mmrazor

MMRazor Author

Jun 25, 2023

GET STARTED:

CHAPTER

ONE

OVERVIEW

1.1 Why MMRazor

MMRazor is a model compression toolkit for model slimming, which includes 4 mainstream technologies:

- Neural Architecture Search (NAS)
- Pruning
- Knowledge Distillation (KD)
- Quantization

It is a part of the OpenMMLab project. If you want to use it now, please refer to Installation.

1.1.1 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.

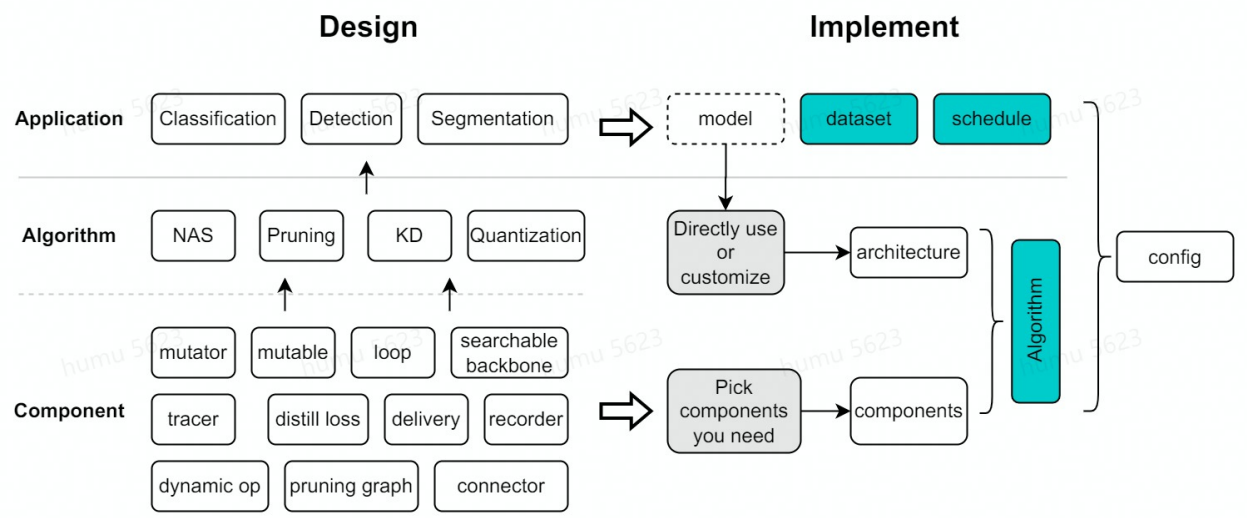
- **Flexibility**

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.

1.2 Design and Implement



1.2.1 Design

There are 3 layers (**Application / Algorithm / Component**) in overview design. MMRazor mainly includes both of **Component** and **Algorithm**, while **Application** consist of some OpenMMLab upstream repos, such as MMClassification, MMDetection, MMSegmentation and so on.

Component provides many useful functions for quickly implementing **Algorithm.** And thanks to OpenMMLab 's powerful and highly flexible config mode and registry mechanism, **Algorithm** can be conveniently applied to **Application.**

How to apply our lightweight algorithms to some upstream tasks? Please refer to the below.

1.2.2 Implement

In OpenMMLab, implementing vision tasks commonly includes 3 parts (model / dataset / schedule). And just like that, implementing lightweight model also includes 3 parts (algorithm / dataset / schedule) in MMRazor.

`Algorithm` consist of `architecture` and `components`.

`Architecture` is similar to `model` of the upstream repos. You can chose to directly use the original `model` or customize the new `model` as your architecture according to different tasks. For example, you can directly use ResNet-34 and ResNet-18 of MMClassification to implement some KD algorithms, but in NAS, you may need to customize a searchable model.

`Components` consist of various special functions for supporting different lightweight algorithms. They can be directly used in config because of registered into MMEngine. Thus, you can pick some components you need to quickly implement your algorithm. For example, you may need `mutator` / `mutable` / `searchle backbone` if you want to implement a NAS algorithm, and you can pick from `distill loss` / `recorder` / `delivery` / `connector` if you need a KD algorithm.

Please refer to the next section for more details about **Implement**.

Note: The arg name of `algorithm` in config is **model** rather than **algorithm** in order to get better supports of MMCV and MMEngine.

1.3 Key concepts

For better understanding and using MMRazor, it is highly recommended to read the following user documents according to your own needs.

Global

- Algorithm

NAS & Pruning

- Mutator
- Mutable

KD

- Delivery
- Recorder

1.4 User guide

If you want to run mmrazor quickly, you can refer to as the follows.

- Learn about Configs
- Train different types algorithms
- Train with different devices
- Test a model

1.5 Tutorials

We provide the following general tutorials according to some typical requirements. If you want to further use MMRazor, you can refer to our source code and API Reference.

Tutorial list

- Customize Architectures
- Customize NAS algorithms
- Customize Pruning algorithms
- Customize KD algorithms
- Customize mixed algorithms
- Apply existing algorithms to new tasks

1.6 F&Q

If you encounter some trouble using MMRazor, you can find whether your question has existed in F&Q. If not existed, welcome to open a Github issue for getting support, we will reply it as soon.

1.7 Get support and contribute back

MMRazor is maintained on the MMRazor Github repository. We collect feedback and new proposals/ideas on Github. You can:

- Open a GitHub issue for bugs and feature requests.
- Open a pull request to contribute code (make sure to read the contribution guide before doing this).

CHAPTER

TWO

INSTALLATION

2.1 Prerequisites

In this section we demonstrate how to prepare an environment with PyTorch.

MMRazor works on Linux, Windows and macOS. It requires Python 3.6+, CUDA 9.2+ and PyTorch 1.8+.

Note: If you are experienced with PyTorch and have already installed it, just skip this part and jump to the next section. Otherwise, you can follow these steps for the preparation.

Step 0. Download and install Miniconda from the official website.

Step 1. Create a conda environment and activate it.

```
conda create --name openmmlab python=3.8 -y
conda activate openmmlab
```

Step 2. Install PyTorch following official instructions, e.g.

On GPU platforms:

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

On CPU platforms:

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

2.2 Installation

We recommend that users follow our best practices to install MMRazor.

2.2.1 Best Practices

Step 0. Install MMCV using MIM.

```
pip install -U openmim
mim install mmengine
mim install "mmcv>=2.0.0"
```

Step 1. Install MMRazor.

Case a: If you develop and run mmrazor directly, install it from source:

```
git clone -b main https://github.com/open-mmlab/mmrazor.git
cd mmrazor
pip install -v -e .
# '-v' means verbose, or more output
# '-e' means installing a project in editable mode,
# thus any local modifications made to the code will take effect without reinstallation.
```

Case b: If you use mmrazor as a dependency or third-party package, install it with pip:

```
pip install "mmrazor>=1.0.0"
```

CHAPTER

THREE

MODEL ZOO

3.1 Baselines

Type	Name	Link
nas	SPOS	README.md
nas	DARTS	README.md
nas	DetNAS	README.md
pruning	AutoSlim	README.md
pruning	L1-norm	README.md
pruning	Group Fisher	README.md
pruning	DMCP	README.md
ditill	ABLoss	README.md
ditill	BYOT	README.md
ditill	DAFL	README.md
ditill	DFAD	README.md
ditill	DKD	README.md
ditill	Factor Transfer	README.md
ditill	FitNets	README.md
ditill	KD	README.md
ditill	OFD	README.md
ditill	RKD	README.md
ditill	WSLD	README.md
ditill	ZSKT	README.md
ditill	CWD	README.md
ditill	FBKD	README.md
quantization	PTQ	README.md
quantization	QAT	README.md
quantization	LSQ	README.md

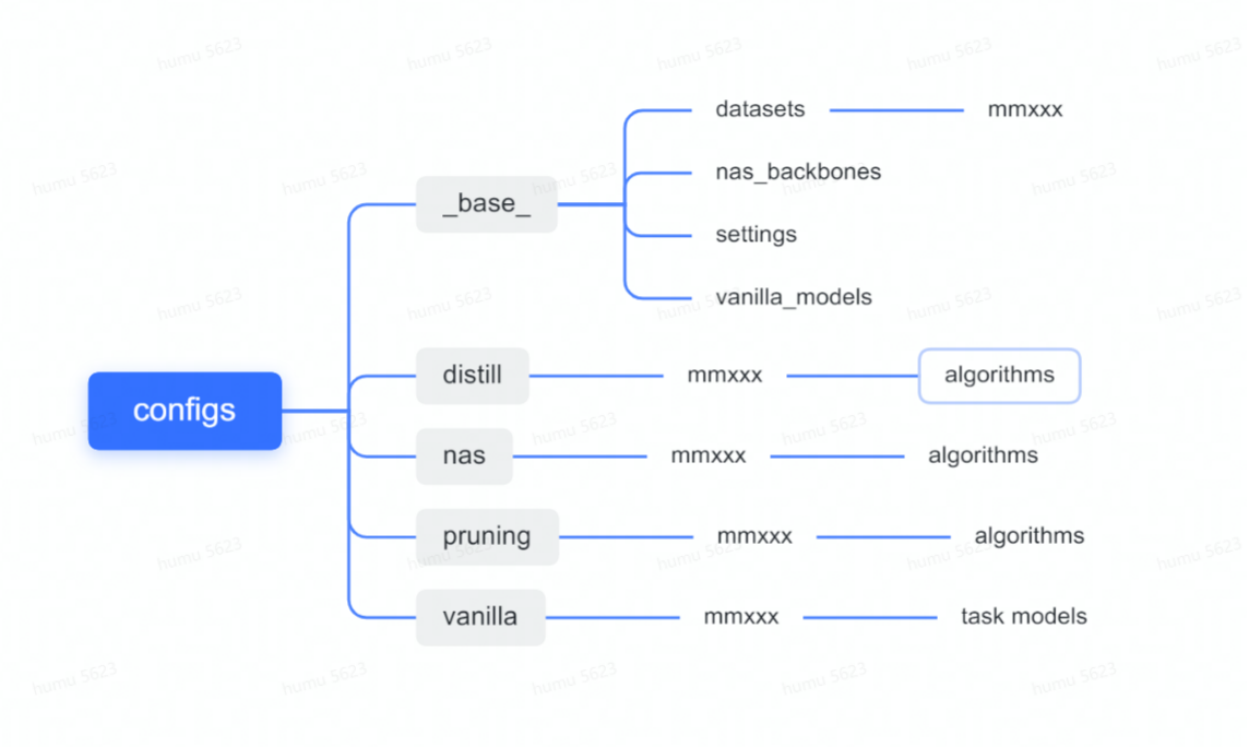
CHAPTER

FOUR

TRAIN & TEST

4.1 Learn about Configs

4.1.1 Directory structure of configs in mmrazor



`mmxxx`: means some task repositories of OpenMMLab, such mmcls, mmdet, mmseg and so on.

`_base_`: includes configures of datasets, experiment settings and model architectures.

`distill/nas/pruning`: model compression algorithms.

`vanilla`: task models owned by mmrazor.

4.1.2 More about config

Please refer to config in mmengine.

4.2 Train different types algorithms

Before running our algorithms, you may need to prepare the datasets according to the instructions in the corresponding document.

Note:

- With the help of mmengine, mmrazor unified entered interfaces for various tasks, thus our algorithms will adapt all OpenMMLab upstream repos in theory.
- 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 argument.

4.2.1 NAS

Here we take SPOS(Single Path One Shot) as an example. There are three steps to start neural network search(NAS), including **supernet pre-training**, **search for subnet on the trained supernet** and **subnet retraining**.

Supernet Pre-training

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

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

For example,

```
python ./tools/train.py \
  configs/nas/mmcls/spos/spos_shufflenet_supernet_8xb128_in1k.py
  --work-dir $WORK_DIR
```

Search for Subnet on The Trained Supernet

```
python tools/train.py ${CONFIG_FILE} --cfg-options load_from=${CHECKPOINT_PATH}␣
↪[optional arguments]
```

For example,

```
python ./tools/train.py \
  configs/nas/mmcls/spos/spos_shufflenet_search_8xb128_in1k.py \
  --cfg-options load_from=$STEP1_CKPT \
  --work-dir $WORK_DIR
```

Subnet Retraining

```
python tools/train.py ${CONFIG_FILE} \
    --cfg-options algorithm.fix_subnet=${MUTABLE_CFG_PATH} [optional arguments]
```

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

For example,

```
python ./tools/train.py \
  configs/nas/mmcls/spos/spos_shufflenet_subnet_8xb128_in1k.py \
  --work-dir $WORK_DIR \
  --cfg-options algorithm.fix_subnet=$YAML_FILE_BY_STEP2
```

We note that instead of using `--cfg-options`, you can also directly modify `configs/nas/mmcls/spos/ ``spos_shufflenet_subnet_8xb128_in1k``.py` like this:

```
fix_subnet = 'configs/nas/mmcls/spos/SPOS_SHUFFLENETV2_330M_IN1k_PAPER.yaml'
model = dict(fix_subnet=fix_subnet)
```

4.2.2 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.

Subnet Retraining

```
python tools/train.py ${CONFIG_FILE} --cfg-options model._channel_cfg_paths=${CHANNEL_
↪CFG_PATH} [optional arguments]
```

Different from NAS, the argument that needs to be specified here is `channel_cfg_paths` .

- `CHANNEL_CFG_PATH`: Path of `_channel_cfg_path`. `channel_cfg` represents **config for channel of the subnet searched out**, used to specify different subnets for testing.

For example, the default `_channel_cfg_paths` is set in the config below.

```
python ./tools/train.py \
  configs/pruning/mmcls/autoslim/autoslim_mbv2_1.5x_subnet_8xb256_in1k_flops-530M.py \
  --work-dir your_work_dir
```

4.2.3 Distillation

There is only one step to start knowledge distillation.

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

For example,

```
python ./tools/train.py \
  configs/distill/mmcls/kd/kd_logits_r34_r18_8xb32_in1k.py \
  --work-dir your_work_dir
```

4.3 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/dist_train.sh` even with 1 gpu, since some methods do not support non-distributed training.

4.3.1 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.

4.3.2 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.sh ${PARTITION} ${JOB_NAME} ${CONFIG_FILE} ${YOUR_WORK_DIR} [optional arguments]
```

4.3.3 Train with multiple machines

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

On the first machine:

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

On the second machine:

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

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

If you 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.

4.3.4 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/slurm_train.sh ${PARTITION} ${JOB_NAME}␣
↪config1.py tmp_work_dir_1
CUDA_VISIBLE_DEVICES=4,5,6,7 GPUS=4 sh tools/slurm_train.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/slurm_train.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/slurm_train.sh ${PARTITION} ${JOB_NAME}␣
↪config2.py tmp_work_dir_2 --cfg-options dist_params.port=29501
```

4.4 Test a model

4.4.1 NAS

To test nas method, you can use the following command.

```
python tools/test.py ${CONFIG_FILE} ${CHECKPOINT_PATH} --cfg-options algorithm.fix_
↪subnet=${FIX_SUBNET_PATH} [optional arguments]
```

- `FIX_SUBNET_PATH`: Path of `fix_subnet`. `fix_subnet` represents **config for mutable of the subnet searched out**, used to specify different subnets for testing. An example for `fix_subnet` can be found here.

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

For example,

```
python tools/test.py \
  configs/nas/mmcls/spos/spos_subnet_shufflenetv2_8xb128_in1k.py \
  your_subnet_checkpoint_path \
  --cfg-options algorithm.fix_subnet=configs/nas/mmcls/spos/SPOS_SHUFFLENETV2_330M_IN1k_
↪PAPER.yaml
```

4.4.2 Pruning

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,

```
python tools/model_converters/split_checkpoint.py \
  configs/pruning/autoslim/autoslim_mbv2_subnet_8xb256_in1k.py \
  your_retraining_checkpoint_path \
  --channel-cfgs configs/pruning/autoslim/AUTOSLIM_MBV2_530M_OFFICIAL.yaml configs/
↪pruning/autoslim/AUTOSLIM_MBV2_320M_OFFICIAL.yaml configs/pruning/autoslim/AUTOSLIM_
↪MBV2_220M_OFFICIAL.yaml
```

Test

To test pruning method, you can use following command

```
python tools/test.py ${CONFIG_FILE} ${CHECKPOINT_PATH} --cfg-options model._channel_cfg_
↪paths=${CHANNEL_CFG_PATH} [optional arguments]
```

- `task`: one of `mmclsmmdet` 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,

```
python ./tools/test.py \
  configs/pruning/mmcls/autoslim/autoslim_mbv2__1.5x_subnet_8xb256_in1k-530M.py \
  your_splitted_checkpoint_path --metrics accuracy
```

4.4.3 Distillation

To test the distillation method, you can use the following command

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

For example,

```
python ./tools/test.py \
  configs/distill/mmseg/cwd/cwd_logits_pspnet_r101_d8_pspnet_r18_d8_512x1024_cityscapes_
↪80k.py \
  your_splitted_checkpoint_path --show
```

CHAPTER

FIVE

QUANTIZATION

5.1 Quantization

5.1.1 Introduction

MMRazor's quantization is OpenMMLab's quantization toolkit, which has got through task models and model deployment. With its help, we can quantize and deploy pre-trained models in OpenMMLab to specified backend quickly. Of course, it can also contribute to implementing some custom quantization algorithms easier.

Major features

- **Ease of use**. Benefited from PyTorch fx, we can quantize our model without modifying the original model, but with user-friendly config.
- **Multiple backends deployment support**. Because of the specificity of each backend, a gap in performance usually exists between before and after deployment. We provided some common backend deployment support to reduce the gap as much.
- **Multiple task repos support.** Benefited from OpenMMLab 2.0, our quantization can support all task repos of OpenMMLab without extra code.
- **Be compatible with PyTorch's core module in quantization**. Some core modules in PyTorch can be used directly in mmrazor, such as `Observer`, `FakeQuantize`, `BackendConfig` and so on.

5.1.2 Quick run

Note: MMRazor's quantization is based on `torch==1.13`. Other requirements are the same as MMRazor's

Model quantization is in mmrazor, but quantized model deployment is in mmdeploy. So we need to the another branches as follows if we need to delopy our quantized model:

mmdeploy: https://github.com/open-mmlab/mmdeploy/tree/for_mmrazor

Note: If you try to compress mmdet's models and have used `dense_heads`, you can use this branch: https://github.com/HIT-cwh/mmdetection/tree/for_mmrazor to avoid the problem that some code can not be traced by `torch.fx.tracer`.

1. Quantize the float model in mmrazor.

```
# For QAT (Quantization Aware Training)
python tools/train.py ${CONFIG_PATH} [optional arguments]

# For PTQ (Post-training quantization)
python tools/ptq.py ${CONFIG_PATH} [optional arguments]
```

2. Evaluate the quantized model. (optional)

```
python tools/test.py ${CONFIG_PATH} ${CHECKPOINT_PATH}
```

3. Export quantized model to a specific backend in mmdeploy. (required by model deployment)

```
# MODEL_CFG_PATH is the used config in mmrazor.
python ./tools/deploy.py \
    ${DEPLOY_CFG_PATH} \
    ${MODEL_CFG_PATH} \
    ${MODEL_CHECKPOINT_PATH} \
    ${INPUT_IMG} \
    [optional arguments]
```

This step is the same as how to export an OpenMMLab model to a specific backend. For more details, please refer to How to convert model

4. Evaluate the quantized backend model. (optional)

```
python tools/test.py \
    ${DEPLOY_CFG} \
    ${MODEL_CFG} \
    --model ${BACKEND_MODEL_FILES} \
    [optional arguments]
```

This step is the same as evaluating backend models. For more details, please refer to How to evaluate model

5.1.3 How to quantize your own model quickly

If you want to try quantize your own model quickly, you just need to learn about how to change our provided config.

Case 1: If the model you want to quantize is in our provided configs.

You can refer to the previous chapter Quick Run.

Case 2: If the model you want to quantize is not in our provided configs.

Let us take `resnet50` as an example to show how to handle case 2.

```
_base_ = [
    'mmcls::resnet/resnet18_8xb32_in1k.py',
    '../../deploy_cfgs/mmcls/classification_openvino_dynamic-224x224.py'
]

val_dataloader = dict(batch_size=32)

test_cfg = dict(
    type='mmrazor.PTQLoop',
    calibrate_dataloader=val_dataloader,
```

(continues on next page)

(continued from previous page)

```
    calibrate_steps=32,
)

global_qconfig = dict(
    w_observer=dict(type='mmrazor.PerChannelMinMaxObserver'),
    a_observer=dict(type='mmrazor.MovingAverageMinMaxObserver'),
    w_fake_quant=dict(type='mmrazor.FakeQuantize'),
    a_fake_quant=dict(type='mmrazor.FakeQuantize'),
    w_qscheme=dict(
        qdtype='qint8', bit=8, is_symmetry=True, is_symmetric_range=True),
    a_qscheme=dict(
        qdtype='quint8', bit=8, is_symmetry=True, averaging_constant=0.1),
)

float_checkpoint = 'https://download.openmmlab.com/mmclassification/v0/resnet/resnet18_
↪8xb32_in1k_20210831-fbbb1da6.pth'  # noqa: E501

model = dict(
    _delete_=True,
    type='mmrazor.MMArchitectureQuant',
    data_preprocessor=dict(
        type='mmcls.ClsDataPreprocessor',
        num_classes=1000,
        # RGB format normalization parameters
        mean=[123.675, 116.28, 103.53],
        std=[58.395, 57.12, 57.375],
        # convert image from BGR to RGB
        to_rgb=True),
    architecture=_base_.model,
    deploy_cfg=_base_.deploy_cfg,
    float_checkpoint=float_checkpoint,
    quantizer=dict(
        type='mmrazor.OpenVINOQuantizer',
        global_qconfig=global_qconfig,
        tracer=dict(
            type='mmrazor.CustomTracer',
            skipped_methods=[
                'mmcls.models.heads.ClsHead._get_loss',
                'mmcls.models.heads.ClsHead._get_predictions'
            ])))

model_wrapper_cfg = dict(type='mmrazor.MMArchitectureQuantDDP', )
```

This is a config that quantize `resnet18` with OpenVINO backend. You just need to modify two args: `_base_` and `float_checkpoint`.

```
# before
_base_ = ['mmcls::resnet/resnet18_8xb32_in1k.py']
float_checkpoint = 'https://download.openmmlab.com/mmclassification/v0/resnet/resnet18_
↪8xb32_in1k_20210831-fbbb1da6.pth'

# after
```

(continues on next page)

(continued from previous page)

```
_base_ = ['mmcls::resnet/resnet50_8xb32_in1k.py']
float_checkpoint = 'https://download.openmmlab.com/mmclassification/v0/resnet/resnet50_
↪8xb32_in1k_20210831-ea4938fc.pth'
```

- `_base_` will be called from mmcls by mmengine, so you can just use mmcls provided configs directly. Other repos are similar.
- `float_checkpoint` is a pre-trained float checkpoint by OpenMMLab. You can find it in the corresponding repo.

After modifying required config, we can use it the same as case 1.

5.1.4 How to improve your quantization performance

If you can not be satisfied with quantization performance by applying our provided configs to your own model, you can try to improve it with our provided various quantization schemes by modifying `global_qconfig`.

```
global_qconfig = dict(
    w_observer=dict(type='mmrazor.PerChannelMinMaxObserver'),
    a_observer=dict(type='mmrazor.MovingAverageMinMaxObserver'),
    w_fake_quant=dict(type='mmrazor.FakeQuantize'),
    a_fake_quant=dict(type='mmrazor.FakeQuantize'),
    w_qscheme=dict(
        qdtype='qint8', bit=8, is_symmetry=True, is_symmetric_range=True),
    a_qscheme=dict(
        qdtype='quint8', bit=8, is_symmetry=True, averaging_constant=0.1),
)
```

As shown above, `global_qconfig` contains server common core args as follows:

- Observes

In `forward`, they will update the statistics of the observed Tensor. And they should provide a `calculate_qparams` function that computes the quantization parameters given the collected statistics.

Note: Whether it is per channel quantization depends on whether `PerChannel` is in the observer name.

Because mmrazor's quantization has been compatible with PyTorch's observers, we can use observers in PyTorch and our custom observers.

Supported observers list in Pytorch.

```
FixedQParamsObserver
HistogramObserver
MinMaxObserver
MovingAverageMinMaxObserver
MovingAveragePerChannelMinMaxObserver
NoopObserver
ObserverBase
PerChannelMinMaxObserver
PlaceholderObserver
RecordingObserver
```

(continues on next page)

(continued from previous page)

```
ReuseInputObserver
UniformQuantizationObserverBase
```

- Fake quants

In `forward`, they will update the statistics of the observed Tensor and fake quantize the input. They should also provide a `calculate_qparams` function that computes the quantization parameters given the collected statistics.

Because mmrazor's quantization has been compatible with PyTorch's fakequants, we can use fakequants in PyTorch and our custom fakequants.

Supported fakequants list in Pytorch.

```
FakeQuantize
FakeQuantizeBase
FixedQParamsFakeQuantize
FusedMovingAvgObsFakeQuantize
```

- Qschemes

Include some basic quantization configurations.

`qdtype`: to specify whether quantized data type is sign or unsign. It can be chosen from ['qint8', 'quint8']

Note: If your model need to be deployed, `qdtype` must be consistent with the dtype in the corresponding backend-config. Otherwise fakequant will not be inserted in front of the specified OPs.

backendconfigs dir: mmrazor/mmrazor/structures/quantization/backend_config

---

`bit`: to specify the quantized data bit. It can be chosen from [1 ~ 16].

`is_symmetry`: to specify whether to use symmetry quantization. It can be chosen from [True, False]

The specified qscheme is actually implemented by observers, so how to configurate other args needs to be based on the given observers, such as `is_symmetric_range` and `averaging_constant`.

5.1.5 How to customize your quantization algorithm

If you try to customize your quantization algorithm, you can refer to the following link for more details.

Customize Quantization algorithms

CHAPTER

SIX

USEFUL TOOLS

please refer to upstream applied repositories' docs

CHAPTER

SEVEN

KEY CONCEPTS

7.1 Algorithm

7.1.1 Introduction

What is algorithm in MMRazor

MMRazor is a model compression toolkit, which includes 4 mianstream technologies:

- Neural Architecture Search (NAS)
- Pruning
- Knowledge Distillation (KD)
- Quantization (come soon)

And in MMRazor, `algorithm` is a general item for these technologies. For example, in NAS,

SPOS is an `algorithm`, CWD is also an `algorithm` of knowledge distillation.

`algorithm` is the entrance of `mmrazor/models` . Its role in MMRazor is the same as both `classifier` in MMClassification and `detector` in MMDetection.

About base algorithm

In the directory of `models/algorithms`, all model compression algorithms are divided into 4 subdirectories: nas / pruning / distill / quantization. These algorithms must inherit from `BaseAlgorithm`, whose definition is as below.

```
from typing import Dict, List, Optional, Tuple, Union
from mmengine.model import BaseModel
from mmrazor.registry import MODELS

LossResults = Dict[str, torch.Tensor]
TensorResults = Union[Tuple[torch.Tensor], torch.Tensor]
PredictResults = List[BaseDataElement]
ForwardResults = Union[LossResults, TensorResults, PredictResults]

@MODELS.register_module()
class BaseAlgorithm(BaseModel):

    def __init__(self,
                 architecture: Union[BaseModel, Dict],
```

(continues on next page)

(continued from previous page)

```
                 data_preprocessor: Optional[Union[Dict, nn.Module]] = None,
                 init_cfg: Optional[Dict] = None):

        ......

        super().__init__(data_preprocessor, init_cfg)
        self.architecture = architecture

    def forward(self,
                batch_inputs: torch.Tensor,
                data_samples: Optional[List[BaseDataElement]] = None,
                mode: str = 'tensor') -> ForwardResults:

        if mode == 'loss':
            return self.loss(batch_inputs, data_samples)
        elif mode == 'tensor':
            return self._forward(batch_inputs, data_samples)
        elif mode == 'predict':
            return self._predict(batch_inputs, data_samples)
        else:
            raise RuntimeError(f'Invalid mode "{mode}". '
                               'Only supports loss, predict and tensor mode')

    def loss(
        self,
        batch_inputs: torch.Tensor,
        data_samples: Optional[List[BaseDataElement]] = None,
    ) -> LossResults:
        """Calculate losses from a batch of inputs and data samples."""
        return self.architecture(batch_inputs, data_samples, mode='loss')

    def _forward(
        self,
        batch_inputs: torch.Tensor,
        data_samples: Optional[List[BaseDataElement]] = None,
    ) -> TensorResults:
        """Network forward process."""
        return self.architecture(batch_inputs, data_samples, mode='tensor')

    def _predict(
        self,
        batch_inputs: torch.Tensor,
        data_samples: Optional[List[BaseDataElement]] = None,
    ) -> PredictResults:
        """Predict results from a batch of inputs and data samples with post-
        processing."""
        return self.architecture(batch_inputs, data_samples, mode='predict')
```

As you can see from above, `BaseAlgorithm` is inherited from `BaseModel` of MMEngine. `BaseModel` implements the basic functions of the algorithmic model, such as weights initialize,

batch inputs preprocess (see more information in `BaseDataPreprocessor` class of MMEngine), parse losses, and update model parameters. For more details of `BaseModel` , you can see docs for `BaseModel`.

`BaseAlgorithm`'s forward is just a wrapper of `BaseModel`'s forward. Sub-classes inherited from BaseAlgorithm only need to override the `loss` method, which implements the logic to calculate loss, thus various algorithms can be trained in the runner.

7.1.2 How to use existing algorithms in MMRazor

1. Configure your architecture that will be slimmed

- Use the model config of other repos of OpenMMLab directly as below, which is an example of setting Faster-RCNN as our architecture.

```
_base_ = [
    'mmdet::_base_/models/faster_rcnn_r50_fpn.py',
]

architecture = _base_.model
```

- Use your customized model as below, which is an example of defining a VGG model as our architecture.

Note: How to customize architectures can refer to our tutorial: Customize Architectures.

```
default_scope='mmcls'
architecture = dict(
    type='ImageClassifier',
    backbone=dict(type='VGG', depth=11, num_classes=1000),
    neck=None,
    head=dict(
        type='ClsHead',
        loss=dict(type='CrossEntropyLoss', loss_weight=1.0),
        topk=(1, 5),
    ))
```

2. Apply the registered algorithm to your architecture.

Note: The arg name of `algorithm` in config is **model** rather than **algorithm** in order to get better supports of MMCV and MMEngine.

Maybe more args in model need to set according to the used algorithm.

```
model = dict(
    type='BaseAlgorithm',
    architecture=architecture)
```

Note: About the usage of `Config`, refer to config.md please.

---

3. Apply some custom hooks or loops to your algorithm. (optional)

- Custom hooks

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

- Custom loops

```
_base_ = ['./spos_shufflenet_supernet_8xb128_in1k.py']

# To chose from ['train_cfg', 'val_cfg', 'test_cfg'] based on your loop type
train_cfg = dict(
    _delete_=True,
    type='mmrazor.EvolutionSearchLoop',
    dataloader=_base_.val_dataloader,
    evaluator=_base_.val_evaluator)

val_cfg = dict()
test_cfg = dict()
```

7.1.3 How to customize your algorithm

Common pipeline

1. Register a new algorithm

Create a new file `mmrazor/models/algorithms/{subdirectory}/xxx.py`

```
from mmrazor.models.algorithms import BaseAlgorithm
from mmrazor.registry import MODELS

@MODELS.register_module()
class XXX(BaseAlgorithm):
    def __init__(self, architecture):
        super().__init__(architecture)
        pass

    def loss(self, batch_inputs):
        pass
```

2. Rewrite its `loss` method.

```
from mmrazor.models.algorithms import BaseAlgorithm
from mmrazor.registry import MODELS

@MODELS.register_module()
class XXX(BaseAlgorithm):
    def __init__(self, architecture):
        super().__init__(architecture)
        ......

    def loss(self, batch_inputs):
        ......
        return LossResults
```

3. Add the remaining functions of the algorithm

Note: This step is special because of the diversity of algorithms. Some functions of the algorithm may also be implemented in other files.

```
from mmrazor.models.algorithms import BaseAlgorithm
from mmrazor.registry import MODELS

@MODELS.register_module()
class XXX(BaseAlgorithm):
    def __init__(self, architecture):
        super().__init__(architecture)
        ......

    def loss(self, batch_inputs):
        ......
        return LossResults

    def aaa(self):
        ......

    def bbb(self):
        ......
```

4. Import the class

You can add the following line to `mmrazor/models/algorithms/{subdirectory}/__init__.py`

```
from .xxx import XXX

__all__ = ['XXX']
```

In addition, import XXX in `mmrazor/models/algorithms/__init__.py`

5. Use the algorithm in your config file.

Please refer to the previous section about how to use existing algorithms in MMRazor

```
model = dict(
    type='XXX',
    architecture=architecture)
```

Pipelines for different algorithms

Please refer to our tutorials about how to customize different algorithms for more details as below.

1. NAS

Customize NAS algorithms

2. Pruning

Customize Pruning algorithms

3. Distill

Customize KD algorithms

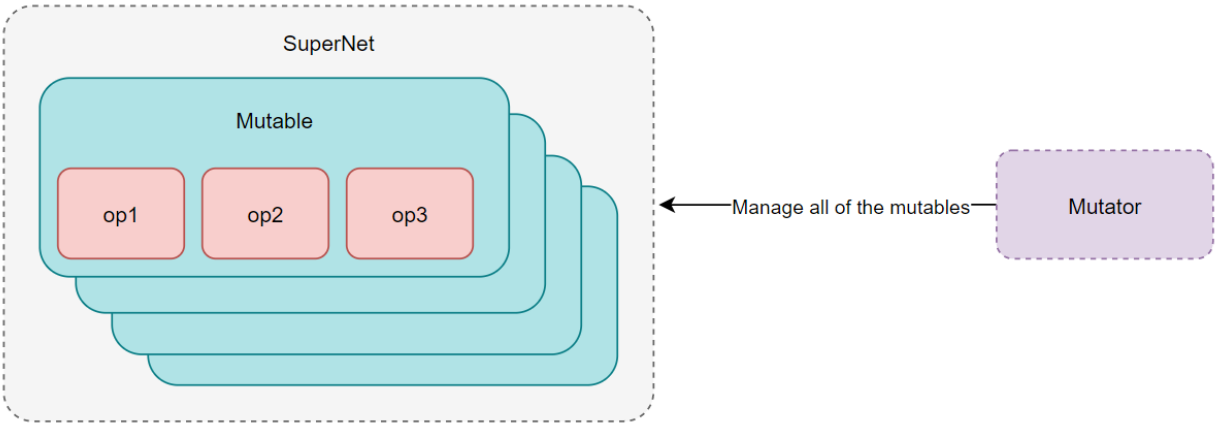
7.2 Mutator

7.2.1 Introduction

What is Mutator

Mutator is one of algorithm components, which provides some useful functions used for mutable management, such as sample choice, set choicet and so on. With Mutator's help, you can implement some NAS or pruning algorithms quickly.

What is the relationship between Mutator and Mutable

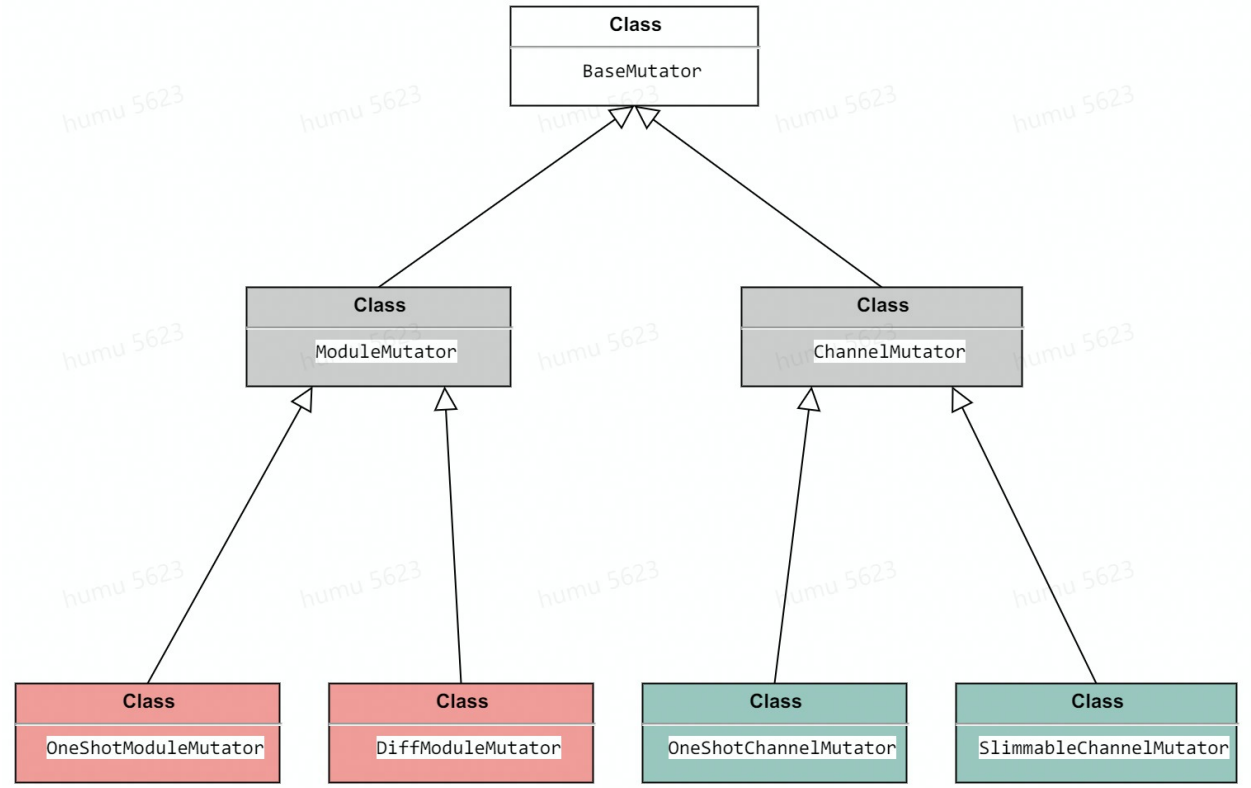


In a word, Mutator is the manager of Mutable. Each different type of mutable is commonly managed by their one correlative mutator, respectively.

As shown in the figure, Mutable is a component of supernet, therefore Mutator can implement some functions about subnet from supernet by handling Mutable.

Supported mutators

In MMRazor, we have implemented some mutators, their relationship is as below.



`BaseMutator`: Base class for all mutators. It has appointed some abstract methods supported by all mutators.

`ModuleMuator`/ `ChannelMutator`: Two different types mutators are for handling mutable module and mutable channel respectively.

Note: Please refer to Mutable for more details about different types of mutable.

`OneShotModuleMutator` / `DiffModuleMutator`: Inherit from `ModuleMuator`, they are for implementing different types algorithms, such as SPOS, Darts and so on.

`OneShotChannelMutator` / `SlimmableChannelMutator`: Inherit from `ChannelMutator`, they are also for meeting the needs of different types algorithms, such as AotuSlim.

7.2.2 How to use existing mutators

You just use them directly in configs as below

```
supernet = dict(
    ...
    )

model = dict(
    type='mmrazor.SPOS',
```

(continues on next page)

(continued from previous page)

```
    architecture=supernet,
    mutator=dict(type='mmrazor.OneShotModuleMutator'))
```

If existing mutators do not meet your needs, you can also customize your needed mutator.

7.2.3 How to customize your mutator

All mutators need to implement at least two of the following interfaces

- `prepare_from_supernet()`
  - Make some necessary preparations according to the given supernet. These preparations may include, but are not limited to, grouping the search space, and initializing mutator with the parameters needed for itself.
- `search_groups`
  - Group of search space.
  - Note that **search groups** and **search space** are two different concepts. The latter defines what choices can be used for searching. The former groups the search space, and searchable blocks that are grouped into the same group will share the same search space and the same sample result.
  - ```
    # Example
    search_space = {op1, op2, op3, op4}
    search_group = {0: [op1, op2], 1: [op3, op4]}
    ```

There are 4 steps to implement a custom mutator.

1. Registry a new mutator
2. Implement abstract methods
3. Implement other methods
4. Import the class

Then you can use your customized mutator in configs as in the previous chapter.

Let's use `OneShotModuleMutator` as an example for customizing mutator.

1.Registry a new mutator

First, you need to determine which type mutator to implement. Thus, you can implement your mutator faster by inheriting from correlative base mutator.

Then create a new file `mmrazor/models/mutators/module_mutator/one_shot_module_mutator`, class `OneShotModuleMutator` inherits from `ModuleMutator`.

```
from mmrazor.registry import MODELS
from .module_mutator import ModuleMutator

@MODELS.register_module()
class OneShotModuleMutator(ModuleMutator):
    ...
```
```

2. Implement abstract methods

2.1. Rewrite the `mutable_class_type` property

```
@MODELS.register_module()
class OneShotModuleMutator(ModuleMutator):

     @property
    def mutable_class_type(self):
        """One-shot mutable class type.
        Returns:
            Type[OneShotMutableModule]: Class type of one-shot mutable.
        """
        return OneShotMutableModule
```

2.2. Rewrite `search_groups` and `prepare_from_supernet()`

As the `prepare_from_supernet()` method and the `search_groups` property are already implemented in the `ModuleMutator` and we don't need to add our own logic, the second step is already over.

If you need to implement them by yourself, you can refer to these as follows.

2.3. **Understand `search_groups` (optional)**

Let's take an example to see what default `search_groups` do.

```
from mmrazor.models import OneShotModuleMutator, OneShotMutableModule

class SearchableModel(nn.Module):
    def __init__(self, one_shot_op_cfg):
        # assume `OneShotMutableModule` contains 4 choices:
        # choice1, choice2, choice3 and choice4
        self.choice_block1 = OneShotMutableModule(**one_shot_op_cfg)
        self.choice_block2 = OneShotMutableModule(**one_shot_op_cfg)
        self.choice_block3 = OneShotMutableModule(**one_shot_op_cfg)

    def forward(self, x: Tensor) -> Tensor:
        x = self.choice_block1(x)
        x = self.choice_block2(x)
        x = self.choice_block3(x)

        return x

supernet = SearchableModel(one_shot_op_cfg)
mutator1 = OneShotModuleMutator()
# build mutator1 from supernet.
mutator1.prepare_from_supernet(supernet)
>>> mutator1.search_groups.keys()
dict_keys([0, 1, 2])
```

In this case, each `OneShotMutableModule` will be divided into a group. Thus, the search groups have 3 groups.

If you want to custom group according to your requirement, you can implement it by passing the arg `custom_group`.

```
custom_group = [
    ['op1', 'op2'],
```

(continues on next page)

(continued from previous page)

```
    ['op3']
]
mutator2 = OneShotMutator(custom_group)
mutator2.prepare_from_supernet(supernet)
```

Then choice_block1 and choice_block2 will share the same search space and the same sample result, and choice_block3 will have its own independent search space. Thus, the search groups have only 2 groups.

```
>>> mutator2.search_groups.keys()
dict_keys([0, 1])
```

3. Implement other methods

After finishing some required methods, we need to add some special methods, such as sample_choices and set_choices.

```
from typing import Any, Dict

from mmrazor.registry import MODELS
from ...mutables import OneShotMutableModule
from .module_mutator import ModuleMutator

@MODELS.register_module()
class OneShotModuleMutator(ModuleMutator):

    def sample_choices(self) -> Dict[int, Any]:
        """Sampling by search groups.
        The sampling result of the first mutable of each group is the sampling
        result of this group.
        Returns:
            Dict[int, Any]: Random choices dict.
        """
        random_choices = dict()
        for group_id, modules in self.search_groups.items():
            random_choices[group_id] = modules[0].sample_choice()

        return random_choices

    def set_choices(self, choices: Dict[int, Any]) -> None:
        """Set mutables' current choice according to choices sample by
        :func:`sample_choices`.
        Args:
            choices (Dict[int, Any]): Choices dict. The key is group_id in
                search groups, and the value is the sampling results
                corresponding to this group.
        """
        for group_id, modules in self.search_groups.items():
            choice = choices[group_id]
            for module in modules:
                module.current_choice = choice
```

(continues on next page)

(continued from previous page)

```
    @property
    def mutable_class_type(self):
        """One-shot mutable class type.
        Returns:
            Type[OneShotMutableModule]: Class type of one-shot mutable.
        """
        return OneShotMutableModule
```

4. Import the class

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

```
from .one_shot_module_mutator import OneShotModuleMutator

__all__ = ['OneShotModuleMutator']
```

or alternatively add

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

to the config file to avoid modifying the original code.

Customize `OneShotModuleMutator` is over, then you can use it directly in your algorithm.

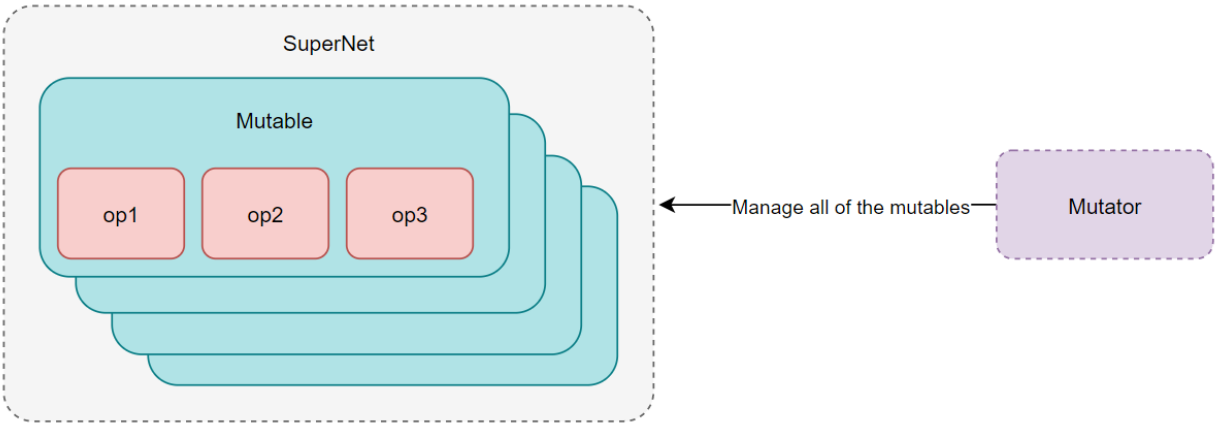
7.3 Mutable

7.3.1 Introduction

What is Mutable

`Mutable` is one of basic function components in NAS algorithms and some pruning algorithms, which makes supernet searchable by providing optional modules or parameters.

To understand it better, we take the mutable module as an example to explain as follows.



As shown in the figure above, `Mutable` is a container that holds some candidate operations, thus it can sample candidates to constitute the subnet. `Supernet` usually consists of multiple `Mutable`, therefore, `Supernet` will be searchable with the help of `Mutable`. And all candidate operations in `Mutable` constitute the search space of `SuperNet`.

Note: If you want to know more about the relationship between Mutable and Mutator, please refer to Mutator

---

Features

1. Support module mutable

It is the common and basic function for NAS algorithms. We can use it to implement some classical one-shot NAS algorithms, such as SPOS, DetNAS and so on.

2. Support parameter mutable

To implement more complicated and funny algorithms easier, we supported making some important parameters searchable, such as input channel, output channel, kernel size and so on.

What is more, we can implement **dynamic op** by using mutable parameters.

3. Support deriving from mutable parameter

Because of the restriction of defined architecture, there may be correlations between some mutable parameters, **such as concat and expand ratio.**

Note: If conv3 = concat (conv1, conv2)

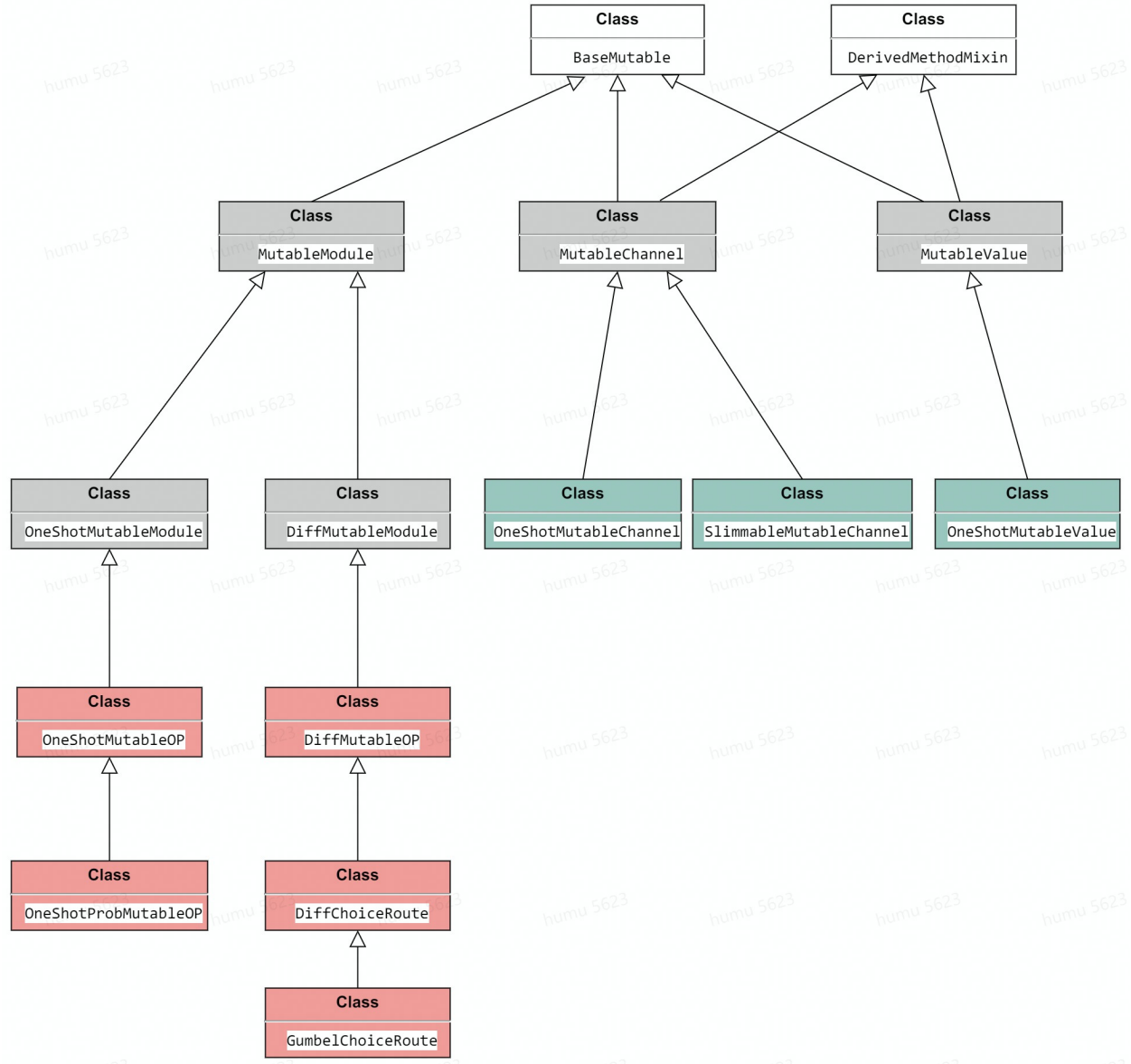
When out_channel (conv1) = 3, out_channel (conv2) = 4

Then in_channel (conv3) must be 7 rather than mutable.

So use derived mutable from conv1 and conv2 to generate in_channel (conv3)

With the help of derived mutable, we can meet these special requirements in some NAS algorithms and pruning algorithms. What is more, it can be used to deal with different granularity between search spaces.

Supported mutables



As shown in the figure above.

- **White blocks** stand the basic classes, which include `BaseMutable` and `DerivedMethodMixin`. `BaseMutable` is the base class for all mutables, which defines required properties and abstracmethods. `DerivedMethodMixin` is a mixin class to provide mutable parameters with some useful methods to derive mutable.
- **Gray blocks** stand different types of base mutables.

Note: Because there are correlations between channels of some layers, we divide mutable parameters into `MutableChannel` and `MutableValue`, so you can also think `MutableChannel` is a special `MutableValue`.

For supporting module and parameters mutable, we provide `MutableModule`, `MutableChannel` and `MutableValue` these base classes to implement required basic functions. And we also add

`OneshotMutableModule` and `DiffMutableModule` two types based on `MutableModule` to meet different types of algorithms' requirements.

For supporting deriving from mutable parameters, we make `MutableChannel` and `MutableValue` inherit from `BaseMutable` and `DerivedMethodMixin`, thus they can get derived functions provided by `DerivedMethodMixin`.

- **Red blocks** and **green blocks** stand registered classes for implementing some specific algorithms, which means that you can use them directly in configs. If they do not meet your requirements, you can also customize your mutable based on our base classes. If you are interested in their realization, please refer to their docstring.

7.3.2 How to use existing mutables to configure searchable backbones

We will use `OneShotMutableOP` to build a `SearchableShuffleNetV2` backbone as follows.

1. Configure needed mutables

```
# we only use OneShotMutableOP, then take 4 ShuffleOP as its candidates.
_STAGE_MUTABLE = dict(
    _scope_='mmrazor',
    type='OneShotMutableOP',
    candidates=dict(
        shuffle_3x3=dict(type='ShuffleBlock', kernel_size=3),
        shuffle_5x5=dict(type='ShuffleBlock', kernel_size=5),
        shuffle_7x7=dict(type='ShuffleBlock', kernel_size=7),
        shuffle_xception=dict(type='ShuffleXception')))
```

2. Configure the `arch_setting` of `SearchableShuffleNetV2`

```
# Use the _STAGE_MUTABLE in various stages.
arch_setting = [
    # Parameters to build layers. 3 parameters are needed to construct a
    # layer, from left to right: channel, num_blocks, mutable_cfg.
    [64, 4, _STAGE_MUTABLE],
    [160, 4, _STAGE_MUTABLE],
    [320, 8, _STAGE_MUTABLE],
    [640, 4, _STAGE_MUTABLE]
]
```

3. Configure searchable backbone.

```
nas_backbone = dict(
    _scope_='mmrazor',
    type='SearchableShuffleNetV2',
    widen_factor=1.0,
    arch_setting=arch_setting)
```

Then you can use it in your architecture. If existing mutables do not meet your needs, you can also customize your needed mutable.

7.3.3 How to customize your mutable

About base mutable

Before customizing mutables, we need to know what some base mutables do.

BaseMutable

In order to implement the searchable mechanism, mutables need to own some base functions, such as changing status from mutable to fixed, recording the current status and current choice and so on. So in `BaseMutable`, these relevant abstract methods and properties will be defined as follows.

```python
# Copyright (c) OpenMMLab. All rights reserved.
from abc import ABC, abstractmethod
from typing import Dict, Generic, Optional, TypeVar

from mmengine.model import BaseModule

CHOICE_TYPE = TypeVar('CHOICE_TYPE')
CHOSEN_TYPE = TypeVar('CHOSEN_TYPE')

class BaseMutable(BaseModule, ABC, Generic[CHOICE_TYPE, CHOSEN_TYPE]):

    def __init__(self,
                 alias: Optional[str] = None,
                 init_cfg: Optional[Dict] = None) -> None:
        super().__init__(init_cfg=init_cfg)

        self.alias = alias
        self._is_fixed = False
        self._current_choice: Optional[CHOICE_TYPE] = None

    @property
    def current_choice(self) -> Optional[CHOICE_TYPE]:
        return self._current_choice

    @current_choice.setter
    def current_choice(self, choice: Optional[CHOICE_TYPE]) -> None:
        self._current_choice = choice

    @property
    def is_fixed(self) -> bool:
        return self._is_fixed

    @is_fixed.setter
    def is_fixed(self, is_fixed: bool) -> None:
        ......
        self._is_fixed = is_fixed

    @abstractmethod
    def fix_chosen(self, chosen: CHOSEN_TYPE) -> None:
       pass

    @abstractmethod
```

(continues on next page)

(continued from previous page)

```
    def dump_chosen(self) -> CHOSEN_TYPE:
        pass

    @property
    @abstractmethod
    def num_choices(self) -> int:
        pass
```

MutableModule

Inherited from `BaseModule`, `MutableModule` not only owns its basic functions, but also needs some specialized functions to implement module mutable, such as getting all choices, executing forward computation.

```
# Copyright (c) OpenMMLab. All rights reserved.
from abc import abstractmethod
from typing import Any, Dict, List, Optional

from ..base_mutable import CHOICE_TYPE, CHOSEN_TYPE, BaseMutable

class MutableModule(BaseMutable[CHOICE_TYPE, CHOSEN_TYPE]):

    def __init__(self,
                 module_kwargs: Optional[Dict[str, Dict]] = None,
                 **kwargs) -> None:
        super().__init__(**kwargs)

        self.module_kwargs = module_kwargs

    @property
    @abstractmethod
    def choices(self) -> List[CHOICE_TYPE]:
        """list: all choices.  All subclasses must implement this method."""

    @abstractmethod
    def forward(self, x: Any) -> Any:
        """Forward computation."""

    @property
    def num_choices(self) -> int:
        """Number of choices."""
        return len(self.choices)
```

If you want to know more about other types mutables, please refer to their docstring.

Steps of customizing mutables

There are 4 steps to implement a custom mutable.

1. Registry a new mutable
2. Implement abstract methods.
3. Implement other methods.
4. Import the class

Then you can use your customized mutable in configs as in the previous chapter.

Let's use `OneShotMutableOP` as an example for customizing mutable.

1. Registry a new mutable

First, you need to determine which type mutable to implement. Thus, you can implement your mutable faster by inheriting from correlative base mutable.

Then create a new file `mmrazor/models/mutables/mutable_module/one_shot_mutable_module`, class `OneShotMutableOP` inherits from `OneShotMutableModule`.

```
# Copyright (c) OpenMMLab. All rights reserved.
import random
from abc import abstractmethod
from typing import Any, Dict, List, Optional, Union

import numpy as np
import torch.nn as nn
from torch import Tensor

from mmrazor.registry import MODELS
from ..base_mutable import CHOICE_TYPE, CHOSEN_TYPE
from .mutable_module import MutableModule

@MODELS.register_module()
class OneShotMutableOP(OneShotMutableModule[str, str]):
    ...
```

2. Implement abstract methods

2.1 Basic abstract methods

These basic abstract methods are mainly from `BaseMutable` and `MutableModule`, such as `fix_chosen`, `dump_chosen`, `choices` and `num_choices`.

```
@MODELS.register_module()
class OneShotMutableOP(OneShotMutableModule[str, str]):
    ......

    def fix_chosen(self, chosen: str) -> None:
        """Fix mutable with subnet config. This operation would convert
```

(continues on next page)

(continued from previous page)

```
        `unfixed` mode to `fixed` mode. The :attr:`is_fixed` will be set to
        True and only the selected operations can be retained.
        Args:
            chosen (str): the chosen key in ``MUTABLE``. Defaults to None.
        """
        if self.is_fixed:
            raise AttributeError(
                'The mode of current MUTABLE is `fixed`. '
                'Please do not call `fix_chosen` function again.')

        for c in self.choices:
            if c != chosen:
                self._candidates.pop(c)

        self._chosen = chosen
        self.is_fixed = True

    def dump_chosen(self) -> str:
        assert self.current_choice is not None

        return self.current_choice

    @property
    def choices(self) -> List[str]:
        """list: all choices. """
        return list(self._candidates.keys())

    @property
    def num_choices(self):
        return len(self.choices)
```

2.2 Specified abstract methods

In `OneShotMutableModule`, sample and forward these required abstract methods are defined, such as `sample_choice`, `forward_choice`, `forward_fixed`, `forward_all`. So we need to implement these abstract methods.

```
@MODELS.register_module()
class OneShotMutableOP(OneShotMutableModule[str, str]):
    ......

    def sample_choice(self) -> str:
        """uniform sampling."""
        return np.random.choice(self.choices, 1)[0]

    def forward_fixed(self, x: Any) -> Tensor:
        """Forward with the `fixed` mutable.
        Args:
            x (Any): x could be a Torch.tensor or a tuple of
                Torch.tensor, containing input data for forward computation.
        Returns:
```

(continues on next page)

(continued from previous page)

```
            Tensor: the result of forward the fixed operation.
        """
        return self._candidates[self._chosen](x)

    def forward_choice(self, x: Any, choice: str) -> Tensor:
        """Forward with the `unfixed` mutable and current choice is not None.
        Args:
            x (Any): x could be a Torch.tensor or a tuple of
                Torch.tensor, containing input data for forward computation.
            choice (str): the chosen key in ``OneShotMutableOP``.
        Returns:
            Tensor: the result of forward the ``choice`` operation.
        """
        assert isinstance(choice, str) and choice in self.choices
        return self._candidates[choice](x)

    def forward_all(self, x: Any) -> Tensor:
        """Forward all choices. Used to calculate FLOPs.
        Args:
            x (Any): x could be a Torch.tensor or a tuple of
                Torch.tensor, containing input data for forward computation.
        Returns:
            Tensor: the result of forward all of the ``choice`` operation.
        """
        outputs = list()
        for op in self._candidates.values():
            outputs.append(op(x))
        return sum(outputs)
```

3. Implement other methods

After finishing some required methods, we need to add some special methods, such as `_build_ops`, because it is needed in building candidates for sampling.

```
@MODELS.register_module()
class OneShotMutableOP(OneShotMutableModule[str, str]):
    ......

    @staticmethod
    def _build_ops(
            candidates: Union[Dict[str, Dict], nn.ModuleDict],
            module_kwargs: Optional[Dict[str, Dict]] = None) -> nn.ModuleDict:
        """Build candidate operations based on choice configures.
        Args:
            candidates (dict[str, dict] | :obj:`nn.ModuleDict`): the configs
                for the candidate operations or nn.ModuleDict.
            module_kwargs (dict[str, dict], optional): Module initialization
                named arguments.
        Returns:
            ModuleDict (dict[str, Any], optional):  the key of ``ops`` is
                the name of each choice in configs and the value of ``ops``
```

(continues on next page)

(continued from previous page)

```
                is the corresponding candidate operation.
        """
        if isinstance(candidates, nn.ModuleDict):
            return candidates

        ops = nn.ModuleDict()
        for name, op_cfg in candidates.items():
            assert name not in ops
            if module_kwargs is not None:
                op_cfg.update(module_kwargs)
            ops[name] = MODELS.build(op_cfg)
        return ops
```

4. Import the class

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

```
from .one_shot_mutable_module import OneShotMutableModule

__all__ = ['OneShotMutableModule']
```

or alternatively add

```
custom_imports = dict(
    imports=['mmrazor.models.mutables.mutable_module.one_shot_mutable_module'],
    allow_failed_imports=False)
```

to the config file to avoid modifying the original code.

Customize `OneShotMutableOP` is over, then you can use it directly in your algorithm.

7.4 Recorder

7.4.1 Introduction of Recorder

`Recorder` is a context manager used to record various intermediate results during the model forward. It can help `Delivery` finish data delivering by recording source data in some distillation algorithms. And it can also be used to obtain some specific data for visual analysis or other functions you want.

To adapt to more requirements, we implement multiple types of recorders to obtain different types of intermediate results in MMRazor. What is more, they can be used in combination with the `RecorderManager`.

In general, `Recorder` will help us expand more functions in implementing algorithms by recording various intermediate results.

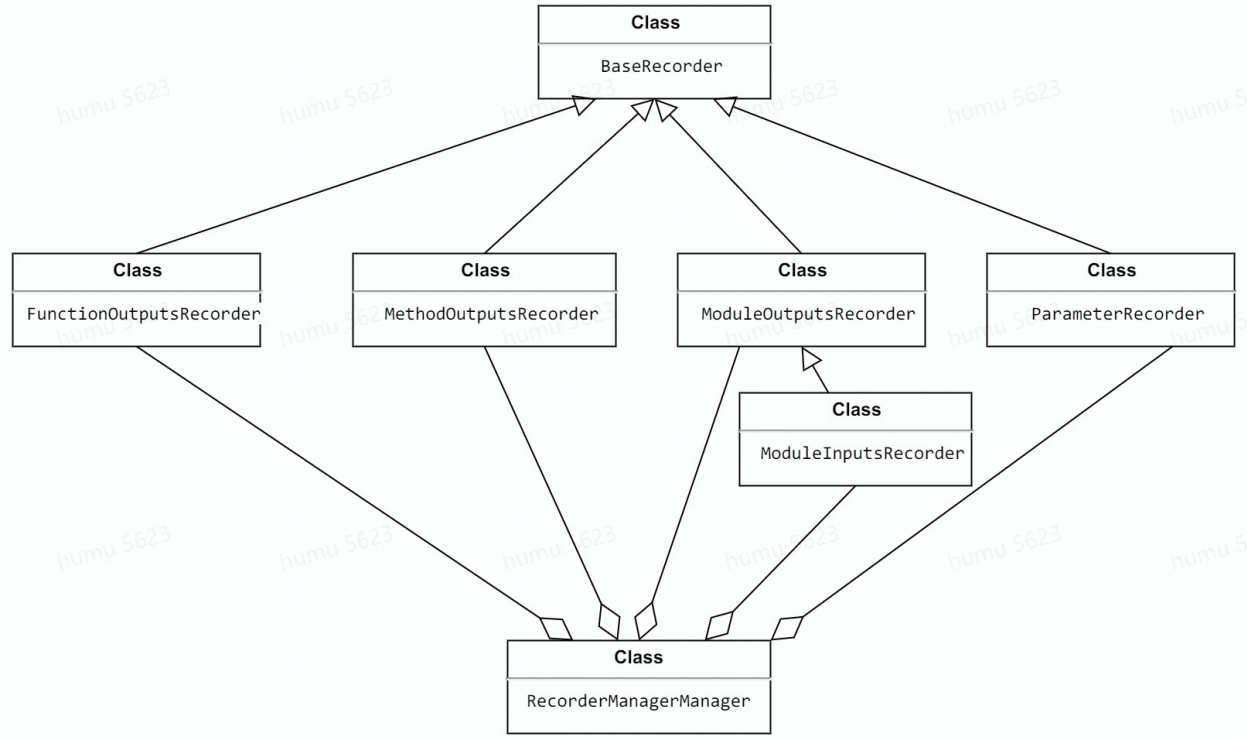
7.4.2 Usage of Recorder

Currently, we support five `Recorder`, as shown in the following table

Recorder name	Description
FunctionOutputsRecorder	Record output results of some functions
MethodOutputsRecorder	Record output results of some methods
ModuleInputsRecorder	Record input results of nn.Module
ModuleOutputsRecorder	Record output results of nn.Module
ParameterRecorder	Record intermediate parameters of nn.Module

All of the recorders inherit from `BaseRecorder`. And these recorders can be managed by `RecorderManager` or just be used on their own.

Their relationship is shown below.



FunctionOutputsRecorder

`FunctionOutputsRecorder` is used to record the output results of intermediate **function**.

Note: When instantiating `FunctionOutputsRecorder`, you need to pass `source` argument, which requires extra attention. For example, `anchor_inside_flags` is a function in mmdetection to check whether the anchors are inside the border. This function is in `mmdet/core/anchor/utils.py` and used in `mmdet/models/dense_heads/anchor_head`. Then the `source` argument should be `mmdet.models.dense_heads.anchor_head.anchor_inside_flags` but not `mmdet.core.anchor.utils.anchor_inside_flags`.

Example

Suppose there is a toy function named `toy_func` in toy_module.py.

```
import random
from typing import List
from mmrazor.structures import FunctionOutputsRecorder

def toy_func() -> int:
    return random.randint(0, 1000000)

# instantiate with specifying used path
r1 = FunctionOutputsRecorder('toy_module.toy_func')

# initialize is to make specified module can be recorded by
# registering customized forward hook.
r1.initialize()
with r1:
    out1 = toy_module.toy_func()
    out2 = toy_module.toy_func()
    out3 = toy_module.toy_func()

# check recorded data
print(r1.data_buffer)
```

Out:

```
[75486, 641059, 119729]
```

Test Correctness of recorded results

```
data_buffer = r1.data_buffer
print(data_buffer[0] == out1 and data_buffer[1] == out2 and data_buffer[2] == out3)
```

Out:

```
True
```

To get the specific recorded data with `get_record_data`

```
print(r1.get_record_data(record_idx=2))
```

Out:

```
119729
```

MethodOutputsRecorder

`MethodOutputsRecorder` is used to record the output results of intermediate **method**.

Example

Suppose there is a toy class `Toy` and it has a toy method `toy_func` in toy_module.py.

```
import random
from mmrazor.core import MethodOutputsRecorder

class Toy():
    def toy_func(self):
        return random.randint(0, 1000000)

toy = Toy()

# instantiate with specifying used path
r1 = MethodOutputsRecorder('toy_module.Toy.toy_func')
# initialize is to make specified module can be recorded by
# registering customized forward hook.
r1.initialize()

with r1:
    out1 = toy.toy_func()
    out2 = toy.toy_func()
    out3 = toy.toy_func()

# check recorded data
print(r1.data_buffer)
```

Out:

```
[217832, 353057, 387699]
```

Test Correctness of recorded results

```
data_buffer = r1.data_buffer
print(data_buffer[0] == out1 and data_buffer[1] == out2 and data_buffer[2] == out3)
```

Out:

```
True
```

To get the specific recorded data with `get_record_data`

```
print(r1.get_record_data(record_idx=2))
```

Out:

```
387699
```

ModuleOutputsRecorder and ModuleInputsRecorder

`ModuleOutputsRecorder`'s usage is similar with `ModuleInputsRecorder`'s, so we will take the former as an example to introduce their usage.

Example

Note:

Different `MethodOutputsRecorder` and `FunctionOutputsRecorder`, `ModuleOutputsRecorder` and `ModuleInputsRecorder` are instantiated with module name rather than used path, and executing `initialize` need arg: `model`. Thus, they can know actually the module needs to be recorded.

Suppose there is a toy Module `ToyModule` in toy_module.py.

```
import torch
from torch import nn
from mmrazor.core import ModuleOutputsRecorder

class ToyModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 1, 1)
        self.conv2 = nn.Conv2d(1, 1, 1)

    def forward(self, x):
        x1 = self.conv1(x)
        x2 = self.conv1(x + 1)
        return self.conv2(x1 + x2)

model = ToyModel()
# instantiate with specifying module name.
r1 = ModuleOutputsRecorder('conv1')

# initialize is to make specified module can be recorded by
# registering customized forward hook.
r1.initialize(model)

x = torch.randn(1, 1, 1, 1)
with r1:
    out = model(x)

print(r1.data_buffer)
```

Out:

```
[tensor([[[[0.0820]]]], grad_fn=<ThnnConv2DBackward0>), tensor([[[[-0.0894]]]], grad_fn=
↪<ThnnConv2DBackward0>)]
```

Test Correctness of recorded results

```
print(torch.equal(r1.data_buffer[0], model.conv1(x)))
print(torch.equal(r1.data_buffer[1], model.conv1(x + 1)))
```

Out:

```
True
True
```

ParameterRecorder

`ParameterRecorder` is used to record the intermediate parameter of `nn.Module`. Its usage is similar to `ModuleOutputsRecorder`'s and `ModuleInputsRecorder`'s, but it instantiates with parameter name instead of module name.

Example

Suppose there is a toy Module `ToyModule` in toy_module.py.

```
from torch import nn
import torch
from mmrazor.core import ModuleOutputsRecorder

class ToyModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.toy_conv = nn.Conv2d(1, 1, 1)

    def forward(self, x):
        return self.toy_conv(x)

model = ToyModel()
# instantiate with specifying parameter name.
r1 = ParameterRecorder('toy_conv.weight')
# initialize is to make specified module can be recorded by
# registering customized forward hook.
r1.initialize(model)

print(r1.data_buffer)
```

Out:

```
[Parameter containing: tensor([[[[0.2971]]]], requires_grad=True)]
```

Test Correctness of recorded results

```
print(torch.equal(r1.data_buffer[0], model.toy_conv.weight))
```

Out:

```
True
```

RecorderManager

`RecorderManager` is actually context manager, which can be used to manage various types of recorders.

With the help of `RecorderManager`, we can manage several different recorders with as little code as possible, which reduces the possibility of errors.

Example

Suppose there is a toy class `Toy` owned has a toy method `toy_func` in toy_module.py.

```
import random
from torch import nn
from mmrazor.core import RecorderManager

class Toy():
    def toy_func(self):
        return random.randint(0, 1000000)

class ToyModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 1, 1)
        self.conv2 = nn.Conv2d(1, 1, 1)
        self.toy = Toy()

    def forward(self, x):
        return self.conv2(self.conv1(x)) + self.toy.toy_func()

# configure multi-recorders
conv1_rec = ConfigDict(type='ModuleOutputs', source='conv1')
conv2_rec = ConfigDict(type='ModuleOutputs', source='conv2')
func_rec = ConfigDict(type='MethodOutputs', source='toy_module.Toy.toy_func')
# instantiate RecorderManager with a dict that contains recorders' configs,
# you can customize their keys.
manager = RecorderManager(
    {'conv1_rec': conv1_rec,
     'conv2_rec': conv2_rec,
     'func_rec': func_rec})

model = ToyModel()
# initialize is to make specified module can be recorded by
# registering customized forward hook.
manager.initialize(model)

x = torch.rand(1, 1, 1, 1)
with manager:
    out = model(x)

conv2_out = manager.get_recorder('conv2_rec').get_record_data()
print(conv2_out)
```

Out:

```
tensor([[[[0.5543]]]], grad_fn=<ThnnConv2DBackward0>)
```

Display output of `toy_func`

```
func_out = manager.get_recorder('func_rec').get_record_data()
print(func_out)
```

Out:

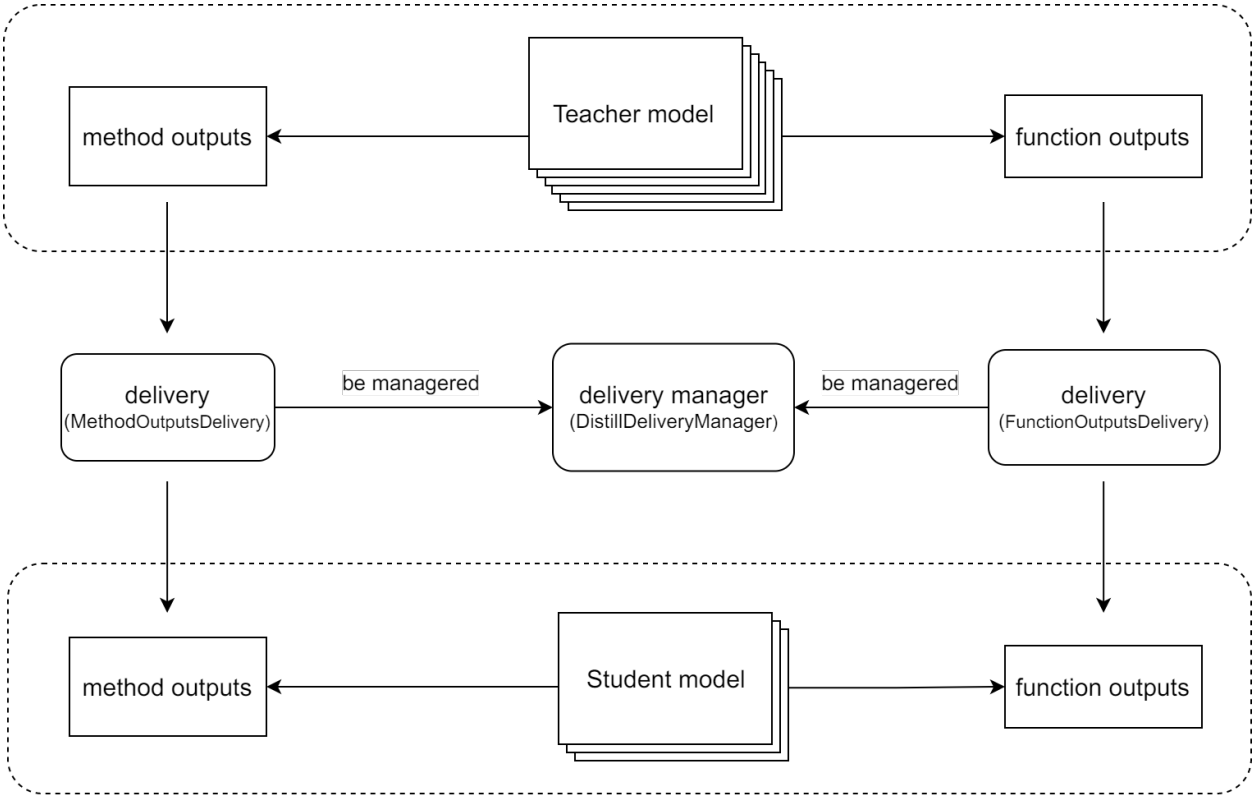
```
313167
```

7.5 Delivery

7.5.1 Introduction of Delivery

`Delivery` is a mechanism used in **knowledge distillation**, which is to **align the intermediate results** between the teacher model and the student model by delivering and rewriting these intermediate results between them. As shown in the figure below, deliveries can be used to:

- **Deliver the output of a layer of the teacher model directly to a layer of the student model.** In some knowledge distillation algorithms, we may need to deliver the output of a layer of the teacher model to the student model directly. For example, in LAD algorithm, the student model needs to obtain the label assignment of the teacher model directly.
- **Align the inputs of the teacher model and the student model.** For example, in the MMClassification framework, some widely used data augmentations such as mixup and CutMix are not implemented in Data Pipelines but in `forward_train`, and due to the randomness of these data augmentation methods, it may lead to a gap between the input of the teacher model and the student model.

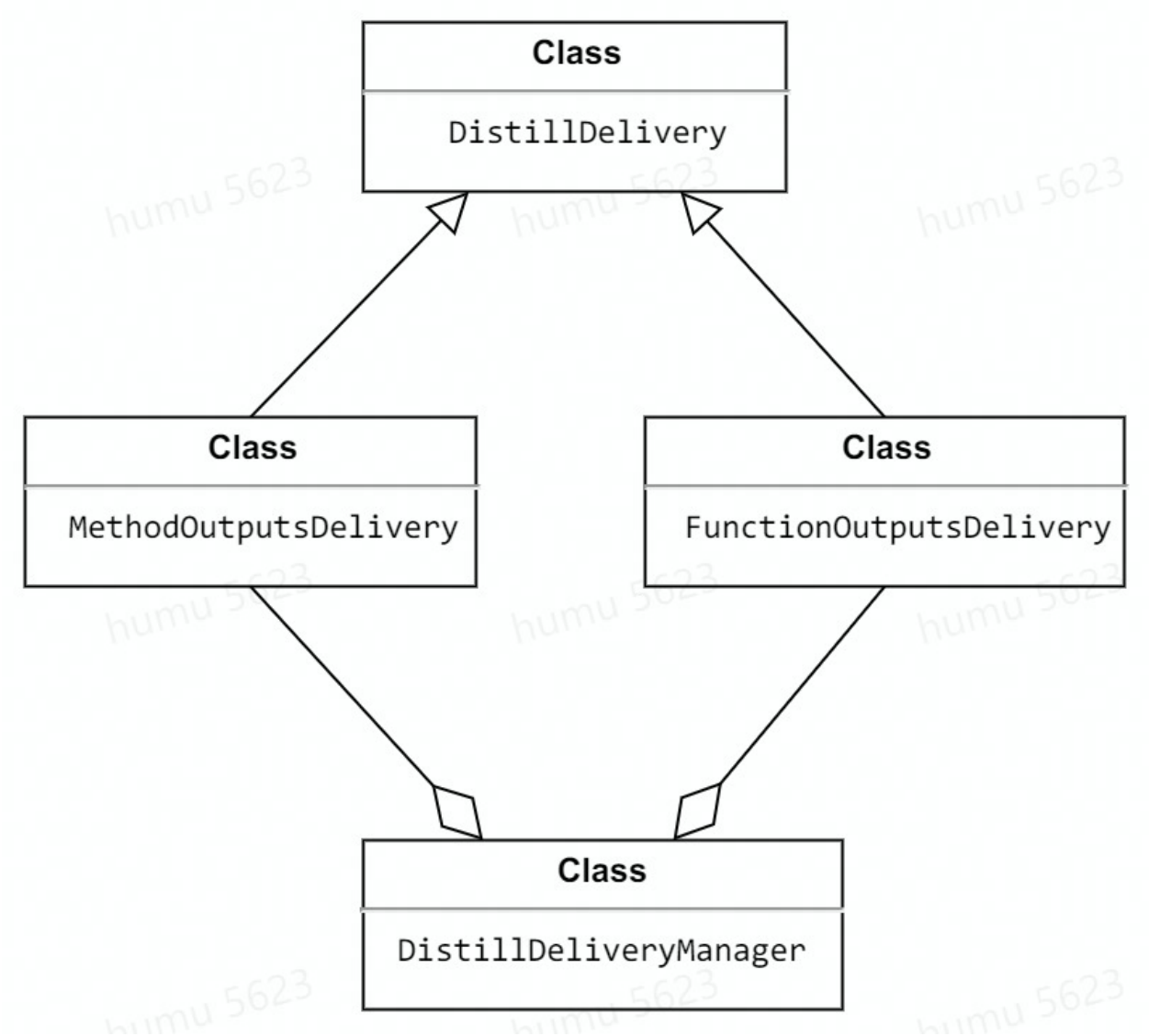


In general, the delivery mechanism allows us to deliver intermediate results between the teacher model and the student model **without adding additional code**, which reduces the hard coding in the source code.

7.5.2 Usage of Delivery

Currently, we support two deliveries: `FunctionOutputsDelivery` and `MethodOutputsDelivery`, both of which inherit from `DistillDiliver`. And these deliveries can be managed by `DistillDeliveryManager` or just be used on their own.

Their relationship is shown below.



FunctionOutputsDelivery

`FunctionOutputsDelivery` is used to align the **function's** intermediate results between the teacher model and the student model.

Note: When initializing `FunctionOutputsDelivery`, you need to pass `func_path` argument, which requires extra attention. For example, `anchor_inside_flags` is a function in mmdetection to check whether the anchors are inside the border. This function is in `mmdet/core/anchor/utils.py` and used in `mmdet/models/dense_heads/anchor_head`. Then the `func_path` should be `mmdet.models.dense_heads.anchor_head.anchor_inside_flags` but not `mmdet.core.anchor.utils.anchor_inside_flags`.

Case 1: Delivery single function's output from the teacher to the student.

```
import random
from mmrazor.core import FunctionOutputsDelivery

def toy_func() -> int:
    return random.randint(0, 1000000)

delivery = FunctionOutputsDelivery(max_keep_data=1, func_path='toy_module.toy_func')

# override_data is False, which means that not override the data with
# the recorded data. So it will get the original output of toy_func
# in teacher model, and it is also recorded to be deliveried to the student.
delivery.override_data = False
with delivery:
    output_teacher = toy_module.toy_func()

# override_data is True, which means that override the data with
# the recorded data, so it will get the output of toy_func
# in teacher model rather than the student's.
delivery.override_data = True
with delivery:
    output_student = toy_module.toy_func()

print(output_teacher == output_student)
```

Out:

```
True
```

Case 2: Delivery multi function's outputs from the teacher to the student.

If a function is executed more than once during the forward of the teacher model, all the outputs of this function will be used to override function outputs from the student model

Note: Delivery order is first-in first-out.

```
delivery = FunctionOutputsDelivery(
    max_keep_data=2, func_path='toy_module.toy_func')

delivery.override_data = False
with delivery:
    output1_teacher = toy_module.toy_func()
    output2_teacher = toy_module.toy_func()

delivery.override_data = True
with delivery:
    output1_student = toy_module.toy_func()
    output2_student = toy_module.toy_func()
```

(continues on next page)

(continued from previous page)

```
print(output1_teacher == output1_student and output2_teacher == output2_student)
```

Out:

```
True
```

MethodOutputsDelivery

`MethodOutputsDelivery` is used to align the **method's** intermediate results between the teacher model and the student model.

Case: Align the inputs of the teacher model and the student model

Here we use mixup as an example to show how to align the inputs of the teacher model and the student model.

- Without Delivery

```
# main.py
from mmcls.models.utils import Augments
from mmrazor.core import MethodOutputsDelivery

augments_cfg = dict(type='BatchMixup', alpha=1., num_classes=10, prob=1.0)
augments = Augments(augments_cfg)

imgs = torch.randn(2, 3, 32, 32)
label = torch.randint(0, 10, (2,))

imgs_teacher, label_teacher = augments(imgs, label)
imgs_student, label_student = augments(imgs, label)

print(torch.equal(label_teacher, label_student))
print(torch.equal(imgs_teacher, imgs_student))
```

Out:

```
False
False
from mmcls.models.utils import Augments
from mmrazor.core import DistillDeliveryManager
```

The results are different due to the randomness of mixup.

- With Delivery

```
delivery = MethodOutputsDelivery(
    max_keep_data=1, method_path='mmcls.models.utils.Augments.__call__')

delivery.override_data = False
with delivery:
    imgs_teacher, label_teacher = augments(imgs, label)
```

(continues on next page)

(continued from previous page)

```
delivery.override_data = True
with delivery:
    imgs_student, label_student = augments(imgs, label)

print(torch.equal(label_teacher, label_student))
print(torch.equal(imgs_teacher, imgs_student))
```

Out:

```
True
True
```

The randomness is eliminated by using `MethodOutputsDelivery`.

2.3 DistillDeliveryManager

`DistillDeliveryManager` is actually a context manager, used to manage delivers. When entering the `DistillDeliveryManager`, all delivers managed will be started.

With the help of `DistillDeliveryManager`, we are able to manage several different DistillDeliveries with as little code as possible, thereby reducing the possibility of errors.

Case: Manager deliveries with DistillDeliveryManager

```
from mmcls.models.utils import Augments
from mmrazor.core import DistillDeliveryManager

augments_cfg = dict(type='BatchMixup', alpha=1., num_classes=10, prob=1.0)
augments = Augments(augments_cfg)

distill_deliveries = [
    ConfigDict(type='MethodOutputs', max_keep_data=1,
               method_path='mmcls.models.utils.Augments.__call__')]

# instantiate DistillDeliveryManager
manager = DistillDeliveryManager(distill_deliveries)

imgs = torch.randn(2, 3, 32, 32)
label = torch.randint(0, 10, (2,))

manager.override_data = False
with manager:
    imgs_teacher, label_teacher = augments(imgs, label)

manager.override_data = True
with manager:
    imgs_student, label_student = augments(imgs, label)

print(torch.equal(label_teacher, label_student))
print(torch.equal(imgs_teacher, imgs_student))
```

Out:

```
True
True
```

7.5.3 Reference

[1] Zhang, Hongyi, et al. “mixup: Beyond empirical risk minimization.” *arXiv* abs/1710.09412 (2017).

[2] Yun, Sangdoo, et al. “Cutmix: Regularization strategy to train strong classifiers with localizable features.” *ICCV* (2019).

[3] Nguyen, Chuong H., et al. “Improving object detection by label assignment distillation.” *WACV* (2022).

CHAPTER

EIGHT

DEVELOPMENT TUTORIALS

8.1 Customize Architectures

Different from other tasks, architectures in MMRazor may consist of some special model components, such as **searchable backbones, connectors, dynamic ops**. In MMRazor, you can not only develop some common model components like other codebases of OpenMMLab, but also develop some special model components. Here is how to develop searchable model components and common model components.

8.1.1 Develop searchable model components

1. Define a new backbone

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

```
# Copyright (c) OpenMMLab. All rights reserved.
import copy
from typing import Dict, List, Optional, Sequence, Tuple, Union

import torch.nn as nn
from mmcls.models.backbones.base_backbone import BaseBackbone
from mmcv.cnn import ConvModule, constant_init, normal_init
from mmcv.runner import ModuleList, Sequential
from torch import Tensor
from torch.nn.modules.batchnorm import _BatchNorm

from mmrazor.registry import MODELS

@MODELS.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
```

(continues on next page)

(continued from previous page)

```
    def init_weights(self):
        pass

    def forward(self, x):
        pass

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

2. Build the architecture of the new backbone based on `arch_setting`

```
@MODELS.register_module()
class SearchableShuffleNetV2(BaseBackbone):
    def __init__(self,
                 arch_setting: List[List],
                 stem_multiplier: int = 1,
                 widen_factor: float = 1.0,
                 out_indices: Sequence[int] = (4, ),
                 frozen_stages: int = -1,
                 with_last_layer: bool = True,
                 conv_cfg: Optional[Dict] = None,
                 norm_cfg: Dict = dict(type='BN'),
                 act_cfg: Dict = dict(type='ReLU'),
                 norm_eval: bool = False,
                 with_cp: bool = False,
                 init_cfg: Optional[Union[Dict, List[Dict]]] = None) -> None:
            layers_nums = 5 if with_last_layer else 4
            for index in out_indices:
                if index not in range(0, layers_nums):
                    raise ValueError('the item in out_indices must in '
                                     f'range(0, 5). But received {index}')

            self.frozen_stages = frozen_stages
            if frozen_stages not in range(-1, layers_nums):
                raise ValueError('frozen_stages must be in range(-1, 5). '
                                 f'But received {frozen_stages}')

            super().__init__(init_cfg)

            self.arch_setting = arch_setting
            self.widen_factor = widen_factor
            self.out_indices = out_indices
            self.conv_cfg = conv_cfg
            self.norm_cfg = norm_cfg
            self.act_cfg = act_cfg
            self.norm_eval = norm_eval
            self.with_cp = with_cp

            last_channels = 1024
            self.in_channels = 16 * stem_multiplier
```

(continues on next page)

(continued from previous page)

```
            # build the first layer
            self.conv1 = ConvModule(
                in_channels=3,
                out_channels=self.in_channels,
                kernel_size=3,
                stride=2,
                padding=1,
                conv_cfg=conv_cfg,
                norm_cfg=norm_cfg,
                act_cfg=act_cfg)

            # build the middle layers
            self.layers = ModuleList()
            for channel, num_blocks, mutable_cfg in arch_setting:
                out_channels = round(channel * widen_factor)
                layer = self._make_layer(out_channels, num_blocks,
                                         copy.deepcopy(mutable_cfg))
                self.layers.append(layer)

            # build the last layer
            if with_last_layer:
                self.layers.append(
                    ConvModule(
                        in_channels=self.in_channels,
                        out_channels=last_channels,
                        kernel_size=1,
                        conv_cfg=conv_cfg,
                        norm_cfg=norm_cfg,
                        act_cfg=act_cfg))
```

3. Implement_make_layer with mutable_cfg

```
@MODELS.register_module()
class SearchableShuffleNetV2(BaseBackbone):

    ...

    def _make_layer(self, out_channels: int, num_blocks: int,
                    mutable_cfg: Dict) -> Sequential:
        """Stack mutable blocks to build a layer for ShuffleNet V2.
        Note:
            Here we use ``module_kwargs`` to pass dynamic parameters such as
            ``in_channels``, ``out_channels`` and ``stride``
            to build the mutable.
        Args:
            out_channels (int): out_channels of the block.
            num_blocks (int): number of blocks.
            mutable_cfg (dict): Config of mutable.
        Returns:
            mmcv.runner.Sequential: The layer made.
        """
        layers = []
```

(continues on next page)

(continued from previous page)

```
        for i in range(num_blocks):
            stride = 2 if i == 0 else 1

            mutable_cfg.update(
                module_kwargs=dict(
                    in_channels=self.in_channels,
                    out_channels=out_channels,
                    stride=stride))
            layers.append(MODELS.build(mutable_cfg))
            self.in_channels = out_channels

        return Sequential(*layers)

    ...
```

4. Implement other common methods

You can refer to the implementation of `ShuffleNetV2` in mmcls for finishing other common methods.

5. Import the module

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

```
from .searchable_shufflenet_v2 import SearchableShuffleNetV2

__all__ = ['SearchableShuffleNetV2']
```

or alternatively add

```
custom_imports = dict(
    imports=['mmrazor.models.architectures.backbones.searchable_shufflenet_v2'],
    allow_failed_imports=False)
```

to the config file to avoid modifying the original code.

6. Use the backbone in your config file

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

8.1.2 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/backbones/xxxnet.py`, then implement the class `xxxNet`.

```
from mmengine.model import BaseModule
from mmrazor.registry import MODELS

@MODELS.register_module()
class xxxNet(BaseModule):

    def __init__(self, arg1, arg2, init_cfg=None):
        super().__init__(init_cfg=init_cfg)
        pass

    def forward(self, x):
        pass
```

2. Import the module

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

```
from .xxxnet import xxxNet

__all__ = ['xxxNet']
```

or alternatively add

```
custom_imports = dict(
    imports=['mmrazor.models.architectures.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',
            arg1=xxx,
            arg2=xxx),
        ...
```

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

8.2 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/nas/spos.py`, class `SPOS` inherits from class `BaseAlgorithm`

```
from mmrazor.registry import MODELS
from ..base import BaseAlgorithm

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

    def loss(self, batch_inputs, data_samples):
        pass
```

2. Develop new algorithm components (optional)

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

a. Create a new file `mmrazor/models/mutators/module_mutator/one_shot_module_mutator.py`, class `OneShotModuleMutator` inherits from class `ModuleMutator`

b. Finish the functions you need in `OneShotModuleMutator`, eg: `sample_choices`, `set_choices` and so on.

```
from mmrazor.registry import MODELS
from .module_mutator import ModuleMutator


@MODELS.register_module()
class OneShotModuleMutator(ModuleMutator):

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

    def sample_choices(self) -> Dict[int, Any]:
        pass

    def set_choices(self, choices: Dict[int, Any]) -> None:
        pass

    @property
    def mutable_class_type(self):
        return OneShotMutableModule
```

c. Import the new mutator

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

```
from .module_mutator import OneShotModuleMutator
```

or alternatively add

```
custom_imports = dict(
    imports=['mmrazor.models.mutators.module_mutator.one_shot_module_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='mmrazor.OneShotModuleMutator')
```

For further information, please refer to Mutator for more details.

3. Rewrite its `loss` function.

Develop key logic of your algorithm in function`loss`. When having special steps to optimize, you should rewrite the function `train_step`.

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

    def sample_subnet(self):
        pass

    def set_subnet(self, subnet):
        pass

    def loss(self, batch_inputs, data_samples):
        if self.is_supernet:
            random_subnet = self.sample_subnet()
            self.set_subnet(random_subnet)
            return self.architecture(batch_inputs, data_samples, mode='loss')
        else:
            return self.architecture(batch_inputs, data_samples, mode='loss')
```

4. Add your custom functions (optional)

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

5. Import the class

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

```
from .spos import SPOS

__all__ = ['SPOS']
```

or alternatively add

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

to the config file to avoid modifying the original code.

6. Use the algorithm in your config file

```
model = dict(
    type='mmrazor.SPOS',
    architecture=supernet,
    mutator=dict(type='mmrazor.OneShotModuleMutator'))
```

8.3 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/prunning/autoslim.py`, class `AutoSlim` inherits from class `BaseAlgorithm`.

```
from mmrazor.registry import MODELS
from .base import BaseAlgorithm

@MODELS.register_module()
class AutoSlim(BaseAlgorithm):
    def __init__(self,
                 mutator,
                 distiller,
                 architecture,
                 data_preprocessor,
                 num_random_samples = 2,
                 init_cfg = None) -> None:
        super().__init__(**kwargs)
        pass

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

2. Develop new algorithm components (optional)

AutoSlim can directly use class `OneShotChannelMutator` as core functions provider. If it can not meet your needs, you can develop new algorithm components for your algorithm like `OneShotChannalMutator`. We will take `OneShotChannelMutator` as an example to introduce how to develop a new algorithm component:

a. Create a new file `mmrazor/models/mutators/channel_mutator/one_shot_channel_mutator.py`, class `OneShotChannelMutator` can inherits from `ChannelMutator`.

b. Finish the functions you need, eg: `build_search_groups`, `set_choices` , `sample_choices` and so on

```
from mmrazor.registry import MODELS
from .channel_mutator import ChannelMutator


@MODELS.register_module()
class OneShotChannelMutator(ChannelMutator):
    def __init__(self, **kwargs):
        super().__init__(**kwargs)
```

(continues on next page)

(continued from previous page)

```
    def sample_choices(self):
        pass

    def set_choices(self, choice_dict):
        pass

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

c. Import the module in mmrazor/models/mutators/channel_mutator/__init__.py

```
from .one_shot_channel_mutator import OneShotChannelMutator

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

3. Rewrite its train_step

Develop key logic of your algorithm in functiontrain_step

```
from mmrazor.registry import MODELS
from ..base import BaseAlgorithm

@ALGORITHMS.register_module()
class AutoSlim(BaseAlgorithm):
    def __init__(self,
                 mutator,
                 distiller,
                 architecture,
                 data_preprocessor,
                 num_random_samples = 2,
                 init_cfg = None) -> None:
        super(AutoSlim, self).__init__(**kwargs)
        pass

    def train_step(self, data: List[dict],
                   optim_wrapper: OptimWrapper) -> Dict[str, torch.Tensor]:

        def distill_step(
                batch_inputs: torch.Tensor, data_samples: List[BaseDataElement]
        ) -> Dict[str, torch.Tensor]:
            ...
            return subnet_losses

        batch_inputs, data_samples = self.data_preprocessor(data, True)

        total_losses = dict()
        for kind in self.sample_kinds:
            # update the max subnet loss.
            if kind == 'max':
                self.set_max_subnet()
                with optim_wrapper.optim_context(
```

(continues on next page)

(continued from previous page)

```
                    self), self.distiller.teacher_recorders:  # type: ignore
                max_subnet_losses = self(batch_inputs, data_samples, mode='loss')
                parsed_max_subnet_losses, _ = self.parse_losses(max_subnet_losses)
                optim_wrapper.update_params(parsed_max_subnet_losses)
            total_losses.update(add_prefix(max_subnet_losses, 'max_subnet'))
        # update the min subnet loss.
        elif kind == 'min':
            self.set_min_subnet()
            min_subnet_losses = distill_step(batch_inputs, data_samples)
            total_losses.update(add_prefix(min_subnet_losses, 'min_subnet'))
        # update the random subnets loss.
        elif 'random' in kind:
            self.set_subnet(self.sample_subnet())
            random_subnet_losses = distill_step(batch_inputs, data_samples)
            total_losses.update(
                add_prefix(random_subnet_losses, f'{kind}_subnet'))

    return total_losses
```

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 .pruning import AutoSlim

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

Or alternatively add

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

to the config file to avoid modifying the original code.

6. Use the algorithm in your config file

```
model = dict(
    type='AutoSlim',
    architecture=...,
    mutator=dict(type='OneShotChannelMutator', ...),
    )
```

8.4 Customize KD algorithms

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

1. Register a new algorithm

Create a new file `mmrazor/models/algorithms/distill/configurable/single_teacher_distill.py`, class `SingleTeacherDistill` inherits from class `BaseAlgorithm`

```
from mmrazor.registry import MODELS
from ..base import BaseAlgorithm

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

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

2. Develop connectors (Optional) .

Take ConvModuleConnector as an example.

```
from mmrazor.registry import MODELS
from .base_connector import BaseConnector

@MODELS.register_module()
class ConvModuleConnector(BaseConnector):
    def __init__(self, in_channel, out_channel, kernel_size = 1, stride = 1):
        ...

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

3. Develop distiller.

Take `ConfigurableDistiller` as an example.

```
from .base_distiller import BaseDistiller
from mmrazor.registry import MODELS


@MODELS.register_module()
class ConfigurableDistiller(BaseDistiller):
    def __init__(self,
                 student_recorders = None,
                 teacher_recorders = None,
                 distill_deliveries = None,
                 connectors = None,
                 distill_losses = None,
                 loss_forward_mappings = None):
        ...
```

(continues on next page)

(continued from previous page)

```
    def build_connectors(self, connectors):
        ...

    def build_distill_losses(self, losses):
        ...

    def compute_distill_losses(self):
        ...
```

4. Develop custom loss (Optional).

Here we take `L1Loss` as an example. Create a new file in `mmrazor/models/losses/l1_loss.py`.

```
from mmrazor.registry import MODELS

@MODELS.register_module()
class L1Loss(nn.Module):
    def __init__(
        self,
        loss_weight: float = 1.0,
        size_average: Optional[bool] = None,
        reduce: Optional[bool] = None,
        reduction: str = 'mean',
    ) -> None:
        super().__init__()
        ...

    def forward(self, s_feature, t_feature):
        loss = F.l1_loss(s_feature, t_feature, self.size_average, self.reduce,
                         self.reduction)
        return self.loss_weight * loss
```

5. Import the class

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

```
from .single_teacher_distill import SingleTeacherDistill

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

or alternatively add

```
custom_imports = dict(
    imports=['mmrazor.models.algorithms.distill.configurable.single_teacher_distill'],
    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='Distill',
    distiller=dict(type='SingleTeacherDistill', ...),
    # you can also use your new algorithm components here
```

(continues on next page)

(continued from previous page)

```
    ...
)
```

8.5 Customize Quantization algorithms

Here we show how to develop new QAT algorithms with an example of LSQ on OpenVINO backend.

This document is mainly aimed at QAT because the ptq process is relatively fixed and the components we provide can meet most of the needs. We will first give an overview of the overall required development components, and then introduce the specific implementation step by step.

8.5.1 Overall

In the mmrazor quantization pipeline, in order to better support the openmmlab environment, we have configured most of the code modules for users. You can configure all the components directly in the config file. How to configure them can be found in our file.

```
global_qconfig = dict(
    w_observer=dict(),
    a_observer=dict(),
    w_fake_quant=dict(),
    a_fake_quant=dict(),
    w_qscheme=dict(),
    a_qscheme=dict(),
)
model = dict(
    type='mmrazor.MMArchitectureQuant',
    architecture=resnet,
    quantizer=dict(
        type='mmrazor.OpenvinoQuantizer',
        global_qconfig=global_qconfig,
        tracer=dict()))
train_cfg = dict(type='mmrazor.LSQEpochBasedLoop')
```

For `algorithm` and `tracer`, we recommend that you use the default configurations `MMArchitectureQuant` and `CustomTracer` provided by us. These two module operators are specially built for the openmmlab environment, while other modules can refer to the following steps and choose or develop new operators according to your needs.

To adapt to different backends, you need to select a different `quantizer`.

To develop new quantization algorithms, you need to define new `observer` and `fakequant`.

If the existing `loop` does not meet your needs, you may need to make some changes to the existing `loop` based on your algorithm.

8.5.2 Detailed steps

1. Select a quantization algorithm

We recommend that you directly use theMMArchitectureQuant in mmrazor/models/algorithms/quantization/mm_architecture.py.The class MMArchitectureQuant inherits from class BaseAlgorithm.

This structure is built for the model in openmmlab. If you have other requirements, you can also refer to this document to design the overall framework.

2. Select quantizer

At present, the quantizers we support are NativeQuantizer, OpenVINOQuantizer, TensorRTQuantizer and AcademicQuantizer in mmrazor/models/quantizers/. AcademicQuantizer and NativeQuantizer inherit from class BaseQuantizer in mmrazor/models/quantizers/base.py:

```
class BaseQuantizer(BaseModule):
    def __init__(self, tracer):
        super().__init__()
        self.tracer = TASK_UTILS.build(tracer)
    @abstractmethod
    def prepare(self, model, graph_module):
        """tmp."""
        pass
    def swap_ff_with_fxff(self, model):
        pass
```

NativeQuantizer is the operator we developed to adapt to the environment of mmrazor according to pytorch's official quantization logic. AcademicQuantizer is an operator designed for academic research to give users more space to operate.

The class OpenVINOQuantizer and TensorRTQuantizer inherits from class NativeQuantize. They adapted OpenVINO and TensorRTbackend respectively. You can also try to develop a quantizer based on other backends according to your own needs.

3. Select tracer

Tracer we use CustomTracer in mmrazor/models/task_modules/tracer/fx/custom_tracer.py. You can inherit this class and customize your own tracer.

4. Develop new fakequant method(optional)

You can use fakequants provided by pytorch in mmrazor/models/fake_quants/torch_fake_quants.py as core functions provider. If you want to use the fakequant methods from other papers, you can also define them yourself. Let's take lsq as an example as follows:

a.Create a new file mmrazor/models/fake_quants/lsq.py, class LearnableFakeQuantize inherits from class FakeQuantizeBase.

b. Finish the functions you need, eg: observe_quant_params, calculate_qparams and so on.

```
from mmrazor.registry import MODELS
from torch.ao.quantization import FakeQuantizeBase

@MODELS.register_module()
class LearnableFakeQuantize(FakeQuantizeBase):
    def __init__(self,
                 observer,
                 quant_min=0,
```

(continues on next page)

(continued from previous page)

```
                quant_max=255,
                scale=1.,
                zero_point=0.,
                use_grad_scaling=True,
                zero_point_trainable=False,
                **observer_kwargs):
        super(LearnableFakeQuantize, self).__init__()
        pass

    def observe_quant_params(self):
        pass

    def calculate_qparams(self):
        pass

    def forward(self, X):
        pass
```

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

```
from .lsq import LearnableFakeQuantize

__all__ = ['LearnableFakeQuantize']
```

5. Develop new observer(optional)

You can directly use observers provided by pytorch in `mmrazor/models/observers/torch_observers.py` or use observers customized by yourself. Let's take `LSQObserver` as follows:

a.Create a new observer file `mmrazor/models/observers/lsq.py`, class `LSQObserver` inherits from class `MinMaxObserver` and `LSQObserverMixIn`. These two observers can calculate `zero_point` and `scale`, respectively.

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

```
from mmrazor.registry import MODELS
from torch.ao.quantization.observer import MinMaxObserver

class LSQObserverMixIn:
    def __init__(self):
        self.tensor_norm = None

    @torch.jit.export
    def _calculate_scale(self):
        scale = 2 * self.tensor_norm / math.sqrt(self.quant_max)
        sync_tensor(scale)
        return scale

@MODELS.register_module()
class LSQObserver(MinMaxObserver, LSQObserverMixIn):
    """LSQ observer.
    Paper: Learned Step Size Quantization. <https://arxiv.org/abs/1902.08153>
    """
    def __init__(self, *args, **kwargs):
        MinMaxObserver.__init__(self, *args, **kwargs)
```

(continues on next page)

(continued from previous page)

```
        LSQObserverMixIn.__init__(self)

    def forward(self, x_orig):
        """Records the running minimum, maximum and tensor_norm of ``x``."""
        if x_orig.numel() == 0:
            return x_orig
        x = x_orig.detach()  # avoid keeping autograd tape
        x = x.to(self.min_val.dtype)
        self.tensor_norm = x.abs().mean()
        min_val_cur, max_val_cur = torch.aminmax(x)
        min_val = torch.min(min_val_cur, self.min_val)
        max_val = torch.max(max_val_cur, self.max_val)
        self.min_val.copy_(min_val)
        self.max_val.copy_(max_val)
        return x_orig

    @torch.jit.export
    def calculate_qparams(self):
        """Calculates the quantization parameters."""
        _, zero_point = MinMaxObserver.calculate_qparams(self)
        scale = LSQObserverMixIn._calculate_scale(self)
        return scale, zero_point
```

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

```
from .lsq import LSQObserver

__all__ = ['LSQObserver']
```

6. Select loop or develop new loop

At present, the QAT loops we support are `PTQLoop` and `QATEpochBasedLoop`, in `mmrazor/engine/runner/quantization_loops.py`. We can develop a new `LSQEpochBasedLoop` inherits from class `QATEpochBasedLoop` and finish the functions we need in LSQ method.

```
from mmengine.runner import EpochBasedTrainLoop

@LOOPS.register_module()
class LSQEpochBasedLoop(QATEpochBasedLoop):
    def __init__(
            self,
            runner,
            dataloader: Union[DataLoader, Dict],
            max_epochs: int,
            val_begin: int = 1,
            val_interval: int = 1,
            freeze_bn_begin: int = -1,
            dynamic_intervals: Optional[List[Tuple[int, int]]] = None) -> None:
        super().__init__(
            runner,
            dataloader,
            max_epochs,
            val_begin,
```

(continues on next page)

(continued from previous page)

```
            val_interval,
            freeze_bn_begin=freeze_bn_begin,
            dynamic_intervals=dynamic_intervals)

        self.is_first_batch = True

    def prepare_for_run_epoch(self):
        pass

    def prepare_for_val(self):
        pass

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

And then Import the module in `mmrazor/engine/runner/__init__.py`

```
from .quantization_loops import LSQEpochBasedLoop

__all__ = ['LSQEpochBasedLoop']
```

7. Use the algorithm in your config file

After completing the above steps, we have all the components of the qat algorithm, and now we can combine them in the config file.

a.First, `_base_` stores the location of the model that needs to be quantized.

b.Second, configure observer,fakequant and qscheme in `global_qconfig` in detail. You can configure the required quantization bit width and quantization methods in `qscheme`, such as symmetric quantization or asymmetric quantization.

c.Third, build the whole mmrazor model in `model`.

d.Finally, complete all the remaining required configuration files.

```
_base_ = ['mmcls::resnet/resnet18_8xb16_cifar10.py']

global_qconfig = dict(
    w_observer=dict(type='mmrazor.LSQPerChannelObserver'),
    a_observer=dict(type='mmrazor.LSQObserver'),
    w_fake_quant=dict(type='mmrazor.LearnableFakeQuantize'),
    a_fake_quant=dict(type='mmrazor.LearnableFakeQuantize'),
    w_qscheme=dict(
        qdtype='qint8', bit=8, is_symmetry=True, is_symmetric_range=True),
    a_qscheme=dict(qdtype='quint8', bit=8, is_symmetry=True),
)

model = dict(
    _delete_=True,
    _scope_='mmrazor',
    type='MMArchitectureQuant',
    data_preprocessor=dict(
        type='mmcls.ClsDataPreprocessor',
        num_classes=1000,
```

(continues on next page)

(continued from previous page)

```
        # RGB format normalization parameters
        mean=[123.675, 116.28, 103.53],
        std=[58.395, 57.12, 57.375],
        # convert image from BGR to RGB
        to_rgb=True),
    architecture=resnet,
    float_checkpoint=float_ckpt,
    quantizer=dict(
        type='mmrazor.OpenVINOQuantizer',
        is_qat=True,
        global_qconfig=global_qconfig,
        tracer=dict(
            type='mmrazor.CustomTracer',
            skipped_methods=[
                'mmcls.models.heads.ClsHead._get_loss',
                'mmcls.models.heads.ClsHead._get_predictions'
            ])))

# learning policy
optim_wrapper = dict()
param_scheduler = dict()
model_wrapper_cfg = dict()

# train, val, test setting
train_cfg = dict(type='mmrazor.LSQEpochBasedLoop')
val_cfg = dict()
test_cfg = val_cfg
```

8.6 Customize mixed algorithms

Here we show how to customize mixed algorithms with our algorithm components. We take AutoSlim as an example.

Note: Why is AutoSlim a mixed algorithm?

In AutoSlim, the sandwich rule and the inplace distillation will be introduced to enhance the training process, which is called as the slimmable training. The sandwich rule means that we train the model at smallest width, largest width and (n 2) random widths, instead of n random widths. And the inplace distillation means that 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 AutoSlim.

---

1. Register a new algorithm

Create a new file `mmrazor/models/algorithms/nas/autoslim.py`, class `AutoSlim` inherits from class `BaseAlgorithm`. You need to build the KD algorithm component (distiller) and the pruning algorithm component (mutator) because AutoSlim is a mixed algorithm.

Note: You can also inherit from the existing algorithm instead of `BaseAlgorithm` if your algorithm is similar to the existing algorithm.

Note: You can choose existing algorithm components in MMRazor, such as `OneShotChannelMutator` and `ConfigurableDistiller` in AutoSlim.

If these in MMRazor don't meet your needs, you can customize new algorithm components for your algorithm. Reference is as follows:

Customize NAS algorithms Customize Pruning algorithms Customize KD algorithms

```python
# Copyright (c) OpenMMLab. All rights reserved.
from typing import Dict, List, Optional, Union
import torch
from torch import nn

from mmrazor.models.distillers import ConfigurableDistiller
from mmrazor.models.mutators import OneShotChannelMutator
from mmrazor.registry import MODELS
from ..base import BaseAlgorithm

VALID_MUTATOR_TYPE = Union[OneShotChannelMutator, Dict]
VALID_DISTILLER_TYPE = Union[ConfigurableDistiller, Dict]

@MODELS.register_module()
class AutoSlim(BaseAlgorithm):
    def __init__(self,
                 mutator: VALID_MUTATOR_TYPE,
                 distiller: VALID_DISTILLER_TYPE,
                 architecture: Union[BaseModel, Dict],
                 data_preprocessor: Optional[Union[Dict, nn.Module]] = None,
                 num_random_samples: int = 2,
                 init_cfg: Optional[Dict] = None) -> None:
        super().__init__(architecture, data_preprocessor, init_cfg)
        self.mutator = self._build_mutator(mutator)
        # `prepare_from_supernet` must be called before distiller initialized
        self.mutator.prepare_from_supernet(self.architecture)

        self.distiller = self._build_distiller(distiller)
        self.distiller.prepare_from_teacher(self.architecture)
        self.distiller.prepare_from_student(self.architecture)

        ......

    def _build_mutator(self,
                       mutator: VALID_MUTATOR_TYPE) -> OneShotChannelMutator:
        """build mutator."""
        if isinstance(mutator, dict):
            mutator = MODELS.build(mutator)
        if not isinstance(mutator, OneShotChannelMutator):
            raise TypeError('mutator should be a `dict` or '
                            '`OneShotModuleMutator` instance, but got '
                            f'{type(mutator)}')

        return mutator
```

(continues on next page)

(continued from previous page)

```
    def _build_distiller(
            self, distiller: VALID_DISTILLER_TYPE) -> ConfigurableDistiller:
        if isinstance(distiller, dict):
            distiller = MODELS.build(distiller)
        if not isinstance(distiller, ConfigurableDistiller):
            raise TypeError('distiller should be a `dict` or '
                            '`ConfigurableDistiller` instance, but got '
                            f'{type(distiller)}')

        return distiller
```

2. Implement the core logic in `train_step`

In `train_step`, both the `mutator` and the `distiller` play an important role. For example, `sample_subnet`, `set_max_subnet` and `set_min_subnet` are supported by the `mutator`, and the function of`distill_step` is mainly implemented by the `distiller`.

```
@MODELS.register_module()
class AutoSlim(BaseAlgorithm):

    ......

    def train_step(self, data: List[dict],
                   optim_wrapper: OptimWrapper) -> Dict[str, torch.Tensor]:

        def distill_step(
                batch_inputs: torch.Tensor, data_samples: List[BaseDataElement]
        ) -> Dict[str, torch.Tensor]:
            ......

        ......

        batch_inputs, data_samples = self.data_preprocessor(data, True)

        total_losses = dict()
        for kind in self.sample_kinds:
            # update the max subnet loss.
            if kind == 'max':
                self.set_max_subnet()
                ......
                total_losses.update(add_prefix(max_subnet_losses, 'max_subnet'))
            # update the min subnet loss.
            elif kind == 'min':
                self.set_min_subnet()
                min_subnet_losses = distill_step(batch_inputs, data_samples)
                total_losses.update(add_prefix(min_subnet_losses, 'min_subnet'))
            # update the random subnets loss.
            elif 'random' in kind:
                self.set_subnet(self.sample_subnet())
                random_subnet_losses = distill_step(batch_inputs, data_samples)
                total_losses.update(
                    add_prefix(random_subnet_losses, f'{kind}_subnet'))
```

(continues on next page)

(continued from previous page)

```
        return total_losses
```

3. Import the class

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

```
from .autoslim import AutoSlim

__all__ = ['AutoSlim']
```

or alternatively add

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

to the config file to avoid modifying the original code.

4. Use the algorithm in your config file

```
model= dict(
    type='mmrazor.AutoSlim',
    architecture=...,
    mutator=dict(
        type='OneShotChannelMutator',
        ...),
    distiller=dict(
        type='ConfigurableDistiller',
        ...),
    ...)
```

8.7 Apply existing algorithms to new tasks

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

SPOS: Single Path One-Shot NAS for classification

DetNAS: Single Path One-Shot NAS for detection

You just need to configure the existing algorithms in your config only by replacing the architecture of mmcls with mmdet 's

You can implement a new algorithm by inheriting from the existing algorithm quickly if the new task's specificity leads to the failure of applying directly.

SPOS config VS DetNAS config

- SPOS

```
_base_ = [
    'mmrazor::_base_/settings/imagenet_bs1024_spos.py',
    'mmrazor::_base_/nas_backbones/spos_shufflenet_supernet.py',
    'mmcls::_base_/default_runtime.py',
```

(continues on next page)

(continued from previous page)

```
]

# model
supernet = dict(
    type='ImageClassifier',
    data_preprocessor=_base_.preprocess_cfg,
    backbone=_base_.nas_backbone,
    neck=dict(type='GlobalAveragePooling'),
    head=dict(
        type='LinearClsHead',
        num_classes=1000,
        in_channels=1024,
        loss=dict(
            type='LabelSmoothLoss',
            num_classes=1000,
            label_smooth_val=0.1,
            mode='original',
            loss_weight=1.0),
        topk=(1, 5)))

model = dict(
    type='mmrazor.SPOS',
    architecture=supernet,
    mutator=dict(type='mmrazor.OneShotModuleMutator'))

find_unused_parameters = True
```

- DetNAS

```
_base_ = [
    'mmdet::_base_/models/faster-rcnn_r50_fpn.py',
    'mmdet::_base_/datasets/coco_detection.py',
    'mmdet::_base_/schedules/schedule_1x.py',
    'mmdet::_base_/default_runtime.py',
    'mmrazor::_base_/nas_backbones/spos_shufflenet_supernet.py'
]

norm_cfg = dict(type='SyncBN', requires_grad=True)

supernet = _base_.model

supernet.backbone = _base_.nas_backbone
supernet.backbone.norm_cfg = norm_cfg
supernet.backbone.out_indices = (0, 1, 2, 3)
supernet.backbone.with_last_layer = False

supernet.neck.norm_cfg = norm_cfg
supernet.neck.in_channels = [64, 160, 320, 640]

supernet.roi_head.bbox_head.norm_cfg = norm_cfg
supernet.roi_head.bbox_head.type = 'Shared4Conv1FCBBoxHead'
```

(continues on next page)

(continued from previous page)

```
model = dict(
    _delete_=True,
    type='mmrazor.SPOS',
    architecture=supernet,
    mutator=dict(type='mmrazor.OneShotModuleMutator'))

find_unused_parameters = True
```

CHAPTER

NINE

CHANGELOG OF V1.X

9.1 v1.0.0 (24/04/2023)

We are excited to announce the first official release of MMRazor 1.0.

9.1.1 Highlights

- MMRazor quantization is released, which has got through task models and model deployment. With its help, we can quantize and deploy pre-trained models in OpenMMLab to specified backend quickly.

9.1.2 New Features & Improvements

NAS

- Update searchable model. (https://github.com/open-mmlab/mmrazor/pull/438)
- Update NasMutator to build search_space in NAS. (https://github.com/open-mmlab/mmrazor/pull/426)

Pruning

- Add a new pruning algorithm named GroupFisher. We support the full pipeline for GroupFisher, including pruning, finetuning and deployment.(https://github.com/open-mmlab/mmrazor/pull/459)

KD

- Support stopping distillation after a certain epoch. (https://github.com/open-mmlab/mmrazor/pull/455)
- Support distilling rtmdet with mmrazor, refer to here. (https://github.com/open-mmlab/mmyolo/pull/544)
- Add mask channel in MGD Loss. (https://github.com/open-mmlab/mmrazor/pull/461)

Quantization

- Support two quantization types: QAT and PTQ (https://github.com/open-mmlab/mmrazor/pull/513)
- Support various quantization bits. (https://github.com/open-mmlab/mmrazor/pull/513)
- Support various quantization methods, such as per_tensor / per_channel, symmetry / asymmetry and so on. (https://github.com/open-mmlab/mmrazor/pull/513)
- Support deploy quantized models to multiple backends, such as OpenVINO, TensorRT and so on. (https://github.com/open-mmlab/mmrazor/pull/513)
- Support applying quantization algorithms to multiple task repos directly, such as mmcls, mmdet and so on. (https://github.com/open-mmlab/mmrazor/pull/513)

9.1.3 Bug Fixes

- Fix split in Darts config. (https://github.com/open-mmlab/mmrazor/pull/451)
- Fix a bug in Recorders. (https://github.com/open-mmlab/mmrazor/pull/446)
- Fix a bug when using get_channel_unit.py. (https://github.com/open-mmlab/mmrazor/pull/432)
- Fix a bug when deploying a pruned model to cuda. (https://github.com/open-mmlab/mmrazor/pull/495)

9.1.4 Contributors

A total of 10 developers contributed to this release. Thanks @415905716 @gaoyang07 @humu789 @LKJacky @HIT-cwh @aptsunny @cape-zck @vansin @twmht @wm901115nwpu

9.2 v1.0.0rc2 (06/01/2023)

We are excited to announce the release of MMRazor 1.0.0rc2.

9.2.1 New Features

NAS

- Add Performance Predictor: Support 4 performance predictors with 4 basic machine learning algorithms, which can be used to directly predict model accuracy without evaluation.(https://github.com/open-mmlab/mmrazor/pull/306)
- Support Autoformer, a one-shot architecture search algorithm dedicated to vision transformer search.(https://github.com/open-mmlab/mmrazor/pull/315)
- Support BigNAS, a NAS algorithm which searches the following items in MobileNetV3 with the one-shot paradigm: kernel_sizes, out_channels, expand_ratios, block_depth and input sizes. (https://github.com/open-mmlab/mmrazor/pull/219)

Pruning

- Support DCFF, a filter channel pruning algorithm dedicated to efficient image classification.(https://github.com/open-mmlab/mmrazor/pull/295)
- We release a powerful tool to automatically analyze channel dependency, named ChannelAnalyzer. Here is an example as shown below.(https://github.com/open-mmlab/mmrazor/pull/371)

Now, ChannelAnalyzer supports most of CNN models in torchvision, mmcls, mmseg and mmdet. We will continue to support more models.

```
from mmrazor.models.task_modules import ChannelAnalyzer
from mmengine.hub import get_model
import json

model = get_model('mmdet::retinanet/retinanet_r18_fpn_1x_coco.py')
unit_configs: dict = ChannelAnalyzer().analyze(model)
unit_config0 = list(unit_configs.values())[0]
print(json.dumps(unit_config0, indent=4))
# # short version of the config
# {
#     "channels": {
#         "input_related": [
#             {"name": "backbone.layer2.0.bn1"},
#             {“name": "backbone.layer2.0.conv2"}
#         ],
#         "output_related": [
#             {"name": "backbone.layer2.0.conv1"},
#             {"name": "backbone.layer2.0.bn1"}
#         ]
#     },
#}
```

KD

- Support MGD, a detection distillation algorithm.(https://github.com/open-mmlab/mmrazor/pull/381)

9.2.2 Bug Fixes

- Fix `FpnTeacherDistll` techer forward from `backbone + neck + head` to `backbone + neck`(#387)
- Fix some expire configs and checkpoints(#373 #372 #422)

9.2.3 Ongoing Changes

We will release Quantization in next version(1.0.0rc3)!

9.2.4 Contributors

A total of 11 developers contributed to this release: @wutongshenqiu @sunnyxiaohu @aptsunny @humu789 @TinyTigerPan @FreakieHuang @LKJacky @wilxy @gaoyang07 @spynccat @yivona08.

9.3 v1.0.0rc1 (27/10/2022)

We are excited to announce the release of MMRazor 1.0.0rc1.

9.3.1 Highlights

- **New Pruning Framework**We have systematically refactored the Pruning module. The new Pruning module can more automatically resolve the dependencies between channels and cover more corner cases.

9.3.2 New Features

Pruning

- A new pruning framework is released in this release. (#311, #313) It consists of five core modules, including Algorithm, `ChannelMutator`, `MutableChannelUnit`, `MutableChannel` and `DynamicOp`.
- MutableChannelUnit is introduced for the first time. Each MutableChannelUnit manages all channels with channel dependency.

```
from mmrazor.registry import MODELS

ARCHITECTURE_CFG = dict(
    _scope_='mmcls',
    type='ImageClassifier',
    backbone=dict(type='MobileNetV2', widen_factor=1.5),
    neck=dict(type='GlobalAveragePooling'),
    head=dict(type='mmcls.LinearClsHead', num_classes=1000, in_channels=1920))
model = MODELS.build(ARCHITECTURE_CFG)
from mmrazor.models.mutators import ChannelMutator

channel_mutator = ChannelMutator()
channel_mutator.prepare_from_supernet(model)
units = channel_mutator.mutable_units
print(units[0])
# SequentialMutableChannelUnit(
#   name=backbone.conv1.conv_(0, 48)_48
#   (output_related): ModuleList(
#     (0): Channel(backbone.conv1.conv, index=(0, 48), is_output_channel=true,␣
↪expand_ratio=1)
#     (1): Channel(backbone.conv1.bn, index=(0, 48), is_output_channel=true, expand_
↪ratio=1)
#     (2): Channel(backbone.layer1.0.conv.0.conv, index=(0, 48), is_output_
↪channel=true, expand_ratio=1)
#     (3): Channel(backbone.layer1.0.conv.0.bn, index=(0, 48), is_output_
↪channel=true, expand_ratio=1)
```

(continues on next page)

(continued from previous page)

```
#   )
#   (input_related): ModuleList(
#     (0): Channel(backbone.conv1.bn, index=(0, 48), is_output_channel=false,␣
↪expand_ratio=1)
#     (1): Channel(backbone.layer1.0.conv.0.conv, index=(0, 48), is_output_
↪channel=false, expand_ratio=1)
#     (2): Channel(backbone.layer1.0.conv.0.bn, index=(0, 48), is_output_
↪channel=false, expand_ratio=1)
#     (3): Channel(backbone.layer1.0.conv.1.conv, index=(0, 48), is_output_
↪channel=false, expand_ratio=1)
#   )
#   (mutable_channel): SquentialMutableChannel(num_channels=48, activated_
↪channels=48)
# )
```

Our new pruning algorithm can help you develop pruning algorithm more fluently. Pelease refer to our documents PruningUserGuide for model detail.

Distillation

- Support CRD, a distillation algorithm based on contrastive representation learning. (#281)
- Support PKD, a distillation algorithm that can be used in `MMDetection` and `MMDetection3D`. #304
- Support DEIT, a classic **Transformer** distillation algorithm.(#332)
- Add a more powerful baseline setting for KD. (#305)
- Add `MethodInputsRecorder` and `FuncInputsRecorder` to record the input of a class method or a function.(#320)

NAS

- Support DSNAS, a nas algorithm that does not require retraining. (#226)

Tools

- Support configurable immediate feature map visualization. (#293) A useful tool is supported in this release to visualize the immediate features of a neural network. Please refer to our documents VisualizationUserGuide for more details.

9.3.3 Bug Fixes

- Fix the bug that `FunctionXXRecorder` and `FunctionXXDelivery` can not be pickled. (#320)

9.3.4 Ongoing changes

- Quantization: We are developing the basic interface of PTQ and QAT. RFC(Request for Comments) will be released soon.
- AutoSlim: AutoSlim is not yet available and is being refactored.
- Fx Pruning Tracer: Currently, the model topology can only be resolved through the backward tracer. In the future, both backward tracer and fx tracer will be supported.
- More Algorithms: BigNASAutoFormerGreedyNAS and Resrep will be released in the next few versions.
- Documentation: we will add more design docs, tutorials, and migration guidance so that the community can deep dive into our new design, participate the future development, and smoothly migrate downstream libraries to MMRazor 1.x.

9.3.5 Contributors

A total of 12 developers contributed to this release. Thanks @FreakieHuang @gaoyang07 @HIT-cwh @humu789 @LKJacky @pppppM @pprp @spynccat @sunnyxiaohu @wilxy @kitecats @SheffieldCao

9.4 v1.0.0rc0 (31/8/2022)

We are excited to announce the release of MMRazor 1.0.0rc0. MMRazor 1.0.0rc0 is the first version of MMRazor 1.x, a part of the OpenMMLab 2.0 projects. Built upon the new training engine, MMRazor 1.x simplified the interaction with other OpenMMLab repos, and upgraded the basic APIs of KD / Pruning / NAS. It also provides a series of knowledge distillation algorithms.

9.4.1 Highlights

- **New engines**. MMRazor 1.x is based on MMEngine, which provides a general and powerful runner that allows more flexible customizations and significantly simplifies the entrypoints of high-level interfaces.
- **Unified interfaces**. As a part of the OpenMMLab 2.0 projects, MMRazor 1.x unifies and refactors the interfaces and internal logic of train, testing, datasets, models, evaluation, and visualization. All the OpenMMLab 2.0 projects share the same design in those interfaces and logic to allow the emergence of multi-task/modality algorithms.
- **More configurable KD**. MMRazor 1.x add *Recorder* to get the data needed for KD more automatically*Delivery* to automatically pass the teacher's intermediate results to the student and connector to handle feature dimension mismatches between teacher and student.
- **More kinds of KD algorithms**. Benefitting from the powerful APIs of KD we have added several categories of KD algorithms, data-free distillation, self-distillation, and zero-shot distillation.
- **Unify the basic interface of NAS and Pruning**. We refactored *Mutable*, adding mutable value and mutable channel. Both NAS and Pruning can be developed based on mutables.
- **More documentation and tutorials**. We add a bunch of documentation and tutorials to help users get started more smoothly. Read it here.

9.4.2 Breaking Changes

Training and testing

- MMRazor 1.x runs on PyTorch>=1.6. We have deprecated the support of PyTorch 1.5 to embrace the mixed precision training and other new features since PyTorch 1.6. Some models can still run on PyTorch 1.5, but the full functionality of MMRazor 1.x is not guaranteed.
- MMRazor 1.x uses Runner in MMEngine rather than that in MMCV. The new Runner implements and unifies the building logic of dataset, model, evaluation, and visualizer. Therefore, MMRazor 1.x no longer maintains the building logics of those modules in `mmdet.train.apis` and `tools/train.py`. Those code have been migrated into MMEngine.
- The Runner in MMEngine also supports testing and validation. The testing scripts are also simplified, which has similar logic as that in training scripts to build the runner.

Configs

- The Runner in MMEngine uses a different config structures
- Config and model names

Components

- Algorithms
- Distillers
- Mutators
- Mutables
- Hooks

9.4.3 Improvements

- Support mixed precision training of all the models. However, some models may got Nan results due to some numerical issues. We will update the documentation and list their results (accuracy of failure) of mixed precision training.

9.4.4 Bug Fixes

- AutoSlim: Models of different sizes will no longer have the same size checkpoint

9.4.5 New Features

- Support Activation Boundaries Loss
- Support Be Your Own Teacher
- Support Data-Free Learning of Student Networks
- Support Data-Free Adversarial Distillation
- Support Decoupled Knowledge Distillation
- Support Factor Transfer
- Support FitNets
- Support Distilling the Knowledge in a Neural Network
- Support Overhaul
- Support Zero-shot Knowledge Transfer via Adversarial Belief Matching

9.4.6 Ongoing changes

- Quantization: We are developing the basic interface of PTQ and QAT. RFC(Request for Comments) will be released soon.
- AutoSlim: AutoSlim is not yet available and is being refactored.
- Fx Pruning Tracer: Currently, the model topology can only be resolved through the backward tracer. In the future, both backward tracer and fx tracer will be supported.
- More Algorithms: BigNASAutoFormerGreedyNAS and Resrep will be released in the next few versions.
- Documentation: we will add more design docs, tutorials, and migration guidance so that the community can deep dive into our new design, participate the future development, and smoothly migrate downstream libraries to MMRazor 1.x.

9.4.7 Contributors

A total of 13 developers contributed to this release. Thanks @FreakieHuang @gaoyang07 @HIT-cwh @humu789 @LKJacky @pppppM @pprp @spynccat @sunnyxiaohu @wilxy @wutongshenqiu @NickYangMin @Hiwyl Special thanks to @Davidgzx for his contribution to the data-free distillation algorithms

CHAPTER

TEN

CONTRIBUTE GUIDE

All kinds of contributions are welcome, including but not limited to the following.

- Fix typo or bugs
- Add documentation or translate the documentation into other languages
- Add new features and components

10.1 Workflow

1. fork and pull the latest OpenMMLab repository
2. checkout a new branch (do not use master branch for PRs)
3. commit your changes
4. create a PR

Note: If you plan to add some new features that involve large changes, it is encouraged to open an issue for discussion first.

10.2 Code style

10.2.1 Python

We adopt PEP8 as the preferred code style.

We use the following tools for linting and formatting:

- flake8: A wrapper around some linter tools.
- isort: A Python utility to sort imports.
- yapf: A formatter for Python files.
- codespell: A Python utility to fix common misspellings in text files.
- mdformat: Mdformat is an opinionated Markdown formatter that can be used to enforce a consistent style in Markdown files.
- docformatter: A formatter to format docstring.

Style configurations of yapf and isort can be found in setup.cfg.

We use pre-commit hook that checks and formats for `flake8`, `yapf`, `isort`, `trailing whitespaces`, `markdown files`, fixes `end-of-files`, `double-quoted-strings`, `python-encoding-pragma`, `mixed-line-ending`, sorts `requirments.txt` automatically on every commit. The config for a pre-commit hook is stored in .pre-commit-config.

After you clone the repository, you will need to install initialize pre-commit hook.

```
pip install -U pre-commit
```

From the repository folder

```
pre-commit install
```

After this on every commit check code linters and formatter will be enforced.

> Before you create a PR, make sure that your code lints and is formatted by yapf.

10.2.2 C++ and CUDA

We follow the Google C++ Style Guide.

CHAPTER

ELEVEN

FREQUENTLY ASKED QUESTIONS

CHAPTER

TWELVE

MMRAZOR.ENGINE

12.1 hooks

12.2 optimizers

12.3 runner

CHAPTER

THIRTEEN

MMRAZOR.MODELS

13.1 algorithms

13.2 architectures

13.3 distillers

13.4 losses

13.5 mutables

class mmrazor.models.mutables.**BaseMutable**(*alias: Optional[str] = None*, *init_cfg: Optional[Dict] = None*)

Base Class for mutables. Mutable means a searchable module widely used in Neural Architecture Search(NAS).

It mainly consists of some optional operations, and achieving searchable function by handling choice with `MUTATOR`.

All subclass should implement the following APIs:

- `fix_chosen()`
- `dump_chosen()`
- `current_choice.setter()`
- `current_choice.getter()`

Parameters

- **alias** (*str, optional*) – alias of the *MUTABLE*.
- **init_cfg** (*dict, optional*) – initialization configuration dict for `BaseModule`. OpenMMLab has implement 5 initializer including *Constant*, *Xavier*, *Normal*, *Uniform*, *Kaiming*, and *Pretrained*.

abstract property current_choice

Current choice will affect `forward()` and will be used in `mmrazor.core.subnet.utils.export_fix_subnet()` or mutator.

abstract dump_chosen() → mmrazor.utils.typing.DumpChosen
Save the current state of the mutable as a dictionary.

DumpChosen has chosen and meta fields. chosen is necessary, fix_chosen will use the chosen . meta is used to store some non-essential information.

abstract fix_chosen(*chosen*) → None
Fix mutable with chosen. This function would fix the chosen of mutable. The *is_fixed* will be set to True and only the selected operations can be retained. All subclasses must implement this method.

Note: This operation is irreversible.

---

property is_fixed: bool
whether the mutable is fixed.

Note:

If a mutable is fixed, it is no longer a searchable module, just a normal fixed module.

If a mutable is not fixed, it still is a searchable module.

Type bool

class mmrazor.models.mutables.**BaseMutableChannel**(*num_channels: int*, ***kwargs*)
BaseMutableChannel works as a channel mask for DynamicOps to select channels.

|----------| |mutable_in_channel(BaseMutableChannel) | |----------| | DynamicOp | |----------| |mutable_out_channel(BaseMutableChannel)| |----------|

All subclasses should implement the following APIs and the other abstract method in BaseMutable

- current_mask

Parameters **num_channels** (*int*) – number(dimension) of channels(mask).

property activated_channels: int
Number of activated channels.

abstract property current_mask: torch.Tensor
Return a mask indicating the channel selection.

dump_chosen() → mmrazor.utils.typing.DumpChosen
Dump chosen.

fix_chosen(*chosen=None*)
Fix the mutable with chosen.

num_choices() → int
Number of available choices.

class mmrazor.models.mutables.**DCFFChannelUnit**(*num_channels: int*, *candidate_choices: List[Union[int, float]] = [1.0]*, *choice_mode: str = 'ratio'*, *divisor: int = 1*, *min_value: int = 1*, *min_ratio: float = 0.9*)
DCFFChannelUnit is for supernet DCFF and based on OneShotMutableChannelUnit. In DCFF supernet, each module only has one choice. The channel choice is fixed before training.

Parameters

- **num_channels** (*int*) – The raw number of channels.
- **candidate_choices** (*List[Union[int, float]], optional*) – A list of candidate width numbers or ratios. Each candidate indicates how many channels to be reserved. Defaults to [1.0](choice_mode='number').
- **choice_mode** (*str, optional*) – Mode of candidates. One of "ratio" or "number". Defaults to 'ratio'.
- **divisor** (*int*) – Used to make choice divisible.
- **min_value** (*int*) – the minimal value used when make divisible.
- **min_ratio** (*float*) – the minimal ratio used when make divisible.

prepare_for_pruning(*model: torch.nn.modules.module.Module*)
In DCFFChannelGroup nn.Conv2d is replaced with FuseConv2d.

class mmrazor.models.mutables.**DMCPChannelUnit**(*num_channels: int*, *choice_mode: str = 'number'*, *divisor: int = 1*, *min_value: int = 1*, *min_ratio: float = 0.5*)

DMCPChannelUnit is for supernet DMCP and based on OneShotMutableChannelUnit. In DMCP supernet, each module only has one choice. The channel choice is fixed before training.

Note: In dmcpunit, a new attribute *activated_tensor_channels* is defined

---

in self.mutable_channel, which is specifically used to store the number of channels in the form of tensor. Defaults to None.

Parameters

- **num_channels** (*int*) – The raw number of channels.
- **choice_mode** (*str, optional*) – Mode of candidates. One of "ratio" or "number". Defaults to 'ratio'.
- **divisor** (*int*) – Used to make choice divisible.
- **min_value** (*int*) – the minimal value used when make divisible.
- **min_ratio** (*float*) – the minimal ratio used when make divisible.

prepare_for_pruning(*model: torch.nn.modules.module.Module*)
In DMCPChannelGroup nn.BatchNorm2d is replaced with DMCPBatchNorm2d.

class mmrazor.models.mutables.**DerivedMutable**(*choice_fn: Callable*, *mask_fn: Optional[Callable] = None*, *source_mutables: Optional[Iterable[*mmrazor.models.mutables.base_mutable.BaseMutable*]] = None*, *alias: Optional[str] = None*, *init_cfg: Optional[Dict] = None*)

Class for derived mutable.

A derived mutable is a mutable derived from other mutables that has *current_choice* and *current_mask* attributes (if any).

Note: A derived mutable does not have its own search space, so it is not legal to modify its *current_choice* or *current_mask* directly. And the only way to modify them is by modifying *current_choice* or *current_mask* in corresponding source mutables.

Parameters

- **choice_fn** (*callable*) – A closure that controls how to generate *current_choice*.
- **mask_fn** (*callable, optional*) – A closure that controls how to generate *current_mask*. Defaults to None.
- **source_mutables** (*iterable, optional*) – Specify source mutables for this derived mutable. If the argument is None, source mutables will be traced automatically by parsing mutables in closure variables. Defaults to None.
- **alias** (*str, optional*) – alias of the *MUTABLE*. Defaults to None.
- **init_cfg** (*dict, optional*) – initialization configuration dict for `BaseModule`. OpenMMLab has implement 5 initializer including *Constant*, *Xavier*, *Normal*, *Uniform*, *Kaiming*, and *Pretrained*. Defaults to None.

Examples

```
>>> from mmrazor.models.mutables import SquentialMutableChannel
>>> mutable_channel = SquentialMutableChannel(num_channels=3)
>>> # derive expand mutable
>>> derived_mutable_channel = mutable_channel * 2
>>> # source mutables will be traced automatically
>>> derived_mutable_channel.source_mutables
{SquentialMutableChannel(name=unbind, num_channels=3, current_choice=3)}  # noqa:␣
↪E501
>>> # modify `current_choice` of `mutable_channel`
>>> mutable_channel.current_choice = 2
>>> # `current_choice` and `current_mask` of derived mutable will be modified␣
↪automatically  # noqa: E501
>>> derived_mutable_channel
DerivedMutable(current_choice=4, activated_channels=4, source_mutables=
↪{SquentialMutableChannel(name=unbind, num_channels=3, current_choice=2)}, is_
↪fixed=False)  # noqa: E501
```

property current_choice
Current choice of derived mutable.

property current_mask: torch.Tensor
Current mask of derived mutable.

dump_chosen() → mmrazor.utils.typing.DumpChosen
Dump information of chosen.

Returns Dumped information.

Return type Dict

fix_chosen(*chosen*) → None
Fix mutable with subnet config.

> **Warning:** Fix derived mutable will have no actually effect.

property is_fixed: bool
Whether the derived mutable is fixed.

Note: Depends on whether all source mutables are already fixed.

static is_source_mutable(*mutable: object*) → bool
Judge whether an object is source mutable(not derived mutable).

Parameters **mutable** (*object*) – An object.

Returns Indicate whether the object is source mutable or not.

Return type bool

property num_choices: int
Number of all choices.

Note: Since derive mutable does not have its own search space, the number of choices will always be *1*.

Returns Number of choices.

Return type int

class mmrazor.models.mutables.**DiffChoiceRoute**(*edges: torch.nn.modules.container.ModuleDict*, *num_chosen: int = 2*, *with_arch_param: bool = False*, *alias: Optional[str] = None*, *init_cfg: Optional[Dict] = None*)
A type of MUTABLES for Neural Architecture Search, which can select inputs from different edges in a differentiable or non-differentiable way. It is commonly used in DARTS.

Parameters

- **edges** (*nn.ModuleDict*) – the key of *edges* is the name of different edges. The value of *edges* can be nn.Module or *DiffMutableModule*.
- **with_arch_param** (*bool*) – whether forward with arch_param. When set to *True*, a differentiable way is adopted. When set to *False*, a non-differentiable way is adopted.
- **alias** (*str, optional*) – alias of the *DiffChoiceRoute*.
- **init_cfg** (*dict, optional*) – initialization configuration dict for BaseModule. OpenMMLab has implement 6 initializers including *Constant*, *Xavier*, *Normal*, *Uniform*, *Kaiming*, and *Pretrained*.

Examples

```
>>> import torch
>>> import torch.nn as nn
>>> edges_dict=nn.ModuleDict()
>>> edges_dict.add_module('first_edge', nn.Conv2d(32, 32, 3, 1, 1))
>>> edges_dict.add_module('second_edge', nn.Conv2d(32, 32, 5, 1, 2))
>>> edges_dict.add_module('third_edge', nn.MaxPool2d(3, 1, 1))
>>> edges_dict.add_module('fourth_edge', nn.MaxPool2d(5, 1, 2))
>>> edges_dict.add_module('fifth_edge', nn.MaxPool2d(7, 1, 3))
>>> diff_choice_route_cfg = dict(
...     type="DiffChoiceRoute",
...     edges=edges_dict,
```

(continues on next page)

(continued from previous page)

```
...     with_arch_param=True,
... )
>>> arch_param
Parameter containing:
tensor([-6.1426e-04,  2.3596e-04,  1.4427e-03,  7.1668e-05,
    -8.9739e-04], requires_grad=True)
>>> x = [torch.randn(4, 32, 64, 64) for _ in range(5)]
>>> output=diffchoiceroute.forward_arch_param(x, arch_param)
>>> output.shape
torch.Size([4, 32, 64, 64])
```

property choices: List[str]
all choices.

Type list

dump_chosen() → mmrazor.utils.typing.DumpChosen
Save the current state of the mutable as a dictionary.

DumpChosen has chosen and meta fields. chosen is necessary, fix_chosen will use the chosen . meta is used to store some non-essential information.

fix_chosen(*chosen: List[str]*) → None
Fix mutable with *choice*. This operation would convert to *fixed* mode. The is_fixed will be set to True and only the selected operations can be retained.

Parameters **chosen** (*list(str)*) – the chosen key in MUTABLE.

forward(*x: Any*, *arch_param: Optional[torch.nn.parameter.Parameter] = None*)
Calls either *forward_fixed()* or *forward_arch_param()* depending on whether is_fixed() is True and whether arch_param() is None.

To reduce the coupling between *Mutable* and *Mutator*, the *arch_param* is generated by the *Mutator* and is passed to the forward function as an argument.

Note: *forward_fixed()* is called when in *fixed* mode. *forward_arch_param()* is called when in *unfixed* mode.

Parameters

- **x** (*Any*) – input data for forward computation.
- **arch_param** (*nn.Parameter, optional*) – the architecture parameters for DiffMutableModule.

Returns the result of forward

Return type Any

forward_all(*x*)
Forward all choices.

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

Returns the result of forward all of the choice operation.

Return type Tensor

forward_arch_param(*x*, *arch_param: torch.nn.parameter.Parameter*) → torch.Tensor
Forward with architecture parameters.

Parameters

- **x** (*list[Any] | tuple[Any]*) – x could be a list or a tuple of Torch.tensor, containing input data for forward selection.
- **arch_param** (*nn.Parameter*) – architecture parameters for for `DiffMutableModule`.

Returns the result of forward with `arch_param`.

Return type Tensor

forward_fixed(*inputs: Union[List, Tuple]*) → torch.Tensor
Forward when the mutable is in *fixed* mode.

Parameters **inputs** (*Union[List[Any], Tuple[Any]]*) – inputs could be a list or a tuple of Torch.tensor, containing input data for forward computation.

Returns the result of forward the fixed operation.

Return type Tensor

sample_choice(*arch_param: torch.Tensor*) → List[str]
sample choice based on *arch_param*.

class mmrazor.models.mutables.**DiffMutableModule**(***kwargs*)
Base class for differentiable mutables.

Parameters

- **module_kwargs** (*dict[str, dict], optional*) – Module initialization named arguments. Defaults to None.
- **alias** (*str, optional*) – alias of the *MUTABLE*.
- **init_cfg** (*dict, optional*) – initialization configuration dict for `BaseModule`. OpenMMLab has implement 5 initializer including *Constant*, *Xavier*, *Normal*, *Uniform*, *Kaiming*, and *Pretrained*.

Note: `forward_all()` is called when calculating FLOPs.

---

compute_arch_probs(*arch_param: torch.nn.parameter.Parameter*) → torch.Tensor
compute chosen probs according to architecture params.

forward(*x: Any*, *arch_param: Optional[torch.nn.parameter.Parameter] = None*)
Calls either `forward_fixed()` or *forward_arch_param()* depending on whether `is_fixed()` is `True` and whether `arch_param()` is None.

To reduce the coupling between *Mutable* and *Mutator*, the *arch_param* is generated by the *Mutator* and is passed to the forward function as an argument.

Note: `forward_fixed()` is called when in *fixed* mode. *forward_arch_param()* is called when in *unfixed* mode.

Parameters

- **x** (*Any*) – input data for forward computation.

- **arch_param** (*nn.Parameter, optional*) – the architecture parameters for DiffMutableModule.

Returns the result of forward

Return type Any

abstract forward_arch_param(*x*, *arch_param: torch.nn.parameter.Parameter*)
Forward when the mutable is not fixed.

All subclasses must implement this method.

abstract sample_choice(*arch_param: torch.Tensor*)
Sample choice according arch parameters.

set_forward_args(*arch_param: torch.nn.parameter.Parameter*) → None
Interface for modifying the arch_param using partial.

class mmrazor.models.mutables.**DiffMutableOP**(*candidates: Dict[str, Dict]*, *fix_threshold: float = 1.0*, *module_kwargs: Optional[Dict[str, Dict]] = None*, *alias: Optional[str] = None*, *init_cfg: Optional[Dict] = None*)
A type of MUTABLES for differentiable architecture search, such as DARTS. Search the best module by learnable parameters *arch_param*.

Parameters

- **candidates** (*dict[str, dict]*) – the configs for the candidate operations.
- **fix_threshold** (*float*) – The threshold that determines whether to fix the choice of current module as the op with the maximum *probs*. It happens when the maximum prob is *fix_threshold* or more higher then all the other probs. Default to 1.0.
- **module_kwargs** (*dict[str, dict], optional*) – Module initialization named arguments. Defaults to None.
- **alias** (*str, optional*) – alias of the *MUTABLE*.
- **init_cfg** (*dict, optional*) – initialization configuration dict for BaseModule. OpenMMLab has implement 5 initializer including *Constant*, *Xavier*, *Normal*, *Uniform*, *Kaiming*, and *Pretrained*.

property choices: List[str]
all choices.

Type list

dump_chosen() → mmrazor.utils.typing.DumpChosen
Save the current state of the mutable as a dictionary.

DumpChosen has chosen and meta fields. chosen is necessary, fix_chosen will use the chosen . meta is used to store some non-essential information.

fix_chosen(*chosen: Union[str, List[str]]*) → None
Fix mutable with *choice*. This operation would convert *unfixed* mode to *fixed* mode. The is_fixed will be set to True and only the selected operations can be retained.

Parameters **chosen** (*str*) – the chosen key in MUTABLE. Defaults to None.

forward_all(*x*) → torch.Tensor
Forward all choices. Used to calculate FLOPs.

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

Returns the result of forward all of the `choice` operation.

Return type Tensor

forward_arch_param(*x*, *arch_param: torch.nn.parameter.Parameter*) → torch.Tensor
Forward with architecture parameters.

Parameters

- **x** (*Any*) – x could be a Torch.tensor or a tuple of Torch.tensor, containing input data for forward computation.
- **arch_param** (*str, optional*) – architecture parameters for *DiffMutableModule*

Returns the result of forward with `arch_param`.

Return type Tensor

forward_fixed(*x*) → torch.Tensor
Forward when the mutable is in *fixed* mode.

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

Returns the result of forward the fixed operation.

Return type Tensor

sample_choice(*arch_param: torch.Tensor*) → str
Sample choice based on arch_parameters.

class mmrazor.models.mutables.**L1MutableChannelUnit**(*num_channels: int*, *choice_mode='number'*, *divisor=1*, *min_value=1*, *min_ratio=0.9*)
Implementation of L1-norm pruning algorithm. It compute the l1-norm of modules and preferly prune the modules with less l1-norm.

Please refer to papre *https://arxiv.org/pdf/1608.08710.pdf* for more detail.

property current_choice: Union[int, float]
return current choice.

class mmrazor.models.mutables.**MutableChannelContainer**(*num_channels: int*, ***kwargs*)
MutableChannelContainer inherits from BaseMutableChannel. However, it's not a single BaseMutableChannel, but a container for BaseMutableChannel. The mask of MutableChannelContainer consists of all masks of stored MutableChannels.

MutableChannelContainer |

Important interfaces:

register_mutable: register/store BaseMutableChannel in the MutableChannelContainer

property current_choice: torch.Tensor
Get current choices.

property current_mask: torch.Tensor
Return current mask.

register_mutable(*mutable_channel:* mmrazor.models.mutables.mutable_channel.base_mutable_channel.BaseMutableChannel, *start: int*, *end: int*)
Register/Store BaseMutableChannel in the MutableChannelContainer in the range [start,end)

classmethod register_mutable_channel_to_module(*module: mmrazor.models.architectures.dynamic_ops.mixins.dynamic_mixins.Dyna*, *mutable:* mmrazor.models.mutables.mutable_channel.base_mutable_channel.Basel, *is_to_output_channel=True*, *start=0*, *end=- 1*)

Register a BaseMutableChannel to a module with MutableChannelContainers.

class mmrazor.models.mutables.**MutableChannelUnit**(*num_channels: int*, ***kwargs*)

config_template(*with_init_args=False*, *with_channels=False*) → Dict

Return the config template of this unit. By default, the config template only includes a key 'choice'.

Parameters

- **with_init_args** (*bool*) – if the config includes args for initialization.
- **with_channels** (*bool*) – if the config includes info about channels. the config with info about channels can used to parse channel units without tracer.

property current_choice

Choice of this unit.

fix_chosen(*choice=None*)

Make the channels in this unit fixed.

classmethod init_from_cfg(*model: torch.nn.modules.module.Module*, *config: Dict*)

init a Channel using a config which can be generated by self.config_template(), include init choice.

classmethod init_from_predefined_model(*model: torch.nn.modules.module.Module*)

Initialize units using the model with pre-defined dynamicops and mutable-channels.

property is_mutable: bool

If the channel-unit is prunable.

property mutable_prefix: str

Mutable prefix.

abstract prepare_for_pruning(*model*)

Post process after parse units.

For example, we need to register mutables to dynamic-ops.

abstract sample_choice()

Randomly sample a valid choice and return.

class mmrazor.models.mutables.**MutableValue**(*value_list: List[Union[int, float]]*, *default_value: Optional[Any] = None*, *alias: Optional[str] = None*, *init_cfg: Optional[Dict] = None*)

Base class for mutable value.

A mutable value is actually a mutable that adds some functionality to a list containing objects of the same type.

Parameters

- **value_list** (*list*) – List of value, each value must have the same type.
- **default_value** (*any, optional*) – Default value, must be one in *value_list*. Default to None.
- **alias** (*str, optional*) – alias of the *MUTABLE*.

- **init_cfg** (*dict, optional*) – initialization configuration dict for BaseModule. OpenMMLab has implement 5 initializer including *Constant*, *Xavier*, *Normal*, *Uniform*, *Kaiming*, and *Pretrained*.

property choices: List[Any]
List of choices.

property current_choice: Union[int, float]
Current choice of mutable value.

dump_chosen() → mmrazor.utils.typing.DumpChosen
Dump information of chosen.

Returns Dumped information.

Return type Dict[str, Any]

fix_chosen(*chosen: Union[int, float]*) → None
Fix mutable value with subnet config.

Parameters **chosen** (*dict*) – the information of chosen.

property mutable_prefix: str
Mutable prefix.

property num_choices: int
Number of all choices.

Returns Number of choices.

Return type int

class mmrazor.models.mutables.**OneHotMutableOP**(*candidates: Dict[str, Dict]*, *fix_threshold: float = 1.0*, *module_kwargs: Optional[Dict[str, Dict]] = None*, *alias: Optional[str] = None*, *init_cfg: Optional[Dict] = None*)
A type of MUTABLES for one-hot sample based architecture search, such as DSNAS. Search the best module by learnable parameters *arch_param*.

Parameters

- **candidates** (*dict[str, dict]*) – the configs for the candidate operations.
- **module_kwargs** (*dict[str, dict], optional*) – Module initialization named arguments. Defaults to None.
- **alias** (*str, optional*) – alias of the *MUTABLE*.
- **init_cfg** (*dict, optional*) – initialization configuration dict for BaseModule. OpenMMLab has implement 5 initializer including *Constant*, *Xavier*, *Normal*, *Uniform*, *Kaiming*, and *Pretrained*.

forward_arch_param(*x: Any*, *arch_param: torch.nn.parameter.Parameter*) → torch.Tensor
Forward with architecture parameters.

Parameters

- **x** (*Any*) – x could be a Torch.tensor or a tuple of Torch.tensor, containing input data for forward computation.
- **arch_param** (*str, optional*) – architecture parameters for *DiffMutableModule*.

Returns the result of forward with arch_param.

Return type Tensor

sample_weights(*arch_param: torch.nn.parameter.Parameter*, *probs: torch.Tensor*, *random_sample: bool = False*) → torch.Tensor

Use one-hot distributions to sample the arch weights based on the arch params.

Parameters

- **arch_param** (*nn.Parameter*) – architecture parameters for *DiffMutableModule*.
- **probs** (*Tensor*) – the probs of choice.
- **random_sample** (*bool*) – Whether to random sample arch weights or not Defaults to False.

Returns Sampled one-hot arch weights.

Return type Tensor

class mmrazor.models.mutables.**OneShotMutableChannel**(*num_channels: int*, *candidate_choices: List[Union[int, float]] = []*, *choice_mode='number'*, ***kwargs*)

OneShotMutableChannel is a subclass of SquentialMutableChannel. The difference is that a OneShotMutableChannel limits the candidates of the choice.

Parameters

- **num_channels** (*int*) – number of channels.
- **candidate_choices** (*List[Union[float, int]], optional*) – A list of candidate width ratios. Each candidate indicates how many channels to be reserved. Defaults to [].
- **choice_mode** (*str, optional*) – Mode of choices. Defaults to 'number'.

property current_choice: Union[int, float]

Get current choice.

class mmrazor.models.mutables.**OneShotMutableChannelUnit**(*num_channels: int*, *candidate_choices: List[Union[int, float]] = [0.5, 1.0]*, *choice_mode='ratio'*, *divisor=1*, *min_value=1*, *min_ratio=0.9*)

OneShotMutableChannelUnit is for single path supernet such as AutoSlim. In single path supernet, each module only has one choice invoked at the same time. A path is obtained by sampling all the available choices. It is the base class for one shot mutable channel.

Parameters

- **num_channels** (*_type_*) – The raw number of channels.
- **candidate_choices** (*List[Union[int, float]], optional*) – A list of candidate width ratios. Each candidate indicates how many channels to be reserved. Defaults to [0.5, 1.0](choice_mode='ratio').
- **choice_mode** (*str, optional*) – Mode of candidates. One of "ratio" or "number". Defaults to 'ratio'.
- **divisor** (*int*) – Used to make choice divisible.
- **min_value** (*int*) – the minimal value used when make divisible.
- **min_ratio** (*float*) – the minimal ratio used when make divisible.

config_template(*with_init_args=False*, *with_channels=False*) → Dict

Config template of the OneShotMutableChannelUnit.

property current_choice: Union[int, float]

Get current choice.

property max_choice: Union[int, float]
Get Maximal choice.

property min_choice: Union[int, float]
Get Minimal choice.

prepare_for_pruning(*model: torch.nn.modules.module.Module*)
Prepare for pruning.

sample_choice() → Union[int, float]
Sample a valid choice.

class mmrazor.models.mutables.**OneShotMutableModule**(*module_kwargs: Optional[Dict[str, Dict]] = None*, *alias: Optional[str] = None*, *init_cfg: Optional[Dict] = None*)
Base class for one shot mutable module. A base type of MUTABLES for single path supernet such as Single Path One Shot.

All subclass should implement the following APIs and the other abstract method in MutableModule:

- sample_choice()
- forward_choice()

Note: forward_all() is called when calculating FLOPs.

---

forward(*x: Any*) → Any
Calls either forward_fixed() or *forward_choice()* depending on whether is_fixed() is True and whether current_choice() is None.

Note: forward_fixed() is called in *fixed* mode. forward_all() is called in *unfixed* mode with current_choice() is None.

***forward_choice()* is called in *unfixed* mode with** current_choice() is not None.

Parameters

- **x** (*Any*) – input data for forward computation.
- **choice** (*CHOICE_TYPE, optional*) – the chosen key in MUTABLE.

Returns the result of forward

Return type Any

abstract forward_choice(*x*, *choice: str*)
Forward with the unfixed mutable and current_choice is not None.

All subclasses must implement this method.

abstract sample_choice() → str
Sample random choice.

Returns the chosen key in MUTABLE.

Return type str

class mmrazor.models.mutables.**OneShotMutableOP**(*candidates: Union[Dict[str, Dict], torch.nn.modules.container.ModuleDict]*, *module_kwargs: Optional[Dict[str, Dict]] = None*, *alias: Optional[str] = None*, *init_cfg: Optional[Dict] = None*)

A type of MUTABLES for single path supernet, such as Single Path One Shot. In single path supernet, each choice block only has one choice invoked at the same time. A path is obtained by sampling all the choice blocks.

Parameters

- **candidates** (*dict[str, dict]*) – the configs for the candidate operations.
- **module_kwargs** (*dict[str, dict], optional*) – Module initialization named arguments. Defaults to None.
- **alias** (*str, optional*) – alias of the *MUTABLE*.
- **init_cfg** (*dict, optional*) – initialization configuration dict for BaseModule. OpenMMLab has implement 5 initializer including *Constant*, *Xavier*, *Normal*, *Uniform*, *Kaiming*, and *Pretrained*.

Examples

```
>>> import torch
>>> from mmrazor.models.mutables import OneShotMutableOP
```

```
>>> candidates = nn.ModuleDict({
...     'conv3x3': nn.Conv2d(32, 32, 3, 1, 1),
...     'conv5x5': nn.Conv2d(32, 32, 5, 1, 2),
```

```
>>> input = torch.randn(1, 32, 64, 64)
>>> op = OneShotMutableOP(candidates)
```

```
>>> op.choices
['conv3x3', 'conv5x5', 'conv7x7']
>>> op.num_choices
3
>>> op.is_fixed
False
```

```
>>> op.current_choice = 'conv3x3'
>>> unfix_output = op.forward(input)
>>> torch.all(unfixed_output == candidates['conv3x3'](input))
True
```

```
>>> op.fix_chosen('conv3x3')
>>> fix_output = op.forward(input)
>>> torch.all(fix_output == unfix_output)
True
```

```
>>> op.choices
['conv3x3']
>>> op.num_choices
```

(continues on next page)

(continued from previous page)

```
1
>>> op.is_fixed
True
```

property choices: List[str]
all choices.

Type list

dump_chosen() → mmrazor.utils.typing.DumpChosen
Save the current state of the mutable as a dictionary.

DumpChosen has chosen and meta fields. chosen is necessary, fix_chosen will use the chosen . meta is used to store some non-essential information.

fix_chosen(*chosen: str*) → None
Fix mutable with subnet config. This operation would convert *unfixed* mode to *fixed* mode. The is_fixed will be set to True and only the selected operations can be retained.

Parameters **chosen** (*str*) – the chosen key in MUTABLE. Defaults to None.

forward_all(*x*) → torch.Tensor
Forward all choices. Used to calculate FLOPs.

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

Returns the result of forward all of the choice operation.

Return type Tensor

forward_choice(*x*, *choice: str*) → torch.Tensor
Forward with the *unfixed* mutable and current choice is not None.

Parameters

- **x** (*Any*) – x could be a Torch.tensor or a tuple of Torch.tensor, containing input data for forward computation.
- **choice** (*str*) – the chosen key in OneShotMutableOP.

Returns the result of forward the choice operation.

Return type Tensor

forward_fixed(*x: Any*) → torch.Tensor
Forward with the *fixed* mutable.

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

Returns the result of forward the fixed operation.

Return type Tensor

sample_choice() → str
uniform sampling.

class mmrazor.models.mutables.**OneShotMutableValue**(*value_list: List[Any]*, *default_value: Optional[Any] = None*, *alias: Optional[str] = None*, *init_cfg: Optional[Dict] = None*)
Class for one-shot mutable value.

one-shot mutable value provides *sample_choice* method and *min_choice*, *max_choice* properties on the top of mutable value.

Parameters

- **value_list** (*list*) – List of value, each value must have the same type.
- **default_value** (*any, optional*) – Default value, must be one in *value_list*. Default to None.
- **alias** (*str, optional*) – alias of the *MUTABLE*.
- **init_cfg** (*dict, optional*) – initialization configuration dict for `BaseModule`. OpenMMLab has implement 5 initializer including *Constant*, *Xavier*, *Normal*, *Uniform*, *Kaiming*, and *Pretrained*.

property max_choice: Any
Max choice of all choices.

Returns Max choice.

Return type Any

property min_choice: Any
Min choice of all choices.

Returns Min choice.

Return type Any

sample_choice() → Any
Random sampling from choices.

Returns Selected choice.

Return type Any

class mmrazor.models.mutables.**SequentialMutableChannelUnit**(*num_channels: int*, *choice_mode='number'*, *divisor=1*, *min_value=1*, *min_ratio=0.9*)
SequentialMutableChannelUnit accepts a intger(number) or float(ratio) as the choice, which indicates how many of the channels are remained from left to right, like 11110000.

Parameters

- **num_channels** (*int*) – number of channels.
- **choice_mode** (*str*) – mode of choice, which is one of 'number' or 'ratio'.
- **divisor** (*int*) – Used to make choice divisible.
- **min_value** (*int*) – the minimal value used when make divisible.
- **min_ratio** (*float*) – the minimal ratio used when make divisible.

config_template(*with_init_args=False*, *with_channels=False*) → Dict
Template of config.

property current_choice: Union[int, float]
return current choice.

fix_chosen(*choice=None*)
fix chosen.

prepare_for_pruning(*model: torch.nn.modules.module.Module*)
Prepare for pruning, including register mutable channels.

sample_choice() → Union[int, float]
Sample a choice in (0,1]

class mmrazor.models.mutables.**SimpleMutableChannel**(*num_channels: int*, ***kwargs*)
SimpleMutableChannel is a simple BaseMutableChannel, it directly take a mask as a choice.

Parameters **num_channels** (*int*) – number of channels.

property current_choice: torch.Tensor
Get current choice.

property current_mask: torch.Tensor
Get current mask.

expand_mutable_channel(*expand_ratio: Union[int, float]*) → *mmrazor.models.mutables.derived_mutable.DerivedMutable*
Get a derived SimpleMutableChannel with expanded mask.

class mmrazor.models.mutables.**SlimmableChannelUnit**(*num_channels: int*, *candidate_choices: List[Union[int, float]] = []*, *choice_mode='number'*, *divisor=1*, *min_value=1*, *min_ratio=0.9*)
A type of `MutableChannelUnit` to train several subnets together.

Parameters

- **num_channels** (*int*) – The raw number of channels.
- **candidate_choices** (*List[Union[int, float]], optional*) – A list of candidate width ratios. Each candidate indicates how many channels to be reserved. Defaults to [0.5, 1.0](choice_mode='ratio').
- **choice_mode** (*str, optional*) – Mode of candidates. One of 'ratio' or 'number'. Defaults to 'number'.
- **divisor** (*int, optional*) – Used to make choice divisible.
- **min_value** (*int, optional*) – The minimal value used when make divisible.
- **min_ratio** (*float, optional*) – The minimal ratio used when make divisible.

alter_candidates_of_switchbn(*candidates: List*)
Change candidates of SwitchableBatchNorm2d.

prepare_for_pruning(*model: torch.nn.modules.module.Module*)
Prepare for pruning.

class mmrazor.models.mutables.**SquentialMutableChannel**(*num_channels: int*, *choice_mode='number'*, ***kwargs*)
SquentialMutableChannel defines a BaseMutableChannel which switch off channel mask from right to left sequentially, like '11111000'.

A choice of SquentialMutableChannel is an integer, which indicates how many channel are activated from left to right.

Parameters **num_channels** (*int*) – number of channels.

property current_choice: Union[int, float]
Get current choice.

property current_mask: torch.Tensor
Return current mask.

fix_chosen(*chosen=Ellipsis*)
Fix chosen.

property is_num_mode
Get if the choice is number mode.

13.6 mutators

13.7 ops

13.8 task_modules

13.9 utils

mmrazor.models.utils.**add_prefix**(*inputs: Dict*, *prefix: str*) → Dict
Add prefix for dict.

Parameters

- **inputs** (*dict*) – The input dict with str keys.
- **prefix** (*str*) – The prefix to add.

Returns The dict with keys updated with `prefix`.

Return type dict

mmrazor.models.utils.**get_module_device**(*module: torch.nn.modules.module.Module*) → torch.device
Get the device of a module.

Parameters **module** (*nn.Module*) – A module contains the parameters.

mmrazor.models.utils.**make_divisible**(*value: int*, *divisor: int*, *min_value: Optional[int] = None*, *min_ratio: float = 0.9*) → int
Make divisible function.

This function rounds the channel number down to the nearest value that can be divisible by the divisor.

Parameters

- **value** (*int*) – The original channel number.
- **divisor** (*int*) – The divisor to fully divide the channel number.
- **min_value** (*int, optional*) – The minimum value of the output channel. Default: None, means that the minimum value equal to the divisor.
- **min_ratio** (*float*) – The minimum ratio of the rounded channel number to the original channel number. Default: 0.9.

Returns The modified output channel number

Return type int

mmrazor.models.utils.**parse_values**(*candidate_lists: List[list]*)
Parse a list with format *(min_range, max_range, step)*.

NOTE: this method is required when customizing search space in configs.

mmrazor.models.utils.**pop_rewriter_function_record**(*rewriter_context*, *function_record_to_pop*)

Delete user-specific rewriters from *RewriterContext._rewriter_manager*.

We use the model which is rewritten by mmdeploy to build quantized models. However not all the functions rewritten by mmdeploy need to be rewritten in mmrazor. For example, mmdeploy rewrite *mmcls.models.classifiers.ImageClassifier.forward* and *mmcls.models.classifiers.BaseClassifier.forward* for deployment. But they can't be rewritten by mmrazor as ptq and qat are done in mmrazor. So to ensure ptq and qat proceed normally, we have to remove these record from *RewriterContext._rewriter_manager*.

mmrazor.models.utils.**set_requires_grad**(*nets: Union[torch.nn.modules.module.Module, List[torch.nn.modules.module.Module]]*, *requires_grad: bool = False*) → None

Set requires_grad for all the networks.

Parameters

- **nets** (*nn.Module | list[nn.Module]*) – A list of networks or a single network.
- **requires_grad** (*bool*) – Whether the networks require gradients or not

CHAPTER

FOURTEEN

MMRAZOR.REGISTRY

CHAPTER

FIFTEEN

MMRAZOR.STRUCTURES

15.1 delivery

15.2 graph

15.3 recorder

15.4 subnet

class mmrazor.structures.subnet.**Candidates**(*initdata: Optional[Union[Dict, List[Dict], Dict[str, Dict], List[Dict[str, Dict]]]] = None*)

The data structure of sampled candidate. The format is Union[Dict[str, Dict], List[Dict[str, Dict]]]. .. rubric:: Examples

```
>>> candidates = Candidates()
>>> subnet_1 = {'1': 'choice1', '2': 'choice2'}
>>> candidates.append(subnet_1)
>>> candidates
[{"{'1': 'choice1', '2': 'choice2'}":
{'score': 0.0, 'flops': 0.0, 'params': 0.0, 'latency': 0.0}}]
>>> candidates.set_resources(0, 49.9, 'flops')
>>> candidates.set_score(0, 100.)
>>> candidates
[{"{'1': 'choice1', '2': 'choice2'}":
{'score': 100.0, 'flops': 49.9, 'params': 0.0, 'latency': 0.0}}]
>>> subnet_2 = {'choice_3': 'layer_3', 'choice_4': 'layer_4'}
>>> candidates.append(subnet_2)
>>> candidates
[{"{'1': 'choice1', '2': 'choice2'}":
{'score': 100.0, 'flops': 49.9, 'params': 0.0, 'latency': 0.0}},
{"{'choice_3': 'layer_3', 'choice_4':'layer_4'}":
{'score': 0.0, 'flops': 0.0, 'params': 0.0, 'latency': 0.0}}]
>>> candidates.subnets
[{'1': 'choice1', '2': 'choice2'},
{'choice_3': 'layer_3', 'choice_4': 'layer_4'}]
>>> candidates.resources('flops')
[49.9, 0.0]
```

(continues on next page)

(continued from previous page)

```
>>> candidates.scores
[100.0, 0.0]
```

append(*item: Union[Dict, List[Dict], Dict[str, Dict], List[Dict[str, Dict]]]*) → None
Append operation.

extend(*other: Any*) → None
Extend operation.

insert(*i: int*, *item: Union[Dict, List[Dict], Dict[str, Dict], List[Dict[str, Dict]]]*) → None
Insert operation.

resources(*key_indicator: str = 'flops'*) → List[float]
The resources of candidates.

property scores: List[float]
The scores of candidates.

set_resource(*i: int*, *resources: float*, *key_indicator: str = 'flops'*) → None
Set resources to the specified subnet by index.

set_score(*i: int*, *score: float*) → None
Set score to the specified subnet by index.

sort_by(*key_indicator: str = 'score'*, *reverse: bool = True*) → None
Sort by a specific indicator in descending order.

Parameters

- **key_indicator** (*str*) – sort all candidates by key_indicator. Defaults to 'score'.
- **reverse** (*bool*) – sort all candidates in descending order.

property subnets: List[Dict]
The subnets of candidates.

update_resources(*resources: list*, *start: int = 0*) → None
Update resources to the specified candidate.

mmrazor.structures.subnet.**convert_fix_subnet**(*fix_subnet: Dict[str, mmrazor.utils.typing.DumpChosen]*)
Convert the fixed subnet to avoid python typing error.

mmrazor.structures.subnet.**export_fix_subnet**(*model: torch.nn.modules.module.Module*, *export_subnet_mode: str = 'mutable'*, *slice_weight: bool = False*, *export_channel: bool = False*) → Tuple[Dict[str, Any], Optional[Dict]]
Export subnet that can be loaded by *load_fix_subnet()*. Include subnet structure and subnet weight.

Parameters

- **model** (*nn.Module*) – The target model to export.
- **export_subnet_mode** (*bool*) – Subnet export method choice. Export by *mutable.dump_chosen()* when set to 'mutable' (NAS) Export by *mutator.config_template()* when set to 'mutator' (Prune)
- **slice_weight** (*bool*) – Export subnet weight. Default to False.
- **export_channel** (*bool*) – Whether to export the mutator's channel. Often required when finetune is needed for the exported subnet. Default to False.

Returns

Exported subnet choice config. static_model (Optional[Dict]): Exported static model state_dict.

Valid when `slice_weight`=True.

Return type fix_subnet (ValidFixMutable)

mmrazor.structures.subnet.**load_fix_subnet**(*model: torch.nn.modules.module.Module*, *subnet_dict: Union[str, pathlib.Path, Dict[str, Any]]*, *load_subnet_mode: str = 'mutable'*, *prefix: str = ''*, *extra_prefix: str = ''*) → None

Load fix subnet.

15.5 tracer

CHAPTER

SIXTEEN

MMRAZOR.UTILS

class mmrazor.utils.**IndexDict**

IndexDict inherits from OrderedDict[Tuple[int, int], VT]. Each IndexDict object is a OrderDict object which using index(Tuple[int,int]) as key and Any as value.

The key type is Tuple[a: int,b: int]. It indicates a range in the [a,b).

IndexDict has three features: 1. ensure a key always is a index(Tuple[int,int]). 1. ensure the the indexes are sorted by ascending order. 2. ensure there is no overlap among indexes.

class mmrazor.utils.**RuntimeInfo**

A tools to get runtime info in MessageHub.

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.**get_package_placeholder**(*string: str*) → object

Get placeholder instance which can avoid raising errors when down-stream dependency is not installed properly.

Parameters **string** (*str*) – the dependency's name, i.e. *mmcls*

Raises **ImportError** – raise it when the dependency is not installed properly.

Returns PlaceHolder instance.

Return type object

mmrazor.utils.**get_placeholder**(*string: str*) → object

Get placeholder instance which can avoid raising errors when down-stream dependency is not installed properly.

Parameters **string** (*str*) – the dependency's name, i.e. *mmcls*

Raises **ImportError** – raise it when the dependency is not installed properly.

Returns PlaceHolder instance.

Return type object

mmrazor.utils.**register_all_modules**(*init_default_scope: bool = True*) → None
Register all modules in mmrazor into the registries.

Parameters **init_default_scope** (*bool*) – Whether initialize the mmrazor default scope. When *init_default_scope=True*, the global default scope will be set to *mmrazor*, and all registries will build modules from mmrazor's registry node. To understand more about the registry, please refer to https://github.com/open-mmlab/mmengine/blob/main/docs/en/tutorials/registry.md Defaults to True.

mmrazor.utils.**setup_multi_processes**(*cfg*)
Setup multi-processing environment variables.

CHAPTER

SEVENTEEN

ENGLISH

CHAPTER

EIGHTEEN

CHAPTER

NINETEEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

m

INDEX

D

E

F

G

I

L

M

N

O

P

R

S

U