mmfewshot

MMFewShot Author

Nov 25, 2021

GET STARTED

CHAPTER

ONE

PREREQUISITES

- Linux (Windows is not officially supported)
- Python 3.7+
- PyTorch 1.5+
- CUDA 9.2+
- GCC 5+
- mmcv 1.3.12+
- mmdet 2.16.0+
- mmcls 0.15.0+

Compatible MMCV, MMClassification and MMDetection versions are shown as below. Please install the correct version of them to avoid installation issues.

Note: You need to run `pip uninstall mmcv` first if you have mmcv installed. If mmcv and mmcv-full are both installed, there will be `ModuleNotFoundError`.

CHAPTER

TWO

INSTALLATION

2.1 Prepare environment

1. Create a conda virtual environment and activate it.

```
conda create -n openmmlab python=3.7 -y
conda activate openmmlab
```

2. Install PyTorch and torchvision following the official instructions, e.g.,

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

Note: Make sure that your compilation CUDA version and runtime CUDA version match. You can check the supported CUDA version for precompiled packages on the PyTorch website.

`E.g` If you have CUDA 10.1 installed under `/usr/local/cuda` and would like to install PyTorch 1.7, you need to install the prebuilt PyTorch with CUDA 10.1.

```
conda install pytorch==1.7.0 torchvision==0.8.0 torchaudio==0.7.0 cudatoolkit=10.1 -
↪c pytorch
```

2.2 Install MMFewShot

It is recommended to install MMFewShot with MIM, which automatically handle the dependencies of OpenMMLab projects, including mmcv and other python packages.

```
pip install openmim
mim install mmfewshot
```

Or you can still install MMFewShot manually:

1. Install mmcv-full.

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

Please replace `{cu_version}` and `{torch_version}` in the url to your desired one. For example, to install the latest `mmcv-full` with `CUDA 11.0` and `PyTorch 1.7.0`, use the following command:

```
pip install mmcv-full -f https://download.openmmlab.com/mmcv/dist/cu110/torch1.7.0/
↪index.html
```

See here for different versions of MMCV compatible to different PyTorch and CUDA versions.

Optionally you can compile mmcv from source if you need to develop both mmcv and mmdet. Refer to the guide for details.

2. Install MMClassification and MMDetection.

   You can simply install mmclassification and mmdetection with the following command:

```
pip install mmcls mmdet
```

3. Install MMFewShot.

   You can simply install mmfewshot with the following command:

```
pip install mmfewshot
```

   or clone the repository and then install it:

```
git clone https://github.com/open-mmlab/mmfewshot.git
cd mmfewshot
pip install -r requirements/build.txt
pip install -v -e .  # or "python setup.py develop"
```

Note:

a. When specifying `-e` or `develop`, MMFewShot is installed on dev mode , any local modifications made to the code will take effect without reinstallation.

b. If you would like to use `opencv-python-headless` instead of `opencv-python`, you can install it before installing MMCV.

c. Some dependencies are optional. Simply running `pip install -v -e .` will only install the minimum runtime requirements. To use optional dependencies like `albumentations` and `imagecorruptions` either install them manually with `pip install -r requirements/optional.txt` or specify desired extras when calling `pip` (e.g. `pip install -v -e .[optional]`). Valid keys for the extras field are: `all`, `tests`, `build`, and `optional`.

2.3 Another option: Docker Image

We provide a Dockerfile to build an image. Ensure that you are using docker version >=19.03.

```
# build an image with PyTorch 1.6, CUDA 10.1
docker build -t mmfewshot docker/
```

Run it with

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

2.4 A from-scratch setup script

Assuming that you already have CUDA 10.1 installed, here is a full script for setting up MMDetection with conda.

```
conda create -n openmmlab python=3.7 -y
conda activate openmmlab

conda install pytorch==1.7.0 torchvision==0.8.0 torchaudio==0.7.0 cudatoolkit=10.1 -c␣
↪pytorch

# install the latest mmcv
pip install mmcv-full -f https://download.openmmlab.com/mmcv/dist/cu101/torch1.7.0/index.
↪html

# install mmclassification mmdetection
pip install mmcls mmdet

# install mmfewshot
git clone https://github.com/open-mmlab/mmfewshot.git
cd mmfewshot
pip install -r requirements/build.txt
pip install -v -e .  # or "python setup.py develop"
```

CHAPTER

THREE

VERIFICATION

To verify whether MMFewShot is installed correctly, we can run the demo code and inference a demo image.

Please refer to few shot classification demo or few shot detection demo for more details. The demo code is supposed to run successfully upon you finish the installation.

CHAPTER

FOUR

TEST A MODEL

- single GPU
- single node multiple GPU
- multiple node

You can use the following commands to infer a dataset.

```
# single-gpu
python tools/test.py ${CONFIG_FILE} ${CHECKPOINT_FILE} [optional arguments]

# multi-gpu
./tools/dist_test.sh ${CONFIG_FILE} ${CHECKPOINT_FILE} ${GPU_NUM} [optional arguments]

# multi-node in slurm environment
python tools/test.py ${CONFIG_FILE} ${CHECKPOINT_FILE} [optional arguments] --launcher␣
↪slurm
```

Examples:

For classification, inference Baseline on CUB under 5way 1shot setting.

```
python ./tools/classification/test.py \
  configs/classification/baseline/cub/baseline_conv4_1xb64_cub_5way-1shot.py \
  checkpoints/SOME_CHECKPOINT.pth
```

For detection, inference TFA on VOC split1 1shot setting.

```
python ./tools/detection/test.py \
  configs/detection/tfa/voc/split1/tfa_r101_fpn_voc-split1_1shot-fine-tuning.py \
  checkpoints/SOME_CHECKPOINT.pth --eval mAP
```

CHAPTER

FIVE

TRAIN A MODEL

5.1 Train with a single GPU

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

If you want to specify the working directory in the command, you can add an argument `--work_dir ${YOUR_WORK_DIR}`.

5.2 Train with multiple GPUs

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

Optional arguments are:

- `--no-validate` (**not suggested**): By default, the codebase will perform evaluation during the training. To disable this behavior, use `--no-validate`.
- `--work-dir ${WORK_DIR}`: Override the working directory specified in the config file.
- `--resume-from ${CHECKPOINT_FILE}`: Resume from a previous checkpoint file.

Difference between `resume-from` and `load-from`: `resume-from` loads both the model weights and optimizer status, and the epoch is also inherited from the specified checkpoint. It is usually used for resuming the training process that is interrupted accidentally. `load-from` only loads the model weights and the training epoch starts from 0. It is usually used for finetuning.

5.3 Train with multiple machines

If you run MMClassification on a cluster managed with slurm, you can use the script `slurm_train.sh`. (This script also supports single machine training.)

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

You can check slurm_train.sh for full arguments and environment variables.

If you have just multiple machines connected with ethernet, you can refer to PyTorch launch utility. Usually it is slow if you do not have high speed networking like InfiniBand.

5.4 Launch multiple jobs on a single machine

If you launch multiple jobs on a single machine, e.g., 2 jobs of 4-GPU training on a machine with 8 GPUs, you need to specify different ports (29500 by default) for each job to avoid communication conflict.

If you use `dist_train.sh` to launch training jobs, you can set the port in commands.

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

If you use launch training jobs with Slurm, you need to modify the config files (usually the 6th line from the bottom in config files) to set different communication ports.

In `config1.py`,

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

In `config2.py`,

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

Then you can launch two jobs with `config1.py` ang `config2.py`.

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

CHAPTER

SIX

MODEL ZOO

6.1 Few Shot Classification Model Zoo

6.1.1 Baseline

Please refer to Baseline for details.

6.1.2 Baseline++

Please refer to Baseline++ for details.

6.1.3 ProtoNet

Please refer to ProtoNet for details.

6.1.4 RelationNet

Please refer to RelationNet for details.

6.1.5 MatchingNet

Please refer to MatchingNet for details.

6.1.6 MAML

Please refer to MAML for details.

6.1.7 NegMargin

Please refer to NegMargin for details.

6.1.8 Meta Baseline

Please refer to Meta Baseline for details.

6.2 Few Shot Detection Model Zoo

6.2.1 TFA

Please refer to TFA for details.

6.2.2 FSCE

Please refer to FSCE for details.

6.2.3 Meta RCNN

Please refer to Meta RCNN for details.

6.2.4 FSDetView

Please refer to FSDetView for details.

6.2.5 Attention RPN

Please refer to Attention RPN for details.

6.2.6 MPSR

Please refer to MPSR for details.

CHAPTER

SEVEN

TUTORIAL 0: OVERVIEW OF MMFEWSHOT CLASSIFICATION

The main difference between general classification task and few shot classification task is the data usage. Therefore, the design of MMFewShot target at data sampling, meta test and models apis for few shot setting based on mmcls. Additionally, the modules in mmcls can be imported and reused in the code or config.

7.1 Design of data sampling

In MMFewShot, we suggest customizing the data pipeline using a dataset wrapper and modify the arguments in forward function when returning the dict with customize keys.

```
class CustomizeDataset:

    def __init__(self, dataset, ...):
        self.dataset = dataset
        self.customize_list = generate_function(dataset)

    def generate_function(self, dataset):
        pass

    def __getitem__(self, idx):
        return {
            'support_data': [self.dataset[i] for i in self.customize_list],
            'query_data': [self.dataset[i] for i in self.customize_list]
        }
```

More details can refer to Tutorial 2: Adding New Dataset

7.2 Design of model APIs

Each model in MMFewShot should implement following functions to support meta testing. More details can refer to Tutorial 3: Customize Models

```
@CLASSIFIERS.register_module()
class BaseFewShotClassifier(BaseModule):

    def forward(self, mode, ...):
        if mode == 'train':
            return self.forward_train(...)
```

(continues on next page)

(continued from previous page)

```
        elif mode == 'query':
            return self.forward_query(...)
        elif mode == 'support':
            return self.forward_support(...)
        ...

    def forward_train(self, **kwargs):
        pass

    # --------- for meta testing ----------
    def forward_support(self, **kwargs):
        pass

    def forward_query(self, **kwargs):
        pass

    def before_meta_test(self, meta_test_cfg, **kwargs):
        pass

    def before_forward_support(self, **kwargs):
        pass

    def before_forward_query(self, **kwargs):
        pass
```

7.3 Design of meta testing

Meta testing performs prediction on random sampled tasks multiple times. Each task contains support and query data. More details can refer to `mmfewshot/classification/apis/test.py`. Here is the basic pipeline for meta testing:

```
# the model may from training phase and may generate or fine-tine weights
1. Copy model
# prepare for the meta test (generate or freeze weights)
2. Call model.before_meta_test()
# some methods with fixed backbone can pre-compute the features for acceleration
3. Extracting features of all images for acceleration(optional)
# test different random sampled tasks
4. Test tasks (loop)
    # make sure all the task share the same initial weight
    a. Copy model
    # prepare model for support data
    b. Call model.before_forward_support()
    # fine-tune or none fine-tune models with given support data
    c. Forward support data: model(*data, mode='support')
    # prepare model for query data
    d. Call model.before_forward_query()
    # predict results of query data
    e. Forward query data: model(*data, mode='query')
```

7.3.1 meta testing on multiple gpus

In MMFewShot, we also support multi-gpu meta testing during validation or testing phase. In multi-gpu meta testing, the model will be copied and wrapped with `MetaTestParallel`, which will send data to the device of model. Thus, the original model will not be affected by the operations in Meta Testing. More details can refer to `mmfewshot/classification/utils/meta_test_parallel.py` Specifically, each gpu will be assigned with (num_test_tasks / world_size) task. Here is the distributed logic for multi gpu meta testing:

```
sub_num_test_tasks = num_test_tasks // world_size
sub_num_test_tasks += 1 if num_test_tasks % world_size != 0 else 0
for i in range(sub_num_test_tasks):
    task_id = (i * world_size + rank)
    if task_id >= num_test_tasks:
        continue
    # test task with task_id
    ...
```

If user want to customize the way to test a task, more details can refer to Tutorial 4: Customize Runtime Settings

CHAPTER

EIGHT

TUTORIAL 1: LEARN ABOUT CONFIGS

We incorporate modular and inheritance design into our config system, which is convenient to conduct various experiments. If you wish to inspect the config file, you may run `python tools/misc/print_config.py /PATH/TO/CONFIG` to see the complete config. The classification part of mmfewshot is built upon the mmcls, thus it is highly recommended learning the basic of mmcls.

8.1 Modify config through script arguments

When submitting jobs using "tools/classification/train.py" or "tools/classification/test.py", you may specify `--cfg-options` to in-place modify the config.

- Update config keys of dict chains.

  The config options can be specified following the order of the dict keys in the original config. For example, `--cfg-options model.backbone.norm_eval=False` changes the all BN modules in model backbones to `train` mode.

- Update keys inside a list of configs.

  Some config dicts are composed as a list in your config. For example, the training pipeline `data.train.pipeline` is normally a list e.g. `[dict(type='LoadImageFromFile'), ...]`. If you want to change `'LoadImageFromFile'` to `'LoadImageFromWebcam'` in the pipeline, you may specify `--cfg-options data.train.pipeline.0.type=LoadImageFromWebcam`.

- Update values of list/tuples.

  If the value to be updated is a list or a tuple. For example, the config file normally sets `workflow=[('train', 1)]`. If you want to change this key, you may specify `--cfg-options workflow="[(train,1),(val,1)]"`. Note that the quotation mark " is necessary to support list/tuple data types, and that **NO** white space is allowed inside the quotation marks in the specified value.

8.2 Config name style

We follow the below style to name config files. Contributors are advised to follow the same style.

```
{algorithm}_[algorithm setting]_{backbone}_{gpu x batch_per_gpu}_[misc]_{dataset}_{meta␣
↪test setting}.py
```

`{xxx}` is required field and `[yyy]` is optional.

- `{algorithm}`: model type like `faster_rcnn`, `mask_rcnn`, etc.

- [algorithm setting]: specific setting for some model, like without_semantic for htc, moment for reppoints, etc.
- {backbone}: backbone type like conv4, resnet12.
- [norm_setting]: bn (Batch Normalization) is used unless specified, other norm layer type could be gn (Group Normalization), syncbn (Synchronized Batch Normalization). gn-head/gn-neck indicates GN is applied in head/neck only, while gn-all means GN is applied in the entire model, e.g. backbone, neck, head.
- [gpu x batch_per_gpu]: GPUs and samples per GPU. For episodic training methods we use the total number of images in one episode, i.e. n classes x (support images+query images).
- [misc]: miscellaneous setting/plugins of model.
- {dataset}: dataset like cub, mini-imagenet and tiered-imagenet.
- {meta test setting}: n way k shot setting like 5way_1shot or 5way_5shot.

8.3 An example of Baseline

To help the users have a basic idea of a complete config and the modules in a modern classification system, we make brief comments on the config of Baseline for MiniImageNet in 5 way 1 shot setting as the following. For more detailed usage and the corresponding alternative for each module, please refer to the API documentation.

```
# config of model
model = dict(
    # classifier name
    type='Baseline',
    # config of backbone
    backbone=dict(type='Conv4'),
    # config of classifier head
    head=dict(type='LinearHead', num_classes=64, in_channels=1600),
    # config of classifier head used in meta test
    meta_test_head=dict(type='LinearHead', num_classes=5, in_channels=1600))

# data pipeline for training
train_pipeline = [
    # first pipeline to load images from file path
    dict(type='LoadImageFromFile'),
    # random resize crop
    dict(type='RandomResizedCrop', size=84),
    # random flip
    dict(type='RandomFlip', flip_prob=0.5, direction='horizontal'),
    # color jitter
    dict(type='ColorJitter', brightness=0.4, contrast=0.4, saturation=0.4),
    dict(type='Normalize',  # normalization
         # mean values used to normalization
         mean=[123.675, 116.28, 103.53],
         # standard variance used to normalization
         std=[58.395, 57.12, 57.375],
         # whether to invert the color channel, rgb2bgr or bgr2rgb
         to_rgb=True),
    # convert img into torch.Tensor
    dict(type='ImageToTensor', keys=['img']),
    # convert gt_label into torch.Tensor
```

(continues on next page)

(continued from previous page)

```
    dict(type='ToTensor', keys=['gt_label']),
    # pipeline that decides which keys in the data should be passed to the runner
    dict(type='Collect', keys=['img', 'gt_label'])
]

# data pipeline for testing
test_pipeline = [
    # first pipeline to load images from file path
    dict(type='LoadImageFromFile'),
    # resize image
    dict(type='Resize', size=(96, -1)),
    # center crop
    dict(type='CenterCrop', crop_size=84),
    dict(type='Normalize',  # normalization
         # mean values used to normalization
         mean=[123.675, 116.28, 103.53],
         # standard variance used to normalization
         std=[58.395, 57.12, 57.375],
         # whether to invert the color channel, rgb2bgr or bgr2rgb
         to_rgb=True),
    # convert img into torch.Tensor
    dict(type='ImageToTensor', keys=['img']),
    # pipeline that decides which keys in the data should be passed to the runner
    dict(type='Collect', keys=['img', 'gt_label'])
]

# config of fine-tuning using support set in Meta Test
meta_finetune_cfg = dict(
    # number of iterations in fine-tuning
    num_steps=150,
    # optimizer config in fine-tuning
    optimizer=dict(
        type='SGD',  # optimizer name
        lr=0.01,  # learning rate
        momentum=0.9,  # momentum
        dampening=0.9,  # dampening
        weight_decay=0.001)),  # weight decay

data = dict(
    # batch size of a single GPU
    samples_per_gpu=64,
    # worker to pre-fetch data for each single GPU
    workers_per_gpu=4,
    # config of training set
    train=dict(
        # name of dataset
        type='MiniImageNetDataset',
        # prefix of image
        data_prefix='data/mini_imagenet',
        # subset of dataset
        subset='train',
        # train pipeline
```

(continues on next page)

(continued from previous page)

```
        pipeline=train_pipeline),
    # config of validation set
    val=dict(
        # dataset wrapper for Meta Test
        type='MetaTestDataset',
        # total number of test tasks
        num_episodes=100,
        num_ways=5,  # number of class in each task
        num_shots=1,  # number of support images in each task
        num_queries=15,  # number of query images in each task
        dataset=dict(  # config of dataset
            type='MiniImageNetDataset',  # dataset name
            subset='val',  # subset of dataset
            data_prefix='data/mini_imagenet',  # prefix of images
            pipeline=test_pipeline),
        meta_test_cfg=dict(  # config of Meta Test
            num_episodes=100,  # total number of test tasks
            num_ways=5,  # number of class in each task
            # whether to pre-compute features from backbone for acceleration
            fast_test=True,
            # dataloader setting for feature extraction of fast test
            test_set=dict(batch_size=16, num_workers=2),
            support=dict(  # support set setting in meta test
                batch_size=4,  # batch size for fine-tuning
                num_workers=0,  # number of worker set 0 since the only 5 images
                drop_last=True,  # drop last
                train=dict(  # config of fine-tuning
                    num_steps=150,  # number of steps in fine-tuning
                    optimizer=dict(  # optimizer config in fine-tuning
                        type='SGD',  # optimizer name
                        lr=0.01,  # learning rate
                        momentum=0.9,  # momentum
                        dampening=0.9,  # dampening
                        weight_decay=0.001))),  # weight decay
            # query set setting predict 75 images
            query=dict(batch_size=75, num_workers=0))),
    test=dict(  # used for model validation in Meta Test fashion
        type='MetaTestDataset',  # dataset wrapper for Meta Test
        num_episodes=2000,  # total number of test tasks
        num_ways=5,  # number of class in each task
        num_shots=1,  # number of support images in each task
        num_queries=15,  # number of query images in each task
        dataset=dict(  # config of dataset
            type='MiniImageNetDataset',  # dataset name
            subset='test',  # subset of dataset
            data_prefix='data/mini_imagenet',  # prefix of images
            pipeline=test_pipeline),
        meta_test_cfg=dict(  # config of Meta Test
            num_episodes=2000,  # total number of test tasks
            num_ways=5,  # number of class in each task
            # whether to pre-compute features from backbone for acceleration
            fast_test=True,
```

(continues on next page)

(continued from previous page)

```
            # dataloader setting for feature extraction of fast test
            test_set=dict(batch_size=16, num_workers=2),
            support=dict(  # support set setting in meta test
                batch_size=4,  # batch size for fine-tuning
                num_workers=0,  # number of worker set 0 since the only 5 images
                drop_last=True,  # drop last
                train=dict(  # config of fine-tuning
                    num_steps=150,  # number of steps in fine-tuning
                    optimizer=dict(  # optimizer config in fine-tuning
                        type='SGD',  # optimizer name
                        lr=0.01,  # learning rate
                        momentum=0.9,  # momentum
                        dampening=0.9,  # dampening
                        weight_decay=0.001))),  # weight decay
            # query set setting predict 75 images
            query=dict(batch_size=75, num_workers=0))))
log_config = dict(
    interval=50,  # interval to print the log
    hooks=[dict(type='TextLoggerHook')])
checkpoint_config = dict(interval=20)  # interval to save a checkpoint
evaluation = dict(
    by_epoch=True,  # eval model by epoch
    metric='accuracy',  # Metrics used during evaluation
    interval=5)  # interval to eval model
# parameters to setup distributed training, the port can also be set.
dist_params = dict(backend='nccl')
log_level = 'INFO'  # the output level of the log.
load_from = None  # load a pre-train checkpoints
# resume checkpoints from a given path, the training will be resumed from
# the epoch when the checkpoint's is saved.
resume_from = None
# workflow for runner. [('train', 1)] means there is only one workflow and
# the workflow named 'train' is executed once.
workflow = [('train', 1)]
pin_memory = True  # whether to use pin memory
# whether to use infinite sampler; infinite sampler can accelerate training efficient
use_infinite_sampler = True
seed = 0  # random seed
runner = dict(type='EpochBasedRunner', max_epochs=200)  # runner type and epochs of␣
↪training
optimizer = dict(  # the configuration file used to build the optimizer, support all␣
↪optimizers in PyTorch.
    type='SGD',  # optimizer type
    lr=0.05,  # learning rat
    momentum=0.9,  # momentum
    weight_decay=0.0001)  # weight decay of SGD
optimizer_config = dict(grad_clip=None)  # most of the methods do not use gradient clip
lr_config = dict(
    # the policy of scheduler, also support CosineAnnealing, Cyclic, etc. Refer to␣
↪details of supported LrUpdater
    # from https://github.com/open-mmlab/mmcv/blob/master/mmcv/runner/hooks/lr_updater.py
↪#L9.
```

(continues on next page)

(continued from previous page)

```
    policy='step',
    warmup='linear',  # warmup type
    warmup_iters=3000,  # warmup iterations
    warmup_ratio=0.25,  # warmup ratio
    step=[60, 120])  # Steps to decay the learning rate
```

8.4 FAQ

8.4.1 Use intermediate variables in configs

Some intermediate variables are used in the configuration file. The intermediate variables make the configuration file clearer and easier to modify.

For example, `train_pipeline` / `test_pipeline` is the intermediate variable of the data pipeline. We first need to define `train_pipeline` / `test_pipeline`, and then pass them to `data`. If you want to modify the size of the input image during training and testing, you need to modify the intermediate variables of `train_pipeline` / `test_pipeline`.

```
img_norm_cfg = dict(
    mean=[123.675, 116.28, 103.53], std=[58.395, 57.12, 57.375], to_rgb=True)
train_pipeline = [
    dict(type='LoadImageFromFile'),
    dict(type='RandomResizedCrop', size=384, backend='pillow',),
    dict(type='RandomFlip', flip_prob=0.5, direction='horizontal'),
    dict(type='Normalize', **img_norm_cfg),
    dict(type='ImageToTensor', keys=['img']),
    dict(type='ToTensor', keys=['gt_label']),
    dict(type='Collect', keys=['img', 'gt_label'])
]
test_pipeline = [
    dict(type='LoadImageFromFile'),
    dict(type='Resize', size=384, backend='pillow'),
    dict(type='Normalize', **img_norm_cfg),
    dict(type='ImageToTensor', keys=['img']),
    dict(type='Collect', keys=['img'])
]
data = dict(
    train=dict(pipeline=train_pipeline),
    val=dict(pipeline=test_pipeline),
    test=dict(pipeline=test_pipeline))
```

8.4.2 Ignore some fields in the base configs

Sometimes, you need to set `_delete_=True` to ignore some domain content in the basic configuration file. You can refer to mmcv for more instructions.

The following is an example. If you want to use cosine schedule, just using inheritance and directly modify it will report `get unexcepected keyword'step'` error, because the `'step'` field of the basic config in `lr_config` domain information is reserved, and you need to add `_delete_ =True` to ignore the content of `lr_config` related fields in the basic configuration file:

```
lr_config = dict(
    _delete_=True,
    policy='CosineAnnealing',
    min_lr=0,
    warmup='linear',
    by_epoch=True,
    warmup_iters=5,
    warmup_ratio=0.1
)
```

CHAPTER

NINE

TUTORIAL 2: ADDING NEW DATASET

9.1 Customize datasets by reorganizing data

9.1.1 Customize loading annotations

You can write a new Dataset class inherited from `BaseFewShotDataset`, and overwrite `load_annotations(self)`, like CUB and MiniImageNet. Typically, this function returns a list, where each sample is a dict, containing necessary data information, e.g., `img` and `gt_label`.

Assume we are going to implement a `Filelist` dataset, which takes filelists for both training and testing. The format of annotation list is as follows:

```
000001.jpg 0
000002.jpg 1
```

We can create a new dataset in `mmfewshot/classification/datasets/filelist.py` to load the data.

```python
import mmcv
import numpy as np

from mmcls.datasets.builder import DATASETS
from .base import BaseFewShotDataset


@DATASETS.register_module()
class Filelist(BaseFewShotDataset):

    def load_annotations(self):
        assert isinstance(self.ann_file, str)

        data_infos = []
        with open(self.ann_file) as f:
            samples = [x.strip().split(' ') for x in f.readlines()]
            for filename, gt_label in samples:
                info = {'img_prefix': self.data_prefix}
                info['img_info'] = {'filename': filename}
                info['gt_label'] = np.array(gt_label, dtype=np.int64)
                data_infos.append(info)
            return data_infos
```

And add this dataset class in `mmcls/datasets/__init__.py`

```
from .base_dataset import BaseDataset
...
from .filelist import Filelist

__all__ = [
    'BaseDataset', ... ,'Filelist'
]
```

Then in the config, to use `Filelist` you can modify the config as the following

```
train = dict(
    type='Filelist',
    ann_file = 'image_list.txt',
    pipeline=train_pipeline
)
```

9.1.2 Customize different subsets

To support different subset, we first predefine the classes of different subsets. Then we modify `get_classes` to handle different classes of subset.

```
import mmcv
import numpy as np

from mmcls.datasets.builder import DATASETS
from .base import BaseFewShotDataset

@DATASETS.register_module()
class Filelist(BaseFewShotDataset):

    TRAIN_CLASSES = ['train_a', ...]
    VAL_CLASSES = ['val_a', ...]
    TEST_CLASSES = ['test_a', ...]

    def __init__(self, subset, *args, **kwargs):
        ...
        self.subset = subset
        super().__init__(*args, **kwargs)

    def get_classes(self):
        if self.subset == 'train':
            class_names = self.TRAIN_CLASSES
        ...
        return class_names
```

9.2 Customize datasets sampling

9.2.1 EpisodicDataset

We use `EpisodicDataset` as wrapper to perform N way K shot sampling. For example, suppose the original dataset is Dataset_A, the config looks like the following

```
dataset_A_train = dict(
        type='EpisodicDataset',
        num_episodes=100000, # number of total episodes = length of dataset wrapper
        # each call of `__getitem__` will return
        # {'support_data': [(num_ways * num_shots) images],
        #  'query_data': [(num_ways * num_queries) images]}
        num_ways=5, # number of way (different classes)
        num_shots=5, # number of support shots of each class
        num_queries=5, # number of query shots of each class
        dataset=dict(  # This is the original config of Dataset_A
            type='Dataset_A',
            ...
            pipeline=train_pipeline
        )
    )
```

9.2.2 Customize sampling logic

An example of customizing data sampling logic for training:

Create a new dataset wrapper

We can create a new dataset wrapper in mmfewshot/classification/datasets/dataset_wrappers.py to customize sampling logic.

```
class MyDatasetWrapper:
    def __init__(self, dataset, args_a, args_b, ...):
        self.dataset = dataset
        ...
        self.episode_idxes = self.generate_episodic_idxes()

    def generate_episodic_idxes(self):
        episode_idxes = []
        # sampling each episode
        for _ in range(self.num_episodes):
            episodic_a_idx, episodic_b_idx, episodic_c_idx= [], [], []
            # customize sampling logic
            # select the index of data_infos from original dataset
            ...
            episode_idxes.append({
                'a': episodic_a_idx,
                'b': episodic_b_idx,
                'c': episodic_c_idx,
            })
```

(continues on next page)

(continued from previous page)

```
        return episode_idxes

    def __getitem__(self, idx):
        # the key can be any value, but it needs to modify the code
        # in the forward function of model.
        return {
            'a_data' : [self.dataset[i] for i in self.episode_idxes[idx]['a']],
            'b_data' : [self.dataset[i] for i in self.episode_idxes[idx]['b']],
            'c_data' : [self.dataset[i] for i in self.episode_idxes[idx]['c']]
        }
```

Update dataset builder

We need to add the build code in mmfewshot/classification/datasets/builder.py for our customize dataset wrapper.

```
def build_dataset(cfg, default_args=None):
    if isinstance(cfg, (list, tuple)):
        dataset = ConcatDataset([build_dataset(c, default_args) for c in cfg])
    ...
    elif cfg['type'] == 'MyDatasetWrapper':
        dataset = MyDatasetWrapper(
            build_dataset(cfg['dataset'], default_args),
            # pass customize arguments
            args_a=cfg['args_a'],
            args_b=cfg['args_b'],
            ...)
    else:
        dataset = build_from_cfg(cfg, DATASETS, default_args)

    return dataset
```

Update the arguments in model

The argument names in forward function need to be consistent with the customize dataset wrapper.

```
class MyClassifier(BaseFewShotClassifier):
    ...
    def forward(self, a_data=None, b_data=None, c_data=None, ...):
        # pass the modified arguments name.
        if mode == 'train':
            return self.forward_train(a_data=a_data, b_data=b_data, c_data=None,␣
↪**kwargs)
        elif mode == 'query':
            return self.forward_query(img=img, **kwargs)
        elif mode == 'support':
            return self.forward_support(img=img, **kwargs)
        elif mode == 'extract_feat':
            return self.extract_feat(img=img)
        else:
            raise ValueError()
```

Using customize dataset wrapper in config

Then in the config, to use `MyDatasetWrapper` you can modify the config as the following,

```
dataset_A_train = dict(
        type='MyDatasetWrapper',
        args_a=None,
        args_b=None,
        dataset=dict(  # This is the original config of Dataset_A
            type='Dataset_A',
            ...
            pipeline=train_pipeline
        )
    )
```

CHAPTER

TEN

TUTORIAL 3: CUSTOMIZE MODELS

10.1 Add a new classifier

Here we show how to develop a new classifier with an example as follows

10.1.1 1. Define a new classifier

Create a new file `mmfewshot/classification/models/classifiers/my_classifier.py`.

```
from mmcls.models.builder import CLASSIFIERS
from .base import BaseFewShotClassifier

@CLASSIFIERS.register_module()
class MyClassifier(BaseFewShotClassifier):

    def __init__(self, arg1, arg2):
        pass

    # customize input for different mode
    # the input should keep consistent with the dataset
    def forward(self, img, mode='train',**kwargs):
        if mode == 'train':
            return self.forward_train(img=img, **kwargs)
        elif mode == 'query':
            return self.forward_query(img=img,  **kwargs)
        elif mode == 'support':
            return self.forward_support(img=img, **kwargs)
        elif mode == 'extract_feat':
            assert img is not None
            return self.extract_feat(img=img)
        else:
            raise ValueError()

    # customize forward function for training data
    def forward_train(self, img, gt_label, **kwargs):
        pass

    # customize forward function for meta testing support data
    def forward_support(self, img, gt_label, **kwargs):
        pass
```

(continues on next page)

(continued from previous page)

```
    # customize forward function for meta testing query data
    def forward_query(self, img):
        pass

    # prepare meta testing
    def before_meta_test(self, meta_test_cfg, **kwargs):
        pass

    # prepare forward meta testing query images
    def before_forward_support(self, **kwargs):
        pass

    # prepare forward meta testing support images
    def before_forward_query(self, **kwargs):
        pass
```

10.1.2 2. Import the module

You can either add the following line to `mmfewshot/classification/models/heads/__init__.py`

```
from .my_classifier import MyClassifier
```

or alternatively add

```
custom_imports = dict(
    imports=['mmfewshot.classification.models.classifier.my_classifier'],
    allow_failed_imports=False)
```

to the config file to avoid modifying the original code.

10.1.3 3. Use the classifier in your config file

```
model = dict(
    type="MyClassifier",
    ...
)
```

10.2 Add a new backbone

Here we show how to develop a new backbone with an example as follows

10.2.1 1. Define a new backbone

Create a new file `mmfewshot/classification/models/backbones/mynet.py`.

```
import torch.nn as nn

from mmcls.models.builder import BACKBONES

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

    def __init__(self, arg1, arg2):
        pass

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

10.2.2 2. Import the module

You can either add the following line to `mmfewshot/classification/models/backbones/__init__.py`

```
from .mynet import MyNet
```

or alternatively add

```
custom_imports = dict(
    imports=['mmfewshot.classification.models.backbones.mynet'],
    allow_failed_imports=False)
```

to the config file to avoid modifying the original code.

10.2.3 3. Use the backbone in your config file

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

10.3 Add new heads

Here we show how to develop a new head with an example as follows

10.3.1 1. Define a new head

Create a new file `mmfewshot/classification/models/heads/myhead.py`.

```
from mmcls.models.builder import HEADS
from .base_head import BaseFewShotHead

@HEADS.register_module()
class MyHead(BaseFewShotHead):

    def __init__(self, arg1, arg2) -> None:
        pass

    def forward_train(self, x, gt_label, **kwargs):
        pass

    def forward_support(self, x, gt_label, **kwargs):
        pass

    def forward_query(self, x, **kwargs):
        pass

    def before_forward_support(self) -> None:
        pass

    def before_forward_query(self) -> None:
        pass
```

10.3.2 2. Import the module

You can either add the following line to `mmfewshot/classification/models/heads/__init__.py`

```
from .myhead import MyHead
```

or alternatively add

```
custom_imports = dict(
    imports=['mmfewshot.classification.models.backbones.myhead'],
    allow_failed_imports=False)
```

to the config file to avoid modifying the original code.

10.3.3 3. Use the head in your config file

```
model = dict(
    ...
    head=dict(
        type='MyHead',
        arg1=xxx,
        arg2=xxx),
    ...
```

10.4 Add new loss

To add a new loss function, the users need implement it in `mmfewshot/classification/models/losses/my_loss.py`. The decorator `weighted_loss` enable the loss to be weighted for each element.

```
import torch
import torch.nn as nn

from ..builder import LOSSES
from .utils import weighted_loss

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

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

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

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

Then the users need to add it in the `mmfewshot/classification/models/losses/__init__.py`.

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

Alternatively, you can add

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

to the config file and achieve the same goal.

To use it, modify the `loss_xxx` field. Since MyLoss is for regression, you need to modify the `loss_bbox` field in the head.

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

CHAPTER

ELEVEN

TUTORIAL 4: CUSTOMIZE RUNTIME SETTINGS

11.1 Customize optimization settings

11.1.1 Customize an optimizer supported by Pytorch

We already support to use all the optimizers implemented by PyTorch, and the only modification is to change the `optimizer` field of config files. For example, if you want to use `ADAM` (note that the performance could drop a lot), the modification could be as the following.

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

To modify the learning rate of the model, the users only need to modify the `lr` in the config of optimizer. The users can directly set arguments following the API doc of PyTorch.

11.1.2 Customize self-implemented optimizer

1. Define a new optimizer

A customized optimizer could be defined as following.

Assume you want to add a optimizer named `MyOptimizer`, which has arguments `a`, `b`, and `c`. You need to create a new directory named `mmfewshot/classification/core/optimizer`. And then implement the new optimizer in a file, e.g., in `mmfewshot/classification/core/optimizer/my_optimizer.py`:

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


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

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

2. Add the optimizer to registry

To find the above module defined above, this module should be imported into the main namespace at first. There are two options to achieve it.

- Modify `mmfewshot/classification/core/optimizer/__init__.py` to import it.

  The newly defined module should be imported in `mmfewshot/classification/core/optimizer/__init__.py` so that the registry will find the new module and add it:

```
from .my_optimizer import MyOptimizer
```

- Use `custom_imports` in the config to manually import it

```
custom_imports = dict(imports=['mmfewshot.classification.core.optimizer.my_optimizer'],␣
↪allow_failed_imports=False)
```

The module `mmfewshot.classification.core.optimizer.my_optimizer` will be imported at the beginning of the program and the class `MyOptimizer` is then automatically registered. Note that only the package containing the class `MyOptimizer` should be imported. `mmfewshot.classification.core.optimizer.my_optimizer.MyOptimizer` **cannot** be imported directly.

Actually users can use a totally different file directory structure using this importing method, as long as the module root can be located in `PYTHONPATH`.

3. Specify the optimizer in the config file

Then you can use `MyOptimizer` in `optimizer` field of config files. In the configs, the optimizers are defined by the field `optimizer` like the following:

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

To use your own optimizer, the field can be changed to

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

11.1.3 Customize optimizer constructor

Some models may have some parameter-specific settings for optimization, e.g. weight decay for BatchNorm layers. The users can do those fine-grained parameter tuning through customizing optimizer constructor.

```
from mmcv.utils import build_from_cfg

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


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

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

    def __call__(self, model):
```

(continues on next page)

(continued from previous page)

```
        return my_optimizer
```

The default optimizer constructor is implemented here, which could also serve as a template for new optimizer constructor.

11.1.4 Additional settings

Tricks not implemented by the optimizer should be implemented through optimizer constructor (e.g., set parameter-wise learning rates) or hooks. We list some common settings that could stabilize the training or accelerate the training. Feel free to create PR, issue for more settings.

- **Use gradient clip to stabilize training**: Some models need gradient clip to clip the gradients to stabilize the training process. An example is as below:

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

- **Use momentum schedule to accelerate model convergence**: We support momentum scheduler to modify model's momentum according to learning rate, which could make the model converge in a faster way. Momentum scheduler is usually used with LR scheduler, for example, the following config is used in 3D detection to accelerate convergence. For more details, please refer to the implementation of CyclicLrUpdater and CyclicMomentumUpdater.

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

11.2 Customize training schedules

By default we use step learning rate with 1x schedule, this calls StepLRHook in MMCV. We support many other learning rate schedule here, such as `CosineAnnealing` and `Poly` schedule. Here are some examples

- Poly schedule:

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

- ConsineAnnealing schedule:

```
lr_config = dict(
    policy='CosineAnnealing',
    warmup='linear',
    warmup_iters=1000,
```

(continues on next page)

(continued from previous page)

```
    warmup_ratio=1.0 / 10,
    min_lr_ratio=1e-5)
```

11.3 Customize workflow

Workflow is a list of (phase, epochs) to specify the running order and epochs. By default it is set to be

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

which means running 1 epoch for training. Sometimes user may want to check some metrics (e.g. loss, accuracy) about the model on the validate set. In such case, we can set the workflow as

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

so that 1 epoch for training and 1 epoch for validation will be run iteratively.

Note:

1. The parameters of model will not be updated during val epoch.
2. Keyword `total_epochs` in the config only controls the number of training epochs and will not affect the validation workflow.
3. Workflows `[('train', 1), ('val', 1)]` and `[('train', 1)]` will not change the behavior of `EvalHook` because `EvalHook` is called by `after_train_epoch` and validation workflow only affect hooks that are called through `after_val_epoch`. Therefore, the only difference between `[('train', 1), ('val', 1)]` and `[('train', 1)]` is that the runner will calculate losses on validation set after each training epoch.

11.4 Customize hooks

11.4.1 Customize self-implemented hooks

1. Implement a new hook

Here we give an example of creating a new hook in MMFewShot and using it in training.

```
from mmcv.runner import HOOKS, Hook


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

    def __init__(self, a, b):
        pass

    def before_run(self, runner):
        pass

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

(continues on next page)

(continued from previous page)

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

    def after_epoch(self, runner):
        pass

    def before_iter(self, runner):
        pass

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

Depending on the functionality of the hook, the users need to specify what the hook will do at each stage of the training in `before_run`, `after_run`, `before_epoch`, `after_epoch`, `before_iter`, and `after_iter`.

2. Register the new hook

Then we need to make `MyHook` imported. Assuming the file is in `mmfewshot/classification/core/utils/my_hook.py` there are two ways to do that:

- Modify `mmfewshot/core/utils/__init__.py` to import it.

  The newly defined module should be imported in `mmfewshot/classification/core/utils/__init__.py` so that the registry will find the new module and add it:

```
from .my_hook import MyHook
```

- Use `custom_imports` in the config to manually import it

```
custom_imports = dict(imports=['mmfewshot.classification.core.utils.my_hook'], allow_
↪failed_imports=False)
```

3. Modify the config

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

You can also set the priority of the hook by adding key `priority` to `'NORMAL'` or `'HIGHEST'` as below

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

By default the hook's priority is set as `NORMAL` during registration.

11.4.2 Use hooks implemented in MMCV

If the hook is already implemented in MMCV, you can directly modify the config to use the hook as below

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

11.4.3 Customize self-implemented eval hooks with a dataset

Here we give an example of creating a new hook in MMFewShot and using it to evaluate a dataset. To achieve this, we can add following code in `mmfewshot/classification/apis/test.py`.

```
if validate:
    ...
    # build dataset and dataloader
    my_eval_dataset = build_dataset(cfg.data.my_eval)
    my_eval_data_loader = build_dataloader(my_eval_dataset)
    runner.register_hook(eval_hook(my_eval_data_loader), priority='LOW')
```

The arguments used in `test_my_single_task` can be defined in `meta_test_cfg`, for example:

```
data = dict(
    test=dict(
        type='MetaTestDataset',
        ...,
        dataset=dict(...),
        meta_test_cfg=dict(
            ...,
            test_my_single_task=dict(arg1=...)
        )
    )
)
```

Then we can replace the `test_single_task` with customized `test_my_single_task` in `single_gpu_meta_test` and `multiple_gpu_meta_test`

11.4.4 Modify default runtime hooks

There are some common hooks that are not registered through `custom_hooks`, they are

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

In those hooks, only the logger hook has the `VERY_LOW` priority, others' priority are `NORMAL`. The above-mentioned tutorials already covers how to modify `optimizer_config`, `momentum_config`, and `lr_config`. Here we reveals how what we can do with `log_config`, `checkpoint_config`, and `evaluation`.

Checkpoint config

The MMCV runner will use `checkpoint_config` to initialize `CheckpointHook`.

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

The users could set `max_keep_ckpts` to only save only small number of checkpoints or decide whether to store state dict of optimizer by `save_optimizer`. More details of the arguments are here

Log config

The `log_config` wraps multiple logger hooks and enables to set intervals. Now MMCV supports `WandbLoggerHook`, `MlflowLoggerHook`, and `TensorboardLoggerHook`. The detail usages can be found in the doc.

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

Evaluation config

The config of `evaluation` will be used to initialize the `EvalHook`. Except the key `interval`, other arguments such as `metric` will be passed to the `dataset.evaluate()`

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

11.5 Customize meta testing

We already support two ways to handle the support data, fine-tuning and straight forwarding. To customize the code for a meta test task, we need to add a new function `test_my_single_task` in the `mmfewshot/classification/apis/test.py`.

```
def test_my_single_task(model: MetaTestParallel,
                        support_dataloader: DataLoader,
                        query_dataloader: DataLoader,
                        meta_test_cfg: Dict):
    # use copy of model for each task
    model = copy.deepcopy(model)
    ...
    # forward support set
    model.before_forward_support()
    # customize code for support data
    ...

    # forward query set
    model.before_forward_query()
    ...
    results_list, gt_label_list = [], []
```

(continues on next page)

(continued from previous page)

```
    # customize code for query data
    ...

    # return predict results and gt labels for evaluation
    return results_list, gt_labels
```

The arguments used in `test_my_single_task` can be defined in `meta_test_cfg`, for example:

```
data = dict(
    test=dict(
        type='MetaTestDataset',
        ...,
        dataset=dict(...),
        meta_test_cfg=dict(
            ...,
            test_my_single_task=dict(arg1=...)
        )
    )
)
```

Then we can replace the `test_single_task` with customized `test_my_single_task` in `single_gpu_meta_test` and `multiple_gpu_meta_test`

CHAPTER

TWELVE

TUTORIAL 0: OVERVIEW OF MMFEWSHOT DETECTION

The main difference between general classification task and few shot classification task is the data usage. Therefore, the design of MMFewShot targets at data flows for few shot setting based on mmdet. Additionally, the modules in mmdet can be imported and reused in the code or config.

12.1 Design of data flow

Since MMFewShot is built upon the mmdet, all the datasets in mmdet can be configured in the config file. If user want to use the dataset from mmdet, please refer to mmdet for more details.

In MMFewShot, there are three important components for fetching data:

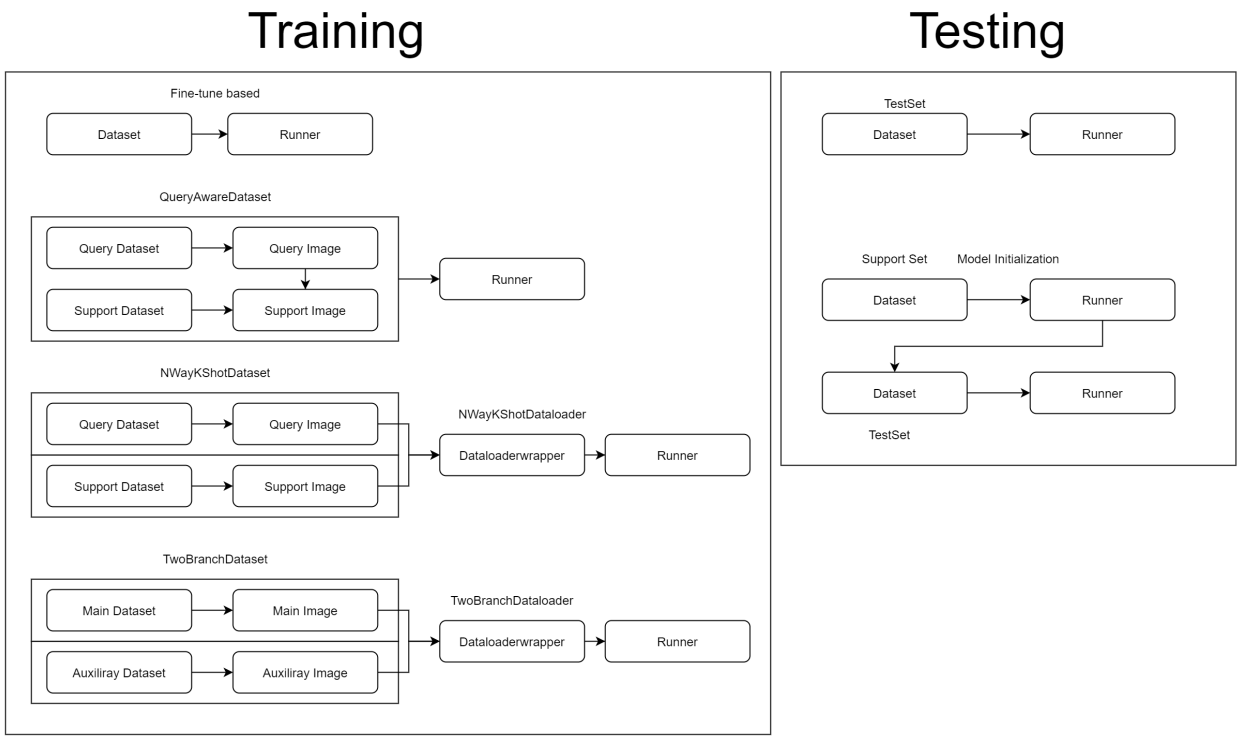
- Datasets: loading annotations from `ann_cfg` and filtering images and annotations for few shot setting.
- Dataset Wrappers: determining the sampling logic, such as sampling support images according to query image.
- Dataloader Wrappers: encapsulate the data from multiple datasets.

In summary, we currently support 4 different data flow for training:

- fine-tune based: it is the same as regular detection.
- query aware: it will return query data and support data from same dataset.
- n way k shot: it will first sample query data (regular) and support data (N way k shot) from separate datasets and then encapsulate them by dataloader wrapper.
- two branch: it will first sample main data (regular) and auxiliary data (regular) from separate datasets and then encapsulate them by dataloader wrapper.

For testing:

- regular testing: it is the same as regular detection.
- testing for query-support based detector: there will be a model initialization step before testing, it is implemented by `QuerySupportEvalHook`. More implementation details can refer to `mmfewshot.detection.core.evaluation.eval_hooks`



More usage details and customization can refer to Tutorial 2: Adding New Dataset

CHAPTER

THIRTEEN

TUTORIAL 1: LEARN ABOUT CONFIGS

We incorporate modular and inheritance design into our config system, which is convenient to conduct various experiments. If you wish to inspect the config file, you may run `python tools/misc/print_config.py /PATH/TO/CONFIG` to see the complete config. The detection part of mmfewshot is built upon the mmdet, thus it is highly recommended learning the basic of mmdet.

13.1 Modify a config through script arguments

When submitting jobs using "tools/train.py" or "tools/test.py", you may specify `--cfg-options` to in-place modify the config.

- Update config keys of dict chains.

  The config options can be specified following the order of the dict keys in the original config. For example, `--cfg-options model.backbone.norm_eval=False` changes the all BN modules in model backbones to `train` mode.

- Update keys inside a list of configs.

  Some config dicts are composed as a list in your config. For example, the training pipeline `data.train.pipeline` is normally a list e.g. `[dict(type='LoadImageFromFile'), ...]`. If you want to change `'LoadImageFromFile'` to `'LoadImageFromWebcam'` in the pipeline, you may specify `--cfg-options data.train.pipeline.0.type=LoadImageFromWebcam`.

- Update values of list/tuples.

  If the value to be updated is a list or a tuple. For example, the config file normally sets `workflow=[('train', 1)]`. If you want to change this key, you may specify `--cfg-options workflow="[(train,1),(val,1)]"`. Note that the quotation mark " is necessary to support list/tuple data types, and that **NO** white space is allowed inside the quotation marks in the specified value.

13.2 Config file naming convention

We follow the below style to name config files. Contributors are advised to follow the same style.

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

`{xxx}` is required field and `[yyy]` is optional.

- `{model}`: model type like `faster_rcnn`, `mask_rcnn`, etc.
- `[model setting]`: specific setting for some model, like `contrastive-loss` for `fsce`, etc.

- {backbone}: backbone type like r50 (ResNet-50), x101 (ResNeXt-101).
- {neck}: neck type like fpn, c4.
- [norm_setting]: bn (Batch Normalization) is used unless specified, other norm layer type could be gn (Group Normalization), syncbn (Synchronized Batch Normalization). gn-head/gn-neck indicates GN is applied in head/neck only, while gn-all means GN is applied in the entire model, e.g. backbone, neck, head.
- [misc]: miscellaneous setting/plugins of model, e.g. dconv, gcb, attention, albu, mstrain.
- [gpu x batch_per_gpu]: GPUs and samples per GPU, 8xb2 is used by default.
- {dataset}: dataset like coco, voc-split1, voc-split2 and voc-split3.
- {data setting}: like base-training or 1shot-fine-tuning.

13.3 An example of TFA

To help the users have a basic idea of a complete config and the modules in a modern classification system, we make brief comments on the config of TFA in coco 10 shot fine-tuning setting as the following. For more detailed usage and the corresponding alternative for each module, please refer to the API documentation.

```
train_pipeline = [  # Training pipeline
    # First pipeline to load images from file path
    dict(type='LoadImageFromFile'),
    # Second pipeline to load annotations for current image
    dict(type='LoadAnnotations', with_bbox=True),
    # Augmentation pipeline that resize the images and their annotations
    dict(type='Resize',
         # The multiple scales of image
         img_scale=[(1333, 640), (1333, 672), (1333, 704), (1333, 736),
                    (1333, 768), (1333, 800)],
         # whether to keep the ratio between height and width
         keep_ratio=True,
         # the scales will be sampled from img_scale
         multiscale_mode='value'),
    # RandomFlip config, flip_ratio: the ratio or probability to flip
    dict(type='RandomFlip', flip_ratio=0.5),
    # Image normalization config to normalize the input images
    dict(type='Normalize',
         # Mean values used to in pre-trained backbone models
         mean=[103.53, 116.28, 123.675],
         # Standard variance used to in pre-trained backbone models
         std=[1.0, 1.0, 1.0],
         # The channel orders of image used in pre-trained backbone models
         to_rgb=False),
    # Padding config, size_divisor: the number the padded images should be divisible
    dict(type='Pad', size_divisor=32),
    # Default format bundle to gather data in the pipeline
    dict(type='DefaultFormatBundle'),
    # Pipeline that decides which keys in the data should be passed to the detector
    dict(type='Collect', keys=['img', 'gt_bboxes', 'gt_labels'])
]
test_pipeline = [  # test pipeline
    # First pipeline to load images from file path
```

(continues on next page)

(continued from previous page)

```
    dict(type='LoadImageFromFile'),
    # An encapsulation that encapsulates the testing augmentations
    dict(type='MultiScaleFlipAug',
         # Decides the largest scale for testing, used for the Resize pipeline
         img_scale=(1333, 800),
         flip=False,  # Whether to flip images during testing
         transforms=[
             # Use resize augmentation
             dict(type='Resize', keep_ratio=True),
             # Augmentation pipeline that flip the images and their annotations
             dict(type='RandomFlip'),
             # Augmentation pipeline that normalize the input images
             dict(type='Normalize',
                  # Mean values used in pre-trained backbone models
                  mean=[103.53, 116.28, 123.675],
                  # Standard variance used in pre-trained backbone models
                  std=[1.0, 1.0, 1.0],
                  # The channel orders of image used in pre-trained backbone models
                  to_rgb=False),
             # Padding config, size_divisor: the number the padded images should be␣
↪divisible
             dict(type='Pad', size_divisor=32),
             # Default format bundle to gather data in the pipeline
             dict(type='ImageToTensor', keys=['img']),
             # Pipeline that decides which keys in the data should be passed to the␣
↪detector
             dict(type='Collect', keys=['img'])
         ])
]
data = dict(
    # Batch size of a single GPU
    samples_per_gpu=2,
    # Worker to pre-fetch data for each single GPU
    workers_per_gpu=2,
    train=dict(  # Train dataset config
        save_dataset=False,  # whether to save data_information into json file
        # the pre-defined few shot setting are saved in `FewShotCocoDefaultDataset`
        type='FewShotCocoDefaultDataset',
        ann_cfg=[dict(method='TFA', setting='10SHOT')],  # pre-defined few shot setting
        img_prefix='data/coco/',  # prefix of images
        num_novel_shots=10,  # the max number of instances for novel classes
        num_base_shots=10,  # the max number of instances for base classes
        pipeline=train_pipeline,  # training pipeline
        classes='ALL_CLASSES',  # pre-defined classes split saved in dataset
        # whether to split the annotation (each image only contains one instance)
        instance_wise=False),
    val=dict(  # Validation dataset config
        type='FewShotCocoDataset',  # type of dataset
        ann_cfg=[dict(type='ann_file',  # type of ann_file
                      # path to ann_file
                      ann_file='data/few_shot_ann/coco/annotations/val.json')],
        # prefix of image
```

(continues on next page)

(continued from previous page)

```
        img_prefix='data/coco/',
        pipeline=test_pipeline,  # testing pipeline
        classes='ALL_CLASSES'),
    test=dict(  # Testing dataset config
        type='FewShotCocoDataset',  # type of dataset
        ann_cfg=[dict(type='ann_file',  # type of ann_file
                      # path to ann_file
                      ann_file='data/few_shot_ann/coco/annotations/val.json')],
        # prefix of image
        img_prefix='data/coco/',
        pipeline=test_pipeline,  # testing pipeline
        test_mode=True,  # indicate in test mode
        classes='ALL_CLASSES'))  # pre-defined classes split saved in dataset
# The config to build the evaluation hook, refer to
# https://github.com/open-mmlab/mmdetection/blob/master/mmdet/core/evaluation/eval_hooks.
↪py#L7
# for more details.
evaluation = dict(
    interval=80000,  # Evaluation interval
    metric='bbox',  # Metrics used during evaluation
    classwise=True,  # whether to show result of each class
    # eval results in pre-defined split of classes
    class_splits=['BASE_CLASSES', 'NOVEL_CLASSES'])
# Config used to build optimizer, support all the optimizers
# in PyTorch whose arguments are also the same as those in PyTorch
optimizer = dict(type='SGD', lr=0.001, momentum=0.9, weight_decay=0.0001)
# Config used to build the optimizer hook, refer to
# https://github.com/open-mmlab/mmcv/blob/master/mmcv/runner/hooks/optimizer.py#L8
# for implementation details. Most of the methods do not use gradient clip.
optimizer_config = dict(grad_clip=None)
# Learning rate scheduler config used to register LrUpdater hook
lr_config = dict(
    # The policy of scheduler, also support CosineAnnealing, Cyclic, etc.
    # Refer to details of supported LrUpdater from
    # https://github.com/open-mmlab/mmcv/blob/master/mmcv/runner/hooks/lr_updater.py#L9.
    policy='step',
    # The warmup policy, also support `exp` and `constant`.
    warmup='linear',
    # The number of iterations for warmup
    warmup_iters=10,
    # The ratio of the starting learning rate used for warmup
    warmup_ratio=0.001,
    # Steps to decay the learning rate
    step=[144000])
# Type of runner to use (i.e. IterBasedRunner or EpochBasedRunner)
runner = dict(type='IterBasedRunner', max_iters=160000)
model = dict(  # The config of backbone
    type='TFA',  # The name of detector
    backbone=dict(
        type='ResNet',  # The name of detector
        # The depth of backbone, usually it is 50 or 101 for ResNet and ResNext␣
↪backbones.
```

(continues on next page)

(continued from previous page)

```
        depth=101,
        num_stages=4,  # Number of stages of the backbone.
        # The index of output feature maps produced in each stages
        out_indices=(0, 1, 2, 3),
        # The weights from stages 1 to 4 are frozen
        frozen_stages=4,
        # The config of normalization layers.
        norm_cfg=dict(type='BN', requires_grad=False),
        # Whether to freeze the statistics in BN
        norm_eval=True,
        # The style of backbone, 'pytorch' means that stride 2 layers are in 3x3 conv,
        # 'caffe' means stride 2 layers are in 1x1 convs.
        style='caffe'),
    neck=dict(
        # The neck of detector is FPN. For more details, please refer to
        # https://github.com/open-mmlab/mmdetection/blob/master/mmdet/models/necks/fpn.py
↪#L10
        type='FPN',
        # The input channels, this is consistent with the output channels of backbone
        in_channels=[256, 512, 1024, 2048],
        # The output channels of each level of the pyramid feature map
        out_channels=256,
        # The number of output scales
        num_outs=5,
        # the initialization of specific layer. For more details, please refer to
        # https://mmdetection.readthedocs.io/en/latest/tutorials/init_cfg.html
        init_cfg=[
            # initialize lateral_convs layer with Caffe2Xavier
            dict(type='Caffe2Xavier',
                 override=dict(type='Caffe2Xavier', name='lateral_convs')),
            # initialize fpn_convs layer with Caffe2Xavier
            dict(type='Caffe2Xavier',
                 override=dict(type='Caffe2Xavier', name='fpn_convs'))
        ]),
    rpn_head=dict(
        # The type of RPN head is 'RPNHead'. For more details, please refer to
        # https://github.com/open-mmlab/mmdetection/blob/master/mmdet/models/dense_heads/
↪rpn_head.py#L12
        type='RPNHead',
        # The input channels of each input feature map,
        # this is consistent with the output channels of neck
        in_channels=256,
        # Feature channels of convolutional layers in the head.
        feat_channels=256,
        anchor_generator=dict(  # The config of anchor generator
            # Most of methods use AnchorGenerator, For more details, please refer to
            # https://github.com/open-mmlab/mmdetection/blob/master/mmdet/core/anchor/
↪anchor_generator.py#L10
            type='AnchorGenerator',
            # Basic scale of the anchor, the area of the anchor in one position
            # of a feature map will be scale * base_sizes
            scales=[8],
```

(continues on next page)

(continued from previous page)

```
            # The ratio between height and width.
            ratios=[0.5, 1.0, 2.0],
            # The strides of the anchor generator. This is consistent with the FPN
            # feature strides. The strides will be taken as base_sizes if base_sizes is␣
↪not set.
            strides=[4, 8, 16, 32, 64]),
        bbox_coder=dict(  # Config of box coder to encode and decode the boxes during␣
↪training and testing
            # Type of box coder. 'DeltaXYWHBBoxCoder' is applied for most of methods. For␣
↪more details refer to
            # https://github.com/open-mmlab/mmdetection/blob/master/mmdet/core/bbox/
↪coder/delta_xywh_bbox_coder.py#L9
            type='DeltaXYWHBBoxCoder',
            # The target means used to encode and decode boxes
            target_means=[0.0, 0.0, 0.0, 0.0],
            # The standard variance used to encode and decode boxes
            target_stds=[1.0, 1.0, 1.0, 1.0]),
        # Config of loss function for the classification branch
        loss_cls=dict(
            # Type of loss for classification branch.
            type='CrossEntropyLoss',
            # RPN usually perform two-class classification,
            # so it usually uses sigmoid function.
            use_sigmoid=True,
            # Loss weight of the classification branch.
            loss_weight=1.0),
        # Config of loss function for the regression branch.
        loss_bbox=dict(
            # Type of loss, we also support many IoU Losses and smooth L1-loss. For␣
↪implementation refer to
            # https://github.com/open-mmlab/mmdetection/blob/master/mmdet/models/losses/
↪smooth_l1_loss.py#L56
            type='L1Loss',
            # Loss weight of the regression branch.
            loss_weight=1.0)),
    roi_head=dict(
        # Type of the RoI head, for more details refer to
        # https://github.com/open-mmlab/mmdetection/blob/master/mmdet/models/roi_heads/
↪standard_roi_head.py#L10
        type='StandardRoIHead',
        # RoI feature extractor for bbox regression.
        bbox_roi_extractor=dict(
            # Type of the RoI feature extractor. For more details refer to
            # https://github.com/open-mmlab/mmdetection/blob/master/mmdet/models/roi_
↪heads/roi_extractors/single_level.py#L10
            type='SingleRoIExtractor',
            roi_layer=dict(  # Config of RoI Layer
                # Type of RoI Layer, for more details refer to
                # https://github.com/open-mmlab/mmdetection/blob/master/mmdet/ops/roi_
↪align/roi_align.py#L79
                type='RoIAlign',
                output_size=7,  # The output size of feature maps.
```

(continues on next page)

(continued from previous page)

```
                # Sampling ratio when extracting the RoI features.
                # 0 means adaptive ratio.
                sampling_ratio=0),
            # output channels of the extracted feature.
            out_channels=256,
            # Strides of multi-scale feature maps. It should be consistent to the␣
↪architecture of the backbone.
            featmap_strides=[4, 8, 16, 32]),
        bbox_head=dict(  # Config of box head in the RoIHead.
            # Type of the bbox head, for more details refer to
            # https://github.com/open-mmlab/mmdetection/blob/master/mmdet/models/roi_
↪heads/bbox_heads/convfc_bbox_head.py#L177
            type='CosineSimBBoxHead',
            # Input channels for bbox head. This is consistent with the out_channels in␣
↪roi_extractor
            in_channels=256,
            # Output feature channels of FC layers.
            fc_out_channels=1024,
            roi_feat_size=7,  # Size of RoI features
            num_classes=80,  # Number of classes for classification
            bbox_coder=dict(  # Box coder used in the second stage.
                # Type of box coder. 'DeltaXYWHBBoxCoder' is applied for most of methods.
                type='DeltaXYWHBBoxCoder',
                # Means used to encode and decode box
                target_means=[0.0, 0.0, 0.0, 0.0],
                # Standard variance for encoding and decoding. It is smaller since
                # the boxes are more accurate. [0.1, 0.1, 0.2, 0.2] is a conventional␣
↪setting.
                target_stds=[0.1, 0.1, 0.2, 0.2]),
            reg_class_agnostic=False,  # Whether the regression is class agnostic.
            loss_cls=dict(  # Config of loss function for the classification branch
                # Type of loss for classification branch, we also support FocalLoss etc.
                type='CrossEntropyLoss',
                use_sigmoid=False,  # Whether to use sigmoid.
                loss_weight=1.0),  # Loss weight of the classification branch.
            loss_bbox=dict(  # Config of loss function for the regression branch.
                # Type of loss, we also support many IoU Losses and smooth L1-loss, etc.
                type='L1Loss',
                # Loss weight of the regression branch.
                loss_weight=1.0),
            # the initialization of specific layer. For more details, please refer to
            # https://mmdetection.readthedocs.io/en/latest/tutorials/init_cfg.html
            init_cfg=[
                # initialize shared_fcs layer with Caffe2Xavier
                dict(type='Caffe2Xavier',
                     override=dict(type='Caffe2Xavier', name='shared_fcs')),
                # initialize fc_cls layer with Normal
                dict(type='Normal',
                     override=dict(type='Normal', name='fc_cls', std=0.01)),
                # initialize fc_cls layer with Normal
                dict(type='Normal',
                     override=dict(type='Normal', name='fc_reg', std=0.001))
```

(continues on next page)

(continued from previous page)

```
            ],
            # number of shared fc layers
            num_shared_fcs=2)),
    train_cfg=dict(
        rpn=dict(  # Training config of rpn
            assigner=dict(  # Config of assigner
                # Type of assigner. For more details, please refer to
                # https://github.com/open-mmlab/mmdetection/blob/master/mmdet/core/bbox/
↪assigners/max_iou_assigner.py#L10
                type='MaxIoUAssigner',
                pos_iou_thr=0.7,  # IoU >= threshold 0.7 will be taken as positive␣
↪samples
                neg_iou_thr=0.3,  # IoU < threshold 0.3 will be taken as negative samples
                min_pos_iou=0.3,  # The minimal IoU threshold to take boxes as positive␣
↪samples
                # Whether to match the boxes under low quality (see API doc for more␣
↪details).
                match_low_quality=True,
                ignore_iof_thr=-1),  # IoF threshold for ignoring bboxes
            sampler=dict(  # Config of positive/negative sampler
                # Type of sampler. For more details, please refer to
                # https://github.com/open-mmlab/mmdetection/blob/master/mmdet/core/bbox/
↪samplers/random_sampler.py#L8
                type='RandomSampler',
                num=256,  # Number of samples
                pos_fraction=0.5,  # The ratio of positive samples in the total samples.
                # The upper bound of negative samples based on the number of positive␣
↪samples.
                neg_pos_ub=-1,
                # Whether add GT as proposals after sampling.
                add_gt_as_proposals=False),
            # The border allowed after padding for valid anchors.
            allowed_border=-1,
            # The weight of positive samples during training.
            pos_weight=-1,
            debug=False),  # Whether to set the debug mode
        rpn_proposal=dict(  # The config to generate proposals during training
            nms_pre=2000,  # The number of boxes before NMS
            max_per_img=1000,  # The number of boxes to be kept after NMS.
            nms=dict(  # Config of NMS
                type='nms',  # Type of NMS
                iou_threshold=0.7),  # NMS threshold
            min_bbox_size=0),  # The allowed minimal box size
        rcnn=dict(  # The config for the roi heads.
            assigner=dict(  # Config of assigner for second stage, this is different for␣
↪that in rpn
                # Type of assigner, MaxIoUAssigner is used for all roi_heads for now.␣
↪For more details, please refer to
                # https://github.com/open-mmlab/mmdetection/blob/master/mmdet/core/bbox/
↪assigners/max_iou_assigner.py#L10 for more details.
                type='MaxIoUAssigner',
                pos_iou_thr=0.5,  # IoU >= threshold 0.5 will be taken as positive␣
↪samples
```

(continues on next page)

(continued from previous page)

```
                neg_iou_thr=0.5,  # IoU < threshold 0.5 will be taken as negative samples
                min_pos_iou=0.5,  # The minimal IoU threshold to take boxes as positive␣
↪samples
                # Whether to match the boxes under low quality (see API doc for more␣
↪details).
                match_low_quality=False,
                ignore_iof_thr=-1),  # IoF threshold for ignoring bboxes
            sampler=dict(
                # Type of sampler, PseudoSampler and other samplers are also supported.␣
↪For more details, please refer to
                # https://github.com/open-mmlab/mmdetection/blob/master/mmdet/core/bbox/
↪samplers/random_sampler.py#L8
                type='RandomSampler',
                num=512,  # Number of samples
                pos_fraction=0.25,  # The ratio of positive samples in the total samples.
                # The upper bound of negative samples based on the number of positive␣
↪samples.
                neg_pos_ub=-1,
                # Whether add GT as proposals after sampling.
                add_gt_as_proposals=True),
            # The weight of positive samples during training.
            pos_weight=-1,
            # Whether to set the debug mode
            debug=False)),
    test_cfg=dict(  # Config for testing hyperparameters for rpn and rcnn
        rpn=dict(  # The config to generate proposals during testing
            # The number of boxes before NMS
            nms_pre=1000,
            # The number of boxes to be kept after NMS.
            max_per_img=1000,
            # Config of NMS
            nms=dict(type='nms', iou_threshold=0.7),
            # The allowed minimal box size
            min_bbox_size=0),
        rcnn=dict(  # The config for the roi heads.
            score_thr=0.05,  # Threshold to filter out boxes
            # Config of NMS in the second stage
            nms=dict(type='nms', iou_threshold=0.5),
            # Max number of detections of each image
            max_per_img=100)),
    # parameters with the prefix listed in frozen_parameters will be frozen
    frozen_parameters=[
        'backbone', 'neck', 'rpn_head', 'roi_head.bbox_head.shared_fcs'
    ])
# Config to set the checkpoint hook, Refer to
# https://github.com/open-mmlab/mmcv/blob/master/mmcv/runner/hooks/checkpoint.py for␣
↪implementation.
checkpoint_config = dict(interval=80000)
# The logger used to record the training process.
log_config = dict(interval=50, hooks=[dict(type='TextLoggerHook')])
custom_hooks = [dict(type='NumClassCheckHook')]  # cumstomize hook
dist_params = dict(backend='nccl')  # parameters to setup distributed training, the port␣
↪can also be set.
```

(continues on next page)

(continued from previous page)

```
log_level = 'INFO'  # the output level of the log.
# use base training model as model initialization.
load_from = 'work_dirs/tfa_r101_fpn_coco_base-training/base_model_random_init_bbox_head.
↪pth'
# workflow for runner. [('train', 1)] means there is only one workflow and the workflow␣
↪named 'train' is executed once.
workflow = [('train', 1)]
use_infinite_sampler = True  # whether to use infinite sampler
seed = 0  # random seed
```

13.4 FAQ

13.4.1 Use intermediate variables in configs

Some intermediate variables are used in the configs files, like `train_pipeline/test_pipeline` in datasets. It's worth noting that when modifying intermediate variables in the children configs, user need to pass the intermediate variables into corresponding fields again. For example, we would like to use multi scale strategy to train a Mask R-CNN. `train_pipeline/test_pipeline` are intermediate variable we would like modify.

```
_base_ = './faster_rcnn_r50_caffe_fpn.py'
img_norm_cfg = dict(
    mean=[123.675, 116.28, 103.53], std=[58.395, 57.12, 57.375], to_rgb=True)
train_pipeline = [
    dict(type='LoadImageFromFile'),
    dict(type='LoadAnnotations', with_bbox=True, with_mask=True),
    dict(
        type='Resize',
        img_scale=[(1333, 640), (1333, 672), (1333, 704), (1333, 736),
                   (1333, 768), (1333, 800)],
        multiscale_mode="value",
        keep_ratio=True),
    dict(type='RandomFlip', flip_ratio=0.5),
    dict(type='Normalize', **img_norm_cfg),
    dict(type='Pad', size_divisor=32),
    dict(type='DefaultFormatBundle'),
    dict(type='Collect', keys=['img', 'gt_bboxes', 'gt_labels', 'gt_masks']),
]
test_pipeline = [
    dict(type='LoadImageFromFile'),
    dict(
        type='MultiScaleFlipAug',
        img_scale=(1333, 800),
        flip=False,
        transforms=[
            dict(type='Resize', keep_ratio=True),
            dict(type='RandomFlip'),
            dict(type='Normalize', **img_norm_cfg),
            dict(type='Pad', size_divisor=32),
            dict(type='ImageToTensor', keys=['img']),
            dict(type='Collect', keys=['img']),
```

(continues on next page)

(continued from previous page)

```
        ])
]
data = dict(
    train=dict(pipeline=train_pipeline),
    val=dict(pipeline=test_pipeline),
    test=dict(pipeline=test_pipeline))
```

We first define the new `train_pipeline/test_pipeline` and pass them into `data`.

Similarly, if we would like to switch from `SyncBN` to `BN` or `MMSyncBN`, we need to substitute every `norm_cfg` in the config.

```
_base_ = './faster_rcnn_r50_caffe_fpn.py'
norm_cfg = dict(type='BN', requires_grad=True)
model = dict(
    backbone=dict(norm_cfg=norm_cfg),
    neck=dict(norm_cfg=norm_cfg),
    ...)
```

CHAPTER

FOURTEEN

TUTORIAL 2: ADDING NEW DATASET

14.1 Customize dataset

14.1.1 Load annotations from file

Different from the config in mmdet using `ann_file` to load a single dataset, we use `ann_cfg` to support the complex few shot setting.

The `ann_cfg` is a list of dict and support two type of file:

- loading annotation from the regular `ann_file` of dataset.

```
ann_cfg = [dict(type='ann_file', ann_file='path/to/ann_file'), ...]
```

  For `FewShotVOCDataset`, we also support load specific class from `ann_file` in `ann_classes`.

```
dict(type='ann_file', ann_file='path/to/ann_file', ann_classes=['dog', 'cat'])
```

- loading annotation from a json file saved by a dataset.

```
ann_cfg = [dict(type='saved_dataset', ann_file='path/to/ann_file'), ...]
```

  To save a dataset, we can set the `save_dataset=True` in config file, and the dataset will be saved as `${WORK_DIR}/{TIMESTAMP}_saved_data.json`

```
dataset=dict(type='FewShotVOCDataset', save_dataset=True, ...)
```

14.1.2 Load annotations from predefined benchmark

Unlike few shot classification can test on thousands of tasks in a short time, it is hard to follow the same protocol in few shot detection because of the computation cost. Thus, we provide the predefined data split for reproducibility. These data splits directly use the files released from TFA repo. The details of data preparation can refer to here.

To load these predefined data splits, the type of dataset need to be set to `FewShotVOCDefaultDataset` or `FewShotCocoDefaultDataset`. We provide data splits of each reproduced checkpoint for each method. In config file, we can use `method` and `setting` to determine which data split to load.

Here is an example of config:

```
dataset = dict(
        type='FewShotVOCDefaultDataset',
```

(continues on next page)

(continued from previous page)

```
        ann_cfg=[dict(method='TFA', setting='SPLIT1_1SHOT')]
)
```

14.1.3 Load annotations from another dataset during runtime

In few shot setting, we can use `FewShotVOCCopyDataset` or `FewShotCocoCopyDataset` to copy a dataset from other dataset during runtime for some special cases, such as copying online random sampled support set for model initialization before evaluation. It needs user to modify code in `mmfewshot.detection.apis`. More details can refer to mmfewshot/detection/apis/train.py. Here is an example of config:

```
dataset = dict(
        type='FewShotVOCCopyDataset',
        ann_cfg=[dict(data_infos=FewShotVOCDataset.data_infos)])
```

14.1.4 Use predefined class splits

The predefined class splits are supported in datasets. For VOC, we support [`ALL_CLASSES_SPLIT1`,`ALL_CLASSES_SPLIT2`, `ALL_CLASSES_SPLIT3`, `NOVEL_CLASSES_SPLIT1`, `NOVEL_CLASSES_SPLIT2`, `NOVEL_CLASSES_SPLIT3`, `BASE_CLASSES_SPLIT1`, `BASE_CLASSES_SPLIT2`, `BASE_CLASSES_SPLIT3`]. For COCO, we support [`ALL_CLASSES`, `NOVEL_CLASSES`, `BASE_CLASSES`]

Here is an example of config:

```
data = dict(
    train=dict(type='FewShotVOCDataset', classes='ALL_CLASSES_SPLIT1'),
    val=dict(type='FewShotVOCDataset', classes='ALL_CLASSES_SPLIT1'),
    test=dict(type='FewShotVOCDataset', classes='ALL_CLASSES_SPLIT1'))
```

Also, the class splits can be used to report the evaluation results on different class splits. Here is an example of config:

```
evaluation = dict(class_splits=['BASE_CLASSES_SPLIT1', 'NOVEL_CLASSES_SPLIT1'])
```

14.1.5 Customize the number of annotations

For FewShotDataset, we support two ways to filter extra annotations.

- `ann_shot_filter`: use a dict to specify the class, and the corresponding maximum number of instances when loading the annotation file. For example, we only want 10 instances of dog and 5 instances of person, while other instances from other classes remain unchanged:

  ```
  dataset=dict(type='FewShotVOCDataset',
               ann_shot_filter=dict(dog=10, person=5),
               ...)
  ```

- `num_novel_shots` and `num_base_shots`: use predefined class splits to indicate the corresponding maximum number of instances. For example, we only want 1 instance for each novel class and 3 instances for base class:

  ```
  dataset=dict(
      type='FewShotVOCDataset',
      num_novel_shots=1,
  ```

(continues on next page)

(continued from previous page)

```
    num_base_shots=2,
    ...)
```

14.1.6 Customize the organization of annotations

We also support to split the annotation into instance wise, i.e. each image only have one instance, and the images can be repeated.

```
dataset=dict(
    type='FewShotVOCDataset',
    instance_wise=True,
    ...)
```

14.1.7 Customize pipeline

To support different pipelines in single dataset, we can use `multi_pipelines`. In config file, `multi_pipelines` use the name of keys to indicate specific piplines. Here is an example of config:

```
multi_pipelines = dict(
    query=[
        dict(type='LoadImageFromFile'),
        dict(type='LoadAnnotations', with_bbox=True),
        dict(type='Resize', img_scale=(1000, 600), keep_ratio=True),
        dict(type='RandomFlip', flip_ratio=0.5),
        dict(type='Normalize', **img_norm_cfg),
        dict(type='DefaultFormatBundle'),
        dict(type='Collect', keys=['img', 'gt_bboxes', 'gt_labels'])
    ],
    support=[
        dict(type='LoadImageFromFile'),
        dict(type='LoadAnnotations', with_bbox=True),
        dict(type='Normalize', **img_norm_cfg),
        dict(type='GenerateMask', target_size=(224, 224)),
        dict(type='RandomFlip', flip_ratio=0.0),
        dict(type='DefaultFormatBundle'),
        dict(type='Collect', keys=['img', 'gt_bboxes', 'gt_labels'])
    ])
train=dict(
    type='NWayKShotDataset',
    dataset=dict(
        type='FewShotCocoDataset',
        ...
        multi_pipelines=train_multi_pipelines))
```

When `multi_pipelines` is used, we need to specific the pipeline names in `prepare_train_img` to fetch the image. For example

```
dataset.prepare_train_img(self, idx, 'query')
```

14.2 Customize a new dataset wrapper

In few shot setting, the various sampling logic is implemented by dataset wrapper. An example of customizing query-support data sampling logic for training:

14.2.1 Create a new dataset wrapper

We can create a new dataset wrapper in mmfewshot/detection/datasets/dataset_wrappers.py to customize sampling logic.

```
class MyDatasetWrapper:
    def __init__(self, dataset, support_dataset=None, args_a, args_b, ...):
        # query_dataset and support_dataset can use same dataset
        self.query_dataset = dataset
        self.support_dataset = support_dataset
        if support_dataset is None:
            self.support_dataset = dataset
        ...

    def __getitem__(self, idx):
        ...
        query_data = self.query_dataset.prepare_train_img(idx, 'query')
        # customize sampling logic
        support_idxes = ...
        support_data = [
            self.support_dataset.prepare_train_img(idx, 'support')
            for idx in support_idxes
        ]
        return {'query_data': query_data, 'support_data': support_data}
```

14.2.2 Update dataset builder

We need to add the building code in mmfewshot/detection/datasets/builder.py for our customize dataset wrapper.

```
def build_dataset(cfg, default_args=None):
    if isinstance(cfg, (list, tuple)):
        dataset = ConcatDataset([build_dataset(c, default_args) for c in cfg])
    ...
    elif cfg['type'] == 'MyDatasetWrapper':
        dataset = MyDatasetWrapper(
            build_dataset(cfg['dataset'], default_args),
            build_dataset(cfg['support_dataset'], default_args) if cfg.get('support_
↪dataset', False) else None,
            # pass customize arguments
            args_a=cfg['args_a'],
            args_b=cfg['args_b'],
            ...)
    else:
        dataset = build_from_cfg(cfg, DATASETS, default_args)

    return dataset
```

14.2.3 Update dataloader builder

We need to add the building code of dataloader in mmfewshot/detection/datasets/builder.py, when the customize dataset wrapper will return list of Tensor. We can use `multi_pipeline_collate_fn` to handle this case.

```
def build_dataset(cfg, default_args=None):
    ...
    if isinstance(dataset, MyDatasetWrapper):
      from mmfewshot.utils import multi_pipeline_collate_fn
      # `multi_pipeline_collate_fn` are designed to handle
      # the data with list[list[DataContainer]]
      data_loader = DataLoader(
          dataset,
          batch_size=batch_size,
          sampler=sampler,
          num_workers=num_workers,
          collate_fn=partial(
              multi_pipeline_collate_fn, samples_per_gpu=samples_per_gpu),
          pin_memory=False,
          worker_init_fn=init_fn,
          **kwargs)
    ...
```

14.2.4 Update the arguments in model

The argument names in forward function need to be consistent with the customize dataset wrapper.

```
class MyDetector(BaseDetector):
    ...
    def forward(self, query_data, support_data, ...):
        ...
```

14.2.5 using customize dataset wrapper in config

Then in the config, to use `MyDatasetWrapper` you can modify the config as the following,

```
dataset_A_train = dict(
        type='MyDatasetWrapper',
        args_a=None,
        args_b=None,
        dataset=dict(  # This is the original config of Dataset_A
            type='Dataset_A',
            ...
            multi_pipelines=train_multi_pipelines
        ),
        support_dataset=None
    )
```

14.3 Customize a dataloader wrapper

We also support to iterate two different dataset simultaneously by dataloader wrapper.

An example of customizing dataloader wrapper for query and support dataset:

14.3.1 Create a new dataloader wrapper

We can create a new dataset wrapper in mmfewshot/detection/datasets/dataloader_wrappers.py to customize sampling logic.

```
class MyDataloader:
    def __init__(self, query_data_loader, support_data_loader):
        self.dataset = query_data_loader.dataset
        self.sampler = query_data_loader.sampler
        self.query_data_loader = query_data_loader
        self.support_data_loader = support_data_loader

    def __iter__(self):
        self.query_iter = iter(self.query_data_loader)
        self.support_iter = iter(self.support_data_loader)
        return self

    def __next__(self):
        query_data = self.query_iter.next()
        support_data = self.support_iter.next()
        return {'query_data': query_data, 'support_data': support_data}

    def __len__(self) -> int:
        return len(self.query_data_loader)
```

14.3.2 Update dataloader builder

We need to add the build code in mmfewshot/detection/datasets/builder.py for our customize dataset wrapper.

```
def build_dataloader(dataset, ...):
    if isinstance(dataset, MyDataset):
        ...
        query_data_loader = DataLoader(...)
        support_data_loader = DataLoader(...)
        # wrap two dataloaders with dataloader wrapper
        data_loader = MyDataloader(
            query_data_loader=query_data_loader,
            support_data_loader=support_data_loader)

    return dataset
```

CHAPTER

FIFTEEN

TUTORIAL 3: CUSTOMIZE MODELS

We basically categorize model components into 5 types the same as mmdet.

- backbone: usually an FCN network to extract feature maps, e.g., ResNet, MobileNet.
- neck: the component between backbones and heads, e.g., FPN, PAFPN.
- head: the component for specific tasks, e.g., bbox prediction and mask prediction.
- roi extractor: the part for extracting RoI features from feature maps, e.g., RoI Align.
- loss: the component in head for calculating losses, e.g., FocalLoss, L1Loss, and GHMLoss.

15.1 Develop new components

15.1.1 Add a new detector

Here we show how to develop new components with an example.

15.1.2 Add a new backbone

Here we show how to develop new components with an example of MobileNet.

1. Define a new backbone (e.g. MobileNet)

Create a new file `mmfewshot/detection/models/backbones/mobilenet.py`.

```
import torch.nn as nn

from ..builder import BACKBONES


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

    def __init__(self, arg1, arg2):
        pass

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

2. Import the module

You can either add the following line to `mmfewshot/detection/models/backbones/__init__.py`

```
from .mobilenet import MobileNet
```

or alternatively add

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

to the config file to avoid modifying the original code.

3. Use the backbone in your config file

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

15.1.3 Add new necks

1. Define a neck (e.g. PAFPN)

Create a new file `mmfewshot/detection/models/necks/pafpn.py`.

```
from ..builder import NECKS

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

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

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

2. Import the module

You can either add the following line to `mmfewshot/detection/models/necks/__init__.py`,

```
from .pafpn import PAFPN
```

or alternatively add

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

to the config file and avoid modifying the original code.

3. Modify the config file

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

15.1.4 Add new heads

Here we show how to develop a new head with the example of Double Head R-CNN as the following.

First, add a new bbox head in `mmfewshot/detection/models/roi_heads/bbox_heads/double_bbox_head.py`. Double Head R-CNN implements a new bbox head for object detection. To implement a bbox head, basically we need to implement three functions of the new module as the following.

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

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

                                    /-> cls
                /-> shared convs ->
                                    \-> reg
    roi features
                                    /-> cls
                \-> shared fc    ->
                                    \-> reg
    """  # noqa: W605

    def __init__(self,
                 num_convs=0,
                 num_fcs=0,
                 conv_out_channels=1024,
                 fc_out_channels=1024,
                 conv_cfg=None,
```

(continues on next page)

(continued from previous page)

```
                 norm_cfg=dict(type='BN'),
                 **kwargs):
        kwargs.setdefault('with_avg_pool', True)
        super(DoubleConvFCBBoxHead, self).__init__(**kwargs)


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

Second, implement a new RoI Head if it is necessary. We plan to inherit the new `DoubleHeadRoIHead` from `StandardRoIHead`. We can find that a `StandardRoIHead` already implements the following functions.

```
import torch

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


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

    def init_assigner_sampler(self):

    def init_bbox_head(self, bbox_roi_extractor, bbox_head):

    def init_mask_head(self, mask_roi_extractor, mask_head):


    def forward_dummy(self, x, proposals):


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

    def _bbox_forward(self, x, rois):

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

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

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

(continues on next page)

(continued from previous page)

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

Double Head's modification is mainly in the bbox_forward logic, and it inherits other logics from the StandardRoIHead. In the mmfewshot/detection/models/roi_heads/double_roi_head.py, we implement the new RoI Head as the following:

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


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

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

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

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

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

Last, the users need to add the module in mmfewshot/detection/models/bbox_heads/__init__.py and mmfewshot/detection/models/roi_heads/__init__.py thus the corresponding registry could find and load them.

Alternatively, the users can add

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

to the config file and achieve the same goal.

The config file of Double Head R-CNN is as the following

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

Since MMDetection 2.0, the config system supports to inherit configs such that the users can focus on the modification. The Double Head R-CNN mainly uses a new DoubleHeadRoIHead and a new `DoubleConvFCBBoxHead`, the arguments are set according to the `__init__` function of each module.

15.1.5 Add new loss

Assume you want to add a new loss as `MyLoss`, for bounding box regression. To add a new loss function, the users need implement it in `mmfewshot/detection/models/losses/my_loss.py`. The decorator `weighted_loss` enable the loss to be weighted for each element.

```
import torch
import torch.nn as nn

from ..builder import LOSSES
from .utils import weighted_loss

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

(continues on next page)

(continued from previous page)

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

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

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

Then the users need to add it in the `mmfewshot/detection/models/losses/__init__.py`.

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

Alternatively, you can add

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

to the config file and achieve the same goal.

To use it, modify the `loss_xxx` field. Since MyLoss is for regression, you need to modify the `loss_bbox` field in the head.

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

15.2 Customize frozen parameters

We support `frozen_parameters` to freeze the parameters during training by parameters' prefix. For example, in `roi_head` if we only want to freeze the `shared_fcs` in `bbox_head`, we can add `roi_head.bbox_head.shared_fcs` into `frozen_parameters` list.

```
model = dict(
    frozen_parameters=[
        'backbone', 'neck', 'rpn_head', 'roi_head.bbox_head.shared_fcs'
    ])
```

15.3 Customize a query-support based detector

Here we show how to develop a new query-support based detector with the example.

15.3.1 1. Define a new detector

Create a new file `mmfewshot/detection/models/detector/my_detector.py`.

```
@DETECTORS.register_module()
class MyDetector(QuerySupportDetector):
    # customize the input data
    def forward(self, query_data, support_data, img, img_metas, mode, **kwargs):
        if mode == 'train':
            return self.forward_train(query_data, support_data, **kwargs)
        elif mode == 'model_init':
            return self.forward_model_init(img, img_metas, **kwargs)
        elif mode == 'test':
            return self.forward_test(img, img_metas, **kwargs)
        ...

    def forward_train(self, query_data, support_data, proposals, **kwargs):
        ...

    # before testing the model will forward the whole support set
    # customize the forward logic and save all the needed information
    def forward_model_init(self, img, img_metas, gt_bboxes, gt_labels):
        ...
    # customize the process logic for the saved information from images
    def model_init(self, **kwargs):
        ...
```

15.3.2 2. Import the module

You can either add the following line to `mmfewshot/detection/models/detectors/__init__.py`

```
from .my_detector import MyDetector
```

or alternatively add

```
custom_imports = dict(
    imports=['mmfewshot.detection.models.detectors.my_detector'],
    allow_failed_imports=False)
```

to the config file to avoid modifying the original code.

15.3.3 3. Use the detector in your config file

```
model = dict(
    type='MyDetector',
    ...
```

15.3.4 Customize an aggregation layer

we also support to reuse the code of feature fusion from different data usually used in query support based methods. Here we show how to develop a new aggregator with the example.

1. Define a new aggregator

Add customize code in `mmfewshot/detection/models/utils/aggregation_layer.py`.

```
@AGGREGATORS.register_module()
class MyAggregator(BaseModule):

    def __init__(self,...):

    def forward(self, query_feat, support_feat):
        ...
        return feat
```

2. Use the aggregator in your config file

The `aggregation_layer` can build from single aggregator:

```
aggregation_layer = dict(type='MyAggregator',...)
```

or build with multiple aggregators and wrap by a `AggregationLayer`.

```
aggregation_layer = dict(
    type = 'AggregationLayer',
    aggregator_cfgs = [
        dict(type = 'MyAggregator',...),
        ...]
)
```

3. Use the aggregator in your model

```
from mmfewshot.detection.models.utils import build_aggregator
@HEADS.register_module()
class MyHead(...):
    def __init__(self, ..., aggregation_layer):
        self.aggregation_layer = build_aggregator(copy.deepcopy(aggregation_layer))

    def forward_train(self, ...):
        ...
```

(continues on next page)

(continued from previous page)

```
        self.aggregation_layer(query_feat=..., support_feat=...)
        ...
```

CHAPTER

SIXTEEN

TUTORIAL 4: CUSTOMIZE RUNTIME SETTINGS

16.1 Customize optimization settings

16.1.1 Customize optimizer supported by Pytorch

We already support to use all the optimizers implemented by PyTorch, and the only modification is to change the `optimizer` field of config files. For example, if you want to use `ADAM` (note that the performance could drop a lot), the modification could be as the following.

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

To modify the learning rate of the model, the users only need to modify the `lr` in the config of optimizer. The users can directly set arguments following the API doc of PyTorch.

16.1.2 Customize self-implemented optimizer

1. Define a new optimizer

A customized optimizer could be defined as following.

Assume you want to add a optimizer named `MyOptimizer`, which has arguments `a`, `b`, and `c`. You need to create a new directory named `mmfewshot/detection/core/optimizer`. And then implement the new optimizer in a file, e.g., in `mmfewshot/detection/core/optimizer/my_optimizer.py`:

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


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

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

2. Add the optimizer to registry

To find the above module defined above, this module should be imported into the main namespace at first. There are two options to achieve it.

- Modify `mmfewshot/detection/core/optimizer/__init__.py` to import it.

  The newly defined module should be imported in `mmfewshot/detection/core/optimizer/__init__.py` so that the registry will find the new module and add it:

```
from .my_optimizer import MyOptimizer
```

- Use `custom_imports` in the config to manually import it

```
custom_imports = dict(imports=['mmfewshot.detection.core.optimizer.my_optimizer'], allow_
↪failed_imports=False)
```

The module `mmfewshot.detection.core.optimizer.my_optimizer` will be imported at the beginning of the program and the class `MyOptimizer` is then automatically registered. Note that only the package containing the class `MyOptimizer` should be imported. `mmfewshot.detection.core.optimizer.my_optimizer.MyOptimizer` **cannot** be imported directly.

Actually users can use a totally different file directory structure using this importing method, as long as the module root can be located in `PYTHONPATH`.

3. Specify the optimizer in the config file

Then you can use `MyOptimizer` in `optimizer` field of config files. In the configs, the optimizers are defined by the field `optimizer` like the following:

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

To use your own optimizer, the field can be changed to

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

16.1.3 Customize optimizer constructor

Some models may have some parameter-specific settings for optimization, e.g. weight decay for BatchNorm layers. The users can do those fine-grained parameter tuning through customizing optimizer constructor.

```
from mmcv.utils import build_from_cfg

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


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

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

    def __call__(self, model):
```

(continues on next page)

(continued from previous page)

```
        return my_optimizer
```

The default optimizer constructor is implemented here, which could also serve as a template for new optimizer constructor.

16.1.4 Additional settings

Tricks not implemented by the optimizer should be implemented through optimizer constructor (e.g., set parameter-wise learning rates) or hooks. We list some common settings that could stabilize the training or accelerate the training. Feel free to create PR, issue for more settings.

- **Use gradient clip to stabilize training**: Some models need gradient clip to clip the gradients to stabilize the training process. An example is as below:

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

- **Use momentum schedule to accelerate model convergence**: We support momentum scheduler to modify model's momentum according to learning rate, which could make the model converge in a faster way. Momentum scheduler is usually used with LR scheduler, for example, the following config is used in 3D detection to accelerate convergence. For more details, please refer to the implementation of CyclicLrUpdater and CyclicMomentumUpdater.

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

16.2 Customize training schedules

By default we use step learning rate with 1x schedule, this calls StepLRHook in MMCV. We support many other learning rate schedule here, such as `CosineAnnealing` and `Poly` schedule. Here are some examples

- Poly schedule:

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

- ConsineAnnealing schedule:

```
lr_config = dict(
    policy='CosineAnnealing',
    warmup='linear',
    warmup_iters=1000,
```

(continues on next page)

(continued from previous page)

```
    warmup_ratio=1.0 / 10,
    min_lr_ratio=1e-5)
```

16.3 Customize workflow

Workflow is a list of (phase, epochs) to specify the running order and epochs. By default it is set to be

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

which means running 1 epoch for training. Sometimes user may want to check some metrics (e.g. loss, accuracy) about the model on the validate set. In such case, we can set the workflow as

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

so that 1 epoch for training and 1 epoch for validation will be run iteratively.

Note:

1. The parameters of model will not be updated during val epoch.
2. Keyword `total_epochs` in the config only controls the number of training epochs and will not affect the validation workflow.
3. Workflows `[('train', 1), ('val', 1)]` and `[('train', 1)]` will not change the behavior of `EvalHook` because `EvalHook` is called by `after_train_epoch` and validation workflow only affect hooks that are called through `after_val_epoch`. Therefore, the only difference between `[('train', 1), ('val', 1)]` and `[('train', 1)]` is that the runner will calculate losses on validation set after each training epoch.

16.4 Customize hooks

16.4.1 Customize self-implemented hooks

1. Implement a new hook

Here we give an example of creating a new hook in MMPose and using it in training.

```
from mmcv.runner import HOOKS, Hook


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

    def __init__(self, a, b):
        pass

    def before_run(self, runner):
        pass

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

(continues on next page)

(continued from previous page)

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

    def after_epoch(self, runner):
        pass

    def before_iter(self, runner):
        pass

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

Depending on the functionality of the hook, the users need to specify what the hook will do at each stage of the training in `before_run`, `after_run`, `before_epoch`, `after_epoch`, `before_iter`, and `after_iter`.

2. Register the new hook

Then we need to make `MyHook` imported. Assuming the file is in `mmfewshot/detection/core/utils/my_hook.py` there are two ways to do that:

- Modify `mmfewshot/core/utils/__init__.py` to import it.

  The newly defined module should be imported in `mmfewshot/detection/core/utils/__init__.py` so that the registry will find the new module and add it:

```
from .my_hook import MyHook
```

- Use `custom_imports` in the config to manually import it

```
custom_imports = dict(imports=['mmfewshot.detection.core.utils.my_hook'], allow_failed_
↪imports=False)
```

3. Modify the config

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

You can also set the priority of the hook by adding key `priority` to `'NORMAL'` or `'HIGHEST'` as below

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

By default the hook's priority is set as `NORMAL` during registration.

16.4.2 Use hooks implemented in MMCV

If the hook is already implemented in MMCV, you can directly modify the config to use the hook as below

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

16.4.3 Modify default runtime hooks

There are some common hooks that are not registered through `custom_hooks`, they are

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

In those hooks, only the logger hook has the `VERY_LOW` priority, others' priority are `NORMAL`. The above-mentioned tutorials already covers how to modify `optimizer_config`, `momentum_config`, and `lr_config`. Here we reveals how what we can do with `log_config`, `checkpoint_config`, and `evaluation`.

Checkpoint config

The MMCV runner will use `checkpoint_config` to initialize CheckpointHook.

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

The users could set `max_keep_ckpts` to only save only small number of checkpoints or decide whether to store state dict of optimizer by `save_optimizer`. More details of the arguments are here

Log config

The `log_config` wraps multiple logger hooks and enables to set intervals. Now MMCV supports `WandbLoggerHook`, `MlflowLoggerHook`, and `TensorboardLoggerHook`. The detail usages can be found in the doc.

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

Evaluation config

The config of `evaluation` will be used to initialize the `EvalHook`. Except the key `interval`, other arguments such as `metric` will be passed to the `dataset.evaluate()`

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

CHAPTER

SEVENTEEN

CHANGELOG

CHAPTER

EIGHTEEN

FREQUENTLY ASKED QUESTIONS

We list some common troubles faced by many users and their corresponding solutions here. Feel free to enrich the list if you find any frequent issues and have ways to help others to solve them. If the contents here do not cover your issue, please create an issue using the provided templates and make sure you fill in all required information in the template.

18.1 MMCV Installation

- Compatibility issue between MMCV and MMDetection; "ConvWS is already registered in conv layer"; "AssertionError: MMCV==xxx is used but incompatible. Please install mmcv>=xxx, <=xxx."

  Please install the correct version of MMCV for the version of your MMDetection following the installation instruction.

- "No module named 'mmcv.ops'"; "No module named 'mmcv._ext'".

  1. Uninstall existing mmcv in the environment using `pip uninstall mmcv`.
  2. Install mmcv-full following the installation instruction.

18.2 PyTorch/CUDA Environment

- "invalid device function" or "no kernel image is available for execution".

  1. Check if your cuda runtime version (under `/usr/local/`), `nvcc --version` and `conda list cudatoolkit` version match.
  2. Run `python mmdet/utils/collect_env.py` to check whether PyTorch, torchvision, and MMCV are built for the correct GPU architecture. You may need to set `TORCH_CUDA_ARCH_LIST` to reinstall MMCV. The GPU arch table could be found here, i.e. run `TORCH_CUDA_ARCH_LIST=7.0 pip install mmcv-full` to build MMCV for Volta GPUs. The compatibility issue could happen when using old GPUS, e.g., Tesla K80 (3.7) on colab.
  3. Check whether the running environment is the same as that when mmcv/mmdet has compiled. For example, you may compile mmcv using CUDA 10.0 but run it on CUDA 9.0 environments.

- "undefined symbol" or "cannot open xxx.so".

  1. If those symbols are CUDA/C++ symbols (e.g., libcudart.so or GLIBCXX), check whether the CUDA/GCC runtimes are the same as those used for compiling mmcv, i.e. run `python mmdet/utils/collect_env.py` to see if `"MMCV Compiler"`/`"MMCV CUDA Compiler"` is the same as `"GCC"`/`"CUDA_HOME"`.
  2. If those symbols are PyTorch symbols (e.g., symbols containing caffe, aten, and TH), check whether the PyTorch version is the same as that used for compiling mmcv.

3. Run `python mmdet/utils/collect_env.py` to check whether PyTorch, torchvision, and MMCV are built by and running on the same environment.

- setuptools.sandbox.UnpickleableException: DistutilsSetupError("each element of 'ext_modules' option must be an Extension instance or 2-tuple")

1. If you are using miniconda rather than anaconda, check whether Cython is installed as indicated in #3379. You need to manually install Cython first and then run command `pip install -r requirements.txt`.
2. You may also need to check the compatibility between the `setuptools`, `Cython`, and `PyTorch` in your environment.

- "Segmentation fault".

1. Check you GCC version and use GCC 5.4. This usually caused by the incompatibility between PyTorch and the environment (e.g., GCC < 4.9 for PyTorch). We also recommend the users to avoid using GCC 5.5 because many feedbacks report that GCC 5.5 will cause "segmentation fault" and simply changing it to GCC 5.4 could solve the problem.
2. Check whether PyTorch is correctly installed and could use CUDA op, e.g. type the following command in your terminal.

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

   And see whether they could correctly output results.

3. If Pytorch is correctly installed, check whether MMCV is correctly installed.

```
python -c 'import mmcv; import mmcv.ops'
```

   If MMCV is correctly installed, then there will be no issue of the above two commands.

4. If MMCV and Pytorch is correctly installed, you man use `ipdb`, `pdb` to set breakpoints or directly add 'print' in mmdetection code and see which part leads the segmentation fault.

18.3 Training

- "Loss goes Nan"

1. Check if the dataset annotations are valid: zero-size bounding boxes will cause the regression loss to be Nan due to the commonly used transformation for box regression. Some small size (width or height are smaller than 1) boxes will also cause this problem after data augmentation (e.g., instaboost). So check the data and try to filter out those zero-size boxes and skip some risky augmentations on the small-size boxes when you face the problem.
2. Reduce the learning rate: the learning rate might be too large due to some reasons, e.g., change of batch size. You can rescale them to the value that could stably train the model.
3. Extend the warmup iterations: some models are sensitive to the learning rate at the start of the training. You can extend the warmup iterations, e.g., change the `warmup_iters` from 500 to 1000 or 2000.
4. Add gradient clipping: some models requires gradient clipping to stabilize the training process. The default of `grad_clip` is `None`, you can add gradient clippint to avoid gradients that are too large, i.e., set `optimizer_config=dict(_delete_=True, grad_clip=dict(max_norm=35, norm_type=2))` in your config file. If your config does not inherits from any basic config that contains `optimizer_config=dict(grad_clip=None)`, you can simply add `optimizer_config=dict(grad_clip=dict(max_norm=35, norm_type=2))`.

- 'GPU out of memory"

1. There are some scenarios when there are large amounts of ground truth boxes, which may cause OOM during target assignment. You can set `gpu_assign_thr=N` in the config of assigner thus the assigner will calculate box overlaps through CPU when there are more than N GT boxes.
2. Set `with_cp=True` in the backbone. This uses the sublinear strategy in PyTorch to reduce GPU memory cost in the backbone.
3. Try mixed precision training using following the examples in `config/fp16`. The `loss_scale` might need further tuning for different models.

- "RuntimeError: Expected to have finished reduction in the prior iteration before starting a new one"
  1. This error indicates that your module has parameters that were not used in producing loss. This phenomenon may be caused by running different branches in your code in DDP mode.
  2. You can set `find_unused_parameters = True` in the config to solve the above problems or find those unused parameters manually.

18.4 Evaluation

- COCO Dataset, AP or AR = -1
  1. According to the definition of COCO dataset, the small and medium areas in an image are less than 1024 (32*32), 9216 (96*96), respectively.
  2. If the corresponding area has no object, the result of AP and AR will set to -1.

CHAPTER

NINETEEN

ENGLISH

CHAPTER

TWENTY

CHAPTER

TWENTYONE

MMFEWSHOT.CLASSIFICATION

21.1 classification.apis

mmfewshot.classification.apis.**inference_classifier**(*model: torch.nn.modules.module.Module*, *query_img: str*) → Dict

Inference single image with the classifier.

Parameters

- **model** (*nn.Module*) – The loaded classifier.
- **query_img** (*str*) – The image filename.

Returns

The classification results that contains *pred_score* of each class.

Return type dict

mmfewshot.classification.apis.**init_classifier**(*config: Union[str, mmcv.utils.config.Config]*, *checkpoint: Optional[str] = None*, *device: str = 'cuda:0'*, *options: Optional[Dict] = None*) → torch.nn.modules.module.Module

Prepare a few shot classifier from config file.

Parameters

- **config** (str or mmcv.Config) – Config file path or the config object.
- **checkpoint** (*str | None*) – Checkpoint path. If left as None, the model will not load any weights. Default: None.
- **device** (*str*) – Runtime device. Default: 'cuda:0'.
- **options** (*dict | None*) – Options to override some settings in the used config. Default: None.

Returns The constructed classifier.

Return type nn.Module

mmfewshot.classification.apis.**multi_gpu_meta_test**(*model: mmcv.parallel.distributed.MMDistributedDataParallel*, *num_test_tasks: int*, *support_dataloader: torch.utils.data.dataloader.DataLoader*, *query_dataloader: torch.utils.data.dataloader.DataLoader*, *test_set_dataloader: Optional[torch.utils.data.dataloader.DataLoader] = None*, *meta_test_cfg: Optional[Dict] = None*, *eval_kwargs: Optional[Dict] = None*, *logger: Optional[object] = None*, *confidence_interval: float = 0.95*, *show_task_results: bool = False*) → Dict

Distributed meta testing on multiple gpus.

During meta testing, model might be further fine-tuned or added extra parameters. While the tested model need to be restored after meta testing since meta testing can be used as the validation in the middle of training. To detach model from previous phase, the model will be copied and wrapped with `MetaTestParallel`. And it has full independence from the training model and will be discarded after the meta testing.

In the distributed situation, the `MetaTestParallel` on each GPU is also independent. The test tasks in few shot leaning usually are very small and hardly benefit from distributed acceleration. Thus, in distributed meta testing, each task is done in single GPU and each GPU is assigned a certain number of tasks. The number of test tasks for each GPU is ceil(num_test_tasks / world_size). After all GPUs finish their tasks, the results will be aggregated to get the final result.

Parameters

- **model** (`MMDistributedDataParallel`) – Model to be meta tested.
- **num_test_tasks** (*int*) – Number of meta testing tasks.
- **support_dataloader** (`DataLoader`) – A PyTorch dataloader of support data.
- **query_dataloader** (`DataLoader`) – A PyTorch dataloader of query data.
- **test_set_dataloader** (`DataLoader`) – A PyTorch dataloader of all test data. Default: None.
- **meta_test_cfg** (*dict*) – Config for meta testing. Default: None.
- **eval_kwargs** (*dict*) – Any keyword argument to be used for evaluation. Default: None.
- **logger** (*logging.Logger | None*) – Logger used for printing related information during evaluation. Default: None.
- **confidence_interval** (*float*) – Confidence interval. Default: 0.95.
- **show_task_results** (*bool*) – Whether to record the eval result of each task. Default: False.

Returns

Dict of meta evaluate results, containing *accuracy_mean* and *accuracy_std* of all test tasks.

Return type dict | None

mmfewshot.classification.apis.**process_support_images**(*model: torch.nn.modules.module.Module*, *support_imgs: List[str]*, *support_labels: List[str]*) → None

Process support images.

Parameters

- **model** (*nn.Module*) – Classifier model.
- **support_imgs** (*list[str]*) – The image filenames.
- **support_labels** (*list[str]*) – The class names of support images.

mmfewshot.classification.apis.**show_result_pyplot**(*img: str*, *result: Dict*, *fig_size: Tuple[int] = (15, 10)*, *wait_time: int = 0*, *out_file: Optional[str] = None*) → numpy.ndarray

Visualize the classification results on the image.

Parameters

- **img** (*str*) – Image filename.
- **result** (*dict*) – The classification result.
- **fig_size** (*tuple*) – Figure size of the pyplot figure. Default: (15, 10).
- **wait_time** (*int*) – How many seconds to display the image. Default: 0.
- **out_file** (*str | None*) – Default: None

Returns pyplot figure.

Return type np.ndarray

mmfewshot.classification.apis.**single_gpu_meta_test**(*model: Union[mmcv.parallel.data_parallel.MMDataParallel, torch.nn.modules.module.Module]*, *num_test_tasks: int*, *support_dataloader: torch.utils.data.dataloader.DataLoader*, *query_dataloader: torch.utils.data.dataloader.DataLoader*, *test_set_dataloader: Optional[torch.utils.data.dataloader.DataLoader] = None*, *meta_test_cfg: Optional[Dict] = None*, *eval_kwargs: Optional[Dict] = None*, *logger: Optional[object] = None*, *confidence_interval: float = 0.95*, *show_task_results: bool = False*) → Dict

Meta testing on single gpu.

During meta testing, model might be further fine-tuned or added extra parameters. While the tested model need to be restored after meta testing since meta testing can be used as the validation in the middle of training. To detach model from previous phase, the model will be copied and wrapped with `MetaTestParallel`. And it has full independence from the training model and will be discarded after the meta testing.

Parameters

- **model** (MMDataParallel | nn.Module) – Model to be meta tested.
- **num_test_tasks** (*int*) – Number of meta testing tasks.
- **support_dataloader** (DataLoader) – A PyTorch dataloader of support data and it is used to fetch support data for each task.
- **query_dataloader** (DataLoader) – A PyTorch dataloader of query data and it is used to fetch query data for each task.
- **test_set_dataloader** (DataLoader) – A PyTorch dataloader of all test data and it is used for feature extraction from whole dataset to accelerate the testing. Default: None.
- **meta_test_cfg** (*dict*) – Config for meta testing. Default: None.

- **eval_kwargs** (*dict*) – Any keyword argument to be used for evaluation. Default: None.
- **logger** (*logging.Logger | None*) – Logger used for printing related information during evaluation. Default: None.
- **confidence_interval** (*float*) – Confidence interval. Default: 0.95.
- **show_task_results** (*bool*) – Whether to record the eval result of each task. Default: False.

Returns

Dict of meta evaluate results, containing *accuracy_mean* and *accuracy_std* of all test tasks.

Return type dict

mmfewshot.classification.apis.**test_single_task**(*model:* mmfewshot.classification.utils.meta_test_parallel.MetaTestParallel, *support_dataloader: torch.utils.data.dataloader.DataLoader*, *query_dataloader: torch.utils.data.dataloader.DataLoader*, *meta_test_cfg: Dict*)

Test a single task.

A task has two stages: handling the support set and predicting the query set. In stage one, it currently supports fine-tune based and metric based methods. In stage two, it simply forward the query set and gather all the results.

Parameters

- **model** (MetaTestParallel) – Model to be meta tested.
- **support_dataloader** (DataLoader) – A PyTorch dataloader of support data.
- **query_dataloader** (DataLoader) – A PyTorch dataloader of query data.
- **meta_test_cfg** (*dict*) – Config for meta testing.

Returns

- results_list (list[np.ndarray]): Predict results.
- gt_labels (np.ndarray): Ground truth labels.

Return type tuple

21.2 classification.core

21.3 classification.datasets

class mmfewshot.classification.datasets.**BaseFewShotDataset**(*data_prefix: str*, *pipeline: List[Dict]*, *classes: Optional[Union[str, List[str]]] = None*, *ann_file: Optional[str] = None*)

Base few shot dataset.

Parameters

- **data_prefix** (*str*) – The prefix of data path.

- **pipeline** (*list*) – A list of dict, where each element represents a operation defined in *mmcls.datasets.pipelines*.
- **classes** (*str | Sequence[str] | None*) – Classes for model training and provide fixed label for each class. Default: None.
- **ann_file** (*str | None*) – The annotation file. When *ann_file* is str, the subclass is expected to read from the *ann_file*. When *ann_file* is None, the subclass is expected to read according to data_prefix. Default: None.

property class_to_idx: Mapping

Map mapping class name to class index.

Returns mapping from class name to class index.

Return type dict

static evaluate(*results: List*, *gt_labels: numpy.array*, *metric: Union[str, List[str]] = 'accuracy'*, *metric_options: Optional[dict] = None*, *logger: Optional[object] = None*) → Dict

Evaluate the dataset.

Parameters

- **results** (*list*) – Testing results of the dataset.
- **gt_labels** (*np.ndarray*) – Ground truth labels.
- **metric** (*str | list[str]*) – Metrics to be evaluated. Default value is *accuracy*.
- **metric_options** (*dict | None*) – Options for calculating metrics. Allowed keys are 'topk', 'thrs' and 'average_mode'. Default: None.
- **logger** (*logging.Logger | None*) – Logger used for printing related information during evaluation. Default: None.

Returns evaluation results

Return type dict

classmethod get_classes(*classes: Optional[Union[Sequence[str], str]] = None*) → Sequence[str]

Get class names of current dataset.

Parameters **classes** (*Sequence[str] | str | None*) – Three types of input will correspond to different processing logics:

- If *classes* is a tuple or list, it will override the CLASSES predefined in the dataset.
- If *classes* is None, we directly use pre-defined CLASSES will be used by the dataset.
- If *classes* is a string, it is the path of a classes file that contains the name of all classes. Each line of the file contains a single class name.

Returns Names of categories of the dataset.

Return type tuple[str] or list[str]

sample_shots_by_class_id(*class_id: int*, *num_shots: int*) → List[int]

Random sample shots of given class id.

class mmfewshot.classification.datasets.**CUBDataset**(*classes_id_seed: Optional[int] = None*, *subset: typing_extensions.Literal[train, test, val] = 'train'*, **args*, ***kwargs*)

CUB dataset for few shot classification.

Parameters

- **classes_id_seed** (*int | None*) – A random seed to shuffle order of classes. If seed is None, the classes will be arranged in alphabetical order. Default: None.
- **subset** (*str| list[str]*) – The classes of whole dataset are split into three disjoint subset: train, val and test. If subset is a string, only one subset data will be loaded. If subset is a list of string, then all data of subset in list will be loaded. Options: ['train', 'val', 'test']. Default: 'train'.

get_classes(*classes: Optional[Union[Sequence[str], str]] = None*) → Sequence[str]
Get class names of current dataset.

Parameters **classes** (*Sequence[str] | str | None*) – Three types of input will correspond to different processing logics:

- If *classes* is a tuple or list, it will override the CLASSES predefined in the dataset.
- If *classes* is None, we directly use pre-defined CLASSES will be used by the dataset.
- If *classes* is a string, it is the path of a classes file that contains the name of all classes. Each line of the file contains a single class name.

Returns Names of categories of the dataset.

Return type tuple[str] or list[str]

load_annotations() → List[Dict]
Load annotation according to the classes subset.

class mmfewshot.classification.datasets.**EpisodicDataset**(*dataset: torch.utils.data.dataset.Dataset*, *num_episodes: int*, *num_ways: int*, *num_shots: int*, *num_queries: int*, *episodes_seed: Optional[int] = None*)
A wrapper of episodic dataset.

It will generate a list of support and query images indices for each episode (support + query images). Every call of *__getitem__* will fetch and return (*num_ways * num_shots*) support images and (*num_ways * num_queries*) query images according to the generated images indices. Note that all the episode indices are generated at once using a specific random seed to ensure the reproducibility for same dataset.

Parameters

- **dataset** (Dataset) – The dataset to be wrapped.
- **num_episodes** (*int*) – Number of episodes. Noted that all episodes are generated at once and will not be changed afterwards. Make sure setting the *num_episodes* larger than your needs.
- **num_ways** (*int*) – Number of ways for each episode.
- **num_shots** (*int*) – Number of support data of each way for each episode.
- **num_queries** (*int*) – Number of query data of each way for each episode.
- **episodes_seed** (*int | None*) – A random seed to reproduce episodic indices. If seed is None, it will use runtime random seed. Default: None.

class mmfewshot.classification.datasets.**LoadImageFromBytes**(*to_float32=False*, *color_type='color'*, *file_client_args={'backend': 'disk'}*)
Load an image from bytes.

class mmfewshot.classification.datasets.**MetaTestDataset**(**args*, ***kwargs*)
A wrapper of the episodic dataset for meta testing.

During meta test, the *MetaTestDataset* will be copied and converted into three mode: *test_set*, *support*, and *test*. Each mode of dataset will be used in different dataloader, but they share the same episode and image information.

- In *test_set* mode, the dataset will fetch all images from the whole test set to extract features from the fixed backbone, which can accelerate meta testing.
- In *support* or *query* mode, the dataset will fetch images according to the *episode_idxes* with the same *task_id*. Therefore, the support and query dataset must be set to the same *task_id* in each test task.

cache_feats(*feats: torch.Tensor*, *img_metas: Dict*) → None
Cache extracted feats into dataset.

set_task_id(*task_id: int*) → None
Query and support dataset use same task id to make sure fetch data from same episode.

class mmfewshot.classification.datasets.**MiniImageNetDataset**(*subset: typing_extensions.Literal[train, test, val] = 'train'*, *file_format: str = 'JPEG'*, **args*, ***kwargs*)

MiniImageNet dataset for few shot classification.

Parameters

- **subset** (*str| list[str]*) – The classes of whole dataset are split into three disjoint subset: train, val and test. If subset is a string, only one subset data will be loaded. If subset is a list of string, then all data of subset in list will be loaded. Options: ['train', 'val', 'test']. Default: 'train'.
- **file_format** (*str*) – The file format of the image. Default: 'JPEG'

get_classes(*classes: Optional[Union[Sequence[str], str]] = None*) → Sequence[str]
Get class names of current dataset.

Parameters **classes** (*Sequence[str] | str | None*) – Three types of input will correspond to different processing logics:

- If *classes* is a tuple or list, it will override the CLASSES predefined in the dataset.
- If *classes* is None, we directly use pre-defined CLASSES will be used by the dataset.
- If *classes* is a string, it is the path of a classes file that contains the name of all classes. Each line of the file contains a single class name.

Returns Names of categories of the dataset.

Return type tuple[str] or list[str]

load_annotations() → List
Load annotation according to the classes subset.

class mmfewshot.classification.datasets.**TieredImageNetDataset**(*subset: typing_extensions.Literal[train, test, val] = 'train'*, **args*, ***kwargs*)

TieredImageNet dataset for few shot classification.

Parameters **subset** (*str| list[str]*) – The classes of whole dataset are split into three disjoint subset: train, val and test. If subset is a string, only one subset data will be loaded. If subset is a list of string, then all data of subset in list will be loaded. Options: ['train', 'val', 'test']. Default: 'train'.

get_classes(*classes: Optional[Union[Sequence[str], str]] = None*) → Sequence[str]
Get class names of current dataset.

Parameters **classes** (*Sequence[str] | str | None*) – Three types of input will correspond to different processing logics:

- If *classes* is a tuple or list, it will override the CLASSES predefined in the dataset.
- If *classes* is None, we directly use pre-defined CLASSES will be used by the dataset.
- If *classes* is a string, it is the path of a classes file that contains the name of all classes. Each line of the file contains a single class name.

Returns Names of categories of the dataset.

Return type tuple[str] or list[str]

get_general_classes() → List[str]
Get general classes of each classes.

load_annotations() → List[Dict]
Load annotation according to the classes subset.

mmfewshot.classification.datasets.**build_dataloader**(*dataset: torch.utils.data.dataset.Dataset*, *samples_per_gpu: int*, *workers_per_gpu: int*, *num_gpus: int = 1*, *dist: bool = True*, *shuffle: bool = True*, *round_up: bool = True*, *seed: Optional[int] = None*, *pin_memory: bool = False*, *use_infinite_sampler: bool = False*, ***kwargs*) → torch.utils.data.dataloader.DataLoader

Build PyTorch DataLoader.

In distributed training, each GPU/process has a dataloader. In non-distributed training, there is only one dataloader for all GPUs.

Parameters

- **dataset** (*Dataset*) – A PyTorch dataset.
- **samples_per_gpu** (*int*) – Number of training samples on each GPU, i.e., batch size of each GPU.
- **workers_per_gpu** (*int*) – How many subprocesses to use for data loading for each GPU.
- **num_gpus** (*int*) – Number of GPUs. Only used in non-distributed training.
- **dist** (*bool*) – Distributed training/test or not. Default: True.
- **shuffle** (*bool*) – Whether to shuffle the data at every epoch. Default: True.
- **round_up** (*bool*) – Whether to round up the length of dataset by adding extra samples to make it evenly divisible. Default: True.
- **seed** (*int | None*) – Random seed. Default:None.
- **pin_memory** (*bool*) – Whether to use pin_memory for dataloader. Default: False.
- **use_infinite_sampler** (*bool*) – Whether to use infinite sampler. Noted that infinite sampler will keep iterator of dataloader running forever, which can avoid the overhead of worker initialization between epochs. Default: False.
- **kwargs** – any keyword argument to be used to initialize DataLoader

Returns A PyTorch dataloader.

Return type DataLoader

mmfewshot.classification.datasets.**build_meta_test_dataloader**(*dataset: torch.utils.data.dataset.Dataset*, *meta_test_cfg: Dict*, ***kwargs*) → torch.utils.data.dataloader.DataLoader

Build PyTorch DataLoader.

In distributed training, each GPU/process has a dataloader. In non-distributed training, there is only one dataloader for all GPUs.

Parameters

- **dataset** (*Dataset*) – A PyTorch dataset.
- **meta_test_cfg** (*dict*) – Config of meta testing.
- **kwargs** – any keyword argument to be used to initialize DataLoader

Returns

support_data_loader*, *query_data_loader and *test_set_data_loader*.

Return type tuple[Dataloader]

mmfewshot.classification.datasets.**label_wrapper**(*labels: Union[torch.Tensor, numpy.ndarray, List]*, *class_ids: List[int]*) → Union[torch.Tensor, numpy.ndarray, list]

Map input labels into range of 0 to numbers of classes-1.

It is usually used in the meta testing phase, in which the class ids are random sampled and discontinuous.

Parameters

- **labels** (*Tensor | np.ndarray | list*) – The labels to be wrapped.
- **class_ids** (*list[int]*) – All class ids of labels.

Returns Same type as the input labels.

Return type (Tensor | np.ndarray | list)

21.4 classification.models

21.5 classification.utils

class mmfewshot.classification.utils.**MetaTestParallel**(*module: torch.nn.modules.module.Module*, *dim: int = 0*)

The MetaTestParallel module that supports DataContainer.

Note that each task is tested on a single GPU. Thus the data and model on different GPU should be independent. MMDistributedDataParallel always automatically synchronizes the grad in different GPUs when doing the loss backward, which can not meet the requirements. Thus we simply copy the module and wrap it with an *MetaTestParallel*, which will send data to the device model.

MetaTestParallel has two main differences with PyTorch DataParallel:

- It supports a custom type DataContainer which allows more flexible control of input data during both GPU and CPU inference.
- It implement three more APIs before_meta_test(), before_forward_support() and before_forward_query().

Parameters

- **module** (nn.Module) – Module to be encapsulated.
- **dim** (*int*) – Dimension used to scatter the data. Defaults to 0.

forward(**inputs*, ***kwargs*)

Override the original forward function.

The main difference lies in the CPU inference where the data in DataContainers will still be gathered.

CHAPTER

TWENTYTWO

MMFEWSHOT.DETECTION

22.1 detection.apis

mmfewshot.detection.apis.**inference_detector**(*model: torch.nn.modules.module.Module*, *imgs: Union[List[str], str]*) → List

Inference images with the detector.

Parameters

- **model** (*nn.Module*) – Detector.
- **imgs** (*list[str] | str*) – Batch or single image file.

Returns

If imgs is a list or tuple, the same length list type results will be returned, otherwise return the detection results directly.

Return type list

mmfewshot.detection.apis.**init_detector**(*config: Union[str, mmcv.utils.config.Config]*, *checkpoint: Optional[str] = None*, *device: str = 'cuda:0'*, *cfg_options: Optional[Dict] = None*, *classes: Optional[List[str]] = None*) → torch.nn.modules.module.Module

Prepare a detector from config file.

Parameters

- **config** (str | mmcv.Config) – Config file path or the config object.
- **checkpoint** (*str | None*) – Checkpoint path. If left as None, the model will not load any weights.
- **device** (*str*) – Runtime device. Default: 'cuda:0'.
- **cfg_options** (*dict | None*) – Options to override some settings in the used config.
- **classes** (*list[str] | None*) – Options to override classes name of model. Default: None.

Returns The constructed detector.

Return type nn.Module

mmfewshot.detection.apis.**multi_gpu_model_init**(*model: torch.nn.modules.module.Module*, *data_loader: torch.utils.data.dataloader.DataLoader*) → List

Forward support images for meta-learning based detector initialization.

The function usually will be called before *single_gpu_test* in *QuerySupportEvalHook*. It firstly forwards support images with *mode=model_init* and the features will be saved in the model. Then it will call *:func:model_init* to process the extracted features of support images to finish the model initialization.

Noted that the *data_loader* should NOT use distributed sampler, all the models in different gpus should be initialized with same images.

Parameters

- **model** (*nn.Module*) – Model used for extracting support template features.
- **data_loader** (*nn.Dataloader*) – Pytorch data loader.

Returns Extracted support template features.

Return type list[Tensor]

mmfewshot.detection.apis.**multi_gpu_test**(*model: torch.nn.modules.module.Module*, *data_loader: torch.utils.data.dataloader.DataLoader*, *tmpdir: Optional[str] = None*, *gpu_collect: bool = False*) → List

Test model with multiple gpus for meta-learning based detector.

The model forward function requires *mode*, while in mmdet it requires *return_loss*. And the *encode_mask_results* is removed. This method tests model with multiple gpus and collects the results under two different modes: gpu and cpu modes. By setting 'gpu_collect=True' it encodes results to gpu tensors and use gpu communication for results collection. On cpu mode it saves the results on different gpus to 'tmpdir' and collects them by the rank 0 worker.

Parameters

- **model** (*nn.Module*) – Model to be tested.
- **data_loader** (*Dataloader*) – Pytorch data loader.
- **tmpdir** (*str*) – Path of directory to save the temporary results from different gpus under cpu mode. Default: None.
- **gpu_collect** (*bool*) – Option to use either gpu or cpu to collect results. Default: False.

Returns The prediction results.

Return type list

mmfewshot.detection.apis.**process_support_images**(*model: torch.nn.modules.module.Module*, *support_imgs: List[str]*, *support_labels: List[List[str]]*, *support_bboxes: Optional[List[List[float]]] = None*, *classes: Optional[List[str]] = None*) → None

Process support images for query support detector.

Parameters

- **model** (*nn.Module*) – Detector.
- **support_imgs** (*list[str]*) – Support image filenames.
- **support_labels** (*list[list[str]]*) – Support labels of each bbox.
- **support_bboxes** (*list[list[list[float]]] | None*) – Bbox in support images. If it set to None, it will use the [0, 0, image width, image height] as bbox. Default: None.
- **classes** (*list[str] | None*) – Options to override classes name of model. Default: None.

mmfewshot.detection.apis.**single_gpu_model_init**(*model: torch.nn.modules.module.Module*, *data_loader: torch.utils.data.dataloader.DataLoader*) → List

Forward support images for meta-learning based detector initialization.

The function usually will be called before *single_gpu_test* in *QuerySupportEvalHook*. It firstly forwards support images with *mode=model_init* and the features will be saved in the model. Then it will call *:func:model_init* to process the extracted features of support images to finish the model initialization.

Parameters

- **model** (*nn.Module*) – Model used for extracting support template features.
- **data_loader** (*nn.Dataloader*) – Pytorch data loader.

Returns Extracted support template features.

Return type list[Tensor]

mmfewshot.detection.apis.**single_gpu_test**(*model: torch.nn.modules.module.Module*, *data_loader: torch.utils.data.dataloader.DataLoader*, *show: bool = False*, *out_dir: Optional[str] = None*, *show_score_thr: float = 0.3*) → List

Test model with single gpu for meta-learning based detector.

The model forward function requires *mode*, while in mmdet it requires *return_loss*. And the *encode_mask_results* is removed.

Parameters

- **model** (*nn.Module*) – Model to be tested.
- **data_loader** (*DataLoader*) – Pytorch data loader.
- **show** (*bool*) – Whether to show the image. Default: False.
- **out_dir** (*str | None*) – The directory to write the image. Default: None.
- **show_score_thr** (*float*) – Minimum score of bboxes to be shown. Default: 0.3.

Returns The prediction results.

Return type list

22.2 detection.core

22.3 detection.datasets

class mmfewshot.detection.datasets.**BaseFewShotDataset**(*ann_cfg: List[Dict]*, *classes: Optional[Union[str, Sequence[str]]]*, *pipeline: Optional[List[Dict]] = None*, *multi_pipelines: Optional[Dict[str, List[Dict]]] = None*, *data_root: Optional[str] = None*, *img_prefix: str = ''*, *seg_prefix: Optional[str] = None*, *proposal_file: Optional[str] = None*, *test_mode: bool = False*, *filter_empty_gt: bool = True*, *min_bbox_size: Optional[Union[float, int]] = None*, *ann_shot_filter: Optional[Dict] = None*, *instance_wise: bool = False*, *dataset_name: Optional[str] = None*)

Base dataset for few shot detection.

The main differences with normal detection dataset fall in two aspects.

- **It allows to specify single (used in normal dataset) or multiple** (used in query-support dataset) pipelines for data processing.
- **It supports to control the maximum number of instances of each class** when loading the annotation file.

The annotation format is shown as follows. The *ann* field is optional for testing.

```
[
    {
        'id': '0000001'
        'filename': 'a.jpg',
        'width': 1280,
        'height': 720,
        'ann': {
            'bboxes': <np.ndarray> (n, 4) in (x1, y1, x2, y2) order.
            'labels': <np.ndarray> (n, ),
            'bboxes_ignore': <np.ndarray> (k, 4), (optional field)
            'labels_ignore': <np.ndarray> (k, 4) (optional field)
        }
    },
    ...
]
```

Parameters

- **ann_cfg** (*list[dict]*) – Annotation config support two type of config.
  - loading annotation from common ann_file of dataset with or without specific classes. example:dict(type='ann_file', ann_file='path/to/ann_file', ann_classes=['dog', 'cat'])
  - loading annotation from a json file saved by dataset. example:dict(type='saved_dataset', ann_file='path/to/ann_file')

- **classes** (*str | Sequence[str] | None*) – Classes for model training and provide fixed label for each class.
- **pipeline** (*list[dict] | None*) – Config to specify processing pipeline. Used in normal dataset. Default: None.
- **multi_pipelines** (*dict[list[dict]]*) – Config to specify data pipelines for corresponding data flow. For example, query and support data can be processed with two different pipelines, the dict should contain two keys like:
  - query (list[dict]): Config for query-data process pipeline.
  - support (list[dict]): Config for support-data process pipeline.
- **data_root** (*str | None*) – Data root for `ann_cfg`, *img_prefix*`, `seg_prefix`, `proposal_file` if specified. Default: None.
- **test_mode** (*bool*) – If set True, annotation will not be loaded. Default: False.
- **filter_empty_gt** (*bool*) – If set true, images without bounding boxes of the dataset's classes will be filtered out. This option only works when *test_mode=False*, i.e., we never filter images during tests. Default: True.
- **min_bbox_size** (*int | float | None*) – The minimum size of bounding boxes in the images. If the size of a bounding box is less than `min_bbox_size`, it would be added to ignored field. Default: None.
- **ann_shot_filter** (*dict | None*) – Used to specify the class and the corresponding maximum number of instances when loading the annotation file. For example: {'dog': 10, 'person': 5}. If set it as None, all annotation from ann file would be loaded. Default: None.
- **instance_wise** (*bool*) – If set true, *self.data_infos* would change to instance-wise, which means if the annotation of single image has more than one instance, the annotation would be split to num_instances items. Often used in support datasets, Default: False.
- **dataset_name** (*str | None*) – Name of dataset to display. For example: 'train_dataset' or 'query_dataset'. Default: None.

ann_cfg_parser(*ann_cfg: List[Dict]*) → List[Dict]
Parse annotation config to annotation information.

Parameters **ann_cfg** (*list[dict]*) – Annotation config support two type of config.

- **'ann_file': loading annotation from common ann_file of** dataset. example: dict(type='ann_file', ann_file='path/to/ann_file', ann_classes=['dog', 'cat'])
- **'saved_dataset': loading annotation from saved dataset.** example:dict(type='saved_dataset', ann_file='path/to/ann_file')

Returns Annotation information.

Return type list[dict]

get_ann_info(*idx: int*) → Dict
Get annotation by index.

When override this function please make sure same annotations are used during the whole training.

Parameters **idx** (*int*) – Index of data.

Returns Annotation info of specified index.

Return type dict

load_annotations_saved(*ann_file: str*) → List[Dict]
Load data_infos from saved json.

prepare_train_img(*idx: int*, *pipeline_key: Optional[str] = None*, *gt_idx: Optional[List[int]] = None*) → Dict
Get training data and annotations after pipeline.

Parameters

- **idx** (*int*) – Index of data.
- **pipeline_key** (*str*) – Name of pipeline
- **gt_idx** (*list[int]*) – Index of used annotation.

Returns Training data and annotation after pipeline with new keys introduced by pipeline.

Return type dict

save_data_infos(*output_path: str*) → None
Save data_infos into json.

class mmfewshot.detection.datasets.**CropResizeInstance**(*num_context_pixels: int = 16*, *target_size: Tuple[int] = (320, 320)*)
Crop and resize instance according to bbox form image.

Parameters

- **num_context_pixels** (*int*) – Padding pixel around instance. Default: 16.
- **target_size** (*tuple[int, int]*) – Resize cropped instance to target size. Default: (320, 320).

class mmfewshot.detection.datasets.**FewShotCocoDataset**(*classes: Optional[Union[Sequence[str], str]] = None*, *num_novel_shots: Optional[int] = None*, *num_base_shots: Optional[int] = None*, *ann_shot_filter: Optional[Dict[str, int]] = None*, *min_bbox_area: Optional[Union[float, int]] = None*, *dataset_name: Optional[str] = None*, *test_mode: bool = False*, ***kwargs*)
COCO dataset for few shot detection.

Parameters

- **classes** (*str | Sequence[str] | None*) – Classes for model training and provide fixed label for each class. When classes is string, it will load pre-defined classes in *FewShotCocoDataset*. For example: 'BASE_CLASSES', 'NOVEL_CLASSES` or *ALL_CLASSES*.
- **num_novel_shots** (*int | None*) – Max number of instances used for each novel class. If is None, all annotation will be used. Default: None.
- **num_base_shots** (*int | None*) – Max number of instances used for each base class. If is None, all annotation will be used. Default: None.
- **ann_shot_filter** (*dict | None*) – Used to specify the class and the corresponding maximum number of instances when loading the annotation file. For example: {'dog': 10, 'person': 5}. If set it as None, *ann_shot_filter* will be created according to *num_novel_shots* and *num_base_shots*.
- **min_bbox_area** (*int | float | None*) – Filter images with bbox whose area smaller *min_bbox_area*. If set to None, skip this filter. Default: None.

- **dataset_name** (*str | None*) – Name of dataset to display. For example: 'train dataset' or 'query dataset'. Default: None.
- **test_mode** (*bool*) – If set True, annotation will not be loaded. Default: False.

evaluate(*results: List[Sequence]*, *metric: Union[str, List[str]] = 'bbox'*, *logger: Optional[object] = None*, *jsonfile_prefix: Optional[str] = None*, *classwise: bool = False*, *proposal_nums: Sequence[int] = (100, 300, 1000)*, *iou_thrs: Optional[Union[float, Sequence[float]]] = None*, *metric_items: Optional[Union[str, List[str]]] = None*, *class_splits: Optional[List[str]] = None*) → Dict

Evaluation in COCO protocol and summary results of different splits of classes.

Parameters

- **results** (*list[list | tuple]*) – Testing results of the dataset.
- **metric** (*str | list[str]*) – Metrics to be evaluated. Options are 'bbox', 'proposal', 'proposal_fast'. Default: 'bbox'
- **logger** (*logging.Logger | None*) – Logger used for printing related information during evaluation. Default: None.
- **jsonfile_prefix** (*str | None*) – The prefix of json files. It includes the file path and the prefix of filename, e.g., "a/b/prefix". If not specified, a temp file will be created. Default: None.
- **classwise** (*bool*) – Whether to evaluating the AP for each class.
- **proposal_nums** (*Sequence[int]*) – Proposal number used for evaluating recalls, such as recall@100, recall@1000. Default: (100, 300, 1000).
- **iou_thrs** (*Sequence[float] | float | None*) – IoU threshold used for evaluating recalls/mAPs. If set to a list, the average of all IoUs will also be computed. If not specified, [0.50, 0.55, 0.60, 0.65, 0.70, 0.75, 0.80, 0.85, 0.90, 0.95] will be used. Default: None.
- **metric_items** (*list[str] | str | None*) – Metric items that will be returned. If not specified, `['AR@100', 'AR@300', 'AR@1000', 'AR_s@1000', 'AR_m@1000', 'AR_l@1000' ]` will be used when `metric=='proposal'`, `['mAP', 'mAP_50', 'mAP_75', 'mAP_s', 'mAP_m', 'mAP_l']` will be used when `metric=='bbox'`.
- **class_splits** – (list[str] | None): Calculate metric of classes split in COCO_SPLIT. For example: ['BASE_CLASSES', 'NOVEL_CLASSES']. Default: None.

Returns COCO style evaluation metric.

Return type dict[str, float]

get_cat_ids(*idx: int*) → List[int]

Get category ids by index.

Overwrite the function in CocoDataset.

Parameters **idx** (*int*) – Index of data.

Returns All categories in the image of specified index.

Return type list[int]

get_classes(*classes: Union[str, Sequence[str]]*) → List[str]

Get class names.

It supports to load pre-defined classes splits. The pre-defined classes splits are: ['ALL_CLASSES', 'NOVEL_CLASSES', 'BASE_CLASSES']

Parameters **classes** (*str | Sequence[str]*) – Classes for model training and provide fixed label for each class. When classes is string, it will load pre-defined classes in *FewShotCocoDataset*. For example: 'NOVEL_CLASSES'.

Returns list of class names.

Return type list[str]

load_annotations(*ann_cfg: List[Dict]*) → List[Dict]
Support to Load annotation from two type of ann_cfg.

- type of 'ann_file': COCO-style annotation file.
- type of 'saved_dataset': Saved COCO dataset json.

Parameters **ann_cfg** (*list[dict]*) – Config of annotations.

Returns Annotation infos.

Return type list[dict]

load_annotations_coco(*ann_file: str*) → List[Dict]
Load annotation from COCO style annotation file.

Parameters **ann_file** (*str*) – Path of annotation file.

Returns Annotation info from COCO api.

Return type list[dict]

class mmfewshot.detection.datasets.**FewShotVOCDataset**(*classes: Optional[Union[Sequence[str], str]] = None*, *num_novel_shots: Optional[int] = None*, *num_base_shots: Optional[int] = None*, *ann_shot_filter: Optional[Dict] = None*, *use_difficult: bool = False*, *min_bbox_area: Optional[Union[float, int]] = None*, *dataset_name: Optional[str] = None*, *test_mode: bool = False*, *coordinate_offset: List[int] = [- 1, - 1, 0, 0]*, ***kwargs*)

VOC dataset for few shot detection.

Parameters

- **classes** (*str | Sequence[str]*) – Classes for model training and provide fixed label for each class. When classes is string, it will load pre-defined classes in *FewShotVOCDataset*. For example: 'NOVEL_CLASSES_SPLIT1'.
- **num_novel_shots** (*int | None*) – Max number of instances used for each novel class. If is None, all annotation will be used. Default: None.
- **num_base_shots** (*int | None*) – Max number of instances used for each base class. When it is None, all annotations will be used. Default: None.
- **ann_shot_filter** (*dict | None*) – Used to specify the class and the corresponding maximum number of instances when loading the annotation file. For example: {'dog': 10, 'person': 5}. If set it as None, *ann_shot_filter* will be created according to *num_novel_shots* and *num_base_shots*. Default: None.
- **use_difficult** (*bool*) – Whether use the difficult annotation or not. Default: False.
- **min_bbox_area** (*int | float | None*) – Filter images with bbox whose area smaller *min_bbox_area*. If set to None, skip this filter. Default: None.

- **dataset_name** (*str | None*) – Name of dataset to display. For example: 'train dataset' or 'query dataset'. Default: None.
- **test_mode** (*bool*) – If set True, annotation will not be loaded. Default: False.
- **coordinate_offset** (*list[int]*) – The bbox annotation will add the coordinate offsets which corresponds to [x_min, y_min, x_max, y_max] during training. For testing, the gt annotation will not be changed while the predict results will minus the coordinate offsets to inverse data loading logic in training. Default: [-1, -1, 0, 0].

evaluate(*results: List[Sequence]*, *metric: Union[str, List[str]] = 'mAP'*, *logger: Optional[object] = None*, *proposal_nums: Sequence[int] = (100, 300, 1000)*, *iou_thr: Optional[Union[float, Sequence[float]]] = 0.5*, *class_splits: Optional[List[str]] = None*) → Dict

Evaluation in VOC protocol and summary results of different splits of classes.

Parameters

- **results** (*list[list | tuple]*) – Predictions of the model.
- **metric** (*str | list[str]*) – Metrics to be evaluated. Options are 'mAP', 'recall'. Default: mAP.
- **logger** (*logging.Logger | None*) – Logger used for printing related information during evaluation. Default: None.
- **proposal_nums** (*Sequence[int]*) – Proposal number used for evaluating recalls, such as recall@100, recall@1000. Default: (100, 300, 1000).
- **iou_thr** (*float | list[float]*) – IoU threshold. Default: 0.5.
- **class_splits** – (list[str] | None): Calculate metric of classes split defined in VOC_SPLIT. For example: ['BASE_CLASSES_SPLIT1', 'NOVEL_CLASSES_SPLIT1']. Default: None.

Returns AP/recall metrics.

Return type dict[str, float]

get_classes(*classes: Union[str, Sequence[str]]*) → List[str]

Get class names.

It supports to load pre-defined classes splits. The pre-defined classes splits are: ['ALL_CLASSES_SPLIT1', 'ALL_CLASSES_SPLIT2', 'ALL_CLASSES_SPLIT3',

'BASE_CLASSES_SPLIT1', 'BASE_CLASSES_SPLIT2', 'BASE_CLASSES_SPLIT3', 'NOVEL_CLASSES_SPLIT1','NOVEL_CLASSES_SPLIT2','NOVEL_CLASSES_SPLIT3']

Parameters **classes** (*str | Sequence[str]*) – Classes for model training and provide fixed label for each class. When classes is string, it will load pre-defined classes in *FewShotVOCDataset*. For example: 'NOVEL_CLASSES_SPLIT1'.

Returns List of class names.

Return type list[str]

load_annotations(*ann_cfg: List[Dict]*) → List[Dict]

Support to load annotation from two type of ann_cfg.

Parameters

- **ann_cfg** (*list[dict]*) – Support two type of config.

- **loading annotation from common ann_file of dataset** (-) – with or without specific classes. example:dict(type='ann_file', ann_file='path/to/ann_file', ann_classes=['dog', 'cat'])
- **loading annotation from a json file saved by dataset.** (-) – example:dict(type='saved_dataset', ann_file='path/to/ann_file')

Returns Annotation information.

Return type list[dict]

load_annotations_xml(*ann_file: str*, *classes: Optional[List[str]] = None*) → List[Dict]
Load annotation from XML style ann_file.

It supports using image id or image path as image names to load the annotation file.

Parameters

- **ann_file** (*str*) – Path of annotation file.
- **classes** (*list[str] | None*) – Specific classes to load form xml file. If set to None, it will use classes of whole dataset. Default: None.

Returns Annotation info from XML file.

Return type list[dict]

class mmfewshot.detection.datasets.**GenerateMask**(*target_size: Tuple[int] = (224, 224)*)
Resize support image and generate a mask.

Parameters **target_size** (*tuple[int, int]*) – Crop and resize to target size. Default: (224, 224).

class mmfewshot.detection.datasets.**NWayKShotDataloader**(*query_data_loader: torch.utils.data.dataloader.DataLoader*, *support_data_loader: torch.utils.data.dataloader.DataLoader*)
A dataloader wrapper.

It Create a iterator to generate query and support batch simultaneously. Each batch contains query data and support data, and the lengths are batch_size and (num_support_ways * num_support_shots) respectively.

Parameters

- **query_data_loader** (*DataLoader*) – DataLoader of query dataset
- **support_data_loader** (*DataLoader*) – DataLoader of support datasets.

class mmfewshot.detection.datasets.**NWayKShotDataset**(*query_dataset: mmfewshot.detection.datasets.base.BaseFewShotDataset*, *support_dataset: Optional[mmfewshot.detection.datasets.base.BaseFewShotDataset]*, *num_support_ways: int*, *num_support_shots: int*, *one_support_shot_per_image: bool = False*, *num_used_support_shots: int = 200*, *repeat_times: int = 1*)
A dataset wrapper of NWayKShotDataset.

Building NWayKShotDataset requires query and support dataset, the behavior of NWayKShotDataset is determined by *mode*. When dataset in 'query' mode, dataset will return regular image and annotations. While dataset in 'support' mode, dataset will build batch indices firstly and each batch indices contain (num_support_ways * num_support_shots) samples. In other words, for support mode every call of *__getitem__* will return a batch of

samples, therefore the outside dataloader should set batch_size to 1. The default *mode* of NWayKShotDataset is 'query' and by using convert function *convert_query_to_support* the *mode* will be converted into 'support'.

Parameters

- **query_dataset** (*BaseFewShotDataset*) – Query dataset to be wrapped.
- **support_dataset** (*BaseFewShotDataset* | None) – Support dataset to be wrapped. If support dataset is None, support dataset will copy from query dataset.
- **num_support_ways** (*int*) – Number of classes for support in mini-batch.
- **num_support_shots** (*int*) – Number of support shot for each class in mini-batch.
- **one_support_shot_per_image** (*bool*) – If True only one annotation will be sampled from each image. Default: False.
- **num_used_support_shots** (*int | None*) – The total number of support shots sampled and used for each class during training. If set to None, all shots in dataset will be used as support shot. Default: 200.
- **shuffle_support** (*bool*) – If allow generate new batch indices for each epoch. Default: False.
- **repeat_times** (*int*) – The length of repeated dataset will be *times* larger than the original dataset. Default: 1.

convert_query_to_support(*support_dataset_len: int*) → None
Convert query dataset to support dataset.

Parameters **support_dataset_len** (*int*) – Length of pre sample batch indices.

generate_support_batch_indices(*dataset_len: int*) → List[List[Tuple[int]]]
Generate batch indices from support dataset.

Batch indices is in the shape of [length of datasets * [support way * support shots]]. And the *dataset_len* will be the length of support dataset.

Parameters **dataset_len** (*int*) – Length of batch indices.

Returns Pre-sample batch indices.

Return type list[list[(data_idx, gt_idx)]]

get_support_data_infos() → List[Dict]
Get support data infos from batch indices.

save_data_infos(*output_path: str*) → None
Save data infos of query and support data.

save_support_data_infos(*support_output_path: str*) → None
Save support data infos.

class mmfewshot.detection.datasets.**NumpyEncoder**(*, *skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *sort_keys=False*, *indent=None*, *separators=None*, *default=None*)

Save numpy array obj to json.

default(*obj: object*) → object
Implement this method in a subclass such that it returns a serializable object for `o`, or calls the base implementation (to raise a `TypeError`).

For example, to support arbitrary iterators, you could implement default like this:

```
def default(self, o):
    try:
        iterable = iter(o)
    except TypeError:
        pass
    else:
        return list(iterable)
    # Let the base class default method raise the TypeError
    return JSONEncoder.default(self, o)
```

class mmfewshot.detection.datasets.**QueryAwareDataset**(*query_dataset:* mmfewshot.detection.datasets.base.BaseFewShotDataset, *support_dataset: Optional[*mmfewshot.detection.datasets.base.BaseFewShotDataset*]*, *num_support_ways: int*, *num_support_shots: int*, *repeat_times: int = 1*)

A wrapper of QueryAwareDataset.

Building QueryAwareDataset requires query and support dataset. Every call of *__getitem__* will firstly sample a query image and its annotations. Then it will use the query annotations to sample a batch of positive and negative support images and annotations. The positive images share same classes with query, while the annotations of negative images don't have any category from query.

Parameters

- **query_dataset** (*BaseFewShotDataset*) – Query dataset to be wrapped.
- **support_dataset** (*BaseFewShotDataset* | None) – Support dataset to be wrapped. If support dataset is None, support dataset will copy from query dataset.
- **num_support_ways** (*int*) – Number of classes for support in mini-batch, the first one always be the positive class.
- **num_support_shots** (*int*) – Number of support shots for each class in mini-batch, the first K shots always from positive class.
- **repeat_times** (*int*) – The length of repeated dataset will be *times* larger than the original dataset. Default: 1.

generate_support(*idx: int*, *query_class: int*, *support_classes: List[int]*) → List[Tuple[int]]

Generate support indices of query images.

Parameters

- **idx** (*int*) – Index of query data.
- **query_class** (*int*) – Query class.
- **support_classes** (*list[int]*) – Classes of support data.

Returns

A mini-batch (num_support_ways * num_support_shots) of support data (idx, gt_idx).

Return type list[tuple(int)]

get_support_data_infos() → List[Dict]

Return data_infos of support dataset.

sample_support_shots(*idx: int*, *class_id: int*, *allow_same_image: bool = False*) → List[Tuple[int]]

Generate support indices according to the class id.

Parameters

- **idx** (*int*) – Index of query data.
- **class_id** (*int*) – Support class.
- **allow_same_image** (*bool*) – Allow instance sampled from same image as query image. Default: False.

Returns

Support data (num_support_shots) of specific class.

Return type list[tuple[int]]

save_data_infos(*output_path: str*) → None
Save data_infos into json.

mmfewshot.detection.datasets.**build_dataloader**(*dataset: torch.utils.data.dataset.Dataset*, *samples_per_gpu: int*, *workers_per_gpu: int*, *num_gpus: int = 1*, *dist: bool = True*, *shuffle: bool = True*, *seed: Optional[int] = None*, *data_cfg: Optional[Dict] = None*, *use_infinite_sampler: bool = False*, ***kwargs*) → torch.utils.data.dataloader.DataLoader
Build PyTorch DataLoader.

In distributed training, each GPU/process has a dataloader. In non-distributed training, there is only one dataloader for all GPUs.

Parameters

- **dataset** (*Dataset*) – A PyTorch dataset.
- **samples_per_gpu** (*int*) – Number of training samples on each GPU, i.e., batch size of each GPU.
- **workers_per_gpu** (*int*) – How many subprocesses to use for data loading for each GPU.
- **num_gpus** (*int*) – Number of GPUs. Only used in non-distributed training. Default:1.
- **dist** (*bool*) – Distributed training/test or not. Default: True.
- **shuffle** (*bool*) – Whether to shuffle the data at every epoch. Default: True.
- **seed** (*int*) – Random seed. Default:None.
- **data_cfg** (*dict | None*) – Dict of data configure. Default: None.
- **use_infinite_sampler** (*bool*) – Whether to use infinite sampler. Noted that infinite sampler will keep iterator of dataloader running forever, which can avoid the overhead of worker initialization between epochs. Default: False.
- **kwargs** – any keyword argument to be used to initialize DataLoader

Returns A PyTorch dataloader.

Return type DataLoader

mmfewshot.detection.datasets.**get_copy_dataset_type**(*dataset_type: str*) → str
Return corresponding copy dataset type.

22.4 detection.models

mmfewshot.detection.models.**build_backbone**(*cfg*)
Build backbone.

mmfewshot.detection.models.**build_detector**(*cfg: mmcv.utils.config.ConfigDict*, *logger: Optional[object] = None*)
Build detector.

mmfewshot.detection.models.**build_head**(*cfg*)
Build head.

mmfewshot.detection.models.**build_loss**(*cfg*)
Build loss.

mmfewshot.detection.models.**build_neck**(*cfg*)
Build neck.

mmfewshot.detection.models.**build_roi_extractor**(*cfg*)
Build roi extractor.

mmfewshot.detection.models.**build_shared_head**(*cfg*)
Build shared head.

22.5 detection.utils

class mmfewshot.detection.utils.**ContrastiveLossDecayHook**(*decay_steps: Sequence[int]*, *decay_rate: float = 0.5*)
Hook for contrast loss weight decay used in FSCE.

Parameters

- **decay_steps** (*list[int] | tuple[int]*) – Each item in the list is the step to decay the loss weight.
- **decay_rate** (*float*) – Decay rate. Default: 0.5.

CHAPTER

TWENTYTHREE

MMFEWSHOT.UTILS

class mmfewshot.utils.**DistributedInfiniteGroupSampler**(*dataset: Iterable*, *samples_per_gpu: int = 1*, *num_replicas: Optional[int] = None*, *rank: Optional[int] = None*, *seed: int = 0*, *shuffle: bool = True*)

Similar to *InfiniteGroupSampler* but in distributed version.

The length of sampler is set to the actual length of dataset, thus the length of dataloader is still determined by the dataset. The implementation logic is referred to https://github.com/facebookresearch/detectron2/blob/main/detectron2/data/samplers/grouped_batch_sampler.py

Parameters

- **dataset** (*Iterable*) – The dataset.
- **samples_per_gpu** (*int*) – Number of training samples on each GPU, i.e., batch size of each GPU. Default: 1.
- **num_replicas** (*int | None*) – Number of processes participating in distributed training. Default: None.
- **rank** (*int | None*) – Rank of current process. Default: None.
- **seed** (*int*) – Random seed. Default: 0.
- **shuffle** (*bool*) – Whether shuffle the indices of a dummy *epoch*, it should be noted that *shuffle* can not guarantee that you can generate sequential indices because it need to ensure that all indices in a batch is in a group. Default: True.

class mmfewshot.utils.**DistributedInfiniteSampler**(*dataset: Iterable*, *num_replicas: Optional[int] = None*, *rank: Optional[int] = None*, *seed: int = 0*, *shuffle: bool = True*)

Similar to *InfiniteSampler* but in distributed version.

The length of sampler is set to the actual length of dataset, thus the length of dataloader is still determined by the dataset. The implementation logic is referred to https://github.com/facebookresearch/detectron2/blob/main/detectron2/data/samplers/grouped_batch_sampler.py

Parameters

- **dataset** (*Iterable*) – The dataset.
- **num_replicas** (*int | None*) – Number of processes participating in distributed training. Default: None.
- **rank** (*int | None*) – Rank of current process. Default: None.
- **seed** (*int*) – Random seed. Default: 0.
- **shuffle** (*bool*) – Whether shuffle the dataset or not. Default: True.

class mmfewshot.utils.**InfiniteEpochBasedRunner**(*model*, *batch_processor=None*, *optimizer=None*, *work_dir=None*, *logger=None*, *meta=None*, *max_iters=None*, *max_epochs=None*)

Epoch-based Runner supports dataloader with InfiniteSampler.

The workers of dataloader will re-initialize, when the iterator of dataloader is created. InfiniteSampler is designed to avoid these time consuming operations, since the iterator with InfiniteSampler will never reach the end.

class mmfewshot.utils.**InfiniteGroupSampler**(*dataset: Iterable*, *samples_per_gpu: int = 1*, *seed: int = 0*, *shuffle: bool = True*)

Similar to *InfiniteSampler*, but all indices in a batch should be in the same group of flag.

The length of sampler is set to the actual length of dataset, thus the length of dataloader is still determined by the dataset. The implementation logic is referred to https://github.com/facebookresearch/detectron2/blob/main/detectron2/data/samplers/grouped_batch_sampler.py

Parameters

- **dataset** (*Iterable*) – The dataset.
- **samples_per_gpu** (*int*) – Number of training samples on each GPU, i.e., batch size of each GPU. Default: 1.
- **seed** (*int*) – Random seed. Default: 0.
- **shuffle** (*bool*) – Whether shuffle the indices of a dummy *epoch*, it should be noted that *shuffle* can not guarantee that you can generate sequential indices because it need to ensure that all indices in a batch is in a group. Default: True.

class mmfewshot.utils.**InfiniteSampler**(*dataset: Iterable*, *seed: int = 0*, *shuffle: bool = True*)

Return a infinite stream of index.

The length of sampler is set to the actual length of dataset, thus the length of dataloader is still determined by the dataset. The implementation logic is referred to https://github.com/facebookresearch/detectron2/blob/main/detectron2/data/samplers/grouped_batch_sampler.py

Parameters

- **dataset** (*Iterable*) – The dataset.
- **seed** (*int*) – Random seed. Default: 0.
- **shuffle** (*bool*) – Whether shuffle the dataset or not. Default: True.

mmfewshot.utils.**local_numpy_seed**(*seed: Optional[int] = None*) → None

Run numpy codes with a local random seed.

If seed is None, the default random state will be used.

mmfewshot.utils.**multi_pipeline_collate_fn**(*batch*, *samples_per_gpu: int = 1*)

Puts each data field into a tensor/DataContainer with outer dimension batch size. This is designed to support the case that the `__getitem__()` of dataset return more than one images, such as query_support dataloader. The main difference with the `collate_fn()` in mmcv is it can process list[list[DataContainer]].

Extend default_collate to add support for **:type:`~mmcv.parallel.DataContainer`**. There are 3 cases:

1. cpu_only = True, e.g., meta data.
2. cpu_only = False, stack = True, e.g., images tensors.
3. cpu_only = False, stack = False, e.g., gt bboxes.

:param batch (list[list[mmcv.parallel.DataContainer]] |: list[mmcv.parallel.DataContainer]): Data of single batch.

Parameters **samples_per_gpu** (*int*) – The number of samples of single GPU.

CHAPTER

TWENTYFOUR

INDICES AND TABLES

- genindex
- search

PYTHON MODULE INDEX

m

INDEX

I

L

M

N

P

Q

S

T