mmfewshot

MMFewShot Author

Apr 20, 2022

LEARN THE BASICS

1 Learn the Basics 1

2 Prerequisites 5

3 Installation 7

4 Verification 11

5 Dataset Preparation 13

6 Test a model 15

7 Train a model 17

8 Model Zoo 19

9 Tutorial 0: Overview of MMFewShot Classification 21

10 Tutorial 1: Learn about Configs 25

11 Tutorial 2: Adding New Dataset 33

12 Tutorial 3: Customize Models 39

13 Tutorial 4: Customize Runtime Settings 45

14 Tutorial 0: Overview of MMFewShot Detection 53

15 Tutorial 1: Learn about Configs 55

16 Tutorial 2: Adding New Dataset 67

17 Tutorial 3: Customize Models 73

18 Tutorial 4: Customize Runtime Settings 83

19 Changelog 91

20 Frequently Asked Questions 93

21 English 97

22 99

23 mmfewshot.classification 101

24 mmfewshot.detection 121

25 mmfewshot.utils 159

26 Indices and tables 163

Python Module Index 165

Index 167

CHAPTER

ONE

LEARN THE BASICS

This chapter introduces you to the basic conception of few shot learning, and the framework of MMFewShot, and provides links to detailed tutorials about MMFewShot.

1.1 What is Few Shot Learning

1.1.1 Problem definition

Few shot learning aims at generalizing to new tasks based on a limited number of samples using prior knowledge. The prior knowledge usually refers to a large scale training set that has many classes and samples, while the samples in new tasks are never seen in the training set. For example, in few shot image classification, a pre-trained model only can see five bird images (each class has one image and doesn't exist in the pretrained dataset) and predict the class of bird in the query image. Another example in few shot detection is that a detector needs to detect the new categories based on a few instances.

In summary, few shot learning focus on two aspects:

- how to embed prior knowledge into models (pre-train with large scale dataset)
- how to transfer knowledge to adapt to new tasks (learn on a few labeled samples).

1.1.2 Terminologies in few-shot learning

- Training set: every class in the training set has many samples, and it is big enough for training a deep neural network.
- Support set: a small set of labeled images and all the classes do not exist in the training set.
- Query set: unlabeled images to predict and share the same classes with support set.
- N way K shot: the support set setting, and it means support images contain N classes and each class has K samples.
  - For classification, there will be NxK support images in a support set.
  - For detection, there will be NxK support instances in a support set, and the number of images can be less than NxK.

1.1.3 Evaluation

Few shot classification

The classes of a dataset will be divided into three disjoint groups: train, test and val set. The evaluation also called meta test, will randomly sample (N way x K shot) labeled support images + Q unlabeled query images from the test set to form a task and get the prediction accuracy of query images in that task. Usually, meta test will repeatedly sample numerous tasks to get a sufficient evaluation and calculate the mean and std of accuracy from all tasks.

Few shot detection

The classes of dataset are split into two group, base classes and novel classes. The training set contains all the annotations from base classes and a few annotations from novel classes. The novel classes performance (mAP or AP50) on test set are used for evaluating a few shot detector.

1.1.4 The basic pipeline for few shot learning

We will introduce a simple baseline for all the few shot learning tasks to further illustrate how few shot learning work. The most obvious pipeline is fine-tuning. It usually consists of two steps: train a model on a large scale dataset and then fine-tune on few shot data. For image classification, we first pretrain a model with training set using cross-entropy loss, and then we can transfer the backbone and fine tune a new classification head. For detection, we can first pretrain a faster-rcnn on training set, and then fine tune a new bbox head on a few instances to detect the novel class. In many cases, the fine-tuning is a simple but effective strategy for few shot learning.

1.2 What is MMFewShot

MMFewShot is the first toolbox that provides a framework for unified implementation and evaluation of few shot classification and detection methods, and below is its whole framework:

MMFewShot consists of 4 main parts, `datasets`, `models`, `core` and `apis`.

- `datasets` is for data loading and data augmentation. In this part, we support various datasets for classification and detection algorithms, useful data augmentation transforms in `pipelines` for pre-processing image and flexible data sampling in `datasetswrappers`.
- `models` contains models and loss functions.
- `core` provides evaluation tools and customized hooks for model training and evaluation.
- `apis` provides high-level APIs for models training, testing, and inference.

1.3 How to Use this Guide

Here is a detailed step-by-step guide to learn more about MMFewShot:

1. For installation instructions, please see *install*.
2. *get_started* is for the basic usage of MMFewShot.
3. Refer to the below tutorials to dive deeper:

- Few Shot Classification
  - *Overview*

  - *Config*
  - *Customize Dataset*
  - *Customize Model*
  - *Customize Runtime*
- Few Shot Detection
  - *Overview*
  - *Config*
  - *Customize Dataset*
  - *Customize Model*
  - *Customize Runtime*

CHAPTER

TWO

PREREQUISITES

- Linux | Windows | macOS
- 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

THREE

INSTALLATION

3.1 A from-scratch setup script

Assuming that you already have CUDA 10.1 installed, here is a full script for setting up MMFewShot with conda. You can refer to the step-by-step installation instructions in the next section.

```
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

pip install openmim
mim install mmcv-full

# install mmclassification mmdetection
mim install mmcls
mim install 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"
```

3.2 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
```

3.3 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
   pip install mmcv-full -f https://download.openmmlab.com/mmcv/dist/cu102/torch1.10.0/
   ↪index.html
   ```

   mmcv-full is only compiled on PyTorch 1.x.0 because the compatibility usually holds between 1.x.0 and 1.x.1. If your PyTorch version is 1.x.1, you can install mmcv-full compiled with PyTorch 1.x.0 and it usually works well.

   ```
   # We can ignore the micro version of PyTorch
   pip install mmcv-full -f https://download.openmmlab.com/mmcv/dist/cu102/torch1.10/
   ↪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 mmfewshot. 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`.

3.4 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
```

CHAPTER

FOUR

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

FIVE

DATASET PREPARATION

Please refer to data preparation for dataset preparation.

CHAPTER

SIX

TEST A MODEL

- single GPU
- CPU
- 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]

# CPU: disable GPUs and run single-gpu testing script
export CUDA_VISIBLE_DEVICES=-1
python tools/test.py ${CONFIG_FILE} ${CHECKPOINT_FILE} [optional arguments]

# multi-gpu
sh ./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

SEVEN

TRAIN A MODEL

7.1 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}`.

7.2 Train on CPU

The process of training on the CPU is consistent with single GPU training. We just need to disable GPUs before the training process.

```
export CUDA_VISIBLE_DEVICES=-1
python tools/train.py ${CONFIG_FILE} [optional arguments]
```

Note:

We do not recommend users to use CPU for training because it is too slow. We support this feature to allow users to debug on machines without GPU for convenience.

7.3 Train with multiple GPUs

```
sh ./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.

7.4 Train with multiple machines

If you launch with multiple machines simply connected with ethernet, you can simply run following commands:

On the first machine:

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

On the second machine:

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

Usually it is slow if you do not have high speed networking like InfiniBand.

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}] sh ./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.

7.5 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 sh ./tools/dist_train.sh ${CONFIG_FILE} 4
CUDA_VISIBLE_DEVICES=4,5,6,7 PORT=29501 sh ./tools/dist_train.sh ${CONFIG_FILE} 4
```

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

CHAPTER

EIGHT

MODEL ZOO

8.1 Few Shot Classification Model Zoo

8.1.1 Baseline

Please refer to Baseline for details.

8.1.2 Baseline++

Please refer to Baseline++ for details.

8.1.3 ProtoNet

Please refer to ProtoNet for details.

8.1.4 RelationNet

Please refer to RelationNet for details.

8.1.5 MatchingNet

Please refer to MatchingNet for details.

8.1.6 MAML

Please refer to MAML for details.

8.1.7 NegMargin

Please refer to NegMargin for details.

8.1.8 Meta Baseline

Please refer to Meta Baseline for details.

8.2 Few Shot Detection Model Zoo

8.2.1 TFA

Please refer to TFA for details.

8.2.2 FSCE

Please refer to FSCE for details.

8.2.3 Meta RCNN

Please refer to Meta RCNN for details.

8.2.4 FSDetView

Please refer to FSDetView for details.

8.2.5 Attention RPN

Please refer to Attention RPN for details.

8.2.6 MPSR

Please refer to MPSR for details.

CHAPTER

NINE

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.

9.1 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

9.2 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
```

9.3 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')
```

9.3.1 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

TEN

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.

10.1 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.

10.2 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.

10.3 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
```

10.4 FAQ

10.4.1 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))
```

10.4.2 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

ELEVEN

TUTORIAL 2: ADDING NEW DATASET

11.1 Customize datasets by reorganizing data

11.1.1 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.

```
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
)
```

11.1.2 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
```

11.2 Customize datasets sampling

11.2.1 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
        )
    )
```

11.2.2 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)
```

(continues on next page)

(continued from previous page)

```
        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

TWELVE

TUTORIAL 3: CUSTOMIZE MODELS

12.1 Add a new classifier

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

12.1.1 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
```

12.1.2 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.

12.1.3 3. Use the classifier in your config file

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

12.2 Add a new backbone

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

12.2.1 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
```

12.2.2 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.

12.2.3 3. Use the backbone in your config file

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

12.3 Add new heads

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

12.3.1 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
```

12.3.2 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.

12.3.3 3. Use the head in your config file

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

12.4 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

THIRTEEN

TUTORIAL 4: CUSTOMIZE RUNTIME SETTINGS

13.1 Customize optimization settings

13.1.1 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.

13.1.2 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)
```

13.1.3 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.

13.1.4 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,
)
```

13.2 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',
```

(continues on next page)

(continued from previous page)

```
    warmup_iters=1000,
    warmup_ratio=1.0 / 10,
    min_lr_ratio=1e-5)
```

13.3 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.

13.4 Customize hooks

13.4.1 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):
```

(continues on next page)

(continued from previous page)

```
        pass

    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.

13.4.2 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')
]
```

13.4.3 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`

13.4.4 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')
```

13.5 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

FOURTEEN

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.

14.1 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:

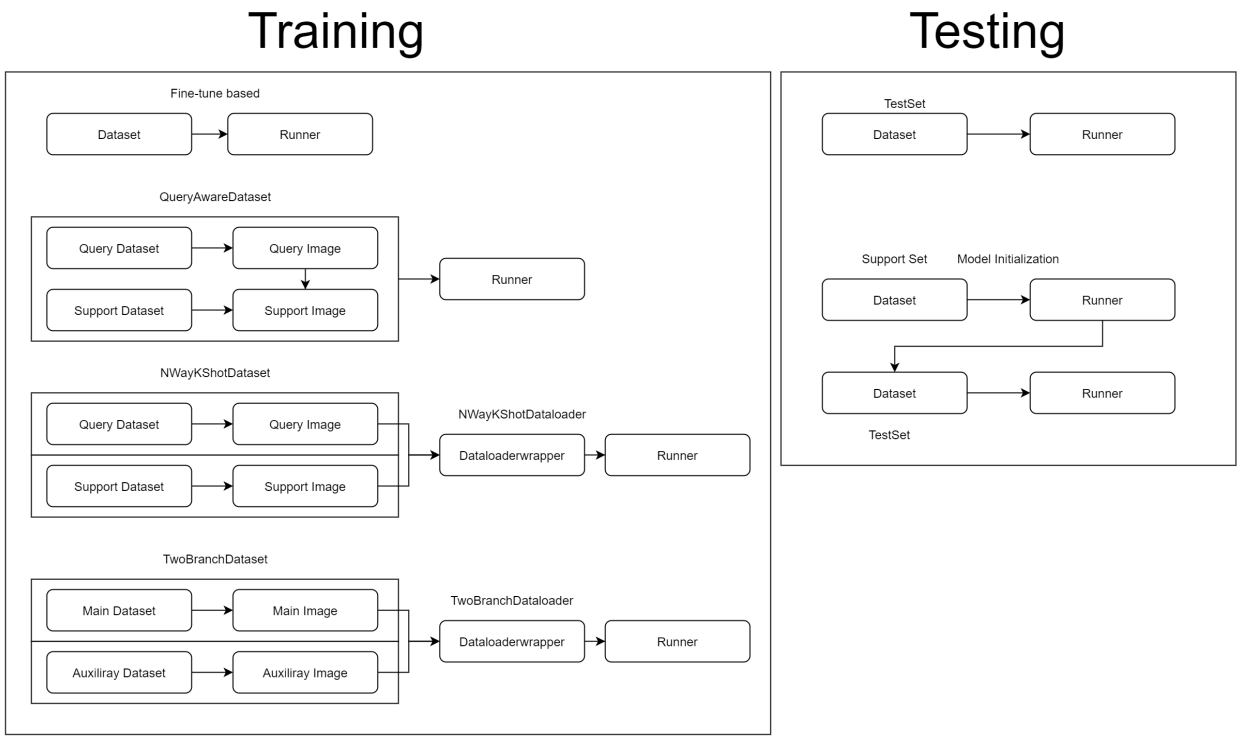
- Datasets: loading annotations from `ann_cfg` and filtering images and annotations for few shot setting.
- Dataset Wrappers: determining the sampling logic, such as sampling support images according to query image.
- Dataloader Wrappers: encapsulate the data from multiple datasets.

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

FIFTEEN

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.

15.1 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.

15.2 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.

15.3 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
```

15.4 FAQ

15.4.1 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

SIXTEEN

TUTORIAL 2: ADDING NEW DATASET

16.1 Customize dataset

16.1.1 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, ...)
```

16.1.2 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')]
)
```

16.1.3 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)])
```

16.1.4 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'])
```

16.1.5 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,
    ...)
```

16.1.6 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,
    ...)
```

16.1.7 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')
```

16.2 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:

16.2.1 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}
```

16.2.2 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)
```

(continues on next page)

(continued from previous page)

```
    return dataset
```

16.2.3 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)
    ...
```

16.2.4 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, ...):
        ...
```

16.2.5 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
        ),
```

(continues on next page)

(continued from previous page)

```
        support_dataset=None
    )
```

16.3 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:

16.3.1 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)
```

16.3.2 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

SEVENTEEN

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.

17.1 Develop new components

17.1.1 Add a new detector

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

17.1.2 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),
    ...
```

17.1.3 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)
```

17.1.4 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.

17.1.5 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)
```

(continues on next page)

(continued from previous page)

```
    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/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))
```

17.2 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'
    ])
```

17.3 Customize a query-support based detector

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

17.3.1 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):
        ...
```

17.3.2 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.

17.3.3 3. Use the detector in your config file

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

17.3.4 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

EIGHTEEN

TUTORIAL 4: CUSTOMIZE RUNTIME SETTINGS

18.1 Customize optimization settings

18.1.1 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.

18.1.2 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)
```

18.1.3 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.

18.1.4 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,
)
```

18.2 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',
```

(continues on next page)

(continued from previous page)

```
    warmup_iters=1000,
    warmup_ratio=1.0 / 10,
    min_lr_ratio=1e-5)
```

18.3 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.

18.4 Customize hooks

18.4.1 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):
```

(continues on next page)

(continued from previous page)

```
        pass

    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.

18.4.2 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')
]
```

18.4.3 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

NINETEEN

CHANGELOG

CHAPTER

TWENTY

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.

20.1 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.

20.2 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.

20.3 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.

20.4 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

TWENTYONE

ENGLISH

CHAPTER

TWENTYTWO

CHAPTER

TWENTYTHREE

MMFEWSHOT.CLASSIFICATION

23.1 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

23.2 classification.core

23.2.1 evaluation

class mmfewshot.classification.core.evaluation.**DistMetaTestEvalHook**(*support_dataloader: torch.utils.data.dataloader.DataLoader*, *query_dataloader: torch.utils.data.dataloader.DataLoader*, *test_set_dataloader: torch.utils.data.dataloader.DataLoader*, *num_test_tasks: int*, *interval: int = 1*, *by_epoch: bool = True*, *meta_test_cfg: Optional[Dict] = None*, *confidence_interval: float = 0.95*, *save_best: bool = True*, *key_indicator: str = 'accuracy_mean'*, ***eval_kwargs*)

Distributed evaluation hook.

class mmfewshot.classification.core.evaluation.**MetaTestEvalHook**(*support_dataloader: torch.utils.data.dataloader.DataLoader*, *query_dataloader: torch.utils.data.dataloader.DataLoader*, *test_set_dataloader: torch.utils.data.dataloader.DataLoader*, *num_test_tasks: int*, *interval: int = 1*, *by_epoch: bool = True*, *meta_test_cfg: Optional[Dict] = None*, *confidence_interval: float = 0.95*, *save_best: bool = True*, *key_indicator: str = 'accuracy_mean'*, ***eval_kwargs*)

Evaluation hook for Meta Testing.

Parameters

- **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.
- **num_test_tasks** (*int*) – Number of tasks for meta testing.
- **interval** (*int*) – Evaluation interval (by epochs or iteration). Default: 1.
- **by_epoch** (*bool*) – Epoch based runner or not. Default: True.
- **meta_test_cfg** (*dict*) – Config for meta testing.
- **confidence_interval** (*float*) – Confidence interval. Default: 0.95.
- **save_best** (*bool*) – Whether to save best validated model. Default: True.
- **key_indicator** (*str*) – The validation metric for selecting the best model. Default: 'accuracy_mean'.

- **eval_kwargs** – Any keyword argument to be used for evaluation.

23.3 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.

evaluate(**args*, ***kwargs*) → List
Evaluate prediction.

generate_episodic_idxes() → Tuple[List[Mapping], List[List[int]]]
Generate batch indices for each episodic.

get_episode_class_ids(*idx: int*) → List[int]
Return class ids in one episode.

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)

23.4 classification.models

23.4.1 backbones

class mmfewshot.classification.models.backbones.**Conv4**(*depth: int = 4*, *pooling_blocks: Sequence[int] = (0, 1, 2, 3)*, *padding_blocks: Sequence[int] = (0, 1, 2, 3)*, *flatten: bool = True*)

class mmfewshot.classification.models.backbones.**ConvNet**(*depth: int*, *pooling_blocks: Sequence[int]*, *padding_blocks: Sequence[int]*, *flatten: bool = True*)

Simple ConvNet.

Parameters

- **depth** (*int*) – The number of *ConvBlock*.
- **pooling_blocks** (*Sequence[int]*) – Indicate which block to use 2x2 max pooling.
- **padding_blocks** (*Sequence[int]*) – Indicate which block to use conv layer with padding.
- **flatten** (*bool*) – Whether to flatten features from (N, C, H, W) to (N, C*H*W). Default: True.

forward(*x: torch.Tensor*) → torch.Tensor

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

class mmfewshot.classification.models.backbones.**ResNet12**(*block: torch.nn.modules.module.Module = <class 'mmfewshot.classification.models.backbones.resnet12.BasicBlock'>*, *with_avgpool: bool = True*, *pool_size: Tuple[int, int] = (1, 1)*, *flatten: bool = True*, *drop_rate: float = 0.0*, *drop_block_size: int = 5*)

ResNet12.

Parameters

- **block** (*nn.Module*) – Block to build layers. Default: `BasicBlock`.
- **with_avgpool** (*bool*) – Whether to average pool the features. Default: True.
- **pool_size** (*tuple(int,int)*) – The output shape of average pooling layer. Default: (1, 1).
- **flatten** (*bool*) – Whether to flatten features from (N, C, H, W) to (N, C*H*W). Default: True.
- **drop_rate** (*float*) – Dropout rate. Default: 0.0.
- **drop_block_size** (*int*) – Size of drop block. Default: 5.

forward(*x: torch.Tensor*) → torch.Tensor

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

class mmfewshot.classification.models.backbones.**WRN28x10**(*depth: int = 28*, *widen_factor: int = 10*, *stride: int = 1*, *drop_rate: float = 0.5*, *flatten: bool = True*, *with_avgpool: bool = True*, *pool_size: Tuple[int, int] = (1, 1)*)

class mmfewshot.classification.models.backbones.**WideResNet**(*depth: int*, *widen_factor: int = 1*, *stride: int = 1*, *drop_rate: float = 0.0*, *flatten: bool = True*, *with_avgpool: bool = True*, *pool_size: Tuple[int, int] = (1, 1)*)

WideResNet.

Parameters

- **depth** (*int*) – The number of layers.
- **widen_factor** (*int*) – The widen factor of channels. Default: 1.
- **stride** (*int*) – Stride of first layer. Default: 1.
- **drop_rate** (*float*) – Dropout rate. Default: 0.0.
- **with_avgpool** (*bool*) – Whether to average pool the features. Default: True.
- **flatten** (*bool*) – Whether to flatten features from (N, C, H, W) to (N, C*H*W). Default: True.
- **pool_size** (*tuple(int,int)*) – The output shape of average pooling layer. Default: (1, 1).

forward(*x: torch.Tensor*) → torch.Tensor

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

23.4.2 classifier

23.4.3 heads

class mmfewshot.classification.models.heads.**CosineDistanceHead**(*num_classes: int*, *in_channels: int*, *temperature: Optional[float] = None*, *eps: float = 1e-05*, **args*, ***kwargs*)

Classification head for **`Baseline++ https://arxiv.org/abs/2003.04390`_**.

Parameters

- **num_classes** (*int*) – Number of categories.
- **in_channels** (*int*) – Number of channels in the input feature map.
- **temperature** (*float | None*) – Scaling factor of *cls_score*. Default: None.
- **eps** (*float*) – Constant variable to avoid division by zero. Default: 0.00001.

before_forward_query() → None
Used in meta testing.

This function will be called before model forward query data during meta testing.

before_forward_support() → None
Used in meta testing.

This function will be called before model forward support data during meta testing.

forward_query(*x: torch.Tensor*, ***kwargs*) → List
Forward query data in meta testing.

forward_support(*x: torch.Tensor*, *gt_label: torch.Tensor*, ***kwargs*) → Dict
Forward support data in meta testing.

forward_train(*x: torch.Tensor*, *gt_label: torch.Tensor*, ***kwargs*) → Dict
Forward training data.

class mmfewshot.classification.models.heads.**LinearHead**(*num_classes: int*, *in_channels: int*, **args*, ***kwargs*)

Classification head for Baseline.

Parameters

- **num_classes** (*int*) – Number of categories.
- **in_channels** (*int*) – Number of channels in the input feature map.

before_forward_query() → None
Used in meta testing.

This function will be called before model forward query data during meta testing.

before_forward_support() → None
Used in meta testing.

This function will be called before model forward support data during meta testing.

forward_query(*x: torch.Tensor*, ***kwargs*) → List
Forward query data in meta testing.

forward_support(*x: torch.Tensor*, *gt_label: torch.Tensor*, ***kwargs*) → Dict
Forward support data in meta testing.

forward_train(*x: torch.Tensor*, *gt_label: torch.Tensor*, ***kwargs*) → Dict
Forward training data.

class mmfewshot.classification.models.heads.**MatchingHead**(*temperature: float = 100*, *loss: Dict = {'loss_weight': 1.0, 'type': 'NLLLoss'}*, **args*, ***kwargs*)

Classification head for `MatchingNet.

<https://arxiv.org/abs/1606.04080>`_.

Note that this implementation is without FCE(Full Context Embeddings).

Parameters

- **temperature** (*float*) – The scale factor of *cls_score*.
- **loss** (*dict*) – Config of training loss.

before_forward_query() → None
Used in meta testing.

This function will be called before model forward query data during meta testing.

before_forward_support() → None
Used in meta testing.

This function will be called before model forward support data during meta testing.

forward_query(*x: torch.Tensor*, ***kwargs*) → List
Forward query data in meta testing.

forward_support(*x: torch.Tensor*, *gt_label: torch.Tensor*, ***kwargs*) → None
Forward support data in meta testing.

forward_train(*support_feats: torch.Tensor*, *support_labels: torch.Tensor*, *query_feats: torch.Tensor*, *query_labels: torch.Tensor*, ***kwargs*) → Dict
Forward training data.

Parameters

- **support_feats** (*Tensor*) – Features of support data with shape (N, C).
- **support_labels** (*Tensor*) – Labels of support data with shape (N).
- **query_feats** (*Tensor*) – Features of query data with shape (N, C).
- **query_labels** (*Tensor*) – Labels of query data with shape (N).

Returns A dictionary of loss components.

Return type dict[str, Tensor]

class mmfewshot.classification.models.heads.**MetaBaselineHead**(*temperature: float = 10.0*, *learnable_temperature: bool = True*, **args*, ***kwargs*)
Classification head for **`MetaBaseline https://arxiv.org/abs/2003.04390`_**.

Parameters

- **temperature** (*float*) – Scaling factor of *cls_score*. Default: 10.0.
- **learnable_temperature** (*bool*) – Whether to use learnable scale factor or not. Default: True.

before_forward_query() → None
Used in meta testing.

This function will be called before model forward query data during meta testing.

before_forward_support() → None
Used in meta testing.

This function will be called before model forward support data during meta testing.

forward_query(*x: torch.Tensor*, ***kwargs*) → List
Forward query data in meta testing.

forward_support(*x: torch.Tensor*, *gt_label: torch.Tensor*, ***kwargs*) → None
Forward support data in meta testing.

forward_train(*support_feats: torch.Tensor*, *support_labels: torch.Tensor*, *query_feats: torch.Tensor*, *query_labels: torch.Tensor*, ***kwargs*) → Dict
Forward training data.

Parameters

- **support_feats** (*Tensor*) – Features of support data with shape (N, C).
- **support_labels** (*Tensor*) – Labels of support data with shape (N).
- **query_feats** (*Tensor*) – Features of query data with shape (N, C).
- **query_labels** (*Tensor*) – Labels of query data with shape (N).

Returns A dictionary of loss components.

Return type dict[str, Tensor]

class mmfewshot.classification.models.heads.**NegMarginHead**(*num_classes: int*, *in_channels: int*, *temperature: float = 30.0*, *margin: float = 0.0*, *metric_type: str = 'cosine'*, **args*, ***kwargs*)
Classification head for NegMargin.

Parameters

- **num_classes** (*int*) – Number of categories.
- **in_channels** (*int*) – Number of channels in the input feature map.
- **temperature** (*float*) – Scaling factor of *cls_score*. Default: 30.0.
- **margin** (*float*) – Margin of *cls_score*. Default: 0.0.
- **metric_type** (*str*) – The way to calculate similarity. Options:[‘cosine’, ‘softmax’]. Default: ‘cosine’

before_forward_query() → None
Used in meta testing.

This function will be called before model forward query data during meta testing.

before_forward_support() → None
Used in meta testing.

This function will be called before model forward support data during meta testing.

forward_query(*x: torch.Tensor*, ***kwargs*) → List
Forward query data in meta testing.

forward_support(*x: torch.Tensor*, *gt_label: torch.Tensor*, ***kwargs*) → Dict
Forward support data in meta testing.

forward_train(*x: torch.Tensor*, *gt_label: torch.Tensor*, ***kwargs*) → Dict
Forward training data.

class mmfewshot.classification.models.heads.**PrototypeHead**(**args*, ***kwargs*)
Classification head for `ProtoNet.

<https://arxiv.org/abs/1703.05175>`_.

before_forward_query() → None
Used in meta testing.

This function will be called before model forward query data during meta testing.

before_forward_support() → None
Used in meta testing.

This function will be called before model forward support data during meta testing.

forward_query(*x: torch.Tensor*, ***kwargs*) → List
Forward query data in meta testing.

forward_support(*x: torch.Tensor*, *gt_label: torch.Tensor*, ***kwargs*) → None
Forward support data in meta testing.

forward_train(*support_feats: torch.Tensor*, *support_labels: torch.Tensor*, *query_feats: torch.Tensor*, *query_labels: torch.Tensor*, ***kwargs*) → Dict
Forward training data.

Parameters

- **support_feats** (*Tensor*) – Features of support data with shape (N, C).
- **support_labels** (*Tensor*) – Labels of support data with shape (N).
- **query_feats** (*Tensor*) – Features of query data with shape (N, C).
- **query_labels** (*Tensor*) – Labels of query data with shape (N).

Returns A dictionary of loss components.

Return type dict[str, Tensor]

class mmfewshot.classification.models.heads.**RelationHead**(*in_channels: int*, *feature_size: Tuple[int] = (7, 7)*, *hidden_channels: int = 8*, *loss: Dict = {'loss_weight': 1.0, 'type': 'CrossEntropyLoss'}*, **args*, ***kwargs*)

Classification head for `RelationNet.

<https://arxiv.org/abs/1711.06025>`_.

Parameters

- **in_channels** (*int*) – Number of channels in the input feature map.
- **feature_size** (*tuple(int, int)*) – Size of the input feature map. Default: (7, 7).
- **hidden_channels** (*int*) – Number of channels for the hidden fc layer. Default: 8.
- **loss** (*dict*) – Training loss. Options are CrossEntropyLoss and MSELoss.

before_forward_query() → None
Used in meta testing.

This function will be called before model forward query data during meta testing.

before_forward_support() → None
Used in meta testing.

This function will be called before model forward support data during meta testing.

forward_query(*x: torch.Tensor*, ***kwargs*) → List
Forward query data in meta testing.

forward_relation_module(*x: torch.Tensor*) → torch.Tensor
Forward function for relation module.

forward_support(*x: torch.Tensor*, *gt_label: torch.Tensor*, ***kwargs*) → None
Forward support data in meta testing.

forward_train(*support_feats: torch.Tensor*, *support_labels: torch.Tensor*, *query_feats: torch.Tensor*, *query_labels: torch.Tensor*, ***kwargs*) → Dict
Forward training data.

Parameters

- **support_feats** (*Tensor*) – Features of support data with shape (N, C, H, W).
- **support_labels** (*Tensor*) – Labels of support data with shape (N).
- **query_feats** (*Tensor*) – Features of query data with shape (N, C, H, W).
- **query_labels** (*Tensor*) – Labels of query data with shape (N).

Returns A dictionary of loss components.

Return type dict[str, Tensor]

init_weights() → None
Initialize the weights.

23.4.4 losses

class mmfewshot.classification.models.losses.**MSELoss**(*reduction: typing_extensions.Literal[none, mean, sum] = 'mean'*, *loss_weight: float = 1.0*)
MSELoss.

Parameters

- **reduction** (*str*) – The method that reduces the loss to a scalar. Options are "none", "mean" and "sum". Default: 'mean'.
- **loss_weight** (*float*) – The weight of the loss. Default: 1.0.

forward(*pred: torch.Tensor*, *target: torch.Tensor*, *weight: Optional[torch.Tensor] = None*, *avg_factor: Optional[Union[float, int]] = None*, *reduction_override: Optional[str] = None*) → torch.Tensor
Forward function of loss.

Parameters

- **pred** (*Tensor*) – The prediction with shape (N, *), where * means any number of additional dimensions.
- **target** (*Tensor*) – The learning target of the prediction with shape (N, *) same as the input.
- **weight** (*Tensor | None*) – Weight of the loss for each prediction. Default: None.
- **avg_factor** (*float | int | None*) – Average factor that is used to average the loss. Default: None.
- **reduction_override** (*str | None*) – The reduction method used to override the original reduction method of the loss. Options are "none", "mean" and "sum". Default: None.

Returns The calculated loss

Return type Tensor

class mmfewshot.classification.models.losses.**NLLLoss**(*reduction: typing_extensions.Literal[none, mean, sum] = 'mean'*, *loss_weight: float = 1.0*)
NLLLoss.

Parameters

- **reduction** (*str*) – The method that reduces the loss to a scalar. Options are "none", "mean" and "sum". Default: 'mean'.
- **loss_weight** (*float*) – The weight of the loss. Default: 1.0.

forward(*pred: torch.Tensor*, *target: torch.Tensor*, *weight: Optional[torch.Tensor] = None*, *avg_factor: Optional[Union[float, int]] = None*, *reduction_override: Optional[str] = None*) → torch.Tensor

Forward function of loss.

Parameters

- **pred** (*Tensor*) – The prediction with shape (N, C).
- **target** (*Tensor*) – The learning target of the prediction. with shape (N, 1).
- **weight** (*Tensor | None*) – Weight of the loss for each prediction. Default: None.
- **avg_factor** (*float | int | None*) – Average factor that is used to average the loss. Default: None.
- **reduction_override** (*str | None*) – The reduction method used to override the original reduction method of the loss. Options are "none", "mean" and "sum". Default: None.

Returns The calculated loss

Return type Tensor

23.4.5 utils

mmfewshot.classification.models.utils.**convert_maml_module**(*module: torch.nn.modules.module.Module*) → torch.nn.modules.module.Module

Convert a normal model to MAML model.

Replace nn.Linear with LinearWithFastWeight, nn.Conv2d with Conv2dWithFastWeight and BatchNorm2d with BatchNorm2dWithFastWeight.

Parameters **module** (*nn.Module*) – The module to be converted.

Returns : nn.Module: A MAML module.

23.5 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

TWENTYFOUR

MMFEWSHOT.DETECTION

24.1 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

24.2 detection.core

24.2.1 evaluation

class mmfewshot.detection.core.evaluation.**QuerySupportDistEvalHook**(*model_init_dataloader: torch.utils.data.dataloader.DataLoader*, *val_dataloader: torch.utils.data.dataloader.DataLoader*, ***eval_kwargs*)

Distributed evaluation hook for query support data pipeline.

This hook will first traverse *model_init_dataloader* to extract support features for model initialization and then evaluate the data from *val_dataloader*.

Noted that *model_init_dataloader* should NOT use distributed sampler to make all the models on different gpus get same data results in same initialized models.

Parameters

- **model_init_dataloader** (*DataLoader*) – A PyTorch dataloader of *model_init* dataset.
- **val_dataloader** (*DataLoader*) – A PyTorch dataloader of dataset to be evaluated.
- ****eval_kwargs** – Evaluation arguments fed into the evaluate function of the dataset.

class mmfewshot.detection.core.evaluation.**QuerySupportEvalHook**(*model_init_dataloader: torch.utils.data.dataloader.DataLoader*, *val_dataloader: torch.utils.data.dataloader.DataLoader*, ***eval_kwargs*)

Evaluation hook for query support data pipeline.

This hook will first traverse *model_init_dataloader* to extract support features for model initialization and then evaluate the data from *val_dataloader*.

Parameters

- **model_init_dataloader** (*DataLoader*) – A PyTorch dataloader of *model_init* dataset.
- **val_dataloader** (*DataLoader*) – A PyTorch dataloader of dataset to be evaluated.
- ****eval_kwargs** – Evaluation arguments fed into the evaluate function of the dataset.

mmfewshot.detection.core.evaluation.**eval_map**(*det_results: List[List[numpy.ndarray]]*, *annotations: List[Dict]*, *classes: List[str]*, *scale_ranges: Optional[List[Tuple]] = None*, *iou_thr: float = 0.5*, *dataset: Optional[Union[str, List[str]]] = None*, *logger: Optional[object] = None*, *tpfp_fn: Optional[callable] = None*, *nproc: int = 4*, *use_legacy_coordinate: bool = False*) → Tuple[List, List[Dict]]

Evaluate mAP of a dataset.

eval_map() in mmdet predefines the names of classes and thus not supports report map results of arbitrary class splits.

Parameters

- **det_results** (*list[list[np.ndarray]] | list[tuple[np.ndarray]]*) – The outer list indicates images, and the inner list indicates per-class detected bboxes.
- **annotations** (*list[dict]*) – Ground truth annotations where each item of the list indicates an image. Keys of annotations are:
  - *bboxes*: numpy array of shape (n, 4)
  - *labels*: numpy array of shape (n,)
  - *bboxes_ignore* (optional): numpy array of shape (k, 4)
  - *labels_ignore* (optional): numpy array of shape (k,)
- **classes** (*list[str]*) – Names of class.
- **scale_ranges** (*list[tuple] | None*) – Range of scales to be evaluated, in the format [(min1, max1), (min2, max2), ...]. A range of (32, 64) means the area range between (32**2, 64**2). Default: None.
- **iou_thr** (*float*) – IoU threshold to be considered as matched. Default: 0.5.
- **dataset** (*list[str] | str | None*) – Dataset name or dataset classes, there are minor differences in metrics for different datasets, e.g. "voc07", "imagenet_det", etc. Default: None.

- **logger** (*logging.Logger | None*) – The way to print the mAP summary. See *mmcv.utils.print_log()* for details. Default: None.
- **tpfp_fn** (*callable | None*) – The function used to determine true false positives. If None, `tpfp_default()` is used as default unless dataset is 'det' or 'vid' (`tpfp_imagenet()` in this case). If it is given as a function, then this function is used to evaluate tp & fp. Default None.
- **nproc** (*int*) – Processes used for computing TP and FP. Default: 4.
- **use_legacy_coordinate** (*bool*) – Whether to use coordinate system in mmdet v1.x. which means width, height should be calculated as 'x2 - x1 + 1` and 'y2 - y1 + 1' respectively. Default: False.

Returns (list, [dict, dict, ...])

Return type tuple

24.2.2 utils

class mmfewshot.detection.core.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.

24.3 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.

24.4 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.

24.4.1 backbones

class mmfewshot.detection.models.backbones.**ResNetWithMetaConv**(***kwargs*)

ResNet with *meta_conv* to handle different inputs in metarcnn and fsdetview.

When input with shape (N, 3, H, W) from images, the network will use *conv1* as regular ResNet. When input with shape (N, 4, H, W) from (image + mask) the network will replace *conv1* with *meta_conv* to handle additional channel.

forward(*x: torch.Tensor*, *use_meta_conv: bool = False*) → Tuple[torch.Tensor]

Forward function.

When input with shape (N, 3, H, W) from images, the network will use *conv1* as regular ResNet. When input with shape (N, 4, H, W) from (image + mask) the network will replace *conv1* with *meta_conv* to handle additional channel.

Parameters

- **x** (*Tensor*) – Tensor with shape (N, 3, H, W) from images or (N, 4, H, W) from (images + masks).
- **use_meta_conv** (*bool*) – If set True, forward input tensor with *meta_conv* which require tensor with shape (N, 4, H, W). Otherwise, forward input tensor with *conv1* which require tensor with shape (N, 3, H, W). Default: False.

Returns

Tuple of features, each item with shape (N, C, H, W).

Return type tuple[Tensor]

24.4.2 dense_heads

class mmfewshot.detection.models.dense_heads.**AttentionRPNHead**(*num_support_ways: int*, *num_support_shots: int*, *aggregation_layer: Dict = {'aggregator_cfgs': [{'type': 'DepthWiseCorrelationAggregator', 'in_channels': 1024, 'with_fc': False}], 'type': 'AggregationLayer'}*, *roi_extractor: Dict = {'featmap_strides': [16], 'out_channels': 1024, 'roi_layer': {'output_size': 14, 'sampling_ratio': 0, 'type': 'RoIAlign'}, 'type': 'SingleRoIExtractor'}*, ***kwargs*)

RPN head for Attention RPN.

Parameters

- **num_support_ways** (*int*) – Number of sampled classes (pos + neg).
- **num_support_shots** (*int*) – Number of shot for each classes.
- **aggregation_layer** (*dict*) – Config of *aggregation_layer*.
- **roi_extractor** (*dict*) – Config of *roi_extractor*.

extract_roi_feat(*feats: List[torch.Tensor]*, *rois: torch.Tensor*) → torch.Tensor

Forward function.

Parameters

- **feats** (*list[Tensor]*) – Input features with shape (N, C, H, W).
- **rois** – with shape (m, 5).

forward_train(*query_feats: List[torch.Tensor]*, *support_feats: List[torch.Tensor]*, *query_gt_bboxes: List[torch.Tensor]*, *query_img_metas: List[Dict]*, *support_gt_bboxes: List[torch.Tensor]*, *query_gt_bboxes_ignore: Optional[List[torch.Tensor]] = None*, *proposal_cfg: Optional[mmcv.utils.config.ConfigDict] = None*, ***kwargs*) → Tuple[Dict, List[Tuple]]

Forward function in training phase.

Parameters

- **query_feats** (*list[Tensor]*) – List of query features, each item with shape (N, C, H, W)..
- **support_feats** (*list[Tensor]*) – List of support features, each item with shape (N, C, H, W).
- **query_gt_bboxes** (*list[Tensor]*) – List of ground truth bboxes of query image, each item with shape (num_gts, 4).
- **query_img_metas** (*list[dict]*) – List of query image info dict where each dict has: *img_shape*, *scale_factor*, *flip*, and may also contain *filename*, *ori_shape*, *pad_shape*, and *img_norm_cfg*. For details on the values of these keys see `mmdet.datasets.pipelines.Collect`.
- **support_gt_bboxes** (*list[Tensor]*) – List of ground truth bboxes of support image, each item with shape (num_gts, 4).
- **query_gt_bboxes_ignore** (*list[Tensor]*) – List of ground truth bboxes to be ignored of query image with shape (num_ignored_gts, 4). Default: None.
- **proposal_cfg** (`ConfigDict`) – Test / postprocessing configuration. if None, test_cfg would be used. Default: None.

Returns

loss components and proposals of each image.

- losses: (dict[str, Tensor]): A dictionary of loss components.
- proposal_list (list[Tensor]): Proposals of each image.

Return type tuple

loss(*cls_scores: List[torch.Tensor]*, *bbox_preds: List[torch.Tensor]*, *gt_bboxes: List[torch.Tensor]*, *img_metas: List[Dict]*, *gt_labels: Optional[List[torch.Tensor]] = None*, *gt_bboxes_ignore: Optional[List[torch.Tensor]] = None*, *pair_flags: Optional[List[bool]] = None*) → Dict

Compute losses of rpn head.

Parameters

- **cls_scores** (*list[Tensor]*) – Box scores for each scale level with shape (N, num_anchors * num_classes, H, W)
- **bbox_preds** (*list[Tensor]*) – Box energies / deltas for each scale level with shape (N, num_anchors * 4, H, W)
- **gt_bboxes** (*list[Tensor]*) – Ground truth bboxes for each image with shape (num_gts, 4) in [tl_x, tl_y, br_x, br_y] format.
- **img_metas** (*list[dict]*) – list of image info dict where each dict has: *img_shape*, *scale_factor*, *flip*, and may also contain *filename*, *ori_shape*, *pad_shape*, and

img_norm_cfg. For details on the values of these keys see `mmdet.datasets.pipelines.Collect`.

- **gt_labels** (*list[Tensor]*) – Class indices corresponding to each box. Default: None.
- **gt_bboxes_ignore** (*None | list[Tensor]*) – Specify which bounding boxes can be ignored when computing the loss. Default: None
- **pair_flags** (*list[bool]*) – Indicate predicted result is from positive pair or negative pair with shape (N). Default: None.

Returns A dictionary of loss components.

Return type dict[str, Tensor]

simple_test(*query_feats: List[torch.Tensor]*, *support_feat: torch.Tensor*, *query_img_metas: List[Dict]*, *rescale: bool = False*) → List[torch.Tensor]

Test function without test time augmentation.

Parameters

- **query_feats** (*list[Tensor]*) – List of query features, each item with shape(N, C, H, W).
- **support_feat** (*Tensor*) – Support features with shape (N, C, H, W).
- **query_img_metas** (*list[dict]*) – List of query image info dict where each dict has: *img_shape*, *scale_factor*, *flip*, and may also contain *filename*, *ori_shape*, *pad_shape*, and *img_norm_cfg*. For details on the values of these keys see `mmdet.datasets.pipelines.Collect`.
- **rescale** (*bool*) – Whether to rescale the results. Default: False.

Returns

Proposals of each image, each item has shape (n, 5), where 5 represent (tl_x, tl_y, br_x, br_y, score).

Return type List[Tensor]

class mmfewshot.detection.models.dense_heads.**TwoBranchRPNHead**(*mid_channels: int = 64*, ***kwargs*)

RPN head for MPSR.

Parameters **mid_channels** (*int*) – Input channels of *rpn_cls_conv*. Default: 64.

forward_auxiliary(*feats: List[torch.Tensor]*) → List[torch.Tensor]

Forward auxiliary features at multiple scales.

Parameters **feats** (*list[Tensor]*) – List of features at multiple scales, each is a 4D-tensor.

Returns

Classification scores for all scale levels, each is a 4D-tensor, the channels number is num_anchors * num_classes.

Return type list[Tensor]

forward_auxiliary_single(*feat: torch.Tensor*) → Tuple[torch.Tensor]

Forward auxiliary feature map of a single scale level.

forward_single(*feat: torch.Tensor*) → Tuple[torch.Tensor, torch.Tensor]

Forward feature map of a single scale level.

forward_train(*x: List[torch.Tensor]*, *auxiliary_rpn_feats: List[torch.Tensor]*, *img_metas: List[Dict]*, *gt_bboxes: List[torch.Tensor]*, *gt_labels: Optional[List[torch.Tensor]] = None*, *gt_bboxes_ignore: Optional[List[torch.Tensor]] = None*, *proposal_cfg: Optional[mmcv.utils.config.ConfigDict] = None*, ***kwargs*) → Tuple[Dict, List[torch.Tensor]]

Parameters

- **x** (*list[Tensor]*) – Features from FPN, each item with shape (N, C, H, W).
- **auxiliary_rpn_feats** (*list[Tensor]*) – Auxiliary features from FPN, each item with shape (N, C, H, W).
- **img_metas** (*list[dict]*) – Meta information of each image, e.g., image size, scaling factor, etc.
- **gt_bboxes** (*list[Tensor]*) – Ground truth bboxes of the image, shape (num_gts, 4).
- **gt_labels** (*list[Tensor]*) – Ground truth labels of each box, shape (num_gts,). Default: None.
- **gt_bboxes_ignore** (*list[Tensor]*) – Ground truth bboxes to be ignored, shape (num_ignored_gts, 4). Default: None.
- **proposal_cfg** (*ConfigDict*) – Test / postprocessing configuration, if None, test_cfg would be used. Default: None.

Returns losses: (dict[str, Tensor]): A dictionary of loss components. proposal_list (List[Tensor]): Proposals of each image.

Return type tuple

get_bboxes(*cls_scores: List[torch.Tensor]*, *bbox_preds: List[torch.Tensor]*, *img_metas: List[Dict]*, *cfg: Optional[mmcv.utils.config.ConfigDict] = None*, *rescale: bool = False*, *with_nms: bool = True*) → List[torch.Tensor]

Transform network output for a batch into bbox predictions.

Parameters

- **cls_scores** (*list[Tensor]*) – Box scores for each scale level Has shape (N, num_anchors * num_classes, H, W)
- **bbox_preds** (*list[Tensor]*) – Box energies / deltas for each scale level with shape (N, num_anchors * 4, H, W)
- **img_metas** (*list[dict]*) – Meta information of each image, e.g., image size, scaling factor, etc.
- **cfg** (*ConfigDict | None*) – Test / postprocessing configuration, if None, test_cfg would be used
- **rescale** (*bool*) – If True, return boxes in original image space. Default: False.
- **with_nms** (*bool*) – If True, do nms before return boxes. Default: True.

Returns

Proposals of each image, each item has shape (n, 5), where 5 represent (tl_x, tl_y, br_x, br_y, score).

Return type List[Tensor]

loss(*cls_scores: List[torch.Tensor]*, *bbox_preds: List[torch.Tensor]*, *gt_bboxes: List[torch.Tensor]*, *gt_labels: List[torch.Tensor]*, *img_metas: List[Dict]*, *gt_bboxes_ignore: Optional[List[torch.Tensor]] = None*, *auxiliary_cls_scores: Optional[List[torch.Tensor]] = None*) → Dict

Compute losses of the head.

Parameters

- **cls_scores** (*list[Tensor]*) – Box scores for each scale level, each item with shape (N, num_anchors * num_classes, H, W).
- **bbox_preds** (*list[Tensor]*) – Box energies / deltas for each scale level with shape (N, num_anchors * 4, H, W)
- **gt_bboxes** (*list[Tensor]*) – Ground truth bboxes for each image with shape (num_gts, 4) in [tl_x, tl_y, br_x, br_y] format.
- **gt_labels** (*list[Tensor]*) – class indices corresponding to each box
- **img_metas** (*list[dict]*) – Meta information of each image, e.g., image size, scaling factor, etc.
- **gt_bboxes_ignore** (*list[Tensor] | None*) – specify which bounding boxes can be ignored when computing the loss. Default: None.
- **auxiliary_cls_scores** (*list[Tensor] | None*) – Box scores for each scale level, each item with shape (N, num_anchors * num_classes, H, W). Default: None.

Returns A dictionary of loss components.

Return type dict[str, Tensor]

loss_bbox_single(*bbox_pred: torch.Tensor*, *anchors: torch.Tensor*, *bbox_targets: torch.Tensor*, *bbox_weights: torch.Tensor*, *num_total_samples: int*) → Tuple[Dict]

Compute loss of a single scale level.

Parameters

- **bbox_pred** (*Tensor*) – Box energies / deltas for each scale level with shape (N, num_anchors * 4, H, W).
- **anchors** (*Tensor*) – Box reference for each scale level with shape (N, num_total_anchors, 4).
- **bbox_targets** (*Tensor*) – BBox regression targets of each anchor weight shape (N, num_total_anchors, 4).
- **bbox_weights** (*Tensor*) – BBox regression loss weights of each anchor with shape (N, num_total_anchors, 4).
- **num_total_samples** (*int*) – If sampling, num total samples equal to the number of total anchors; Otherwise, it is the number of positive anchors.

Returns A dictionary of loss components.

Return type tuple[dict[str, Tensor]]

loss_cls_single(*cls_score: torch.Tensor*, *labels: torch.Tensor*, *label_weights: torch.Tensor*, *num_total_samples: int*) → Tuple[Dict]

Compute loss of a single scale level.

Parameters

- **cls_score** (*Tensor*) – Box scores for each scale level Has shape (N, num_anchors * num_classes, H, W).

- **labels** (*Tensor*) – Labels of each anchors with shape (N, num_total_anchors).
- **label_weights** (*Tensor*) – Label weights of each anchor with shape (N, num_total_anchors)
- **num_total_samples** (*int*) – If sampling, num total samples equal to the number of total anchors; Otherwise, it is the number of positive anchors.

Returns A dictionary of loss components.

Return type tuple[dict[str, Tensor]]

24.4.3 detectors

class mmfewshot.detection.models.detectors.**AttentionRPNDetector**(*backbone: mmcv.utils.config.ConfigDict*, *neck: Optional[mmcv.utils.config.ConfigDict] = None*, *support_backbone: Optional[mmcv.utils.config.ConfigDict] = None*, *support_neck: Optional[mmcv.utils.config.ConfigDict] = None*, *rpn_head: Optional[mmcv.utils.config.ConfigDict] = None*, *roi_head: Optional[mmcv.utils.config.ConfigDict] = None*, *train_cfg: Optional[mmcv.utils.config.ConfigDict] = None*, *test_cfg: Optional[mmcv.utils.config.ConfigDict] = None*, *pretrained: Optional[mmcv.utils.config.ConfigDict] = None*, *init_cfg: Optional[mmcv.utils.config.ConfigDict] = None*)

Implementation of AttentionRPN.

Parameters

- **backbone** (*dict*) – Config of the backbone for query data.
- **neck** (*dict | None*) – Config of the neck for query data and probably for support data. Default: None.
- **support_backbone** (*dict | None*) – Config of the backbone for support data only. If None, support and query data will share same backbone. Default: None.
- **support_neck** (*dict | None*) – Config of the neck for support data only. Default: None.
- **rpn_head** (*dict | None*) – Config of rpn_head. Default: None.
- **roi_head** (*dict | None*) – Config of roi_head. Default: None.
- **train_cfg** (*dict | None*) – Training config. Useless in CenterNet, but we keep this variable for SingleStageDetector. Default: None.
- **test_cfg** (*dict | None*) – Testing config of CenterNet. Default: None.
- **pretrained** (*str | None*) – model pretrained path. Default: None.
- **init_cfg** (*dict | list[dict] | None*) – Initialization config dict. Default: None.

extract_support_feat(*img: torch.Tensor*) → List[torch.Tensor]
Extract features of support data.

Parameters **img** (*Tensor*) – Input images of shape (N, C, H, W). Typically these should be mean centered and std scaled.

Returns

Features of support images, each item with shape (N, C, H, W).

Return type list[Tensor]

forward_model_init(*img: torch.Tensor*, *img_metas: List[Dict]*, *gt_bboxes: Optional[List[torch.Tensor]] = None*, *gt_labels: Optional[List[torch.Tensor]] = None*, ***kwargs*) → Dict
Extract and save support features for model initialization.

Parameters

- **img** (*Tensor*) – Input images of shape (N, C, H, W). Typically these should be mean centered and std scaled.
- **img_metas** (*list[dict]*) – list of image info dict where each dict has: *img_shape*, *scale_factor*, *flip*, and may also contain *filename*, *ori_shape*, *pad_shape*, and *img_norm_cfg*. For details on the values of these keys see `mmdet.datasets.pipelines.Collect`.
- **gt_bboxes** (*list[Tensor]*) – Ground truth bboxes for each image with shape (num_gts, 4) in [tl_x, tl_y, br_x, br_y] format.
- **gt_labels** (*list[Tensor]*) – class indices corresponding to each box.

Returns

A dict contains following keys:

- ***gt_labels* (Tensor): class indices corresponding to each** feature.
- *res4_roi_feat* (Tensor): roi features of res4 layer.
- *res5_roi_feat* (Tensor): roi features of res5 layer.

Return type dict

model_init() → None
process the saved support features for model initialization.

simple_test(*img: torch.Tensor*, *img_metas: List[Dict]*, *proposals: Optional[List[torch.Tensor]] = None*, *rescale: bool = False*) → List[List[numpy.ndarray]]
Test without augmentation.

Parameters

- **img** (*Tensor*) – Input images of shape (N, C, H, W). Typically these should be mean centered and std scaled.
- **img_metas** (*list[dict]*) – list of image info dict where each dict has: *img_shape*, *scale_factor*, *flip*, and may also contain *filename*, *ori_shape*, *pad_shape*, and *img_norm_cfg*. For details on the values of these keys see `mmdet.datasets.pipelines.Collect`.
- **proposals** (*list[Tensor] | None*) – override rpn proposals with custom proposals. Use when *with_rpn* is False. Default: None.
- **rescale** (*bool*) – If True, return boxes in original image space.

Returns

BBox results of each image and classes. The outer list corresponds to each image. The inner list corresponds to each class.

Return type list[list[np.ndarray]]

class mmfewshot.detection.models.detectors.**FSCE**(*backbone*, *neck=None*, *rpn_head=None*, *roi_head=None*, *train_cfg=None*, *test_cfg=None*, *pretrained=None*, *init_cfg=None*)

Implementation of FSCE

class mmfewshot.detection.models.detectors.**FSDetView**(*backbone: mmcv.utils.config.ConfigDict*, *neck: Optional[mmcv.utils.config.ConfigDict] = None*, *support_backbone: Optional[mmcv.utils.config.ConfigDict] = None*, *support_neck: Optional[mmcv.utils.config.ConfigDict] = None*, *rpn_head: Optional[mmcv.utils.config.ConfigDict] = None*, *roi_head: Optional[mmcv.utils.config.ConfigDict] = None*, *train_cfg: Optional[mmcv.utils.config.ConfigDict] = None*, *test_cfg: Optional[mmcv.utils.config.ConfigDict] = None*, *pretrained: Optional[mmcv.utils.config.ConfigDict] = None*, *init_cfg: Optional[mmcv.utils.config.ConfigDict] = None*)

Implementation of FSDetView.

class mmfewshot.detection.models.detectors.**MPSR**(*rpn_select_levels: List[int]*, *roi_select_levels: List[int]*, **args*, ***kwargs*)

Implementation of MPSR..

Parameters

- **rpn_select_levels** (*list[int]*) – Specify the corresponding level of fpn features for each scale of image. The selected features will be fed into rpn head.
- **roi_select_levels** (*list[int]*) – Specific which level of fpn features to be selected for each scale of image. The selected features will be fed into roi head.

extract_auxiliary_feat(*auxiliary_img_list: List[torch.Tensor]*) → Tuple[List[torch.Tensor], List[torch.Tensor]]

Extract and select features from data list at multiple scale.

Parameters **auxiliary_img_list** (*list[Tensor]*) – List of data at different scales. In most cases, each dict contains: *img*, *img_metas*, *gt_bboxes*, *gt_labels*, *gt_bboxes_ignore*.

Returns

rpn_feats (list[Tensor]): Features at multiple scale used for rpn head training.

roi_feats (list[Tensor]): Features at multiple scale used for roi head training.

Return type tuple

extract_feat(*img: torch.Tensor*) → List[torch.Tensor]

Directly extract features from the backbone+neck.

forward(*main_data: Dict = None*, *auxiliary_data: Dict = None*, *img: List[torch.Tensor] = None*, *img_metas: List[Dict] = None*, *return_loss: bool = True*, ***kwargs*) → Dict

Calls either *forward_train()* or `forward_test()` depending on whether `return_loss` is `True`.

Note this setting will change the expected inputs. When `return_loss=True`, the input will be main and auxiliary data for training., and when `resturn_loss=False`, the input will be img and img_meta for testing.

Parameters

- **main_data** (*dict*) – Used for *forward_train()*. Dict of data and data info, where each dict has: *img*, *img_metas*, *gt_bboxes*, *gt_labels*, *gt_bboxes_ignore*. Default: None.
- **auxiliary_data** (*dict*) – Used for *forward_train()*. Dict of data and data info at multiple scales, where each key use different suffix to indicate different scale. For example, *img_scale_i*, *img_metas_scale_i*, *gt_bboxes_scale_i*, *gt_labels_scale_i*, *gt_bboxes_ignore_scale_i*, where *i* in range of 0 to number of scales. Default: None.
- **img** (*list[Tensor]*) – Used for func:*forward_test* or `forward_model_init()`. List of tensors of shape (1, C, H, W). Typically these should be mean centered and std scaled. Default: None.
- **img_metas** (*list[dict]*) – Used for func:*forward_test* or `forward_model_init()`. List of image info dict where each dict has: *img_shape*, *scale_factor*, *flip*, and may also contain *filename*, *ori_shape*, *pad_shape*, and *img_norm_cfg*. For details on the values of these keys, see `mmdet.datasets.pipelines.Collect`. Default: None.
- **return_loss** (*bool*) – If set True call *forward_train()*, otherwise call `forward_test()`. Default: True.

forward_train(*main_data: Dict*, *auxiliary_data_list: List[Dict]*, ***kwargs*) → Dict

Parameters

- **main_data** (*dict*) – In most cases, dict of main data contains: *img*, *img_metas*, *gt_bboxes*, *gt_labels*, *gt_bboxes_ignore*.
- **auxiliary_data_list** (*list[dict]*) – List of data at different scales. In most cases, each dict contains: *img*, *img_metas*, *gt_bboxes*, *gt_labels*, *gt_bboxes_ignore*.

Returns a dictionary of loss components

Return type dict[str, Tensor]

train_step(*data: Dict*, *optimizer: Union[object, Dict]*) → Dict

The iteration step during training.

This method defines an iteration step during training, except for the back propagation and optimizer updating, which are done in an optimizer hook. Note that in some complicated cases or models, the whole process including back propagation and optimizer updating is also defined in this method, such as GAN.

Parameters

- **data** (*dict*) – The output of dataloader.
- **optimizer** (`torch.optim.Optimizer` | dict) – The optimizer of runner is passed to `train_step()`. This argument is unused and reserved.

Returns

It should contain at least 3 keys: `loss`, `log_vars`, `num_samples`.

- `loss` is a tensor for back propagation, which can be a weighted sum of multiple losses.

- `log_vars` contains all the variables to be sent to the

logger. - `num_samples` indicates the batch size (when the model is DDP, it means the batch size on each GPU), which is used for averaging the logs.

Return type dict

val_step(*data: Dict*, *optimizer: Optional[Union[object, Dict]] = None*) → Dict
The iteration step during validation.

This method shares the same signature as *train_step()*, but used during val epochs. Note that the evaluation after training epochs is not implemented with this method, but an evaluation hook.

class mmfewshot.detection.models.detectors.**MetaRCNN**(*backbone: mmcv.utils.config.ConfigDict*, *neck: Optional[mmcv.utils.config.ConfigDict] = None*, *support_backbone: Optional[mmcv.utils.config.ConfigDict] = None*, *support_neck: Optional[mmcv.utils.config.ConfigDict] = None*, *rpn_head: Optional[mmcv.utils.config.ConfigDict] = None*, *roi_head: Optional[mmcv.utils.config.ConfigDict] = None*, *train_cfg: Optional[mmcv.utils.config.ConfigDict] = None*, *test_cfg: Optional[mmcv.utils.config.ConfigDict] = None*, *pretrained: Optional[mmcv.utils.config.ConfigDict] = None*, *init_cfg: Optional[mmcv.utils.config.ConfigDict] = None*)

Implementation of Meta R-CNN..

Parameters

- **backbone** (*dict*) – Config of the backbone for query data.
- **neck** (*dict | None*) – Config of the neck for query data and probably for support data. Default: None.
- **support_backbone** (*dict | None*) – Config of the backbone for support data only. If None, support and query data will share same backbone. Default: None.
- **support_neck** (*dict | None*) – Config of the neck for support data only. Default: None.
- **rpn_head** (*dict | None*) – Config of rpn_head. Default: None.
- **roi_head** (*dict | None*) – Config of roi_head. Default: None.
- **train_cfg** (*dict | None*) – Training config. Useless in CenterNet, but we keep this variable for SingleStageDetector. Default: None.
- **test_cfg** (*dict | None*) – Testing config of CenterNet. Default: None.
- **pretrained** (*str | None*) – model pretrained path. Default: None.
- **init_cfg** (*dict | list[dict] | None*) – Initialization config dict. Default: None

extract_support_feat(*img*)
Extracting features from support data.

Parameters **img** (*Tensor*) – Input images of shape (N, C, H, W). Typically these should be mean centered and std scaled.

Returns

Features of input image, each item with shape (N, C, H, W).

Return type list[Tensor]

`forward_model_init`(*img: torch.Tensor*, *img_metas: List[Dict]*, *gt_bboxes: Optional[List[torch.Tensor]] = None*, *gt_labels: Optional[List[torch.Tensor]] = None*, ***kwargs*)

extract and save support features for model initialization.

Parameters

- **img** (*Tensor*) – Input images of shape (N, C, H, W). Typically these should be mean centered and std scaled.
- **img_metas** (*list[dict]*) – list of image info dict where each dict has: *img_shape*, *scale_factor*, *flip*, and may also contain *filename*, *ori_shape*, *pad_shape*, and *img_norm_cfg*. For details on the values of these keys see `mmdet.datasets.pipelines.Collect`.
- **gt_bboxes** (*list[Tensor]*) – Ground truth bboxes for each image with shape (num_gts, 4) in [tl_x, tl_y, br_x, br_y] format.
- **gt_labels** (*list[Tensor]*) – class indices corresponding to each box.

Returns

A dict contains following keys:

- ***gt_labels* (Tensor): class indices corresponding to each** feature.
- *res5_rois* (list[Tensor]): roi features of res5 layer.

Return type dict

`model_init`()

process the saved support features for model initialization.

`simple_test`(*img: torch.Tensor*, *img_metas: List[Dict]*, *proposals: Optional[List[torch.Tensor]] = None*, *rescale: bool = False*)

Test without augmentation.

Parameters

- **img** (*Tensor*) – Input images of shape (N, C, H, W). Typically these should be mean centered and std scaled.
- **img_metas** (*list[dict]*) – list of image info dict where each dict has: *img_shape*, *scale_factor*, *flip*, and may also contain *filename*, *ori_shape*, *pad_shape*, and *img_norm_cfg*. For details on the values of these keys see `mmdet.datasets.pipelines.Collect`.
- **proposals** (*list[Tensor] | None*) – override rpn proposals with custom proposals. Use when *with_rpn* is False. Default: None.
- **rescale** (*bool*) – If True, return boxes in original image space.

Returns

BBox results of each image and classes. The outer list corresponds to each image. The inner list corresponds to each class.

Return type list[list[np.ndarray]]

class mmfewshot.detection.models.detectors.**QuerySupportDetector**(*backbone: mmcv.utils.config.ConfigDict*, *neck: Optional[mmcv.utils.config.ConfigDict] = None*, *support_backbone: Optional[mmcv.utils.config.ConfigDict] = None*, *support_neck: Optional[mmcv.utils.config.ConfigDict] = None*, *rpn_head: Optional[mmcv.utils.config.ConfigDict] = None*, *roi_head: Optional[mmcv.utils.config.ConfigDict] = None*, *train_cfg: Optional[mmcv.utils.config.ConfigDict] = None*, *test_cfg: Optional[mmcv.utils.config.ConfigDict] = None*, *pretrained: Optional[mmcv.utils.config.ConfigDict] = None*, *init_cfg: Optional[mmcv.utils.config.ConfigDict] = None*)

Base class for two-stage detectors in query-support fashion.

Query-support detectors typically consisting of a region proposal network and a task-specific regression head. There are two pipelines for query and support data respectively.

Parameters

- **backbone** (*dict*) – Config of the backbone for query data.
- **neck** (*dict | None*) – Config of the neck for query data and probably for support data. Default: None.
- **support_backbone** (*dict | None*) – Config of the backbone for support data only. If None, support and query data will share same backbone. Default: None.
- **support_neck** (*dict | None*) – Config of the neck for support data only. Default: None.
- **rpn_head** (*dict | None*) – Config of rpn_head. Default: None.
- **roi_head** (*dict | None*) – Config of roi_head. Default: None.
- **train_cfg** (*dict | None*) – Training config. Useless in CenterNet, but we keep this variable for SingleStageDetector. Default: None.
- **test_cfg** (*dict | None*) – Testing config of CenterNet. Default: None.
- **pretrained** (*str | None*) – model pretrained path. Default: None.
- **init_cfg** (*dict | list[dict] | None*) – Initialization config dict. Default: None

aug_test(***kwargs*)

Test with augmentation.

extract_feat(*img: torch.Tensor*) → List[torch.Tensor]

Extract features of query data.

Parameters **img** (*Tensor*) – Input images of shape (N, C, H, W). Typically these should be mean centered and std scaled.

Returns Features of query images.

Return type list[Tensor]

extract_query_feat(*img: torch.Tensor*) → List[torch.Tensor]
Extract features of query data.

Parameters **img** (*Tensor*) – Input images of shape (N, C, H, W). Typically these should be mean centered and std scaled.

Returns

Features of support images, each item with shape (N, C, H, W).

Return type list[Tensor]

abstract extract_support_feat(*img: torch.Tensor*)
Extract features of support data.

forward(*query_data: Optional[Dict] = None*, *support_data: Optional[Dict] = None*, *img: Optional[List[torch.Tensor]] = None*, *img_metas: Optional[List[Dict]] = None*, *mode: typing_extensions.Literal[train,* model_init*, test] = 'train'*, ***kwargs*) → Dict
Calls one of (*forward_train()*, forward_test() and *forward_model_init()*) according to the *mode*. The inputs of forward function would change with the *mode*.

- When *mode* is 'train', the input will be query and support data

for training.

- When *mode* is 'model_init', the input will be support template

data at least including (img, img_metas).

- When *mode* is 'test', the input will be test data at least

including (img, img_metas).

Parameters

- **query_data** (*dict*) – Used for *forward_train()*. Dict of query data and data info where each dict has: *img*, *img_metas*, *gt_bboxes*, *gt_labels*, *gt_bboxes_ignore*. Default: None.
- **support_data** (*dict*) – Used for *forward_train()*. Dict of support data and data info dict where each dict has: *img*, *img_metas*, *gt_bboxes*, *gt_labels*, *gt_bboxes_ignore*. Default: None.
- **img** (*list[Tensor]*) – Used for func:*forward_test* or *forward_model_init()*. List of tensors of shape (1, C, H, W). Typically these should be mean centered and std scaled. Default: None.
- **img_metas** (*list[dict]*) – Used for func:*forward_test* or *forward_model_init()*. List of image info dict where each dict has: *img_shape*, *scale_factor*, *flip*, and may also contain *filename*, *ori_shape*, *pad_shape*, and *img_norm_cfg*. For details on the values of these keys, see mmdet.datasets.pipelines.Collect. Default: None.
- **mode** (*str*) – Indicate which function to call. Options are 'train', 'model_init' and 'test'. Default: 'train'.

abstract forward_model_init(*img: torch.Tensor*, *img_metas: List[Dict]*, *gt_bboxes: Optional[List[torch.Tensor]] = None*, *gt_labels: Optional[List[torch.Tensor]] = None*, ***kwargs*)
extract and save support features for model initialization.

forward_train(*query_data: Dict*, *support_data: Dict*, *proposals: Optional[List] = None*, ***kwargs*) → Dict
Forward function for training.

Parameters

- **query_data** (*dict*) – In most cases, dict of query data contains: *img*, *img_metas*, *gt_bboxes*, *gt_labels*, *gt_bboxes_ignore*.
- **support_data** (*dict*) – In most cases, dict of support data contains: *img*, *img_metas*, *gt_bboxes*, *gt_labels*, *gt_bboxes_ignore*.
- **proposals** (*list*) – Override rpn proposals with custom proposals. Use when *with_rpn* is False. Default: None.

Returns a dictionary of loss components

Return type dict[str, Tensor]

abstract model_init(***kwargs*)
process the saved support features for model initialization.

simple_test(*img: torch.Tensor*, *img_metas: List[Dict]*, *proposals: Optional[List[torch.Tensor]] = None*, *rescale: bool = False*)
Test without augmentation.

train_step(*data: Dict*, *optimizer: Union[object, Dict]*) → Dict
The iteration step during training.

This method defines an iteration step during training, except for the back propagation and optimizer updating, which are done in an optimizer hook. Note that in some complicated cases or models, the whole process including back propagation and optimizer updating is also defined in this method, such as GAN. For most of query-support detectors, the batch size denote the batch size of query data.

Parameters

- **data** (*dict*) – The output of dataloader.
- **optimizer** (torch.optim.Optimizer | dict) – The optimizer of runner is passed to `train_step()`. This argument is unused and reserved.

Returns

It should contain at least 3 keys: loss, log_vars, num_samples.

- loss is a tensor for back propagation, which can be a

weighted sum of multiple losses. - log_vars contains all the variables to be sent to the logger. - num_samples indicates the batch size (when the model is DDP, it means the batch size on each GPU), which is used for averaging the logs.

Return type dict

val_step(*data: Dict*, *optimizer: Optional[Union[object, Dict]] = None*) → Dict
The iteration step during validation.

This method shares the same signature as *train_step()*, but used during val epochs. Note that the evaluation after training epochs is not implemented with this method, but an evaluation hook.

class mmfewshot.detection.models.detectors.**TFA**(*backbone*, *neck=None*, *rpn_head=None*, *roi_head=None*, *train_cfg=None*, *test_cfg=None*, *pretrained=None*, *init_cfg=None*)
Implementation of TFA

24.4.4 losses

class mmfewshot.detection.models.losses.**SupervisedContrastiveLoss**(*temperature: float = 0.2*, *iou_threshold: float = 0.5*, *reweight_type: typing_extensions.Literal[none, exp, linear] = 'none'*, *reduction: typing_extensions.Literal[none, mean, sum] = 'mean'*, *loss_weight: float = 1.0*)

Supervised Contrastive LOSS.

This part of code is modified from https://github.com/MegviiDetection/FSCE.

Parameters

- **temperature** (*float*) – A constant to be divided by consine similarity to enlarge the magnitude. Default: 0.2.
- **iou_threshold** (*float*) – Consider proposals with higher credibility to increase consistency. Default: 0.5.
- **reweight_type** (*str*) – Reweight function for contrastive loss. Options are ('none', 'exp', 'linear'). Default: 'none'.
- **reduction** (*str*) – The method used to reduce the loss into a scalar. Default: 'mean'. Options are "none", "mean" and "sum".
- **loss_weight** (*float*) – Weight of loss. Default: 1.0.

forward(*features: torch.Tensor*, *labels: torch.Tensor*, *ious: torch.Tensor*, *decay_rate: Optional[float] = None*, *weight: Optional[torch.Tensor] = None*, *avg_factor: Optional[int] = None*, *reduction_override: Optional[str] = None*) → torch.Tensor

Forward function.

Parameters

- **features** (*tensor*) – Shape of (N, K) where N is the number of features to be compared and K is the channels.
- **labels** (*tensor*) – Shape of (N).
- **ious** (*tensor*) – Shape of (N).
- **decay_rate** (*float | None*) – The decay rate for total loss. Default: None.
- **weight** (*Tensor | None*) – The weight of loss for each prediction with shape of (N). Default: None.
- **avg_factor** (*int | None*) – Average factor that is used to average the loss. Default: None.
- **reduction_override** (*str | None*) – The reduction method used to override the original reduction method of the loss. Options are "none", "mean" and "sum". Default: None.

Returns The calculated loss.

Return type Tensor

24.4.5 roi_heads

class mmfewshot.detection.models.roi_heads.**ContrastiveBBoxHead**(*mlp_head_channels: int = 128*, *with_weight_decay: bool = False*, *loss_contrast: Dict = {'iou_threshold': 0.5, 'loss_weight': 1.0, 'reweight_type': 'none', 'temperature': 0.1, 'type': 'SupervisedContrastiveLoss'}*, *scale: int = 20*, *learnable_scale: bool = False*, *eps: float = 1e-05*, **args*, ***kwargs*)

BBoxHead for FSCE.

Parameters

- **mlp_head_channels** (*int*) – Output channels of contrast branch mlp. Default: 128.
- **with_weight_decay** (*bool*) – Whether to decay loss weight. Default: False.
- **loss_contrast** (*dict*) – Config of contrast loss.
- **scale** (*int*) – Scaling factor of *cls_score*. Default: 20.
- **learnable_scale** (*bool*) – Learnable global scaling factor. Default: False.
- **eps** (*float*) – Constant variable to avoid division by zero.

forward(*x: torch.Tensor*) → Tuple[torch.Tensor, torch.Tensor, torch.Tensor]

Forward function.

Parameters **x** (*Tensor*) – Shape of (num_proposals, C, H, W).

Returns

cls_score (Tensor): Cls scores, has shape (num_proposals, num_classes).

bbox_pred (Tensor): Box energies / deltas, has shape (num_proposals, 4).

contrast_feat (Tensor): Box features for contrast loss, has shape (num_proposals, C).

Return type tuple

loss_contrast(*contrast_feat: torch.Tensor*, *proposal_ious: torch.Tensor*, *labels: torch.Tensor*, *reduction_override: Optional[str] = None*) → Dict

Loss for contract.

Parameters

- **contrast_feat** (*tensor*) – BBox features with shape (N, C) used for contrast loss.
- **proposal_ious** (*tensor*) – IoU between proposal and ground truth corresponding to each BBox features with shape (N).
- **labels** (*tensor*) – Labels for each BBox features with shape (N).
- **reduction_override** (*str | None*) – The reduction method used to override the original reduction method of the loss. Options are “none”, “mean” and “sum”. Default: None.

Returns The calculated loss.

Return type Dict

set_decay_rate(*decay_rate: float*) → None

Contrast loss weight decay hook will set the *decay_rate* according to iterations.

Parameters **decay_rate** (*float*) – Decay rate for weight decay.

class mmfewshot.detection.models.roi_heads.**ContrastiveRoIHead**(*bbox_roi_extractor=None*, *bbox_head=None*, *mask_roi_extractor=None*, *mask_head=None*, *shared_head=None*, *train_cfg=None*, *test_cfg=None*, *pretrained=None*, *init_cfg=None*)

RoI head for FSCE.

class mmfewshot.detection.models.roi_heads.**CosineSimBBoxHead**(*scale: int = 20*, *learnable_scale: bool = False*, *eps: float = 1e-05*, **args*, ***kwargs*)

BBOxHead for TFA.

The code is modified from the official implementation https://github.com/ucbdrive/few-shot-object-detection/

Parameters

- **scale** (*int*) – Scaling factor of *cls_score*. Default: 20.
- **learnable_scale** (*bool*) – Learnable global scaling factor. Default: False.
- **eps** (*float*) – Constant variable to avoid division by zero.

forward(*x: torch.Tensor*) → Tuple[torch.Tensor, torch.Tensor]

Forward function.

Parameters **x** (*Tensor*) – Shape of (num_proposals, C, H, W).

Returns

cls_score (Tensor): Cls scores, has shape (num_proposals, num_classes).

bbox_pred (Tensor): Box energies / deltas, has shape (num_proposals, 4).

Return type tuple

class mmfewshot.detection.models.roi_heads.**FSDetViewRoIHead**(*aggregation_layer: Optional[Dict] = None*, ***kwargs*)

Roi head for FSDetView.

Parameters **aggregation_layer** (*dict*) – Config of *aggregation_layer*. Default: None.

class mmfewshot.detection.models.roi_heads.**MetaRCNNResLayer**(**args*, ***kwargs*)

Shared resLayer for metarcnn and fsdetview.

It provides different forward logics for query and support images.

forward(*x: torch.Tensor*) → torch.Tensor

Forward function for query images.

Parameters **x** (*Tensor*) – Features from backbone with shape (N, C, H, W).

Returns Shape of (N, C).

Return type Tensor

forward_support(*x: torch.Tensor*) → torch.Tensor

Forward function for support images.

Parameters **x** (*Tensor*) – Features from backbone with shape (N, C, H, W).

Returns Shape of (N, C).

Return type Tensor

class mmfewshot.detection.models.roi_heads.**MetaRCNNRoIHead**(*aggregation_layer: Optional[mmcv.utils.config.ConfigDict] = None*, ***kwargs*)

Roi head for MetaRCNN.

Parameters **aggregation_layer** (*ConfigDict*) – Config of *aggregation_layer*. Default: None.

extract_query_roi_feat(*feats: List[torch.Tensor]*, *rois: torch.Tensor*) → torch.Tensor

Extracting query BBOX features, which is used in both training and testing.

Parameters

- **feats** (*list[Tensor]*) – List of query features, each item with shape (N, C, H, W).
- **rois** (*Tensor*) – shape with (m, 5).

Returns RoI features with shape (N, C).

Return type Tensor

extract_support_feats(*feats: List[torch.Tensor]*) → List[torch.Tensor]

Forward support features through shared layers.

Parameters **feats** (*list[Tensor]*) – List of support features, each item with shape (N, C, H, W).

Returns

List of support features, each item with shape (N, C).

Return type list[Tensor]

forward_train(*query_feats: List[torch.Tensor]*, *support_feats: List[torch.Tensor]*, *proposals: List[torch.Tensor]*, *query_img_metas: List[Dict]*, *query_gt_bboxes: List[torch.Tensor]*, *query_gt_labels: List[torch.Tensor]*, *support_gt_labels: List[torch.Tensor]*, *query_gt_bboxes_ignore: Optional[List[torch.Tensor]] = None*, ***kwargs*) → Dict

Forward function for training.

Parameters

- **query_feats** (*list[Tensor]*) – List of query features, each item with shape (N, C, H, W).
- **support_feats** (*list[Tensor]*) – List of support features, each item with shape (N, C, H, W).
- **proposals** (*list[Tensor]*) – List of region proposals with positive and negative pairs.
- **query_img_metas** (*list[dict]*) – List of query image info dict where each dict has: 'img_shape', 'scale_factor', 'flip', and may also contain 'filename', 'ori_shape', 'pad_shape', and 'img_norm_cfg'. For details on the values of these keys see *mmdet/datasets/pipelines/formatting.py:Collect*.
- **query_gt_bboxes** (*list[Tensor]*) – Ground truth bboxes for each query image, each item with shape (num_gts, 4) in [tl_x, tl_y, br_x, br_y] format.
- **query_gt_labels** (*list[Tensor]*) – Class indices corresponding to each box of query images, each item with shape (num_gts).
- **support_gt_labels** (*list[Tensor]*) – Class indices corresponding to each box of support images, each item with shape (1).

- **query_gt_bboxes_ignore** (*list[Tensor] | None*) – Specify which bounding boxes can be ignored when computing the loss. Default: None.

Returns A dictionary of loss components

Return type dict[str, Tensor]

simple_test(*query_feats: List[torch.Tensor]*, *support_feats_dict: Dict*, *proposal_list: List[torch.Tensor]*, *query_img_metas: List[Dict]*, *rescale: bool = False*) → List[List[numpy.ndarray]]

Test without augmentation.

Parameters

- **query_feats** (*list[Tensor]*) – Features of query image, each item with shape (N, C, H, W).
- **support_feats_dict** (*dict[int, Tensor]*) – used for inference only, each key is the class id and value is the support template features with shape (1, C).
- **proposal_list** (*list[Tensors]*) – list of region proposals.
- **query_img_metas** (*list[dict]*) – list of image info dict where each dict has: *img_shape*, *scale_factor*, *flip*, and may also contain *filename*, *ori_shape*, *pad_shape*, and *img_norm_cfg*. For details on the values of these keys see `mmdet.datasets.pipelines.Collect`.
- **rescale** (*bool*) – Whether to rescale the results. Default: False.

Returns

BBox results of each image and classes. The outer list corresponds to each image. The inner list corresponds to each class.

Return type list[list[np.ndarray]]

simple_test_bboxes(*query_feats: List[torch.Tensor]*, *support_feats_dict: Dict*, *query_img_metas: List[Dict]*, *proposals: List[torch.Tensor]*, *rcnn_test_cfg: mmcv.utils.config.ConfigDict*, *rescale: bool = False*) → Tuple[List[torch.Tensor], List[torch.Tensor]]

Test only det bboxes without augmentation.

Parameters

- **query_feats** (*list[Tensor]*) – Features of query image, each item with shape (N, C, H, W).
- **support_feats_dict** (*dict[int, Tensor]*) – used for inference only, each key is the class id and value is the support template features with shape (1, C).
- **query_img_metas** (*list[dict]*) – list of image info dict where each dict has: *img_shape*, *scale_factor*, *flip*, and may also contain *filename*, *ori_shape*, *pad_shape*, and *img_norm_cfg*. For details on the values of these keys see `mmdet.datasets.pipelines.Collect`.
- **proposals** (*list[Tensor]*) – Region proposals.
- **(obj** (*rcnn_test_cfg*) – *ConfigDict*): *test_cfg* of R-CNN.
- **rescale** (*bool*) – If True, return boxes in original image space. Default: False.

Returns

Each tensor in first list with shape (num_boxes, 4) and with shape (num_boxes,) in second list. The length of both lists should be equal to batch_size.

Return type tuple[list[Tensor], list[Tensor]]

class mmfewshot.detection.models.roi_heads.**MultiRelationBBoxHead**(*patch_relation: bool = True*, *local_correlation: bool = True*, *global_relation: bool = True*, **args*, ***kwargs*)

BBox head for Attention RPN.

Parameters

- **patch_relation** (*bool*) – Whether use patch_relation head for classification. Following the official implementation, *patch_relation* always be True, because only patch relation head contain regression head. Default: True.
- **local_correlation** (*bool*) – Whether use local_correlation head for classification. Default: True.
- **global_relation** (*bool*) – Whether use global_relation head for classification. Default: True.

forward(*query_feat: torch.Tensor*, *support_feat: torch.Tensor*) → Tuple[torch.Tensor, torch.Tensor]

Forward function.

Parameters

- **query_feat** (*Tensor*) – Shape of (num_proposals, C, H, W).
- **support_feat** (*Tensor*) – Shape of (1, C, H, W).

Returns

cls_score (Tensor): Cls scores, has shape (num_proposals, num_classes).

bbox_pred (Tensor): Box energies / deltas, has shape (num_proposals, 4).

Return type tuple

loss(*cls_scores: torch.Tensor*, *bbox_preds: torch.Tensor*, *rois: torch.Tensor*, *labels: torch.Tensor*, *label_weights: torch.Tensor*, *bbox_targets: torch.Tensor*, *bbox_weights: torch.Tensor*, *num_pos_pair_samples: int*, *reduction_override: Optional[str] = None*, *sample_fractions: Sequence[Union[int, float]] = (1, 2, 1)*) → Dict

Compute losses of the head.

Parameters

- **cls_scores** (*Tensor*) – Box scores with shape of (num_proposals, num_classes)
- **bbox_preds** (*Tensor*) – Box energies / deltas with shape of (num_proposals, num_classes * 4)
- **rois** (*Tensor*) – shape (N, 4) or (N, 5)
- **labels** (*Tensor*) – Labels of proposals with shape (num_proposals).
- **label_weights** (*Tensor*) – Label weights of proposals with shape (num_proposals).
- **bbox_targets** (*Tensor*) – BBox regression targets of each proposal weight with shape (num_proposals, num_classes * 4).
- **bbox_weights** (*Tensor*) – BBox regression loss weights of each proposal with shape (num_proposals, num_classes * 4).
- **num_pos_pair_samples** (*int*) – Number of samples from positive pairs.
- **reduction_override** (*str | None*) – The reduction method used to override the original reduction method of the loss. Options are “none”, “mean” and “sum”. Default: None.

- **sample_fractions** (*Sequence[int | float]*) – Fractions of positive samples, negative samples from positive pair, negative samples from negative pair. Default: (1, 2, 1).

Returns A dictionary of loss components.

Return type dict[str, Tensor]

class mmfewshot.detection.models.roi_heads.**MultiRelationRoIHead**(*num_support_ways: int = 2*, *num_support_shots: int = 5*, *sample_fractions: Sequence[Union[int, float]] = (1, 2, 1)*, ***kwargs*)

Roi head for AttentionRPN.

Parameters

- **num_support_ways** (*int*) – Number of sampled classes (pos + neg).
- **num_support_shots** (*int*) – Number of shot for each classes.
- **sample_fractions** (*Sequence[int | float]*) – Fractions of positive samples, negative samples from positive pair, negative samples from negative pair. Default: (1, 2, 1).

extract_roi_feat(*feats: List[torch.Tensor]*, *rois: torch.Tensor*) → torch.Tensor

Extract BBOX feature function used in both training and testing.

Parameters

- **feats** (*list[Tensor]*) – Features from backbone, each item with shape (N, C, W, H).
- **rois** (*Tensor*) – shape (num_proposals, 5).

Returns Roi features with shape (num_proposals, C).

Return type Tensor

forward_train(*query_feats: List[torch.Tensor]*, *support_feats: List[torch.Tensor]*, *proposals: List[torch.Tensor]*, *query_img_metas: List[Dict]*, *query_gt_bboxes: List[torch.Tensor]*, *query_gt_labels: List[torch.Tensor]*, *support_gt_bboxes: List[torch.Tensor]*, *query_gt_bboxes_ignore: Optional[List[torch.Tensor]] = None*, ***kwargs*) → Dict

All arguments excepted proposals are passed in tuple of (query, support).

Parameters

- **query_feats** (*list[Tensor]*) – List of query features, each item with shape (N, C, H, W).
- **support_feats** (*list[Tensor]*) – List of support features, each item with shape (N, C, H, W).
- **proposals** (*list[Tensor]*) – List of region proposals with positive and negative query-support pairs.
- **query_img_metas** (*list[dict]*) – List of query image info dict where each dict has: 'img_shape', 'scale_factor', 'flip', and may also contain 'filename', 'ori_shape', 'pad_shape', and 'img_norm_cfg'. For details on the values of these keys see *mmdet/datasets/pipelines/formatting.py:Collect*.
- **query_gt_bboxes** (*list[Tensor]*) – Ground truth bboxes for each query image with shape (num_gts, 4) in [tl_x, tl_y, br_x, br_y] format.
- **query_gt_labels** (*list[Tensor]*) – Class indices corresponding to each bbox from query image.

- **support_gt_bboxes** (*list[Tensor]*) – Ground truth bboxes for each support image with shape (num_gts, 4) in [tl_x, tl_y, br_x, br_y] format.
- **query_gt_bboxes_ignore** (*None | list[Tensor]*) – Specify which bounding boxes from query image can be ignored when computing the loss. Default: None.

Returns A dictionary of loss components.

Return type dict[str, Tensor]

simple_test(*query_feats: List[torch.Tensor]*, *support_feat: torch.Tensor*, *proposals: List[torch.Tensor]*, *query_img_metas: List[Dict]*, *rescale: bool = False*) → List[List[numpy.ndarray]]

Test without augmentation.

Parameters

- **query_feats** (*list[Tensor]*) – List of query features, each item with shape (N, C, H, W).
- **support_feat** (*Tensor*) – Support features with shape (N, C, H, W).
- **proposals** (*Tensor or list[Tensor]*) – list of region proposals.
- **query_img_metas** (*list[dict]*) – list of query image info dict where each dict has: *img_shape*, *scale_factor*, *flip*, and may also contain *filename*, *ori_shape*, *pad_shape*, and *img_norm_cfg*. For details on the values of these keys see `mmdet.datasets.pipelines.Collect`.
- **proposals** – Region proposals. Default: None.
- **rescale** (*bool*) – Whether to rescale the results. Default: False.

Returns

BBox results of each image and classes. The outer list corresponds to each image. The inner list corresponds to each class.

Return type list[list[np.ndarray]]

simple_test_bboxes(*query_feats: List[torch.Tensor]*, *support_feat: torch.Tensor*, *query_img_metas: List[Dict]*, *proposals: List[torch.Tensor]*, *rcnn_test_cfg: mmcv.utils.config.ConfigDict*, *rescale: bool = False*) → Tuple[List[torch.Tensor], List[torch.Tensor]]

Test only det bboxes without augmentation.

Parameters

- **query_feats** (*list[Tensor]*) – List of query features, each item with shape (N, C, H, W).
- **support_feat** (*Tensor*) – Support feature with shape (N, C, H, W).
- **query_img_metas** (*list[dict]*) – list of image info dict where each dict has: *img_shape*, *scale_factor*, *flip*, and may also contain *filename*, *ori_shape*, *pad_shape*, and *img_norm_cfg*. For details on the values of these keys see `mmdet.datasets.pipelines.Collect`.
- **proposals** (*list[Tensor]*) – Region proposals.
- **(obj** (*rcnn_test_cfg*) – *ConfigDict*): *test_cfg* of R-CNN.
- **rescale** (*bool*) – If True, return boxes in original image space. Default: False.

Returns

BBox of shape [N, num_bboxes, 5] and class labels of shape [N, num_bboxes].

Return type tuple[Tensor, Tensor]

class mmfewshot.detection.models.roi_heads.**TwoBranchRoIHead**(*bbox_roi_extractor=None*, *bbox_head=None*, *mask_roi_extractor=None*, *mask_head=None*, *shared_head=None*, *train_cfg=None*, *test_cfg=None*, *pretrained=None*, *init_cfg=None*)

RoI head for MPSR.

forward_auxiliary_train(*feats: Tuple[torch.Tensor]*, *gt_labels: List[torch.Tensor]*) → Dict

Forward function and calculate loss for auxiliary data in training.

Parameters

- **feats** (*tuple[Tensor]*) – List of features at multiple scales, each is a 4D-tensor.
- **gt_labels** (*list[Tensor]*) – List of class indices corresponding to each features, each is a 4D-tensor.

Returns a dictionary of loss components

Return type dict[str, Tensor]

24.4.6 utils

24.5 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

TWENTYFIVE

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.

mmfewshot.utils.**sync_random_seed**(*seed=None*, *device='cuda'*)

Propagating the seed of rank 0 to all other ranks.

Make sure different ranks share the same seed. All workers must call this function, otherwise it will deadlock. This method is generally used in *DistributedSampler*, because the seed should be identical across all processes in the distributed group. In distributed sampling, different ranks should sample non-overlapped data in the dataset. Therefore, this function is used to make sure that each rank shuffles the data indices in the same order based on the same seed. Then different ranks could use different indices to select non-overlapped data from the same data list. :param seed: The seed. Default to None. :type seed: int, Optional :param device: The device where the seed will be put on.

Default to 'cuda'.

Returns Seed to be used.

Return type int

CHAPTER

TWENTYSIX

INDICES AND TABLES

- genindex
- search

PYTHON MODULE INDEX

m

mmfewshot.classification.apis, 101
mmfewshot.classification.core, 105
mmfewshot.classification.core.evaluation, 105
mmfewshot.classification.datasets, 106
mmfewshot.classification.models, 111
mmfewshot.classification.models.backbones, 111
mmfewshot.classification.models.heads, 112
mmfewshot.classification.models.losses, 117
mmfewshot.classification.models.utils, 118
mmfewshot.classification.utils, 118
mmfewshot.detection.apis, 121
mmfewshot.detection.core, 123
mmfewshot.detection.core.evaluation, 123
mmfewshot.detection.core.utils, 125
mmfewshot.detection.datasets, 125
mmfewshot.detection.models, 135
mmfewshot.detection.models.backbones, 136
mmfewshot.detection.models.dense_heads, 136
mmfewshot.detection.models.detectors, 141
mmfewshot.detection.models.losses, 150
mmfewshot.detection.models.roi_heads, 151
mmfewshot.detection.models.utils, 158
mmfewshot.detection.utils, 158
mmfewshot.utils, 159

INDEX

A

ann_cfg_parser() (*mmfewshot.detection.datasets.BaseFewShotDataset method*), 127
AttentionRPNDetector (*class in mmfewshot.detection.models.detectors*), 141
AttentionRPNHead (*class in mmfewshot.detection.models.dense_heads*), 136
aug_test() (*mmfewshot.detection.models.detectors.QuerySupportDetector method*), 147

B

BaseFewShotDataset (*class in mmfewshot.classification.datasets*), 106
BaseFewShotDataset (*class in mmfewshot.detection.datasets*), 125
before_forward_query() (*mmfewshot.classification.models.heads.CosineDistanceHead method*), 113
before_forward_query() (*mmfewshot.classification.models.heads.LinearHead method*), 113
before_forward_query() (*mmfewshot.classification.models.heads.MatchingHead method*), 114
before_forward_query() (*mmfewshot.classification.models.heads.MetaBaselineHead method*), 114
before_forward_query() (*mmfewshot.classification.models.heads.NegMarginHead method*), 115
before_forward_query() (*mmfewshot.classification.models.heads.PrototypeHead method*), 115
before_forward_query() (*mmfewshot.classification.models.heads.RelationHead method*), 116
before_forward_support() (*mmfewshot.classification.models.heads.CosineDistanceHead method*), 113
before_forward_support() (*mmfewshot.classification.models.heads.LinearHead method*), 113
before_forward_support() (*mmfewshot.classification.models.heads.MatchingHead method*), 114
before_forward_support() (*mmfewshot.classification.models.heads.MetaBaselineHead method*), 114
before_forward_support() (*mmfewshot.classification.models.heads.NegMarginHead method*), 115
before_forward_support() (*mmfewshot.classification.models.heads.PrototypeHead method*), 115
before_forward_support() (*mmfewshot.classification.models.heads.RelationHead method*), 116
build_backbone() (*in module mmfewshot.detection.models*), 135
build_dataloader() (*in module mmfewshot.classification.datasets*), 109
build_dataloader() (*in module mmfewshot.detection.datasets*), 134
build_detector() (*in module mmfewshot.detection.models*), 135
build_head() (*in module mmfewshot.detection.models*), 135
build_loss() (*in module mmfewshot.detection.models*), 135
build_meta_test_dataloader() (*in module mmfewshot.classification.datasets*), 110
build_neck() (*in module mmfewshot.detection.models*), 135
build_roi_extractor() (*in module mmfewshot.detection.models*), 135
build_shared_head() (*in module mmfewshot.detection.models*), 135

C

cache_feats() (*mmfewshot.classification.datasets.MetaTestDataset method*), 108
class_to_idx (*mmfew-*

shot.classification.datasets.BaseFewShotDataset property), 106
ContrastiveBBoxHead (*class in mmfewshot.detection.models.roi_heads*), 151
ContrastiveLossDecayHook (*class in mmfewshot.detection.core.utils*), 125
ContrastiveLossDecayHook (*class in mmfewshot.detection.utils*), 158
ContrastiveRoIHead (*class in mmfewshot.detection.models.roi_heads*), 152
Conv4 (*class in mmfewshot.classification.models.backbones*), 111
convert_maml_module() (*in module mmfewshot.classification.models.utils*), 118
convert_query_to_support() (*mmfewshot.detection.datasets.NWayKShotDataset method*), 132
ConvNet (*class in mmfewshot.classification.models.backbones*), 111
CosineDistanceHead (*class in mmfewshot.classification.models.heads*), 112
CosineSimBBoxHead (*class in mmfewshot.detection.models.roi_heads*), 152
CropResizeInstance (*class in mmfewshot.detection.datasets*), 127
CUBDataset (*class in mmfewshot.classification.datasets*), 107

D

default() (*mmfewshot.detection.datasets.NumpyEncoder method*), 133
DistMetaTestEvalHook (*class in mmfewshot.classification.core.evaluation*), 105
DistributedInfiniteGroupSampler (*class in mmfewshot.utils*), 159
DistributedInfiniteSampler (*class in mmfewshot.utils*), 159

E

EpisodicDataset (*class in mmfewshot.classification.datasets*), 107
eval_map() (*in module mmfewshot.detection.core.evaluation*), 124
evaluate() (*mmfewshot.classification.datasets.BaseFewShotDataset static method*), 106
evaluate() (*mmfewshot.classification.datasets.EpisodicDataset method*), 108
evaluate() (*mmfewshot.detection.datasets.FewShotCocoDataset method*), 128
evaluate() (*mmfewshot.detection.datasets.FewShotVOCDataset method*), 130
extract_auxiliary_feat() (*mmfewshot.detection.models.detectors.MPSR method*), 143
extract_feat() (*mmfewshot.detection.models.detectors.MPSR method*), 143
extract_feat() (*mmfewshot.detection.models.detectors.QuerySupportDetector method*), 147
extract_query_feat() (*mmfewshot.detection.models.detectors.QuerySupportDetector method*), 148
extract_query_roi_feat() (*mmfewshot.detection.models.roi_heads.MetaRCNNRoIHead method*), 153
extract_roi_feat() (*mmfewshot.detection.models.dense_heads.AttentionRPNHead method*), 136
extract_roi_feat() (*mmfewshot.detection.models.roi_heads.MultiRelationRoIHead method*), 156
extract_support_feat() (*mmfewshot.detection.models.detectors.AttentionRPNDetector method*), 142
extract_support_feat() (*mmfewshot.detection.models.detectors.MetaRCNN method*), 145
extract_support_feat() (*mmfewshot.detection.models.detectors.QuerySupportDetector method*), 148
extract_support_feats() (*mmfewshot.detection.models.roi_heads.MetaRCNNRoIHead method*), 153

F

FewShotCocoDataset (*class in mmfewshot.detection.datasets*), 127
FewShotVOCDataset (*class in mmfewshot.detection.datasets*), 129
forward() (*mmfewshot.classification.models.backbones.ConvNet method*), 111
forward() (*mmfewshot.classification.models.backbones.ResNet12 method*), 111
forward() (*mmfewshot.classification.models.backbones.WideResNet method*), 112
forward() (*mmfewshot.classification.models.losses.MSELoss method*), 117
forward() (*mmfewshot.classification.models.losses.NLLLoss method*), 118
forward() (*mmfewshot.classification.utils.MetaTestParallel method*), 119
forward() (*mmfewshot.detection.models.backbones.ResNetWithMetaConv method*), 136
forward() (*mmfewshot.detection.models.detectors.MPSR method*), 143
forward() (*mmfewshot.detection.models.detectors.QuerySupportDetector method*), 148

forward() (*mmfewshot.detection.models.losses.SupervisedContrastiveLoss method*), 150
forward() (*mmfewshot.detection.models.roi_heads.ContrastiveBBoxHead method*), 151
forward() (*mmfewshot.detection.models.roi_heads.CosineSimBBoxHead method*), 152
forward() (*mmfewshot.detection.models.roi_heads.MetaRCNNResLayer method*), 152
forward() (*mmfewshot.detection.models.roi_heads.MultiRelationBBoxHead method*), 155
forward_auxiliary() (*mmfewshot.detection.models.dense_heads.TwoBranchRPNHead method*), 138
forward_auxiliary_single() (*mmfewshot.detection.models.dense_heads.TwoBranchRPNHead method*), 138
forward_auxiliary_train() (*mmfewshot.detection.models.roi_heads.TwoBranchRoIHead method*), 158
forward_model_init() (*mmfewshot.detection.models.detectors.AttentionRPNDetector method*), 142
forward_model_init() (*mmfewshot.detection.models.detectors.MetaRCNN method*), 146
forward_model_init() (*mmfewshot.detection.models.detectors.QuerySupportDetector method*), 148
forward_query() (*mmfewshot.classification.models.heads.CosineDistanceHead method*), 113
forward_query() (*mmfewshot.classification.models.heads.LinearHead method*), 113
forward_query() (*mmfewshot.classification.models.heads.MatchingHead method*), 114
forward_query() (*mmfewshot.classification.models.heads.MetaBaselineHead method*), 114
forward_query() (*mmfewshot.classification.models.heads.NegMarginHead method*), 115
forward_query() (*mmfewshot.classification.models.heads.PrototypeHead method*), 116
forward_query() (*mmfewshot.classification.models.heads.RelationHead method*), 116
forward_relation_module() (*mmfewshot.classification.models.heads.RelationHead method*), 116
forward_single() (*mmfewshot.detection.models.dense_heads.TwoBranchRPNHead method*), 138
forward_support() (*mmfewshot.classification.models.heads.CosineDistanceHead method*), 113
forward_support() (*mmfewshot.classification.models.heads.LinearHead method*), 113
forward_support() (*mmfewshot.classification.models.heads.MatchingHead method*), 114
forward_support() (*mmfewshot.classification.models.heads.MetaBaselineHead method*), 114
forward_support() (*mmfewshot.classification.models.heads.NegMarginHead method*), 115
forward_support() (*mmfewshot.classification.models.heads.PrototypeHead method*), 116
forward_support() (*mmfewshot.classification.models.heads.RelationHead method*), 116
forward_support() (*mmfewshot.detection.models.roi_heads.MetaRCNNResLayer method*), 152
forward_train() (*mmfewshot.classification.models.heads.CosineDistanceHead method*), 113
forward_train() (*mmfewshot.classification.models.heads.LinearHead method*), 113
forward_train() (*mmfewshot.classification.models.heads.MatchingHead method*), 114
forward_train() (*mmfewshot.classification.models.heads.MetaBaselineHead method*), 114
forward_train() (*mmfewshot.classification.models.heads.NegMarginHead method*), 115
forward_train() (*mmfewshot.classification.models.heads.PrototypeHead method*), 116
forward_train() (*mmfewshot.classification.models.heads.RelationHead method*), 116
forward_train() (*mmfewshot.detection.models.dense_heads.AttentionRPNHead method*), 137
forward_train() (*mmfewshot.detection.models.dense_heads.TwoBranchRPNHead method*), 138
forward_train() (*mmfewshot.detection.models.detectors.MPSR*

method), 144
forward_train() (*mmfewshot.detection.models.detectors.QuerySupportDetector method*), 148
forward_train() (*mmfewshot.detection.models.roi_heads.MetaRCNNRoIHead method*), 153
forward_train() (*mmfewshot.detection.models.roi_heads.MultiRelationRoIHead method*), 156
FSCE (*class in mmfewshot.detection.models.detectors*), 143
FSDetView (*class in mmfewshot.detection.models.detectors*), 143
FSDetViewRoIHead (*class in mmfewshot.detection.models.roi_heads*), 152

G

generate_episodic_idxes() (*mmfewshot.classification.datasets.EpisodicDataset method*), 108
generate_support() (*mmfewshot.detection.datasets.QueryAwareDataset method*), 134
generate_support_batch_indices() (*mmfewshot.detection.datasets.NWayKShotDataset method*), 132
GenerateMask (*class in mmfewshot.detection.datasets*), 131
get_ann_info() (*mmfewshot.detection.datasets.BaseFewShotDataset method*), 127
get_bboxes() (*mmfewshot.detection.models.dense_heads.TwoBranchRPNHead method*), 139
get_cat_ids() (*mmfewshot.detection.datasets.FewShotCocoDataset method*), 129
get_classes() (*mmfewshot.classification.datasets.BaseFewShotDataset class method*), 106
get_classes() (*mmfewshot.classification.datasets.CUBDataset method*), 107
get_classes() (*mmfewshot.classification.datasets.MiniImageNetDataset method*), 108
get_classes() (*mmfewshot.classification.datasets.TieredImageNetDataset method*), 109
get_classes() (*mmfewshot.detection.datasets.FewShotCocoDataset method*), 129
get_classes() (*mmfewshot.detection.datasets.FewShotVOCDataset method*), 131
get_copy_dataset_type() (*in module mmfewshot.detection.datasets*), 135
get_episode_class_ids() (*mmfewshot.classification.datasets.EpisodicDataset method*), 108
get_general_classes() (*mmfewshot.classification.datasets.TieredImageNetDataset method*), 109
get_support_data_infos() (*mmfewshot.detection.datasets.NWayKShotDataset method*), 133
get_support_data_infos() (*mmfewshot.detection.datasets.QueryAwareDataset method*), 134

I

inference_classifier() (*in module mmfewshot.classification.apis*), 101
inference_detector() (*in module mmfewshot.detection.apis*), 121
InfiniteEpochBasedRunner (*class in mmfewshot.utils*), 159
InfiniteGroupSampler (*class in mmfewshot.utils*), 160
InfiniteSampler (*class in mmfewshot.utils*), 160
init_classifier() (*in module mmfewshot.classification.apis*), 101
init_detector() (*in module mmfewshot.detection.apis*), 121
init_weights() (*mmfewshot.classification.models.heads.RelationHead method*), 117

L

label_wrapper() (*in module mmfewshot.classification.datasets*), 110
LinearHead (*class in mmfewshot.classification.models.heads*), 113
load_annotations() (*mmfewshot.classification.datasets.CUBDataset method*), 107
load_annotations() (*mmfewshot.classification.datasets.MiniImageNetDataset method*), 109
load_annotations() (*mmfewshot.classification.datasets.TieredImageNetDataset method*), 109
load_annotations() (*mmfewshot.detection.datasets.FewShotCocoDataset method*), 129
load_annotations() (*mmfewshot.detection.datasets.FewShotVOCDataset*

method), 131
load_annotations_coco() (*mmfewshot.detection.datasets.FewShotCocoDataset method*), 129
load_annotations_saved() (*mmfewshot.detection.datasets.BaseFewShotDataset method*), 127
load_annotations_xml() (*mmfewshot.detection.datasets.FewShotVOCDataset method*), 131
LoadImageFromBytes (*class in mmfewshot.classification.datasets*), 108
local_numpy_seed() (*in module mmfewshot.utils*), 160
loss() (*mmfewshot.detection.models.dense_heads.AttentionRPNHead method*), 137
loss() (*mmfewshot.detection.models.dense_heads.TwoBranchRPNHead method*), 139
loss() (*mmfewshot.detection.models.roi_heads.MultiRelationBBoxHead method*), 155
loss_bbox_single() (*mmfewshot.detection.models.dense_heads.TwoBranchRPNHead method*), 140
loss_cls_single() (*mmfewshot.detection.models.dense_heads.TwoBranchRPNHead method*), 140
loss_contrast() (*mmfewshot.detection.models.roi_heads.ContrastiveBBoxHead method*), 151

M

MatchingHead (*class in mmfewshot.classification.models.heads*), 113
MetaBaselineHead (*class in mmfewshot.classification.models.heads*), 114
MetaRCNN (*class in mmfewshot.detection.models.detectors*), 145
MetaRCNNResLayer (*class in mmfewshot.detection.models.roi_heads*), 152
MetaRCNNRoIHead (*class in mmfewshot.detection.models.roi_heads*), 153
MetaTestDataset (*class in mmfewshot.classification.datasets*), 108
MetaTestEvalHook (*class in mmfewshot.classification.core.evaluation*), 105
MetaTestParallel (*class in mmfewshot.classification.utils*), 118
MiniImageNetDataset (*class in mmfewshot.classification.datasets*), 108
mmfewshot.classification.apis
    module, 101
mmfewshot.classification.core
    module, 105
mmfewshot.classification.core.evaluation
    module, 105
mmfewshot.classification.datasets
    module, 106
mmfewshot.classification.models
    module, 111
mmfewshot.classification.models.backbones
    module, 111
mmfewshot.classification.models.heads
    module, 112
mmfewshot.classification.models.losses
    module, 117
mmfewshot.classification.models.utils
    module, 118
mmfewshot.classification.utils
    module, 118
mmfewshot.detection.apis
    module, 121
mmfewshot.detection.core
    module, 123
mmfewshot.detection.core.evaluation
    module, 123
mmfewshot.detection.core.utils
    module, 125
mmfewshot.detection.datasets
    module, 125
mmfewshot.detection.models
    module, 135
mmfewshot.detection.models.backbones
    module, 136
mmfewshot.detection.models.dense_heads
    module, 136
mmfewshot.detection.models.detectors
    module, 141
mmfewshot.detection.models.losses
    module, 150
mmfewshot.detection.models.roi_heads
    module, 151
mmfewshot.detection.models.utils
    module, 158
mmfewshot.detection.utils
    module, 158
mmfewshot.utils
    module, 159
model_init() (*mmfewshot.detection.models.detectors.AttentionRPNDetector method*), 142
model_init() (*mmfewshot.detection.models.detectors.MetaRCNN method*), 146
model_init() (*mmfewshot.detection.models.detectors.QuerySupportDetector method*), 149
module
    mmfewshot.classification.apis, 101
    mmfewshot.classification.core, 105

mmfewshot.classification.core.evaluation, 105
mmfewshot.classification.datasets, 106
mmfewshot.classification.models, 111
mmfewshot.classification.models.backbones, 111
mmfewshot.classification.models.heads, 112
mmfewshot.classification.models.losses, 117
mmfewshot.classification.models.utils, 118
mmfewshot.classification.utils, 118
mmfewshot.detection.apis, 121
mmfewshot.detection.core, 123
mmfewshot.detection.core.evaluation, 123
mmfewshot.detection.core.utils, 125
mmfewshot.detection.datasets, 125
mmfewshot.detection.models, 135
mmfewshot.detection.models.backbones, 136
mmfewshot.detection.models.dense_heads, 136
mmfewshot.detection.models.detectors, 141
mmfewshot.detection.models.losses, 150
mmfewshot.detection.models.roi_heads, 151
mmfewshot.detection.models.utils, 158
mmfewshot.detection.utils, 158
mmfewshot.utils, 159
MPSR (*class in mmfewshot.detection.models.detectors*), 143
MSELoss (*class in mmfewshot.classification.models.losses*), 117
multi_gpu_meta_test() (*in module mmfewshot.classification.apis*), 101
multi_gpu_model_init() (*in module mmfewshot.detection.apis*), 121
multi_gpu_test() (*in module mmfewshot.detection.apis*), 122
multi_pipeline_collate_fn() (*in module mmfewshot.utils*), 160
MultiRelationBBoxHead (*class in mmfewshot.detection.models.roi_heads*), 155
MultiRelationRoIHead (*class in mmfewshot.detection.models.roi_heads*), 156

N

NegMarginHead (*class in mmfewshot.classification.models.heads*), 115
NLLLoss (*class in mmfewshot.classification.models.losses*), 117
NumpyEncoder (*class in mmfewshot.detection.datasets*), 133
NWayKShotDataloader (*class in mmfewshot.detection.datasets*), 131
NWayKShotDataset (*class in mmfewshot.detection.datasets*), 132

P

prepare_train_img() (*mmfewshot.detection.datasets.BaseFewShotDataset method*), 127
process_support_images() (*in module mmfewshot.classification.apis*), 102
process_support_images() (*in module mmfewshot.detection.apis*), 122
PrototypeHead (*class in mmfewshot.classification.models.heads*), 115

Q

QueryAwareDataset (*class in mmfewshot.detection.datasets*), 133
QuerySupportDetector (*class in mmfewshot.detection.models.detectors*), 146
QuerySupportDistEvalHook (*class in mmfewshot.detection.core.evaluation*), 123
QuerySupportEvalHook (*class in mmfewshot.detection.core.evaluation*), 124

R

RelationHead (*class in mmfewshot.classification.models.heads*), 116
ResNet12 (*class in mmfewshot.classification.models.backbones*), 111
ResNetWithMetaConv (*class in mmfewshot.detection.models.backbones*), 136

S

sample_shots_by_class_id() (*mmfewshot.classification.datasets.BaseFewShotDataset method*), 107
sample_support_shots() (*mmfewshot.detection.datasets.QueryAwareDataset method*), 134
save_data_infos() (*mmfewshot.detection.datasets.BaseFewShotDataset method*), 127
save_data_infos() (*mmfewshot.detection.datasets.NWayKShotDataset method*), 133
save_data_infos() (*mmfewshot.detection.datasets.QueryAwareDataset method*), 134
save_support_data_infos() (*mmfewshot.detection.datasets.NWayKShotDataset method*), 133
set_decay_rate() (*mmfewshot.detection.models.roi_heads.ContrastiveBBoxHead method*), 151

set_task_id() (*mmfewshot.classification.datasets.MetaTestDataset method*), 108
show_result_pyplot() (*in module mmfewshot.classification.apis*), 103
simple_test() (*mmfewshot.detection.models.dense_heads.AttentionRPNHead method*), 138
simple_test() (*mmfewshot.detection.models.detectors.AttentionRPNDetector method*), 142
simple_test() (*mmfewshot.detection.models.detectors.MetaRCNN method*), 146
simple_test() (*mmfewshot.detection.models.detectors.QuerySupportDetector method*), 149
simple_test() (*mmfewshot.detection.models.roi_heads.MetaRCNNRoIHead method*), 154
simple_test() (*mmfewshot.detection.models.roi_heads.MultiRelationRoIHead method*), 157
simple_test_bboxes() (*mmfewshot.detection.models.roi_heads.MetaRCNNRoIHead method*), 154
simple_test_bboxes() (*mmfewshot.detection.models.roi_heads.MultiRelationRoIHead method*), 157
single_gpu_meta_test() (*in module mmfewshot.classification.apis*), 103
single_gpu_model_init() (*in module mmfewshot.detection.apis*), 122
single_gpu_test() (*in module mmfewshot.detection.apis*), 123
SupervisedContrastiveLoss (*class in mmfewshot.detection.models.losses*), 150
sync_random_seed() (*in module mmfewshot.utils*), 161

T

test_single_task() (*in module mmfewshot.classification.apis*), 104
TFA (*class in mmfewshot.detection.models.detectors*), 149
TieredImageNetDataset (*class in mmfewshot.classification.datasets*), 109
train_step() (*mmfewshot.detection.models.detectors.MPSR method*), 144
train_step() (*mmfewshot.detection.models.detectors.QuerySupportDetector method*), 149
TwoBranchRoIHead (*class in mmfewshot.detection.models.roi_heads*), 158
TwoBranchRPNHead (*class in mmfewshot.detection.models.dense_heads*), 138

V

val_step() (*mmfewshot.detection.models.detectors.MPSR method*), 145
val_step() (*mmfewshot.detection.models.detectors.QuerySupportDetector method*), 149

W

WideResNet (*class in mmfewshot.classification.models.backbones*), 112
WRN28x10 (*class in mmfewshot.classification.models.backbones*), 112