5, 10, 15, and 20, but the checkpoint `epoch_5.pth` will be deleted at epoch 15, and at epoch 20 the checkpoint `epoch_10.pth` will be deleted, so that only the `epoch_15.pth` and `epoch_20.pth` will be saved.

- Save the best checkpoints

  If you want to save the best checkpoints of the validation set for the training process, you can set the `save_best` parameter. If set to `'auto'`, the current checkpoint are judged to be best based on the first evaluation metric of the validation set (the evaluation metrics returned by evaluator are an ordered dictionary).

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

  You can also directly specify the value of `save_best` as the evaluation metric, for example, in a classification task, you can specify `save_best='top-1'`, then the current checkpoint will be judged as best based on the value of `'top-1'`.

  In addition to the `save_best` parameter, other parameters related to saving the best checkpoint are `rule`, `greater_keys` and `less_keys`, which are used to imply whether its good to have large value or not. For example, if you specify `save_best='top-1'`, you can specify `rule='greater'` to imply that the larger the value, the better the checkpoint.

- Specify the path to save the checkpoints

  The checkpoints are saved in `work_dir` by default, but the path can be changed by setting `out_dir`.

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

LoggerHook collects logs from different components of `Runner` and write them to terminal, JSON file, tensorboard and wandb .etc.

If we want to output (or save) the logs every 20 iterations, we can set the `interval` parameter and configure it as follows.

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

If you are interested in how MMEngine manages logging, you can refer to *logging*.

14.1.2 ParamSchedulerHook

ParamSchedulerHook iterates through all optimizer parameter schedulers of the Runner and calls their `step` method to update the optimizer parameters in order. See *Parameter Schedulers* for more details about what are parameter schedulers.

`ParamSchedulerHook` is registered to the Runner by default and has no configurable parameters, so there is no need to configure it.

14.1.3 IterTimerHook

IterTimerHook is used to record the time taken to load data and iterate once.

`IterTimerHook` is registered to the Runner by default and has no configurable parameters, so there is no need to configure it.

14.1.4 DistSamplerSeedHook

DistSamplerSeedHook calls the `step` method of the Sampler during distributed training to ensure that the shuffle operation takes effect.

`DistSamplerSeedHook` is registered to the Runner by default and has no configurable parameters, so there is no need to configure it.

14.1.5 RuntimeInfoHook

RuntimeInfoHook will update the current runtime information (e.g. epoch, iter, max_epochs, max_iters, lr, metrics, etc.) to the message hub at different mount points in the Runner so that other modules without access to the Runner can obtain this information.

`RuntimeInfoHook` is registered to the Runner by default and has no configurable parameters, so there is no need to configure it.

14.1.6 EMAHook

EMAHook performs an exponential moving average operation on the model during training, with the aim of improving the robustness of the model. Note that the model generated by exponential moving average is only used for validation and testing, and does not affect training.

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

`EMAHook` uses *ExponentialMovingAverage* by default, with optional values of *StochasticWeightAverage* and *MomentumAnnealingEMA*. Other averaging strategies can be used by setting `ema_type`.

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

See *EMAHook API Reference* for more usage.

14.1.7 EmptyCacheHook

EmptyCacheHook calls `torch.cuda.empty_cache()` to release all unoccupied cached GPU memory. The timing of releasing memory can be controlled by setting parameters like `before_epoch`, `after_iter`, and `after_epoch`, meaning before the start of each epoch, after each iteration, and after each epoch respectively.

```
# The release operation is performed at the end of each epoch
custom_hooks = [dict(type='EmptyCacheHook', after_epoch=True)]
runner = Runner(custom_hooks=custom_hooks, ...)
runner.train()
```

14.1.8 SyncBuffersHook

SyncBuffersHook synchronizes the buffer of the model at the end of each epoch during distributed training, e.g. `running_mean` and `running_var` of the BN layer.

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

14.2 Customize Your Hooks

If the built-in hooks provided by MMEngine do not cover your demands, you are encouraged to customize your own hooks by simply inheriting the base *hook* class and overriding the corresponding mount point methods.

For example, if you want to check whether the loss value is valid, i.e. not infinite, during training, you can simply override the `after_train_iter` method as below. The check will be performed after each training iteration.

```
import torch
from mmengine.registry import HOOKS
from mmengine.hooks import Hook
@HOOKS.register_module()
class CheckInvalidLossHook(Hook):
```

(continues on next page)

(continued from previous page)

```
    """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!')
```

We simply pass the hook config to the `custom_hooks` parameter of the Runner, which will register the hooks when the Runner is initialized.

```
from mmengine.runner import Runner
custom_hooks = dict(
    dict(type='CheckInvalidLossHook', interval=50)
)
runner = Runner(custom_hooks=custom_hooks, ...)
runner.train()  # start training
```

Then the loss value are checked after iteration.

Note that the priority of the custom hook is `NORMAL (50)` by default, if you want to change the priority of the hook, then you can set the priority key in the config.

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

You can also set priority when defining classes.

```
@HOOKS.register_module()
class CheckInvalidLossHook(Hook):
    priority = 'ABOVE_NORMAL'
```

CHAPTER

FIFTEEN

REGISTRY

OpenMMLab supports a rich collection of algorithms and datasets, therefore, many modules with similar functionality are implemented. For example, the implementations of `ResNet` and `SE-ResNet` are based on the classes `ResNet` and `SEResNet`, respectively, which have similar functions and interfaces and belong to the model components of the algorithm library. To manage these functionally similar modules, MMEngine implements the registry. Most of the algorithm libraries in OpenMMLab use `registry` to manage their modules, including MMDetection, MMDetection3D, MMClassification and MMEditing, etc.

15.1 What is a registry

The *registry* in MMEngine can be considered as a union of a mapping table and a build function of modules. The mapping table maintains a mapping from strings to **classes or functions**, allowing the user to find the corresponding class or function with its name/notation. For example, the mapping from the string `"ResNet"` to the `ResNet` class. The module build function defines how to find the corresponding class or function based on a string and how to instantiate the class or call the function. For example, finding `nn.BatchNorm2d` and instantiating the `BatchNorm2d` module by the string `"bn"`, or finding the `build_batchnorm2d` function by the string `"build_batchnorm2d"` and then returning the result. The registries in MMEngine use the *build_from_cfg* function by default to find and instantiate the class or function corresponding to the string.

The classes or functions managed by a registry usually have similar interfaces and functionality, so the registry can be treated as an abstraction of those classes or functions. For example, the registry `MODELS` can be treated as an abstraction of all models, which manages classes such as `ResNet`, `SEResNet` and `RegNetX` and constructors such as `build_ResNet`, `build_SEResNet` and `build_RegNetX`.

15.2 Getting started

There are three steps required to use the registry to manage modules in the codebase.

1. Create a registry.
2. Create a build method for instantiating the class (optional because in most cases you can just use the default method).
3. Add the module to the registry

Suppose we want to implement a series of activation modules and want to be able to switch to different modules by just modifying the configuration without modifying the code.

Let's create a registry first.

```
from mmengine import Registry
# `scope` represents the domain of the registry. If not set, the default value is the␣
↪package name.
# e.g. in mmdetection, the scope is mmdet
# `locations` indicates the location where the modules in this registry are defined.
# The Registry will automatically import the modules when building them according to␣
↪these predefined locations.
ACTIVATION = Registry('activation', scope='mmengine', locations=['mmengine.models.
↪activations'])
```

The module `mmengine.models.activations` specified by `locations` corresponds to the `mmengine/models/activations.py` file. When building modules with registry, the ACTIVATION registry will automatically import implemented modules from this file. Therefore, we can implement different activation layers in the `mmengine/models/activations.py` file, such as `Sigmoid`, `ReLU`, and `Softmax`.

```
import torch.nn as nn

# use the register_module
@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):
        print('call Softmax.forward')
        return x
```

The key of using the registry module is to register the implemented modules into the `ACTIVATION` registry. With the `@ACTIVATION.register_module()` decorator added before the implemented module, the mapping between strings and classes or functions can be built and maintained by `ACTIVATION`. We can achieve the same functionality with `ACTIVATION.register_module(module=ReLU)` as well.

By registering, we can create a mapping between strings and classes or functions via `ACTIVATION`.

```
print(ACTIVATION.module_dict)
# {
```

(continues on next page)

(continued from previous page)

```
#     'Sigmoid': __main__.Sigmoid,
#     'ReLU': __main__.ReLU,
#     'Softmax': __main__.Softmax
# }
```

Note: The key to trigger the registry mechanism is to make the module imported. There are three ways to register a module into the registry

1. Implement the module in the `locations`. The registry will automatically import modules in the predefined locations. This is to ease the usage of algorithm libraries so that users can directly use `REGISTRY.build(cfg)`.
2. Import the file manually. This is common when developers implement a new module in/out side the algorithm library.
3. Use `custom_imports` field in config. Please refer to *Importing custom Python modules* for more details.

Once the implemented module is successfully registered, we can use the activation module in the configuration file.

```
import torch

input = torch.randn(2)

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

We can switch to `ReLU` by just changing this configuration.

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

If we want to check the type of input parameters (or any other operations) before creating an instance, we can implement a build method and pass it to the registry to implement a custom build process.

Create a `build_activation` function.

```
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
```

Pass the `buid_activation` to `build_func`.

```
ACTIVATION = Registry('activation', build_func=build_activation, scope='mmengine',␣
↪locations=['mmengine.models.activations'])

@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)
```

Note: In the above example, we demonstrate how to customize the method of building an instance of a class using the `build_func`. This is similar to the default `build_from_cfg` method. In most cases, using the default method will be fine.

MMEngine's registry can register classes as well as functions.

```
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)
```

15.3 Advanced usage

The registry in MMEngine supports hierarchical registration, which enables cross-project calls, meaning that modules from one project can be used in another project. Though there are other ways to implement this, the registry provides a much easier solution.

To easily make cross-library calls, MMEngine provides twenty root registries, including:

- RUNNERS: the registry for Runner.
- RUNNER_CONSTRUCTORS: the constructors for Runner.
- LOOPS: manages training, validation and testing processes, such as `EpochBasedTrainLoop`.
- HOOKS: the hooks, such as `CheckpointHook`, and `ParamSchedulerHook`.
- DATASETS: the datasets.
- DATA_SAMPLERS: `Sampler` of `DataLoader`, used to sample the data.

- TRANSFORMS: various data preprocessing methods, such as `Resize`, and `Reshape`.
- MODELS: various modules of the model.
- MODEL_WRAPPERS: model wrappers for parallelizing distributed data, such as `MMDistributedDataParallel`.
- WEIGHT_INITIALIZERS: the tools for weight initialization.
- OPTIMIZERS: registers all `Optimizers` and custom `Optimizers` in PyTorch.
- OPTIM_WRAPPER: the wrapper for Optimizer-related operations such as `OptimWrapper`, and `AmpOptimWrapper`.
- OPTIM_WRAPPER_CONSTRUCTORS: the constructors for optimizer wrappers.
- PARAM_SCHEDULERS: various parameter schedulers, such as `MultiStepLR`.
- METRICS: the evaluation metrics for computing model accuracy, such as `Accuracy`.
- EVALUATOR: one or more evaluation metrics used to calculate the model accuracy.
- TASK_UTILS: the task-intensive components, such as `AnchorGenerator`, and `BboxCoder`.
- VISUALIZERS: the management drawing module that draws prediction boxes on images, such as `DetVisualizer`.
- VISBACKENDS: the backend for storing training logs, such as `LocalVisBackend`, and `TensorboardVisBackend`.
- LOG_PROCESSORS: controls the log statistics window and statistics methods, by default we use `LogProcessor`. You may customize `LogProcessor` if you have special needs.

15.3.1 Use the module of the parent node

Let's define a `RReLU` module in `MMEngine` and register it to the `MODELS` root registry.

```
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
```

Now suppose there is a project called `MMAlpha`, which also defines a `MODELS` and sets its parent node to the `MODELS` of `MMEngine`, which creates a hierarchical structure.

```
from mmengine import Registry, MODELS as MMENGINE_MODELS

MODELS = Registry('model', parent=MMENGINE_MODELS, scope='mmalpha', locations=['mmalpha.
↪models'])
```

The following figure shows the hierarchy of `MMEngine` and `MMAlpha`.

The *count_registered_modules* function can be used to print the modules that have been registered to MMEngine and their hierarchy.

```
from mmengine.registry import count_registered_modules

count_registered_modules()
```

We define a customized `LogSoftmax` module in `MMAlpha` and register it to the `MODELS` in `MMAlpha`.

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

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

Here we use the `LogSoftmax` in the configuration of `MMAlpha`.

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

We can also use the modules of the parent node `MMEngine` here in the `MMAlpha`.

```
model = MODELS.build(cfg=dict(type='RReLU', lower=0.2))
# scope is optional
model = MODELS.build(cfg=dict(type='mmengine.RReLU'))
```

If no prefix is added, the `build` method will first find out if the module exists in the current node and return it if there is one. Otherwise, it will continue to look up the parent nodes or even the ancestor node until it finds the module. If the same module exists in both the current node and the parent nodes, we need to specify the `scope` prefix to indicate that we want to use the module of the parent nodes.

```
import torch

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

15.3.2 Use the module of a sibling node

In addition to using the module of the parent nodes, users can also call the module of a sibling node.

Suppose there is another project called `MMBeta`, which, like `MMAlpha`, defines `MODELS` and set its parent node to `MMEngine`.

```
from mmengine import Registry, MODELS as MMENGINE_MODELS

MODELS = Registry('model', parent=MMENGINE_MODELS, scope='mmbeta')
```

The following figure shows the registry structure of `MMAlpha` and `MMBeta`.

Now we call the modules of `MMAlpha` in `MMBeta`.

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

Calling a module of a sibling node requires the `scope` prefix to be specified in `type`, so the above configuration requires the prefix `mmalpha`.

However, if you need to call several modules of a sibling node, each with a prefix, this requires a lot of modification. Therefore, `MMEngine` introduces the *DefaultScope*, with which `Registry` can easily support temporary switching of the current node to the specified node.

If you need to switch the current node to the specified node temporarily, just set `_scope_` to the scope of the specified node in `cfg`.

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

CHAPTER

SIXTEEN

CONFIG

MMEngine implements an abstract configuration class (`Config`) to provide a unified configuration access interface for users. `Config` supports different type of configuration file, including `python`, `json` and `yaml`, and you can choose the type according to your preference. `Config` overrides some magic method, which could help you access the data stored in `Config` just like getting values from `dict`, or getting attributes from instances. Besides, `Config` also provides an inheritance mechanism, which could help you better organize and manage the configuration files.

Before starting the tutorial, let's download the configuration files needed in the tutorial (it is recommended to execute them in a temporary directory to facilitate deleting these files latter.):

```
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
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
```

16.1 Read the configuration file

`Config` provides a uniform interface `Config.fromfile()` to read and parse configuration files.

A valid configuration file should define a set of key-value pairs, and here are a few examples:

Python

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

Json

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

YAML

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

For the above three formats, assuming the file names are `config.py`, `config.json`, and `config.yml`. Loading these files with `Config.fromfile('config.xxx')` will return the same result, which contain `test_int`, `test_list` and `test_dict` 3 variables.

Let's take `config.py` as an example:

```python
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}}
```

16.2 How to use `Config`

After loading the configuration file, we can access the data stored in `Config` instance just like getting/setting values from `dict`, or getting/setting attributes from instances.

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

print(cfg['test_int'])
```

(continues on next page)

(continued from previous page)

```
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]
```

Note: The `dict` object parsed by `Config` will be converted to `ConfigDict`, and then we can access the value of the `dict` the same as accessing the attribute of an instance.

We can use the `Config` combination with the *Registry* to build registered instance easily.

Here is an example of defining optimizers in a configuration file.

`config_sgd.py`

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

Suppose we have defined a registry `OPTIMIZERS`, which includes various optimizers. Then we can build the optimizer as below

```
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
)
```

16.3 Inheritance between configuration files

Sometimes, the difference between two different configuration files is so small that only one field may be changed. Therefore, it's unwise to copy and paste everything only to modify one line, which makes it hard for us to locate the specific difference after a long time.

In another case, multiple configuration files may have the same batch of fields, and we have to copy and paste them in different configuration files. It will also be hard to maintain these fields in a long time.

We address these issues with inheritance mechanism, detailed as below.

16.3.1 Overview of inheritance mechanism

Here is an example to illustrate the inheritance mechanism.

`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)
```

Although we don't define `optimizer` in `resnet50.py`, since we wrote `_base_ = ['optimizer_cfg.py']`, it will inherit the fields defined in `optimizer_cfg.py`.

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

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

`_base_` is a reserved field for the configuration file. It specifies the inherited base files for the current file. Inheriting multiple files will get all the fields at the same time, but it requires that there are no repeated fields defined in all base files.

`runtime_cfg.py`

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

`resnet50_runtime.py`

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

In this case, reading the `resnet50_runtime.py` will give you 3 fields `model`, `optimizer`, and `gpu_ids`.

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

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

By this way, we can disassemble the configuration file, define some general configuration files, and inherit them in the specific configuration file. This could avoid defining a lot of duplicated contents in multiple configuration files.

16.3.2 Modify the inherited fields

Sometimes, we want to modify some of the fields in the inherited files. For example we want to modify the learning rate from 0.02 to 0.01 after inheriting `optimizer_cfg.py`.

In this case, you can simply redefine the fields in the new configuration file. Note that since the optimizer field is a dictionary, we only need to redefine the modified fields. This rule also applies to adding fields.

`resnet50_lr0.01.py`

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

After reading this configuration file, you can get the desired result.

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

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

For non-dictionary fields, such as integers, strings, lists, etc., they can be completely overwritten by redefining them. For example, the code block below will change the value of the `gpu_ids` to `[0]`.

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

16.3.3 Delete key in `dict`

Sometimes we not only want to modify or add the keys, but also want to delete them. In this case, we need to set `_delete_=True` in the target field(`dict`) to delete all the keys that do not appear in the newly defined dictionary.

`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)
```

At this point, `optimizer` will only have the keys `type` and `lr`. `momentum` and `weight_decay` will no longer exist.

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

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

16.3.4 Reference of the inherited file

Sometimes we want to reuse the field defined in _base_, we can get a copy of the corresponding variable by using {{_base_.xxxx}}:

refer_base_var.py

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

After parsing, the value of a becomes model defined in resnet50.py

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

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

We can use this way to get the variables defined in _base_ in the json, yaml, and python configuration files.

Although this way is general for all types of files, there are some syntactic limitations that prevent us from taking full advantage of the dynamic nature of the python configuration file. For example, if we want to modify a variable defined in _base_:

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

The Config is not able to parse such a configuration file (it will raise an error when parsing). The Config provides a more pythonic way to modify base variables for python configuration files.

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}
```

16.4 Dump the configuration file

The user may pass some parameters to modify some fields of the configuration file at the entry point of the training script. Therefore, we provide the dump method to export the changed configuration file.

Similar to reading the configuration file, the user can choose the format of the dumped file by using cfg.dump('config.xxx'). dump can also export configuration files with inheritance relationships, and the dumped files can be used independently without the files defined in _base_.

Based on the resnet50.py defined above, we can load and dump it like this:

```
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)
```

Similarly, we can dump configuration files in `json`, `yaml` format

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}}

In addition, `dump` can also dump `cfg` loaded from a dictionary.

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

dump_dict.py

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

16.5 Advanced usage

In this section, we'll introduce some advanced usage of the `Config`, and some tips that could make it easier for users to develop and use downstream repositories.

16.5.1 Predefined fields

Sometimes we need some fields in the configuration file, which are related to the path to the workspace. For example, we define a working directory in the configuration file that holds the models and logs for this set of experimental configurations. We expect to have different working directories for different configuration files. A common choice is to use the configuration file name directly as part of the working directory name. Taking `predefined_var.py` as an example:

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

Here `{{fileBasenameNoExtension}}` means the filename without suffix `.py` of the config file, and the variable in `{{}}` will be interpreted as `predefined_var`
```

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

```
./work_dir/predefined_var
```

Currently, there are 4 predefined fields referenced from the relevant fields defined in VS Code.

- `{{fileDirname}}` - the directory name of the current file, e.g. `/home/your-username/your-project/folder`
- `{{fileBasename}}` - the filename of the current file, e.g. `file.py`
- `{{fileBasenameNoExtension}}` - the filename of the current file without the extension, e.g. `file`
- `{{fileExtname}}` - the extension of the current file, e.g. `.py`

16.5.2 Modify the fields in command line

Sometimes we only want to modify part of the configuration and do not want to modify the configuration file itself. For example, if we want to change the learning rate during the experiment but do not want to write a new configuration file, the common practice is to pass the parameters at the command line to override the relevant configuration.

If we want to modify some internal parameters, such as the learning rate of the optimizer, the number of channels in the convolution layer etc., `Config` provides a standard procedure that allows us to modify the parameters at any level easily from the command line.

Training script:

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()
```

(continues on next page)

(continued from previous page)

```
    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()
```

The sample configuration file is as follows.

`example.py`

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

We can modify the internal fields from the command line by `.` For example, if we want to modify the learning rate, we only need to execute the script like this:

```
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}}
```

We successfully modified the learning rate from 0.01 to 0.1. If we want to change a list or a tuple, such as `in_channels` in the above example. We need to put double quotes around `()`, `[]` when assigning the value on the command line.

```
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}}
```

Note: The standard procedure only supports modifying String, Integer, Floating Point, Boolean, None, List, and Tuple fields from the command line. For the elements of list and tuple instance, each of them must be one of the above seven types.

Note: The behavior of `DictAction` is similar with `"extend"`. It stores a list, and extends each argument value to the list, like:

```
python demo_train.py ./example.py --cfg-options optimizer.type="Adam" --cfg-options␣
↪model.in_channels="[1, 1, 1]"
```

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

16.5.3 import the custom module

If we customize a module and register it into the corresponding registry, could we directly build it from the configuration file as the previous *section* does? The answer is "I don't know" since I'm not sure the registration process has been triggered. To solve this "unknown" case, `Config` provides the `custom_imports` function, to make sure your module could be registered as expected.

For example, we customize an optimizer:

```
from mmengine.registry import OPTIMIZERS

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

A matched config file:

my_module.py

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

To make sure `CustomOptim` will be registered, we should set the `custom_imports` field like this:

custom_imports.py

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

And then, once the `custom_imports` can be loaded successfully, we can build the `CustomOptim` from the `custom_imports.py`.

```
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>
```

16.5.4 Inherit configuration files across repository

It is annoying to copy a large number of configuration files when developing a new repository based on some existing repositories. To address this issue, `Config` support inherit configuration files from other repositories. For example, based on MMDetection, we want to develop a repository, we can use the MMDetection configuration file like this:

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',
]
```

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

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

`Config` will parse `mmdet::` to find mmdet package and inherits the specified configuration file. Actually, as long as the `setup.py` of the repository(package) conforms to MMEngine Installation specification, `Config` can use `{package_name}::` to inherit the specific configuration file.

16.5.5 Get configuration files across repository

`Config` also provides `get_config` and `get_model` to get the configuration file and the trained model from the downstream repositories.

The usage of `get_config` and `get_model` are similar to the previous section:

An example of `get_config`:

```
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
```

An example of `get_model`:

```
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'>
```

CHAPTER

SEVENTEEN

BASEDATASET

17.1 Introduction

The Dataset class in the algorithm toolbox is responsible for providing input data for the model during the training/testing process. The Dataset class in each algorithm toolbox under OpenMMLab projects has some common characteristics and requirements, such as the need for efficient internal data storage format, support for the concatenation of different datasets, dataset repeated sampling, and so on.

Therefore, **MMEngine** implements *BaseDataset* which provides some basic interfaces and implements some DatasetWrappers with the same interfaces. Most of the Dataset Classes in the OpenMMLab algorithm toolbox meet the interface defined by the `BaseDataset` and use the same DatasetWrappers.

The basic function of the BaseDataset is to load the dataset information. Here, we divide the dataset information into two categories. One is meta information, which represents the information related to the dataset itself and sometimes needs to be obtained by the model or other external components. For example, the meta information of the dataset generally includes the category information `classes` in the image classification task, since the classification model usually needs to record the category information of the dataset. The other is data information, which defines the file path and corresponding label information of specific data info. In addition, another function of the BaseDataset is to continuously send data into the data pipeline for data preprocessing.

17.1.1 The standard data annotation file

In order to unify the dataset interface of different tasks and facilitate multiple tasks training in one model, OpenMMLab formulate the **OpenMMLab 2.0 dataset format specification**. Dataset annotation files should conform to this specification, and the `BaseDataset` reads and parses data annotation files based on this specification. If the data annotation file provided by the user does not conform to the specified format, the user can choose to convert it to the specified format and use OpenMMLab's algorithm toolbox to conduct algorithm training and testing based on the converted data annotation file.

The OpenMMLab 2.0 dataset format specification states that annotation files must be in the format of `json` or `yaml`, `yml` or `pickle`, `pkl`. The dictionary stored in the annotation file must contain two fields, `metainfo` and `data_list`. The `metainfo` is a dictionary containing meta information about the dataset. The `data_list` is a list in which each element is a dictionary and the dictionary defines a raw data info. Each raw data info contains one or more training/test samples.

Here is an example of a JSON annotation file (where each raw data info contains only one training/test sample):

```
{
    'metainfo':
        {
            'classes': ('cat', 'dog'),
```

(continues on next page)

(continued from previous page)

```
            ...
        },
    'data_list':
        [
            {
                'img_path': "xxx/xxx_0.jpg",
                'img_label': 0,
                ...
            },
            {
                'img_path': "xxx/xxx_1.jpg",
                'img_label': 1,
                ...
            },
            ...
        ]
}
```

We assume that the data is stored in the following path:

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

17.1.2 The initialization process of the BaseDataset

The initialization process of the `BaseDataset` is shown as follows:

1. `load metainfo`: Obtain the meta information of the dataset. The meta information can be obtained from three sources with the priority from high to low:

- The dict of `metainfo` passed by the user in the `__init__()` function. The priority is high since the user can pass this argument when the `BaseDataset` is instantiated;
- The dict of `BaseDataset.METAINFO` in the class attributes of BaseDataset. The priority is medium since the user can change the class attributes `BaseDataset.METAINFO` in the custom dataset class;
- The dict of `metainfo` included in the annotation file. The priority is low since the annotation file is generally not changed.

If three sources have the same field, the source with the highest priority determines the value of the field. The priority comparison of these fields is: The fields in the `metainfo` dictionary passed by the user > The fields in the `BaseDataset.METAINFO` of BaseDataset > the fields in the `metainfo` of annotation file.

2. `join path`: Process the path of datainfo and annotating files;
3. `build pipeline`: Build data pipeline for the data preprocessing and data preparation;
4. `full init`: Fully initializes the BaseDataset. This step mainly includes the following operations:

- `load data list`: Read and parse the annotation files that meet the OpenMMLab 2.0 dataset format specification. In this step, the `parse_data_info()` method is called. This method is responsible for parsing each raw data info in the annotation file;
- `filter data` (optional): Filters unnecessary data based on `filter_cfg`, such as data samples that do not contain annotations. By default, there is no filtering operation, and downstream subclasses can override it according to their own needs.
- `get subset` (optional): Sample a subset of dataset based on a given index or an integer value, such as only the first 10 samples for training/testing. By default, all data samples are used.
- `serialize data` (optional): Serialize all data samples to save memory. Please see *Save memory* for more details. we serialize all data samples by default.

The `parse_data_info()` method in the BaseDataset is used to process a raw data info in the annotation file into one or more training/test data samples. The user needs to implement the `parse_data_info()` method if they want to customize dataset class.

17.1.3 The interface of BaseDataset

Once the BaseDataset is initialized, it supports `__getitem__` method to index a data info and `__len__` method to get the length of dataset, just like `torch.utils.data.Dataset`. The Basedataset provides the following interfaces:

- `metainfo`: Return the meta information with a dictionary value.
- `get_data_info(idx)`: Return the full data information of the given `idx`, and the return value is a dictionary.
- `__getitem__(idx)`: Return the results of data pipeline(The input data of model) of the given 'idx', and the return value is a dictionary.
- `__len__()`: Return the length of the dataset. The return value is an integer.
- `get_subset_(indices)`: Modify the original dataset class **in inplace** according to `indices`. If `indices` is `int`, then the original dataset class contains only the first few data samples. If `indices` is `Sequence[int]`, the raw dataset class contains data samples specified according to `Sequence[int]`.
- `get_subset(indices)`: Return a **new** sub-dataset class according to indices, i.e., re-copies a sub-dataset. If `indices` is `int`, the returned sub-dataset object contains only the first few data samples. If `indices` is `Sequence[int]`, the returned sub-dataset object contains the data samples specified according to `Sequence[int]`.

17.2 Customize dataset class based on BaseDataset

We can customize the dataset class based on BaseDataset, after we understand the initialization process of BaseDataset and the provided interfaces of BaseDataset.

17.2.1 Annotation files that meet the OpenMMLab 2.0 dataset format specification

As mentioned above, users can overload `parse_data_info()` to load annotation files that meet the OpenMMLab 2.0 dataset format specification. Here is an example of using BaseDataset to implement a specific dataset.

```
import os.path as osp

from mmengine.dataset import BaseDataset


class ToyDataset(BaseDataset):

    # Take the above annotation file as example. The raw_data_info represents a␣
↪dictionary in the data_list list:
    # {
    #    '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
```

Using Customized dataset class

The `ToyDataset` can be instantiated with the following configuration, once it has been defined:

```

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/',
```

(continues on next page)

(continued from previous page)

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

At the same time, the external interface provided by the BaseDataset can be used to access specific data sample information:

```
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]
# {
#     '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) ,
#     ...
# }

# The `get_subset` interface does not modify the original dataset class, i.e. make a␣
↪complete copy of it
sub_toy_dataset = toy_dataset.get_subset(1)
len(toy_dataset), len(sub_toy_dataset)
# 2, 1

# The `get_subset_` interface modify the original dataset class in inplace
toy_dataset.get_subset_(1)
len(toy_dataset)
# 1
```

Following the above steps, we can see how to customize a dataset based on the BaseDataset and how to use the customized dataset.

Customize dataset for videos

In the above examples, each raw data info of the annotation file contains only one training/test sample (usually in the image field). If each raw data info contains several training/test samples (usually in the video domain), we only need to ensure that the return value of `parse_data_info()` is `list[dict]`:

```
from mmengine.dataset import BaseDataset


class ToyVideoDataset(BaseDataset):

    # raw_data_info is still a dict, but it contains multiple samples
    def parse_data_info(self, raw_data_info):
        data_list = []

        ...

        for ... :

            data_info = dict()

            ...

            data_list.append(data_info)

        return data_list
```

The usage of `ToyVideoDataset` is similar to that of `ToyDataset`, which will not be repeated here.

17.2.2 Annotation files that do not meet the OpenMMLab 2.0 dataset format specification

For annotated files that do not meet the OpenMMLab 2.0 dataset format specification, there are two ways to use:

1. Convert the annotation files that do not meet the specifications into the annotation files that do meet the specifications, and then use the BaseDataset in the above way.
2. Implement a new dataset class that inherits from the `BaseDataset` and overloads the `load_data_list(self)`: function of the `BaseDataset` to handle annotation files that don't meet the specification and guarantee a return value of `list[dict]`, where each `dict` represents a data sample.

17.3 Other features of BaseDataset

The BaseDataset also contains the following features:

17.3.1 lazy init

When the BaseDataset is instantiated, the annotation file needs to be read and parsed, therefore it will take some time. However, in some cases, such as the visualization of prediction, only the meta information of the BaseDataset is required, and reading and parsing the annotation file may not be necessary. To save time on instantiating the BaseDataset in this case, the BaseDataset supports lazy init:

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

toy_dataset = ToyDataset(
    data_root='data/',
    data_prefix=dict(img_path='train/'),
    ann_file='annotations/train.json',
    pipeline=pipeline,
    # Pass the lazy_init variable in here
    lazy_init=True)
```

When `lazy_init=True`, the initialization of ToyDataset's only performs steps 1, 2, and 3 of the BaseDataset initialization process. At this time, `toy_dataset` was not fully initialized, since `toy_dataset` will not read and parse the annotation file. The `toy_dataset` only set the meta information of the dataset (`metainfo`).

Naturally, if you need to access specific data information later, you can manually call the `toy_dataset.full_init()` interface to perform the complete initialization process, during which the data annotation file will be read and parsed. Calling the `get_data_info (independence idx)`, `__len__ ()`, `__getitem__ (independence idx)`, `get_subset_ (indices)` and `get_subset(indices)` interface will also automatically call the `full_init()` interface to perform the full initialization process (only on the first call, later calls will not call the `full_init()` interface repeatedly):

```
# Full initialization
toy_dataset.full_init()

# After initialization, you can now get the data info
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) ,
#     ...
# }
```

Notice:

Performing full initialization by calling the `__getitem__()` interface directly carries some risks: If a dataset object is not fully initialized by setting `lazy_init=True` firstly, then it is directly sent to the dataloader. Different dataloader workers will read and parse the annotation file at the same time in the subsequent data reading process. Although this may work normally, it consumes a lot of time and memory. **Therefore, it is recommended to manually call the `full_init()` interface to perform the full initialization process before you need to access specific data.**

The above is not fully initialized by setting `lazy_init=True`, and then complete initialization according to the demand, called lazy init.

17.3.2 Save memory

In the specific process of reading data, the dataloader will usually prefetch data from multiple dataloader workers, and multiple workers have complete dataset object backup, so there will be multiple copies of the same `data_list` in the memory. In order to save this part of memory consumption, The `BaseDataset` can serialize `data_list` into memory in advance, so that multiple workers can share the same copy of `data_list`, so as to save memory.

By default, the BaseDataset stores the serialization of `data_list` into memory. It is also possible to control whether the data will be serialized into memory ahead of time by using the `serialize_data` argument (default is `True`) :

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

toy_dataset = ToyDataset(
    data_root='data/',
    data_prefix=dict(img_path='train/'),
    ann_file='annotations/train.json',
    pipeline=pipeline,
    # Pass the serialize data argument in here
    serialize_data=False)
```

The above example does not store the `data_list` serialization into memory in advance, so it is not recommended to instantiate the dataset class, when using the dataloader to open multiple dataloader workers to load the data.

17.4 DatasetWrappers

In addition to BaseDataset, MMEngine also provides several DatasetWrappers: `ConcatDataset`, `RepeatDataset`, `ClassBalancedDataset`. These dataset wrappers also support lazy init and have memory-saving features.

17.4.1 ConcatDataset

MMEngine provides a `ConcatDataset` wrapper to concatenate datasets in the following way:

```
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/'),
```

(continues on next page)

(continued from previous page)

```
    ann_file='annotations/val.json',
    pipeline=pipeline)

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

The above example combines the `train` set and the `val` set of the dataset into one large dataset.

17.4.2 RepeatDataset

MMEngine provides `RepeatDataset` wrapper to repeat a dataset several times, as follows:

```
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)
```

The above example samples the `train` set of the dataset five times.

17.4.3 ClassBalancedDataset

MMEngine provides `ClassBalancedDataset` wrapper to repeatedly sample the corresponding samples based on the frequency of category occurrence in the dataset.

Notice:

The `ClassBalancedDataset` wrapper assumes that the wrapped dataset class supports the `get_cat_ids(idx)` method, which returns a list. The list contains the categories of `data_info` given by 'idx'. The usage is as follows:

```
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'])
        return data_info
```

(continues on next page)

(continued from previous page)

```
    # The necessary method that needs to return the category of data sample
    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)
```

The above example resamples the `train` set of the dataset with `oversample_thr=1e-3`. Specifically, for categories whose frequency is less than `1e-3` in the dataset, samples corresponding to this category will be sampled repeatedly; otherwise, samples will not be sampled repeatedly. Please refer to the API documentation of `ClassBalancedDataset` for specific sampling policies.

17.4.4 Customize DatasetWrapper

Since the BaseDataset support lazy init, some rules need to be followed when customizing the DatasetWrapper. Here is an example to show how to customize the DatasetWrapper:

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


@DATASETS.register_module()
class ExampleDatasetWrapper:

    def __init__(self, dataset, lazy_init=False, ...):
        # Build the source datasetself.dataset
        if isinstance(dataset, dict):
            self.dataset = DATASETS.build(dataset)
        elif isinstance(dataset, BaseDataset):
            self.dataset = dataset
        else:
            raise TypeError(
                'elements in datasets sequence should be config or '
                f'`BaseDataset` instance, but got {type(dataset)}')
        # Record the meta information of source dataset
        self._metainfo = self.dataset.metainfo

        '''
        1. Implement some code here to record some of the hyperparameters used to wrap␣
↪the dataset.
```

(continues on next page)

(continued from previous page)

```
        '''

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

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

        # Initialize the source dataset completely
        self.dataset.full_init()

        '''
        2. Implement some code here to wrap the source dataset.
        '''

        self._fully_initialized = True

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

        '''
        3. Implement some code here to map the wrapped index `idx` to the index of the␣
↪source dataset 'ori_idx'.
        '''
        ori_idx = ...

        return ori_idx

    # Provide the same external interface as `self.dataset `.
    @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)

    # Provide the same external interface as `self.dataset `.
    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]

    # Provide the same external interface as `self.dataset `.
    @force_full_init
    def __len__(self):

        '''
        4. Implement some code here to calculate the length of the wrapped dataset.
```

(continues on next page)

(continued from previous page)

```
        '''
        len_wrapper = ...

        return len_wrapper

    # Provide the same external interface as `self.dataset `.
    @property
    def metainfo(self)
        return copy.deepcopy(self._metainfo)
```

CHAPTER

EIGHTEEN

DATA TRANSFORM

In the OpenMMLab repositories, dataset construction and data preparation are decoupled from each other. Usually, the dataset construction only parses the dataset and records the basic information of each sample, while the data preparation is performed by a series of data transforms, such as data loading, preprocessing, and formatting based on the basic information of the samples.

18.1 To use Data Transforms

In MMEngine, we use various callable data transforms classes to perform data manipulation. These data transformation classes can accept several configuration parameters for instantiation and then process the input data dictionary by calling. Also, all data transforms accept a dictionary as input and output the processed data as a dictionary. A simple example is as belows:

Note: In MMEngine, we don't have the implementations of data transforms. you can find the base data transform class and many other data transforms in MMCV. So you need to install MMCV before learning this tutorial, see the MMCV installation guild.

```
>>> 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)
```

18.2 To use in Config Files

In config files, we can compose multiple data transforms as a list, called a data pipeline. And the data pipeline is an argument of the dataset.

Usually, a data pipeline consists of the following parts:

1. Data loading, use `LoadImageFromFile` to load image files.
2. Label loading, use `LoadAnnotations` to load the bboxes, semantic segmentation and keypoint annotations.
3. Data processing and augmentation, like `RandomResize`.

4. Data formatting, we use different data transforms for different tasks. And the data transform for specified task is implemented in the corresponding repository. For example, the data formatting transform for image classification task is `PackClsInputs` and it's in MMClassification.

Here, taking the classification task as an example, we show a typical data pipeline in the figure below. For each sample, the basic information stored in the dataset is a dictionary as shown on the far left side of the figure, after which, every blue block represents a data transform, and in every data transform, we add some new fields (marked in green) or update some existing fields (marked in orange) in the data dictionary.

If want to use the above data pipeline in our config file, use the below settings:

```
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'),
        ]
    )
)
```

18.3 Common Data Transforms

According to the functionality, the data transform classes can be divided into data loading, data pre-processing & augmentation and data formatting.

18.3.1 Data Loading

To support loading large-scale dataset, usually we won't load all dense data during dataset construction, but only load the file path of these data. Therefore, we need to load these data in the data pipeline.

Data Transforms	Functionality
`LoadImageFromFile`	Load images according to the path.
`LoadAnnotations`	Load and format annotations information, including bbox, segmentation map and others.

18.3.2 Data Pre-processing & Augmentation

Data transforms for pre-processing and augmentation usually manipulate the image and annotation data, like cropping, padding, resizing and others.

Data Transforms	Functionality
`Pad`	Pad the margin of images.
`CenterCrop`	Crop the image and keep the center part.
`Normalize`	Normalize the image pixels.
`Resize`	Resize images to the specified scale or ratio.
`RandomResize`	Resize images to a random scale in the specified range.
`RandomChoiceResize`	Resize images to a random scale from several specified scales.
`RandomGrayscale`	Randomly grayscale images.
`RandomFlip`	Randomly flip images.

18.3.3 Data Formatting

Data formatting transforms will convert the data to some specified type.

Data Transforms	Functionality
`ToTensor`	Convert the data of specified field to `torch.Tensor`
`ImageToTensor`	Convert images to `torch.Tensor` in PyTorch format.

18.4 Custom Data Transform Classes

To implement a new data transform class, the class needs to inherit `BaseTransform` and implement `transform` method. Here, we use a simple flip transforms (`MyFlip`) as example:

```
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
```

Then, we can instantiate a `MyFlip` object and use it to process our data dictionary.

```
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']
```

Or, use it in the data pipeline by modifying our config file:

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

Please note that to use the class in our config file, we need to confirm the `MyFlip` class will be imported during running.

CHAPTER

NINETEEN

INITIALIZATION

Usually, we'll customize our module based on nn.Module, which is implemented by Native PyTorch. Also, torch.nn.init could help us initialize the parameters of the model easily. To simplify the process of model construction and initialization, MMEngine designed the *BaseModule* to help us define and initialize the model from config easily.

19.1 Initialize the model from config

The core function of `BaseModule` is that it could help us to initialize the model from config. Subclasses inherited from `BaseModule` could define the `init_cfg` in the `__init__` function, and we can choose the method of initialization by configuring `init_cfg`.

Currently, we support the following initialization methods:

19.1.1 Initialize the model with pretrained model

Defining the `ToyNet` as below:

```
import torch
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.Linear(1, 1)


# Save the checkpoint.
toy_net = ToyNet()
torch.save(toy_net.state_dict(), './pretrained.pth')
pretrained = './pretrained.pth'

toy_net = ToyNet(init_cfg=dict(type='Pretrained', checkpoint=pretrained))
```

and then we can configure the `init_cfg` to make it load the pretrained model by calling `initi_weights()` after its construction.

```
# Initialize the model with the saved checkpoint.
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
```

If `init_cfg` is a `dict`, `type` means a kind of initializer registered in `WEIGHT_INITIALIZERS`. The `Pretrained` means `PretrainedInit`, which could help us to load the target checkpoint. All initializers have the same mapping relationship like `Pretrained` -> `PretrainedInit`, which strips the suffix `Init` of the class name. The `checkpoint` argument of `PretrainedInit` means the path of the checkpoint. It could be a local path or a URL.

19.1.2 Commonly used initialization methods

Similarly, we could use the `Kaiming` initialization just like `Pretrained` initializer. For example, we could make `init_cfg=dict(type='Kaiming', layer='Conv2d')` to initialize all `Conv2d` module with `Kaiming` initialization.

Sometimes we need to initialize the model with different initialization methods for different modules. For example, we could initialize the `Conv2d` module with `Kaiming` initialization and initialize the `Linear` module with `Xavier` initialization. We could make `init_cfg=dict(type='Kaiming', layer='Conv2d')`:

```
import torch.nn as nn

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)


# Apply `Kaiming` initialization to `Conv2d` module and `Xavier` initialization to `Linear`␣
↪module.
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 -
```

(continues on next page)

(continued from previous page)

```
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
```

`layer` could also be a list, each element of which means a type of applied module.

```
# Apply Kaiming initialization to `Conv2d` and `Linear` module.
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]):
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
```

19.1.3 More fine-grained initialization

Sometimes we need to initialize the same type of module with different types of initialization. For example, we've defined `conv1` and `conv2` submodules, and we want to initialize the `conv1` with `Kaiming` initialization and `conv2` with `Xavier` initialization. We could configure the init_cfg with `override`:

```
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)


# Apllly `Kaiming` initialization to `conv1` and `Xavier` initialization to `conv2`.
toy_net = ToyNet(
    init_cfg=[
```

(continues on next page)

(continued from previous page)

```
        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` could be understood as an nested `init_cfg`, which could also be a `list` or `dict`, and we should also set "`type`" for it. The difference is that we must set `name` in `override` to specify the applied scope for submodule. As the example above, we set `name='conv2'` to specify that the `Xavier` initialization is applied to all submodules of `toy_net.conv2`.

19.1.4 Customize the initialization method

Although the `init_cfg` could control the initialization method for different modules, we would have to register a new initialization method to `WEIGHT_INITIALIZERS` if we want to customize initialization process. It is not convenient right? Actually, we could also override the `init_weights` method to customize the initialization process.

Assuming we've defined the following modules:

- `ToyConv` inherit from `nn.Module`, implements `init_weights`which initialize `custom_weight`(`parameter` of `ToyConv`) with 1 and initialize `custom_bias` with 0
- `ToyNet` defines a `ToyConv` submodule.

`ToyNet.init_weights` will call `init_weights` of all submodules sequentially.

```
import torch
import torch.nn as nn

from mmengine.model import BaseModule


class ToyConv(nn.Module):

    def __init__(self):
        super().__init__()
```

(continues on next page)

(continued from previous page)

```
        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'))
    ])

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

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
```

19.1.5 Conclusion

1. Configure `init_cfg` to initialize model

- Commonly used for the initialization of `Conv2d`, `Linear` and other underlying module. All initialization methods should be managed by `WEIGHT_INITIALIZERS`
- Dynamic initialization controlled by `init_cfg`

2. Customize `init_weights`

- Compared to configuring the `init_cfg`, implementing the `init_weights` is simpler and does not require registration. However, it is not as flexible as `init_cfg`, and it is not possible to initialize the module dynamically.

Note:

- The priorify of init_weights is higher than `init_cfg`
- Runner will call `init_weights` in Runner.train()

19.1.6 Ininitailize module with function

As mentioned in prior *section*, we could customize our initialization in `init_weights`. To make it more convenient to initialize modules, MMEngine provides a series of **module initialization functions** to initialize the whole module based on `torch.nn.init`. For example, we want to initialize the weights of the convolutional layer with normal distribution and initialize the bias of the convolutional layer with a constant. The implementation of `torch.nn.init` is as follows:

```
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)
```

The above process is actually a standard process for initializing a convolutional module with normal distribution, so MMEngine simplifies this by implementing a series of common **module** initialization functions. Compared with `torch.nn.init`, the module initialization functions could accept the convolution module directly:

```
from mmengine.model import normal_init

normal_init(model, mean=0, std=0.01, bias=0)
```

Similarly, we could also use Kaiming initialization and Xavier initialization

```
from mmengine.model import kaiming_init, xavier_init

kaiming_init(model)
xavier_init(model)
```

Currently, MMEngine provide the following initialization function:

CHAPTER

TWENTY

VISUALIZATION

Visualization provides an intuitive explanation of the training and testing process of the deep learning model.

MMEngine provides `Visualizer` to visualize and store the state and intermediate results of the model training and testing process, with the following features:

- It supports basic drawing interface and feature map visualization
- It enables recording training states (such as loss and lr), performance evaluation metrics, and visualization results to a specified or multiple backends, including local device, TensorBoard, and WandB.
- It can be used in any location in the code base.

20.1 Basic Drawing APIs

`Visualizer` provides drawing APIs for common objects such as **detection bboxes, points, text, lines, circles, polygons, and binary masks**.

These APIs have the following features:

- Can be called multiple times to achieve overlay drawing requirements.
- All support multiple input types such as Tensor, Numpy array, etc.

Typical usages are as follows.

1. Draw detection bboxes, masks, text, etc.

```
import torch
import mmcv
from mmengine.visualization import Visualizer

image = mmcv.imread('docs/en/_static/image/cat_dog.png', channel_order='rgb')
visualizer = Visualizer(image=image)
# single bbox formatted as [xyxy]
visualizer.draw_bboxes(torch.tensor([72, 13, 179, 147]))
# draw multiple bboxes
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()
```

You can also customize things like color and width using the parameters in each API.

```
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. Overlay display

These APIs can be called multiple times to get an overlay result.

```
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()
```

20.2 Feature Map Visualization

Feature map visualization has many functions. Currently, we only support single feature map visualization.

```
@staticmethod
def draw_featmap(featmap: torch.Tensor, # input format must be CHW
                 overlaid_image: Optional[np.ndarray] = None, # if image data is input␣
↪at the same time, the feature map will be overlaid on the image
                 channel_reduction: Optional[str] = 'squeeze_mean', # strategy to reduce␣
↪multiple channels into a single channel
                 topk: int = 10, # topk feature maps to show
                 arrangement: Tuple[int, int] = (5, 2), # the layout when multiple␣
↪channels are expanded into multiple images
                 resize_shapeOptional[tuple] = None, # scale the feature map
                 alpha: float = 0.5) -> np.ndarray: # overlay ratio between input image␣
↪and generated feature map
```

The main features can be concluded as follows:

- As the input Tensor usually includes multiple channels, `channel_reduction` can reduce them into a single channel and overlay the result to the image.
  - `squeeze_mean` reduces the input channel C into a single channel using the mean function, so the output dimension becomes (1, H, W)
  - `select_max` select the channel with the maximum activation, where 'activation' refers to the sum across spatial dimensions of a channel.
  - `None` indicates that no reduction is needed, which allows the user to select the top k feature maps with the highest activation degree through the `topk` parameter.
- `topk` is only valid when the `channel_reduction` is `None`. It selects the top k channels according to the activation degree and then displays them overlaid with the image. The display layout can be specified using the `--arrangement` parameter.
  - If `topk` is not -1, `topk` channels with the largest activation will be selected for display.
  - If `topk` is -1, channel number C must be either 1 or 3 to indicate if the input is a picture. Otherwise, an error will be raised to prompt the user to reduce the channel with `channel_reduction`.

- Considering that the input feature map is usually very small, the function can upsample the feature map through `resize_shape` before the visualization.

For example, we would like to get the feature map from the layer4 output of a pre-trained ResNet18 model and visualize it.

1. Reduce the multi-channel feature map into a single channel using `select_max` and display it.

```
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)
    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)
```

Since the output feat feature map size is 7x7, the visualization effect is not good if we directly work on it. Users can scale the feature map by overlaying the input image or the `resize_shape` parameter. If the size of the incoming image is not the same as the size of the feature map, the feature map will be forced to be resampled to the same spatial size as the input image.

```
drawn_img = visualizer.draw_featmap(feat, image, channel_reduction='select_max')
visualizer.show(drawn_img)
```

2. Select the top five channels with the highest activation in the multi-channel feature map by setting `topk=5`, then format them into a 2x3 layout.

```
drawn_img = visualizer.draw_featmap(feat, image, channel_reduction=None, topk=5,␣
↪arrangement=(2, 3))
visualizer.show(drawn_img)
```

Users can set their own desired layout through `arrangement`.

```
drawn_img = visualizer.draw_featmap(feat, image, channel_reduction=None, topk=5,␣
↪arrangement=(4, 2))
visualizer.show(drawn_img)
```

20.3 Basic Storage APIs

Once the drawing is completed, users can choose to display the result directly or save it to different backends. The backends currently supported by MMEngine include local storage, `Tensorboard` and `WandB`. The data supported include drawn pictures, scalars, and configurations.

1. Save the result image

Suppose you want to save to your local device.

```
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]))

# temp_dir/vis_data/vis_image/demo_0.png will be generated
visualizer.add_image('demo', visualizer.get_image())
```

The zero in the result file name is used to distinguish different steps.

```
# temp_dir/vis_data/vis_image/demo_1.png will be generated
visualizer.add_image('demo', visualizer.get_image(), step=1)
# temp_dir/vis_data/vis_image/demo_3.png will be generated
visualizer.add_image('demo', visualizer.get_image(), step=3)
```

If you want to switch to other backends, you can change the configuration file like this:

```
# TensorboardVisBackend
visualizer = Visualizer(image=image, vis_backends=[dict(type='TensorboardVisBackend')],␣
↪save_dir='temp_dir')
# WandbVisBackend
visualizer = Visualizer(image=image, vis_backends=[dict(type='WandbVisBackend')], save_
↪dir='temp_dir')
```

2. Store feature maps

```
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))
# temp_dir/vis_data/vis_image/feat_0.png will be generated
visualizer.add_image('feat', drawn_img)
```

3. Save scalar data such as loss

```
# temp_dir/vis_data/scalars.json will be generated
# save loss
visualizer.add_scalar('loss', 0.2, step=0)
visualizer.add_scalar('loss', 0.1, step=1)
# save acc
visualizer.add_scalar('acc', 0.7, step=0)
visualizer.add_scalar('acc', 0.8, step=1)
```

Multiple scalar data can also be saved at once.

```
# New contents will be added to the temp_dir/vis_data/scalars.json
visualizer.add_scalars({'loss': 0.3, 'acc': 0.8}, step=3)
```

4. Save configurations

```
from mmengine import Config
cfg=Config.fromfile('tests/data/config/py_config/config.py')
# temp_dir/vis_data/config.py will be saved
visualizer.add_config(cfg)
```

20.4 Various Storage Backends

Any `Visualizer` can be configured with any number of storage backends. `Visualizer` will loop through all the configured backends and save the results to each one.

```
visualizer = Visualizer(image=image, vis_backends=[dict(type='TensorboardVisBackend'),
                                                    dict(type='LocalVisBackend')],
                        save_dir='temp_dir')
# temp_dir/vis_data/events.out.tfevents.xxx files will be generated
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())
```

Note: If there are multiple backends used at the same time, the `name` field must be specified. Otherwise, it is impossible to distinguish which backend it is.

```
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')],
                        save_dir='temp_dir')
```

20.5 Visualize at Anywhere

During the development, users may need to add visualization functions somewhere in their codes and save the results to different backends, which is very common for analysis and debugging. `Visualizer` in MMEngine can obtain the data from the same visualizers and then visualize them.

Users only need to instantiate the visualizer through `get_instance` during initialization. The visualizer obtained this way is unique and globally accessible. Then it can be accessed anywhere in the code through `Visualizer.get_current_instance()`.

```
# call during the initialization stage
visualizer1 = Visualizer.get_instance(name='vis', vis_backends=[dict(type=
↪'LocalVisBackend')])

# call anywhere
visualizer2 = Visualizer.get_current_instance()
visualizer2.add_scalar('map', 0.7, step=0)

assert id(visualizer1) == id(visualizer2)
```

It can also be initialized globally through the config field.

```
from mmengine.registry import VISUALIZERS

visualizer_cfg=dict(
                type='Visualizer',
                name='vis_new',
                vis_backends=[dict(type='LocalVisBackend')])
VISUALIZERS.build(visualizer_cfg)
```

20.6 Customize Storage Backends and Visualizers

1. Call a specific storage backend

The storage backend only provides basic functions such as saving configurations and scalars. However, users may want to utilize other powerful backend features like WandB and Tensorboard. Therefore, the storage backend provides the `experiment` attribute to facilitate users to obtain backend objects and meet various customized functions.

For example, WandB provides an API to display tables. Users can obtain the WandB objects through the `experiment` attribute and then call a specific API to save the data as a table to show.

```
visualizer = Visualizer(image=image, vis_backends=[dict(type='WandbVisBackend')],
                        save_dir='temp_dir')

# get WandB object
wandb = visualizer.get_backend('WandbVisBackend').experiment
# add data to the table
table = wandb.Table(columns=["step", "mAP"])
table.add_data(1, 0.2)
table.add_data(2, 0.5)
table.add_data(3, 0.9)
# save
wandb.log({"table": table})
```

2. Customize storage backends

Users only need to inherit `BaseVisBackend` and implement various `add_xx` methods to customize the storage backend easily.

```
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. Customize visualizers

Similarly, users can easily customize the visualizer by inheriting `Visualizer` and implementing the functions they want to override.

In most cases, users need to override `add_datasample`. The data usually includes detection bboxes and instance masks from annotations or model predictions. This interface is for drawing `datasample` data for various downstream libraries. Taking MMDetection as an example, the `datasample` data usually includes labeled bboxs, labeled masks, predicted bboxs, or predicted masks. MMDetection will inherit `Visualizer` and implement the `add_datasample` interface, drawing the data related to the detection task.

```
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,
                       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')

# global initialize
VISUALIZERS.build(visualizer_cfg)

# call anywhere in your code
det_local_visualizer = Visualizer.get_current_instance()
det_local_visualizer.add_datasample('det', image, data_sample)
```

CHAPTER

TWENTYONE

ABSTRACT DATA ELEMENT

Coming soon. Please refer to chinese documentation.

CHAPTER

TWENTYTWO

DISTRIBUTION COMMUNICATION

In distributed training, different processes sometimes need to apply different logics depending on their ranks, local_ranks, etc. They also need to communicate with each other and do synchronizations on data. These demands rely on distributed communication. PyTorch provides a set of basic distributed communication primitives. Based on these primitives, MMEngine provides some higher level APIs to meet more diverse demands. Using these APIs provided by MMEngine, modules can:

- ignore the differences between distributed/non-distributed environment
- deliver data in various types apart from Tensor
- ignore the frameworks or backends used for communication

These APIs are roughly categorized into 3 types:

- Initialization: `init_dist` for setting up distributed environment for the runner
- Query & control: functions including `get_world_size` for querying `world_size`, `rank` and other distributed information
- Collective communication: collective communication functions such as `all_reduce`

We will detail on these APIs in the following chapters.

22.1 Initialization

- *init_dist*: Launch function of distributed training. Currently it supports 3 launchers including pytorch, slurm and MPI. It also setup the given communication backends, defaults to NCCL.

22.2 Query and control

The query and control functions are all argument free. They can be used in both distributed and non-distributed environment. Their functionalities are listed below:

- *get_world_size*: Returns the number of processes in current process group. Returns 1 when non-distributed
- *get_rank*: Returns the global rank of current process in current process group. Returns 0 when non-distributed
- *get_backend*: Returns the communication backends used by current process group. Returns `None` when non-distributed
- *get_local_rank*: Returns the local rank of current process in current process group. Returns 0 when non-distributed

- *get_local_size*: Returns the number of processes which are both in current process group and on the same machine as the current process. Returns 1 when non-distributed
- *get_dist_info*: Returns the world_size and rank of the current process group. Returns world_size = 1, rank = 0 when non-distributed
- *is_main_process*: Returns `True` if current process is rank 0 in current process group, otherwise `False` . Always returns `True` when non-distributed
- *master_only*: A function decorator. Functions decorated by `master_only` will only execute on rank 0 process.
- *barrier*: A synchronization primitive. Every process will hold until all processes in the current process group reach the same barrier location

22.3 Collective communication

Collective communication functions are used for data transfer between processes in the same process group. We provide the following APIs based on PyTorch native functions including all_reduce, all_gather, gather, broadcast. These APIs are compatible with non-distributed environment and support more data types apart from Tensor.

- *all_reduce*: AllReduce operation on Tensors in the current process group
- *all_gather*: AllGather operation on Tensors in the current process group
- *gather*: Gather Tensors in the current process group to a destinated rank
- *broadcast*: Broadcast a Tensor to all processes in the current process group
- *sync_random_seed*: Synchronize random seed between processes in the current process group
- *broadcast_object_list*: Broadcast a list of Python objects. It requires the object can be serialized by Pickle.
- *all_reduce_dict*: AllReduce operation on dict. It is based on broadcast and all_reduce.
- *all_gather_object*: AllGather operations on any Python object than can be serialized by Pickle. It is based on all_gather
- *gather_object*: Gather Python objects that can be serialized by Pickle
- *collect_results*: Unified API for collecting a list of data in current process group. It support both CPU and GPU communication

CHAPTER

TWENTYTHREE

LOGGING

Runner will produce a lot of logs during the running process, such as loss, iteration time, learning rate, etc. MMEngine implements a flexible logging system that allows us to choose different types of log statistical methods when configuring the runner. It could help us set/get the recorded log at any location in the code.

23.1 Flexible Logging System

Logging system is configured by passing a LogProcessor to the runner. If no log processor is passed, the runner will use the default log processor, which is equivalent to:

```
log_processor = dict(window_size=10, by_epoch=True, custom_cfg=None, num_digits=4)
```

The format of the output log is as follows:

```
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,
```

(continues on next page)

(continued from previous page)

```
    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
```

LogProcessor will output the log in the following format:

- The prefix of the log:
  - epoch mode(by_epoch=True): `Epoch(train) [{current_epoch}/{current_iteration}]/{dataloader_length}`
  - iteration mode(by_epoch=False): `Iter(train) [{current_iteration}/{max_iteration}]`
- Learning rate (`lr`): The learning rate of the last iteration.
- Time:
  - `time`: The averaged time for infernce of the last `window_size` iterations.
  - `data_time`: The averaged time for loading data of the last `window_size` iterations.
  - `eta`: The estimated time of arrival to finish the training.
- Loss: The averaged loss output by model of the last `window_size` iterations.

Note: `window_size=10` by default.

The significant digits(`num_digits`) of the log is 4 by default.

Output the value of all custom logsthe at last iteration by default.

Based on the rules above, the code snippet will count the average value of the `loss1` and `loss2` every 10 iterations.

If we want to count the global average value of `loss1`, we can set `custom_cfg` like this:

```
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=[
            dict(data_src='loss1',  # original loss nameloss1
                 method_name='mean',  # statistical methodmean
                 window_size='global')])  # window_sizeglobal
)
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
```

`data_src` means the original loss name, `method_name` means the statistic method, `window_size` means the window size of the statistic method. Since we want to count the global average value of `loss1`, we set `window_size` to `global`.

Currently, MMEngine supports the following statistical methods:

`window_size` mentioned above could be:

- int number: The window size of the statistic method.
- `global`: Equivalent to `window_size=cur_iteration`.
- `epoch`: Equivalent to `window_size=len(dataloader)`.

If we want to statistic the average value of `loss1` of the last 10 iterations, and also want to statistic the global average value of `loss1`. We need to set `log_name` additionally:

```
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 means the second name of loss1
            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
```

Similarly, we can also statistic the global/local maximum value of `loss` at the same time.

```
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=[
        # statistic loss1 with the local maximum value
        dict(data_src='loss1',
             log_name='loss1_local_max',
             window_size=10,
             method_name='max'),
```

(continues on next page)

(continued from previous page)

```
        # statistic loss1 with the global maximum value
        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
```

More examples can be found in log_processor.

23.2 Customize log

The logging system could not only log the `loss`, `lr`, .etc but also collect and output the custom log. For example, if we want to statistic the intermediate `loss`:

```
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()
        # update the intermediate `loss_tmp` in the message hub
        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(
```

(continues on next page)

(continued from previous page)

```
        custom_cfg=[
        # statistic the loss_tmp with the averaged value
            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
```

The custom log will be recorded by updating the *messagehub*:

1. Calling `MessageHub.get_current_instance()` to get the message of runner
2. Calling `MessageHub.update_scalar` to update the custom log. The first argument means the log name with the mode prefix(`train/val/test`). The output log will only retain the log name without the mode prefix.
3. Configure statistic method of `loss_tmp` in `log_processor`. If it is not configured, only the latest value of `loss_tmp` will be logged.

23.3 Export the debug log

Set `log_level=DEBUG` for runner, and the debug log will be exported to the `work_dir`:

```
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
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"
...
```

Besides, logs of different ranks will be saved in `debug` mode if you are training your model with the shared storage. The hierarchy of the log is as follows:

```
./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
```

The log of Multiple machine with independent storage:

```
# device: 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

# device: 7
work_dir/
└── exp_name_logs
    ├── exp_name_rank56.log
    ├── exp_name_rank57.log
    ├── exp_name_rank58.log
    ...
    └── exp_name_rank63.log
```

CHAPTER

TWENTYFOUR

FILE IO

MMEngine implements a unified set of file reading and writing interfaces in `fileio` module. With the `fileio` module, we can use the same function to handle different file formats, such as `json`, `yaml` and `pickle`. Other file formats can also be easily extended.

The `fileio` module also supports reading and writing files from a variety of file storage backends, including disk, Petrel (for internal use), Memcached, LMDB, and HTTP.

24.1 Load and dump data

MMEngine provides a universal API for loading and dumping data, currently supported formats are `json`, `yaml`, and `pickle`.

24.1.1 Load from disk or dump to disk

```python
from mmengine import load, dump

# load data from a file
data = load('test.json')
data = load('test.yaml')
data = load('test.pkl')
# load data from a file-like object
with open('test.json', 'r') as f:
    data = load(f, file_format='json')

# dump data to a string
json_str = dump(data, file_format='json')

# dump data to a file with a filename (infer format from file extension)
dump(data, 'out.pkl')

# dump data to a file with a file-like object
with open('test.yaml', 'w') as f:
    data = dump(data, f, file_format='yaml')
```

24.1.2 Load from other backends or dump to other backends

```
from mmengine import load, dump

# load data from a file
data = load('s3://bucket-name/test.json')
data = load('s3://bucket-name/test.yaml')
data = load('s3://bucket-name/test.pkl')

# dump data to a file with a filename (infer format from file extension)
dump(data, 's3://bucket-name/out.pkl')
```

It is also very convenient to extend the API to support more file formats. All you need to do is to write a file handler inherited from `BaseFileHandler` and register it with one or several file formats.

```
from mmengine import register_handler, BaseFileHandler

# To register multiple file formats, a list can be used as the argument.
# @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)
```

Here is an example of `PickleHandler`.

```
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)
```

(continues on next page)

(continued from previous page)

```
    def dump_to_path(self, obj, filepath, **kwargs):
        super(PickleHandler, self).dump_to_path(
            obj, filepath, mode='wb', **kwargs)
```

24.2 Load a text file as a list or dict

For example `a.txt` is a text file with 5 lines.

```
a
b
c
d
e
```

24.2.1 Load from disk

Use `list_from_file` to load the list from `a.txt`.

```
from mmengine import list_from_file

print(list_from_file('a.txt'))
# ['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']
```

For example `b.txt` is a text file with 3 lines.

```
1 cat
2 dog cow
3 panda
```

Then use `dict_from_file` to load the dict from `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'}
```

24.2.2 Load from other backends

Use `list_from_file` to load the list from `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']
```

Use `dict_from_file` to load the dict from `s3://bucket-name/b.txt`.

```
from mmengine import dict_from_file

print(dict_from_file('s3://bucket-name/b.txt'))
# {'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'}
```

24.3 Load and dump checkpoints

We can read the checkpoints from disk or internet in the following way.

```
import torch

filepath1 = '/path/of/your/checkpoint1.pth'
filepath2 = 'http://path/of/your/checkpoint3.pth'

# read filepath1 from disk
checkpoint = torch.load(filepath1)
# save checkpoints to disk
torch.save(checkpoint, filepath1)

# read filepath2 from internet
checkpoint = torch.utils.model_zoo.load_url(filepath2)
```

In `MMEngine`, reading and writing checkpoints in different storage forms can be uniformly implemented with `load_checkpoint` and `save_checkpoint`.

```
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'

# read checkpoints from disk
checkpoint = load_checkpoint(filepath1)
```

(continues on next page)

(continued from previous page)

```
# save checkpoints from disk
save_checkpoint(checkpoint, filepath1)

# read checkpoints from s3
checkpoint = load_checkpoint(filepath2)
# save checkpoints from s3
save_checkpoint(checkpoint, filepath2)

# read checkpoints from internet
checkpoint = load_checkpoint(filepath3)
```

CHAPTER

TWENTYFIVE

GLOBAL MANAGER (MANAGERMIXIN)

During the training process, it is inevitable that we need to access some variables globally. Here are some examples:

- Accessing the *logger* in model to print some initialization information
- Accessing the *Visualizer* anywhere to visualize the predictions and feature maps.
- Accessing the scope in *Registry* to get the current scope.

In order to unify the mechanism to get the global variable built from different classes, MMEngine designs the *ManagerMixin*.

25.1 Interface introduction

- get_instance(name=”, **kwargs): Create or get the instance by name.
- get_current_instance(): Get the currently built instance.
- instance_name: Get the name of the instance.

25.2 How to use

1. Define a class inherited from `ManagerMixin`

```
from mmengine.utils import ManagerMixin


class GlobalClass(ManagerMixin):
    def __init__(self, name, value):
        super().__init__(name)
        self.value = value
```

Note: Subclasses of `ManagerMixin` must accept `name` argument in `__init__`. The `name` argument is used to identify the instance, and you can get the instance by `get_instance(name)`.

2. Instantiate the instance anywhere. let’s take the hook as an example:

```
from mmengine import Hook

class CustomHook(Hook):
```

(continues on next page)

(continued from previous page)

```
    def before_run(self, runner):
        GlobalClass.get_instance('mmengine', value=50)
        GlobalClass.get_instance(runner.experiment_name, value=100)
```

`GlobalClass.get_instance({name})` will first check whether the instance with the name `{name}` has been built. If not, it will build a new instance with the name `{name}`, otherwise it will return the existing instance. As the above example shows, when we call `GlobalClass.get_instance('mmengine')` at the first time, it will build a new instance with the name `mmengine`. Then we call `GlobalClass.get_instance(runner.experiment_name)`, it will also build a new instance with a different name.

Here we build two instances for the convenience of the subsequent introduction of `get_current_instance`.

3. Accessing the instance anywhere

```
import torch.nn as nn


class CustomModule(nn.Module):
    def forward(self, x):
        value = GlobalClass.get_current_instance().value
        # Since the name of the latest built instance is
        # `runner.experiment_name`, value will be 100.

        value = GlobalClass.get_instance('mmengine').value
        # The value of instance with the name mmengine is 50.

        value = GlobalClass.get_instance('mmengine', 1000).value
        # `mmengine` instance has been built, an error will be raised
        # if `get_instance` accepts other parameters.
```

We can get the instance with the specified name by `get_instance(name)`, or get the currently built instance by `get_current_instance` anywhere.

Warning: If the instance with the specified name has already been built, `get_instance` will raise an error if it accepts its construct parameters.

CHAPTER

TWENTYSIX

USE MODULES FROM OTHER LIBRARIES

Based on MMEngine's *Registry* and *Config*, users can build modules across libraries. For example, use MMClassification's backbones in MMDetection, or MMDetection's data transforms in MMRotate, or using MMDetection's detectors in MMTracking.

Modules registered in the same registry tree can be called across libraries by adding the **package name prefix** before the module's type in the config. Here are some common examples:

26.1 Use backbone across libraries

Taking the example of using MMClassification's ConvNeXt in MMDetection:

Firstly, adding the `custom_imports` field to the config to register the backbones of MMClassification to the registry.

Secondly, adding the package name of MMClassification `mmcls` to the `type` of the backbone as a prefix: `mmcls.ConvNeXt`

```
# Use custom_imports to register mmcls models to the registry
custom_imports = dict(imports=['mmcls.models'], allow_failed_imports=False)

model = dict(
  type='MaskRCNN',
  data_preprocessor=dict(...),
  backbone=dict(
      type='mmcls.ConvNeXt', # Add mmcls prefix to enable cross-library mechanism
      arch='tiny',
      out_indices=[0, 1, 2, 3],
      drop_path_rate=0.4,
      layer_scale_init_value=1.0,
      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(...))
```

26.2 Use data transform across libraries

As with the example of backbone above, cross-library calls can be simply achieved by adding custom_imports and prefix in the config:

```
# Use custom_imports to register mmdet transforms to the registry
custom_imports = dict(imports=['mmdet.datasets.transforms'], allow_failed_imports=False)

# Add mmdet prefix to enable cross-library mechanism
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')
]
```

26.3 Use detector across libraries

Using an algorithm from another library is a little bit complex.

An algorithm contains multiple submodules. Each submodule needs to add a prefix to its `type`. Take using MMDetection's YOLOX in MMTracking as an example:

```
# Use custom_imports to register mmdet models to the registry
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',
        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)))
```

To prevent adding prefix to all of the submodules manually, the `_scope_` keyword is introduced. When the `_scope_` keyword is added to the config of a module, all submodules' scope will be changed by the `_scope_` keyword. Here is an example config:

```
# Use custom_imports to register mmdet models to the registry
custom_imports = dict(imports=['mmdet.models'], allow_failed_imports=False)

model = dict(
    _scope_='mmdet', # use the _scope_ keyword to avoid adding prefix to all submodules
    type='YOLOX',
    backbone=dict(type='CSPDarknet', deepen_factor=1.33, widen_factor=1.25),
```

(continues on next page)

(continued from previous page)

```
    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)))
```

These two examples are equivalent to each other.

If you want to know more about the registry and config, please refer to *Config Tutorial* and *Registry Tutorial*

CHAPTER

TWENTYSEVEN

TEST TIME AUGMENTATION

Test time augmentation (TTA) is a data augmentation strategy used during the testing phase. It involves applying various augmentations, such as flipping and scaling, to the same image and then merging the predictions of each augmented image to produce a more accurate prediction. To make it easier for users to use TTA, MMEngine provides *BaseTTAModel* class, which allows users to implement different TTA strategies by simply extending the `BaseTTAModel` class according to their needs.

The core implementation of TTA is usually divided into two parts

1. Data augmentation: This part is implemented in MMCV, see the api docs TestTimeAug for more information.
2. Merge the predictions: The subclasses of `BaseTTAModel` will merge the predictions of enhanced data in the `test_step` method to improve the accuracy of predictions.

27.1 Get started

A simple example of TTA is given in examples/test_time_augmentation.py

27.1.1 Prepare test time augmentation pipeline

`BaseTTAModel` needs to be used with `TestTimeAug` implemented in MMCV:

```
tta_pipeline = [
    dict(type='LoadImageFromFile'),
    dict(
        type='TestTimeAug',
        transforms=[
            [dict(type='Resize', img_scale=(1333, 800), keep_ratio=True)],
            [dict(type='RandomFlip', flip_ratio=0.),
             dict(type='RandomFlip', flip_ratio=1.)],
            [dict(type='PackXXXInputs', keys=['img'])],
        ])
]
```

The above data augmentation pipeline will first perform a scaling enhancement on the image, followed by 2 flipping enhancements (flipping and not flipping). Finally, the image is packaged into the final result using `PackXXXInputs`.

27.1.2 Define the merge strategy

Commonly, users only need to inherit `BaseTTAModel` and override the `BaseTTAModel.merge_preds` to merge the predictions of enhanced data. `merge_preds` accepts a list of enhanced batch data, and each element of the list means the enhanced single data of the batch.

The BaseTTAModel class requires inferencing on both flipped and unflipped images and then merges the results. The merge_preds method accepts a list where each element represents the results of applying data augmentation to a single element of the batch. For example, if batch_size is 3, and we flip each image in the batch as an augmentation, merge_preds would accept a parameter like the following:

```
# `data_{i}_{j}` represents the result of applying the jth data augmentation to
#  the ith image in the batch. So, if batch_size is 3, i can take on values of
# 0, 1, and 2. If there are 2 augmentation methods
# (such as flipping the image), then j can take on values of 0 and 1.
# For example, data_2_1 would represent the result of applying the second
# augmentation method (flipping) to the third image in the batch.

demo_results = [
    [data_0_0, data_0_1],
    [data_1_0, data_1_1],
    [data_2_0, data_2_1],
]
```

The `merge_preds` method will merge the predictions `demo_results` into single batch results. For example, if we want to merge multiple classification results:

```
class AverageClsScoreTTA(BaseTTAModel):
    def merge_preds(
        self,
        data_samples_list: List[List[ClsDataSample]],
    ) -> List[ClsDataSample]:

        merged_data_samples = []
        for data_samples in data_samples_list:
            merged_data_sample: ClsDataSample = data_samples[0].new()
            merged_score = sum(data_sample.pred_label.score
                               for data_sample in data_samples) / len(data_samples)
            merged_data_sample.set_pred_score(merged_score)
            merged_data_samples.append(merged_data_sample)
        return merged_data_samples
```

The configuration file for the above example is as follows:

```
tta_model = dict(type='AverageClsScoreTTA')
```

27.1.3 Changes to test script

```
cfg.model = ConfigDict(**cfg.tta_model, module=cfg.model)
cfg.test_dataloader.dataset.pipeline = cfg.tta_pipeline
```

27.2 Advanced usage

In general, users who inherit the `BaseTTAModel` class only need to implement the merge_preds method to perform result fusion. However, for more complex cases, such as fusing the results of a multi-stage detector, it may be necessary to override the test_step method. This requires an understanding of the data flow in the BaseTTAModel class and its relationship with other components.

27.2.1 The relationship between BaseTTAModel and other components

The BaseTTAModel class acts as an intermediary between the DDPWrapper and Model classes. When the Runner.test() method is executed, it will first call DDPWrapper.test_step(), followed by TTAModel.test_step(), and finally model.test_step().

The following diagram illustrates this sequence of method calls:

27.2.2 data flow

After data augmentation with TestTimeAug, the resulting data will have the following format:

```
image1  = dict(
    inputs=[data_1_1, data_1_2],
    data_sample=[data_sample1_1, data_sample1_2])


image2  = dict(
    inputs=[data_2_1, data_2_2],
    data_sample=[data_sample2_1, data_sample2_2])


image3  = dict(
    inputs=[data_3_1, data_3_2],
    data_sample=[data_sample3_1, data_sample3_2])
```

where `data_{i}_{j}` means the enhanced dataand `data_sample_{i}_{j}` means the ground truth of enhanced data. Then the data will be processed by `Dataloader`, which contributes to the following format:

```
data_batch = dict(
    inputs = [
               (data_1_1, data_2_1, data_3_1),
               (data_1_2, data_2_2, data_3_2),
              ]
    data_samples=[
         (data_samples1_1, data_samples2_1, data_samples3_1),
         (data_samples1_2, data_samples2_2, data_samples3_2)
```

(continues on next page)

(continued from previous page)

```
    ]
)
```

To facilitate model inferencing, the `BaseTTAModel` will convert the data into the following format:

```
data_batch_aug1 = dict(
    inputs = (data_1_1, data_2_1, data_3_1),
    data_samples=(data_samples1_1, data_samples2_1, data_samples3_1)
)

data_batch_aug2 = dict(
    inputs = (data_1_2, data_2_2, data_3_2),
    data_samples=(data_samples1_2, data_samples2_2, data_samples3_2)
)
```

At this point, each `data_batch_aug` can be passed directly to the model for inferencing. After the model has performed inferencing, the `BaseTTAModel` will reorganize the predictions as follows for the convenience of merging:

```
preds = [
    [data_samples1_1, data_samples_1_2],
    [data_samples2_1, data_samples_2_2],
    [data_samples3_1, data_samples_3_2],
]
```

Now that we understand the data flow in TTA, we can override the BaseTTAModel.test_step() method to implement more complex fusion strategies based on specific requirements.

CHAPTER

TWENTYEIGHT

HOOK

Hook programming is a programming pattern in which a mount point is set in one or more locations of a program. When the program runs to a mount point, all methods registered to it at runtime are automatically called. Hook programming can increase the flexibility and extensibility of the program since users can register custom methods to the mount point to be called without modifying the code in the program.

28.1 Examples

Here is an example of how it works.

```
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()
```

Output of the above example.

```
hello
do something here
goodbye
```

As we can see, the `main` function calls `print` defined in hooks in two locations without making any changes.

Hook is also used everywhere in PyTorch, for example in the neural network module (nn.Module) to get the forward input and output of the module as well as the reverse input and output. For example, the `register_forward_hook` method registers a forward hook with the module, and the hook can get the forward input and output of the module.

The following is an example of the `register_forward_hook` usage.

```
import torch
import torch.nn as nn

def forward_hook_fn(
    module,  # object to be registered hooks
```

(continues on next page)

(continued from previous page)

```
    input,   # forward input of module
    output,  # forward output of module
):
    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()
# Register forward_hook_fn to each submodule of model
for module in model.children():
    module.register_forward_hook(forward_hook_fn)

x = torch.Tensor([[0.0, 1.0, 2.0]])
y = model(x)
```

Output of the above example.

```
"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>)
```

We can see that the `forward_hook_fn` hook registered to the `nn.Linear` module is called, and in that hook the weights, biases, module inputs, and outputs of the Linear module are printed. For more information on the use of PyTorch hooks you can read nn.Module.

28.2 Design on MMEngine

Before introducing the design of the `Hook` in MMEngine, let's briefly introduce the basic steps of model training using PyTorch (copied from 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):
```

(continues on next page)

(continued from previous page)

```
    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()
            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 = ...
```

The above pseudo-code is the basic step to train a model. If we want to add custom operations to the above code, we need to modify and extend the `main` function continuously. To increase the flexibility and extensibility of the `main` function, we can insert mount points into the `main` function and implement the logic of calling hooks at the corresponding mount points. In this case, we only need to insert hooks into these locations to implement custom logic, such as loading model weights, updating model parameters, etc.

```
def main():
    ...
    call_hooks('before_run', hooks)
    call_hooks('after_load_checkpoint', hooks)
    call_hooks('before_train', hooks)
    for i in range(max_epochs):
        call_hooks('before_train_epoch', hooks)
        for inputs, labels in train_dataloader:
            call_hooks('before_train_iter', hooks)
            outputs = net(inputs)
```

(continues on next page)

(continued from previous page)

```
            loss = criterion(outputs, labels)
            call_hooks('after_train_iter', hooks)
            loss.backward()
            optimizer.step()
        call_hooks('after_train_epoch', hooks)

        call_hooks('before_val_epoch', hooks)
        with torch.no_grad():
            for inputs, labels in val_dataloader:
                call_hooks('before_val_iter', hooks)
                outputs = net(inputs)
                loss = criterion(outputs, labels)
                call_hooks('after_val_iter', hooks)
        call_hooks('after_val_epoch', hooks)

        call_hooks('before_save_checkpoint', hooks)
    call_hooks('after_train', hooks)

    call_hooks('before_test_epoch', hooks)
    with torch.no_grad():
        for inputs, labels in test_dataloader:
            call_hooks('before_test_iter', hooks)
            outputs = net(inputs)
            accuracy = ...
            call_hooks('after_test_iter', hooks)
    call_hooks('after_test_epoch', hooks)

    call_hooks('after_run', hooks)
```

In MMEngine, we encapsulates the training process into an executor (`Runner`). The `Runner` calls hooks at specific mount points to complete the customization logic. For more information about `Runner`, please read the *Runner documentation*.

To facilitate management, MMEngine defines mount points as methods and integrates them into *Base Hook*. We just need to inherit the base hook and implement custom logic at specific location according to our needs, then register the hooks to the `Runner`. Those hooks will be called automatically.

There are 22 mount points in the *Base Hook*.

- 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

Further readings: *Hook tutorial* and *Hook API documentations*

CHAPTER

TWENTYNINE

RUNNER

Deep learning algorithms usually share similar pipelines for training, validation and testing. Therefore, MMengine designed `Runner` to simplify the construction of these pipelines. In most cases, users can use our default `Runner` directly. If you find it not feasible to implement your ideas, you can also modify it or customize your own runner.

Before introducing the design of `Runner`, let's walk through some examples to better understand why we should use runner. Below is a few lines of pseudo codes for training models in PyTorch:

```
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()
```

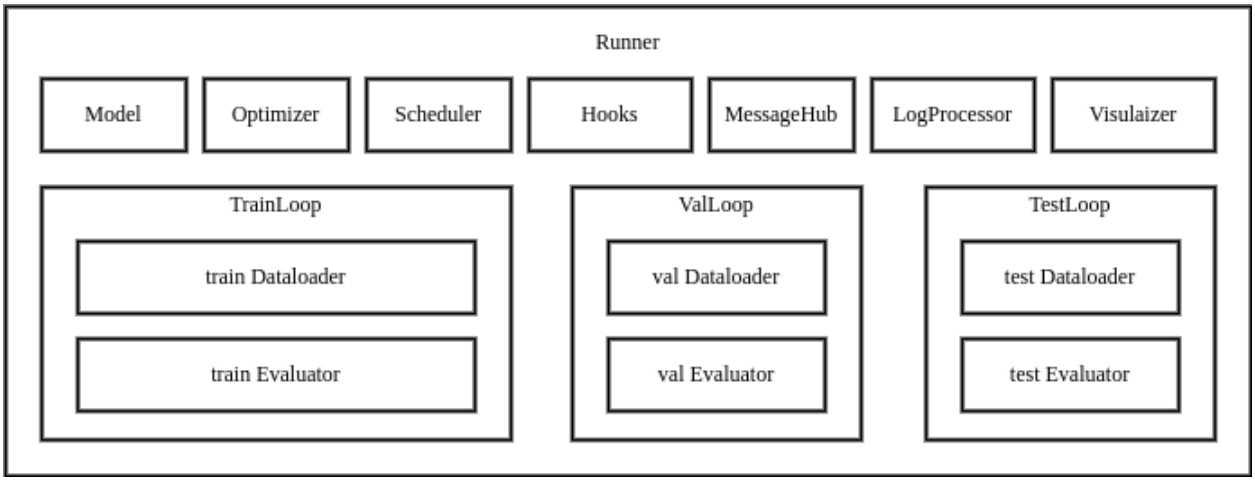
Pseudo codes for model validation in PyTorch:

```
model = ResNet()
model.load_state_dict(torch.load(CKPT_PATH))
model.eval()

test_dataset = ImageNetDataset(...)
test_dataloader = DataLoader(test_dataset, ...)

for data_batch in test_dataloader:
    outputs = model(data_batch)
    acc = calculate_acc(outputs, data_batch)
```

Pseudo codes for model inference in PyTorch:

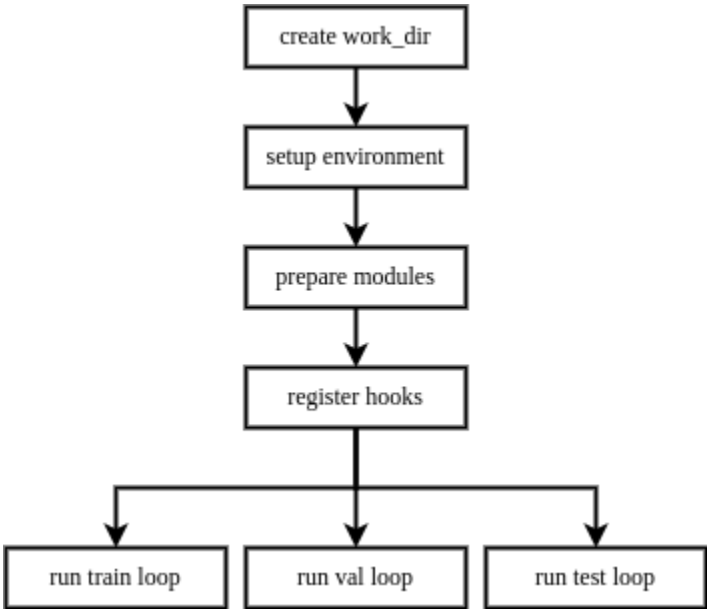
```
model = ResNet()
model.load_state_dict(torch.load(CKPT_PATH))
model.eval()

for img in imgs:
    prediction = model(img)
```

The observation from the above 3 pieces of codes is that they are similar. They can all be divided into some distinct steps, such as model construction, data loading and loop iterations. Although the above examples are based on image classification tasks, the same holds for many other tasks as well, including object detection, image segmentation, etc. Based on the observation above, we propose runner, which structures the training, validation and testing pipeline. With runner, the only thing you need to do is to prepare necessary components (models, data, etc.) of your pipeline, and leave the schedule and execution to `Runner`. You are free of constructing similar pipelines one and another time. You are free of annoying details like the differences between distributed and non-distributed training. You can focus on your own awesome ideas. These are all achieved by runner and various practical modules in MMEngine.



The `Runner` in MMEngine contains various modules required for training, testing and validation, as well as loop controllers(`Loop`) and *Hook*, as shown in the figure above. It provides 3 APIs for users: `train`, `val` and `test`, each correspond to a specific `Loop`. You can use `Runner` either by providing a config file, or by providing manually constructed modules. Once activated, the `Runner` will automatically setup the runtime environment, build/compose your modules, execute the loop iterations in `Loop` and call registered hooks during iterations.

The execution order of `Runner` is as follows:

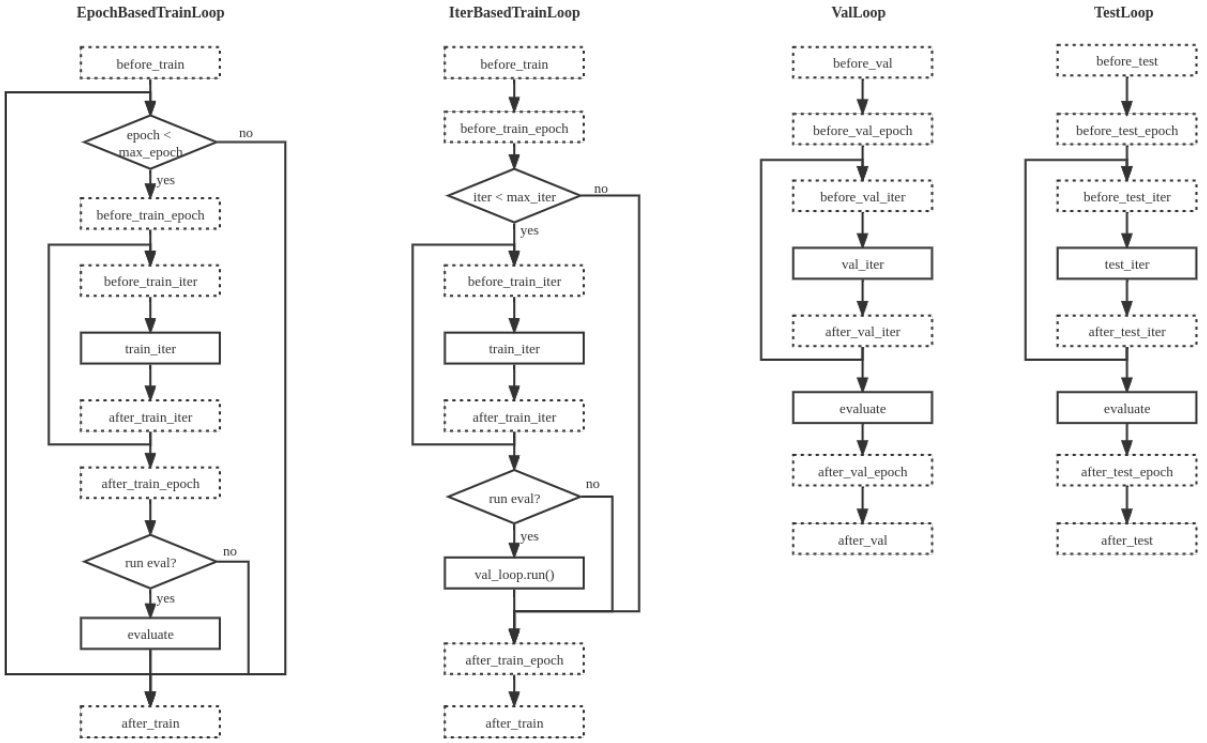


A feature of `Runner` is that it will always lazily initialize modules managed by itself. To be specific, `Runner` won't build every module on initialization, and it won't build a module until it is needed in current `Loop`. Therefore, if you are running only one of the `train`, `val`, or `test` pipelines, you only need to provide the relevant configs/modules.

29.1 Loop

In MMEngine, we abstract the execution process of the task into `Loop`, based on the observation that most deep learning tasks can be summarized as a model iterating over datasets. We provide 4 built-in loops in MMEngine:

- EpochBasedTrainLoop
- IterBasedTrainLoop
- ValLoop
- TestLoop



The built-in runner and loops are capable of most deep learning tasks, but surely not all. Some tasks need extra modifications and refactorizations. Therefore, we make it possible for users to customize their own pipelines for model training, validation and testing.

You can write your own pipeline by subclassing *BaseLoop*, which needs 2 arguments for initialization: 1) `runner` the Runner instance, and 2) `dataloader` the dataloader used in this loop. You are free to add more arguments to your own loop subclass. After defining your own loop subclass, you should register it to LOOPS(mmengine.registry.LOOPS), and specify it in config files by `type` field in `train_cfg`, `val_cfg` and `test_cfg`. In fact, you can write any execution order, any hook position in your own loop. However, built-in hooks may not work if you change hook positions, which may lead to inconsistent behavior during training. Therefore, we strongly recommend you to implement you subclass with similar execution order illustrated in the figure above, and with the same hook positions defined in *hook documentation*.

```
from mmengine.registry import LOOPS, HOOKS
from mmengine.runner import BaseLoop
from mmengine.hooks import Hook
```

(continues on next page)

(continued from previous page)

```
# Customized validation loop
@LOOPS.register_module()
class CustomValLoop(BaseLoop):
    def __init__(self, runner, dataloader, evaluator, dataloader2):
        super().__init__(runner, dataloader, evaluator)
        self.dataloader2 = runner.build_dataloader(dataloader2)

    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()

        # add extra loop for validation purpose
        for idx, data_batch in enumerate(self.dataloader2):
            # add new hooks
            self.runner.call_hooks(
                'before_valloader2_iter', batch_idx=idx, data_batch=data_batch)
            self.run_iter(idx, data_batch)
            # add new hooks
            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')


# Define a hook with extra hook positions
@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):
        ...
```

The example above shows how to implement a different validation loop. The new loop validates on two different validation datasets. It also defines a new hook position in the second validation. You can easily use it by setting `type='CustomValLoop'` in `val_cfg` in your config file.

```
# Customized validation loop
val_cfg = dict(type='CustomValLoop', dataloader2=dict(dataset=dict(type='ValDataset2'), .
↪..))
```

(continues on next page)

(continued from previous page)

```
# Customized hook with extra hook position
custom_hooks = [dict(type='CustomValHook')]
```

29.2 Customize Runner

Moreover, you can write your own runner by subclassing `Runner` if the built-in `Runner` is not feasible. The method is similar to writing other modules: write your subclass inherited from `Runner`, overrides some functions, register it to RUNNERS and access it by assigning `runner_type` in your config file.

```
from mmengine.registry import RUNNERS
from mmengine.runner import Runner

@RUNNERS.register_module()
class CustomRunner(Runner):

    def setup_env(self):
        ...
```

The example above shows how to implement a customized runner which overrides the `setup_env` function and is registered to RUNNERS. Now `CustomRunner` is prepared to be used by setting `runner_type='CustomRunner'` in your config file.

Further readings: *Runner tutorial* and *Runner API documentations*

CHAPTER

THIRTY

EVALUATION

Coming soon. Please refer to chinese documentation.

CHAPTER

THIRTYONE

VISUALIZATION

31.1 1 Overall Design

Visualization provides an intuitive explanation of the training and testing process of the deep learning model. In OpenMMLab, we expect the visualization module to meet the following requirements:

- Provides rich out-of-the-box features that can meet most computer vision visualization tasks.
- Versatile, expandable, and can be customized easily
- Able to visualize at anywhere in the training and testing process.
- Unified APIs for all OpenMMLab libraries, which is convenient for users to understand and use.

Based on the above requirements, we proposed the `Visualizer` and various `VisBackend` such as `LocalVisBackend`, `WandbVisBackend`, and `TensorboardVisBackend` in OpenMMLab 2.0. The visualizer could not only visualize the image data, but also things like configurations, scalars, and model structure.

- For convenience, the APIs provided by the `Visualizer` implement the drawing and storage functions. As an internal property of `Visualizer`, `VisBackend` will be called by `Visualizer` to write data to different backends.
- Considering that you may want to write data to multiple backends after drawing, `Visualizer` can be configured with multiple backends. When the user calls the storage API of the `Visualizer`, it will traverse and call all the specified APIs of `VisBackend` internally.

The UML diagram of the two is as follows.

31.2 2 Visualizer

The external interface of `Visualizer` can be divided into three categories.

1. Drawing APIs

- *draw_bboxes* draws a single or multiple bounding boxes
- *draw_points* draws a single or multiple points
- *draw_texts* draws a single or multiple text boxes
- draw_lines draws a single or multiple line segments
- *draw_circles* draws a single or multiple circles
- *draw_polygons* draws a single or multiple polygons
- draw_binary_masks draws single or multiple binary masks
- *draw_featmap* draws feature map (**static method**)

The above APIs can be called in a chain except for `draw_featmap` because the image size may change after this method is called. To avoid confusion, `draw_featmap` is a static method.

2. Storage APIs

- add_config writes configuration to a specific storage backend
- add_graph writes model graph to a specific storage backend
- add_image writes image to a specific storage backend
- add_scalar writes scalar to a specific storage backend
- add_scalars writes multiple scalars to a specific storage backend at once
- add_datasample the abstract interface for each repositories to draw data sample

Interfaces beginning with the `add` prefix represent storage APIs. [datasample] (`./data_element.md`)is the unified interface of each downstream repository in the OpenMMLab 2.0, and `add_datasample` can process the data sample directly .

3. Other APIs

- *set_image* sets the original image data, the default input image format is RGB
- *get_image* gets the image data in Numpy format after drawing, the default output format is RGB
- *show* for visualization
- *get_backend* gets a specific storage backend by name
- *close* closes all resources, including `VisBackend`

For more details, you can refer to Visualizer Tutorial.

31.3 3 VisBackend

After drawing, the drawn data can be stored in multiple visualization storage backends. To unify the interfaces, MMEngine provides an abstract class, `BaseVisBackend`, and some commonly used backends such as `LocalVisBackend`, `WandbVisBackend`, and `TensorboardVisBackend`. The main interfaces and properties of `BaseVisBackend` are as follows:

- add_config writes configuration to a specific storage backend
- add_graph writes model graph to a specific backend
- add_image writes image to a specific backend
- add_scalar writes scalar to a specific backend
- add_scalars writes multiple scalars to a specific backend at once
- close closes the resource that has been opened
- experiment writes backend objects, such as WandB objects and Tensorboard objects

`BaseVisBackend` defines five common data writing interfaces. Some writing backends are very powerful, such as WandB, which could write tables and videos. Users can directly obtain the `experiment` object for such needs and then call native APIs of the corresponding backend. `LocalVisBackend`, `WandbVisBackend`, and `TensorboardVisBackend` are all inherited from `BaseVisBackend` and implement corresponding storage functions according to their features. Users can also customize `BaseVisBackend` to extend the storage backends and implement custom storage requirements.

For more details, you can refer to *Storage Backend Tutorial*.

CHAPTER

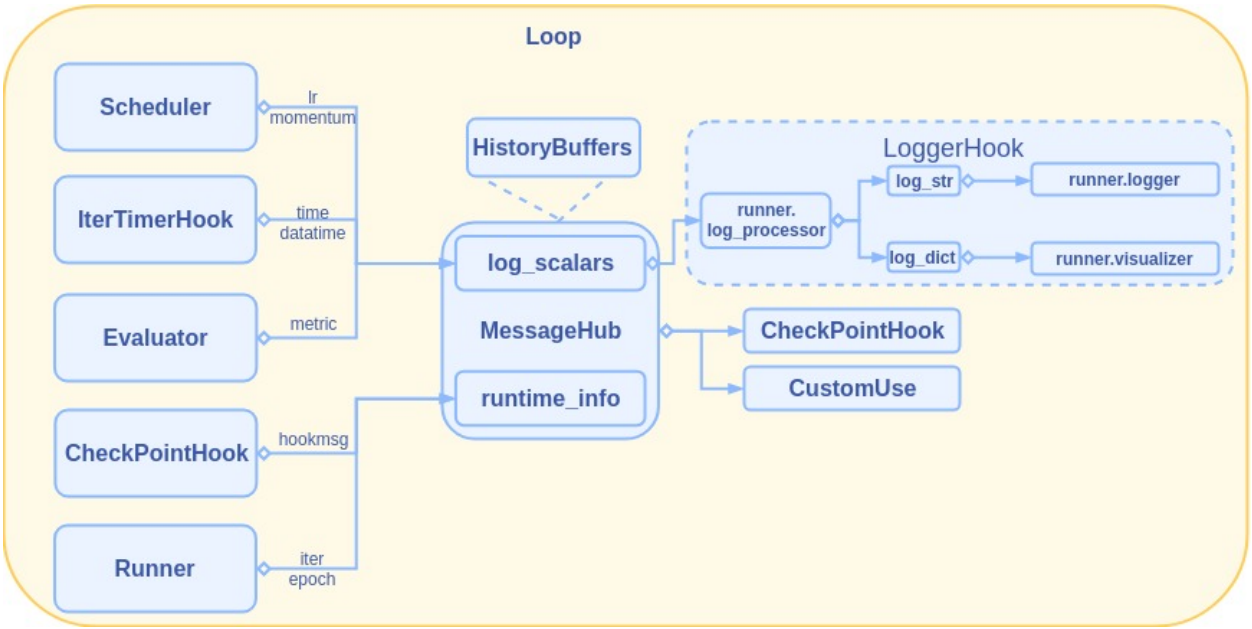
THIRTYTWO

LOGGING

32.1 Overview

Runner produces amounts of logs during execution. These logs include dataset information, model initialization, learning rates, losses, etc. In order to make these logs easily accessed by users, MMEngine designs *MessageHub*, *HistoryBuffer*, *LogProcessor* and *MMLogger*, which enable:

- Configure statistical methods in config files. For example, losses can be globally averaged or smoothed by a sliding window.
- Query training states (iterations, epochs, etc.) in any module
- Configure whether save the multi-process log or not during distributed training.



Each scalar (losses, learning rates, etc.) during training is encapsulated by HistoryBuffer, managed by MessageHub in key-value pairs, formatted by LogProcessor and then exported to various visualization backends by LoggerHook. **In most cases, statistical methods of these scalars can be configured through the LogProcessor without understanding the data flow.** Before diving into the design of the logging system, please read through *logging tutorial* first for familiarizing basic use cases.

32.2 HistoryBuffer

`HistoryBuffer` records the history of the corresponding scalar such as losses, learning rates, and iteration time in an array. As an internal class, it works with *MessageHub*, LoggerHook and *LogProcessor* to make training log configurable. Meanwhile, HistoryBuffer can also be used alone, which enables users to manage their training logs and do various statistics in an easy manner.

We will first introduce the usage of HistoryBuffer in the following section. The association between HistoryBuffer and MessageHub will be introduced later in the MessageHub section.

32.2.1 HistoryBuffer Initialization

HistoryBuffer accepts `log_history`, `count_history` and `max_length` for initialization.

- `log_history` records the history of the scaler. For example, if the loss in the previous 3 iterations is 0.3, 0.2, 0.1 respectively, there will be `log_history=[0.3, 0.2, 0.1]`.
- `count_history` controls the statistical granularity and will be used when counting the average. Take the above example, if we count the average loss across iterations, we have `count_history=[1, 1, 1]`. Instead, if we count the average loss across images with `batch_size=8`, then we have `count_history=[8, 8, 8]`.
- `max_length` controls the maximum length of the history. If the length of `log_history` and `count_history` exceeds `max_length`, the earliest elements will be removed.

Besides, we can access the history of the data through `history_buffer.data`.

```
from mmengine.logging import HistoryBuffer

history_buffer = HistoryBuffer()  # Default initialization
log_history, count_history = history_buffer.data
# [] []
history_buffer = HistoryBuffer([1, 2, 3], [1, 2, 3])  # Init with lists
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
# [2 3] [2 3]
```

32.2.2 HistoryBuffer Update

We can update the `log_history` and `count_history` through `HistoryBuffer.update(log_history, count_history)`.

```
history_buffer = HistoryBuffer([1, 2, 3], [1, 1, 1])
history_buffer.update(4)  # count default to 1
log_history, count_history = history_buffer.data
# [1, 2, 3, 4] [1, 1, 1, 1]
history_buffer.update(5, 2)
log_history, count_history = history_buffer.data
# [1, 2, 3, 4, 5] [1, 1, 1, 1, 2]
```

32.2.3 Basic Statistical Methods

HistoryBuffer provides some basic statistical methods:

- `current()`: Get the latest data.
- `mean(window_size=None)`: Count the mean value of the previous `window_size` data. Defaults to None, as global mean.
- `max(window_size=None)`: Count the max value of the previous `window_size` data. Defaults to None, as global maximum.
- `min(window_size=None)`: Count the min value of the previous `window_size` data. Defaults to None, as global minimum.

```
history_buffer = HistoryBuffer([1, 2, 3], [1, 1, 1])
history_buffer.min(2)
# 2, the minimum in [2, 3]
history_buffer.min()
# 1, the global minimum

history_buffer.max(2)
# 3the maximum in [2, 3]
history_buffer.min()
# 3, the global maximum
history_buffer.mean(2)
# 2.5the mean value in [2, 3], (2 + 3) / (1 + 1)
history_buffer.mean()
# 2, the global mean, (1 + 2 + 3) / (1 + 1 + 1)
history_buffer = HistoryBuffer([1, 2, 3], [2, 2, 2])  # Cases when counts are not 1
history_buffer.mean()
# 1, (1 + 2 + 3) / (2 + 2 + 2)
history_buffer = HistoryBuffer([1, 2, 3], [1, 1, 1])
history_buffer.update(4, 1)
history_buffer.current()
# 4
```

32.2.4 Statistical Methods Invoking

Statistical methods can be accessed through `HistoryBuffer.statistics` with method name and arguments. The `name` parameter should be a registered method name (i.e. built-in methods like `min` and `max`), while arguments should be the corresponding method's arguments.

```
history_buffer = HistoryBuffer([1, 2, 3], [1, 1, 1])
history_buffer.statistics('mean')
# 2, as global mean
history_buffer.statistics('mean', 2)
# 2.5, as the mean of [2, 3]
history_buffer.statistics('mean', 2, 3)
# Error! mismatch arguments given to `mean(window_size)`
history_buffer.statistics('data')
# Error! `data` method not registered
```

32.2.5 Statistical Methods Registration

Custom statistical methods can be registered through `@HistoryBuffer.register_statistics`.

```
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)
```

32.2.6 Use Cases

```
logs = dict(lr=HistoryBuffer(), loss=HistoryBuffer())  # different keys for different␣
↪logs
max_iter = 10
log_interval = 5
for iter in range(1, max_iter+1):
    lr = iter / max_iter * 0.1  # linear scaling of lr
    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')  # select statistical methods by␣
↪name
        mean_loss = logs['loss'].statistics('mean', log_interval)  # mean loss of the␣
↪latest `log_interval` iterations
        print(f'lr:   {latest_lr}\n'
              f'loss: {mean_loss}')
# lr:   0.05
# loss: 0.45666666666666667
# lr:   0.1
# loss: 0.12912698412698415
```

32.3 MessageHub

As shown above, HistoryBuffer can easily handle the update and statistics of a single variable. However, there are multiple variables to log during training, each potentially coming from a different module. This makes it an issue to collect and distribute different variables. To address this issue, we provide MessageHub in MMEngine. It is derived from *ManagerMixin* and thus can be accessed globally. It can be used to simplify the sharing of data across modules.

MessageHub stores data into 2 internal dictionaries, each has its own definition:

- `log_scalars`: Scalars including losses, learning rates and iteration time are collected from different modules and stored into the HistoryBuffer with corresponding key in this dict. Values in this dict will be formatted by *LogProcessor* and then output to terminal or saved locally. If you want to customize your logging info, you can add new keys to this dict and update in the subsequent training steps.
- `runtime_info`: Some runtime information including epochs and iterations are stored in this dict. This dict makes it easy to share some necessary information across modules.

Note: You may need to use MessageHub only if you want to add extra data to logs or share custom data across modules.

The following examples show the usage of MessageHub, including scalars update, data sharing and log customization.

32.3.1 Update & get training log

HistoryBuffers are stored in MessageHub's `log_scalars` dictionary as values. You can call `update_scalars` method to update the HistoryBuffer with the given key. On first call with an unseen key, a HistoryBuffer will be initialized. In the subsequent calls with the same key, the corresponding HistoryBuffer's `update` method will be invoked. You can get values or statistics of a HistoryBuffer by specifying a key in `get_scalar` method. You can also get full logs by directly accessing the `log_scalars` attribute of a MessageHub.

```
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, the latest updated train/loss
message_hub.update_scalar('train/loss', 3, 1)
message_hub.get_scalar('train/loss').mean()  # 2, the mean calculated as (1 + 3) / (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')  # 1

log_dict = message_hub.log_scalars  # return the whole dict
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'])
```

Note: Losses, learning rates and iteration time are automatically updated by runner and hooks. You are not supposed to manually update them.

Note: MessageHub has no special requirements for keys in `log_scalars`. However, MMEngine will only output a scalar to logs if it has a key prfixed with train/val/test.

32.3.2 Update & get runtime info

Runtime information is stored in `runtime_info` dict. The dict accepts data in any data types. Different from HistoryBuffer, the value will be overwritten on every update.

```
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, overwritten by the above command
```

32.3.3 Share MessageHub across modules

During the execution of a runner, different modules receive and post data through MessageHub. Then, *RuntimeInfoHook* gathers data such as losses and learning rates before exporting them to user defined backends (Tensorboard, WandB, etc). Following is an example to show the communication between logger hook and other modules.

```
from mmengine import MessageHub

class LogProcessor:
    # gather data from other modules. similar to logger hook
    def __init__(self, name):
        self.message_hub = MessageHub.get_instance(name)  # access 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:
    # update the learning rate
    def __init__(self, name):
        self.message_hub = MessageHub.get_instance(name)  # access MessageHub

    def run(self):
        self.message_hub.update_scalar('train/lr', 0.001)
        # update the learning rate, saved as HistoryBuffer


class MetaUpdater:
    # update meta information
    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'))    # meta info will be overwritten on every update
```

(continues on next page)

(continued from previous page)

```
class LossUpdater:
    # update losses
    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:
    # compose of different modules
    def __init__(self, name):
        self.message_hub = MessageHub.get_instance(name)  # this will create a global␣
↪MessageHub instance
        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__':
    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'}
```

32.3.4 Add custom logs

Users can update scalars in MessageHub anywhere in any module. All data in `log_scalars` with valid keys are exported to user defined backends after statistical methods.

Note: Only those data in `log_scalars` with keys prefixed with train/val/test are exported.

```
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})
```

By default, the latest value of the custom data(a, b and c) are exported. Users can also configure the *LogProcessor* to switch between statistical methods.

32.4 LogProcessor

Users can configure the LogProcessor to specify the statistical methods and extra arguments. By default, learning rates are displayed by the latest value, while losses and iteration time are counted with an iteration-based smooth method.

32.4.1 Minimum example

```
log_processor = dict(
    window_size=10
)
```

In this configuration, losses and iteration time will be averaged in the latest 10 iterations. The output might be:

```
04/15 12:34:24 - mmengine - INFO - Iter [10/12]  , eta: 0:00:00, time: 0.003, data_time:␣
↪0.002, loss: 0.13
```

32.4.2 Custom statistical methods

Users can configure the `custom_cfg` list to specify the statistical method. Each element in `custom_cfg` must be a dict consisting of the following keys:

- `data_src`: Required argument representing the data source of the log. A data source may have multiple statistical methods. Default sources, which are automatically added to logs, include all keys in loss dict(i.e. `loss`), learning rate(`lr`) and iteration time(`time` & `data_time`). Besides, all scalars updated by MessageHub's `update_scalar/update_scalars` methods with valid keys are configurable data sources, but be aware that the prefix('train/', 'val/', 'test/') should be removed.
- `method_name`: Required argument representing the statistical method. It supports both built-in methods and custom methods.
- `log_name`: Optional argument representing the output name after statistics. If not specified, the new log will overwrite the old one.
- Other arguments: Extra arguments needed by your specified method. `window_size` is a special key, which can be either an int, 'epoch' or 'global'. LogProcessor will parse these arguments and return statistical result based on iteration/epoch/global smooth.

1. Overwrite the old statistical method

```
log_processor = dict(
    window_size=10,
    by_epoch=True,
    custom_cfg=[
        dict(data_src='loss',
             method_name='mean',
             window_size=100)])
```

In this configuration, LogProcessor will overwrite the default window size 10 by a larger window size 100 and output the mean value to 'loss' field in logs.

```
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. New statistical method without overwriting

```
log_processor = dict(
    window_size=10,
    by_epoch=True,
    custom_cfg=[
        dict(data_src='loss',
             log_name='loss_min',
             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
```

32.5 MMLogger

In order to export logs with clear hierarchies, unified formats and less disturbation from third-party logging systems, MMengine implements a `MMLogger` class based on `logging`. It is derived from ManagerMixin. Compared with `logging.logger`, it enables accessing logger in current runner without knowing the logger name.

32.5.1 Instantiate MMLogger

Users can create a global logger by calling `get_instance`. The default log format is shown as below

```
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
```

Apart from user defined messages, the logger will also export timestamps, logger name and log level. ERROR messages are treated specially with red highlight and extra information like error locations.

```
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
```

32.5.2 Export logs

When `get_instance` is invoked with log_file argument, logs will be additionally exported to local storage in text format.

```
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
```

Since distributed applications will create multiple log files, we add a directory with the same name to the exported log file name. Logs from different processes are all saved in this directory. Therefore, the actual log file path in the above example is `tmp/tmp.log`.

32.5.3 Export logs in distributed training

When training with pytorch distributed methods, users can set `distributed=True` in config file to export multiple logs from all processes. If not specified, only master process will export log file.

```
logger = MMLogger.get_instance('mmengine', log_file='tmp.log', distributed=True, log_
↪level='INFO')
```

In the case of multiple processes in a single node, or multiple processes in multiple nodes with shared storage, the exported log files have the following hierarchy

```
#  shared storage case
./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
```

In the case of multiple processes in multiple nodes without storage, logs are organized as follows

```
# without shared storage
# node 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

# node 7
work_dir/
└── exp_name_logs
    ├── exp_name_rank56.log
    ├── exp_name_rank57.log
    ├── exp_name_rank58.log
    ...
    └── exp_name_rank63.log
```

CHAPTER

THIRTYTHREE

MIGRATE RUNNER FROM MMCV TO MMENGINE

33.1 Introduction

As MMCV supports more and more deep learning tasks, and users' needs become much more complicated, we have higher requirements for the flexibility and versatility of the existing `Runner` of MMCV. Therefore, MMEngine implements a more general and flexible `Runner` based on MMCV to support more complicated training processes.

The `Runner` in MMEngine expands the scope and takes on more functions. we abstracted training loop controller (EpochBasedTrainLoop/IterBasedTrainLoop), *validation loop controller (ValLoop)* and *TestLoop* to make it more convenient for users to customize their training process.

Firstly, we will introduce how to migrate the entry point of training from MMCV to MMEngine, to simplify and unify the training script. Then, we'll introduce the difference in the instantiation of `Runner` between MMCV and MMEngine in detail.

33.2 Migrate the entry point

Take MMDet as an example, the differences between training scripts in MMCV and MMEngine are as follows:

33.2.1 Migrate the configuration file

```
# default_runtime.py
checkpoint_config = dict(interval=1)
log_config = dict(
    interval=50,
    hooks=[
        dict(type='TextLoggerHook'),
        # dict(type='TensorboardLoggerHook')
    ])
custom_hooks = [dict(type='NumClassCheckHook')]

dist_params = dict(backend='nccl')
log_level = 'INFO'
load_from = None
resume_from = None
workflow = [('train', 1)]
```

(continues on next page)

(continued from previous page)

```
opencv_num_threads = 0
mp_start_method = 'fork'
auto_scale_lr = dict(enable=False, base_batch_size=16)
```

```
# default_runtime.py
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'),
)

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
```

```
# scheduler.py
# 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)
```

```
# scheduler.py
# 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 = [
```

(continues on next page)

(continued from previous page)

```
    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',
    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)
```

```
# coco_detection.py

# 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']),
```

(continues on next page)

(continued from previous page)

```
        ])
]
data = dict(
    samples_per_gpu=2,
    workers_per_gpu=2,
    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))
evaluation = dict(interval=1, metric='bbox')
```

```
# coco_detection.py

# 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),
```

(continues on next page)

(continued from previous page)

```
    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
```

`Runner` in MMEngine provides more customizable components, including training/validation/testing process and DataLoader. Therefore, the configuration file is a bit longer compared to MMCV.

`MMEngine` follows the WYSIWYG principle and reorganizes the hierarchy of each component in configuration so that most of the first-level fields of configuration correspond to the core components in the `Runner`, such as DataLoader, *Evaluator*, *Hook*, etc. The new format configuration file could help users to read and understand the core components in `Runner`, and ignore the relatively unimportant parts.

33.2.2 Migrate the training script

Compared with the `Runner` in MMCV, `Runner` in MMEngine takes on more functions, such as building DataLoader and distributed model. Therefore, we do not need to build the components like DataLoader and distributed model manually anymore. We can configure them during the instantiation of `Runner`, and then build them in the training/validation/testing process. Take the training script of MMDet as an example:

```
# tools/train.py
args = parse_args()

cfg = Config.fromfile(args.config)

# replace the ${key} with the value of cfg.key
cfg = replace_cfg_vals(cfg)
```

(continues on next page)

(continued from previous page)

```
# 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
    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]
```

(continues on next page)

(continued from previous page)

```
# 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
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 = []
```

(continues on next page)

(continued from previous page)

```
train_detector(
    model,
    datasets,
    cfg,
    distributed=distributed,
    validate=(not args.no_validate),
    timestamp=timestamp,
    meta=meta)
```

```
# tools/train.py
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)

# 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
```

(continues on next page)

(continued from previous page)

```
    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.')

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()
```

```
# apis/train.py
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,
            device_ids=[int(os.environ['LOCAL_RANK'])],
            broadcast_buffers=False,
```

(continues on next page)

(continued from previous page)

```
            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):
            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)
```

(continues on next page)

(continued from previous page)

```
        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`
```

Table above shows the differences between training script of MMEngine `Runner` and MMCV `Runner`. Repositories of OpenMMLab 1.x organize their own process to build `Runner`, which contributes to the large amount of redundant code. MMEngine unifies and formats the building process, such as setting random seed, initializing distributed environment, building DataLoader, building `Optimizer`, etc. This help the downstream repositories simplify the process to prepare the runner, and only need to configure the parameters of `Runner`.

For the downstream repositories, training script based on MMEngine Runner not only simplify the `tools/train.py`, but also can directly omit the `apis/train.py`. Similarly, we can also set random seed, initialize distributed environment by configuring the parameters of `Runner`, and do not need to implement the corresponding code.

33.3 Migrate Runner

This section describes the differences in the training, validation, and testing processes between the MMCV Runner and the MMEngine Runner, as follows.

1. *Prepare logger*
2. *Set random seed*
3. *Initialize environment variables*
4. *Prepare data*
5. *Prepare model*
6. *Prepare optimizer*
7. *Prepare hooks*
8. *Prepare testing/validation components*
9. *Build runner*
10. *Load checkpoint*
11. *Training process*, *Testing process*
12. *Custom training process*

The following tutorial will describe the difference above in detail.

33.3.1 Prepare logger

Prepare logger in MMCV

MMCV needs to call the `get_logger` to get a formatted logger and use it to output and log the training information.

```
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()])
dash_line = '-' * 60 + '\n'
logger.info('Environment info:\n' + dash_line + env_info + '\n' +
            dash_line)
```

The instantiation of the Runner also relies on the logger:

```
runner = Runner(
    ...
    logger=logger
    ...)
```

Prepare logger in MMEngine

Configure the `log_level` for `Runner`, and it will build the logger automatically.

```
log_level = 'INFO'
```

33.3.2 Set random seed

Set random seed in MMCV

Set random seed manually in training script:

```
...
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)
...
```

Set random seed in MMEngine

Configure the `randomness` for `Runner`, see more information in *Runner.set_randomness*

Configuration changes

```
seed = 1
deterministic=False
diff_seed=False
```

```
randomness=dict(seed=1,
                deterministic=True,
                diff_rank_seed=False)
```

33.3.3 Initialize environment variables

Initialize the environment variables

MMCV needs to setup launcher of distributed training, set environment variables for multi-process communication, initialize the distributed environment and wrap model with the distributed wrapper like this:

```
...
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)
```

As for MMEngine, you can setup launcher by configuring `launcher` of `Runner`, and configure other items mentioned above in `env_cfg`. See more information in the table below:

Configuration changes

```
launcher = 'pytorch'  # enable distributed training
dist_params = dict(backend='nccl')  # choose communication backend
```

```
launcher = 'pytorch'
env_cfg = dict(dist_cfg=dict(backend='nccl'))
```

In this tutorial, we set `env_cfg` to:

```
env_cfg = dict(dist_cfg=dict(backend='nccl'))
```

33.3.4 Prepare data

Both MMEngine and MMCV `Runner` can accept built `DataLoader`

```
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)
```

Configuration changes

```
data = dict(
    samples_per_gpu=2,  # batch_size of single gpu
    workers_per_gpu=2,  # num_workers of DataLoader
    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,
    num_workers=2,
    persistent_workers=True,
```

(continues on next page)

(continued from previous page)

```
    # Configurable sampler
    sampler=dict(type='DefaultSampler', shuffle=True),
    # Configurable 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, # batch_size of validation process
    num_workers=2,
    persistent_workers=True,
    drop_last=False, # whether drop the last 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
```

33.3.5 Prepare model

See *Migrate model from mmcv* for more information

```
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)))
```

(continues on next page)

(continued from previous page)

```
        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()
```

33.3.6 Prepare optimizer

Prepare optimizer in MMCV

MMCV Runner can accept built optimizer

```
optimizer = SGD(model.parameters(), lr=0.1, momentum=0.9)
```

For complicated configurations of optimizers, MMCV needs to build optimizers based on the optimizer constructors.

```
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
```

(continues on next page)

(continued from previous page)

```
optimizer = build_optimizer(model, optimizer_cfg)
```

Prepare optimizer in MMEngine

MMEngine needs to configure *optim_wrapper* for `Runner`. For more complicated cases, you can also configure the `optim_wrapper` more specifically. See more information in the API *documents*

Configuration changes

```
optimizer = dict(
    constructor='CustomConstructor',
    type='AdamW',
    lr=0.0001,
    betas=(0.9, 0.999),
    weight_decay=0.05,
    paramwise_cfg={  # parameters of constructor
        'decay_rate': 0.95,
        'decay_type': 'layer_wise',
        'num_layers': 6
    })

# MMCV needs to configure `optim_config` additionally
optimizer_config = dict(grad_clip=None)
```

```
optim_wrapper = dict(
    constructor='CustomConstructor',
    type='OptimWrapper',  # Specify the type of OptimWrapper
    optimizer=dict(  # optimizer configuration
        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
    })
```

Note: For the high-level tasks like detection and classification, MMCV needs to configure `optim_config` to build `OptimizerHook`, while not necessary for MMEngine.

`optim_wrapper` used in this tutorial is as follows:

```
from torch.optim import SGD

optimizer = SGD(model.parameters(), lr=0.1, momentum=0.9)
optim_wrapper = dict(optimizer=optimizer)
```

33.3.7 Prepare hooks

Prepare hooks in MMCV

The commonly used hooks configuration in MMCV is as follows:

```
# 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)
```

Among them:

- `lr_config` is used for `LrUpdaterHook`
- `optimizer_config` is used for `OptimizerHook`
- `checkpoint_config` is used for `CheckPointHook`
- `log_config` is used for `LoggerHook`

Besides the hooks mentioned above, MMCV Runner will build `IterTimerHook` automatically. MMCV `Runner` will register the training hooks after instantiating the model, while MMEngine Runner will initialize the hooks during instantiating the model.

Prepare hooks in MMEngine

MMEngine `Runner` takes some commonly used hooks in MMCV as the default hooks.

- *RuntimeInfoHook*
- *IterTimerHook*
- *DistSamplerSeedHook*
- *LoggerHook*
- *CheckpointHook*
- *ParamSchedulerHook*

Compared with the example of MMCV

- `LrUpdaterHook` correspond to the `ParamSchedulerHook`, find more details in *migrate scheduler*
- MMEngine optimize the model in *train_step*, therefore we do not need `OptimizerHook` in MMEngine anymore
- MMEngine takes `CheckPointHook` as the default hook
- MMEngine take `LoggerHook` as the default hook

Therefore, we can achieve the same effect as the MMCV example as long as we configure the *param_scheduler* correctly.

We can also register custom hooks in MMEngine runner, find more details in *runner tutorial* and *migrate hook*.

```
# Configure training hooks
# Configure LrUpdaterHook
lr_config = dict(
    policy='step',
    warmup='linear',
    warmup_iters=500,
    warmup_ratio=0.001,
    step=[8, 11])

# Configure OptimizerHook
optimizer_config = dict(grad_clip=None)

# Configure LoggerHook
log_config = dict(  # LoggerHook
    interval=50,
    hooks=[
        dict(type='TextLoggerHook'),
        # dict(type='TensorboardLoggerHook')
    ])

# Configure CheckPointHook
checkpoint_config = dict(interval=1)  # CheckPointHook
```

```
# Configure parameter scheduler
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)
]

# Configure default hooks
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'))
```

The parameter scheduler used in this tutorial is as follows:

```
from math import gamma

param_scheduler = dict(type='MultiStepLR', milestones=[2, 3], gamma=0.1)
```

33.3.8 Prepare testing/validation components

MMCV implements the validation process by `EvalHook`, and we'll not talk too much about it here. Given that validation is a common process in training, MMEngine abstracts validation as two independent modules: *Evaluator* and *ValLoop*. We can customize the metric or the validation process by defining a new *loop* or a new metric.

```
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()
            num_sample += batch_size
        return dict(Accuracy=acc / num_sample)
```

After defining the metric, we should also configure the evaluator and loop for `Runner`. The example used in this tutorial is as follows:

```
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
runner.register_hook(
    eval_hook(val_dataloader, **eval_cfg), priority='LOW')
```

```
val_dataloader = val_dataloader
val_evaluator = dict(type='ToyAccuracyMetric')
val_cfg = dict(type='ValLoop')
```

33.3.9 Build Runner

Building Runner in MMCV

```
runner = EpochBasedRunner(
    model=model,
    optimizer=optimizer,
    work_dir=work_dir,
    logger=logger,
    max_epochs=4
)
```

Building Runner in MMEngine

The `EpochBasedRunner` and `max_epochs` arguments in `MMCV` are moved to `train_cfg` in MMEngine. All parameters configurable in `train_cfg` are listed below:

- by_epoch: `True` equivalent to `EpochBasedRunner`. `False` equivalent to `IterBasedRunner`
- `max_epoch/max_iter`: Equivalent to `max_epochs` and `max_iters` in MMCV
- `val_iterval`: Equivalent to `interval` in MMCV

```
from mmengine.runner import Runner

runner = Runner(
    model=model,  # model to be optimized
    work_dir='./work_dir',  # working directory
    randomness=randomness,  # random seed
    env_cfg=env_cfg,  # environment config
    launcher='none',  # launcher for distributed training
    optim_wrapper=optim_wrapper,  # configure optimizer wrapper
    param_scheduler=param_scheduler,  # configure parameter scheduler
    train_dataloader=train_dataloader,  # configure train dataloader
    train_cfg=dict(by_epoch=True, max_epochs=4, val_interval=1),  # Configure training␣
↪loop
    val_dataloader=val_dataloader,  # Configure validation dataloader
    val_evaluator=val_evaluator,  # Configure evaluator and metrics
    val_cfg=val_cfg)  # Configure validation loop
```

33.3.10 Load checkpoint

Loading checkpoint in MMCV

```
if cfg.resume_from:
    runner.resume(cfg.resume_from)
elif cfg.load_from:
    runner.load_checkpoint(cfg.load_from)
```

Loading checkpoint in MMEngine

```
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
```

33.3.11 Training process

Training process in MMCV

Resume or load checkpoint firstly, and then start training.

```
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)
```

Training process in MMEngine

Complete the process mentioned above the `Runner.__init__` and `Runner.train`

```
runner.train()
```

33.3.12 Testing process

Since MMCV Runner does not integrate the test function, we need to implement the test scripts by ourselves.

For MMEngine Runner, as long as we have configured the `test_dataloader`, `test_cfg` and `test_evaluator` for the `Runner`, we can call `Runner.test` to start the testing process.

`work_dir` is the same for training

```
runner = Runner(
    model=model,
    work_dir='./work_dir',
    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()
```

`work_dir` is the different for training, configure load_from manually

```
runner = Runner(
    model=model,
    work_dir='./test_work_dir',
    load_from='./work_dir/epoch_5.pth',  # set load_from additionally
    randomness=randomness,
    env_cfg=env_cfg,
    launcher='none',
    optim_wrapper=optim_wrapper,
```

(continues on next page)

(continued from previous page)

```
    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()
```

33.3.13 Customize training process

If we want to customize a training/validation process, we need to override the `Runner.val` or `Runner.train` in a custom `Runner`. Take overriding `runner.train` as an example, suppose we need to train with the same batch twice for each iteration, we can override the `Runner.train` like this:

```
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
            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
```

In MMEngine, we need to customize a train loop.

```
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)
```

and then, we need to set `type` as `CustomEpochBasedTrainLoop` in `train_cfg`. Note that `by_epoch` and `type` cannot be configured at the same time. Once `by_epoch` is configured, the type of the training loop will be inferred as

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'),
)
runner.train()
```

For more complicated migration needs of Runner, you can refer to the *runner tutorials* and *runner design*.

CHAPTER

THIRTYFOUR

MIGRATE HOOK FROM MMCV TO MMENGINE

Coming soon. Please refer to chinese documentation.

CHAPTER

THIRTYFIVE

MIGRATE MODEL FROM MMCV TO MMENGINE

35.1 Introduction

The early computer vision tasks supported by MMCV, such as detection and classification, used a general process to optimize model. It can be summarized as the following four steps:

1. Calculate the loss
2. Calculate the gradients
3. Update the model parameters
4. Clean the gradients of the last iteration

For most of the high-level tasks, "where" and "when" to perform the above processes is commonly fixed, therefore it seems reasonable to use *Hook* to implement it. MMCV implements series of hooks, such as `OptimizerHook`, `Fp16OptimizerHook` and `GradientCumulativeFp16OptimizerHook` to provide varies of optimization strategies.

On the other hand, tasks like GAN (Generative adversarial network) and Self-supervision require more flexible training processes, which do not meet the characteristics mentioned above, and it could be hard to use hooks to implement them. To meet the needs of these tasks, MMCV will pass `optimizer` to `train_step` and users can customize the optimization process as they want. Although it works, it cannot utilize various `OptimizerHook` implemented in MMCV, and downstream repositories have to implement mix-precision training, and gradient accumulation on their own.

To unify the training process of various deep learning tasks, MMEngine designed the *OptimWrapper*, which integrates the mixed-precision training, gradient accumulation and other optimization strategies into a unified interface.

35.2 Migrate optimization process

Since MMEngine designs the `OptimWrapper` and deprecates series of `OptimizerHook`, there would be some differences between the optimization process in MMCV and MMEngine.

35.2.1 Commonly used optimization process

Considering tasks like detection and classification, the optimization process is usually the same, so `BaseModel` integrates the process into `train_step`.

Model based on MMCV

Before describing how to migrate the model, let's look at a minimal example to train a model based on the MMCV.

```
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)


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(
```

(continues on next page)

(continued from previous page)

```
    lr_config=lr_config,
    optimizer_config=optimizer_config,
    log_config=log_config)
runner.run([train_dataloader], [('train', 1)])
```

Model based on MMCV must implement `train_step`, and return a `dict` which contains the following keys:

- `loss`: Passed to `OptimizerHook` to calculate gradient.
- num_samples: Passed to `LogBuffer` to count the averaged loss
- log_vars: Passed to `LogBuffer` to count the averaged loss

Model based on MMEngine

The same model based on MMEngine

```
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 MMEngineToyModel(BaseModel):

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

    def forward(self, img, label, mode):
        feat = self.linear(img)
        # Called by train_step and return the loss dict
        if mode == 'loss':
            loss1 = (feat - label).pow(2)
            loss2 = (feat - label).abs()
            return dict(loss1=loss1, loss2=loss2)
        # Called by val_step and return the predictions
        elif mode == 'predict':
            return [_feat for _feat in feat]
        # tensor model, find more details in 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),
```

(continues on next page)

(continued from previous page)

```
    optim_wrapper=dict(optimizer=dict(type='SGD', lr=0.01)))
runner.train()
```

In MMEngine, users can customize their model based on `BaseModel`, which implements the same logic as `OptimizerHook` in `train_step`. For high-level tasks, `train_step` will be called in train loop with specific arguments, and users do not need to care about the optimization process. For low-level tasks, users can override the `train_step` to customize the optimization process.

```
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)
```

```
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:
            pass

    # The equivalent code snippet of `train_step`
    # def train_step(self, data, optim_wrapper):
    #     data = self.data_preprocessor(data)
    #     loss_dict = self(*data, mode='loss')
```

(continues on next page)

(continued from previous page)

```
    #     loss_dict['loss1'] = loss_dict['loss1'].sum()
    #     loss_dict['loss2'] = loss_dict['loss2'].sum()
    #     loss = (loss_dict['loss1'] + loss_dict['loss2']).sum()
    #     Call the optimizer wrapper to update parameters.
    #     optim_wrapper.update_params(loss)
    #     return loss_dict
```

Note: See more information about `data_preprocessor` and `optim_wrapper` in docs *optim_wrapper* and *data_preprocessor*.

The main differences of model in MMCV and MMEngine can be summarized as follows:

- `MMCVToyModel` inherits from `nn.Module`, and `MMEngineToyModel` inherits from `BaseModel`
- `MMCVToyModel` must implement `train_step` method and return a `dict` with keys `loss`, `log_vars`, and `num_samples`. `MMEngineToyModel` only needs to implement `forward` method for high level tasks, and return a `dict` with differentiable losses.
- `MMCVToyModel.forward` and `MMEngineToyModel.forward` must match with `train_step` which will call it. Since `MMEngineToyModel` does not override the `train_step`, `BaseModel.train_step` will be directly called, which requires that forward must accept `mode` parameter. Find more details in *tutorials of model*

35.2.2 Custom optimization process

Takes training a GAN model as an example, generator and discriminator need to be optimized in turn and the optimization strategy could change as the training iteration grows. Therefore it could be hard to use `OptimizerHook` to meet such requirements in MMCV. GAN model based on MMCV will accept an optimizer in `train_step` and update parameters in it. Actually, MMEngine borrows this way and simplifies it by passing an *optim_wrapper* rather than an optimizer.

Referred to *training a GAN model*, The differences of MMCV and MMEngine are as follows:

```
    def train_discriminator(self, inputs, optimizer):
        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)
        parsed_losses.backward()
        optimizer.step()
        optimizer.zero_grad()
        return log_vars

    def train_generator(self, inputs, optimizer):
        real_imgs = inputs['inputs']
        z = torch.randn(inputs['inputs'].shape[0], self.noise_size).type_as(
```

(continues on next page)

(continued from previous page)

```
            real_imgs)

        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, 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
```

Apart from the differences mentioned in the previous section, the main difference in the optimization process in MMCV and MMEngine is that the latter can use `optim_wrapper` in a more simple way. The convenience of `optim_wrapper` would be more obvious if gradient accumulation and mix-precision training are applied.

35.3 Migrate validation/testing process

Model based on MMCV usually does not need to provide `test_step` or `val_step` for testing/validation. However, MMEngine performs the testing/validation by *ValLoop* and *TestLoop*, which will call `runner.model.val_step` and `runner.model.test_step`. Therefore model based on MMEngine needs to implement `val_step` and `test_step`, of which input data and output predictions should be compatible with DataLoader and *Evaluator.process* respectively. You can find more details in the *model tutorial*. Therefore, `MMEngineToyModel.forward` will slice the feat and return the predictions as a list.

```
class MMEngineToyModel(BaseModel):

    ...
    def forward(self, img, label, mode):
        if mode == 'loss':
            ...
        elif mode == 'predict':
            # Slice the data to a list
            return [_feat for _feat in feat]
        else:
            ...
```

35.4 Migrate the distributed training

MMCV will wrap the model with distributed wrapper before building the runner, while MMEngine will wrap the model in Runner. Therefore, we need to configure the `launcher` and `model_wrapper_cfg` for Runner. *Migrate Runner from MMCV to MMEngine* will introduce it in detail.

1. **Commonly used training process**

   For the high-level tasks mentioned in *introduction*, the default *distributed model wrapper* is enough. Therefore, we only need to configure the `launcher` for MMEngine Runner.

   ```
   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', # enable distributed training
       ...,
   )
   ```

2. **optimize modules independently with custom optimization process**

   Again, taking the example of training a GAN model, the generator and discriminator need to be optimized separately. Therefore, the model needs to be wrapped by `MMSeparateDistributedDataParallel`, which need to

be specified when building the runner.

```
cfg = dict(model_wrapper_cfg='MMSeparateDistributedDataParallel')
runner = Runner(
    model=model,
    ..., #
    launcher='pytorch',
    cfg=cfg)
```

3. **Optimize a model with a custom optimization process**

Sometimes we need to optimize the whole model with a custom optimization process, where we cannot reuse `BaseModel.train_step`, but need to override it, e.g. we want to optimize the model twice with the same batch of images: the first time with batch data augmentation on, and the second time with it off

```
class CustomModel(BaseModel):

    def train_step(self, data, optim_wrapper):
        data = self.data_preprocessor(data, training=True)  # Enable batch augmentation
        loss = self(data, mode='loss')
        optim_wrapper.update_params(loss)
        data = self.data_preprocessor(data, training=False)  # Disable batch augmentation
        loss = self(data, mode='loss')
        optim_wrapper.update_params(loss)
```

In this case, we need to customize a model wrapper that overrides the `train_step` and performs the same process as `CustomModel.train_step`.

```
   class CustomDistributedDataParallel(MMSeparateDistributedDataParallel):

       def train_step(self, data, optim_wrapper):
           data = self.data_preprocessor(data, training=True)  # Enable batch␣
↪augmentation
           loss = self(data, mode='loss')
           optim_wrapper.update_params(loss)
           data = self.data_preprocessor(data, training=False)  # Disable batch␣
↪augmentation
           loss = self(data, mode='loss')
           optim_wrapper.update_params(loss)
```

Then we can specify it when building Runner:

```
cfg = dict(model_wrapper_cfg=dict(type='CustomDistributedDataParallel'))
runner = Runner(
    model=model,
    ...,
    launcher='pytorch',
    cfg=cfg
)
```

CHAPTER

THIRTYSIX

MIGRATE PARAMETER SCHEDULER FROM MMCV TO MMENGINE

MMCV 1.x version uses LrUpdaterHook and MomentumUpdaterHook to adjust the learning rate and momentum. However, the design of LrUpdaterHook has been difficult to meet more abundant customization requirements due to the development of the training strategies. Hence, MMEngine proposes parameter schedulers (ParamScheduler).

The interface of the parameter scheduler is consistent with PyTroch's learning rate scheduler (LRScheduler). In addition, the parameter scheduler provides stronger functions. For details, please refer to *Parameter Scheduler User Guide*.

36.1 Learning rate scheduler (LrUpdater) migration

MMEngine uses LRScheduler instead of LrUpdaterHook. The field in the config file is changed from the original `lr_config` to `param_scheduler`. The learning rate config in MMCV corresponds to the parameter scheduler config in MMEngine as follows:

36.1.1 Learning rate warm-up migration

The learning rate warm-up can be achieved through the combination of schedulers by specifying the effective range `begin` and `end`. There are 3 learning rate warm-up methods in MMCV, namely `'constant'`, `'linear'`, `'exp'`. The corresponding config in MMEngine should be modified as follows:

Constant warm-up

```
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(...) # the main learning rate scheduler
]
```

Linear warm-up

```
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(...) # the main learning rate scheduler
]
```

Exponential warm-up

```
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(...) # the main learning rate scheduler
]
```

36.1.2 Fixed learning rate (FixedLrUpdaterHook) migration

```
lr_config = dict(policy='fixed')
```

```
param_scheduler = [
    dict(type='ConstantLR', factor=1)
]
```

36.1.3 Step learning rate (StepLrUpdaterHook) migration

```
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)
]
```

36.1.4 Poly learning rate (PolyLrUpdaterHook) migration

```
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)
]
```

36.1.5 Exponential learning rate (ExpLrUpdaterHook) migration

```
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)
]
```

36.1.6 Cosine annealing learning rate (CosineAnnealingLrUpdaterHook) migration

```
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)
]
```

36.1.7 FlatCosineAnnealingLrUpdaterHook migration

The learning rate strategy combined by multiple phases like FlatCosineAnnealing originally needs to be achieved by rewriting a Hook. But in MMEngine, it can be achieved with combining two parameter scheduler configs:

```
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)
]
```

36.1.8 CosineRestartLrUpdaterHook migration

```
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',
```

(continues on next page)

(continued from previous page)

```
        periods=[5, 10, 15],
        restart_weights=[1, 0.7, 0.3],
        eta_min=0.001,
        by_epoch=True)
]
```

36.1.9 OneCycleLrUpdaterHook migration

```
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)
]
```

Notice: by_epoch defaults to False in MMCV. It now defaults to True in MMEngine.

36.1.10 LinearAnnealingLrUpdaterHook migration

```
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)
]
```

36.2 MomentumUpdater migration

MMCV uses `momentum_config` field and MomentumUpdateHook to adjust momentum. The momentum in MMEngine is also controlled by the parameter scheduler. Users can simply change the `LR` of the learning rate scheduler to `Momentum` to use the same strategy to adjust the momentum. The momentum scheduler shares the same `param_scheduler` field in the config with the learning rate scheduler:

```
lr_config = dict(...)
momentum_config = dict(
    policy='CosineAnnealing',
    min_momentum=0.1,
    by_epoch=True
)
```

```
param_scheduler = [
    # config of learning rate schedulers
    dict(...),
    # config of momentum schedulers
    dict(type='CosineAnnealingMomentum',
         eta_min=0.1,
         T_max=num_epochs,
         begin=0,
         end=num_epochs,
         by_epoch=True)
]
```

36.3 Migrate parameter update frequency related config

If you want to update the parameter rate based on iteration while using the epoch-based training loop and setting the effective range (`begin`, `end`) or period (`T_max`) and other variables according to epoch in MMCV, you need to set `by_epoch` to False.

However, in MMEngine, the `by_epoch` in the config still needs to be set to True. Instead, you need to add `convert_to_iter_based=True` in the config to build a parameter scheduler which updates by iteration, see *Parameter Scheduler Tutorial* for more details.

Take the migration of CosineAnnealing as an example:

```
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,  # Notice, by_epoch need to be set to True
        convert_to_iter_based=True  # convert to an iter-based scheduler
```

(continues on next page)

(continued from previous page)

```
    )
]
```

You may also want to read *parameter scheduler tutorial* or parameter scheduler API documentations.

CHAPTER

THIRTYSEVEN

MIGRATE DATA TRANSFORM TO OPENMMLAB 2.0

37.1 Introduction

According to the data transform interface convention of TorchVision, all data transform classes need to implement the `__call__` method. And in the convention of OpenMMLab 1.0, we require the input and output of the `__call__` method should be a dictionary.

In OpenMMLab 2.0, to make the data transform classes more extensible, we use `transform` method instead of `__call__` method to implement data transformation, and all data transform classes should inherit the `mmcv.transforms.BaseTransfrom` class. And you can still use these data transform classes by calling.

A tutorial to implement a data transform class can be found in the *Data Transform*.

In addition, we move some common data transform classes from every repositories to MMCV, and in this document, we will compare the functionalities, usages and implementations between the original data transform classes (in MMClassification v0.23.2, MMDetection v2.25.1) and the new data transform classes (in MMCV v2.0.0rc1)

37.2 Functionality Differences

37.3 Implementation Differences

Take `RandomFlip` as example, the new version RandomFlip in MMCV inherits `BaseTransfrom`, and move the functionality implementation from `__call__` to `transform` method. In addition, the randomness related code is placed in some extra methods and these methods need to be wrapped by `cache_randomness` decorator.

- MMDetection (original version)

```
class RandomFlip:
    def __call__(self, results):
        """Randomly flip images."""
        ...
        # Randomly choose the flip direction
        cur_dir = np.random.choice(direction_list, p=flip_ratio_list)
        ...
        return results
```

- MMCV (new version)

```
class RandomFlip(BaseTransfrom):
    def transform(self, results):
```

(continues on next page)

(continued from previous page)

```
        """Randomly flip images"""
        ...
        cur_dir = self._random_direction()
        ...
        return results

    @cache_randomness
    def _random_direction(self):
        """Randomly choose the flip direction"""
        ...
        return np.random.choice(direction_list, p=flip_ratio_list)
```

CHAPTER

THIRTYEIGHT

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.

38.1 Registry

class mmengine.registry.**Registry**(*name*, *build_func=None*, *parent=None*, *scope=None*, *locations=[]*)

A registry to map strings to classes or functions.

Registered object could be built from registry. Meanwhile, registered functions could be called from registry.

Parameters

- **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 registry. If not specified, scope will be the name of the package where class is defined, e.g. mmdet, mmcls, mmseg. Defaults to None.
- **locations** (*list*) – The locations to import the modules registered in this registry. Defaults to []. New in version 0.4.0.

Examples

```
>>> # 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'))
```

(continues on next page)

(continued from previous page)

```
>>> @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'))
```

```
>>> # add locations to enable auto import
>>> DETECTORS = Registry('detectors', parent=MODELS,
>>>     scope='det', locations=['det.models.detectors'])
>>> # define this class in 'det.models.detectors'
>>> @DETECTORS.register_module()
>>> class MaskRCNN:
>>>     pass
>>> # The registry will auto import det.models.detectors.MaskRCNN
>>> fasterrcnn = DETECTORS.build(dict(type='det.MaskRCNN'))
```

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`.

Parameters **cfg** (*dict*) – Config dict needs to be built.

Returns The constructed object.

Return type Any

Examples

```
>>> 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.

Parameters **key** (*str*) – Name of the registered item, e.g., the class name in string format.

Returns Return the corresponding class if `key` exists, otherwise return None.

Return type Type or None

Examples

```
>>> # 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
>>> # 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')
```

import_from_location()
import modules from the pre-defined locations in self._location.

Return type None

static infer_scope()
Infer the scope of registry.

The name of the package where registry is defined will be returned.

Returns The inferred scope name.

Return type str

Examples

```
>>> # 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.

Parameters

- **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.

Return type Union[type, collections.abc.Callable]

Examples

```
>>> 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.

Returns The former element is the first scope of the key, which can be `None`. The latter is the remaining key.

Return type tuple[str | None, str]

Parameters **key** (*str*) –

Examples

```
>>> 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.

Parameters **scope** (*str*) – The target scope.

Return type Generator

Examples

```
>>> 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
```

(continues on next page)

(continued from previous page)

```
mmcls
>>>
>>> # Check switch back to original scope.
>>> DefaultScope.get_current_instance().scope_name
custom
```

38.2 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 `Registry` 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.

Parameters

- **name** (*str*) – Name of default scope for global access.
- **scope_name** (*str*) – Scope of current task.

Examples

```
>>> 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.

Examples

```
>>> 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
```

Returns Return None If there has not been `DefaultScope` instance created yet, otherwise return the latest created DefaultScope instance.

Return type Optional[*DefaultScope*]

classmethod overwrite_default_scope(*scope_name*)
overwrite the current default scope with *scope_name*

Parameters **scope_name** (*Optional[str]*) –

Return type 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.
init_default_scope	Initialize the given default scope.

38.3 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.

Examples

```
>>> 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()
```

(continues on next page)

(continued from previous page)

```
>>> def resnet50():
>>>     pass
>>> resnet = build_from_cfg(dict(type='resnet50'), MODELS)
>>> # Return a result of the calling function
```

Parameters

- **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.

Returns The constructed object.

Return type object

38.4 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.

Parameters

- **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.

Returns A built nn.Module.

Return type nn.Module

38.5 mmengine.registry.build_runner_from_cfg

mmengine.registry.**build_runner_from_cfg**(*cfg*, *registry*)

Build a Runner object. .. rubric:: Examples

```
>>> 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)
```

Parameters

- **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.

Returns The constructed runner object.

Return type object

38.6 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.

Parameters

- **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.

Returns The constructed `ParamScheduler`.

Return type object

38.7 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.

Parameters

- **save_path** (*str, optional*) – Path to save the json file.
- **verbose** (*bool*) – Whether to print log. Defaults to True.

Returns Statistic results of all registered modules.

Return type dict

38.8 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.

Parameters

- **registry** (Registry) – a registry node in the registry tree.
- **verbose** (*bool*) – Whether to print log. Default: True

Returns Statistic results of all modules in each node of the registry tree.

Return type list

38.9 mmengine.registry.init_default_scope

mmengine.registry.**init_default_scope**(*scope*)

Initialize the given default scope.

Parameters **scope** (*str*) – The name of the default scope.

Return type None

CHAPTER

THIRTYNINE

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.

39.1 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.

Parameters

- **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.

Examples

```
>>> 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
```

(continues on next page)

(continued from previous page)

```
"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.

Parameters

- **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*) –

Returns Config text.

Return type 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.

Parameters

- **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.

Returns Config instance built from config file.

Return type *Config*

static fromstring(*cfg_str*, *file_format*)
Build a Config instance from config text.

Parameters

- **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!

Returns Config object generated from `cfg_str`.

Return type *Config*

merge_from_dict(*options*, *allow_list_keys=True*)
Merge list into cfg_dict.

Merge the dict parsed by MultipleKVAction into this cfg.

Parameters

- **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.

Return type None

Examples

```
>>> 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')})
>>> 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.

39.2 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`.

39.3 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

FORTY

MMENGINE.RUNNER

mmengine.runner

40.1 Runner

Runner	A training helper for PyTorch.

40.1.1 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.

Parameters

- **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.

Examples

```
>>> 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,
>>>     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'),
```

(continues on next page)

(continued from previous page)

```
>>>         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
)
```

Parameters

- **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.

Returns DataLoader build from `dataloader_cfg`.

Return type 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())
# metric is a list
evaluator = dict(metrics=[ToyMetric()])
```

Parameters **evaluator** (*Evaluator or dict or list*) – An Evaluator object or a config dict or list of config dict used to build an Evaluator.

Returns Evaluator build from `evaluator`.

Return type *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')

Parameters

- **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*) –

Returns Log processor object build from `log_processor_cfg`.

Return type *LogProcessor*

build_logger(*log_level='INFO'*, *log_file=None*, ***kwargs*)
Build a global asscessable MMLogger.

Parameters

- **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`.

Returns A MMLogger object build from `logger`.

Return type *MMLogger*

build_message_hub(*message_hub=None*)
Build a global asscessable MessageHub.

Parameters **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.

Returns A MessageHub object build from `message_hub`.

Return type *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')
```

Parameters **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.

Return type torch.nn.modules.module.Module

Note: 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*.

Returns Model build from `model`.

Return type nn.Module

Parameters **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`.

Parameters **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.

Return type Union[*mmengine.optim.optimizer.optimizer_wrapper.OptimWrapper*, *mmengine.optim.optimizer.optimizer_wrapper_dict.OptimWrapperDict*]

Note: 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.

Examples

```
>>> # 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
    momentum: 0
    nesterov: False
    weight_decay: 0
)
>>> # build multiple optimizers
>>> optim_wrapper_cfg = dict(
```

(continues on next page)

(continued from previous page)

```
...     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
)
```

Important: 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.

Returns Optimizer wrapper build from `optimizer_cfg`.

Return type *OptimWrapper*

Parameters **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.

Parameters **scheduler** (*_ParamScheduler or dict or list*) – A Param Scheduler object or a dict or list of dict to build parameter schedulers.

Return type Union[List[*mmengine.optim.scheduler.param_scheduler._ParamScheduler*], Dict[str, List[*mmengine.optim.scheduler.param_scheduler._ParamScheduler*]]]

Examples

```
>>> # 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`_**.

Returns List of parameter schedulers or a dictionary contains list of parameter schedulers build from `scheduler`.

Return type list[*_ParamScheduler*] or dict[str, list[*_ParamScheduler*]]

Parameters **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()
```

(continues on next page)

(continued from previous page)

```
# custom test loop
loop = dict(type='CustomTestLoop')
```

Parameters **loop** (*BaseLoop or dict*) – A test loop or a dict to build test loop. If `loop` is a test loop object, just returns itself.

Returns Test loop object build from `loop_cfg`.

Return type *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)
```

Parameters **loop** (*BaseLoop or dict*) – A training loop or a dict to build training loop. If `loop` is a training loop object, just returns itself.

Returns Training loop object build from `loop`.

Return type *BaseLoop*

build_val_loop(*loop*)

Build validation loop.

Examples of `loop`:

ValLoop will be used loop = dict()

custom validation loop loop = dict(type='CustomValLoop')

Parameters **loop** (*BaseLoop or dict*) – A validation loop or a dict to build validation loop. If `loop` is a validation loop object, just returns itself.

Returns Validation loop object build from `loop`.

Return type *BaseLoop*

build_visualizer(*visualizer=None*)

Build a global asscessable Visualizer.

Parameters **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.

Returns A Visualizer object build from `visualizer`.

Return type *Visualizer*

call_hook(*fn_name*, ***kwargs*)
Call all hooks.

Parameters

- **fn_name** (*str*) – The function name in each hook to be called, such as "before_train_epoch".
- ****kwargs** – Keyword arguments passed to hook.

Return type 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*.

Return type None

property epoch
Current epoch.

Type int

property experiment_name
Name of experiment.

Type str

classmethod from_cfg(*cfg*)
Build a runner from config.

Parameters **cfg** (*ConfigType*) – A config used for building runner. Keys of `cfg` can see `__init__()`.

Returns A runner build from `cfg`.

Return type *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`.

Parameters

- **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.

Return type 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.

Parameters **hooks** (*list[Hook | dict]*) – List of hooks or configs to be registered.

Return type 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`.

Parameters **hooks** (*dict[str, Hook or dict], optional*) – Default hooks or configs to be registered.

Return type 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`.

Parameters

- **hook** (Hook or dict) – The hook to be registered.
- **priority** (int or str or *Priority*, optional) – Hook priority. Lower value means higher priority.

Return type None

register_hooks(*default_hooks=None*, *custom_hooks=None*)

Register default hooks and custom hooks into hook list.

Parameters

- **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.

Return type None

resume(*filename*, *resume_optimizer=True*, *resume_param_scheduler=True*, *map_location='default'*)

Resume model from checkpoint.

Parameters

- **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'.

Return type 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.

Parameters

- **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.

Note: `scale_lr` must be called after building optimizer wrappers and before building parameter schedulers.

Parameters

- **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.

Return type 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.

Parameters

- **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.

Return type 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
)
```

Parameters **env_cfg** (*dict*) – Config for setting environment.

Return type None

test()
Launch test.

Returns A dict of metrics on testing set.

Return type 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.

Returns The model after training.

Return type nn.Module

property train_dataloader
The data loader for training.

property train_loop
A loop to run training.

Type *BaseLoop*

val()
Launch validation.

Returns A dict of metrics on validation set.

Return type 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
)
```

Parameters

- **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.

Returns nn.Module or subclass of `DistributedDataParallel`.

Return type nn.Module or DistributedDataParallel

40.2 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.

40.2.1 BaseLoop

class mmengine.runner.**BaseLoop**(*runner*, *dataloader*)

Base loop class.

All subclasses inherited from `BaseLoop` should overwrite the *run()* method.

Parameters

- **runner** (Runner) – A reference of runner.
- **dataloader** (*Dataloader or dict*) – An iterator to generate one batch of dataset each iteration.

Return type None

abstract run()

Execute loop.

Return type Any

40.2.2 EpochBasedTrainLoop

class mmengine.runner.**EpochBasedTrainLoop**(*runner*, *dataloader*, *max_epochs*, *val_begin=1*, *val_interval=1*, *dynamic_intervals=None*)

Loop for epoch-based training.

Parameters

- **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.

Return type 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.

Return type torch.nn.modules.module.Module

run_epoch()

Iterate one epoch.

Return type None

run_iter(*idx*, *data_batch*)

Iterate one min-batch.

Parameters **data_batch** (*Sequence[dict]*) – Batch of data from dataloader.

Return type None

40.2.3 IterBasedTrainLoop

class mmengine.runner.**IterBasedTrainLoop**(*runner*, *dataloader*, *max_iters*, *val_begin=1*, *val_interval=1000*, *dynamic_intervals=None*)

Loop for iter-based training.

Parameters

- **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.

Return type 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.

Return type None

run_iter(*data_batch*)
Iterate one mini-batch.

Parameters **data_batch** (*Sequence[dict]*) – Batch of data from dataloader.

Return type None

40.2.4 ValLoop

class mmengine.runner.**ValLoop**(*runner*, *dataloader*, *evaluator*, *fp16=False*)
Loop for validation.

Parameters

- **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.

Return type None

run()
Launch validation.

Return type dict

run_iter(*idx*, *data_batch*)
Iterate one mini-batch.

Parameters **data_batch** (*Sequence[dict]*) – Batch of data from dataloader.

40.2.5 TestLoop

class mmengine.runner.**TestLoop**(*runner*, *dataloader*, *evaluator*, *fp16=False*)
Loop for test.

Parameters

- **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.

Return type dict

run_iter(*idx*, *data_batch*)
Iterate one mini-batch.

Parameters **data_batch** (*Sequence[dict]*) – Batch of data from dataloader.

Return type None

40.3 Checkpoints

CheckpointLoader	A general checkpoint loader to manage all schemes.

40.3.1 CheckpointLoader

class mmengine.runner.**CheckpointLoader**

A general checkpoint loader to manage all schemes.

classmethod load_checkpoint(*filename*, *map_location=None*, *logger='current'*)

load checkpoint through URL scheme path.

Parameters

- **filename** (*str*) – checkpoint file name with given prefix
- **map_location** (*str, optional*) – Same as `torch.load()`. Default: None
- **logger** (*str*) – The logger for message. Defaults to 'current'.

Returns The loaded checkpoint.

Return type 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.

Parameters

- **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.
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.

40.3.2 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

Parameters **path** (*str*) – The path to find checkpoints.

Returns File path of the latest checkpoint.

Return type str or None

40.3.3 mmengine.runner.get_deprecated_model_names

mmengine.runner.**get_deprecated_model_names**()

40.3.4 mmengine.runner.get_external_models

mmengine.runner.**get_external_models**()

40.3.5 mmengine.runner.get_mmcls_models

mmengine.runner.**get_mmcls_models**()

40.3.6 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)).

Parameters

- **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.

Returns A dictionary containing a whole state of the module.

Return type dict

40.3.7 mmengine.runner.get_torchvision_models

mmengine.runner.**get_torchvision_models**()

40.3.8 mmengine.runner.load_checkpoint

mmengine.runner.**load_checkpoint**(*model*, *filename*, *map_location=None*, *strict=False*, *logger=None*, *revise_keys=[('^module\\.', '')]*)

Load checkpoint from a file or URI.

Parameters

- **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.', '')].

Returns The loaded checkpoint.

Return type dict or OrderedDict

40.3.9 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.

Parameters

- **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.

40.3.10 mmengine.runner.save_checkpoint

mmengine.runner.**save_checkpoint**(*checkpoint*, *filename*, *file_client_args=None*, *backend_args=None*)
Save checkpoint to file.

Parameters

- **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.

40.3.11 mmengine.runner.weights_to_cpu

mmengine.runner.**weights_to_cpu**(*state_dict*)
Copy a model state_dict to cpu.

Parameters **state_dict** (*OrderedDict*) – Model weights on GPU.

Returns Model weights on GPU.

Return type OrderedDict

40.4 AMP

autocast	A wrapper of `torch.autocast` and `toch.cuda.amp.autocast`.

40.4.1 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.

Note: `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.

Examples

```
>>> # 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
```

Parameters

- **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.

40.5 Miscellaneous

LogProcessor	A log processor used to format log information collected from `runner.message_hub.log_scalars`.
Priority	Hook priority levels.

40.5.1 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.

Parameters

- **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.

Examples

```
>>> # `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,
```

(continues on next page)

(continued from previous page)

```
>>>     custom_cfg=[dict(data_src='loss',
>>>                      method_name='mean',
>>>                      window_size=100),
>>>                 dict(data_src='loss',
>>>                      method_name='max',
>>>                      window_size=100)])
AssertionError
```

get_log_after_epoch(*runner*, *batch_idx*, *mode*)
Format log string after validation or testing epoch.

Parameters

- **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.

Returns Formatted log dict/string which will be recorded by `runner.message_hub` and `runner.visualizer`.

Return type Tuple(dict, str)

get_log_after_iter(*runner*, *batch_idx*, *mode*)
Format log string after training, validation or testing epoch.

Parameters

- **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.

Returns Formatted log dict/string which will be recorded by `runner.message_hub` and `runner.visualizer`.

Return type Tuple(dict, str)

40.5.2 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.

40.5.3 mmengine.runner.get_priority

mmengine.runner.**get_priority**(*priority*)

Get priority value.

Parameters **priority** (int or str or *Priority*) – Priority.

Returns The priority value.

Return type int

CHAPTER

FORTYONE

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.
ProfilerHook	A hook to analyze performance during training and inference.
PrepareTTAHook	Wraps *runner.model* with subclass of `BaseTTAModel` in *before_test*.

41.1 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.

Parameters

- **runner** (*Runner*) – The runner of the training, validation or testing process.
- **checkpoint** (*dict*) – Model's checkpoint.

Return type None

after_run(*runner*)
All subclasses should override this method, if they need any operations before the training validation or testing process.

Parameters **runner** (Runner) – The runner of the training, validation or testing process.

Return type None

after_test(*runner*)
All subclasses should override this method, if they need any operations after testing.

Parameters **runner** (Runner) – The runner of the testing process.

Return type None

after_test_epoch(*runner*, *metrics=None*)
All subclasses should override this method, if they need any operations after each test epoch.

Parameters

- **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.

Return type 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.

Parameters

- **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.

Return type None

after_train(*runner*)
All subclasses should override this method, if they need any operations after train.

Parameters **runner** (Runner) – The runner of the training process.

Return type None

after_train_epoch(*runner*)
All subclasses should override this method, if they need any operations after each training epoch.

Parameters **runner** (Runner) – The runner of the training process.

Return type 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.

Parameters

- **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.

Return type None

after_val(*runner*)
All subclasses should override this method, if they need any operations after validation.

Parameters **runner** (Runner) – The runner of the validation process.

Return type None

after_val_epoch(*runner*, *metrics=None*)
All subclasses should override this method, if they need any operations after each validation epoch.

Parameters

- **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.

Return type 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.

Parameters

- **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.

Return type None

before_run(*runner*)
All subclasses should override this method, if they need any operations before the training validation or testing process.

Parameters **runner** (Runner) – The runner of the training, validation or testing process.

Return type None

before_save_checkpoint(*runner*, *checkpoint*)
All subclasses should override this method, if they need any operations before saving the checkpoint.

Parameters

- **runner** (Runner) – The runner of the training, validation or testing process.
- **checkpoint** (*dict*) – Model's checkpoint.

Return type None

before_test(*runner*)
All subclasses should override this method, if they need any operations before testing.

Parameters **runner** (Runner) – The runner of the testing process.

Return type None

before_test_epoch(*runner*)
All subclasses should override this method, if they need any operations before each test epoch.

Parameters **runner** (Runner) – The runner of the testing process.

Return type 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.

Parameters

- **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.

Return type None

before_train(*runner*)
All subclasses should override this method, if they need any operations before train.

Parameters **runner** (Runner) – The runner of the training process.

Return type None

before_train_epoch(*runner*)
All subclasses should override this method, if they need any operations before each training epoch.

Parameters **runner** (Runner) – The runner of the training process.

Return type 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.

Parameters

- **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.

Return type None

before_val(*runner*)
All subclasses should override this method, if they need any operations before validation.

Parameters **runner** (Runner) – The runner of the validation process.

Return type None

before_val_epoch(*runner*)
All subclasses should override this method, if they need any operations before each validation epoch.

Parameters **runner** (Runner) – The runner of the validation process.

Return type 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.

Parameters

- **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.

Return type None

end_of_epoch(*dataloader*, *batch_idx*)
Check whether the current iteration reaches the last iteration of the dataloader.

Parameters

- **dataloader** (*Dataloader*) – The dataloader of the training, validation or testing process.
- **batch_idx** (*int*) – The index of the current batch in the loop.

Returns Whether reaches the end of current epoch or not.

Return type bool

every_n_epochs(*runner*, *n*)
Test whether current epoch can be evenly divided by n.

Parameters

- **runner** (Runner) – The runner of the training, validation or testing process.
- **n** (*int*) – Whether current epoch can be evenly divided by n.

Returns Whether current epoch can be evenly divided by n.

Return type bool

every_n_inner_iters(*batch_idx*, *n*)
Test whether current inner iteration can be evenly divided by n.

Parameters

- **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.

Returns Whether current inner iteration can be evenly divided by n.

Return type bool

every_n_train_iters(*runner*, *n*)
Test whether current training iteration can be evenly divided by n.

Parameters

- **runner** (Runner) – The runner of the training, validation or testing process.
- **n** (*int*) – Whether current iteration can be evenly divided by n.

Returns Return True if the current iteration can be evenly divided by n, otherwise False.

Return type bool

is_last_train_epoch(*runner*)
Test whether current epoch is the last train epoch.

Parameters **runner** (Runner) – The runner of the training process.

Returns Whether reaches the end of training epoch.

Return type bool

is_last_train_iter(*runner*)
Test whether current iteration is the last train iteration.

Parameters **runner** (Runner) – The runner of the training process.

Returns Whether current iteration is the last train iteration.

Return type bool

41.2 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.

Parameters

- **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.

Return type None

Examples

```
>>> # 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')
>>> # 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.

Parameters **runner** (Runner) – The runner of the training process.

Return type None

after_train_iter(*runner*, *batch_idx*, *data_batch=None*, *outputs=typing.Union[dict, NoneType]*)

Save the checkpoint and synchronize buffers after each iteration.

Parameters

- **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.

Return type None

after_val_epoch(*runner*, *metrics*)

Save the checkpoint and synchronize buffers after each evaluation epoch.

Parameters

- **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.

Parameters **runner** (Runner) – The runner of the training process.

Return type None

41.3 EMAHook

class mmengine.hooks.**EMAHook**(*ema_type='ExponentialMovingAverage'*, *strict_load=False*, *begin_iter=0*, *begin_epoch=0*, ***kwargs*)

A Hook to apply Exponential Moving Average (EMA) on the model during training.

Note:

- 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.

Parameters

- **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 False. Changed in v0.3.0.
- **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.

Parameters

- **runner** (Runner) – The runner of the testing process.
- **checkpoint** (*dict*) –

Return type None

after_test_epoch(*runner*, *metrics=None*)

We recover source model's parameter from ema model after test.

Parameters

- **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.

Return type None

after_train_iter(*runner*, *batch_idx*, *data_batch=None*, *outputs=None*)

Update ema parameter.

Parameters

- **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.

Return type None

after_val_epoch(*runner*, *metrics=None*)
We recover source model's parameter from ema model after validation.

Parameters

- **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.

Return type None

before_run(*runner*)
Create an ema copy of the model.

Parameters **runner** (Runner) – The runner of the training process.

Return type None

before_save_checkpoint(*runner*, *checkpoint*)
Save ema parameters to checkpoint.

Parameters

- **runner** (Runner) – The runner of the testing process.
- **checkpoint** (*dict*) –

Return type None

before_test_epoch(*runner*)
We load parameter values from ema model to source model before test.

Parameters **runner** (Runner) – The runner of the training process.

Return type None

before_train(*runner*)
Check the begin_epoch/iter is smaller than max_epochs/iters.

Parameters **runner** (Runner) – The runner of the training process.

Return type None

before_val_epoch(*runner*)
We load parameter values from ema model to source model before validation.

Parameters **runner** (Runner) – The runner of the training process.

Return type None

41.4 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.

Parameters

- **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.

Examples

```
>>> # 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`

Parameters **runner** (Runner) – The runner of the training/testing/validation process.

Return type None

after_test_epoch(*runner*, *metrics=None*)
All subclasses should override this method, if they need any operations after each test epoch.

Parameters

- **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.

Return type None

after_test_iter(*runner*, *batch_idx*, *data_batch=None*, *outputs=None*)
Record logs after testing iteration.

Parameters

- **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.

Return type None

after_train_iter(*runner*, *batch_idx*, *data_batch=None*, *outputs=None*)
Record logs after training iteration.

Parameters

- **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.

Return type None

after_val_epoch(*runner*, *metrics=None*)
All subclasses should override this method, if they need any operations after each validation epoch.

Parameters

- **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.

Return type None

after_val_iter(*runner*, *batch_idx*, *data_batch=None*, *outputs=None*)
Record logs after validation iteration.

Parameters

- **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.

Return type None

before_run(*runner*)
Infer `self.file_client` from `self.out_dir`. Initialize the `self.start_iter` and record the meta information.

Parameters **runner** (Runner) – The runner of the training process.

Return type None

41.5 NaiveVisualizationHook

class mmengine.hooks.**NaiveVisualizationHook**(*interval=1*, *draw_gt=True*, *draw_pred=True*)
Show or Write the predicted results during the process of testing.

Parameters

- **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.

Parameters

- **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.

Return type None

before_train(*runner*)
Call add_graph method of visualizer.

Parameters **runner** (Runner) – The runner of the training process.

Return type None

41.6 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.

Parameters **runner** (Runner) – The runner of the training process.

Return type None

after_train_iter(*runner*, *batch_idx*, *data_batch=None*, *outputs=None*)

Call step function for each scheduler after each iteration.

Parameters

- **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.

Return type None

41.7 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.

Parameters

- **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.

Return type None

after_train_iter(*runner*, *batch_idx*, *data_batch=None*, *outputs=None*)

Update `log_vars` in model outputs every iteration.

Parameters

- **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.

Return type None

after_val_epoch(*runner*, *metrics=None*)
All subclasses should override this method, if they need any operations after each validation epoch.

Parameters

- **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.

Return type None

before_run(*runner*)
Update metainfo.

Parameters **runner** (Runner) – The runner of the training process.

Return type None

before_train(*runner*)
Update resumed training state.

Parameters **runner** (Runner) – The runner of the training process.

Return type None

before_train_epoch(*runner*)
Update current epoch information before every epoch.

Parameters **runner** (Runner) – The runner of the training process.

Return type None

before_train_iter(*runner*, *batch_idx*, *data_batch=None*)
Update current iter and learning rate information before every iteration.

Parameters

- **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.

Return type None

41.8 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.

Parameters **runner** (Runner) – The runner of the training process.

Return type None

41.9 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.

Parameters **runner** – The runner of the training, validation or testing process.

Return type None

41.10 SyncBuffersHook

class mmengine.hooks.**SyncBuffersHook**
Synchronize model buffers such as running_mean and running_var in BN at the end of each epoch.

Return type None

after_train_epoch(*runner*)
All-reduce model buffers at the end of each epoch.

Parameters **runner** (Runner) – The runner of the training process.

Return type None

41.11 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.

Parameters

- **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.

Return type None

41.12 ProfilerHook

class mmengine.hooks.**ProfilerHook**(**, by_epoch=True, profile_times=1, activity_with_cpu=True, activity_with_cuda=False, schedule=None, on_trace_ready=None, record_shapes=False, profile_memory=False, with_stack=False, with_flops=False, json_trace_path=None*)
A hook to analyze performance during training and inference.

PyTorch Profiler is a tool that allows the collection of the performance metrics during the training. More details on Profiler can be found at official docs

Parameters

- **by_epoch** (*bool*) – Profile performance by epoch or by iteration. Defaults to True.
- **profile_times** (*int*) – The period (epoch/iter) recorded by the profiler. Defaults to 1. For example, profile_iters=10 and by_epoch=False, indicate that 0-10 iterations are recorded.
- **activity_with_cpu** (*bool*) – Activities to be used in the analysis (CPU)
- **activity_with_cuda** (*bool*) – Activities to be used in the analysis (CUDA)
- **schedule** (*dict, optional*) – Key-word arguments passed to torch.profile.schedule. Defaults to None, which means profiling without a schedule
- **on_trace_ready** (*callable, dict, optional*) – Either a handler or a dict of generating handler. Defaults to None, which means profiling without an on_trace_ready.The Callable type needs to construct its own function that can handle 'torch.autograd.profiler.profile'. Two officially recommended ways are provided, namely terminal display or tensorboard display. The terminal display content can be adjusted through 'EventList.table()' from 'torch.autograd.profiler_util.py'. If using tensorboard, save to '{work_dir}/tf_tracing_logs' by default.
- **record_shapes** (*bool*) – Save information about operator's input shapes. Defaults to False.
- **profile_memory** (*bool*) – Track tensor memory allocation/deallocation. Defaults to False.
- **with_stack** (*bool*) – Record source information (file and line number) for the ops. Defaults to False.
- **with_flops** (*bool*) – Use formula to estimate the FLOPS of specific operators (matrix multiplication and 2D convolution). Defaults to False.
- **json_trace_path** (*str, optional*) – Exports the collected trace in Chrome JSON format. Chrome use 'chrome://tracing' view json file. Defaults to None, which means profiling does not store json files.

Return type None

Examples

```
>>> # tensorboard trace
>>> trace_config = dict(type='tb_trace')
>>> profiler_hook_cfg = dict(on_trace_ready=trace_config)
```

after_train_epoch(*runner*)
Determine if the content is exported.

after_train_iter(*runner*, *batch_idx*, *data_batch*, *outputs*)
Update the content according to the schedule, and determine if the content is exported.

before_run(*runner*)
Initialize the profiler.

Through the runner parameter, the validity of the parameter is further determined.

41.13 PrepareTTAHook

class mmengine.hooks.**PrepareTTAHook**(*tta_cfg*)

Wraps *runner.model* with subclass of `BaseTTAModel` in *before_test*.

Note: This function will only be used with `MMFullyShardedDataParallel`.

Parameters **tta_cfg** (*dict*) – Config dictionary of the test time augmentation model.

before_test(*runner*)

Wraps *runner.model* with the subclass of `BaseTTAModel`.

Parameters **runner** (Runner) – The runner of the testing process.

Return type None

CHAPTER

FORTYTWO

MMENGINE.MODEL

mmengine.model

42.1 Module

BaseModule	Base module for all modules in openmmlab.
ModuleDict	ModuleDict in openmmlab.
ModuleList	ModuleList in openmmlab.
Sequential	Sequential module in openmmlab.

42.1.1 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`.

Parameters **init_cfg** (*dict, optional*) – Initialization config dict.

init_weights()

Initialize the weights.

42.1.2 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

Parameters

- **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.

42.1.3 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

Parameters

- **modules** (*iterable, optional*) – An iterable of modules to add.
- **init_cfg** (*dict, optional*) – Initialization config dict.

42.1.4 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

Parameters **init_cfg** (*dict, optional*) – Initialization config dict.

42.2 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.

42.2.1 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.

Examples

```
>>> @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))
>>>         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)
```

Parameters

- **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.

Returns The model itself.

Return type nn.Module

cuda(*device=None*)
Overrides this method to call *BaseDataPreprocessor.cuda()* additionally.

Returns The model itself.

Return type nn.Module

Parameters **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.

Parameters

- **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.

Returns

- 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.

Return type dict or list

npu(*device=None*)
Overrides this method to call `BaseDataPreprocessor.npu()` additionally.

Returns The model itself.

Return type nn.Module

Parameters **device** (*Optional[Union[int, str, torch.device]]*) –

Note: This generation of NPU(Ascend910) does not support the use of multiple cards in a single process, so the index here needs to be consistent with the default device

parse_losses(*losses*)
Parses the raw outputs (losses) of the network.

Parameters **losses** (*dict*) – Raw output of the network, which usually contain losses and other necessary information.

Returns 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.

Return type tuple[Tensor, dict]

test_step(*data*)
`BaseModel` implements `test_step` the same as `val_step`.

Parameters **data** (*dict or tuple or list*) – Data sampled from dataset.

Returns The predictions of given data.

Return type list

to(**args*, ***kwargs*)
Overrides this method to call *BaseDataPreprocessor.to()* additionally.

Returns The model itself.

Return type 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.

Parameters

- **data** (*dict or tuple or list*) – Data sampled from dataset.
- **optim_wrapper** (`OptimWrapper`) – OptimWrapper instance used to update model parameters.

Returns A `dict` of tensor for logging.

Return type 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.

Parameters **data** (*dict or tuple or list*) – Data sampled from dataset.

Returns The predictions of given data.

Return type list

42.2.2 BaseDataPreprocessor

class mmengine.model.**BaseDataPreprocessor**(*non_blocking=False*)

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.

Parameters **non_blocking** (*bool*) – Whether block current process when transferring data to device. New in version 0.3.0.

Note: 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.

Parameters **data** (*dict*) – Data returned by `DataLoader`.

Returns Inputs and data sample at target device.

Return type CollatedResult

cpu(**args*, ***kwargs*)

Overrides this method to set the `device`

Returns The model itself.

Return type nn.Module

cuda(**args*, ***kwargs*)

Overrides this method to set the `device`

Returns The model itself.

Return type 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.

Parameters

- **data** (*dict*) – Data returned by dataloader
- **training** (*bool*) – Whether to enable training time augmentation.

Returns Data in the same format as the model input.

Return type dict or list

to(**args*, ***kwargs*)

Overrides this method to set the `device`

Returns The model itself.

Return type nn.Module

42.2.3 ImgDataPreprocessor

class mmengine.model.**ImgDataPreprocessor**(*mean=None*, *std=None*, *pad_size_divisor=1*, *pad_value=0*, *bgr_to_rgb=False*, *rgb_to_bgr=False*, *non_blocking=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).

Note: `ImgDataPreprocessor` and its subclass is built in the constructor of `BaseDataset`.

Parameters

- **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.
- **non_blocking** (*bool*) – Whether block current process when transferring data to device. New in version v0.3.0.

Note: 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 normalizationpadding and bgr2rgb conversion based on `BaseDataPreprocessor`.

Parameters

- **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`.

Returns Data in the same format as the model input.

Return type dict or list

42.2.4 BaseTTAModel

class mmengine.model.**BaseTTAModel**(*module*, *data_preprocessor=None*)

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=[
        [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()*.

Note: *merge_preds()* is an abstract method, all subclasses should implement it.

Warning: If `data_preprocessor` is not None, it will overwrite the model's `data_preprocessor`.

Parameters

- **module** (*dict or nn.Module*) – Tested model.
- **data_preprocessor** (dict or *BaseDataPreprocessor*, optional) – If model does not define `data_preprocessor`, it will be the default value for model.

forward(*inputs*, *data_samples=None*, *mode='tensor'*)

`BaseTTAModel.forward` should not be called.

Parameters

- **inputs** (*torch.Tensor*) –
- **data_samples** (*Optional[list]*) –
- **mode** (*str*) –

Return type Union[Dict[str, torch.Tensor], list]

abstract merge_preds(*data_samples_list*)

Merge predictions of enhanced data to one prediction.

Parameters **data_samples_list** (*EnhancedBatchDataSamples*) – List of predictions of all enhanced data.

Returns Merged prediction.

Return type List[*BaseDataElement*]

test_step(*data*)

Get predictions of each enhanced data, a multiple predictions.

Parameters **data** (*DataBatch*) – Enhanced data batch sampled from dataloader.

Returns Merged prediction.

Return type MergedDataSamples

42.3 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.

42.3.1 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)
```

Parameters

- **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.

Return type 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.

Parameters

- **averaged_param** (*Tensor*) – The averaged parameters.
- **source_param** (*Tensor*) – The source parameters.
- **steps** (*int*) – The number of times the parameters have been updated.

Return type 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.

Parameters **model** (*nn.Module*) – The model whose parameters will be averaged.

Return type None

42.3.2 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:

$$Xema_{t+1} = (1 - momentum) * Xema_t + momentum * X_t$$

Note: This momentum argument is different from one used in optimizer classes and the conventional notion of momentum. Mathematically, $Xema_{t+1}$ is the moving average and X_t is the new observed value. The value of momentum is usually a small number, allowing observed values to slowly update the ema parameters.

Parameters

- **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.

Return type None

avg_func(*averaged_param*, *source_param*, *steps*)

Compute the moving average of the parameters using exponential moving average.

Parameters

- **averaged_param** (*Tensor*) – The averaged parameters.
- **source_param** (*Tensor*) – The source parameters.
- **steps** (*int*) – The number of times the parameters have been updated.

Return type None

42.3.3 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.

Parameters

- **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.

Return type None

avg_func(*averaged_param*, *source_param*, *steps*)

Compute the moving average of the parameters using the linear momentum strategy.

Parameters

- **averaged_param** (*Tensor*) – The averaged parameters.
- **source_param** (*Tensor*) – The source parameters.
- **steps** (*int*) – The number of times the parameters have been updated.

Return type None

42.3.4 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.

Parameters

- **model** (*torch.nn.modules.module.Module*) –
- **interval** (*int*) –
- **device** (*Optional[torch.device]*) –
- **update_buffers** (*bool*) –

Return type None

avg_func(*averaged_param*, *source_param*, *steps*)

Compute the average of the parameters using stochastic weight average.

Parameters

- **averaged_param** (*Tensor*) – The averaged parameters.
- **source_param** (*Tensor*) – The source parameters.
- **steps** (*int*) – The number of times the parameters have been updated.

Return type None

42.4 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.

42.4.1 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`.

Parameters

- **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`.

Note: If model has multiple submodules and each module has separate optimization strategies, *MMSeparateDistributedDataParallel* should be used to wrap the model.

Note: 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.

Parameters **data** (*dict or tuple or list*) – Data sampled from dataset.

Returns The predictions of given data.

Return type 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.

Parameters

- **data** (*dict or tuple or list*) – Data sampled from dataset.
- **optim_wrapper** (`OptimWrapper`) – A wrapper of optimizer to update parameters.

Returns A `dict` of tensor for logging.

Return type Dict[str, torch.Tensor]

val_step(*data*)
Gets the prediction of module during validation process.

Parameters **data** (*dict or tuple or list*) – Data sampled from dataset.

Returns The predictions of given data.

Return type list

42.4.2 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.

Parameters

- **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.

Parameters **data** (*dict or tuple or list*) – Data sampled from dataset.

Returns The predictions of given data.

Return type 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.

Parameters **mode** (*bool*) – whether to set training mode (True) or evaluation mode (False). Defaults to True.

Returns self.

Return type Module

train_step(*data*, *optim_wrapper*)
Interface for model forward, backward and parameters updating during training process.

Parameters

- **data** (*dict or tuple or list*) – Data sampled from dataset.
- **optim_wrapper** (OptimWrapperDict) – A wrapper of optimizer to update parameters.

Returns A dict of tensor for logging.

Return type Dict[str, torch.Tensor]

val_step(*data*)
Gets the prediction of module during validation process.

Parameters **data** (*dict or tuple or list*) – Data sampled from dataset.

Returns The predictions of given data.

Return type list

42.4.3 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`.

Parameters

- **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.

Parameters **`data`** (*dict*) – Data sampled by dataloader.

Returns The predictions of given data.

Return type 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.

Parameters

- **`data`** (*dict*) – Data sampled by dataloader.
- **`optim_wrapper`** (`OptimWrapper`) – A wrapper of optimizer to update parameters.

Returns A `dict` of tensor for logging.

Return type Dict[str, torch.Tensor]

`val_step`(*data*)
Gets the prediction of module during validation process.

Parameters **`data`** (*dict*) – Data sampled by dataloader.

Returns The predictions of given data.

Return type List[*BaseDataElement*] or dict

is_model_wrapper	Check if a module is a model wrapper.

42.4.4 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.MMSeparateDistributedDataParalle 'MMFullyShardedDataParallel': <class 'mmengine.model.wrappers.fully_sharded_distributed.MMFullyShardedDataParallel'*

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`.

Parameters

- **model** (*nn.Module*) – The model to be checked.
- **registry** (Registry) – The parent registry to search for model wrappers.

Returns True if the input model is a model wrapper.

Return type bool

42.5 Weight Initialization

BaseInit	
Caffe2XavierInit	
ConstantInit	Initialize module parameters with constant values.
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.

42.5.1 BaseInit

class mmengine.model.**BaseInit**(*, *bias=0*, *bias_prob=None*, *layer=None*)

42.5.2 Caffe2XavierInit

class mmengine.model.**Caffe2XavierInit**(***kwargs*)

42.5.3 ConstantInit

class mmengine.model.**ConstantInit**(*val*, ***kwargs*)

Initialize module parameters with constant values.

Parameters

- **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.

42.5.4 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).

Parameters

- **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.

42.5.5 NormalInit

class mmengine.model.**NormalInit**(*mean=0*, *std=1*, ***kwargs*)

Initialize module parameters with the values drawn from the normal distribution $\mathcal{N}(\text{mean}, \text{std}^2)$.

Parameters

- **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.

42.5.6 PretrainedInit

class mmengine.model.**PretrainedInit**(*checkpoint*, *prefix=None*, *map_location='cpu'*)

Initialize module by loading a pretrained model.

Parameters

- **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. Defaults to cpu.

42.5.7 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 $[a, b]$.

Parameters

- **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.

Return type None

42.5.8 UniformInit

class mmengine.model.**UniformInit**(*a=0*, *b=1*, ***kwargs*)

Initialize module parameters with values drawn from the uniform distribution $\mathcal{U}(a, b)$.

Parameters

- **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.

42.5.9 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).

Parameters

- **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	
uniform_init	
update_init_info	Update the *_params_init_info* in the module if the value of parameters are changed.
xavier_init	

42.5.10 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.

42.5.11 mmengine.model.caffe2_xavier_init

mmengine.model.**caffe2_xavier_init**(*module*, *bias=0*)

42.5.12 mmengine.model.constant_init

mmengine.model.**constant_init**(*module*, *val*, *bias=0*)

42.5.13 mmengine.model.initialize

mmengine.model.**initialize**(*module*, *init_cfg*)
Initialize a module.

Parameters

- **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.

Example

```
>>> 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)
```

(continues on next page)

(continued from previous page)

```
>>> # 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.')
```

42.5.14 mmengine.model.kaiming_init

mmengine.model.**kaiming_init**(*module*, *a=0*, *mode='fan_out'*, *nonlinearity='relu'*, *bias=0*, *distribution='normal'*)

42.5.15 mmengine.model.normal_init

mmengine.model.**normal_init**(*module*, *mean=0*, *std=1*, *bias=0*)

42.5.16 mmengine.model.trunc_normal_init

mmengine.model.**trunc_normal_init**(*module*, *mean=0*, *std=1*, *a=- 2*, *b=2*, *bias=0*)

Parameters

- **module** (*torch.nn.modules.module.Module*) –
- **mean** (*float*) –
- **std** (*float*) –
- **a** (*float*) –
- **b** (*float*) –
- **bias** (*float*) –

Return type None

42.5.17 mmengine.model.uniform_init

mmengine.model.**uniform_init**(*module*, *a=0*, *b=1*, *bias=0*)

42.5.18 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.

Parameters

- **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.

42.5.19 mmengine.model.xavier_init

mmengine.model.**xavier_init**(*module*, *gain=1*, *bias=0*, *distribution='normal'*)

42.6 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>_.

42.6.1 mmengine.model.detect_anomalous_params

mmengine.model.**detect_anomalous_params**(*loss*, *model*)

Parameters **loss** (*torch.Tensor*) –

Return type None

42.6.2 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`.

Note: 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`.

Parameters ***args** – dictionary needs to be merged.

Returns Merged dict from args

Return type dict

42.6.3 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`.

Parameters

- **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.

Returns The n dim tensor.

Return type Tensor

42.6.4 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)

Parameters **module** (*nn.Module*) – The module containing *SyncBatchNorm* layers.

Returns The converted module with *BatchNormXd* layers.

Return type module_output

42.6.5 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>_.

Parameters

- **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*.

Returns The converted module with *SyncBatchNorm* layers.

Return type nn.Module

CHAPTER

FORTYTHREE

MMENGINE.OPTIM

mmengine.optim

43.1 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.

43.1.1 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`.

> **Warning:** `AmpOptimWrapper` requires PyTorch >= 1.6.

Parameters

- **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.

Note: 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`.

Parameters

- **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.

Parameters **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.

Parameters **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".

Returns The merged state dict of `loss_scaler` and `optimizer`.

Return type dict

step(***kwargs*)
Update parameters with `loss_scaler`.

Parameters **kwargs** – Keyword arguments passed to `torch.optim.Optimizer.step()`.

43.1.2 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*.

Parameters

- **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_norm_()` or `torch.nn.utils.clip_grad_value_()`. `clip_grad` should be a dict, and the keys could be set as follows:

  If the key `type` is not set, or `type` is "norm", the accepted keys are as follows:

  - max_norm (float or int): Max norm of the gradients.
  - norm_type (float or int): Type of the used p-norm. Can be `'inf'` for infinity norm.
  - error_if_nonfinite (bool): If True, an error is thrown if the total norm of the gradients from `parameters` is `nan`, `inf`, or `-inf`. Default: False (will switch to True in the future)

  If the key `type` is set to "value", the accepted keys are as follows:

  - clip_value (float or int): maximum allowed value of the gradients. The gradients are clipped in the range `(-clip_value, +clip_value)`.

Note: If `accumulative_counts` is larger than 1, perform *update_params()* under the context of `optim_context` could avoid unnecessary gradient synchronization.

Note: If you use `IterBasedRunner` and enable gradient accumulation, the original *max_iters* should be multiplied by `accumulative_counts`.

Note: The subclass should ensure that once *update_params()* is called, `_inner_count += 1` is automatically performed.

Examples

```
>>> # Config sample of OptimWrapper and enable clipping gradient by
>>> # norm.
>>> optim_wrapper_cfg = dict(
>>>     type='OptimWrapper',
>>>     _accumulative_counts=1,
>>>     clip_grad=dict(max_norm=0.2))
>>> # Config sample of OptimWrapper and enable clipping gradient by
>>> # value.
>>> optim_wrapper_cfg = dict(
>>>     type='OptimWrapper',
>>>     _accumulative_counts=1,
>>>     clip_grad=dict(type='value', clip_value=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
>>>
```

(continues on next page)

(continued from previous page)

```
>>> 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.

Note: If subclasses inherit from `OptimWrapper` override `backward`, `_inner_count +=1` must be implemented.

Parameters

- **loss** (*torch.Tensor*) – The loss of current iteration.
- **kwargs** – Keyword arguments passed to `torch.Tensor.backward()`.

Return type None

property defaults: dict

A wrapper of `Optimizer.defaults`.

Make OptimizeWrapper compatible with *_ParamScheduler*.

Returns the `param_groups` of `optimizer`.

Return type dict

get_lr()

Get the learning rate of the optimizer.

Provide unified interface to get learning rate of optimizer.

Returns Learning rate of the optimizer.

Return type Dict[str, List[float]]

`get_momentum()`
Get the momentum of the optimizer.

Provide unified interface to get momentum of optimizer.

Returns Momentum of the optimizer.

Return type 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.

Parameters

- **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.

Return type 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`.

Parameters **state_dict** (*dict*) – The state dictionary of `optimizer`.

Return type 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.

Parameters **model** (*nn.Module*) – The training model.

`property param_groups: List[dict]`
A wrapper of `Optimizer.param_groups`.

Make OptimizeWrapper compatible with *_ParamScheduler*.

Returns the `param_groups` of `optimizer`.

Return type dict

`scale_loss`(*loss*)
Get scaled loss according to `_accumulative_counts`, `_inner_count` and max_counts.

Parameters **loss** (*torch.Tensor*) – Original loss calculated by model.

Returns Scaled loss.

Return type 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.

Returns Whether to block the automatic gradient synchronization.

Return type 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.

Returns Whether to update parameters.

Return type 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`.

Returns The state dictionary of `optimizer`.

Return type 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.

Parameters **kwargs** – Keyword arguments passed to `torch.optim.Optimizer.step()`.

Return type None

`update_params`(*loss*, *step_kwargs=None*, *zero_kwargs=None*)
Update parameters in `optimizer`.

Parameters

- **loss** (*torch.Tensor*) – A tensor for back propagation.

- **step_kwargs** (*dict*) – Arguments for optimizer.step. Defaults to None. New in version v0.4.0.
- **zero_kwargs** (*dict*) – Arguments for optimizer.zero_grad. Defaults to None. New in version v0.4.0.

Return type 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.

Parameters **kwargs** – Keyword arguments passed to `torch.optim.Optimizer.zero_grad()`.

Return type None

43.1.3 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.

Examples

```
>>> 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)
```

Note: The optimizer wrapper contained in `OptimWrapperDict` can be accessed in the same way as *dict*.

Parameters

- ****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.

Parameters **loss** (*torch.Tensor*) –

Return type None

get_lr()
Get the learning rate of all optimizers.

Returns Learning rate of all optimizers.

Return type Dict[str, List[float]]

get_momentum()
Get the momentum of all optimizers.

Returns momentum of all optimizers.

Return type 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.

Parameters **model** (*torch.nn.modules.module.Module*) –

Return type None

items()
A generator to get the name and corresponding *OptimWrapper*

Return type Iterator[Tuple[str, *mmengine.optim.optimizer.optimizer_wrapper.OptimWrapper*]]

keys()
A generator to get the name of *OptimWrapper*

Return type Iterator[str]

load_state_dict(*state_dict*)
Load the state dictionary from the `state_dict`.

Parameters **state_dict** (*dict*) – Each key-value pair in *state_dict* represents the name and the state dictionary of corresponding *OptimWrapper*.

Return type None

optim_context(*model*)
`optim_context` should be called by each optimizer separately.

Parameters **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.

Returns Each key-value pair in the dictionary represents the name and state dictionary of corresponding *OptimWrapper*.

Return type dict

step(***kwargs*)
Since the backward method is not implemented, the step should not be implemented either.

Return type None

update_params(*loss*, *step_kwargs=None*, *zero_kwargs=None*)
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.

Parameters

- **loss** (*torch.Tensor*) –
- **step_kwargs** (*Optional[Dict]*) –
- **zero_kwargs** (*Optional[Dict]*) –

Return type None

values()
A generator to get *OptimWrapper*

Return type Iterator[*mmengine.optim.optimizer.optimizer_wrapper.OptimWrapper*]

zero_grad(***kwargs*)
Set the gradients of all optimizer wrappers to zero.

Return type None

43.1.4 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.
- `flat_decay_mult` (float): It will be multiplied to the weight decay for all one-dimensional parameters
- `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.

Note: 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.

Parameters

- **optim_wrapper_cfg** (*dict*) – The config dict of the optimizer wrapper.

  Required fields of optim_wrapper_cfg are

  - type: class name of the OptimizerWrapper
  - optimizer: The configuration of optimizer.

  Optional fields of optim_wrapper_cfg are

  - any arguments of the corresponding optimizer wrapper type, e.g., accumulative_counts, clip_grad, etc.

  Required fields of optimizer are

  - *type*: class name of the optimizer.

  Optional fields of optimizer are

  - 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)
```

(continues on next page)

(continued from previous page)

```
>>> # 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.

Parameters

- **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.

Return type None

build_optim_wrapper	Build function of OptimWrapper.

43.1.5 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.

Parameters

- **model** (*nn.Module*) – Model to be optimized.
- **cfg** (*dict*) – Config of optimizer wrapper, optimizer constructor and optimizer.

Returns The built optimizer wrapper.

Return type *OptimWrapper*

43.2 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.

continues on next page

Table 3 – continued from previous page

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:
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.
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.

continues on next page

Table 3 – continued from previous page

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.

43.2.1 _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.

Parameters

- **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.

Returns A list of the last computed value of the optimizer's `param_group`.

Return type list

load_state_dict(*state_dict*)

Loads the schedulers state.

Parameters **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.

Parameters

- **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.

Returns scheduler state.

Return type dict

`step()`
Adjusts the parameter value of each parameter group based on the specified schedule.

43.2.2 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.

Parameters

- **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.

43.2.3 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.

Parameters

- **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.

43.2.4 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.

Parameters

- **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.

43.2.5 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.

Parameters

- **optimizer** (*Optimizer or* `OptimWrapper`) – Wrapped optimizer.
- **T_max** (*int*) – Maximum number of iterations.

- **eta_min** (*float*) – Minimum learning rate. Defaults to None.
- **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.
- **eta_min_ratio** (*float, optional*) – The ratio of the minimum parameter value to the base parameter value. Either *eta_min* or *eta_min_ratio* should be specified. Defaults to None. New in version 0.3.2.

43.2.6 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:

$$\eta_t = \eta_{min} + \frac{1}{2}(\eta_{max} - \eta_{min})\left(1 + \cos\left(\frac{T_{cur}}{T_{max}}\pi\right)\right), \quad T_{cur} \neq (2k+1)T_{max};$$
$$\eta_{t+1} = \eta_t + \frac{1}{2}(\eta_{max} - \eta_{min})\left(1 - \cos\left(\frac{1}{T_{max}}\pi\right)\right), \quad T_{cur} = (2k+1)T_{max}.$$

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.

Parameters

- **optimizer** (*Optimizer or* OptimWrapper) – optimizer or Wrapped optimizer.
- **T_max** (*int*) – Maximum number of iterations.
- **eta_min** (*float*) – Minimum momentum value. Defaults to None.
- **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.
- **eta_min_ratio** (*float, optional*) – The ratio of the minimum parameter value to the base parameter value. Either *eta_min* or *eta_min_ratio* should be specified. Defaults to None. New in version 0.3.2.

43.2.7 CosineAnnealingParamScheduler

class mmengine.optim.**CosineAnnealingParamScheduler**(*optimizer*, *param_name*, *T_max=None*, *eta_min=None*, *begin=0*, *end=1000000000*, *last_step=- 1*, *by_epoch=True*, *verbose=False*, *eta_min_ratio=None*)

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:

$$\eta_t = \eta_{min} + \frac{1}{2}(\eta_{max} - \eta_{min})\left(1 + \cos\left(\frac{T_{cur}}{T_{max}}\pi\right)\right), \quad T_{cur} \neq (2k+1)T_{max};$$
$$\eta_{t+1} = \eta_t + \frac{1}{2}(\eta_{max} - \eta_{min})\left(1 - \cos\left(\frac{1}{T_{max}}\pi\right)\right), \quad T_{cur} = (2k+1)T_{max}.$$

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.

Parameters

- **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, optional*) – Minimum parameter value. Defaults to None.
- **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.
- **eta_min_ratio** (*float, optional*) – The ratio of the minimum parameter value to the base parameter value. Either *eta_min* or *eta_min_ratio* should be specified. Defaults to None. New in version 0.3.2.

classmethod **build_iter_from_epoch**(**args*, *T_max=None*, *begin=0*, *end=1000000000*, *by_epoch=True*, *epoch_length=None*, ***kwargs*)

Build an iter-based instance of this scheduler from an epoch-based config.

43.2.8 ExponentialLR

class mmengine.optim.**ExponentialLR**(*optimizer*, **args*, ***kwargs*)

Decays the learning rate of each parameter group by gamma every epoch.

Parameters

- **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.

43.2.9 ExponentialMomentum

class mmengine.optim.**ExponentialMomentum**(*optimizer*, **args*, ***kwargs*)

Decays the momentum of each parameter group by gamma every epoch.

Parameters

- **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.

43.2.10 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.

Parameters

- **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.

43.2.11 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.

Parameters

- **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.

43.2.12 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.

Parameters

- **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.

43.2.13 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.

Parameters

- **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.

43.2.14 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.

Parameters

- **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.

43.2.15 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.

Parameters

- **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.

43.2.16 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.

Parameters

- **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.

43.2.17 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`.

Parameters

- **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.

43.2.18 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`.

Parameters

- **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.

43.2.19 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.

Parameters

- **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.

43.2.20 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.

Parameters

- **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.

43.2.21 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.

Parameters

- **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.

43.2.22 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.

Parameters

- **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.

43.2.23 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.

Parameters

- **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.

43.2.24 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.

Parameters

- **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

FORTYFOUR

MMENGINE.EVALUATOR

mmengine.evaluator

44.1 Evaluator

Evaluator	Wrapper class to compose multiple *BaseMetric* instances.

44.1.1 Evaluator

class mmengine.evaluator.**Evaluator**(*metrics*)

Wrapper class to compose multiple *BaseMetric* instances.

Parameters **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.

Parameters **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.

Returns Evaluation results of all metrics. The keys are the names of the metrics, and the values are corresponding results.

Return type dict

offline_evaluate(*data_samples*, *data=None*, *chunk_size=1*)

Offline evaluate the dumped predictions on the given data .

Parameters

- **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.

Parameters

- **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.

44.2 Metric

BaseMetric	Base class for a metric.
DumpResults	Dump model predictions to a pickle file for offline evaluation.

44.2.1 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.

Parameters

- **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

Return type None

abstract compute_metrics(*results*)
Compute the metrics from processed results.

Parameters **results** (*list*) – The processed results of each batch.

Returns The computed metrics. The keys are the names of the metrics, and the values are corresponding results.

Return type 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.

Parameters **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.

Returns Evaluation metrics dict on the val dataset. The keys are the names of the metrics, and the values are corresponding results.

Return type 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.

Parameters

- **data_batch** (*Any*) – A batch of data from the dataloader.
- **data_samples** (*Sequence[dict]*) – A batch of outputs from the model.

Return type None

44.2.2 DumpResults

class mmengine.evaluator.**DumpResults**(*out_file_path*, *collect_device='cpu'*)
Dump model predictions to a pickle file for offline evaluation.

Parameters

- **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'.

Return type None

compute_metrics(*results*)
dump the prediction results to a pickle file.

Parameters **results** (*list*) –

Return type dict

process(*data_batch*, *predictions*)
transfer tensors in predictions to CPU.

Parameters

- **data_batch** (*Any*) –
- **predictions** (*Sequence[dict]*) –

Return type None

44.3 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.

44.3.1 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.

Parameters

- **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

Returns The specified metric value

Return type Any

CHAPTER

FORTYFIVE

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.

45.1 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 `torch.Tensor` in the `data_fields`, such as `.cuda()`, `.cpu()`, `.numpy()`, `.to()`, `to_tensor()`, `.detach()`.

  Parameters

  - **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.

  Return type None

Examples

```
>>> import torch
>>> 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),
...     bboxes=bboxes, scores=scores)
>>> gt_instances = BaseDataElement(
...     metainfo=dict(img_id=img_id, img_shape=(640, 640)))
```

```
>>> # new
>>> gt_instances1 = gt_instances.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)
(100, 100)
>>> 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)
```

(continues on next page)

(continued from previous page)

```
tensor([0.5230, 0.7885, 0.2426, 0.3911, 0.4876])
>>> 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)
tensor([[0.0900, 0.0424, 0.1755, 0.4469],
        [0.8648, 0.0592, 0.3484, 0.0913],
        [0.5808, 0.1909, 0.6165, 0.7088],
        [0.5490, 0.4209, 0.9416, 0.2374],
        [0.3652, 0.1218, 0.8805, 0.7523]])
```

```
>>> # 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.set_metainfo(dict(img_shape=(1280, 1280)))
>>> gt_instances.img_shape  # (1280, 1280)
>>> gt_instances.bboxes = gt_instances.bboxes * 2
>>> gt_instances.get('img_shape', None)  # (1280, 1280)
>>> gt_instances.get('bboxes', None)  # 6x4 tensor
>>> 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_instances.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
```

(continues on next page)

(continued from previous page)

```
    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
...     @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()

Returns An iterator object whose element is (key, value) tuple pairs for `metainfo` and `data`.

Return type iterator

`all_keys()`

Returns Contains all keys in metainfo and data.

Return type list

`all_values()`

Returns Contains all values in metainfo and data.

Return type list

`clone()`
Deep copy the current data element.

Returns The copy of current data element.

Return type *BaseDataElement*

`cpu()`
Convert all tensors to CPU in data.

Return type *mmengine.structures.base_data_element.BaseDataElement*

`cuda()`
Convert all tensors to GPU in data.

Return type *mmengine.structures.base_data_element.BaseDataElement*

`detach()`
Detach all tensors in data.

Return type *mmengine.structures.base_data_element.BaseDataElement*

`get(key, default=None)`
Get property in data and metainfo as the same as python.

Return type Any

`items()`

Returns An iterator object whose element is (key, value) tuple pairs for `data`.

Return type iterator

`keys()`

Returns Contains all keys in data_fields.

Return type list

`property metainfo: dict`
A dict contains metainfo of current data element.

Type dict

`metainfo_items()`

Returns An iterator object whose element is (key, value) tuple pairs for `metainfo`.

Return type iterator

metainfo_keys()

Returns Contains all keys in metainfo_fields.

Return type list

metainfo_values()

Returns Contains all values in metainfo.

Return type 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.

Parameters

- **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.

Returns A new data element with same type.

Return type *BaseDataElement*

numpy()
Convert all tensors to np.ndarray in data.

Return type *mmengine.structures.base_data_element.BaseDataElement*

pop(**args*)
Pop property in data and metainfo as the same as python.

Return type Any

set_data(*data*)
Set or change key-value pairs in `data_field` by parameter `data`.

Parameters **data** (*dict*) – A dict contains annotations of image or model predictions.

Return type None

set_field(*value*, *name*, *dtype=None*, *field_type='data'*)
Special method for set union field, used as property.setter functions.

Parameters

- **value** (*Any*) –
- **name** (*str*) –
- **dtype** (*Optional[Union[Type, Tuple[Type, ...]]]*) –
- **field_type** (*str*) –

Return type None

set_metainfo(*metainfo*)
Set or change key-value pairs in `metainfo_field` by parameter `metainfo`.

Parameters **metainfo** (*dict*) – A dict contains the meta information of image, such as `img_shape`, `scale_factor`, etc.

Return type None

to(**args*, ***kwargs*)
Apply same name function to all tensors in data_fields.

Return type *mmengine.structures.base_data_element.BaseDataElement*

to_dict()
Convert BaseDataElement to dict.

Return type dict

to_tensor()
Convert all np.ndarray to tensor in data.

Return type *mmengine.structures.base_data_element.BaseDataElement*

update(*instance*)
The update() method updates the BaseDataElement with the elements from another BaseDataElement object.

Parameters **instance** (BaseDataElement) – Another BaseDataElement object for update the current object.

Return type None

values()

Returns Contains all values in data.

Return type list

45.2 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.

Examples

```
>>> # 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 isinstance(item, int):
...             if item >= len(self) or item < -len(self):  # type:ignore
...                 raise IndexError(f'Index {item} out of range!')
```

(continues on next page)

(continued from previous page)

```
...             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
>>> import torch
>>> img_meta = dict(img_shape=(800, 1196, 3), pad_shape=(800, 1216, 3))
>>> instance_data = InstanceData(metainfo=img_meta)
>>> '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
    img_shape: (800, 1196, 3)
    pad_shape: (800, 1216, 3)
    DATA FIELDS
    det_labels: tensor([2, 3])
    det_scores: tensor([0.8000, 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
    img_shape: (800, 1196, 3)
    pad_shape: (800, 1216, 3)
    DATA FIELDS
    det_labels: tensor([2])
    det_scores: tensor([0.8000])
    bboxes: tensor([[0.4997, 0.7707, 0.0595, 0.4188]])
```

(continues on next page)

(continued from previous page)

```
    polygons: [[1, 2, 3, 4]]
) at 0x7f64ecf0ec40>
>>> print(instance_data[instance_data.det_scores > 1])
<InstanceData(
    META INFORMATION
    img_shape: (800, 1196, 3)
    pad_shape: (800, 1216, 3)
    DATA FIELDS
    det_labels: tensor([], dtype=torch.int64)
    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])
    det_scores: tensor([0.8000, 0.7000, 0.8000, 0.7000])
    bboxes: tensor([[0.4997, 0.7707, 0.0595, 0.4188],
                [0.8101, 0.3105, 0.5123, 0.6263],
                [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], [1, 2, 3, 4], [5, 6, 7, 8]]
) at 0x7f203542feb0>
```

Parameters **metainfo** (*Optional[dict]*) –

Return type 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.

Parameters **instances_list** (list[*InstanceData*]) – A list of *InstanceData*.

Returns *InstanceData*

Return type *mmengine.structures.instance_data.InstanceData*

45.3 LabelData

class mmengine.structures.**LabelData**(*, *metainfo=None*, ***kwargs*)

Data structure for label-level annotations or predictions.

Parameters **metainfo** (*Optional[dict]*) –

Return type None

static **label_to_onehot**(*label*, *num_classes*)

Convert the label-format input to one-hot.

Parameters

- **label** (*torch.Tensor*) – The label-format input. The format of item must be label-format.
- **num_classes** (*int*) – The number of classes.

Returns The converted results.

Return type torch.Tensor

static **onehot_to_label**(*onehot*)

Convert the one-hot input to label.

Parameters **onehot** (*torch.Tensor, optional*) – The one-hot input. The format of input must be one-hot.

Returns The converted results.

Return type torch.Tensor

45.4 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.

Examples

```
>>> 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.shape)
(20, 40)
```

```
>>> # slice
>>> slice_data = pixel_data[10:20, 20:40]
>>> assert slice_data.shape == (10, 20)
>>> 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))
```

Parameters **metainfo** (*Optional[dict]*) –

Return type None

property shape

The shape of pixel data.

CHAPTER

FORTYSIX

MMENGINE.DATASET

mmengine.dataset

- *Dataset*
- *Dataset Wrapper*
- *Sampler*
- *Utils*

46.1 Dataset

BaseDataset	BaseDataset for open source projects in OpenMMLab.
Compose	Compose multiple transforms sequentially.

46.1.1 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":
```

(continues on next page)

(continued from previous page)

```
        [
          {
            "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]
          }
        ]
      },
    ]
}
```

Parameters

- **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=True`. 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.

Note: 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.

Note: 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.

Examples

```
>>> # 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.

Returns Filtered results.

Return type 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.

Parameters **idx** (*int*) – The index of data.

Returns All categories in the image of specified index.

Return type list[int]

get_data_info(*idx*)
Get annotation by index and automatically call `full_init` if the dataset has not been fully initialized.

Parameters **idx** (*int*) – The index of data.

Returns The idx-th annotation of the dataset.

Return type 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.

Examples

```
>>> 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
```

Parameters **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.

Returns A subset of dataset.

Return type *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.

Examples

```
>>> 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
```

Parameters **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.

Return type 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.

Returns A list of annotation.

Return type list[dict]

property metainfo: dict

Get meta information of dataset.

Returns meta information collected from `BaseDataset.METAINFO`, annotation file and metainfo argument during instantiation.

Return type 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.

Parameters **raw_data_info** (*dict*) – Raw data information load from `ann_file`

Returns Parsed annotation.

Return type list or list[dict]

prepare_data(*idx*)

Get data processed by `self.pipeline`.

Parameters **idx** (*int*) – The index of `data_info`.

Returns Depends on `self.pipeline`.

Return type Any

46.1.2 Compose

class mmengine.dataset.**Compose**(*transforms*)

Compose multiple transforms sequentially.

Parameters **transforms** (*Sequence[dict, callable], optional*) – Sequence of transform object or config dict to be composed.

46.2 Dataset Wrapper

ClassBalancedDataset	A wrapper of class balanced dataset.
ConcatDataset	A wrapper of concatenated dataset.
RepeatDataset	A wrapper of repeated dataset.

46.2.1 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: $r(I) = max_{cinI} r(c)$

Note: `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`.

Parameters

- **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.

Parameters **idx** (*int*) – Index of data.

Returns All categories in the image of specified index.

Return type List[int]

get_data_info(*idx*)
Get annotation by index.

Parameters **idx** (*int*) – Global index of ConcatDataset.

Returns The idx-th annotation of the dataset.

Return type dict

get_subset(*indices*)
Not supported in ClassBalancedDataset for the ambiguous meaning of sub-dataset.

Parameters **indices** (*Union[List[int], int]*) –

Return type *mmengine.dataset.base_dataset.BaseDataset*

get_subset_(*indices*)
Not supported in ClassBalancedDataset for the ambiguous meaning of sub-dataset.

Parameters **indices** (*Union[List[int], int]*) –

Return type None

property metainfo: dict
Get the meta information of the repeated dataset.

Returns The meta information of repeated dataset.

Return type dict

46.2.2 ConcatDataset

class mmengine.dataset.**ConcatDataset**(*datasets*, *lazy_init=False*, *ignore_keys=None*)
A wrapper of concatenated dataset.

Same as torch.utils.data.dataset.ConcatDataset and support lazy_init.

Note: 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.

Parameters

- **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.

- **ignore_keys** (*List[str] or str*) – Ignore the keys that can be unequal in *dataset.metainfo*. Defaults to None. *New in version 0.3.0.*

full_init()
Loop to `full_init` each dataset.

get_data_info(*idx*)
Get annotation by index.

Parameters **idx** (*int*) – Global index of `ConcatDataset`.

Returns The idx-th annotation of the datasets.

Return type dict

get_subset(*indices*)
Not supported in `ConcatDataset` for the ambiguous meaning of sub- dataset.

Parameters **indices** (*Union[List[int], int]*) –

Return type *mmengine.dataset.base_dataset.BaseDataset*

get_subset_(*indices*)
Not supported in `ConcatDataset` for the ambiguous meaning of sub- dataset.

Parameters **indices** (*Union[List[int], int]*) –

Return type None

property metainfo: dict
Get the meta information of the first dataset in `self.datasets`.

Returns Meta information of first dataset.

Return type dict

46.2.3 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.

Note: `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`.

Parameters

- **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.

Parameters **idx** (*int*) – Global index of ConcatDataset.

Returns The idx-th annotation of the datasets.

Return type dict

get_subset(*indices*)
Not supported in RepeatDataset for the ambiguous meaning of sub- dataset.

Parameters **indices** (*Union[List[int], int]*) –

Return type *mmengine.dataset.base_dataset.BaseDataset*

get_subset_(*indices*)
Not supported in RepeatDataset for the ambiguous meaning of sub- dataset.

Parameters **indices** (*Union[List[int], int]*) –

Return type None

property metainfo: dict
Get the meta information of the repeated dataset.

Returns The meta information of repeated dataset.

Return type dict

46.3 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.

46.3.1 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.

Parameters

- **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.

Parameters **epoch** (*int*) – Epoch number.

Return type None

46.3.2 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

Parameters

- **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.

Parameters **epoch** (*int*) –

Return type None

46.4 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`.
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.

46.4.1 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.

Note: `default_collate` only accept input tensor with the same shape.

Parameters **data_batch** (*Sequence*) – Data sampled from dataset.

Returns 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.

Return type Any

46.4.2 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.

Parameters **data_batch** (*Sequence*) – Batch of data from dataloader.

Returns Transversed Data in the same format as the data_itement of `data_batch`.

Return type Any

46.4.3 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.

Parameters

- **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.

Return type None

CHAPTER

FORTYSEVEN

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_npu_available	Returns True if Ascend PyTorch and npu devices exist.
is_mlu_available	Returns True if Cambricon PyTorch and mlu devices exist.
is_mps_available	Return True if mps devices exist.

47.1 mmengine.device.get_device

mmengine.device.**get_device**()

Returns the currently existing device type.

Returns cuda | npu | mlu | mps | cpu.

Return type str

47.2 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.

Parameters **device** (*torch.device, optional*) – selected device. Returns statistic for the current device, given by `current_device()`, if `device` is None. Defaults to None.

Returns The maximum GPU memory occupied by tensors in megabytes for a given device.

Return type int

47.3 mmengine.device.is_cuda_available

mmengine.device.**is_cuda_available**()
Returns True if cuda devices exist.

Return type bool

47.4 mmengine.device.is_npu_available

mmengine.device.**is_npu_available**()
Returns True if Ascend PyTorch and npu devices exist.

Return type bool

47.5 mmengine.device.is_mlu_available

mmengine.device.**is_mlu_available**()
Returns True if Cambricon PyTorch and mlu devices exist.

Return type bool

47.6 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.

Return type bool

CHAPTER

FORTYEIGHT

MMENGINE.HUB

get_config	Get config from external package.
get_model	Get built model from external package.

48.1 mmengine.hub.get_config

mmengine.hub.**get_config**(*cfg_path*, *pretrained=False*)

Get config from external package.

Parameters

- **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.

Return type *mmengine.config.config.Config*

Examples

```
>>> 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
```

Returns A *Config* parsed from external package.

Return type *Config*

Parameters

- **cfg_path** (*str*) –
- **pretrained** (*bool*) –

48.2 mmengine.hub.get_model

mmengine.hub.**get_model**(*cfg_path*, *pretrained=False*, ***kwargs*)

Get built model from external package.

Parameters

- **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.

Examples

```
>>> model = get_model('mmdet::faster_rcnn/faster-rcnn_r50_fpn_1x_coco.py',␣
↪pretrained=True)
>>> type(model)
<class 'mmdet.models.detectors.faster_rcnn.FasterRCNN'>
```

Returns Built model.

Return type nn.Module

Parameters

- **cfg_path** (*str*) –
- **pretrained** (*bool*) –

CHAPTER

FORTYNINE

MMENGINE.LOGGING

MMLogger	Formatted logger used to record messages.
MessageHub	Message hub for component interaction.
HistoryBuffer	Unified storage format for different log types.

49.1 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.

Note:

- 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`.

Examples

```
>>> 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.
```

(continues on next page)

(continued from previous page)

```
>>> 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)
```

Parameters

- **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.

Parameters **record** (*LogRecord*) – A `LogRecord` instance contains logged message.

Return type 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".

Returns Configured logger instance.

Return type *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.

49.2 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.

Parameters

- **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.

Note: Key in `_resumed_keys` belongs to `_log_scalars` or `_runtime_info`. The corresponding value cannot be set repeatedly.

Examples

```
>>> # 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".

Returns Empty `MessageHub` instance.

Return type *MessageHub*

get_info(*key*)

Get runtime information by key.

Parameters **key** (*str*) – Key of runtime information.

Returns A copy of corresponding runtime information if the key exists.

Return type Any

get_scalar(*key*)
Get `HistoryBuffer` instance by key.

Note: 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.

Parameters **key** (*str*) – Key of `HistoryBuffer`.

Returns Corresponding `HistoryBuffer` instance if the key exists.

Return type *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.

Parameters **state_dict** (*dict or MessageHub*) – A dictionary contains key `log_scalars` `runtime_info` and `resumed_keys`, or a MessageHub instance.

Return type None

property log_scalars: collections.OrderedDict
Get all `HistoryBuffer` instances.

Note: 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.

Returns All `HistoryBuffer` instances.

Return type OrderedDict

property runtime_info: collections.OrderedDict
Get all runtime information.

Returns A copy of all runtime information.

Return type 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()*.

Returns A dictionary contains `log_scalars`, `runtime_info` and `resumed_keys`.

Return type dict

update_info(*key*, *value*, *resumed=True*)
Update runtime information.

The key corresponding runtime information will be overwritten each time calling `update_info`.

Note: The `resumed` argument needs to be consistent for the same `key`.

Examples

```
>>> message_hub = MessageHub()
>>> message_hub.update_info('iter', 100)
```

Parameters

- **key** (*str*) – Key of runtime information.
- **value** (*Any*) – Value of runtime information.
- **resumed** (*bool*) – Whether the corresponding `HistoryBuffer` could be resumed.

Return type 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`.

Note: The `resumed` argument needs to be consistent for the same `info_dict`.

Examples

```
>>> message_hub = MessageHub()
>>> message_hub.update_info({'iter': 100})
```

Parameters

- **info_dict** (*str*) – Runtime information dictionary.
- **resumed** (*bool*) – Whether the corresponding `HistoryBuffer` could be resumed.

Return type 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`.

Examples

```
>>> 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'
```

Note: The `resumed` argument needs to be consistent for the same `key`.

Parameters

- **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.

Return type 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.

Note: The `resumed` argument needs to be consistent for the same `log_dict`.

Parameters

- **log_dict** (*str*) – Used for batch updating `_log_scalars`.
- **resumed** (*bool*) – Whether all `HistoryBuffer` referred in log_dict should be resumed. Defaults to True.

Return type None

Examples

```
>>> 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.
```

49.3 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.

Examples

```
>>> 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
```

Parameters

- **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.

Returns Recently updated values in log histories.

Return type np.ndarray

property data: Tuple[numpy.ndarray, numpy.ndarray]

Get the `_log_history` and `_count_history`.

Returns History logs and the counts of the history logs.

Return type 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.

Parameters **window_size** (*int, optional*) – Size of statistics window.

Returns The maximum value within the window.

Return type 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.

Parameters **window_size** (*int, optional*) – Size of statistics window.

Returns Mean value within the window.

Return type 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.

Parameters **window_size** (*int, optional*) – Size of statistics window.

Returns The minimum value within the window.

Return type 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.

Examples

```
>>> @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
```

Parameters **method** (*Callable*) – Custom statistics method.

Returns Original custom statistics method.

Return type Callable

statistics(*method_name*, **arg*, ***kwargs*)
Access statistics method by name.

Parameters **method_name** (*str*) – Name of method.

Returns Depends on corresponding method.

Return type 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.

Parameters

- **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*) –

Return type None

print_log	Print a log message.

49.4 mmengine.logging.print_log

mmengine.logging.**print_log**(*msg*, *logger=None*, *level=20*)
Print a log message.

Parameters

- **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.

Return type None

CHAPTER

FIFTY

MMENGINE.VISUALIZATION

mmengine.visualization

- *Visualizer*
- *visualization Backend*

50.1 Visualizer

Visualizer	MMEngine provides a Visualizer class that uses the `Matplotlib` library as the backend.

50.1.1 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.

Parameters

- **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.

Return type None

Examples

```
>>> # 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]]),
```

(continues on next page)

(continued from previous page)

```
>>>                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'])
>>> square = np.array([[0, 0], [100, 0], [100, 100], [0, 100]])
>>> vis.draw_polygons(polygons=square, edge_colors='g')
>>> squares = [np.array([[0, 0], [100, 0], [100, 100], [0, 100]]),
>>>            np.array([[0, 0], [50, 0], [50, 50], [0, 50]])]
>>> vis.draw_polygons(polygons=squares, edge_colors=['g', 'r'])
>>> vis.draw_binary_masks(binary_mask, alpha=0.6)
>>> heatmap = vis.draw_featmap(featmap, img,
>>>                            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:
```

(continues on next page)

(continued from previous page)

```
>>>         pass
```

add_config(*config*, ***kwargs*)
Record the config.

Parameters **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.

Parameters

- **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*) –

Return type None

add_graph(*model*, *data_batch*, ***kwargs*)
Record the model graph.

Parameters

- **model** (*torch.nn.Module*) – Model to draw.
- **data_batch** (*Sequence[dict]*) – Batch of data from dataloader.

Return type None

add_image(*name*, *image*, *step=0*)
Record the image.

Parameters

- **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.

Return type None

add_scalar(*name*, *value*, *step=0*, ***kwargs*)
Record the scalar data.

Parameters

- **name** (*str*) – The scalar identifier.
- **value** (*float, int*) – Value to save.
- **step** (*int*) – Global step value to record. Default to 0.

Return type None

add_scalars(*scalar_dict*, *step=0*, *file_path=None*, ***kwargs*)
Record the scalars' data.

Parameters

- **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.

Return type None

close()
close an opened object.

Return type 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.

Parameters

- **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. Defaults to None.
- **alpha** (*Union[int, float]*) – The transparency of bboxes. Defaults to 0.8.

Return type *mmengine.visualization.visualizer.Visualizer*

draw_binary_masks(*binary_masks*, *colors='g'*, *alphas=0.8*)
Draw single or multiple binary masks.

Parameters

- **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.

Return type *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.

Parameters

- **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.

Return type *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.

Parameters

- **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.

Returns RGB image.

Return type np.ndarray

draw_lines(*x_datas*, *y_datas*, *colors='g'*, *line_styles='-'*, *line_widths=2*)

Draw single or multiple line segments.

Parameters

- **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.

Return type *mmengine.visualization.visualizer.Visualizer*

draw_points(*positions*, *colors='g'*, *marker=None*, *sizes=None*)

Draw single or multiple points.

Parameters

- **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. Defaults to None.
- **sizes** (*Optional[Union[np.ndarray, torch.Tensor]]*) – The marker size. Defaults to None.

draw_polygons(*polygons*, *edge_colors='g'*, *line_styles='-'*, *line_widths=2*, *face_colors='none'*, *alpha=0.8*)
Draw single or multiple bboxes.

Parameters

- **polygons** (*Union[Union[np.ndarray, torch.Tensor], List[Union[np.ndarray, torch.Tensor]]]*) – The polygons to draw with the format of (x1,y1,x2,y2,...,xn,yn).
- **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. Defaults to None.
- **alpha** (*Union[int, float]*) – The transparency of polygons. Defaults to 0.8.

Return type *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.

Parameters

- **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.

Return type *mmengine.visualization.visualizer.Visualizer*

get_backend(*name*)
get vis backend by name.

Parameters **name** (*str*) – The name of vis backend

Returns The vis backend.

Return type *BaseVisBackend*

get_image()
Get the drawn image. The format is RGB.

Returns the drawn image which channel is RGB.

Return type 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.

Examples

```
>>> 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)
```

Parameters **name** (*str*) – Name of instance.

Returns Corresponding name instance.

Return type object

set_image(*image*)
Set the image to draw.

Parameters **image** (*np.ndarray*) – The image to draw.

Return type None

show(*drawn_img=None*, *win_name='image'*, *wait_time=0*, *continue_key=' '*)
Show the drawn image.

Parameters

- **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.

Return type None

50.2 visualization Backend

BaseVisBackend	Base class for visualization backend.
LocalVisBackend	Local visualization backend class.
TensorboardVisBackend	Tensorboard visualization backend class.
WandbVisBackend	Wandb visualization backend class.

50.2.1 BaseVisBackend

class mmengine.visualization.**BaseVisBackend**(*save_dir*)
Base class for visualization backend.

All backends must inherit `BaseVisBackend` and implement the required functions.

Parameters **save_dir** (*str, optional*) – The root directory to save the files produced by the backend.

add_config(*config*, ***kwargs*)
Record the config.

Parameters **config** (Config) – The Config object

Return type None

add_graph(*model*, *data_batch*, ***kwargs*)
Record the model graph.

Parameters

- **model** (*torch.nn.Module*) – Model to draw.
- **data_batch** (*Sequence[dict]*) – Batch of data from dataloader.

Return type None

add_image(*name*, *image*, *step=0*, ***kwargs*)
Record the image.

Parameters

- **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.

Return type None

add_scalar(*name*, *value*, *step=0*, ***kwargs*)
Record the scalar.

Parameters

- **name** (*str*) – The scalar identifier.
- **value** (*int, float*) – Value to save.
- **step** (*int*) – Global step value to record. Default to 0.

Return type None

add_scalars(*scalar_dict*, *step=0*, *file_path=None*, ***kwargs*)
Record the scalars' data.

Parameters

- **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.

Return type None

close()
close an opened object.

Return type 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.

50.2.2 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.

Examples

```
>>> 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)
```

Parameters

- **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.

Parameters **config** (Config) – The Config object

Return type None

add_image(*name*, *image*, *step=0*, ***kwargs*)
Record the image to disk.

Parameters

- **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.

Return type None

add_scalar(*name*, *value*, *step=0*, ***kwargs*)
Record the scalar data to disk.

Parameters

- **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.

Return type 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.

Parameters

- **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.

Return type None

property experiment: mmengine.visualization.vis_backend.LocalVisBackend
Return the experiment object associated with this visualization backend.

50.2.3 TensorboardVisBackend

class mmengine.visualization.**TensorboardVisBackend**(*save_dir*)
Tensorboard visualization backend class.

It can write images, config, scalars, etc. to a tensorboard file.

Examples

```
>>> from mmengine.visualization import TensorboardVisBackend
>>> import numpy as np
>>> vis_backend = TensorboardVisBackend(save_dir='temp_dir')
>>> img = np.random.randint(0, 256, size=(10, 10, 3))
>>> vis_backend.add_image('img', img)
>>> vis_backend.add_scaler('mAP', 0.6)
>>> vis_backend.add_scalars({'loss': 0.1,'acc':0.8})
>>> cfg = Config(dict(a=1, b=dict(b1=[0, 1])))
>>> vis_backend.add_config(cfg)
```

Parameters **save_dir** (*str*) – The root directory to save the files produced by the backend.

add_config(*config*, ***kwargs*)
Record the config to tensorboard.

Parameters **config** (Config) – The Config object

Return type None

add_image(*name*, *image*, *step=0*, ***kwargs*)
Record the image to tensorboard.

Parameters

- **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.

Return type None

add_scalar(*name*, *value*, *step=0*, ***kwargs*)
Record the scalar data to tensorboard.

Parameters

- **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.

Return type None

add_scalars(*scalar_dict*, *step=0*, *file_path=None*, ***kwargs*)
Record the scalar's data to tensorboard.

Parameters

- **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.

Return type None

close()
close an opened tensorboard object.

property experiment
Return Tensorboard object.

50.2.4 WandbVisBackend

class mmengine.visualization.**WandbVisBackend**(*save_dir*, *init_kwargs=None*, *define_metric_cfg=None*, *commit=True*, *log_code_name=None*, *watch_kwargs=None*)
Wandb visualization backend class.

Examples

```
>>> 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)
```

Parameters

- **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.
- **log_code_name** (*Optional[str]*) – (str, optional) The name of code artifact. By default, the artifact will be named source-$PROJECT_ID-$ENTRYPOINT_RELPATH. See wandb docs for details. Defaults to None. New in version 0.3.0.
- **watch_kwargs** (*optional, dict*) – Agurments for `wandb.watch`. New in version 0.4.0.

add_config(*config*, ***kwargs*)
Record the config to wandb.

Parameters **config** (Config) – The Config object

Return type None

add_graph(*model*, *data_batch*, ***kwargs*)
Record the model graph.

Parameters

- **model** (*torch.nn.Module*) – Model to draw.
- **data_batch** (*Sequence[dict]*) – Batch of data from dataloader.

Return type None

add_image(*name*, *image*, *step=0*, ***kwargs*)
Record the image to wandb.

Parameters

- **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.

Return type None

add_scalar(*name*, *value*, *step=0*, ***kwargs*)
Record the scalar data to wandb.

Parameters

- **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.

Return type None

add_scalars(*scalar_dict*, *step=0*, *file_path=None*, ***kwargs*)
Record the scalar's data to wandb.

Parameters

- **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.

Return type None

close()
close an opened wandb object.

Return type 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

FIFTYONE

MMENGINE.FILEIO

mmengine.fileio

- *File Backend*
- *File Handler*
- *File IO*
- *Parse File*

51.1 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).

51.1.1 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.

51.1.2 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.

Warning: *FileClient* will be deprecated in future. Please use io functions in https://mmengine.readthedocs.io/en/latest/api/fileio.html#file-io

Parameters

- **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.

Examples

```
>>> # 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
>>> 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.

Parameters **filepath** (*str or Path*) – Path to be checked whether exists.

Returns Return `True` if `filepath` exists, `False` otherwise.

Return type bool

get(*filepath*)

Read data from a given `filepath` with 'rb' mode.

Note: 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()`.

Parameters **filepath** (*str or Path*) – Path to read data.

Returns Expected bytes object or a memory view of the bytes object.

Return type 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.

Note: If the `filepath` is a local path, just return itself.

Warning: `get_local_path` is an experimental interface that may change in the future.

Parameters **filepath** (*str or Path*) – Path to be read data.

Return type Generator[Union[str, pathlib.Path], None, None]

Examples

```
>>> file_client = FileClient(prefix='s3')
>>> with file_client.get_local_path('s3://bucket/abc.jpg') as path:
...     # do something here
```

Yields *Iterable[str]* – Only yield one path.

Parameters **filepath** (*Union[str, pathlib.Path]*) –

Return type Generator[Union[str, pathlib.Path], None, None]

get_text(*filepath*, *encoding='utf-8'*)

Read data from a given `filepath` with 'r' mode.

Parameters

- **filepath** (*str or Path*) – Path to read data.
- **encoding** (*str*) – The encoding format used to open the `filepath`. Default: 'utf-8'.

Returns Expected text reading from `filepath`.

Return type str

classmethod infer_client(*file_client_args=None*, *uri=None*)

Infer a suitable file client based on the URI and arguments.

Parameters

- **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.

Return type *mmengine.fileio.file_client.FileClient*

Examples

```
>>> 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)
```

Returns Instantiated FileClient object.

Return type *FileClient*

Parameters

- **file_client_args** (*Optional[dict]*) –
- **uri** (*Optional[Union[str, pathlib.Path]]*) –

isdir(*filepath*)
Check whether a file path is a directory.

Parameters **filepath** (*str or Path*) – Path to be checked whether it is a directory.

Returns Return `True` if `filepath` points to a directory, `False` otherwise.

Return type bool

isfile(*filepath*)
Check whether a file path is a file.

Parameters **filepath** (*str or Path*) – Path to be checked whether it is a file.

Returns Return `True` if `filepath` points to a file, `False` otherwise.

Return type 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.

Parameters

- **filepath** (*str or Path*) – Path to be concatenated.
- **filepaths** (*Union[str, pathlib.Path]*) –

Returns The result of concatenation.

Return type 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.

Note: *list_dir_or_file()* returns the path relative to `dir_path`.

Parameters

- **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.

Yields *Iterable[str]* – A relative path to `dir_path`.

Return type Iterator[str]

static parse_uri_prefix(*uri*)

Parse the prefix of a uri.

Parameters **uri** (*str | Path*) – Uri to be parsed that contains the file prefix.

Return type Optional[str]

Examples

```
>>> FileClient.parse_uri_prefix('s3://path/of/your/file')
's3'
```

Returns Return the prefix of uri if the uri contains '://' else `None`.

Return type str | None

Parameters **uri** (*Union[str, pathlib.Path]*) –

put(*obj*, *filepath*)

Write data to a given `filepath` with 'wb' mode.

Note: `put` should create a directory if the directory of `filepath` does not exist.

Parameters

- **obj** (*bytes*) – Data to be written.
- **filepath** (*str or Path*) – Path to write data.

Return type None

put_text(*obj*, *filepath*)

Write data to a given `filepath` with 'w' mode.

Note: `put_text` should create a directory if the directory of `filepath` does not exist.

Parameters

- **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'.

Return type 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
```

Parameters

- **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.

Parameters **filepath** (*str, Path*) – Path to be removed.

Return type None

51.1.3 HardDiskBackend

class mmengine.fileio.**HardDiskBackend**
Raw hard disks storage backend.

Return type None

51.1.4 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.

Parameters

- **src** (*str or Path*) – Create a symbolic link pointing to src.
- **dst** (*str or Path*) – Create a symbolic link named dst.

Returns Return True if successfully create a symbolic link pointing to src. Otherwise, return False.

Return type bool

Examples

```
>>> 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.

Parameters

- **src** (*str or Path*) – A file to be copied.
- **dst** (*str or Path*) – Copy file to dst.

Returns The destination file.

Return type str

Raises **SameFileError** – If src and dst are the same file, a SameFileError will be raised.

Examples

```
>>> 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()*.

Parameters

- **src** (*str or Path*) – A local file to be copied.
- **dst** (*str or Path*) – Copy file to dst.

Returns If dst specifies a directory, the file will be copied into dst using the base filename from src.

Return type str

Raises **SameFileError** – If src and dst are the same file, a SameFileError will be raised.

Examples

```
>>> 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.

Parameters

- **src** (*str or Path*) – A file to be copied.

- **dst** (*str or Path*) – Copy file to to local dst.

Returns If dst specifies a directory, the file will be copied into dst using the base filename from src.

Return type str

Examples

```
>>> 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
>>> 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.

Parameters

- **src** (*str or Path*) – A directory to be copied.
- **dst** (*str or Path*) – Copy directory to dst.

Returns The destination directory.

Return type str

Raises **FileExistsError** – If dst had already existed, a FileExistsError will be raised.

Examples

```
>>> 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()*.

Parameters

- **src** (*str or Path*) – A local directory to be copied.

- **dst** (*str or Path*) – Copy directory to dst.

Returns The destination directory.

Return type str

Examples

```
>>> 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.

Parameters

- **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.

Returns The destination directory.

Return type str

Examples

```
>>> 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.

Parameters **filepath** (*str or Path*) – Path to be checked whether exists.

Returns Return `True` if `filepath` exists, `False` otherwise.

Return type bool

Examples

```
>>> backend = LocalBackend()
>>> filepath = '/path/of/file'
>>> backend.exists(filepath)
True
```

get(*filepath*)

Read bytes from a given `filepath` with 'rb' mode.

Parameters **filepath** (*str or Path*) – Path to read data.

Returns Expected bytes object.

Return type bytes

Examples

```
>>> backend = LocalBackend()
>>> filepath = '/path/of/file'
>>> backend.get(filepath)
b'hello world'
```

get_local_path(*filepath*)

Only for unified API and do nothing.

Parameters

- **filepath** (*str or Path*) – Path to be read data.
- **backend_args** (*dict, optional*) – Arguments to instantiate the corresponding backend. Defaults to None.

Return type Generator[Union[str, pathlib.Path], None, None]

Examples

```
>>> 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.

Parameters

- **filepath** (*str or Path*) – Path to read data.
- **encoding** (*str*) – The encoding format used to open the `filepath`. Defaults to 'utf-8'.

Returns Expected text reading from `filepath`.

Return type str

Examples

```
>>> backend = LocalBackend()
>>> filepath = '/path/of/file'
>>> backend.get_text(filepath)
'hello world'
```

isdir(*filepath*)

Check whether a file path is a directory.

Parameters **filepath** (*str or Path*) – Path to be checked whether it is a directory.

Returns Return `True` if `filepath` points to a directory, `False` otherwise.

Return type bool

Examples

```
>>> backend = LocalBackend()
>>> filepath = '/path/of/dir'
>>> backend.isdir(filepath)
True
```

isfile(*filepath*)

Check whether a file path is a file.

Parameters **filepath** (*str or Path*) – Path to be checked whether it is a file.

Returns Return `True` if `filepath` points to a file, `False` otherwise.

Return type bool

Examples

```
>>> 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.

Parameters

- **filepath** (*str or Path*) – Path to be concatenated.
- **filepaths** (*Union[str, pathlib.Path]*) –

Returns The result of concatenation.

Return type str

Examples

```
>>> 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.

Note: *list_dir_or_file()* returns the path relative to `dir_path`.

Parameters

- **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.

Yields *Iterable[str]* – A relative path to `dir_path`.

Return type Iterator[str]

Examples

```
>>> 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.

Note: `put` will create a directory if the directory of `filepath` does not exist.

Parameters

- **obj** (*bytes*) – Data to be written.
- **filepath** (*str or Path*) – Path to write data.

Return type None

Examples

```
>>> 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.

Note: `put_text` will create a directory if the directory of `filepath` does not exist.

Parameters

- **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'.

Return type None

Examples

```
>>> backend = LocalBackend()
>>> filepath = '/path/of/file'
>>> backend.put_text('hello world', filepath)
```

remove(*filepath*)

Remove a file.

Parameters **filepath** (*str or Path*) – Path to be removed.

Raises

- **IsADirectoryError** – If filepath is a directory, an IsADirectoryError will be raised.
- **FileNotFoundError** – If filepath does not exist, an FileNotFoundError will be raised.

Return type None

Examples

```
>>> backend = LocalBackend()
>>> filepath = '/path/of/file'
>>> backend.remove(filepath)
```

rmtree(*dir_path*)
Recursively delete a directory tree.

Parameters **dir_path** (*str or Path*) – A directory to be removed.

Return type None

Examples

```
>>> dir_path = '/path/of/dir'
>>> backend.rmtree(dir_path)
```

51.1.5 HTTPBackend

class mmengine.fileio.**HTTPBackend**
HTTP and HTTPS storage bachend.

get(*filepath*)
Read bytes from a given `filepath`.

Parameters **filepath** (*str*) – Path to read data.

Returns Expected bytes object.

Return type bytes

Examples

```
>>> 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.

Parameters **filepath** (*str*) – Download a file from `filepath`.

Yields *Iterable[str]* – Only yield one temporary path.

Return type Generator[Union[str, pathlib.Path], None, None]

Examples

```
>>> 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`.

Parameters

- **filepath** (*str*) – Path to read data.
- **encoding** (*str*) – The encoding format used to open the `filepath`. Defaults to 'utf-8'.

Returns Expected text reading from `filepath`.

Return type str

Examples

```
>>> backend = HTTPBackend()
>>> backend.get_text('http://path/of/file')
'hello world'
```

51.1.6 LmdbBackend

class mmengine.fileio.**LmdbBackend**(*db_path*, *readonly=True*, *lock=False*, *readahead=False*, ***kwargs*)
Lmdb storage backend.

Parameters

- **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.

Parameters **filepath** (*str or Path*) – Here, filepath is the lmdb key.

Returns Expected bytes object.

Return type bytes

Examples

```
>>> backend = LmdbBackend('path/to/lmdb')
>>> backend.get('key')
b'hello world'
```

51.1.7 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.

Parameters **filepath** (*str or Path*) – Path to read data.

Returns Expected bytes object.

Return type bytes

Examples

```
>>> 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'
```

51.1.8 PetrelBackend

class mmengine.fileio.**PetrelBackend**(*path_mapping=None*, *enable_mc=True*, *conf_path=None*)

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.

Parameters

- **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.
- **conf_path** (*str, optional*) – Config path of Petrel client. Default: None. *New in version 0.3.3*.

Examples

```
>>> 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.

Parameters

- **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.

Returns Return False because PetrelBackend does not support create a symbolic link.

Return type bool

Examples

```
>>> 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.

Parameters

- **src** (*str or Path*) – A file to be copied.
- **dst** (*str or Path*) – Copy file to dst.

Returns The destination file.

Return type str

Raises **SameFileError** – If src and dst are the same file, a SameFileError will be raised.

Examples

```
>>> 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.

Parameters

- **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.

Returns If dst specifies a directory, the file will be copied into dst using the base filename from src.

Return type str

Examples

```
>>> 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.

Parameters

- **src** (*str or Path*) – A file to be copied.
- **dst** (*str or Path*) – Copy file to to local dst.

Returns If dst specifies a directory, the file will be copied into dst using the base filename from src.

Return type str

Examples

```
>>> 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.

Parameters

- **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.

Returns The destination directory.

Return type str

Raises **FileExistsError** – If dst had already existed, a FileExistsError will be raised.

Examples

```
>>> 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.

Parameters

- **src** (*str or Path*) – A local directory to be copied.
- **dst** (*str or Path*) – Copy directory to dst.

Returns The destination directory.

Return type str

Raises **FileExistsError** – If dst had already existed, a FileExistsError will be raised.

Examples

```
>>> 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.

Parameters

- **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.

Returns The destination directory.

Return type str

Examples

```
>>> 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.

Parameters **filepath** (*str or Path*) – Path to be checked whether exists.

Returns Return `True` if `filepath` exists, `False` otherwise.

Return type bool

Examples

```
>>> 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.

Note: Now only work on Petrel backend.

Parameters

- **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.

Returns Generated presigned url.

Return type str

get(*filepath*)

Read bytes from a given `filepath` with 'rb' mode.

Parameters **filepath** (*str or Path*) – Path to read data.

Returns Return bytes read from filepath.

Return type bytes

Examples

```
>>> 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.

Parameters **filepath** (*str or Path*) – Download a file from `filepath`.

Yields *Iterable[str]* – Only yield one temporary path.

Return type Generator[Union[str, pathlib.Path], None, None]

Examples

```
>>> 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.

Parameters

- **filepath** (*str or Path*) – Path to read data.
- **encoding** (*str*) – The encoding format used to open the `filepath`. Defaults to 'utf-8'.

Returns Expected text reading from `filepath`.

Return type str

Examples

```
>>> backend = PetrelBackend()
>>> filepath = 'petrel://path/of/file'
>>> backend.get_text(filepath)
'hello world'
```

isdir(*filepath*)

Check whether a file path is a directory.

Parameters **filepath** (*str or Path*) – Path to be checked whether it is a directory.

Returns Return `True` if `filepath` points to a directory, `False` otherwise.

Return type bool

Examples

```
>>> backend = PetrelBackend()
>>> filepath = 'petrel://path/of/dir'
>>> backend.isdir(filepath)
True
```

isfile(*filepath*)

Check whether a file path is a file.

Parameters **filepath** (*str or Path*) – Path to be checked whether it is a file.

Returns Return `True` if `filepath` points to a file, `False` otherwise.

Return type bool

Examples

```
>>> 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.

Parameters

- **filepath** (*str or Path*) – Path to be concatenated.
- **filepaths** (*Union[str, pathlib.Path]*) –

Returns The result after concatenation.

Return type str

Examples

```
>>> 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.

Note: 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.

Note: *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.

Parameters

- **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.

Yields *Iterable[str]* – A relative path to `dir_path`.

Return type Iterator[str]

Examples

```
>>> 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`.

Parameters

- **obj** (*bytes*) – Data to be saved.
- **filepath** (*str or Path*) – Path to write data.

Return type None

Examples

```
>>> 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`.

Parameters

- **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'.

Return type None

Examples

```
>>> backend = PetrelBackend()
>>> filepath = 'petrel://path/of/file'
>>> backend.put_text('hello world', filepath)
```

remove(*filepath*)
Remove a file.

Parameters **filepath** (*str or Path*) – Path to be removed.

Raises

- **FileNotFoundError** – If filepath does not exist, an FileNotFoundError will be raised.
- **IsADirectoryError** – If filepath is a directory, an IsADirectoryError will be raised.

Return type None

Examples

```
>>> backend = PetrelBackend()
>>> filepath = 'petrel://path/of/file'
>>> backend.remove(filepath)
```

rmtree(*dir_path*)
Recursively delete a directory tree.

Parameters **dir_path** (*str or Path*) – A directory to be removed.

Return type None

Examples

```
>>> backend = PetrelBackend()
>>> dir_path = 'petrel://path/of/dir'
>>> backend.rmtree(dir_path)
```

register_backend	Register a backend.

51.1.9 mmengine.fileio.register_backend

mmengine.fileio.**register_backend**(*name*, *backend=None*, *force=False*, *prefixes=None*)

Register a backend.

Parameters

- **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.

Examples

```
>>> 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
```

51.2 File Handler

BaseFileHandler
JsonHandler
PickleHandler
YamlHandler

51.2.1 BaseFileHandler

class mmengine.fileio.**BaseFileHandler**

51.2.2 JsonHandler

class mmengine.fileio.**JsonHandler**

51.2.3 PickleHandler

class mmengine.fileio.**PickleHandler**

51.2.4 YamlHandler

class mmengine.fileio.**YamlHandler**

register_handler

51.2.5 mmengine.fileio.register_handler

mmengine.fileio.**register_handler**(*file_formats*, ***kwargs*)

51.3 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.

continues on next page

Table 5 – continued from previous page

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.

51.3.1 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.

Parameters

- **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.

Examples

```
>>> dump('hello world', '/path/of/your/file')  # disk
>>> dump('hello world', 's3://path/of/your/file')  # ceph or petrel
```

Returns True for success, False otherwise.

Return type bool

51.3.2 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.

Parameters

- **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.

Examples

```
>>> 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
```

Returns The content from the file.

51.3.3 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.

Parameters

- **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.

Returns Return True if successfully create a symbolic link pointing to src. Otherwise, return False.

Return type bool

Examples

```
>>> 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
```

51.3.4 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.

Parameters

- **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.

Returns The destination file.

Return type str

Raises **SameFileError** – If src and dst are the same file, a SameFileError will be raised.

Examples

```
>>> # 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'
>>> # src will be copied to '/path1/of/dir/file'
>>> copyfile(src, dst)
'/path1/of/dir/file'
```

51.3.5 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.

Note: If the backend is the instance of LocalBackend, it does the same thing with *copyfile()*.

Parameters

- **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.

Returns If dst specifies a directory, the file will be copied into dst using the base filename from src.

Return type str

Examples

```
>>> # 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'
```

51.3.6 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.

Note: If the backend is the instance of LocalBackend, it does the same thing with *copyfile()*.

Parameters

- **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.

Returns If dst specifies a directory, the file will be copied into dst using the base filename from src.

Return type str

Examples

```
>>> # 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'
```

51.3.7 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.

Parameters

- **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.

Returns The destination directory.

Return type str

Raises **FileExistsError** – If dst had already existed, a FileExistsError will be raised.

Examples

```
>>> src = '/path/of/dir1'
>>> dst = '/path/of/dir2'
>>> copytree(src, dst)
'/path/of/dir2'
```

51.3.8 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.

Note: If the backend is the instance of LocalBackend, it does the same thing with *copytree()*.

Parameters

- **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.

Returns The destination directory.

Return type str

Examples

```
>>> src = '/path/of/dir'
>>> dst = 's3://openmmlab/mmengine/dir'
>>> copyfile_from_local(src, dst)
's3://openmmlab/mmengine/dir'
```

51.3.9 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.

Note: If the backend is the instance of LocalBackend, it does the same thing with *copytree()*.

Parameters

- **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.

Returns The destination directory.

Return type str

Examples

```
>>> src = 's3://openmmlab/mmengine/dir'
>>> dst = '/path/of/dir'
>>> copytree_to_local(src, dst)
'/path/of/dir'
```

51.3.10 mmengine.fileio.exists

mmengine.fileio.**exists**(*filepath*, *backend_args=None*)

Check whether a file path exists.

Parameters

- **filepath** (*str or Path*) – Path to be checked whether exists.
- **backend_args** (*dict, optional*) – Arguments to instantiate the corresponding backend. Defaults to None.

Returns Return `True` if `filepath` exists, `False` otherwise.

Return type bool

Examples

```
>>> filepath = '/path/of/file'
>>> exists(filepath)
True
```

51.3.11 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.

Note: Now only work on Petrel backend.

Parameters

- **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.

Returns Generated presigned url.

Return type str

51.3.12 mmengine.fileio.get

mmengine.fileio.**get**(*filepath*, *backend_args=None*)

Read bytes from a given `filepath` with 'rb' mode.

Parameters

- **filepath** (*str or Path*) – Path to read data.
- **backend_args** (*dict, optional*) – Arguments to instantiate the corresponding backend. Defaults to None.

Returns Expected bytes object.

Return type bytes

Examples

```
>>> filepath = '/path/of/file'
>>> get(filepath)
b'hello world'
```

51.3.13 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.

Parameters

- **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.

Returns Instantiated Backend object.

Return type *BaseStorageBackend*

Examples

```
>>> # 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'})
```

51.3.14 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.

Note: If the `filepath` is a local path, just return itself and it will not be released (removed).

Parameters

- **filepath** (*str or Path*) – Path to be read data.
- **backend_args** (*dict, optional*) – Arguments to instantiate the corresponding backend. Defaults to None.

Yields *Iterable[str]* – Only yield one path.

Return type Generator[Union[str, pathlib.Path], None, None]

Examples

```
>>> with get_local_path('s3://bucket/abc.jpg') as path:
...     # do something here
```

51.3.15 mmengine.fileio.get_text

mmengine.fileio.**get_text**(*filepath*, *encoding='utf-8'*, *backend_args=None*)

Read text from a given `filepath` with 'r' mode.

Parameters

- **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.

Returns Expected text reading from `filepath`.

Return type str

Examples

```
>>> filepath = '/path/of/file'
>>> get_text(filepath)
'hello world'
```

51.3.16 mmengine.fileio.isdir

mmengine.fileio.**isdir**(*filepath*, *backend_args=None*)

Check whether a file path is a directory.

Parameters

- **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.

Returns Return `True` if `filepath` points to a directory, `False` otherwise.

Return type bool

Examples

```
>>> filepath = '/path/of/dir'
>>> isdir(filepath)
True
```

51.3.17 mmengine.fileio.isfile

mmengine.fileio.**isfile**(*filepath*, *backend_args=None*)

Check whether a file path is a file.

Parameters

- **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.

Returns Return `True` if `filepath` points to a file, `False` otherwise.

Return type bool

Examples

```
>>> filepath = '/path/of/file'
>>> isfile(filepath)
True
```

51.3.18 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.

Parameters

- **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]*) –

Returns The result of concatenation.

Return type str

Examples

```
>>> filepath1 = '/path/of/dir1'
>>> filepath2 = 'dir2'
>>> filepath3 = 'path/of/file'
>>> join_path(filepath1, filepath2, filepath3)
'/path/of/dir/dir2/path/of/file'
```

51.3.19 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.

Note: *list_dir_or_file()* returns the path relative to `dir_path`.

Parameters

- **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.

Yields *Iterable[str]* – A relative path to `dir_path`.

Return type Iterator[str]

Examples

```
>>> 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)
```

51.3.20 mmengine.fileio.put

mmengine.fileio.**put**(*obj*, *filepath*, *backend_args=None*)

Write bytes to a given `filepath` with 'wb' mode.

Note: `put` should create a directory if the directory of `filepath` does not exist.

Parameters

- **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.

Return type None

Examples

```
>>> filepath = '/path/of/file'
>>> put(b'hello world', filepath)
```

51.3.21 mmengine.fileio.put_text

mmengine.fileio.**put_text**(*obj*, *filepath*, *backend_args=None*)

Write text to a given `filepath` with 'w' mode.

Note: `put_text` should create a directory if the directory of `filepath` does not exist.

Parameters

- **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.

Return type None

Examples

```
>>> filepath = '/path/of/file'
>>> put_text('hello world', filepath)
```

51.3.22 mmengine.fileio.remove

mmengine.fileio.**remove**(*filepath*, *backend_args=None*)

Remove a file.

Parameters

- **filepath** (*str, Path*) – Path to be removed.
- **backend_args** (*dict, optional*) – Arguments to instantiate the corresponding backend. Defaults to None.

Raises

- **FileNotFoundError** – If filepath does not exist, an FileNotFoundError will be raised.
- **IsADirectoryError** – If filepath is a directory, an IsADirectoryError will be raised.

Return type None

Examples

```
>>> filepath = '/path/of/file'
>>> remove(filepath)
```

51.3.23 mmengine.fileio.rmtree

mmengine.fileio.**rmtree**(*dir_path*, *backend_args=None*)

Recursively delete a directory tree.

Parameters

- **dir_path** (*str or Path*) – A directory to be removed.
- **backend_args** (*dict, optional*) – Arguments to instantiate the corresponding backend. Defaults to None.

Return type None

Examples

```
>>> dir_path = '/path/of/dir'
>>> rmtree(dir_path)
```

51.4 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.

51.4.1 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.

Parameters

- **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.

Examples

```
>>> 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'}
```

Returns The parsed contents.

Return type dict

51.4.2 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.

Parameters

- **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.

Examples

```
>>> list_from_file('/path/of/your/file')  # disk
['hello', 'world']
>>> list_from_file('s3://path/of/your/file')  # ceph or petrel
['hello', 'world']
```

Returns A list of strings.

Return type list[str]

CHAPTER

FIFTYTWO

MMENGINE.DIST

mmengine.dist

- *dist*
- *utils*

52.1 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.
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.

52.1.1 mmengine.dist.gather

`mmengine.dist.`**`gather`**(*data*, *dst=0*, *group=None*)

Gather data from the whole group to `dst` process.

Note: Calling `gather` in non-distributed environment dose nothing and just returns a list containing `data` itself.

Note: NCCL backend does not support `gather`.

Note: 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

Parameters

- **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.

Returns `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.

Return type list[Tensor]

Examples

```
>>> 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
```

52.1.2 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.

Note: `NCCL backend` does not support `gather_object`.

Note: 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

---

Parameters

- **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.

Returns list[Any]. On the `dst` rank, return `gather_list` which contains the output of the collective.

Return type Optional[List[Any]]

Examples

```
>>> 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
```

52.1.3 mmengine.dist.all_gather

mmengine.dist.**all_gather**(*data*, *group=None*)

Gather data from the whole group in a list.

Note: Calling `all_gather` in non-distributed environment does nothing and just returns a list containing `data` itself.

Note: 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

Parameters

- **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.

Returns Return a list containing data from the whole group if in distributed environment, otherwise a list only containing `data` itself.

Return type list[Tensor]

Examples

```
>>> 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
```

52.1.4 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.

Note: Calling `all_gather_object` in non-distributed environment does nothing and just returns a list containing `data` itself.

Note: 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

---

Parameters

- **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.

Returns Return a list containing data from the whole group if in distributed environment, otherwise a list only containing `data` itself.

Return type list[Tensor]

Note: 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()`.

Examples

```
>>> 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
```

(continues on next page)

(continued from previous page)

```
['foo', 12, {1: 2}]  # Rank 0
['foo', 12, {1: 2}]  # Rank 1
['foo', 12, {1: 2}]  # Rank 2
```

52.1.5 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.

Note: Calling `all_reduce` in non-distributed environment does nothing.

Parameters

- **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.

Return type None

Examples

```
>>> 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
```

52.1.6 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.

Parameters

- **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.

Return type None

Examples

```
>>> 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
```

52.1.7 mmengine.dist.all_reduce_params

mmengine.dist.**all_reduce_params**(*params*, *coalesce=True*, *bucket_size_mb=- 1*, *op='sum'*, *group=None*)

All-reduce parameters.

Parameters

- **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.

Return type None

Examples

```
>>> 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])]
```

52.1.8 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.

Note: Calling `broadcast` in non-distributed environment does nothing.

Parameters

- **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.

Return type None

Examples

```
>>> 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
```

52.1.9 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.

Parameters **group** (*ProcessGroup, optional*) – The process group to work on. If None, the default process group will be used. Defaults to None.

Returns Random seed.

Return type int

Examples

```
>>> 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
```

52.1.10 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.

Note: Calling `broadcast_object_list` in non-distributed environment does nothing.

Parameters

- **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`.

Return type None

Note: 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()`.

Examples

```
>>> 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
```

52.1.11 mmengine.dist.collect_results

mmengine.dist.**collect_results**(*results*, *size*, *device='cpu'*, *tmpdir=None*)

Collected results in distributed environments.

Parameters

- **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.

Returns The collected results.

Return type list or None

Examples

```
>>> # 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
```

52.1.12 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.

Parameters

- **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.

Returns The collected results.

Return type list or None

Examples

```
>>> # 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
```

52.1.13 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.

Parameters

- **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.

Returns The collected results.

Return type list or None

Examples

```
>>> # 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
['foo', 24, {1: 2}, {'a': 'b'}]  # rank 0
None  # rank 1
```

52.2 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`.

52.2.1 mmengine.dist.get_dist_info

mmengine.dist.**get_dist_info**(*group=None*)

Get distributed information of the given process group.

Note: Calling `get_dist_info` in non-distributed environment will return (0, 1).

Parameters **group** (*ProcessGroup, optional*) – The process group to work on. If None, the default process group will be used. Defaults to None.

Returns Return a tuple containing the `rank` and `world_size`.

Return type tuple[int, int]

52.2.2 mmengine.dist.init_dist

mmengine.dist.**init_dist**(*launcher*, *backend='nccl'*, ***kwargs*)

Initialize distributed environment.

Parameters

- **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`.

Return type None

52.2.3 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

Parameters

- **node_rank** (*int*) – Rank of machines used for training.
- **num_gpus_per_node** (*int*) – Number of gpus used for training in a single machine.

52.2.4 mmengine.dist.get_backend

mmengine.dist.**get_backend**(*group=None*)

Return the backend of the given process group.

Note: Calling `get_backend` in non-distributed environment will return None.

Parameters **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.

Returns Return the backend of the given process group as a lower case string if in distributed environment, otherwise None.

Return type str or None

52.2.5 mmengine.dist.get_world_size

mmengine.dist.**get_world_size**(*group=None*)
Return the number of the given process group.

Note: Calling `get_world_size` in non-distributed environment will return 1.

Parameters **group** (*ProcessGroup, optional*) – The process group to work on. If None, the default process group will be used. Defaults to None.

Returns Return the number of processes of the given process group if in distributed environment, otherwise 1.

Return type int

52.2.6 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`.

Note: Calling `get_rank` in non-distributed environment will return 0.

Parameters **group** (*ProcessGroup, optional*) – The process group to work on. If None, the default process group will be used. Defaults to None.

Returns Return the rank of the process group if in distributed environment, otherwise 0.

Return type int

52.2.7 mmengine.dist.get_local_size

mmengine.dist.**get_local_size**()
Return the number of the current node.

Returns Return the number of processes in the current node if in distributed environment, otherwise 1.

Return type int

52.2.8 mmengine.dist.get_local_rank

mmengine.dist.**get_local_rank**()
Return the rank of current process in the current node.

Returns Return the rank of current process in the current node if in distributed environment, otherwise 0

Return type int

52.2.9 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.

Parameters **group** (*ProcessGroup, optional*) – The process group to work on. If None, the default process group will be used. Defaults to None.

Returns Return True if the current rank of the given process group is equal to 0, otherwise False.

Return type bool

52.2.10 mmengine.dist.master_only

mmengine.dist.**master_only**(*func*)
Decorate those methods which should be executed in master process.

Parameters **func** (*callable*) – Function to be decorated.

Returns Return decorated function.

Return type callable

52.2.11 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.

Note: Calling `barrier` in non-distributed environment will do nothing.

Parameters **group** (*ProcessGroup, optional*) – The process group to work on. If None, the default process group will be used. Defaults to None.

Return type None

52.2.12 mmengine.dist.is_distributed

mmengine.dist.**is_distributed**()
Return True if distributed environment has been initialized.

Return type bool

52.2.13 mmengine.dist.get_local_group

mmengine.dist.**get_local_group**()
Return local process group.

Return type Optional[torch.distributed.distributed_c10d.ProcessGroup]

52.2.14 mmengine.dist.get_default_group

mmengine.dist.**get_default_group**()
Return default process group.

Return type Optional[torch.distributed.distributed_c10d.ProcessGroup]

52.2.15 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.

Parameters **data** (*Tensor or Sequence or dict*) – Inputs to be inferred the device.

Returns The device of `data`.

Return type torch.device

Examples

```
>>> 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')
```

52.2.16 mmengine.dist.get_comm_device

mmengine.dist.**get_comm_device**(*group=None*)

Return the device for communication among groups.

Parameters **group** (*ProcessGroup, optional*) – The process group to work on.

Returns The device of backend.

Return type torch.device

52.2.17 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.

Parameters

- **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.

Returns `data` was casted to `device`.

Return type Tensor or list or dict

CHAPTER

FIFTYTHREE

MMENGINE.UTILS

mmengine.utils

53.1 Manager

ManagerMeta	The metaclass for global accessible class.
ManagerMixin	`ManagerMixin` is the base class for classes that have global access requirements.

53.1.1 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.

Examples

```
>>> 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.
```

53.1.2 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.

Examples

```
>>> 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)
```

Parameters **name** (*str*) – Name of the instance. Defaults to ‘’.

classmethod **check_instance_created**(*name*)

Check whether the name corresponding instance exists.

Parameters **name** (*str*) – Name of instance.

Returns Whether the name corresponding instance exists.

Return type 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
```

(continues on next page)

(continued from previous page)

```
name1
>>> instance = GlobalAccessible.get_current_instance()
>>> instance.instance_name
name1
```

Returns Latest created instance.

Return type 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)
```

Parameters **name** (*str*) – Name of instance. Defaults to ‘’.

Returns Corresponding name instance, the latest instance, or root instance.

Return type object

property instance_name: str

Get the name of instance.

Returns Name of instance.

Return type str

53.2 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	

53.2.1 mmengine.utils.check_file_exist

mmengine.utils.**check_file_exist**(*filename*, *msg_tmpl='file "{}" does not exist'*)

53.2.2 mmengine.utils.fopen

mmengine.utils.**fopen**(*filepath*, **args*, ***kwargs*)

53.2.3 mmengine.utils.is_abs

mmengine.utils.**is_abs**(*path*)

Check if path is an absolute path in different backends.

Parameters **path** (*str*) – path of directory or file.

Returns whether path is an absolute path.

Return type bool

53.2.4 mmengine.utils.is_filepath

mmengine.utils.**is_filepath**(*x*)

53.2.5 mmengine.utils.mkdir_or_exist

mmengine.utils.**mkdir_or_exist**(*dir_name*, *mode=511*)

53.2.6 mmengine.utils.scandir

mmengine.utils.**scandir**(*dir_path*, *suffix=None*, *recursive=False*, *case_sensitive=True*)

Scan a directory to find the interested files.

Parameters

- **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.

Returns A generator for all the interested files with relative paths.

53.2.7 mmengine.utils.symlink

mmengine.utils.**symlink**(*src*, *dst*, *overwrite=True*, ***kwargs*)

53.3 Package

call_command	
install_package	
get_installed_path	Get installed path of package.
is_installed	Check package whether installed.

53.3.1 mmengine.utils.call_command

mmengine.utils.**call_command**(*cmd*)

Parameters **cmd** (*list*) –

Return type None

53.3.2 mmengine.utils.install_package

mmengine.utils.**install_package**(*package*)

Parameters **package** (*str*) –

53.3.3 mmengine.utils.get_installed_path

mmengine.utils.**get_installed_path**(*package*)

Get installed path of package.

Parameters **package** (*str*) – Name of package.

Return type str

Example

```
>>> get_installed_path('mmcls')
>>> '.../lib/python3.7/site-packages/mmcls'
```

53.3.4 mmengine.utils.is_installed

mmengine.utils.**is_installed**(*package*)
Check package whether installed.

Parameters **package** (*str*) – Name of package to be checked.

Return type bool

53.4 Version

digit_version	Convert a version string into a tuple of integers.
get_git_hash	Get the git hash of the current repo.

53.4.1 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.

Parameters

- **version_str** (*str*) – The version string.
- **length** (*int*) – The maximum number of version levels. Default: 4.

Returns The version info in digits (integers).

Return type tuple[int]

53.4.2 mmengine.utils.get_git_hash

mmengine.utils.**get_git_hash**(*fallback='unknown'*, *digits=None*)
Get the git hash of the current repo.

Parameters

- **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.

Returns Git commit hash.

Return type str

53.5 Progress Bar

ProgressBar	A progress bar which can print the progress.

53.5.1 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.

53.5.2 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.

Parameters

- **tasks** (*list or tuple[Iterable, int]*) – A list of tasks or (tasks, total num).
- **bar_width** (*int*) – Width of progress bar.

Yields *list* – The task results.

53.5.3 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()`.

Parameters

- **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.

Returns The task results.

Return type list

53.5.4 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.

Parameters

- **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.

Returns The task results.

Return type list

53.6 Miscellaneous

Timer	A flexible Timer class.
TimerError	

53.6.1 Timer

class mmengine.utils.**Timer**(*start=True*, *print_tmpl=None*)

A flexible Timer class.

Examples

```
>>> 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.

Returns Time in seconds.

Return type float

since_start()
Total time since the timer is started.

Returns Time in seconds.

Return type float

start()
Start the timer.

53.6.2 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.

continues on next page

Table 8 – continued from previous page

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.
deprecated_function	Marks functions as deprecated.
has_method	Check whether the object has a method.
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.

53.6.3 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()*.

53.6.4 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()*.

53.6.5 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.

Parameters

- **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.

Returns Return True if `seq` is valid else False.

Return type bool

Examples

```
>>> from mmengine.utils import is_seq_of
>>> seq = ['a', 'b', 'c']
>>> is_seq_of(seq, str)
True
>>> is_seq_of(seq, int)
False
```

53.6.6 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.

53.6.7 mmengine.utils.iter_cast

mmengine.utils.**iter_cast**(*inputs*, *dst_type*, *return_type=None*)

Cast elements of an iterable object into some type.

Parameters

- **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.

Returns The converted object.

Return type iterator or specified type

53.6.8 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()*.

53.6.9 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()*.

53.6.10 mmengine.utils.concat_list

mmengine.utils.**concat_list**(*in_list*)
Concatenate a list of list into a single list.

Parameters **in_list** (*list*) – The list of list to be merged.

Returns The concatenated flat list.

Return type list

53.6.11 mmengine.utils.slice_list

mmengine.utils.**slice_list**(*in_list*, *lens*)
Slice a list into several sub lists by a list of given length.

Parameters

- **in_list** (*list*) – The list to be sliced.
- **lens** (*int or list*) – The expected length of each out list.

Returns A list of sliced list.

Return type list

53.6.12 mmengine.utils.to_1tuple

mmengine.utils.**to_1tuple**(*x*)

53.6.13 mmengine.utils.to_2tuple

mmengine.utils.**to_2tuple**(*x*)

53.6.14 mmengine.utils.to_3tuple

mmengine.utils.**to_3tuple**(*x*)

53.6.15 mmengine.utils.to_4tuple

mmengine.utils.**to_4tuple**(*x*)

53.6.16 mmengine.utils.to_ntuple

mmengine.utils.**to_ntuple**(*n*)

53.6.17 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.

Parameters

- **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.

Returns A specific decorator.

Return type decorator

53.6.18 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.

Parameters

- **name_dict** (*dict*) – key (str): Deprecate argument names. val (str): Expected argument names.
- **cls_name** (*Optional[str]*) –

Returns New function.

Return type func

53.6.19 mmengine.utils.deprecated_function

mmengine.utils.**deprecated_function**(*since*, *removed_in*, *instructions*)

Marks functions as deprecated.

Throw a warning when a deprecated function is called, and add a note in the docstring. Modified from https://github.com/pytorch/pytorch/blob/master/torch/onnx/_deprecation.py

Parameters

- **since** (*str*) – The version when the function was first deprecated.
- **removed_in** (*str*) – The version when the function will be removed.
- **instructions** (*str*) – The action users should take.

Returns A new function, which will be deprecated soon.

Return type Callable

53.6.20 mmengine.utils.has_method

mmengine.utils.**has_method**(*obj*, *method*)

Check whether the object has a method.

Parameters

- **method** (*str*) – The method name to check.
- **obj** (*object*) – The object to check.

Returns True if the object has the method else False.

Return type bool

53.6.21 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.

Parameters

- **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.

Return type bool

53.6.22 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.

Parameters

- **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.

Returns The imported modules.

Return type list[module] | module | None

Examples

```
>>> osp, sys = import_modules_from_strings(
...     ['os.path', 'sys'])
>>> import os.path as osp_
>>> import sys as sys_
>>> assert osp == osp_
>>> assert sys == sys_
```

53.6.23 mmengine.utils.requires_executable

mmengine.utils.**requires_executable**(*prerequisites*)

A decorator to check if some executable files are installed.

Example

```
>>> @requires_executable('ffmpeg')
>>> func(arg1, args):
>>>     print(1)
1
```

53.6.24 mmengine.utils.requires_package

mmengine.utils.**requires_package**(*prerequisites*)

A decorator to check if some python packages are installed.

Example

```
>>> @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
```

53.6.25 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.

Examples

```
>>> 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
```

Parameters **str** – Timer identifier.

CHAPTER

FIFTYFOUR

MMENGINE.UTILS.DL_UTILS

TimeCounter	A tool that counts the average running time of a function or a method.

54.1 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.

Parameters

- **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.

Examples

```
>>> 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.

Parameters **elapsed** (*Union[int, float]*) –

Return type 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.
is_jit_tracing	

54.2 mmengine.utils.dl_utils.collect_env

mmengine.utils.dl_utils.**collect_env**()
Collect the information of the running environments.

Returns

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.

Return type dict

54.3 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()`.

Parameters

- **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.

Return type Dict[str, Any]

Example

```
>>> state_dict = torch.hub.load_state_dict_from_url('https://s3.amazonaws.com/
↪pytorch/models/resnet18-5c106cde.pth')
```

54.4 mmengine.utils.dl_utils.has_batch_norm

mmengine.utils.dl_utils.**has_batch_norm**(*model*)

Detect whether model has a BatchNormalization layer.

Parameters **model** (*nn.Module*) – training model.

Returns whether model has a BatchNormalization layer

Return type bool

54.5 mmengine.utils.dl_utils.is_norm

mmengine.utils.dl_utils.**is_norm**(*layer*, *exclude=None*)

Check if a layer is a normalization layer.

Parameters

- **layer** (*nn.Module*) – The layer to be checked.
- **exclude** (*type, tuple[type], optional*) – Types to be excluded.

Returns Whether the layer is a norm layer.

Return type bool

54.6 mmengine.utils.dl_utils.mmcv_full_available

mmengine.utils.dl_utils.**mmcv_full_available**()

Check whether mmcv-full is installed.

Returns True if mmcv-full is installed else False.

Return type bool

54.7 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.

Parameters

- **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.

Returns A list that contains multiple images.

Return type list[np.ndarray]

54.8 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:: Examples

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'

54.9 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.

Parameters

- **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.

Return type None

54.10 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.

Parameters **tensors** (*List[Tensor]*) – List of scalars or 1 dimensional tensors.

Returns Sequence of meshgrid tensors.

Return type Sequence[Tensor]

54.11 mmengine.utils.dl_utils.is_jit_tracing

mmengine.utils.dl_utils.**is_jit_tracing**()

Return type bool

CHAPTER

FIFTYFIVE

CHANGELOG OF V0.X

55.1 v0.4.0 (12/28/2022)

55.1.1 Highlights

- Registry supports importing modules automatically
- Upgrade the documentation and provide the **English documentation**
- Provide `ProfileHook` to profile the running process

55.1.2 New Features & Enhancements

- Add `conf_path` in PetrelBackend by @sunyc11 in https://github.com/open-mmlab/mmengine/pull/774
- Support multiple `--cfg-options`. by @mzr1996 in https://github.com/open-mmlab/mmengine/pull/759
- Support passing arguments to `OptimWrapper.update_params` by @twmht in https://github.com/open-mmlab/mmengine/pull/796
- Make `get_torchvision_model` compatible with torch 1.13 by @HAOCHENYE in https://github.com/open-mmlab/mmengine/pull/793
- Support `flat_decay_mult` and fix `bias_decay_mult` of depth-wise-conv in `DefaultOptimWrapperConstructor` by @RangiLyu in https://github.com/open-mmlab/mmengine/pull/771
- Registry supports importing modules automatically. by @RangiLyu in https://github.com/open-mmlab/mmengine/pull/643
- Add profiler hook functionality by @BayMaxBHL in https://github.com/open-mmlab/mmengine/pull/768
- Make TTAModel compatible with FSDP. by @HAOCHENYE in https://github.com/open-mmlab/mmengine/pull/611

55.1.3 Bug Fixes

- `hub.get_model` fails on some MMCls models by @C1rN09 in https://github.com/open-mmlab/mmengine/pull/784
- Fix `BaseModel.to` and `BaseDataPreprocessor.to` to make them consistent with `torch.nn.Module` by @C1rN09 in https://github.com/open-mmlab/mmengine/pull/783
- Fix creating a new logger at PretrainedInit by @xiexinch in https://github.com/open-mmlab/mmengine/pull/791
- Fix `ZeroRedundancyOptimizer` ambiguous error with param groups when PyTorch < 1.12.0 by @C1rN09 in https://github.com/open-mmlab/mmengine/pull/818
- Fix MessageHub set resumed key repeatedly by @HAOCHENYE in https://github.com/open-mmlab/mmengine/pull/839
- Add `progress` argument to `load_from_http` by @austinmw in https://github.com/open-mmlab/mmengine/pull/770
- Ensure metrics is not empty when saving best checkpoint by @zhouzaida in https://github.com/open-mmlab/mmengine/pull/849

55.1.4 Docs

- Add `contributing.md` by @HAOCHENYE in https://github.com/open-mmlab/mmengine/pull/754
- Add gif to 15 min tutorial by @HAOCHENYE in https://github.com/open-mmlab/mmengine/pull/748
- Refactor documentations and translate them to English by @zhouzaida in https://github.com/open-mmlab/mmengine/pull/786
- Fix document link by @MambaWong in https://github.com/open-mmlab/mmengine/pull/775
- Fix typos in EN `contributing.md` by @RangeKing in https://github.com/open-mmlab/mmengine/pull/792
- Translate data transform docs. by @mzr1996 in https://github.com/open-mmlab/mmengine/pull/737
- Replace markdown table with html table by @HAOCHENYE in https://github.com/open-mmlab/mmengine/pull/800
- Fix wrong example in `Visualizer.draw_polygons` by @lyviva in https://github.com/open-mmlab/mmengine/pull/798
- Fix docstring format and rescale the images by @zhouzaida in https://github.com/open-mmlab/mmengine/pull/802
- Fix failed link in registry by @zhouzaida in https://github.com/open-mmlab/mmengine/pull/811
- Fix typos by @shanmo in https://github.com/open-mmlab/mmengine/pull/814
- Fix wrong links and typos in docs by @shanmo in https://github.com/open-mmlab/mmengine/pull/815
- Translate `save_gpu_memory.md` by @xin-li-67 in https://github.com/open-mmlab/mmengine/pull/803
- Translate the documentation of hook design by @zhouzaida in https://github.com/open-mmlab/mmengine/pull/780
- Fix docstring format by @zhouzaida in https://github.com/open-mmlab/mmengine/pull/816
- Translate `registry.md` by @xin-li-67 in https://github.com/open-mmlab/mmengine/pull/817
- Update docstring of `BaseDataElement` by @Xiangxu-0103 in https://github.com/open-mmlab/mmengine/pull/836

- Fix typo by @Xiangxu-0103 in https://github.com/open-mmlab/mmengine/pull/841
- Update docstring of `structures` by @Xiangxu-0103 in https://github.com/open-mmlab/mmengine/pull/840
- Translate `optim_wrapper.md` by @xin-li-67 in https://github.com/open-mmlab/mmengine/pull/833
- Fix link error in initialize tutorial. by @HAOCHENYE in https://github.com/open-mmlab/mmengine/pull/843
- Fix table in `initialized.md` by @HAOCHENYE in https://github.com/open-mmlab/mmengine/pull/844

55.1.5 Contributors

A total of 16 developers contributed to this release. Thanks @BayMaxBHL, @RangeKing, @Xiangxu-0103, @xin-li-67, @twmht, @shanmo, @sunyc11, @lyviva, @austinmw, @xiexinch, @mzr1996, @RangiLyu, @MambaWong, @C1rN09, @zhouzaida, @HAOCHENYE

55.2 v0.3.2 (11/24/2022)

55.2.1 New Features & Enhancements

- Send git errors to subprocess.PIPE by @austinmw in https://github.com/open-mmlab/mmengine/pull/717
- Add a common `TestRunnerTestCase` to build a Runner instance. by @HAOCHENYE in https://github.com/open-mmlab/mmengine/pull/631
- Align the log by @HAOCHENYE in https://github.com/open-mmlab/mmengine/pull/436
- Log the called order of hooks during training process by @songyuc in https://github.com/open-mmlab/mmengine/pull/672
- Support setting `eta_min_ratio` in `CosineAnnealingParamScheduler` by @cir7 in https://github.com/open-mmlab/mmengine/pull/725
- Enhance compatibility of `revert_sync_batchnorm` by @HAOCHENYE in https://github.com/open-mmlab/mmengine/pull/695

55.2.2 Bug Fixes

- Fix `distributed_training.py` in examples by @PingHGao in https://github.com/open-mmlab/mmengine/pull/700
- Format the log of `CheckpointLoader.load_checkpoint` by @HAOCHENYE in https://github.com/open-mmlab/mmengine/pull/685
- Fix bug of CosineAnnealingParamScheduler by @fangyixiao18 in https://github.com/open-mmlab/mmengine/pull/735
- Fix `add_graph` is not called bug by @shenmishajing in https://github.com/open-mmlab/mmengine/pull/632
- Fix .pre-commit-config-zh-cn.yaml pyupgrade-repo github->gitee by @BayMaxBHL in https://github.com/open-mmlab/mmengine/pull/756

55.2.3 Docs

- Add English docs of BaseDataset by @GT9505 in https://github.com/open-mmlab/mmengine/pull/713
- Fix `BaseDataset` typo about lazy initialization by @MengzhangLI in https://github.com/open-mmlab/mmengine/pull/733
- Fix typo by @zhouzaida in https://github.com/open-mmlab/mmengine/pull/734
- Translate visualization docs by @xin-li-67 in https://github.com/open-mmlab/mmengine/pull/692

55.3 v0.3.1 (11/09/2022)

55.3.1 Highlights

- Fix error when saving best checkpoint in ddp-training

55.3.2 New Features & Enhancements

- Replace `print` with `print_log` for those functions called by runner by @HAOCHENYE in https://github.com/open-mmlab/mmengine/pull/686

55.3.3 Bug Fixes

- Fix error when saving best checkpoint in ddp-training by @HAOCHENYE in https://github.com/open-mmlab/mmengine/pull/682

55.3.4 Docs

- Refine Chinese tutorials by @Xiangxu-0103 in https://github.com/open-mmlab/mmengine/pull/694
- Add MMEval in README by @sanbuphy in https://github.com/open-mmlab/mmengine/pull/669
- Fix error URL in runner docstring by @HAOCHENYE in https://github.com/open-mmlab/mmengine/pull/668
- Fix error evaluator type name in `evaluator.md` by @sanbuphy in https://github.com/open-mmlab/mmengine/pull/675
- Fix typo in `utils.md` @sanbuphy in https://github.com/open-mmlab/mmengine/pull/702

55.4 v0.3.0 (11/02/2022)

55.4.1 New Features & Enhancements

- Support running on Ascend chip by @wangjiangben-hw in https://github.com/open-mmlab/mmengine/pull/572
- Support torch `ZeroRedundancyOptimizer` by @nijkah in https://github.com/open-mmlab/mmengine/pull/551
- Add non-blocking feature to `BaseDataPreprocessor` by @shenmishajing in https://github.com/open-mmlab/mmengine/pull/618

- Add documents for `clip_grad`, and support clip grad by value. by @HAOCHENYE in https://github.com/open-mmlab/mmengine/pull/513
- Add ROCm info when collecting env by @zhouzaida in https://github.com/open-mmlab/mmengine/pull/633
- Add a function to mark the deprecated function. by @HAOCHENYE in https://github.com/open-mmlab/mmengine/pull/609
- Call `register_all_modules` in `Registry.get()` by @HAOCHENYE in https://github.com/open-mmlab/mmengine/pull/541
- Deprecate `_save_to_state_dict` implemented in mmengine by @HAOCHENYE in https://github.com/open-mmlab/mmengine/pull/610
- Add `ignore_keys` in ConcatDataset by @BIGWangYuDong in https://github.com/open-mmlab/mmengine/pull/556

55.4.2 Docs

- Fix cannot show `changelog.md` in chinese documents. by @HAOCHENYE in https://github.com/open-mmlab/mmengine/pull/606
- Fix Chinese docs whitespaces by @C1rN09 in https://github.com/open-mmlab/mmengine/pull/521
- Translate installation and 15_min by @xin-li-67 in https://github.com/open-mmlab/mmengine/pull/629
- Refine chinese doc by @Tau-J in https://github.com/open-mmlab/mmengine/pull/516
- Add MMYOLO link in README by @Xiangxu-0103 in https://github.com/open-mmlab/mmengine/pull/634
- Add MMEngine logo in docs by @zhouzaida in https://github.com/open-mmlab/mmengine/pull/641
- Fix docstring of `BaseDataset` by @HAOCHENYE in https://github.com/open-mmlab/mmengine/pull/656
- Fix docstring and documentation used for `hub.get_model` by @zengyh1900 in https://github.com/open-mmlab/mmengine/pull/659
- Fix typo in `docs/zh_cn/advanced_tutorials/visualization.md` by @MambaWong in https://github.com/open-mmlab/mmengine/pull/616
- Fix typo docstring of `DefaultOptimWrapperConstructor` by @triple-Mu in https://github.com/open-mmlab/mmengine/pull/644
- Fix typo in advanced tutorial by @cxiang26 in https://github.com/open-mmlab/mmengine/pull/650
- Fix typo in `Config` docstring by @sanbuphy in https://github.com/open-mmlab/mmengine/pull/654
- Fix typo in `docs/zh_cn/tutorials/config.md` by @Xiangxu-0103 in https://github.com/open-mmlab/mmengine/pull/596
- Fix typo in `docs/zh_cn/tutorials/model.md` by @C1rN09 in https://github.com/open-mmlab/mmengine/pull/598

55.4.3 Bug Fixes

- Fix error calculation of `eta_min` in `CosineRestartParamScheduler` by @Z-Fran in https://github.com/open-mmlab/mmengine/pull/639
- Fix `BaseDataPreprocessor.cast_data` could not handle string data by @HAOCHENYE in https://github.com/open-mmlab/mmengine/pull/602
- Make `autocast` compatible with mps by @HAOCHENYE in https://github.com/open-mmlab/mmengine/pull/587
- Fix error format of log message by @HAOCHENYE in https://github.com/open-mmlab/mmengine/pull/508
- Fix error implementation of `is_model_wrapper` by @HAOCHENYE in https://github.com/open-mmlab/mmengine/pull/640
- Fix `VisBackend.add_config` is not called by @shenmishajing in https://github.com/open-mmlab/mmengine/pull/613
- Change `strict_load` of EMAHook to False by default by @HAOCHENYE in https://github.com/open-mmlab/mmengine/pull/642
- Fix `open` encoding problem of Config in Windows by @sanbuphy in https://github.com/open-mmlab/mmengine/pull/648
- Fix the total number of iterations in log is a float number. by @jbwang1997 in https://github.com/open-mmlab/mmengine/pull/604
- Fix `pip upgrade` CI by @HAOCHENYE in https://github.com/open-mmlab/mmengine/pull/622

55.4.4 New Contributors

- @shenmishajing made their first contribution in https://github.com/open-mmlab/mmengine/pull/618
- @Xiangxu-0103 made their first contribution in https://github.com/open-mmlab/mmengine/pull/596
- @Tau-J made their first contribution in https://github.com/open-mmlab/mmengine/pull/516
- @wangjiangben-hw made their first contribution in https://github.com/open-mmlab/mmengine/pull/572
- @triple-Mu made their first contribution in https://github.com/open-mmlab/mmengine/pull/644
- @sanbuphy made their first contribution in https://github.com/open-mmlab/mmengine/pull/648
- @Z-Fran made their first contribution in https://github.com/open-mmlab/mmengine/pull/639
- @BIGWangYuDong made their first contribution in https://github.com/open-mmlab/mmengine/pull/556
- @zengyh1900 made their first contribution in https://github.com/open-mmlab/mmengine/pull/659

55.5 v0.2.0 (11/10/2022)

55.5.1 New Features & Enhancements

- Add SMDDP backend and support running on AWS by @austinmw in https://github.com/open-mmlab/mmengine/pull/579
- Refactor `FileIO` but without breaking bc by @zhouzaida in https://github.com/open-mmlab/mmengine/pull/533
- Add test time augmentation base model by @HAOCHENYE in https://github.com/open-mmlab/mmengine/pull/538

- Use `torch.lerp\_()` to speed up EMA by @RangiLyu in https://github.com/open-mmlab/mmengine/pull/519
- Support converting `BN` to `SyncBN` by config by @HAOCHENYE in https://github.com/open-mmlab/mmengine/pull/506
- Support defining metric name in wandb backend by @okotaku in https://github.com/open-mmlab/mmengine/pull/509
- Add dockerfile by @zhouzaida in https://github.com/open-mmlab/mmengine/pull/347

55.5.2 Docs

- Fix API files of English documentation by @zhouzaida in https://github.com/open-mmlab/mmengine/pull/525
- Fix typo in `instance_data.py` by @Dai-Wenxun in https://github.com/open-mmlab/mmengine/pull/530
- Fix the docstring of the model sub-package by @zhouzaida in https://github.com/open-mmlab/mmengine/pull/573
- Fix a spelling error in docs/zh_cn by @cxiang26 in https://github.com/open-mmlab/mmengine/pull/548
- Fix typo in docstring by @MengzhangLI in https://github.com/open-mmlab/mmengine/pull/527
- Update `config.md` by @Zhengfei-0311 in https://github.com/open-mmlab/mmengine/pull/562

55.5.3 Bug Fixes

- Fix `LogProcessor` does not smooth loss if the name of loss doesn't start with `loss` by @liuyanyi in https://github.com/open-mmlab/mmengine/pull/539
- Fix failed to enable `detect_anomalous_params` in `MMSeparateDistributedDataParallel` by @HAOCHENYE in https://github.com/open-mmlab/mmengine/pull/588
- Fix CheckpointHook behavior unexpected if given `filename_tmpl` argument by @C1rN09 in https://github.com/open-mmlab/mmengine/pull/518
- Fix error argument sequence in `FSDP` by @HAOCHENYE in https://github.com/open-mmlab/mmengine/pull/520
- Fix uploading image in wandb backend @okotaku in https://github.com/open-mmlab/mmengine/pull/510
- Fix loading state dictionary in `EMAHook` by @okotaku in https://github.com/open-mmlab/mmengine/pull/507
- Fix circle import in `EMAHook` by @HAOCHENYE in https://github.com/open-mmlab/mmengine/pull/523
- Fix unit test could fail caused by `MultiProcessTestCase` by @HAOCHENYE in https://github.com/open-mmlab/mmengine/pull/535
- Remove unnecessary "if statement" in `Registry` by @MambaWong in https://github.com/open-mmlab/mmengine/pull/536
- Fix `_save_to_state_dict` by @HAOCHENYE in https://github.com/open-mmlab/mmengine/pull/542
- Support comparing NumPy array dataset meta in `Runner.resume` by @HAOCHENYE in https://github.com/open-mmlab/mmengine/pull/511
- Use `get` instead of `pop` to dump `runner_type` in `build_runner_from_cfg` by @nijkah in https://github.com/open-mmlab/mmengine/pull/549
- Upgrade pre-commit hooks by @zhouzaida in https://github.com/open-mmlab/mmengine/pull/576
- Delete the error comment in `registry.md` by @vansin in https://github.com/open-mmlab/mmengine/pull/514

- Fix Some out-of-date unit tests by @C1rN09 in https://github.com/open-mmlab/mmengine/pull/586
- Fix typo in `MMFullyShardedDataParallel` by @yhna940 in https://github.com/open-mmlab/mmengine/pull/569
- Update Github Action CI and CircleCI by @zhouzaida in https://github.com/open-mmlab/mmengine/pull/512
- Fix unit test in windows by @HAOCHENYE in https://github.com/open-mmlab/mmengine/pull/515
- Fix merge ci & multiprocessing unit test by @HAOCHENYE in https://github.com/open-mmlab/mmengine/pull/529

55.5.4 New Contributors

- @okotaku made their first contribution in https://github.com/open-mmlab/mmengine/pull/510
- @MengzhangLI made their first contribution in https://github.com/open-mmlab/mmengine/pull/527
- @MambaWong made their first contribution in https://github.com/open-mmlab/mmengine/pull/536
- @cxiang26 made their first contribution in https://github.com/open-mmlab/mmengine/pull/548
- @nijkah made their first contribution in https://github.com/open-mmlab/mmengine/pull/549
- @Zhengfei-0311 made their first contribution in https://github.com/open-mmlab/mmengine/pull/562
- @austinmw made their first contribution in https://github.com/open-mmlab/mmengine/pull/579
- @yhna940 made their first contribution in https://github.com/open-mmlab/mmengine/pull/569
- @liuyanyi made their first contribution in https://github.com/open-mmlab/mmengine/pull/539

CHAPTER

FIFTYSIX

CONTRIBUTING TO OPENMMLAB

Welcome to the MMEngine community, we are committed to building a cutting-edge computer vision foundational library and all kinds of contributions are welcomed, including but not limited to

Fix bug

You can directly post a Pull Request to fix typos in code or documents

The steps to fix the bug of code implementation are as follows.

1. If the modification involves significant changes, you should create an issue first and describe the error information and how to trigger the bug. Other developers will discuss it with you and propose a proper solution.
2. Posting a pull request after fixing the bug and adding the corresponding unit test.

New Feature or Enhancement

1. If the modification involves significant changes, you should create an issue to discuss with our developers to propose a proper design.
2. Post a Pull Request after implementing the new feature or enhancement and add the corresponding unit test.

Document

You can directly post a pull request to fix documents. If you want to add a document, you should first create an issue to check if it is reasonable.

56.1 Pull Request Workflow

If you're not familiar with Pull Request, don't worry! The following guidance will tell you how to create a Pull Request step by step. If you want to dive into the development mode of Pull Request, you can refer to the official documents.

56.1.1 1. Fork and clone

If you are posting a pull request for the first time, you should fork the OpenMMLab repositories by clicking the **Fork** button in the top right corner of the GitHub page, and the forked repositories will appear under your GitHub profile.

Then, you can clone the repositories to local:

```
git clone git@github.com:{username}/mmengine.git
```

After that, you should add the official repository as the upstream repository.

```
git remote add upstream git@github.com:open-mmlab/mmengine
```

Check whether the remote repository has been added successfully by `git remote -v`.

```
origin        git@github.com:{username}/mmengine.git (fetch)
origin        git@github.com:{username}/mmengine.git (push)
upstream        git@github.com:open-mmlab/mmengine (fetch)
upstream        git@github.com:open-mmlab/mmengine (push)
```

Note: Here's a brief introduction to origin and upstream. When we use "git clone", we create an "origin" remote by default, which points to the repository cloned from. As for "upstream", we add it ourselves to point to the target repository. Of course, if you don't like the name "upstream", you could name it as you wish. Usually, we'll push the code to "origin". If the pushed code conflicts with the latest code in official("upstream"), we should pull the latest code from upstream to resolve the conflicts, and then push to "origin" again. The posted Pull Request will be updated automatically.

56.1.2 2. Configure pre-commit

You should configure pre-commit in the local development environment to make sure the code style matches that of OpenMMLab. **Note**: The following code should be executed under the mmengine directory.

```
pip install -U pre-commit
pre-commit install
```

Check that pre-commit is configured successfully, and install the hooks defined in `.pre-commit-config.yaml`.

```
pre-commit run --all-files
```

If the installation process is interrupted, you can repeatedly run `pre-commit run ...` to continue the installation.

If the code does not conform to the code style specification, pre-commit will raise a warning and fixes some of the errors automatically.

If we want to commit our code bypassing the pre-commit hook, we can use the `--no-verify` option(**only for temporary committing**).

```
git commit -m "xxx" --no-verify
```

56.1.3 3. Create a development branch

After configuring the pre-commit, we should create a branch based on the master branch to develop the new feature or fix the bug. The proposed branch name is `username/pr_name`

```
git checkout -b yhc/refactor_contributing_doc
```

In subsequent development, if the master branch of the local repository is behind the master branch of "upstream", we need to pull the upstream for synchronization, and then execute the above command:

```
git pull upstream master
```

56.1.4 4. Commit the code and pass the unit test

- MMEngine introduces mypy to do static type checking to increase the robustness of the code. Therefore, we need to add Type Hints to our code and pass the mypy check. If you are not familiar with Type Hints, you can refer to this tutorial.
- The committed code should pass through the unit test

```
# Pass all unit tests
pytest tests

# Pass the unit test of runner
pytest tests/test_runner/test_runner.py
```

  If the unit test fails for lack of dependencies, you can install the dependencies referring to the *guidance*

- If the documents are modified/added, we should check the rendering result referring to *guidance*

56.1.5 5. Push the code to remote

We could push the local commits to remote after passing through the check of unit test and pre-commit. You can associate the local branch with remote branch by adding `-u` option.

```
git push -u origin {branch_name}
```

This will allow you to use the `git push` command to push code directly next time, without having to specify a branch or the remote repository.

56.1.6 6. Create a Pull Request

(1) Create a pull request in GitHub's Pull request interface

(2) Modify the PR description according to the guidelines so that other developers can better understand your changes

Find more details about Pull Request description in *pull request guidelines*.

note

(a) The Pull Request description should contain the reason for the change, the content of the change, and the impact of the change, and be associated with the relevant Issue (see documentation

(b) If it is your first contribution, please sign the CLA

(c) Check whether the Pull Request pass through the CI

MMEngine will run unit test for the posted Pull Request on different platforms (Linux, Window, Mac), based on different versions of Python, PyTorch, CUDA to make sure the code is correct. We can see the specific test information by clicking `Details` in the above image so that we can modify the code.

(3) If the Pull Request passes the CI, then you can wait for the review from other developers. You'll modify the code based on the reviewer's comments, and repeat the steps *4*-*5* until all reviewers approve it. Then, we will merge it ASAP.

56.1.7 7. Resolve conflicts

If your local branch conflicts with the latest master branch of "upstream", you'll need to resolove them. There are two ways to do this:

```
git fetch --all --prune
git rebase upstream/master
```

or

```
git fetch --all --prune
git merge upstream/master
```

If you are very good at handling conflicts, then you can use rebase to resolve conflicts, as this will keep your commit logs tidy. If you are not familiar with `rebase`, then you can use `merge` to resolve conflicts.

56.2 Guidance

56.2.1 Unit test

We should also make sure the committed code will not decrease the coverage of unit test, we could run the following command to check the coverage of unit test:

```
python -m coverage run -m pytest /path/to/test_file
python -m coverage html
# check file in htmlcov/index.html
```

56.2.2 Document rendering

If the documents are modified/added, we should check the rendering result. We could install the dependencies and run the following command to render the documents and check the results:

```
pip install -r requirements/docs.txt
cd docs/zh_cn/
# or docs/en
make html
# check file in ./docs/zh_cn/_build/html/index.html
```

56.3 Python Code style

We adopt PEP8 as the preferred code style.

We use the following tools for linting and formatting:

- flake8: A wrapper around some linter tools.
- isort: A Python utility to sort imports.
- yapf: A formatter for Python files.
- codespell: A Python utility to fix common misspellings in text files.

- mdformat: Mdformat is an opinionated Markdown formatter that can be used to enforce a consistent style in Markdown files.
- docformatter: A formatter to format docstring.

Style configurations of yapf and isort can be found in setup.cfg.

We use pre-commit hook that checks and formats for `flake8`, `yapf`, `isort`, `trailing whitespaces`, `markdown files`, fixes `end-of-files`, `double-quoted-strings`, `python-encoding-pragma`, `mixed-line-ending`, sorts `requirments.txt` automatically on every commit. The config for a pre-commit hook is stored in .pre-commit-config.

56.4 PR Specs

1. Use pre-commit hook to avoid issues of code style
2. One short-time branch should be matched with only one PR
3. Accomplish a detailed change in one PR. Avoid large PR
    - Bad: Support Faster R-CNN
    - Acceptable: Add a box head to Faster R-CNN
    - Good: Add a parameter to box head to support custom conv-layer number
4. Provide clear and significant commit message
5. Provide clear and meaningful PR description
    - Task name should be clarified in title. The general format is: [Prefix] Short description of the PR (Suffix)
    - Prefix: add new feature [Feature], fix bug [Fix], related to documents [Docs], in developing [WIP] (which will not be reviewed temporarily)
    - Introduce main changes, results and influences on other modules in short description
    - Associate related issues and pull requests with a milestone

CHAPTER

FIFTYSEVEN

ENGLISH

CHAPTER

FIFTYEIGHT

CHAPTER

FIFTYNINE

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

Symbols

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