MMRazor

Release 0.3.1

MMRazor Authors

Oct 25, 2022

GET STARTED

CHAPTER

ONE

PREREQUISITES

- Linux
- Python 3.7+
- PyTorch 1.5+
- CUDA 9.2+ (If you build PyTorch from source, CUDA 9.0 is also compatible)
- GCC 5+
- MMCV

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

CHAPTER

TWO

INSTALLATION

2.1 Prepare environment

1. Create a conda virtual environment and activate it.

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

2. Install PyTorch and torchvision following the official instructions.

   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.1` If you have CUDA 10.2 installed under `/usr/local/cuda` and would like to install PyTorch 1.10, you need to install the prebuilt PyTorch with CUDA 10.2.

```
conda install pytorch torchvision torchaudio cudatoolkit=10.2 -c pytorch
```

   `E.g.2` If you have CUDA 9.2 installed under `/usr/local/cuda` and would like to install PyTorch 1.5.1, you need to install the prebuilt PyTorch with CUDA 9.2.

```
conda install pytorch==1.5.1 torchvision==0.6.1 cudatoolkit=9.2 -c pytorch
```

   If you build PyTorch from source instead of installing the prebuilt package, you can use more CUDA versions such as 9.0.

2.2 Install MMRazor

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

```
pip install openmim
mim install mmrazor
```

Or you can still install MMRazor manually

1. Install mmcv-full.

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

Please replace {cu_version} and {torch_version} in the url to your desired one. For example, to install the latest mmcv-full with CUDA 10.2 and PyTorch 1.10.0, use the following command:

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

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

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

2. Install MMRazor.

You can simply install mmrazor with the following command:

```
pip install mmrazor
```

or:

```
pip install git+https://github.com/open-mmlab/mmrazor.git # install the master␣
↪branch
```

Instead, if you would like to install MMRazor in dev mode, run following:

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

Note:

- When MMRazor is installed on dev mode, any local modifications made to the code will take effect without the need to reinstall it.
- Currently, running pip install -v -e . will install mmcls, mmdet, mmsegmentation. We will work on minimum runtime requirements in future.

2.3 A from-scratch setup script

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

conda install pytorch torchvision cudatoolkit=10.2 -c pytorch
# install the latest mmcv
pip install mmcv-full -f https://download.openmmlab.com/mmcv/dist/cu102/torch1.10.0/
↪index.html
# install mmrazor
git clone https://github.com/open-mmlab/mmrazor.git
cd mmrazor
pip install -v -e .
```

CHAPTER

THREE

MODEL ZOO

3.1 Baselines

3.1.1 CWD

Please refer to CWD for details.

3.1.2 WSLD

Please refer to WSLD for details.

3.1.3 DARTS

Please refer to DARTS for details.

3.1.4 DETNAS

Please refer to DETNAS for details.

3.1.5 SPOS

Please refer to SPOS for details.

3.1.6 AUTOSLIM

Please refer to AUTOSLIM for details.

CHAPTER

FOUR

TRAIN DIFFERENT TYPE ALGORITHMS

Currently our algorithms support mmclassification, mmdetection and mmsegmentation. **Before running our algorithms, you may need to prepare the datasets according to the instructions in the corresponding document.**

Note:

- Since our algorithms **have the same interface for all three tasks**, in the following introduction, we use `${task}` to represent one of `mmclsmmdet` and `mmseg`.
- We dynamically pass arguments `cfg-options` (e.g., `mutable_cfg` in nas algorithm or `channel_cfg` in pruning algorithm) to **avoid the need for a config for each subnet or checkpoint**. If you want to specify different subnets for retraining or testing, you just need to change this arguments.

4.1 NAS

There are three steps to start neural network search(NAS), including **supernet pre-training**, **search for subnet on the trained supernet** and **subnet retraining**.

4.1.1 Supernet Pre-training

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

The usage of optional arguments are the same as corresponding tasks like mmclassification, mmdetection and mmsegmentation.

For example,

4.1.2 Search for Subnet on The Trained Supernet

```
python tools/${task}/search_${task}.py ${CONFIG_FILE} ${CHECKPOINT_PATH} [optional␣
↪arguments]
```

For example,

4.1.3 Subnet Retraining

```
python tools/${task}/train_${task}.py ${CONFIG_FILE} --cfg-options algorithm.mutable_cfg=
↪${MUTABLE_CFG_PATH} [optional arguments]
```

- MUTABLE_CFG_PATH: Path of mutable_cfg. mutable_cfg represents **config for mutable of the subnet searched out**, used to specify different subnets for retraining. An example for mutable_cfg can be found here, and the usage can be found here.

For example,

We note that instead of using --cfg-options, you can also directly modify configs/nas/spos/spos_subnet_shufflenetv2_8xb128_in1k.py like this:

4.2 Pruning

Pruning has three steps, including **supernet pre-training**, **search for subnet on the trained supernet** and **subnet retraining**. The commands of the first two steps are similar to NAS, except that we need to use CONFIG_FILE of Pruning here. The commands of the **subnet retraining** are as follows.

4.2.1 Subnet Retraining

```
python tools/${task}/train_${task}.py ${CONFIG_FILE} --cfg-options algorithm.channel_cfg=
↪${CHANNEL_CFG_PATH} [optional arguments]
```

Different from NAS, the argument that needs to be specified here is channel_cfg instead of mutable_cfg.

- CHANNEL_CFG_PATH: Path of channel_cfg. channel_cfg represents **config for channel of the subnet searched out**, used to specify different subnets for testing. An example for channel_cfg can be found here, and the usage can be found here.

For example,

4.3 Distillation

There is only one step to start knowledge distillation.

```
python tools/${task}/train_${task}.py ${CONFIG_FILE} --cfg-options algorithm.distiller.
↪teacher.init_cfg.type=Pretrained algorithm.distiller.teacher.init_cfg.checkpoint=$
↪{TEACHER_CHECKPOINT_PATH} [optional arguments]
```

- TEACHER_CHECKPOINT_PATH: Path of teacher_checkpoint. teacher_checkpoint represents **checkpoint of teacher model**, used to specify different checkpoints for distillation.

For example,

CHAPTER

FIVE

TRAIN WITH DIFFERENT DEVICES

Note: The default learning rate in config files is for 8 GPUs. If using different number GPUs, the total batch size will change in proportion, you have to scale the learning rate following `new_lr = old_lr * new_ngpus / old_ngpus`. We recommend to use `tools/xxx/dist_train.sh` even with 1 gpu, since some methods do not support non-distributed training.

5.1 Training with CPU

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

Note: We do not recommend users to use CPU for training because it is too slow and some algorithms are using `SyncBN` which is based on distributed training. We support this feature to allow users to debug on machines without GPU for convenience.

5.2 Train with single/multiple GPUs

```
sh tools/dist_train.sh ${CONFIG_FILE} ${GPUS} --work_dir ${YOUR_WORK_DIR} [optional␣
↪arguments]
```

Note: During training, checkpoints and logs are saved in the same folder structure as the config file under `work_dirs/`. Custom work directory is not recommended since evaluation scripts infer work directories from the config file name. If you want to save your weights somewhere else, please use symlink, for example:

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

Alternatively, if you run MMRazor on a cluster managed with slurm:

```
GPUS_PER_NODE=${GPUS_PER_NODE} GPUS=${GPUS} SRUN_ARGS=${SRUN_ARGS} sh tools/xxx/slurm_
↪train_xxx.sh ${PARTITION} ${JOB_NAME} ${CONFIG_FILE} ${YOUR_WORK_DIR} [optional␣
↪arguments]
```

5.3 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/xxx/dist_train.
↪sh $CONFIG $GPUS
```

On the second machine:

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

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

If you launch with slurm, the command is the same as that on single machine described above, but you need refer to slurm_train.sh to set appropriate parameters and environment variables.

5.4 Launch multiple jobs on a single machine

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

If you use `dist_train.sh` to launch training jobs:

```
CUDA_VISIBLE_DEVICES=0,1,2,3 PORT=29500 sh tools/xxx/dist_train.sh ${CONFIG_FILE} 4 --
↪work_dir tmp_work_dir_1
CUDA_VISIBLE_DEVICES=4,5,6,7 PORT=29501 sh tools/xxx/dist_train.sh ${CONFIG_FILE} 4 --
↪work_dir tmp_work_dir_2
```

If you use launch training jobs with slurm, you have two options to set different communication ports:

Option 1:

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 and config2.py.

```
CUDA_VISIBLE_DEVICES=0,1,2,3 GPUS=4 sh tools/xxx/slurm_train_xxx.sh ${PARTITION} ${JOB_
↪NAME} config1.py tmp_work_dir_1
CUDA_VISIBLE_DEVICES=4,5,6,7 GPUS=4 sh tools/xxx/slurm_train_xxx.sh ${PARTITION} ${JOB_
↪NAME} config2.py tmp_work_dir_2
```

Option 2:

You can set different communication ports without the need to modify the configuration file, but have to set the `cfg-options` to overwrite the default port in configuration file.

```
CUDA_VISIBLE_DEVICES=0,1,2,3 GPUS=4 sh tools/xxx/slurm_train_xxx.sh ${PARTITION} ${JOB_
↪NAME} config1.py tmp_work_dir_1 --cfg-options dist_params.port=29500
CUDA_VISIBLE_DEVICES=4,5,6,7 GPUS=4 sh tools/xxx/slurm_train_xxx.sh ${PARTITION} ${JOB_
↪NAME} config2.py tmp_work_dir_2 --cfg-options dist_params.port=29501
```

CHAPTER

SIX

TEST A MODEL

6.1 NAS

To test nas method, you can use following command

```
python tools/${task}/test_${task}.py ${CONFIG_FILE} ${CHECKPOINT_PATH} --cfg-options␣
↪algorithm.mutable_cfg=${MUTABLE_CFG_PATH} [optional arguments]
```

- `task`: one of `mmclsmmdet` and `mmseg`
- `MUTABLE_CFG_PATH`: Path of `mutable_cfg`. `mutable_cfg` represents **config for mutable of the subnet searched out**, used to specify different subnets for testing. An example for `mutable_cfg` can be found here.

The usage of optional arguments are the same as corresponding tasks like mmclassification, mmdetection and mmsegmentation.

For example,

6.2 Pruning

6.2.1 Split Checkpoint(Optional)

If you train a slimmable model during retraining, checkpoints of different subnets are actually fused in only one checkpoint. You can split this checkpoint to multiple independent checkpoints by using the following command

```
python tools/model_converters/split_checkpoint.py ${CONFIG_FILE} ${CHECKPOINT_PATH} --
↪channel-cfgs ${CHANNEL_CFG_PATH} [optional arguments]
```

- `CHANNEL_CFG_PATH`: A list of paths of `channel_cfg`. For example, when you retrain a slimmable model, your command will be like `--cfg-options algorithm.channel_cfg=cfg1,cfg2,cfg3`. And your command here should be `--channel-cfgs cfg1 cfg2 cfg3`. The order of them should be the same.

For example,

6.2.2 Test

To test pruning method, you can use following command

```
python tools/${task}/test_${task}.py ${CONFIG_FILE} ${CHECKPOINT_PATH} --cfg-options␣
↪algorithm.channel_cfg=${CHANNEL_CFG_PATH} [optional arguments]
```

- `task`: one of `mmcls`, `mmdet` and `mmseg`
- `CHANNEL_CFG_PATH`: Path of `channel_cfg`. `channel_cfg` represents **config for channel of the subnet searched out**, used to specify different subnets for testing. An example for `channel_cfg` can be found here, and the usage can be found here.

For example,

6.3 Distillation

To test distillation method, you can use the following command

```
python tools/${task}/test_${task}.py ${CONFIG_FILE} ${CHECKPOINT_PATH} [optional␣
↪arguments]
```

- `task`: one of `mmcls`, `mmdet` and `mmseg`

For example,

CHAPTER

SEVEN

TUTORIAL 1: OVERVIEW

MMRazor is a model compression toolkit for model slimming and AutoML, which includes 3 mainstream technologies:

- Neural Architecture Search (NAS)
- Pruning
- Knowledge Distillation (KD)
- Quantization (in the next release)

It is a part of the OpenMMLab project.

7.1 Major features:

- **Compatibility**

  MMRazor can be easily applied to various projects in OpenMMLab, due to the similar architecture design of OpenMMLab as well as the decoupling of slimming algorithms and vision tasks.

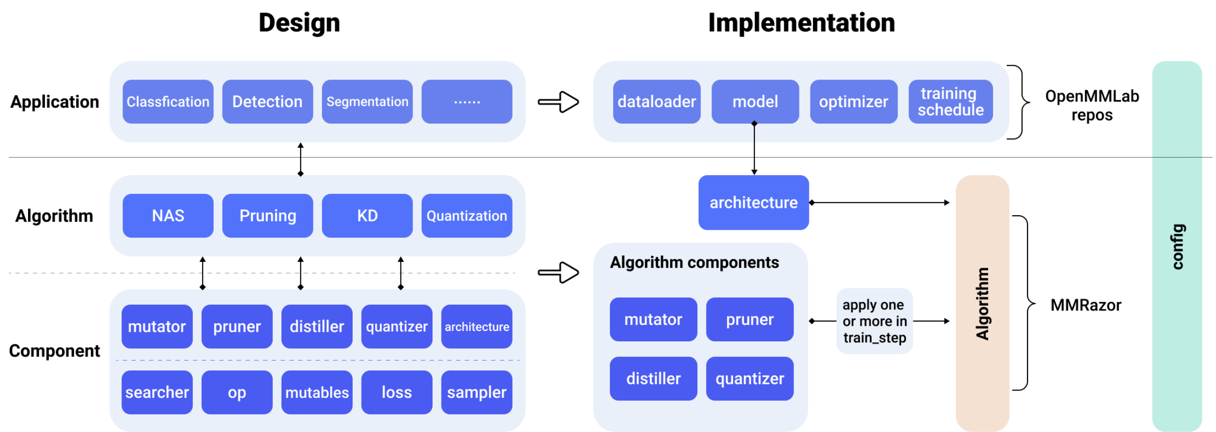
- **Flexibility**

  Different algorithms, e.g., NAS, pruning and KD, can be incorporated in a plug-n-play manner to build a more powerful system.

- **Convenience**

  With better modular design, developers can implement new model compression algorithms with only a few codes, or even by simply modifying config files.

7.2 Design and implement



In terms of the overall design, MMRazor mainly includes Component and Algorithm.

Component can be divided into basic component and algorithm component. The basic component consists of searcher, OP, Mutables and other modules in the figure, which provides basic function support for algorithm component. Algorithm component consists of Mutator, Pruner, Distiller and other modules in the figure. They provide core functionality for implementing various lightweight algorithms. The combination of Algorithm and Application can realize the purpose of slimming various task models.

In terms of implementation, MMRazor's algorithm mainly contains two parts, namely architecture and algorithm components.

Architecture is similar to a model wrapper and can be easily combined with other OpenMMLab repos.

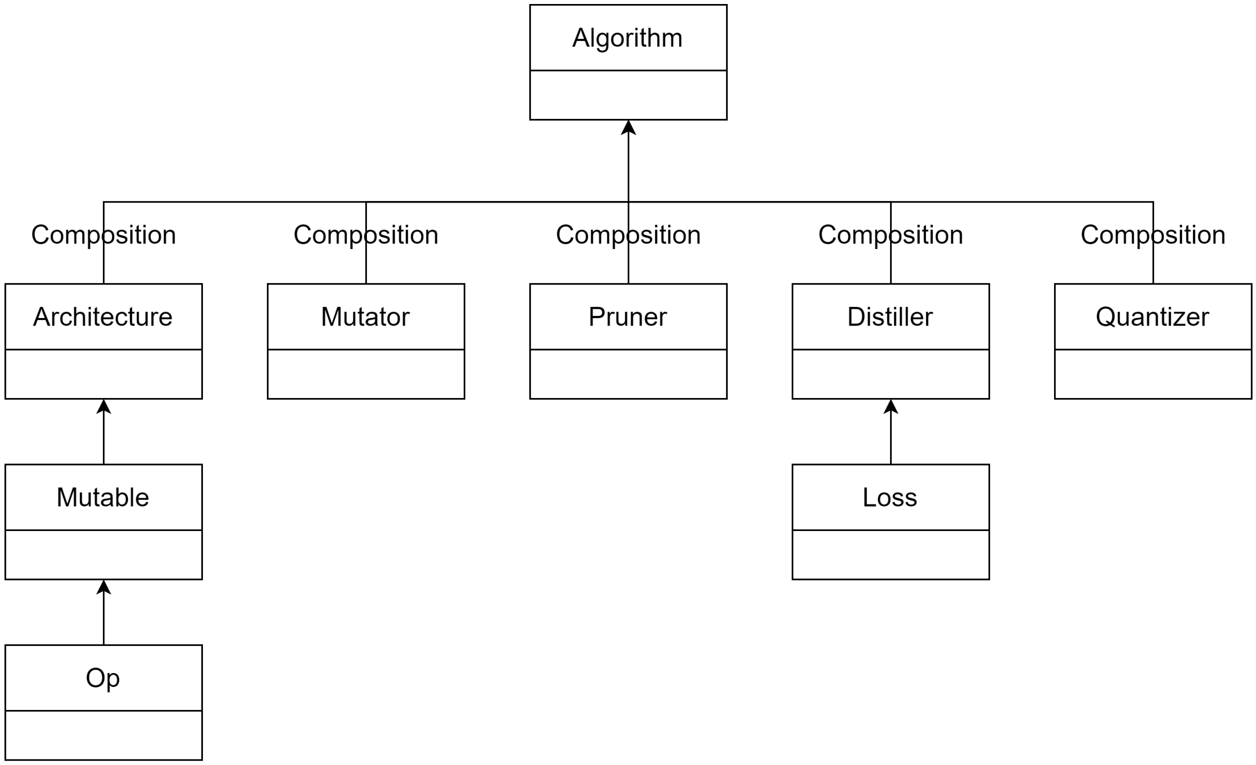
Algorithm components can be flexibly called by each lightweight algorithm. Thanks to the componentization of the algorithm, it can be used alone or in combination in the specific lightweight algorithm. It only needs to rewrite train_step to realize the specific call logic of algorithm components.

The overall style of MMRazor is same as OpenMMLab . Both of them can flexibly configure the experiment through Config. Config mainly includes two parts, one is specific to MMRazor algorithm, and the other part can reuse the experimental configuration of other OpenMMLab repos. Including model definition, data processing, training schedule, etc.

Thanks to OpenMMLab powerful and highly flexible config mode and registry mechanism, MMRazor can perform ablation experiments by editing configuration files without changing the code.

7.3 Key Concepts

MMRazor consists of 4 main parts: 1) apis, 2) core, 3) models, 4) datasets. models is the most vital part, whose structure chart is as follows.



- **Algorithm**: Specific implemented algorithm based on specific task, eg: SPOS (Single Path One-Shot Neural Architecture Search with Uniform Sampling) , applying one-shot NAS to classification. Algorithm consists of two main parts: architecture and algorithm components.
- **Architecture**: Model to be slimmed. There are different roles in different algorithm tasks, eg: architecture is supernet in NAS, also student net in KD, and float model in quantization.
- **Algorithm components**: Core functions providers of 4 lightweight algorithms. In each lightweight algorithm, there are serval classes to handle different types.
  - **Mutator**: Core functions provider of different types of NAS, mainly include some functions of changing the structure of architecture.
  - **Pruner**: Core functions provider of different types of pruning, mainly includes some functions of changing the structure of architecture and getting channel group.
  - **Distiller**: Core functions provider of different types of KD, mainly includes functions of registering some forward hooks, calculate the kd-loss and so on.
  - **Quantizer**: Core functions provider of different types of quantization. It will come soon.
- **Base components**: Core components of architecture and some algorithm components
  - **Op**: Specific operation for building search space in NAS, eg: ShuffleBlock 3*3
  - **Mutable**: Searchable op for building searchable architecture in NAS. It mainly consists of op and mask, and achieving searchable function by handling mask.
  - **Loss**: kd-loss for distiller.

CHAPTER

EIGHT

TUTORIAL 2: LEARN ABOUT CONFIGS

We use python files as our config system. You can find all the provided configs under `$MMRazor/configs`.

8.1 Config Name Style

We follow the below convention to name config files. Contributors are advised to follow the same style. The config file names are divided into four parts: algorithm info, module information, training information and data information. Logically, different parts are concatenated by underscores.

`'_'`, and words in the same part are concatenated by dashes `'-'`.

```
{algorithm info}_{model info}_[experiment setting]_{training info}_{data info}.py
```

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

- `algorithm info`: algorithm information, algorithm name, such as spos, autoslim, cwd, etc.;
- `model info`: model information, model name to be slimmed, such as shufflenet, faster rcnn, etc;
- `experiment setting`: optional, it is used to describe important information about algorithm or model, such as there are 3 stages in spos: pre-training supernet, search, retrain subnet, you can use it to specify which stage, you also can use it to specify teacher network and student network in KD;
- `training info`: Training information, some training schedule, including batch size, lr schedule, data augment and the like;
- `data info`: Data information, dataset name, input size and so on, such as imagenet, cifar, etc.

8.2 Config System

Same as MMDetection, we incorporate modular and inheritance design into our config system, which is convenient to conduct various experiments.

To help the users have a basic idea of a complete config and the modules in a generation system, we make brief comments on the configs of some examples as the following. For more detailed usage and the corresponding alternative for each module, please refer to the API documentation.

8.2.1 An example of NAS - spos

```
_base_ = [
    '../../_base_/datasets/mmcls/imagenet_bs128_colorjittor.py',     # data
    '../../_base_/schedules/mmcls/imagenet_bs1024_spos.py',   # training schedule
    '../../_base_/mmcls_runtime.py'                           # runtime setting
]

# need to specify some parameters baesd on _base_ by rewriting
evaluation = dict(interval=1000, metric='accuracy')
checkpoint_config = dict(interval=10000)
find_unused_parameters = True

# model settings
norm_cfg = dict(type='BN')
model = dict(
    type='mmcls.ImageClassifier',                    # Classifier name
    backbone=dict(
        type='SearchableShuffleNetV2',               # Backbones name
        widen_factor=1.0,
        norm_cfg=norm_cfg),
    neck=dict(type='GlobalAveragePooling'),          # neck network name
    head=dict(
        type='LinearClsHead',                        # linear classification head
        num_classes=1000,                            # The number of output categories,␣
↪consistent with the number of categories in the dataset
        in_channels=1024,                            # The number of input channels,␣
↪consistent with the output channel of the neck
        loss=dict(                                   # Loss function configuration␣
↪information
            type='LabelSmoothLoss',
            num_classes=1000,
            label_smooth_val=0.1,
            mode='original',
            loss_weight=1.0),
        topk=(1, 5),                                  # Evaluation index, Top-k accuracy␣
↪rate, here is the accuracy rate of top1 and top5
    ),
)

# mutator settings
mutator = dict(
    type='OneShotMutator',                            # mutator name registered
    placeholder_mapping=dict(                         # specify mapping dict for␣
↪placeholders in the architecture
        all_blocks=dict(                              # key: placeholder block name␣
↪according to the architecture; value: specify mutable to replace the placeholder
            type='OneShotOP',                         # mutable name registered
            choices=dict(
                shuffle_3x3=dict(
                    type='ShuffleBlock', kernel_size=3, norm_cfg=norm_cfg),
                shuffle_5x5=dict(
                    type='ShuffleBlock', kernel_size=5, norm_cfg=norm_cfg),
```

(continues on next page)

(continued from previous page)

```
                shuffle_7x7=dict(
                    type='ShuffleBlock', kernel_size=7, norm_cfg=norm_cfg),
                shuffle_xception=dict(
                    type='ShuffleXception', norm_cfg=norm_cfg),
            ))))

# algorithm settings
algorithm = dict(
    type='SPOS',                                      # algorithm name registered
    architecture=dict(                                # architecture setting
        type='MMClsArchitecture',                     # architecture name registered
        model=model,                                  # specify defined model to use in␣
↪the architecture
    ),
    mutator=mutator,                                  # specify defined mutator to use␣
↪in the algorithm
    distiller=None,                                   # specify the distiller in the␣
↪algorithm, default None
    retraining=False,                                 # Bool, specify which stage in␣
↪the algorithm. True: sunet retrain; False: pre-training supernet
)
```

8.2.2 An example of KD - cwd

```
_base_ = [
    '../../_base_/datasets/mmseg/cityscapes.py',       # data
    '../../_base_/schedules/mmseg/schedule_80k.py',    # training schedule
    '../../_base_/mmseg_runtime.py'                    # runtime setting
]

# specify norm_cfg for teacher and student as follows
norm_cfg = dict(type='SyncBN', requires_grad=True)

# pspnet r18 as student network, for more detailed usage, please refer to MMSegmentation
↪'s docs
student = dict(
    type='mmseg.EncoderDecoder',
    backbone=dict(
        type='ResNetV1c',
        init_cfg=dict(
            type='Pretrained', checkpoint='open-mmlab://resnet18_v1c'),
        depth=18,
        num_stages=4,
        out_indices=(0, 1, 2, 3),
        dilations=(1, 1, 2, 4),
        strides=(1, 2, 1, 1),
        norm_cfg=norm_cfg,
        norm_eval=False,
        style='pytorch',
        contract_dilation=True),
```

(continues on next page)

(continued from previous page)

```
    decode_head=dict(
        type='PSPHead',
        in_channels=512,
        in_index=3,
        channels=128,
        pool_scales=(1, 2, 3, 6),
        dropout_ratio=0.1,
        num_classes=19,
        norm_cfg=norm_cfg,
        align_corners=False,
        loss_decode=dict(
            type='CrossEntropyLoss', use_sigmoid=False, loss_weight=1.0)),
    auxiliary_head=dict(
        type='FCNHead',
        in_channels=256,
        in_index=2,
        channels=64,
        num_convs=1,
        concat_input=False,
        dropout_ratio=0.1,
        num_classes=19,
        norm_cfg=norm_cfg,
        align_corners=False,
        loss_decode=dict(
            type='CrossEntropyLoss', use_sigmoid=False, loss_weight=0.4)),
    train_cfg=dict(),
    test_cfg=dict(mode='whole'))

# pspnet r101 as teacher network, for more detailed usage, please refer to MMSegmentation
↪'s docs
teacher = dict(
    type='mmseg.EncoderDecoder',
    backbone=dict(
        type='ResNetV1c',
        depth=101,
        num_stages=4,
        out_indices=(0, 1, 2, 3),
        dilations=(1, 1, 2, 4),
        strides=(1, 2, 1, 1),
        norm_cfg=norm_cfg,
        norm_eval=False,
        style='pytorch',
        contract_dilation=True),
    decode_head=dict(
        type='PSPHead',
        in_channels=2048,
        in_index=3,
        channels=512,
        pool_scales=(1, 2, 3, 6),
        dropout_ratio=0.1,
        num_classes=19,
        norm_cfg=norm_cfg,
```

(continues on next page)

(continued from previous page)

```
        align_corners=False,
        loss_decode=dict(
            type='CrossEntropyLoss', use_sigmoid=False, loss_weight=1.0)),
)

# distiller settings
distiller=dict(
    type='SingleTeacherDistiller',                  # distiller name registered
    teacher=teacher,                                # specify defined teacher to use in␣
↪the distiller
    teacher_trainable=False,                        # whether to train teacher
    components=[                                    # specify what moudules to␣
↪calculate kd-loss in teacher and student
        dict(
            student_module='decode_head.conv_seg',  # student module name
            teacher_module='decode_head.conv_seg',  # teacher module name
            losses=[                                # specify kd-loss
                dict(
                    type='ChannelWiseDivergence',   # kd-loss type
                    name='loss_cwd_logits',         # name this loss in order to easy␣
↪get the output of this loss
                    tau=5,                          # temperature coefficient
                    loss_weight=3,                       # weight of this loss
                )
            ])
    ]),


# algorithm settings
algorithm = dict(
    type='Distillation',                               # algorithm name registered
    architecture=dict(                                 # architecture setting
        type='MMSegArchitecture',                      # architecture name registered
        model=student,                                 # specify defined student as the␣
↪model of architecture
    ),
    use_gt=True,                                       # whether to calculate gt_loss␣
↪with gt
    distiller=distiller,                               # specify defined distiller to␣
↪use in the algorithm
)
```

8.2.3 An example of pruning - autoslim

```
_base_ = [
    '../../_base_/datasets/mmcls/imagenet_bs256_autoslim.py',   # data
    '../../_base_/schedules/mmcls/imagenet_bs2048_autoslim.py', # training schedule
    '../../_base_/mmcls_runtime.py'                             # runtime setting
]

# need to specify some parameters baesd on _base_ by rewriting
runner = dict(type='EpochBasedRunner', max_epochs=50)

# model settings
model = dict(
    type='mmcls.ImageClassifier',                             # Classifier name
    backbone=dict(type='MobileNetV2', widen_factor=1.5),      # Backbones name
    neck=dict(type='GlobalAveragePooling'),                   # neck network name
    head=dict(
        type='LinearClsHead',                                 # linear classification head
        num_classes=1000,                            # The number of output categories,␣
↪consistent with the number of categories in the dataset
        in_channels=1920,                            # The number of input channels,␣
↪consistent with the output channel of the neck
        loss=dict(                                   # Loss function configuration␣
↪information
            type='CrossEntropyLoss',
            loss_weight=1.0),
        topk=(1, 5),                                 # Evaluation index, Top-k accuracy␣
↪rate, here is the accuracy rate of top1 and top5
    ))

# distiller settings, for more details, please refer to the previous section: an example␣
↪of KD - cwd
distiller = dict(
    type='SelfDistiller',
    components=[
        dict(
            student_module='head.fc',
            teacher_module='head.fc',
            losses=[
                dict(
                    type='KLDivergence',
                    name='loss_kd',
                    tau=1,
                    loss_weight=1,
                )
            ]),
    ])

# pruner settings
pruner=dict(
    type='RatioPruner',                          # pruner name registered
    ratios=(2 / 12, 3 / 12, 4 / 12, 5 / 12,      # list, specify the ratio range of␣
↪random sampling
```

(continues on next page)

(continued from previous page)

```
            6 / 12, 7 / 12, 8 / 12, 9 / 12,
            10 / 12, 11 / 12, 1.0))

# algorithm settings
algorithm = dict(
    type='AutoSlim',                                # algorithm name registered
    architecture=dict(                              # architecture setting
        type='MMClsArchitecture',                   # architecture name registered
        model=model),                               # specify defined model to use in the␣
↪architecture
    distiller=distiller,                            # specify defined distiller to use in␣
↪the algorithm
    pruner=pruner,                                  # specify defined pruner to use in the␣
↪algorithm
    retraining=False,                               # Bool, specify which stage in the␣
↪algorithm. True: sunet retrain; False: pre-training supernet
    bn_training_mode=True,                          # set bn to training mode when model is␣
↪set to eval mode
    input_shape=None)                               # setting input_shape for getting subnet␣
↪flops

use_ddp_wrapper = True                              # bool, for updating optimizer in train_
↪step to avoid error
```

CHAPTER

NINE

TOTURIAL 3: CUSTOMIZE ARCHITECTURES

Different from other tasks, architecture may consist of some searchable model components in NAS. In MMRazor, you can not only develop some common model components like other codebases of OpenMMLab, but also develop some searchable model components. Here is how to develop searchable model components and common model components.

9.1 Develop searchable model components

Here we show how to add a new searchable backbone with an example of searchable_shufflenet_v2.

1. Define a new backbone

   Create a new file `mmrazor/models/architectures/components/backbones/searchable_shufflenet_v2.py`, class `SearchableShuffleNetV2` inherits from `BaseBackBone` of mmcls, which is the codebase that you will to build the model.

```python
import torch.nn as nn
from mmcls.models.backbones.base_backbone import BaseBackbone
from mmcls.models.builder import BACKBONES

@BACKBONES.register_module()
class SearchableShuffleNetV2(BaseBackbone):

    def __init__(self, ):
        pass

    def _make_layer(self, out_channels, num_blocks, stage_idx):
        pass

    def _freeze_stages(self):
        pass

    def init_weights(self):
        pass

    def forward(self, x):
        pass

    def train(self, mode=True):
        pass
```

2. Replace layers with placeholders

```python
from ...utils import Placeholder

@BACKBONES.register_module()
class SearchableShuffleNetV2(BaseBackbone):

    ...

    def _make_layer(self, out_channels, num_blocks, stage_idx):
        """Stack blocks to make a layer.

        Args:
            out_channels (int): out_channels of the block.
            num_blocks (int): number of blocks.
        """
        layers = []
        for i in range(num_blocks):
            stride = 2 if i == 0 else 1
            layers.append(
                Placeholder(
                    group='all_blocks',
                    space_id=f'stage_{stage_idx}_block_{i}',
                    in_channels=self.in_channels,
                    out_channels=out_channels,
                    stride=stride,
                    conv_cfg=self.conv_cfg,
                    norm_cfg=self.norm_cfg,
                    act_cfg=self.act_cfg,
                    with_cp=self.with_cp,
                    init_cfg=self.init_cfg))
            self.in_channels = out_channels

        return nn.Sequential(*layers)

    ...
```

3. Import the module

   You can either add the following line to `mmrazor/models/architectures/components/backbones/__init__.py`

```python
from .searchable_shufflenet_v2 import SearchableShuffleNetV2
```

   or alternatively add

```python
custom_imports = dict(
    imports=['mmrazor.models.architectures.components.backbones.searchable_
↪shufflenet_v2'],
    allow_failed_imports=False)
```

   to the config file to avoid modifying the original code.

4. Use the backbone in your config file

```
architecture = dict(
    type=xxx,
    model=dict(
        ...
        backbone=dict(
            type='mmcls.SearchableShuffleNetV2',
            arg1=xxx,
            arg2=xxx),
        ...
```

9.2 Develop common model components

Here we show how to add a new backbone with an example of xxxNet.

1. Define a new backbone

   Create a new file `mmrazor/models/architectures/components/backbones/xxxnet.py`.

```
import torch.nn as nn
from ..builder import BACKBONES

@BACKBONES.register_module()
class xxxNet(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 `mmrazor/models/architectures/components/backbones/__init__.py`

```
from .xxxnet import xxxNet
```

   or alternatively add

```
custom_imports = dict(
    imports=['mmrazor.models.architectures.components.backbones.xxxnet'],
    allow_failed_imports=False)
```

   to the config file to avoid modifying the original code.

3. Use the backbone in your config file

```
architecture = dict(
    type=xxx,
    model=dict(
        ...
        backbone=dict(
            type='xxxNet',
```

(continues on next page)

(continued from previous page)

```
            arg1=xxx,
            arg2=xxx),
        ...
```

How to add other model components is similar to backbone's. For more details, please refer to other codebases' docs.

CHAPTER

TEN

TOTURIAL 4: CUSTOMIZE NAS ALGORITHMS

Here we show how to develop new NAS algorithms with an example of SPOS.

1. Register a new algorithm

   Create a new file mmrazor/models/algorithms/spos.py, class SPOS inherits from class BaseAlgorithm

```
from mmrazor.models.builder import ALGORITHMS
from .base import BaseAlgorithm

@ALGORITHMS.register_module()
class SPOS(BaseAlgorithm):
    def __init__(self, **kwargs):
        super(SPOS, self).__init__(**kwargs)
        pass

    def train_step(self, data, optimizer):
        pass
```

2. Develop new algorithm components (optional)

   SPOS can directly use class OneShotMutator as core functions provider. If mutators provided in MMRazor don't meet your needs, you can develop new algorithm components for your algorithm like OneShotMutator, we will take OneShotMutator as an example to introduce how to develop a new algorithm component:

   a. Create a new file mmrazor/models/mutators/one_shot_mutator.py, class OneShotMutator inherits from class BaseMutator

   b. Finish the functions you need in OneShotMutator, eg: sample_subnet, set_subnet and so on

```
from mmrazor.models.builder import MUTATORS
from .base import BaseMutator


@MUTATORS.register_module()
class OneShotMutator(BaseMutator):

    def __init__(self, **kwargs):
        super().__init__(**kwargs)

    @staticmethod
    def get_random_mask(space_info):
        pass

```

(continues on next page)

(continued from previous page)

```
    def sample_subnet(self):
        pass

    def set_subnet(self, subnet_dict):
        pass

    @staticmethod
    def reset_in_subnet(m, in_subnet=True):
        pass

    def set_chosen_subnet(self, subnet_dict):
        pass

    def mutation(self, subnet_dict, prob=0.1):
        pass

    @staticmethod
    def crossover(subnet_dict1, subnet_dict2):
        pass
```

c. Import the new mutator

You can either add the following line to mmrazor/models/mutators/__init__.py

```
from .one_shot_mutator import OneShotMutator
```

or alternatively add

```
custom_imports = dict(
    imports=['mmrazor.models.mutators.one_shot_mutator'],
    allow_failed_imports=False)
```

to the config file to avoid modifying the original code.

d. Use the algorithm component in your config file

```
mutator = dict(
    type='OneShotMutator',
    ...)
```

3. Rewrite its train_step

   Develop key logic of your algorithm in functiontrain_step

```
@ALGORITHMS.register_module()
class SPOS(BaseAlgorithm):
    def __init__(self, **kwargs):
        super(SPOS, self).__init__(**kwargs)
        pass

    def train_step(self, data, optimizer):
        if self.retraining:
            outputs = super(SPOS, self).train_step(data, optimizer)
        else:
```

(continues on next page)

(continued from previous page)

```
            subnet_dict = self.mutator.sample_subnet()
            self.mutator.set_subnet(subnet_dict)
            outputs = super(SPOS, self).train_step(data, optimizer)
        return outputs
```

4. Add your custom functions (optional)

   After finishing your key logic in function `train_step`, if you also need other custom functions, you can add them in class SPOS as follows.

```
@ALGORITHMS.register_module()
class SPOS(BaseAlgorithm):
    def __init__(self, **kwargs):
        super(SPOS, self).__init__(**kwargs)
        pass

    def _init_flops(self):
        pass

    def get_subnet_flops(self):
        pass

    def train_step(self, data, optimizer):
        if self.retraining:
            outputs = super(SPOS, self).train_step(data, optimizer)
        else:
            subnet_dict = self.mutator.sample_subnet()
            self.mutator.set_subnet(subnet_dict)
            outputs = super(SPOS, self).train_step(data, optimizer)
        return outputs
```

5. Import the class

   You can either add the following line to `mmrazor/models/algorithms/__init__.py`

```
from .spos import SPOS
```

   or alternatively add

```
custom_imports = dict(
    imports=['mmrazor.models.algorithms.spos'],
    allow_failed_imports=False)
```

   to the config file to avoid modifying the original code.

6. Use the algorithm in your config file

```
algorithm = dict(
    type='SPOS',
    mutator=mutator,    # you can use it here if you developed new algorithm␣
↪components
    ...
)
```

CHAPTER

ELEVEN

TUTORIAL 5: CUSTOMIZE PRUNING ALGORITHMS

Here we show how to develop new Pruning algorithms with an example of AutoSlim.

1. Register a new algorithm

   Create a new file `mmrazor/models/algorithms/autoslim.py`, class `AutoSlim` inherits from class `BaseAlgorithm`

```
from mmrazor.models.builder import ALGORITHMS
from .base import BaseAlgorithm

@ALGORITHMS.register_module()
class AutoSlim(BaseAlgorithm):
    def __init__(self,
                 num_sample_training=4,
                 input_shape=(3, 224, 224),
                 bn_training_mode=False,
                 **kwargs):
        super(AutoSlim, self).__init__(**kwargs)
        pass

    def train_step(self, data, optimizer):
        pass
```

2. Develop new algorithm components (optional)

   AutoSlim can directly use class `RatioPruner` as core functions provider. If pruners provided in MMRazor don't meet your needs, you can develop new algorithm components for your algorithm like `RatioPruner`. We will take `RatioPruner` as an example to introduce how to develop a new algorithm component:

   a. Create a new file `mmrazor/models/pruners/ratio_pruning.py`, class `RatioPruner` can inherits from StructurePruner

   b. Finish the functions you need, eg: `sample_subnet`, `set_subnet` and so on

```
from mmrazor.models.builder import PRUNERS, build_mutable
from .structure_pruning import StructurePruner

@PRUNERS.register_module()
class RatioPruner(StructurePruner):
    def __init__(self, **kwargs):
        super().__init__(**kwargs)

    def sample_subnet(self):
```

(continues on next page)

(continued from previous page)

```python
        pass

    def set_subnet(self, subnet_dict):
        pass

    ....

    # supernet is a kind of architecture in `mmrazor/models/architectures/`
    def prepare_from_supernet(self, supernet):
        pass
```

c. Import the module in `mmrazor/models/pruners/__init__.py`

```python
from .ratio_pruning import RatioPruner

__all__ = [..., 'RatioPruner']
```

3. Rewrite its train_step

   Develop key logic of your algorithm in function`train_step`

```python
from mmrazor.models.builder import ALGORITHMS
from .base import BaseAlgorithm

@ALGORITHMS.register_module()
class AutoSlim(BaseAlgorithm):
    def __init__(self,
                 num_sample_training=4,
                 input_shape=(3, 224, 224),
                 bn_training_mode=False,
                 **kwargs):
        super(AutoSlim, self).__init__(**kwargs)
        pass

    def train_step(self, data, optimizer):
        optimizer.zero_grad()
        losses = dict()
        if not self.retraining:
            ...
        else:
            ...
        optimizer.step()
        loss, log_vars = self._parse_losses(losses)
        outputs = dict(
            loss=loss, log_vars=log_vars, num_samples=len(data['img'].data))
        return outputs
```

4. Add your custom functions (optional)

   After finishing your key logic in function train_step, if you also need other custom functions, you can add them in class `AutoSlim`.

5. Import the class

   You can either add the following line to `mmrazor/models/algorithms/__init__.py`

```
from .autoslim import AutoSlim

__all__ = [..., 'AutoSlim']
```

Or alternatively add

```
custom_imports = dict(
    imports=['mmrazor.models.algorithms.spos'],
    allow_failed_imports=False)
```

to the config file to avoid modifying the original code.

6. Use the algorithm in your config file

```
algorithm = dict(
    type='AutoSlim',
    architecture=...,
    pruner=dict(type='RatioPruner', ...),
    retraining=...)
```

CHAPTER

TWELVE

TOTURIAL 6: CUSTOMIZE KD ALGORITHMS

Here we show how to develop new KD algorithms with an example of cwd.

1. Register a new algorithm

   Create a new file `mmrazor/models/algorithms/kd.py`, class `Distillation` inherits from class `BaseAlgorithm`

```
from mmrazor.models.builder import ALGORITHMS
from .base import BaseAlgorithm

@ALGORITHMS.register_module()
class Distillation(BaseAlgorithm):
    def __init__(self, use_gt, **kwargs):
        super(Distillation, self).__init__(**kwargs)
        self.use_gt = use_gt
        pass

    def train_step(self, data, optimizer):
        pass
```

2. Develop new algorithm components (optional)

   Distillation can directly use class `SingleTeacherDistiller` or other distillers in `mmrazor/models/distillers/` as core functions provider. If distillers provided in MMRazor don't meet your needs, you can develop new algorithm components for your algorithm as follows:

   a. Create a new file `mmrazor/models/distillers/multi_teachers.py`, class `MultiTeachersDistiller` inherits from class `SingleTeacherDistiller`

   b. Finish the functions you need, eg: `build_teacher`, `compute_distill_loss` and so on

```
from mmrazor.models.builder import DISTILLERS
from .single_teacher import SingleTeacherDistiller

@DISTILLERS.register_module()
class MultiTeachersDistiller(SingleTeacherDistiller):
    def __init__(self, teacher, teacher_ckpt, teacher_trainable, **kwargs):
        super(MultiTeachersDistiller,
            self).__init__(teacher, teacher_trainable, **kwargs)

    def build_teacher(self, cfgs, ckpts):
        pass
```

(continues on next page)

(continued from previous page)

```
    def exec_teacher_forward(self, data, return_output):
        pass

    def compute_distill_loss(self):
        pass
```

c. Import the module in `mmrazor/models/distillers/__init__.py`

```
from .multi_teachers import MultiTeachersDistiller

__all__ = [..., 'MultiTeachersDistiller']
```

3. Rewrite its train_step

   Develop key logic of your algorithm in function`train_step`

```
from mmrazor.models.builder import ALGORITHMS
from .base import BaseAlgorithm

@ALGORITHMS.register_module()
class Distillation(BaseAlgorithm):
    def __init__(self, use_gt, **kwargs):
        super(Distillation, self).__init__(**kwargs)
        self.use_gt = use_gt
        pass

    def train_step(self, data, optimizer):
        losses = dict()
        if self.use_gt:
            _ = self.distiller.exec_teacher_forward(data)
            gt_losses = self.distiller.exec_student_forward(
                self.architecture, data)
            distill_losses = self.distiller.compute_distill_loss()
            losses.update(gt_losses)
            losses.update(distill_losses)
        else:
            _ = self.distiller.exec_teacher_forward(data)
            _ = self.distiller.exec_student_forward(self.architecture, data)
            distill_losses = self.distiller.compute_distill_loss()
            losses.update(distill_losses)

        loss, log_vars = self._parse_losses(losses)
        outputs = dict(
            loss=loss, log_vars=log_vars, num_samples=len(data['img'].data))
        return outputs
```

4. Add your custom functions (optional)

   After finishing your key logic in function `train_step`, if you also need other custom functions, you can add them in class `Distillation`

5. Import the class

   You can either add the following line to `mmrazor/models/algorithms/__init__.py`

```python
from .kd import Distillation

__all__ = [..., 'Distillation']
```

or alternatively add

```python
custom_imports = dict(
    imports=['mmrazor.models.algorithms.spos'],
    allow_failed_imports=False)
```

to the config file to avoid modifying the original code.

6. Use the algorithm in your config file

```python
algorithm = dict(
    type='Distillation',
    distiller=dict(type='MultiTeachersDistiller', ...),  # you can also use your␣
↪new algorithm components here
    ...
)
```

CHAPTER

THIRTEEN

TUTORIAL 7: CUSTOMIZE MIXED ALGORITHMS WITH OUR ALGORITHM COMPONENTS

Here we show how to customize mixed algorithms with our algorithm components. We take the slimmable training in autoslim as an example.

The sandwich rule and inplace distillation were introduced to enhance the training process. The sandwich rule means that we train the model at smallest width, largest width and (n 2) random widths, instead of n random widths. By inplace distillation, we use the predicted label of the model at the largest width as the training label for other widths, while for the largest width we use ground truth. So both the KD algorithm and the pruning algorithm are used in slimmable training.

1. In the distillation part, we can directly use SelfDistiller in `mmrazor/models/distillers/self_distiller.py`. If distillers provided in MMRazor don't meet your needs, you can develop new algorithm components for your algorithm as step2 in Tutorial 6.
2. As the slimmable training is the first step of `Autoslim`, we do not need to register a new algorithm, but rewrite the `train_step`function in AutoSlim as follows:

```
from mmrazor.models.builder import ALGORITHMS
from .base import BaseAlgorithm

@ALGORITHMS.register_module()
class AutoSlim(BaseAlgorithm):
    def train_step(self, data, optimizer):
        optimizer.zero_grad()
        losses = dict()
        if not self.retraining:
            #
        else:
            ...
        optimizer.step()
        loss, log_vars = self._parse_losses(losses)
        outputs = dict(
            loss=loss, log_vars=log_vars, num_samples=len(data['img'].data))
        return outputs
```

3. Use the algorithm in your config file

```
algorithm = dict(
    type='AutoSlim',
    architecture=...,
    pruner=dict(
        type='RatioPruner',
```

(continues on next page)

(continued from previous page)

```
        ratios=(2 / 12, 3 / 12, 4 / 12, 5 / 12, 6 / 12, 7 / 12, 8 / 12, 9 / 12,
                10 / 12, 11 / 12, 1.0)),
    distiller=dict(
        type='SelfDistiller',
        components=...),
    retraining=False)
```

CHAPTER

FOURTEEN

TUTORIAL 8: APPLY EXISTING ALGORITHMS TO NEW TASKS

Here we show how to apply existing algorithms to other existing tasks with an example of SPOS & DetNAS.

1. Register a new algorithm for the other existing task with an existing algorithm

   Create a new file `mmrazor/models/algorithms/detnas.py`, class `DetNAS` inherits from class `SPOS`

```python
from mmrazor.models.builder import ALGORITHMS
from .spos import SPOS

@ALGORITHMS.register_module()
class DetNAS(SPOS):

    def __init__(self, **kwargs):
        super(DetNAS, self).__init__(**kwargs)
```

2. Add your custom functions (optional)

   If you need other custom functions according to the other existing task, you can add them in class `DetNAS` as follows.

```python
class DetNAS(SPOS):

    ...

    def _init_flops(self):
        flops_model = copy.deepcopy(self.architecture)
        flops_model = revert_sync_batchnorm(flops_model)
        flops_model.eval()
        flops, params = get_model_complexity_info(flops_model.model.backbone,
                                                  self.input_shape)
        flops_lookup = dict()
        for name, module in flops_model.named_modules():
            flops = getattr(module, '__flops__', 0)
            flops_lookup[name] = flops
        del (flops_model)

        for name, module in self.architecture.named_modules():
            module.__flops__ = flops_lookup[name]

    ...
```

3. Import the class

You can either add the following line to mmrazor/models/algorithms/__init__.py

```
from .detnas import DetNAS
```

or alternatively add

```
custom_imports = dict(
    imports=['mmrazor.models.algorithms.detnas'],
    allow_failed_imports=False)
```

to the config file to avoid modifying the original code.

4. Use the algorithm in your config file

```
algorithm = dict(
    type='DetNAS',
    ...
)
```

CHAPTER FIFTEEN

ENGLISH

CHAPTER

SIXTEEN

CHAPTER SEVENTEEN

MMRAZOR.APIS

17.1 mmcls

mmrazor.apis.mmcls.**init_mmcls_model**(*config: Union[str, mmcv.utils.config.Config]*, *checkpoint: Optional[str] = None*, *device: str = 'cuda:0'*, *cfg_options: Optional[Dict] = None*) → torch.nn.modules.module.Module

Initialize a mmcls model from config file.

Parameters

- **config** (str or mmcv.Config) – Config file path or the config object.
- **checkpoint** (*str, optional*) – Checkpoint path. If left as None, the model will not load any weights.
- **cfg_options** (*dict*) – cfg_options to override some settings in the used config.

Returns The constructed classifier.

Return type nn.Module

mmrazor.apis.mmcls.**set_random_seed**(*seed*, *deterministic=False*)

Import *set_random_seed* function here was deprecated in v0.3 and will be removed in v0.5.

Parameters

- **seed** (*int*) – Seed to be used.
- **deterministic** (*bool*) – Whether to set the deterministic option for CUDNN backend, i.e., set `torch.backends.cudnn.deterministic` to True and `torch.backends.cudnn.benchmark` to False. Default: False.

mmrazor.apis.mmcls.**train_mmcls_model**(*model*, *dataset*, *cfg*, *distributed=False*, *validate=False*, *timestamp=None*, *device='cuda'*, *meta=None*)

Copy from mmclassification and modify some codes.

This is an ugly implementation, and will be deprecated in the future. In the future, there will be only one train api and no longer distinguish between mmclassificaiton, mmsegmentation or mmdetection.

17.2 mmdet

17.3 mmseg

CHAPTER

EIGHTEEN

MMRAZOR.CORE

18.1 hooks

class mmrazor.core.hooks.**DistSamplerSeedHook**

Data-loading sampler for distributed training.

When distributed training, it is only useful in conjunction with `EpochBasedRunner`, while :obj:`IterBasedRunner` achieves the same purpose with `IterLoader`.

before_epoch(*runner*)

Executed in before_epoch stage.

class mmrazor.core.hooks.**DropPathProbHook**(*max_prob*, *interval=- 1*, *by_epoch=True*, ***kwargs*)

Set drop_path_prob periodically.

Parameters

- **max_prob** (*float*) – The max probability of dropping.
- **interval** (*int*) – The saving period. If `by_epoch=True`, interval indicates epochs, otherwise it indicates iterations. Default: -1, which means "never".
- **by_epoch** (*bool*) – Saving checkpoints by epoch or by iteration. Default: True.

before_train_epoch(*runner*)

Executed in before_train_epoch stage.

class mmrazor.core.hooks.**SearchSubnetHook**(*interval=- 1*, *by_epoch=True*, *out_dir=None*, *max_keep_ckpts=- 1*, *save_last=True*, ***kwargs*)

Save checkpoints periodically.

Parameters

- **interval** (*int*) – The saving period. If `by_epoch=True`, interval indicates epochs, otherwise it indicates iterations. Default: -1, which means "never".
- **by_epoch** (*bool*) – Saving checkpoints by epoch or by iteration. Default: True.
- **out_dir** (*str, optional*) – The directory to save checkpoints. If not specified, `runner.work_dir` will be used by default.
- **max_keep_ckpts** (*int, optional*) – The maximum checkpoints to keep. In some cases we want only the latest few checkpoints and would like to delete old ones to save the disk space. Default: -1, which means unlimited.
- **save_last** (*bool*) – Whether to force the last checkpoint to be saved regardless of interval.

after_train_epoch(*runner*)

Executed in after_train_epoch stage.

after_train_iter(*runner*)
Executed in after_train_iter stage.

before_run(*runner*)
Executed in before_run stage.

18.2 optimizer

mmrazor.core.optimizer.**build_optimizers**(*model*, *cfgs*)
Build multiple optimizers from configs. If *cfgs* contains several dicts for optimizers, then a dict for each constructed optimizers will be returned. If *cfgs* only contains one optimizer config, the constructed optimizer itself will be returned. For example, 1) Multiple optimizer configs: code-block:

```
optimizer_cfg = dict(
    model1=dict(type='SGD', lr=lr),
    model2=dict(type='SGD', lr=lr))
```

The return dict is `dict('model1': torch.optim.Optimizer, 'model2': torch.optim.Optimizer)` 2) Single optimizer config: .. code-block:

```
optimizer_cfg = dict(type='SGD', lr=lr)
```

The return is `torch.optim.Optimizer`. :param model: The model with parameters to be optimized. :type model: `nn.Module` :param cfgs: The config dict of the optimizer. :type cfgs: dict

Returns The initialized optimizers.

Return type dict[`torch.optim.Optimizer`] | `torch.optim.Optimizer`

18.3 runners

class mmrazor.core.runners.**MultiLoaderEpochBasedRunner**(*model: torch.nn.modules.module.Module*, *batch_processor: Optional[Callable] = None*, *optimizer: Optional[Union[Dict, torch.optim.optimizer.Optimizer]] = None*, *work_dir: Optional[str] = None*, *logger: Optional[logging.Logger] = None*, *meta: Optional[Dict] = None*, *max_iters: Optional[int] = None*, *max_epochs: Optional[int] = None*)
Multi Dataloaders EpochBasedRunner.

There are three differences from EpochBaseRunner 1Support load data from multi dataloaders. 2) Support freeze some optimizer's lr update when runner has multi optimizers. 3) Add `search_subnet` api.

register_lr_hook(*lr_config*)
Resister a hook for setting learning rate.

Parameters **lr_config** (*dict*) – Config for setting learning rate.

search_subnet(*out_dir*, *filename_tmpl='epoch_{}.yaml'*, *create_symlink=True*)
Search the best subnet.

Parameters

- **out_dir** (*str*) – The directory that subnets are saved.
- **filename_tmpl** (*str, optional*) – The subnet filename template, which contains a placeholder for the epoch number. Defaults to 'epoch_{}.yaml'.
- **create_symlink** (*bool, optional*) – Whether to create a symlink "latest.yaml" to point to the latest subnet. Defaults to True.

train(*data_loader*, ***kwargs*)
Rewrite the `train` of `EpochBasedRunner`.

class mmrazor.core.runners.**MultiLoaderIterBasedRunner**(*model: torch.nn.modules.module.Module*, *batch_processor: Optional[Callable] = None*, *optimizer: Optional[Union[Dict, torch.optim.optimizer.Optimizer]] = None*, *work_dir: Optional[str] = None*, *logger: Optional[logging.Logger] = None*, *meta: Optional[Dict] = None*, *max_iters: Optional[int] = None*, *max_epochs: Optional[int] = None*)

Multi Dataloaders IterBasedRunner.

There are three differences from IterBasedRunner 1Support load data from multi dataloaders. 2) Support freeze some optimizer's lr update when runner has multi optimizers. 3) Add `search_subnet` api.

register_lr_hook(*lr_config*)
Resister a hook for setting learning rate.

Parameters **lr_config** (*dict*) – Config for setting learning rate.

run(*data_loaders*, *workflow*, *max_iters=None*, ***kwargs*)
Start running.

Parameters

- **data_loaders** (list[`DataLoader`]) – Dataloaders for training and validation.
- **workflow** (*list[tuple]*) – A list of (phase, iters) to specify the running order and iterations. E.g, [('train', 10000), ('val', 1000)] means running 10000 iterations for training and 1000 iterations for validation, iteratively.
- **max_iters** (*int*) – Specify the max iters.

search_subnet(*out_dir*, *filename_tmpl='epoch_{}.yaml'*, *create_symlink=True*)
Search the best subnet.

Parameters

- **out_dir** (*str*) – The directory that subnets are saved.
- **filename_tmpl** (*str, optional*) – The subnet filename template, which contains a placeholder for the epoch number. Defaults to 'epoch_{}.yaml'.
- **create_symlink** (*bool, optional*) – Whether to create a symlink "latest.yaml" to point to the latest subnet. Defaults to True.

18.4 searcher

class mmrazor.core.searcher.**EvolutionSearcher**(*algorithm*, *dataloader*, *test_fn*, *work_dir*, *logger*, *candidate_pool_size=50*, *candidate_top_k=10*, *constraints={'flops': 330000000.0}*, *metrics=None*, *metric_options=None*, *score_key='accuracy_top-1'*, *max_epoch=20*, *num_mutation=25*, *num_crossover=25*, *mutate_prob=0.1*, *resume_from=None*, ***search_kwargs*)

Implement of evolution search.

Parameters

- **algorithm** (torch.nn.Module) – Algorithm to be used.
- **dataloader** (*nn.Dataloader*) – Pytorch data loader.
- **test_fn** (*function*) – Test api to used for evaluation.
- **work_dir** (*str*) – Working direction is to save search result and log.
- **logger** (*logging.Logger*) – To log info in search stage.
- **candidate_pool_size** (*int*) – The length of candidate pool.
- **candidate_top_k** (*int*) – Specify top k candidates based on scores.
- **constraints** (*dict*) – Constraints to be used for screening candidates.
- **metrics** (*str*) – Metrics to be used for evaluating candidates.
- **metric_options** (*str*) – Options to be used for metrics.
- **score_key** (*str*) – To be used for specifying one metric from evaluation results.
- **max_epoch** (*int*) – Specify max epoch to end evolution search.
- **num_mutation** (*int*) – The number of candidates got by mutation.
- **num_crossover** (*int*) – The number of candidates got by crossover.
- **mutate_prob** (*float*) – The probability of mutation.
- **resume_from** (*str*) – Specify the path of saved .pkl file for resuming searching

check_constraints()

Check whether is beyond constraints.

Returns The result of checking.

Return type bool

search()

Execute the pipeline of evolution search.

update_top_k()

Update top k candidates.

class mmrazor.core.searcher.**GreedySearcher**(*algorithm*, *dataloader*, *target_flops*, *test_fn*, *work_dir*, *logger*, *max_channel_bins*, *min_channel_bins=1*, *metrics='accuracy'*, *metric_options=None*, *score_key='accuracy_top-1'*, *resume_from=None*, ***search_kwargs*)

Search with the greedy algorithm.

We start with the largest model and compare the network accuracy among the architectures where each layer is slimmed by one channel bin. We then greedily slim the layer with minimal performance drop. During the iterative slimming, we obtain optimized channel configurations under different resource constraints. We stop until reaching the strictest constraint (e.g., 200M FLOPs).

Parameters

- **algorithm** (`torch.nn.Module`) – Specific implemented algorithm based specific task in mmRazor, eg: AutoSlim.
- **dataloader** (`torch.nn.Dataloader`) – Pytorch data loader.
- **target_flops** (*list*) – The target flops of the searched models.
- **test_fn** (*callable*) – test a model with samples from a dataloader, and return the test results.
- **work_dir** (*str*) – Output result file.
- **logger** (*logging.Logger*) – To log info in search stage.
- **max_channel_bins** (*int*) – The maximum number of channel bins in each layer. Note that each layer is slimmed by one channel bin.
- **min_channel_bins** (*int*) – The minimum number of channel bins in each layer. Default to 1.
- **metrics** (*str | list[str]*) – Metrics to be evaluated. Default value is `accuracy`
- **metric_options** (*dict, optional*) – Options for calculating metrics. Allowed keys are 'topk', 'thrs' and 'average_mode'. Defaults to None.
- **score_key** (*str*) – The metric to judge the performance of a model. Defaults to *accuracy_top-1*.
- **resume_from** (*str, optional*) – Specify the path of saved .pkl file for resuming searching. Defaults to None.

search()

Greedy Slimming.

18.5 utils

`mmrazor.core.utils.`**broadcast_object_list**(*data: List[Any], src: int = 0, group: Optional[torch._C._distributed_c10d.ProcessGroup] = None*) → None

Broadcasts picklable objects in `object_list` to the whole group. Similar to `broadcast()`, but Python objects can be passed in. Note that all objects in `object_list` must be picklable in order to be broadcasted. .. note:

```
Calling ``broadcast_object_list`` in non-distributed environment does
nothing.
```

Parameters

- **data** (*List[Any]*) – List of input objects to broadcast. Each object must be picklable. Only objects on the `src` rank will be broadcast, but each rank must provide lists of equal sizes.
- **src** (*int*) – Source rank from which to broadcast `object_list`.

- **group** – (ProcessGroup, optional): The process group to work on. If None, the default process group will be used. Default is `None`.
- **device** (`torch.device`, optional) – If not None, the objects are serialized and converted to tensors which are moved to the `device` before broadcasting. Default is `None`.

Note: For NCCL-based process groups, internal tensor representations of objects must be moved to the GPU device before communication starts. In this case, the used device is given by `torch.cuda.current_device()` and it is the user's responsibility to ensure that this is correctly set so that each rank has an individual GPU, via `torch.cuda.set_device()`.

Examples

```
>>> import torch
>>> import mmrazor.core.utils as dist
>>> # non-distributed environment
>>> data = ['foo', 12, {1: 2}]
>>> dist.broadcast_object_list(data)
>>> data
['foo', 12, {1: 2}]
>>> # distributed environment
>>> # We have 2 process groups, 2 ranks.
>>> if dist.get_rank() == 0:
>>>     # Assumes world_size of 3.
>>>     data = ["foo", 12, {1: 2}]  # any picklable object
>>> else:
>>>     data = [None, None, None]
>>> dist.broadcast_object_list(data)
>>> data
["foo", 12, {1: 2}]  # Rank 0
["foo", 12, {1: 2}]  # Rank 1
```

`mmrazor.core.utils.`**`get_backend`**(*group: Optional[torch._C._distributed_c10d.ProcessGroup] = None*) → Optional[str]

Return the backend of the given process group.

Note: Calling `get_backend` in non-distributed environment will return None.

Parameters **group** (*ProcessGroup, optional*) – The process group to work on. The default is the general main process group. If another specific group is specified, the calling process must be part of `group`. Defaults to None.

Returns Return the backend of the given process group as a lower case string if in distributed environment, otherwise None.

Return type str or None

`mmrazor.core.utils.`**`get_default_group`**() → Optional[torch._C._distributed_c10d.ProcessGroup]

Return default process group.

mmrazor.core.utils.**get_rank**(*group: Optional[torch._C._distributed_c10d.ProcessGroup] = None*) → int
Return the rank of the given process group.

Rank is a unique identifier assigned to each process within a distributed process group. They are always consecutive integers ranging from 0 to `world_size`. .. note:: Calling `get_rank` in non-distributed environment will return 0.

Parameters **group** (*ProcessGroup, optional*) – The process group to work on. If None, the default process group will be used. Defaults to None.

Returns Return the rank of the process group if in distributed environment, otherwise 0.

Return type int

mmrazor.core.utils.**get_world_size**(*group: Optional[torch._C._distributed_c10d.ProcessGroup] = None*) → int
Return the number of the given process group.

Note: Calling `get_world_size` in non-distributed environment will return 1.

Parameters **group** (*ProcessGroup, optional*) – The process group to work on. If None, the default process group will be used. Defaults to None.

Returns Return the number of processes of the given process group if in distributed environment, otherwise 1.

Return type int

mmrazor.core.utils.**set_lr**(*runner*, *lr_groups*, *freeze_optimizers=[]*)
Set specified learning rate in optimizer.

CHAPTER

NINETEEN

MMRAZOR.DATASETS

19.1 datasets

CHAPTER

TWENTY

MMRAZOR.MODELS

20.1 algorithms

class mmrazor.models.algorithms.**AlignMethodDistill**(***kwargs*)

class mmrazor.models.algorithms.**AutoSlim**(*num_sample_training=4*, *input_shape=(3, 224, 224)*, *bn_training_mode=False*, ***kwargs*)

AutoSlim: A one-shot architecture search for channel numbers.

Please refer to the *paper <https://arxiv.org/abs/1903.11728>* for details.

Parameters

- **num_sample_training** (*int*) – In each iteration we train the model at smallest width, largest width and (*num_sample_training* 2) random widths. It should be no less than 2. Defaults to 4
- **input_shape** (*tuple*) – Input shape used for calculation the flops of the supernet.
- **bn_training_mode** (*bool*) – Whether set bn to training mode when model is set to eval mode. Note that in slimmable networks, accumulating different numbers of channels results in different feature means and variances, which further leads to inaccurate statistics of shared BN. Set `bn_training_mode` to True to use the feature means and variances in a batch.

get_subnet_flops()

A hacky way to get flops information of a subnet.

train(*mode=True*)

Overwrite the train method in `nn.Module` to set `nn.BatchNorm` to training mode when model is set to eval mode when `self.bn_training_mode` is `True`.

Parameters mode (*bool*) – whether to set training mode (`True`) or evaluation mode (`False`). Default: `True`.

train_step(*data*, *optimizer*)

Train step function.

This function implements the standard training iteration for autoslim pretraining and retraining.

Parameters

- **data** (*dict*) – Input data from dataloader.
- **optimizer** (`torch.optim.Optimizer`) – The optimizer to accumulate gradient

class mmrazor.models.algorithms.**Darts**(*unroll*, ***kwargs*) `train_step`(*data*, *optimizer*)

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 are 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

class `mmrazor.models.algorithms.DetNAS`(***kwargs*)

Implementation of DetNAS

class `mmrazor.models.algorithms.GeneralDistill`(*with_student_loss=True*, *with_teacher_loss=False*, ***kwargs*)

General Distillation Algorithm.

Parameters

- **with_student_loss** (*bool*) – Whether to use student loss. Defaults to True.
- **with_teacher_loss** (*bool*) – Whether to use teacher loss. Defaults to False.

class `mmrazor.models.algorithms.SPOS`(*input_shape=(3, 224, 224)*, *bn_training_mode=False*, ***kwargs*)

Implementation of SPOS

`get_subnet_flops`()

Get subnet's flops based on the complexity information of supernet.

`train`(*mode=True*)

Overwrite the train method in *nn.Module* to set *nn.BatchNorm* to training mode when model is set to eval mode when *self.bn_training_mode* is *True*.

Parameters mode (*bool*) – whether to set training mode (*True*) or evaluation mode (*False*). Default: *True*.

`train_step`(*data*, *optimizer*)

The iteration step during training.

In retraining stage, to train subnet like common model. In pre-training stage, First to sample a subnet from supernet, then to train the subnet.

20.2 architectures

class mmrazor.models.architectures.**MMClsArchitecture**(***kwargs*)
Architecture based on MMCls.

cal_pseudo_loss(*pseudo_img*)
Used for executing `forward` with pseudo_img.

forward_dummy(*img*)
Used for calculating network flops.

class mmrazor.models.architectures.**MMDetArchitecture**(***kwargs*)
Architecture based on MMDet.

cal_pseudo_loss(*pseudo_img*)
Used for executing `forward` with pseudo_img.

class mmrazor.models.architectures.**MMSegArchitecture**(***kwargs*)
Architecture based on MMSeg.

20.3 distillers

class mmrazor.models.distillers.**SelfDistiller**(*components*, ***kwargs*)
Transfer knowledge inside a single model.

Parameters **components** (*dict*) – The details of the distillation. It usually includes the module names of the teacher and the student, and the losses used in the distillation.

compute_distill_loss(*data*)
Compute the distillation loss.

exec_student_forward(*student*, *data*)
Forward computation of the student.

Parameters

- **student** (`torch.nn.Module`) – The student model to be used in the distillation.
- **data** (*dict*) – The output of dataloader.

exec_teacher_forward(*teacher*, *data*)
Forward computation of the teacher.

Parameters

- **teacher** (`torch.nn.Module`) – The teacher model to be used in the distillation.
- **data** (*dict*) – The output of dataloader.

prepare_from_student(*student*)
Registers a global forward hook for each teacher module and student module to be used in the distillation.

Parameters **student** (`torch.nn.Module`) – The student model to be used in the distillation.

reset_outputs(*outputs*)
Reset the teacher's outputs or student's outputs.

student_forward_output_hook(*module*, *inputs*, *outputs*)
Save the output.

Parameters

- **module** (`torch.nn.Module`) – the module of register hook
- **inputs** (*tuple*) – input of module
- **outputs** (*tuple*) – out of module

teacher_forward_output_hook(*module*, *inputs*, *outputs*)
Save the output.

Parameters

- **module** (`torch.nn.Module`) – the module of register hook
- **inputs** (*tuple*) – input of module
- **outputs** (*tuple*) – out of module

class mmrazor.models.distillers.**SingleTeacherDistiller**(*teacher*, *teacher_trainable=False*, *teacher_norm_eval=True*, *components=()*, ***kwargs*)

Distiller with single teacher.

Parameters

- **teacher** (*dict*) – The config dict for teacher.
- **teacher_trainable** (*bool*) – Whether the teacher is trainable. Default: False.
- **teacher_norm_eval** (*bool*) – Whether to set teacher's norm layers to eval mode, namely, freeze running stats (mean and var). Note: Effect on Batch Norm and its variants only. Default: True.
- **components** (*dict*) – The details of the distillation. It usually includes the module names of the teacher and the student, and the losses used in the distillation.

build_align_module(*cfg*)
Build `align_module` from the *cfg*.

`align_module` is needed when the number of channels output by the teacher module is not equal to that of the student module, or for some other reasons.

Parameters **cfg** (*dict*) – The config dict for `align_module`.

build_teacher(*cfg*)
Build a model from the *cfg*.

compute_distill_loss(*data=None*)
Compute the distillation loss.

exec_student_forward(*student*, *data*)
Execute the teacher's forward function.

After this function, the student's featuremaps will be saved in `student_outputs`.

exec_teacher_forward(*data*)
Execute the teacher's forward function.

After this function, the teacher's featuremaps will be saved in `teacher_outputs`.

get_teacher_outputs(*teacher_module_name*)
Get the outputs according module name.

prepare_from_student(*student*)
Registers a global forward hook for each teacher module and student module to be used in the distillation.

Parameters **student** (`torch.nn.Module`) – The student model to be used in the distillation.

reset_outputs(*outputs*)
Reset the teacher's outputs or student's outputs.

student_forward_output_hook(*module*, *inputs*, *outputs*)
Save the module's forward output.

Parameters

- **module** (torch.nn.Module) – The module to register hook.
- **inputs** (*tuple*) – The input of the module.
- **outputs** (*tuple*) – The output of the module.

teacher_forward_output_hook(*module*, *inputs*, *outputs*)
Save the module's forward output.

Parameters

- **module** (torch.nn.Module) – The module to register hook.
- **inputs** (*tuple*) – The input of the module.
- **outputs** (*tuple*) – The output of the module.

train(*mode=True*)
Set distiller's forward mode.

20.4 losses

class mmrazor.models.losses.**AngleWiseRKD**(*loss_weight=50.0*, *with_l2_norm=True*)
PyTorch version of `Relational Knowledge Distillation.

<https://arxiv.org/abs/1904.05068>`_.

Parameters

- **loss_weight** (*float*) – Weight of angle-wise distillation loss. Defaults to 50.0.
- **with_l2_norm** (*bool*) – Whether to normalize the model predictions before calculating the loss. Defaults to True.

angle_loss(*preds_S*, *preds_T*)
Calculate the angle-wise distillation loss.

forward(*preds_S*, *preds_T*)
Forward computation.

Parameters

- **preds_S** (*torch.Tensor*) – The student model prediction with shape (N, C, H, W) or shape (N, C).
- **preds_T** (*torch.Tensor*) – The teacher model prediction with shape (N, C, H, W) or shape (N, C).

Returns The calculated loss value.

Return type torch.Tensor

class mmrazor.models.losses.**ChannelWiseDivergence**(*tau=1.0*, *loss_weight=1.0*)
PyTorch version of `Channel-wise Distillation for Semantic Segmentation.

<https://arxiv.org/abs/2011.13256>`_.

Parameters

- **tau** (*float*) – Temperature coefficient. Defaults to 1.0.
- **loss_weight** (*float*) – Weight of loss. Defaults to 1.0.

forward(*preds_S*, *preds_T*)

Forward computation.

Parameters

- **preds_S** (*torch.Tensor*) – The student model prediction with shape (N, C, H, W).
- **preds_T** (*torch.Tensor*) – The teacher model prediction with shape (N, C, H, W).

Returns The calculated loss value.

Return type torch.Tensor

class mmrazor.models.losses.**DistanceWiseRKD**(*loss_weight=25.0*, *with_l2_norm=True*)

PyTorch version of distance-wise loss of `Relational Knowledge Distillation.

<https://arxiv.org/abs/1904.05068>`_.

Parameters

- **loss_weight** (*float*) – Weight of distance-wise distillation loss. Defaults to 25.0.
- **with_l2_norm** (*bool*) – Whether to normalize the model predictions before calculating the loss. Defaults to True.

distance_loss(*preds_S*, *preds_T*)

Calculate distance-wise distillation loss.

forward(*preds_S*, *preds_T*)

Forward computation.

Parameters

- **preds_S** (*torch.Tensor*) – The student model prediction with shape (N, C, H, W) or shape (N, C).
- **preds_T** (*torch.Tensor*) – The teacher model prediction with shape (N, C, H, W) or shape (N, C).

Returns The calculated loss value.

Return type torch.Tensor

class mmrazor.models.losses.**KLDivergence**(*tau=1.0*, *reduction='batchmean'*, *loss_weight=1.0*)

A measure of how one probability distribution Q is different from a second, reference probability distribution P.

Parameters

- **tau** (*float*) – Temperature coefficient. Defaults to 1.0.
- **reduction** (*str*) – Specifies the reduction to apply to the loss: `'none'` | `'batchmean'` | `'sum'` | `'mean'`. `'none'`: no reduction will be applied, `'batchmean'`: the sum of the output will be divided by

  the batchsize,

  `'sum'`: the output will be summed, `'mean'`: the output will be divided by the number of

  elements in the output.

  Default: `'batchmean'`

- **loss_weight** (*float*) – Weight of loss. Defaults to 1.0.

forward(*preds_S*, *preds_T*)

Forward computation.

Parameters

- **preds_S** (*torch.Tensor*) – The student model prediction with shape (N, C, H, W) or shape (N, C).
- **preds_T** (*torch.Tensor*) – The teacher model prediction with shape (N, C, H, W) or shape (N, C).

Returns The calculated loss value.

Return type torch.Tensor

class mmrazor.models.losses.**WSLD**(*tau=1.0*, *loss_weight=1.0*, *num_classes=1000*)

PyTorch version of Rethinking Soft Labels for Knowledge Distillation: A Bias-Variance Tradeoff Perspective.

Parameters

- **tau** (*float*) – Temperature coefficient. Defaults to 1.0.
- **loss_weight** (*float*) – Weight of loss. Defaults to 1.0.
- **num_classes** (*int*) – Defaults to 1000.

forward(*student*, *teacher*)

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.

20.5 mutables

class mmrazor.models.mutables.**DifferentiableEdge**(*with_arch_param*, ***kwargs*)

Differentiable Edge.

Search the best module from choices by learnable parameters.

Parameters **with_arch_param** (*bool*) – whether build learable architecture parameters.

build_arch_param()

build learnable architecture parameters.

compute_arch_probs(*arch_param*)

compute chosen probs according architecture parameters.

forward(*prev_inputs*, *arch_param=None*)

forward function.

In some algorithms, there are several `MutableModule` share the same architecture parameters. So the architecture parameters are passed in as args.

Parameters

- **prev_inputs** (*list[torch.Tensor]*) – each choice's inputs.

- **arch_param** (*torch.nn.Parameter*) – architecture parameters.

class mmrazor.models.mutables.**DifferentiableOP**(*with_arch_param*, ***kwargs*)
Differentiable OP.

Search the best module from choices by learnable parameters.

Parameters **with_arch_param** (*bool*) – whether build learable architecture parameters.

build_arch_param()
build learnable architecture parameters.

compute_arch_probs(*arch_param*)
compute chosen probs according architecture parameters.

forward(*x*, *arch_param=None*)
forward function.

In some algorithms, there are several `MutableModule` share the same architecture parameters. So the architecture parameters are passed in as args.

Parameters

- **prev_inputs** (*list[torch.Tensor]*) – each choice's inputs.
- **arch_param** (*torch.nn.Parameter*) – architecture parameters.

class mmrazor.models.mutables.**GumbelEdge**(***kwargs*)
Gumbel Edge.

Search the best module from choices by gumbel trick.

compute_arch_probs(*arch_param*)
compute chosen probs by gumbel trick.

set_temperature(*temperature*)
Modify the temperature.

class mmrazor.models.mutables.**GumbelOP**(*tau=1.0*, *hard=True*, ***kwargs*)
Gumbel OP.

Search the best module from choices by gumbel trick.

compute_arch_probs(*arch_param*)
compute chosen probs by gumbel trick.

set_temperature(*temperature*)
Modify the temperature.

class mmrazor.models.mutables.**MutableEdge**(*choices*, ***kwargs*)
Mutable Edge. In some NAS algorithms (Darts, AutoDeeplab, etc.), the connections between modules are searchable, such as the connections between a node and its previous nodes in Darts. `MutableEdge` has N modules to process N inputs respectively.

Parameters **choices** (*torch.nn.ModuleDict*) – Unlike `MutableOP`, there are already created modules in choices.

build_choices(*cfg*)
MutableEdge's choices is already built.

class mmrazor.models.mutables.**MutableModule**(*space_id*, *num_chosen=1*, *init_cfg=None*, ***kwargs*)
Base class for `MUTABLES`. Searchable module for building searchable architecture in NAS. It mainly consists of module and mask, and achieving searchable function by handling mask.

Parameters

- **space_id** (*str*) – Used to index Placeholder, it is one and only index for each Placeholder.
- **num_chosen** (*str*) – The number of chosen OPS in the MUTABLES.
- **init_cfg** (*dict*) – Init config for BaseModule.

build_choice_mask()
Generate the choice mask for the choices of MUTABLES.

Returns Init choice mask. Its elements' type is bool.

Return type torch.Tensor

abstract **build_choices**(*cfg*)
Build all chosen OPS used to combine MUTABLES, and the choices will be sampled.

Parameters **cfg** (*dict*) – The config for the choices.

build_space_mask()
Generate the space mask for the search spaces of MUTATORS.

Returns Init choice mask. Its elements' type is float.

Return type torch.Tensor

property **choice_modules**
The choices' modules.

Returns The values of the choices.

Return type tuple

property **choice_names**
The choices' names.

Returns The keys of the choices.

Return type tuple

export(*chosen*)
Delete not chosen OPS in the choices.

Parameters **chosen** (*list[str]*) – Names of chosen OPS.

abstract **forward**(*x*)
Forward computation.

Parameters **x** (*tensor | tuple[tensor]*) – x could be a Torch.tensor or a tuple of Torch.tensor, containing input data for forward computation.

property **num_choices**
The number of the choices.

Returns the length of the choices.

Return type int

set_choice_mask(*mask*)
Use the mask to update the choice mask.

Parameters **mask** (*torch.Tensor*) – Choice mask specified to update the choice mask.

class mmrazor.models.mutables.**MutableOP**(*choices*, *choice_args*, ***kwargs*)
An important type of MUTABLES, inherits from MutableModule.

Parameters

- **choices** (*dict*) – The configs for the choices, the chosen `OPS` used to combine `MUTABLES`.
- **choice_args** (*dict*) – The args used to set chosen `OPS`.

build_choices(*cfgs*, *choice_args*)
Build all chosen `OPS` used to combine `MUTABLES`, and the choices will be sampled.

Parameters

- **cfgs** (*dict*) – The configs for the choices.
- **choice_args** (*dict*) – The args used to set chosen `OPS`.

Returns Consists of chosen `OPS` in the arg *cfgs*.

Return type torch.nn.ModuleDict

class mmrazor.models.mutables.**OneShotOP**(***kwargs*)
A type of `MUTABLES` for the one-shot NAS.

forward(*x*)
Forward computation for chosen `OPS`, in one-shot NAS, the number of chosen `OPS` can only be one.

Parameters **x** (*tensor | tuple[tensor]*) – x could be a Torch.tensor or a tuple of Torch.tensor, containing input data for forward computation.

Returns The result of forward.

Return type torch.Tensor

20.6 mutators

class mmrazor.models.mutators.**DartsMutator**(*ignore_choices=('zero')*, ***kwargs*)

class mmrazor.models.mutators.**DifferentiableMutator**(***kwargs*)
A mutator for the differentiable NAS, which mainly provide some core functions of changing the structure of `ARCHITECTURES`.

build_arch_params(*supernet*)
This function will build many arch params, which are generally used in diffirentiale search algorithms, such as Darts' series. Each space_id corresponds to an arch param, so the Mutable with the same space_id share the same arch param.

Parameters **supernet** (`torch.nn.Module`) – The architecture to be used in your algorithm.

Returns

the arch params are got after traversing the supernet.

Return type torch.nn.ParameterDict

modify_supernet_forward(*supernet*)
Modify the supernet's default value in forward. Traverse all child modules of the model, modify the supernet's default value in :func:'forward' of each Space.

Parameters **supernet** (`torch.nn.Module`) – The architecture to be used in your algorithm.

prepare_from_supernet(*supernet*)
Inherit from `BaseMutator`'s, execute some customized functions exclude implementing origin `prepare_from_supernet`.

Parameters **supernet** (`torch.nn.Module`) – The architecture to be used in your algorithm.

class mmrazor.models.mutators.**OneShotMutator**(***kwargs*)

A mutator for the one-shot NAS, which mainly provide some core functions of changing the structure of ARCHITECTURES.

static crossover(*subnet_dict1*, *subnet_dict2*)

Crossover used in evolution search.

Parameters

- **subnet_dict1** (*dict*) – Record the information to build the subnet from the supernet, its keys are the properties `space_id` of placeholders in the mutator's search spaces, its values are masks.
- **subnet_dict2** (*dict*) – Record the information to build the subnet from the supernet, its keys are the properties `space_id` of placeholders in the mutator's search spaces, its values are masks.

Returns A new subnet_dict after crossover.

Return type dict

static get_random_mask(*space_info*, *searching*)

Generate random mask for randomly sampling.

Parameters

- **space_info** (*dict*) – Record the information of the space need to sample.
- **searching** (*bool*) – Whether is in search stage.

Returns Random mask generated.

Return type torch.Tensor

mutation(*subnet_dict*, *prob=0.1*)

Mutation used in evolution search.

Parameters

- **subnet_dict** (*dict*) – Record the information to build the subnet from the supernet, its keys are the properties `space_id` of placeholders in the mutator's search spaces, its values are masks.
- **prob** (*float*) – The probability of mutation.

Returns A new subnet_dict after mutation.

Return type dict

static reset_in_subnet(*m*, *in_subnet=True*)

Reset the module's attribution.

Parameters

- **m** (`torch.nn.Module`) – The module in the supernet.
- **in_subnet** (*bool*) – If the module in subnet, set `in_subnet` to True, otherwise set to False.

sample_subnet(*searching=False*)

Random sample subnet by random mask.

Parameters **searching** (*bool*) – Whether is in search stage.

Returns

Record the information to build the subnet from the supernet, its keys are the properties `space_id` of placeholders in the mutator's search spaces, its values are random mask generated.

Return type dict

set_chosen_subnet(*subnet_dict*)
Set chosen subnet in the search_spaces after searching stage.

Parameters **subnet_dict** (*dict*) – Record the information to build the subnet from the supernet, its keys are the properties `space_id` of placeholders in the mutator's search spaces, its values are masks.

set_subnet(*subnet_dict*)
Setting subnet in the supernet based on the result of `sample_subnet` by changing the flag: `in_subnet`, which is easy to implement some operations for subnet, such as `forward`, calculate flops and so on.

Parameters **subnet_dict** (*dict*) – Record the information to build the subnet from the supernet, its keys are the properties `space_id` of placeholders in the mutator's search spaces, its values are masks.

20.7 ops

class mmrazor.models.ops.**DartsDilConv**(*kernel_size*, *use_drop_path=False*, *norm_cfg={'type': 'BN'}*, ***kwargs*)

forward(*x*)
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 mmrazor.models.ops.**DartsPoolBN**(*pool_type*, *kernel_size=3*, *norm_cfg={'type': 'BN'}*, *use_drop_path=False*, ***kwargs*)

forward(*x*)
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 mmrazor.models.ops.**DartsSepConv**(*kernel_size*, *use_drop_path=False*, *norm_cfg={'type': 'BN'}*, ***kwargs*)

forward(*x*)
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 mmrazor.models.ops.**DartsSkipConnect**(*use_drop_path=False*, *norm_cfg={'type': 'BN'}*, ***kwargs*)
Reduce feature map size by factorized pointwise (stride=2).

forward(*x*)
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 mmrazor.models.ops.**DartsZero**(***kwargs*)

forward(*x*)
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 mmrazor.models.ops.**Identity**(*conv_cfg=None*, *norm_cfg={'type': 'BN'}*, *act_cfg=None*, ***kwargs*)
Base class for searchable operations.

Parameters

- **conv_cfg** (*dict, optional*) – Config dict for convolution layer. Default: None, which means using conv2d.
- **norm_cfg** (*dict*) – Config dict for normalization layer. Default: dict(type='BN').
- **act_cfg** (*dict*) – Config dict for activation layer. Default: None.

forward(*x*)
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 mmrazor.models.ops.**MBBlock**(*kernel_size*, *expand_ratio*, *se_cfg=None*, *conv_cfg=None*, *norm_cfg={'type': 'BN'}*, *act_cfg={'type': 'ReLU'}*, *drop_path_rate=0.0*, *with_cp=False*, ***kwargs*)
Mobilenet block for Searchable backbone.

Parameters

- **kernel_size** (*int*) – Size of the convolving kernel.
- **expand_ratio** (*int*) – The input channels' expand factor of the depthwise convolution.
- **se_cfg** (*dict, optional*) – Config dict for se layer. Defaults to None, which means no se layer.
- **conv_cfg** (*dict, optional*) – Config dict for convolution layer. Default: None, which means using conv2d.
- **norm_cfg** (*dict*) – Config dict for normalization layer. Default: dict(type='BN').
- **act_cfg** (*dict*) – Config dict for activation layer. Default: dict(type='ReLU').
- **drop_path_rate** (*float*) – stochastic depth rate. Defaults to 0.
- **with_cp** (*bool*) – Use checkpoint or not. Using checkpoint will save some memory while slowing down the training speed. Default: False.

Returns The output tensor.

Return type Tensor

forward(*x*)

Forward function.

Parameters **x** (*torch.Tensor*) – The input tensor.

Returns The output tensor.

Return type torch.Tensor

class mmrazor.models.ops.**ShuffleBlock**(*kernel_size*, *conv_cfg=None*, *norm_cfg={'type': 'BN'}*, *act_cfg={'type': 'ReLU'}*, *with_cp=False*, ***kwargs*)

InvertedResidual block for Searchable ShuffleNetV2 backbone.

Parameters

- **kernel_size** (*int*) – Size of the convolving kernel.
- **stride** (*int*) – Stride of the convolution layer. Default: 1
- **conv_cfg** (*dict, optional*) – Config dict for convolution layer. Default: None, which means using conv2d.
- **norm_cfg** (*dict*) – Config dict for normalization layer. Default: dict(type='BN').
- **act_cfg** (*dict*) – Config dict for activation layer. Default: dict(type='ReLU').
- **with_cp** (*bool*) – Use checkpoint or not. Using checkpoint will save some memory while slowing down the training speed. Default: False.

Returns The output tensor.

Return type Tensor

forward(*x*)

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 mmrazor.models.ops.**ShuffleXception**(*conv_cfg=None*, *norm_cfg={'type': 'BN'}*, *act_cfg={'type': 'ReLU'}*, *with_cp=False*, ***kwargs*)

Xception block for ShuffleNetV2 backbone.

Parameters

- **conv_cfg** (*dict, optional*) – Config dict for convolution layer. Defaults to None, which means using conv2d.
- **norm_cfg** (*dict*) – Config dict for normalization layer. Defaults to dict(type='BN').
- **act_cfg** (*dict*) – Config dict for activation layer. Defaults to dict(type='ReLU').
- **with_cp** (*bool*) – Use checkpoint or not. Using checkpoint will save some memory while slowing down the training speed. Defaults to False.

Returns The output tensor.

Return type Tensor

forward(*x*)

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.

20.8 pruners

class mmrazor.models.pruners.**RatioPruner**(*ratios*, ***kwargs*)

A random ratio pruner.

Each layer can adjust its own width ratio randomly and independently.

Parameters **ratios** (*list | tuple*) – Width ratio of each layer can be chosen from *ratios* randomly. The width ratio is the ratio between the number of reserved channels and that of all channels in a layer. For example, if *ratios* is [0.25, 0.5], there are 2 cases for us to choose from when we sample from a layer with 12 channels. One is sampling the very first 3 channels in this layer, another is sampling the very first 6 channels in this layer. Default to None.

convert_switchable_bn(*module*, *num_bns*)

Convert normal `nn.BatchNorm2d` to `SwitchableBatchNorm2d`.

Parameters

- **module** (`torch.nn.Module`) – The module to be converted.
- **num_bns** (*int*) – The number of `nn.BatchNorm2d` in a `SwitchableBatchNorm2d`.

Returns

The converted module. Each `nn.BatchNorm2d` in this module has been converted to a `SwitchableBatchNorm2d`.

Return type `torch.nn.Module`

`get_channel_mask`(*out_mask*)
Randomly choose a width ratio of a layer from `ratios`

`prepare_from_supernet`(*supernet*)
Prepare for pruning.

`sample_subnet`()
Random sample subnet by random mask.

Returns

Record the information to build the subnet from the supernet, its keys are the properties `space_id` in the pruner's search spaces, and its values are corresponding sampled out_mask.

Return type dict

`set_min_channel`()
Set the number of channels each layer to minimum.

`switch_subnet`(*channel_cfg*, *subnet_ind=None*)
Switch the channel config of the supernet according to channel_cfg.

If we train more than one subnet together, we need to switch the channel_cfg from one to another during one training iteration.

Parameters

- **channel_cfg** (*dict*) – The channel config of a subnet. Key is space_id and value is a dict which includes out_channels (and in_channels if exists).
- **subnet_ind** (*int, optional*) – The index of the current subnet. If we replace normal BatchNorm2d with `SwitchableBatchNorm2d`, we should switch the index of `SwitchableBatchNorm2d` when switch subnet. Defaults to None.

class mmrazor.models.pruners.**StructurePruner**(*except_start_keys=['head.fc']*)
Base class for structure pruning. This class defines the basic functions of a structure pruner. Any pruner that inherits this class should at least define its own *sample_subnet* and *set_min_channel* functions. This part is being continuously optimized, and there may be major changes in the future.

Reference to https://github.com/jshilong/FisherPruning

Parameters **except_start_keys** (*List[str]*) – the module whose name start with a string in except_start_keys will not be prune.

`add_pruning_attrs`(*module*)
Add masks to a `nn.Module`.

`build_channel_spaces`(*name2module*)
Build channel search space.

Parameters **name2module** (*dict*) – A mapping between module_name and module.

Returns

The channel search space. The key is space_id and the value is the corresponding out_mask.

Return type dict

`concat_backward_parser`(*grad_fn*, *module2name*, *var2module*, *cur_path*, *result_paths*, *visited*)
Parse the backward of a concat operation.

Example

```
>>> conv = nn.Conv2d(3, 3, 3)
>>> pseudo_img = torch.rand(1, 3, 224, 224)
>>> out1 = conv(pseudo_img)
>>> out2 = conv(pseudo_img)
>>> out = torch.cat([out1, out2], dim=1)
>>> print(out.grad_fn.next_functions)
((<ThnnConv2DBackward object at 0x0000020E405F24C8>, 0),
(<ThnnConv2DBackward object at 0x0000020E405F2648>, 0))
>>> # the length of ``out.grad_fn.next_functions`` is two means
>>> # ``out`` is obtained by concatenating two tensors
```

conv_backward_parser(*grad_fn*, *module2name*, *var2module*, *cur_path*, *result_paths*, *visited*)
Parse the backward of a conv layer.

Example

```
>>> conv = nn.Conv2d(3, 3, 3)
>>> pseudo_img = torch.rand(1, 3, 224, 224)
>>> out = conv(pseudo_img)
>>> print(out.grad_fn.next_functions)
((None, 0), (<AccumulateGrad object at 0x0000020E405CBD88>, 0),
(<AccumulateGrad object at 0x0000020E405CB588>, 0))
>>> # op.next_functions[0][0] is None means this ThnnConv2DBackward
>>> # op has no parents
>>> # op.next_functions[1][0].variable is the weight of this Conv2d
>>> # module
>>> # op.next_functions[2][0].variable is the bias of this Conv2d
>>> # module
```

deploy_subnet(*supernet*, *channel_cfg*)
Deploy subnet according *channel_cfg*.

export_subnet()
Generate subnet configs according to the in_mask and out_mask of a module.

find_make_group_parser(*node_name*, *name2module*)
Find the corresponding make_group_parser according to the `node_name`

find_node_parents(*paths*)
Find the parent node of a node.

A node in the `paths` can be a module name or a operation name such as *concat_140719322997152*. Note that the string of numbers following `concat` do not have a particular meaning. It just make the operation name unique.

Parameters **paths** (*list*) – The traced paths.

get_max_channel_bins(*max_channel_bins*)
Get the max number of channel bins of all the groups which can be pruned during searching.

Parameters **max_channel_bins** (*int*) – The max number of bins in each layer.

get_space_id(*module_name*)
Get the corresponding space_id of the module_name.

The modules who share the same space_id will share the same out_mask. If the module is the output module(there is no other `nn.Module` whose input is its output), this function will return None. As the output module can not be pruned. If the input of this module is the concatenation of the output of several `nn.Module`, this function will return a dict object. If this module is in one of the groups, this function will return the group name. As the modules in the same group should share the same space_id. Otherwise, this function will return the module_name as space_id.

Parameters **module_name** (*str*) – the name of a `nn.Module`.

Returns the corresponding space_id of the module_name.

Return type str or dict or None

linear_backward_parser(*grad_fn*, *module2name*, *var2module*, *cur_path*, *result_paths*, *visited*)
Parse the backward of a conv layer.

Example

```
>>> fc = nn.Linear(3, 3, bias=True)
>>> input = torch.rand(3, 3)
>>> out = fc(input)
>>> print(out.grad_fn.next_functions)
((<AccumulateGrad object at 0x0000020E405F75C8>, 0), (None, 0),
(<TBackward object at 0x0000020E405F7D48>, 0))
>>> # op.next_functions[0][0].variable is the bias of this Linear
>>> # module
>>> # op.next_functions[1][0] is None means this AddmmBackward op
>>> # has no parents
>>> # op.next_functions[2][0] is the TBackward op, and
>>> # op.next_functions[2][0].next_functions[0][0].variable is
>>> # the transpose of the weight of this Linear module
```

make_same_out_channel_groups(*node2parents*, *name2module*)
Modules have the same child should be in the same group.

static modify_conv_forward(*module*)
Modify the forward method of a conv layer.

static modify_fc_forward(*module*)
Modify the forward method of a linear layer.

prepare_from_supernet(*supernet*)
Prepare for pruning.

abstract sample_subnet()
Sample a subnet from the supernet.

Returns

Record the information to build the subnet from the supernet, its keys are the properties `space_id` in the pruner's search spaces, and its values are corresponding sampled out_mask.

Return type dict

set_channel_bins(*channel_bins_dict*, *max_channel_bins*)
Set subnet according to the number of channel bins in a layer.

Parameters

- **channel_bins_dict** (*dict*) – The number of bins in each layer. Key is the space_id of each layer and value is the corresponding mask of channel bin.
- **max_channel_bins** (*int*) – The max number of bins in each layer.

set_max_channel()
Set the number of channels each layer to maximum.

abstract set_min_channel()
Set the number of channels each layer to minimum.

set_subnet(*subnet_dict*)
Modify the in_mask and out_mask of modules in supernet according to subnet_dict.

Parameters **subnet_dict** (*dict*) – the key is space_id and the value is the corresponding sampled out_mask.

trace_non_pass_path(*grad_fn, module2name, var2module, cur_path, result_paths, visited*)
Trace the topology of all the NON_PASS_MODULE.

trace_norm_conv_links(*grad_fn, module2name, var2module, norm_conv_links, visited*)
Get the convolutional layer placed before a normalization layer in the model.

Example

```
>>> conv = nn.Conv2d(3, 3, 3)
>>> norm = nn.BatchNorm2d(3)
>>> pseudo_img = torch.rand(1, 3, 224, 224)
>>> out = norm(conv(pseudo_img))
>>> print(out.grad_fn)
<NativeBatchNormBackward object at 0x0000022BC709DB08>
>>> print(out.grad_fn.next_functions)
((<ThnnConv2DBackward object at 0x0000020E40639688>, 0),
(<AccumulateGrad object at 0x0000020E40639208>, 0),
(<AccumulateGrad object at 0x0000020E406398C8>, 0))
>>> # op.next_functions[0][0] is ThnnConv2DBackward means
>>> # the parent of this NativeBatchNormBackward op is
>>> # ThnnConv2DBackward
>>> # op.next_functions[1][0].variable is the weight of this
>>> # normalization module
>>> # op.next_functions[2][0].variable is the bias of this
>>> # normalization module
```

```
>>> # Things are different in InstanceNorm
>>> conv = nn.Conv2d(3, 3, 3)
>>> norm = nn.InstanceNorm2d(3, affine=True)
>>> out = norm(conv(pseudo_img))
>>> print(out.grad_fn)
<ViewBackward object at 0x0000022BC709DD48>
>>> print(out.grad_fn.next_functions)
((<NativeBatchNormBackward object at 0x0000022BC81E8A08>, 0),)
>>> print(out.grad_fn.next_functions[0][0].next_functions)
((<ViewBackward object at 0x0000022BC81E8DC8>, 0),
(<RepeatBackward object at 0x0000022BC81E8D08>, 0),
(<RepeatBackward object at 0x0000022BC81E81C8>, 0))
>>> # Hence, a dfs is necessary.
```

trace_shared_module_hook(*module*, *inputs*, *outputs*)

Trace shared modules. Modules such as the detection head in RetinaNet which are visited more than once during `forward()` are shared modules.

Parameters

- **module** (`torch.nn.Module`) – The module to register hook.
- **inputs** (*tuple*) – The input of the module.
- **outputs** (*tuple*) – The output of the module.

CHAPTER

TWENTYONE

MMRAZOR.UTILS

mmrazor.utils.**find_latest_checkpoint**(*path*, *suffix='pth'*)

Find the latest checkpoint from the working directory.

Parameters

- **path** (*str*) – The path to find checkpoints.
- **suffix** (*str*) – File extension. Defaults to pth.

Returns File path of the latest checkpoint.

Return type latest_path(str | None)

References

mmrazor.utils.**setup_multi_processes**(*cfg*)

Setup multi-processing environment variables.

CHAPTER

TWENTYTWO

INDICES AND TABLES

- genindex
- search

PYTHON MODULE INDEX

m

INDEX

D

E

F

G

I

K

L

M

N

O

P

R

S

T

U

W