MMEditing

MMEditing Authors

Dec 08, 2022

GET STARTED

Languages: English |

MMEditing is an open-source toolbox for image and video processing, editing and synthesis.

MMEditing supports various foundamental generative models, including:

- Unconditional Generative Adversarial Networks (GANs)
- Conditional Generative Adversarial Networks (GANs)
- Internal Learning
- Diffusion Models
- And many other generative models are coming soon!

MMEditing supports various applications, including:

- Image super-resolution
- Video super-resolution
- Video frame interpolation
- Image inpainting
- Image matting
- Image-to-image translation
- And many other applications are coming soon!

MMEditing is based on PyTorch and is a part of the OpenMMLab project. Codes are available on GitHub.

CHAPTER

ONE

DOCUMENTATION

1.1 Overview

Welcome to MMEditing! In this section, you will know about

- *What is MMEditing?*
- *Why should I use MMEditing?*
- *Get started*
- *User guides*
- *Advanced guides*

1.1.1 What is MMEditing?

MMEditing is an open-source toolbox for professional AI researchers and machine learning engineers to explore image and video processing, editing and synthesis.

MMEditing allows researchers and engineers to use pre-trained state-of-the-art models, train and develop new customized models easily.

MMEditing supports various foundamental generative models, including:

- Unconditional Generative Adversarial Networks (GANs)
- Conditional Generative Adversarial Networks (GANs)
- Internal Learning
- Diffusion Models
- And many other generative models are coming soon!

MMEditing supports various applications, including:

- Image super-resolution
- Video super-resolution
- Video frame interpolation
- Image inpainting
- Image matting
- Image-to-image translation
- And many other applications are coming soon!

1.1.2 Why should I use MMEditing?

- **State of the Art**

  MMEditing provides state-of-the-art generative models to process, edit and synthesize images and videos.

- **Powerful and Popular Applications**

  MMEditing supports popular and contemporary *inpainting*, *matting*, *super-resolution* and *generation* applications. Specifically, MMEditing supports GAN interpolation, GAN projection, GAN manipulations and many other popular GAN's applications. It's time to play with your GANs!

- **New Modular Design for Flexible Combination:**

  We decompose the editing framework into different modules and one can easily construct a customized editor framework by combining different modules. Specifically, a new design for complex loss modules is proposed for customizing the links between modules, which can achieve flexible combinations among different modules.(Tutorial for *losses*)

- **Efficient Distributed Training:**

  With the support of MMSeparateDistributedDataParallel, distributed training for dynamic architectures can be easily implemented.

1.1.3 Get started

For installation instructions, please see *get_started*.

1.1.4 User guides

For beginners, we suggest learning the basic usage of MMEditing from *user_guides*.

Advanced guides

For users who are familiar with MMEditing, you may want to learn the design of MMEditing, as well as how to extend the repo, how to use multiple repos and other advanced usages, please refer to *advanced_guides*.

1.2 Get Started: Install and Run MMEditing

In this section, you will know about:

- *Installation*
  - *Prerequisites*
  - *Best practices*
  - *Customize installation*
  - *Developing with multiple MMEditing versions*
- *Quick run*

1.2.1 Installation

We recommend that users follow our *Best practices* to install MMEditing 1.x. However, the whole process is highly customizable. See *Customize installation* section for more information.

Prerequisites

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

MMEditing works on Linux, Windows, and macOS. It requires:

- Python >= 3.6
- PyTorch >= 1.5
- MMCV >= 2.0.0rc1

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 official website.

Step 1. Create a conda environment and activate it

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

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

- On GPU platforms:

  ```
  conda install pytorch torchvision cudatoolkit=11.3 -c pytorch
  ```

- On CPU platforms:

  ```
  conda install pytorch=1.10 torchvision cpuonly -c pytorch
  ```

Best practices

Step 0. Install MMCV using MIM.

```
pip install -U openmim
mim install 'mmcv>=2.0.0rc1'
```

Step 1. Install MMEngine.

```
pip install git+https://github.com/open-mmlab/mmengine.git
```

Step 2. Install MMEditing 1.x . Install MMEditing from the source code.

```
git clone -b 1.x https://github.com/open-mmlab/mmediting.git
cd mmediting
pip3 install -e . -v
```

Step 5. Verification.

```
cd ~
python -c "import mmedit; print(mmedit.__version__)"
# Example output: 1.0.0rc1
```

The installation is successful if the version number is output correctly.

Note: You may be curious about what `-e .` means when supplied with `pip install`. Here is the description:

- `-e` means editable mode. When `import mmedit`, modules under the cloned directory are imported. If `pip install` without `-e`, pip will copy cloned codes to somewhere like `lib/python/site-package`. Consequently, modified code under the cloned directory takes no effect unless `pip install` again. Thus, `pip install` with `-e` is particularly convenient for developers. If some codes are modified, new codes will be imported next time without reinstallation.
- `.` means code in this directory

You can also use `pip install -e .[all]`, which will install more dependencies, especially for pre-commit hooks and unittests.

Customize installation

CUDA Version

When installing PyTorch, you need to specify the version of CUDA. If you are not clear on which to choose, follow our recommendations:

- For Ampere-based NVIDIA GPUs, such as GeForce 30 series and NVIDIA A100, CUDA 11 is a must.
- For older NVIDIA GPUs, CUDA 11 is backward compatible, but CUDA 10.2 offers better compatibility and is more lightweight.

Please make sure the GPU driver satisfies the minimum version requirements. See this table for more information.

note Installing CUDA runtime libraries is enough if you follow our best practices, because no CUDA code will be compiled locally. However, if you hope to compile MMCV from source or develop other CUDA operators, you need to install the complete CUDA toolkit from NVIDIA's website, and its version should match the CUDA version of PyTorch. i.e., the specified version of cudatoolkit in `conda install` command.

Install MMCV without MIM

MMCV contains C++ and CUDA extensions, thus depending on PyTorch in a complex way. MIM solves such dependencies automatically and makes the installation easier. However, it is not a must.

To install MMCV with pip instead of MIM, please follow MMCV installation guides. This requires manually specifying a find-url based on PyTorch version and its CUDA version.

For example, the following command install mmcv-full built for PyTorch 1.10.x and CUDA 11.3.

```
pip install 'mmcv>=2.0.0rc1' -f https://download.openmmlab.com/mmcv/dist/cu113/torch1.10/
↪index.html
```

Using MMEditing with Docker

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

```
# build an image with PyTorch 1.8, CUDA 11.1
# If you prefer other versions, just modified the Dockerfile
docker build -t mmediting docker/
```

Run it with

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

Trouble shooting

If you have some issues during the installation, please first view the *FAQ* page. You may open an issue on GitHub if no solution is found.

Developing with multiple MMEditing versions

The train and test scripts already modify the `PYTHONPATH` to ensure the script uses the `MMEditing` in the current directory.

To use the default MMEditing installed in the environment rather than that you are working with, you can remove the following line in those scripts

```
PYTHONPATH="$(dirname $0)/..":$PYTHONPATH
```

1.2.2 Quick run

After installing MMEditing successfully, now you are able to play with MMEditing!

To synthesize an image of a church, you only need several lines of codes by MMEditing!

```
from mmedit.apis import init_model, sample_unconditional_model

config_file = 'configs/styleganv2/stylegan2_c2_8xb4-800kiters_lsun-church-256x256.py'
# you can download this checkpoint in advance and use a local file path.
checkpoint_file = 'https://download.openmmlab.com/mmgen/stylegan2/official_weights/
↪stylegan2-church-config-f-official_20210327_172657-1d42b7d1.pth'
device = 'cuda:0'
# init a generative model
model = init_model(config_file, checkpoint_file, device=device)
# sample images
fake_imgs = sample_unconditional_model(model, 4)
```

Or you can just run the following command.

```
python demo/unconditional_demo.py \
configs/styleganv2/stylegan2_c2_lsun-church_256_b4x8_800k.py \
https://download.openmmlab.com/mmgen/stylegan2/official_weights/stylegan2-church-config-
↪f-official_20210327_172657-1d42b7d1.pth
```

You will see a new image `unconditional_samples.png` in folder `work_dirs/demos/`, which contained generated samples.

What's more, if you want to make these photos much more clear, you only need several lines of codes for image super-resolution by MMEditing!

```
import mmcv
from mmedit.apis import init_model, restoration_inference
from mmedit.engine.misc import tensor2img

config = 'configs/esrgan/esrgan_x4c64b23g32_1xb16-400k_div2k.py'
checkpoint = 'https://download.openmmlab.com/mmediting/restorers/esrgan/esrgan_
↪x4c64b23g32_1x16_400k_div2k_20200508-f8ccaf3b.pth'
img_path = 'tests/data/image/lq/baboon_x4.png'
model = init_model(config, checkpoint)
output = restoration_inference(model, img_path)
output = tensor2img(output)
mmcv.imwrite(output, 'output.png')
```

Now, you can check your fancy photos in `output.png`.

1.3 Tutorial 1: Learn about Configs in MMEditing

We incorporate modular and inheritance design into our config system, which is convenient to conduct various experiments. If you wish to inspect the config file, you may run `python tools/misc/print_config.py /PATH/TO/CONFIG` to see the complete config.

You can learn about the usage of our config system according to the following tutorials.

- *Modify config*
- *Config file structure*
- *Config name style*
- *An example of EDSR*
- *An example of StyleGAN2*
- *Other examples*
  - *An example of config system for inpainting*
  - *An example of config system for matting*
  - *An example of config system for restoration*

1.3.1 Modify config through script arguments

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

- Update config keys of dict chains.

  The config options can be specified following the order of the dict keys in the original config. For example, `--cfg-options test_cfg.use_ema=False` changes the default sampling model to the original generator, and `--cfg-options train_dataloader.batch_size=8` changes the batch size of train dataloader.

- Update keys inside a list of configs.

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

- Update values of list/tuples.

  If the value to be updated is a list or a tuple. You can set `--cfg-options key="[a,b]"` or `--cfg-options key=a,b`. It also allows nested list/tuple values, e.g., `--cfg-options key="[(a,b),(c,d)]"`. Note that the quotation mark " is necessary to support list/tuple data types, and that **NO** white space is allowed inside the quotation marks in the specified value.

1.3.2 Config file structure

There are 3 basic component types under `config/_base_`: datasets, models and default_runtime. Many methods could be easily constructed with one of each like AOT-GAN, EDVR, GLEAN, StyleGAN2, CycleGAN, SinGAN, etc. Configs consisting of components from `_base_` are called *primitive*.

For all configs under the same folder, it is recommended to have only **one** *primitive* config. All other configs should inherit from the *primitive* config. In this way, the maximum of inheritance level is 3.

For easy understanding, we recommend contributors to inherit from existing methods. For example, if some modification is made base on BasicVSR, user may first inherit the basic BasicVSR structure by specifying `_base_ = ../basicvsr/basicvsr_reds4.py`, then modify the necessary fields in the config files. If some modification is made base on StyleGAN2, user may first inherit the basic StyleGAN2 structure by specifying `_base_ = ../styleganv2/stylegan2_c2_ffhq_256_b4x8_800k.py`, then modify the necessary fields in the config files.

If you are building an entirely new method that does not share the structure with any of the existing methods, you may create a folder `xxx` under `configs`,

Please refer to MMEngine for detailed documentation.

1.3.3 Config name style

```
{model}_[module setting]_{training schedule}_{dataset}
```

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

- `{model}`: model type like `stylegan`, `dcgan`, `basicvsr`, `dim`, etc. Settings referred in the original paper are included in this field as well (e.g., `Stylegan2-config-f`, `edvrm` of `edvrm_8xb4-600k_reds`.)
- `[module setting]`: specific setting for some modules, including Encoder, Decoder, Generator, Discriminator, Normalization, loss, Activation, etc. E.g. `c64n7` of `basicvsr-pp_c64n7_8xb1-600k_reds4`, learning rate `Glr4e-4_Dlr1e-4` for dcgan, `gamma32.8` for stylegan3, `woReLUInplace` in sagan. In this section, information from different submodules (e.g., generator and discriminator) are connected with `_`.
- `{training_scheduler}`: specific setting for training, including batch_size, schedule, etc. For example, learning rate (e.g., `lr1e-3`), number of gpu and batch size is used (e.g., `8xb32`), and total iterations (e.g., `160kiter`) or number of images shown in the discriminator (e.g., `12Mimgs`).
- `{dataset}`: dataset name and data size info like `celeba-256x256` of `deepfillv1_4xb4_celeba-256x256`, `reds4` of `basicvsr_2xb4_reds4`, `ffhq`, `lsun-car`, `celeba-hq`.

1.3.4 An example of EDSR

To help the users have a basic idea of a complete config, we make a brief comments on the config of the EDSR model we implemented as the following. For more detailed usage and the corresponding alternative for each modules, please refer to the API documentation and the tutorial in MMEngine.

Model config

In MMEditing's config, we use model fields to set up a model.

```
model = dict(
    type='BaseEditModel',  # Name of the model
    generator=dict(  # Config of the generator
        type='EDSRNet',  # Type of the generator
        in_channels=3,  # Channel number of inputs
        out_channels=3,  # Channel number of outputs
        mid_channels=64,  # Channel number of intermediate features
        num_blocks=16,  # Block number in the trunk network
        upscale_factor=scale, # Upsampling factor
        res_scale=1,  # Used to scale the residual in residual block
        rgb_mean=(0.4488, 0.4371, 0.4040),  # Image mean in RGB orders
        rgb_std=(1.0, 1.0, 1.0)),  # Image std in RGB orders
    pixel_loss=dict(type='L1Loss', loss_weight=1.0, reduction='mean')  # Config for␣
↪pixel loss
    train_cfg=dict(),  # Config of training model.
    test_cfg=dict(),  # Config of testing model.
    data_preprocessor=dict(  # The Config to build data preprocessor
        type='EditDataPreprocessor', mean=[0., 0., 0.], std=[255., 255.,
                                                             255.]))
```

Data config

Dataloaders are required for the training, validation, and testing of the runner. Dataset and data pipeline need to be set to build the dataloader. Due to the complexity of this part, we use intermediate variables to simplify the writing of dataloader configs.

Data pipeline

```
train_pipeline = [  # Training data processing pipeline
    dict(type='LoadImageFromFile',  # Load images from files
        key='img',  # Keys in results to find the corresponding path
        color_type='color',  # Color type of image
        channel_order='rgb',  # Channel order of image
        imdecode_backend='cv2'),  # decode backend
    dict(type='LoadImageFromFile',  # Load images from files
        key='gt',  # Keys in results to find the corresponding path
        color_type='color',  # Color type of image
        channel_order='rgb',  # Channel order of image
        imdecode_backend='cv2'),  # decode backend
    dict(type='SetValues', dictionary=dict(scale=scale)),  # Set value to destination␣
↪keys
```

(continues on next page)

(continued from previous page)

```
    dict(type='PairedRandomCrop', gt_patch_size=96),  # Paired random crop
    dict(type='Flip',  # Flip images
        keys=['lq', 'gt'],  # Images to be flipped
        flip_ratio=0.5,  # Flip ratio
        direction='horizontal'),  # Flip direction
    dict(type='Flip',  # Flip images
        keys=['lq', 'gt'],  # Images to be flipped
        flip_ratio=0.5,  # Flip ratio
        direction='vertical'),  # Flip direction
    dict(type='RandomTransposeHW',  # Random transpose h and w for images
        keys=['lq', 'gt'],  # Images to be transposed
        transpose_ratio=0.5  # Transpose ratio
        ),
    dict(type='PackEditInputs')  # The config of collecting data from the current␣
↪pipeline
]
test_pipeline = [  # Test pipeline
    dict(type='LoadImageFromFile',  # Load images from files
        key='img',  # Keys in results to find corresponding path
        color_type='color',  # Color type of image
        channel_order='rgb',  # Channel order of image
        imdecode_backend='cv2'),  # decode backend
    dict(type='LoadImageFromFile',  # Load images from files
        key='gt',  # Keys in results to find corresponding path
        color_type='color',  # Color type of image
        channel_order='rgb',  # Channel order of image
        imdecode_backend='cv2'),  # decode backend
    dict(type='PackEditInputs')  # The config of collecting data from the current␣
↪pipeline
]
```

Dataloader

```
dataset_type = 'BasicImageDataset'  # The type of dataset
data_root = 'data'  # Root path of data
train_dataloader = dict(
    num_workers=4,  # The number of workers to pre-fetch data for each single GPU
    persistent_workers=False,  # Whether maintain the workers Dataset instances alive
    sampler=dict(type='InfiniteSampler', shuffle=True),  # The type of data sampler
    dataset=dict(  # Train dataset config
        type=dataset_type,  # Type of dataset
        ann_file='meta_info_DIV2K800sub_GT.txt',  # Path of annotation file
        metainfo=dict(dataset_type='div2k', task_name='sisr'),
        data_root=data_root + '/DIV2K',  # Root path of data
        data_prefix=dict(  # Prefix of image path
            img='DIV2K_train_LR_bicubic/X2_sub', gt='DIV2K_train_HR_sub'),
        filename_tmpl=dict(img='{}', gt='{}'),  # Filename template
        pipeline=train_pipeline))
val_dataloader = dict(
    num_workers=4,  # The number of workers to pre-fetch data for each single GPU
```

(continues on next page)

(continued from previous page)

```
    persistent_workers=False,  # Whether maintain the workers Dataset instances alive
    drop_last=False,  # Whether drop the last incomplete batch
    sampler=dict(type='DefaultSampler', shuffle=False),  # The type of data sampler
    dataset=dict(  # Validation dataset config
        type=dataset_type,  # Type of dataset
        metainfo=dict(dataset_type='set5', task_name='sisr'),
        data_root=data_root + '/Set5',  # Root path of data
        data_prefix=dict(img='LRbicx2', gt='GTmod12'),  # Prefix of image path
        pipeline=test_pipeline))
test_dataloader = val_dataloader
```

Evaluation config

Evaluators are used to compute the metrics of the trained model on the validation and testing datasets. The config of evaluators consists of one or a list of metric configs:

```
val_evaluator = [
    dict(type='MAE'),  # The name of metrics to evaluate
    dict(type='PSNR', crop_border=scale),  # The name of metrics to evaluate
    dict(type='SSIM', crop_border=scale),  # The name of metrics to evaluate
]
test_evaluator = val_evaluator # The config for testing evaluator
```

Training and testing config

MMEngine's runner uses Loop to control the training, validation, and testing processes. Users can set the maximum training iteration and validation intervals with these fields.

```
train_cfg = dict(
    type='IterBasedTrainLoop',  # The name of train loop type
    max_iters=300000,  # The number of total iterations
    val_interval=5000,  # The number of validation interval iterations
)
val_cfg = dict(type='ValLoop')  # The name of validation loop type
test_cfg = dict(type='TestLoop')  # The name of test loop type
```

Optimization config

`optim_wrapper` is the field to configure optimization related settings. The optimizer wrapper not only provides the functions of the optimizer, but also supports functions such as gradient clipping, mixed precision training, etc. Find more in optimizer wrapper tutorial.

```
optim_wrapper = dict(
    dict(
        type='OptimWrapper',
        optimizer=dict(type='Adam', lr=0.00001),
    )
)  # Config used to build optimizer, support all the optimizers in PyTorch whose␣
↪arguments are also the same as those in PyTorch.
```

`param_scheduler` is a field that configures methods of adjusting optimization hyper-parameters such as learning rate and momentum. Users can combine multiple schedulers to create a desired parameter adjustment strategy. Find more in parameter scheduler tutorial.

```
param_scheduler = dict(  # Config of learning policy
    type='MultiStepLR', by_epoch=False, milestones=[200000], gamma=0.5)
```

Hook config

Users can attach hooks to training, validation, and testing loops to insert some operations during running. There are two different hook fields, one is `default_hooks` and the other is `custom_hooks`.

`default_hooks` is a dict of hook configs. `default_hooks` are the hooks must required at runtime. They have default priority which should not be modified. If not set, runner will use the default values. To disable a default hook, users can set its config to `None`.

```
default_hooks = dict(  # Used to build default hooks
    checkpoint=dict(  # Config to set the checkpoint hook
        type='CheckpointHook',
        interval=5000,  # The save interval is 5000 iterations
        save_optimizer=True,
        by_epoch=False,  # Count by iterations
        out_dir=save_dir,
    ),
    timer=dict(type='IterTimerHook'),
    logger=dict(type='LoggerHook', interval=100),  # Config to register logger hook
    param_scheduler=dict(type='ParamSchedulerHook'),
    sampler_seed=dict(type='DistSamplerSeedHook'),
)
```

`custom_hooks` is a list of hook configs. Users can develop there own hooks and insert them in this field.

```
custom_hooks = [dict(type='BasicVisualizationHook', interval=1)] # Config of␣
↪visualization hook
```

Runtime config

```
default_scope = 'mmedit' # Used to set registries location
env_cfg = dict(  # Parameters to setup distributed training, the port can also be set
    cudnn_benchmark=False,
    mp_cfg=dict(mp_start_method='fork', opencv_num_threads=4),
    dist_cfg=dict(backend='nccl'),
)
log_level = 'INFO'  # The level of logging
log_processor = dict(type='LogProcessor', window_size=100, by_epoch=False)  # Used to␣
↪build log processor
load_from = None  # load models as a pre-trained model from a given path. This will not␣
↪resume training.
resume = False  # Resume checkpoints from a given path, the training will be resumed␣
↪from the epoch when the checkpoint's is saved.
```

1.3.5 An example of StyleGAN2

Taking Stylegan2 at 1024x1024 scale as an example, we introduce each field in the config according to different function modules.

Model config

In addition to neural network components such as generator, discriminator etc, it also requires `data_preprocessor`, `loss_config`, and some of them contains `ema_config`. `data_preprocessor` is responsible for processing a batch of data output by dataloader. `loss_config` is responsible for weight of loss terms. `ema_config` is responsible for exponential moving average (EMA) operation for generator.

```
model = dict(
    type='StyleGAN2',  # The name of the model
    data_preprocessor=dict(type='GANDataPreprocessor'),  # The config of data␣
↪preprocessor, usually includs image normalization and padding
    generator=dict(  # The config for generator
        type='StyleGANv2Generator',  # The name of the generator
        out_size=1024,  # The output resolution of the generator
        style_channels=512),  # The number of style channels of the generator
    discriminator=dict(  # The config for discriminator
        type='StyleGAN2Discriminator',  # The name of the discriminator
        in_size=1024),  # The input resolution of the discriminator
    ema_config=dict(  # The config for EMA
        type='ExponentialMovingAverage',  # Specific the type of Average model
        interval=1,  # The interval of EMA operation
        momentum=0.9977843871238888),  # The momentum of EMA operation
    loss_config=dict(  # The config for loss terms
        r1_loss_weight=80.0,  # The weight for r1 gradient penalty
        r1_interval=16,  # The interval of r1 gradient penalty
        norm_mode='HWC',  # The normalization mode for r1 gradient penalty
        g_reg_interval=4,  # The interval for generator's regularization
        g_reg_weight=8.0,  # The weight for generator's regularization
        pl_batch_shrink=2))  # The factor of shrinking the batch size in path length␣
↪regularization
```

Dataset and evaluator config

Dataloaders are required for the training, validation, and testing of the runner. Dataset and data pipeline need to be set to build the dataloader. Due to the complexity of this part, we use intermediate variables to simplify the writing of dataloader configs.

```
dataset_type = 'BasicImageDataset'  # Dataset type, this will be used to define the␣
↪dataset
data_root = './data/ffhq/'  # Root path of data

train_pipeline = [  # Training data process pipeline
    dict(type='LoadImageFromFile', key='img'),  # First pipeline to load images from␣
↪file path
    dict(type='Flip', keys=['img'], direction='horizontal'),  # Argumentation pipeline␣
↪that flip the images
    dict(type='PackEditInputs', keys=['img'])  # The last pipeline that formats the␣
↪annotation data (if have) and decides which keys in the data should be packed into␣
↪data_samples
```

(continues on next page)

(continued from previous page)

```
]
val_pipeline = [
    dict(type='LoadImageFromFile', key='img'),  # First pipeline to load images from␣
↪file path
    dict(type='PackEditInputs', keys=['img'])  # The last pipeline that formats the␣
↪annotation data (if have) and decides which keys in the data should be packed into␣
↪data_samples
]
train_dataloader = dict(  # The config of train dataloader
    batch_size=4,  # Batch size of a single GPU
    num_workers=8,  # Worker to pre-fetch data for each single GPU
    persistent_workers=True,  # If ``True``, the dataloader will not shutdown the worker␣
↪processes after an epoch end, which can accelerate training speed.
    sampler=dict(  # The config of training data sampler
        type='InfiniteSampler',  # InfiniteSampler for iteratiion-based training. Refers␣
↪to https://github.com/open-mmlab/mmengine/blob/
↪fe0eb0a5bbc8bf816d5649bfdd34908c258eb245/mmengine/dataset/sampler.py#L107
        shuffle=True),  # Whether randomly shuffle the training data
    dataset=dict(  # The config of the training dataset
        type=dataset_type,
        data_root=data_root,
        pipeline=train_pipeline))
val_dataloader = dict(  # The config of validation dataloader
    batch_size=4,  # Batch size of a single GPU
    num_workers=8,  # Worker to pre-fetch data for each single GPU
    dataset=dict(  # The config of the validation dataset
        type=dataset_type,
        data_root=data_root,
        pipeline=val_pipeline),
    sampler=dict(  # The config of validatioin data sampler
        type='DefaultSampler',  # DefaultSampler which supports both distributed and non-
↪distributed training. Refer to https://github.com/open-mmlab/mmengine/blob/
↪fe0eb0a5bbc8bf816d5649bfdd34908c258eb245/mmengine/dataset/sampler.py#L14
        shuffle=False),  # Whether randomly shuffle the validation data
    persistent_workers=True)
test_dataloader = val_dataloader  # The config of the testing dataloader
```

Evaluators are used to compute the metrics of the trained model on the validation and testing datasets. The config of evaluators consists of one or a list of metric configs:

```
val_evaluator = dict(  # The config for validation evaluator
    type='GenEvaluator',  # The type of evaluation
    metrics=[  # The config for metrics
        dict(
            type='FrechetInceptionDistance',
            prefix='FID-Full-50k',
            fake_nums=50000,
            inception_style='StyleGAN',
            sample_model='ema'),
        dict(type='PrecisionAndRecall', fake_nums=50000, prefix='PR-50K'),
        dict(type='PerceptualPathLength', fake_nums=50000, prefix='ppl-w')
    ])
```

(continues on next page)

(continued from previous page)

```
test_evaluator = val_evaluator  # The config for testing evaluator
```

Training and testing config

MMEngine's runner uses Loop to control the training, validation, and testing processes. Users can set the maximum training iteration and validation intervals with these fields.

```
train_cfg = dict(  # The config for training
    by_epoch=False,  # Set `by_epoch` as False to use iteration-based training
    val_begin=1,  # Which iteration to start the validation
    val_interval=10000,  # Validation intervals
    max_iters=800002)  # Maximum training iterations
val_cfg = dict(type='GenValLoop')  # The validation loop type
test_cfg = dict(type='GenTestLoop')  # The testing loop type
```

Optimization config

`optim_wrapper` is the field to configure optimization related settings. The optimizer wrapper not only provides the functions of the optimizer, but also supports functions such as gradient clipping, mixed precision training, etc. Find more in optimizer wrapper tutorial.

```
optim_wrapper = dict(
    constructor='GenOptimWrapperConstructor',
    generator=dict(
        optimizer=dict(type='Adam', lr=0.0016, betas=(0, 0.9919919678228657))),
    discriminator=dict(
        optimizer=dict(
            type='Adam',
            lr=0.0018823529411764706,
            betas=(0, 0.9905854573074332))))
```

`param_scheduler` is a field that configures methods of adjusting optimization hyperparameters such as learning rate and momentum. Users can combine multiple schedulers to create a desired parameter adjustment strategy. Find more in parameter scheduler tutorial. Since StyleGAN2 do not use parameter scheduler, we use config in CycleGAN as an example:

```
# parameter scheduler in CycleGAN config
param_scheduler = dict(
    type='LinearLrInterval',  # The type of scheduler
    interval=400,  # The interval to update the learning rate
    by_epoch=False,  # The scheduler is called by iteration
    start_factor=0.0002,  # The number we multiply parameter value in the first iteration
    end_factor=0,  # The number we multiply parameter value at the end of linear␣
↪changing process.
    begin=40000,  # The start iteration of the scheduler
    end=80000)  # The end iteration of the scheduler
```

Hook config

Users can attach hooks to training, validation, and testing loops to insert some operations during running. There are two different hook fields, one is `default_hooks` and the other is `custom_hooks`.

`default_hooks` is a dict of hook configs. `default_hooks` are the hooks must required at runtime. They have default priority which should not be modified. If not set, runner will use the default values. To disable a default hook, users can set its config to `None`.

```
default_hooks = dict(
    timer=dict(type='GenIterTimerHook'),
    logger=dict(type='LoggerHook', interval=100, log_metric_by_epoch=False),
    checkpoint=dict(
        type='CheckpointHook',
        interval=10000,
        by_epoch=False,
        less_keys=['FID-Full-50k/fid'],
        greater_keys=['IS-50k/is'],
        save_optimizer=True,
        save_best='FID-Full-50k/fid'))
```

`custom_hooks` is a list of hook configs. Users can develop there own hooks and insert them in this field.

```
custom_hooks = [
    dict(
        type='GenVisualizationHook',
        interval=5000,
        fixed_input=True,
        vis_kwargs_list=dict(type='GAN', name='fake_img'))
]
```

Runtime config

```
default_scope = 'mmedit'  # The default registry scope to find modules. Refer to https://
↪mmengine.readthedocs.io/en/latest/tutorials/registry.html

# config for environment
env_cfg = dict(
    cudnn_benchmark=True,  # whether to enable cudnn benchmark.
    mp_cfg=dict(mp_start_method='fork', opencv_num_threads=0),  # set multi process␣
↪parameters.
    dist_cfg=dict(backend='nccl'),  # set distributed parameters.
)

log_level = 'INFO'  # The level of logging
log_processor = dict(
    type='GenLogProcessor',  # log processor to process runtime logs
    by_epoch=False)  # print log by iteration
load_from = None  # load model checkpoint as a pre-trained model for a given path
resume = False  # Whether to resume from the checkpoint define in `load_from`. If `load_
↪from` is `None`, it will resume the latest checkpoint in `work_dir`
```

1.3.6 Other examples

An example of config system for inpainting

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

```
model = dict(
    type='GLInpaintor', # The name of inpaintor
    data_preprocessor=dict(
        type='EditDataPreprocessor', # The name of data preprocessor
        mean=[127.5], # Mean value used in data normalization
        std=[127.5], # Std value used in data normalization
    ),
    encdec=dict(
        type='GLEncoderDecoder', # The name of encoder-decoder
        encoder=dict(type='GLEncoder', norm_cfg=dict(type='SyncBN')), # The config of␣
↪encoder
        decoder=dict(type='GLDecoder', norm_cfg=dict(type='SyncBN')), # The config of␣
↪decoder
        dilation_neck=dict(
            type='GLDilationNeck', norm_cfg=dict(type='SyncBN'))), # The config of␣
↪dilation neck
    disc=dict(
        type='GLDiscs', # The name of discriminator
        global_disc_cfg=dict(
            in_channels=3, # The input channel of discriminator
            max_channels=512, # The maximum middle channel in discriminator
            fc_in_channels=512 * 4 * 4, # The input channel of last fc layer
            fc_out_channels=1024, # The output channel of last fc channel
            num_convs=6, # The number of convs used in discriminator
            norm_cfg=dict(type='SyncBN') # The config of norm layer
        ),
        local_disc_cfg=dict(
            in_channels=3, # The input channel of discriminator
            max_channels=512, # The maximum middle channel in discriminator
            fc_in_channels=512 * 4 * 4, # The input channel of last fc layer
            fc_out_channels=1024, # The output channel of last fc channel
            num_convs=5, # The number of convs used in discriminator
            norm_cfg=dict(type='SyncBN') # The config of norm layer
        ),
    ),
    loss_gan=dict(
        type='GANLoss', # The name of GAN loss
        gan_type='vanilla', # The type of GAN loss
        loss_weight=0.001 # The weight of GAN loss
    ),
    loss_l1_hole=dict(
        type='L1Loss', # The type of l1 loss
        loss_weight=1.0 # The weight of l1 loss
    ))
```

(continues on next page)

(continued from previous page)

```
train_cfg = dict(
    type='IterBasedTrainLoop',# The name of train loop type
    max_iters=500002, # The number of total iterations
    val_interval=50000, # The number of validation interval iterations
)
val_cfg = dict(type='ValLoop') # The name of validation loop type
test_cfg = dict(type='TestLoop') # The name of test loop type

val_evaluator = [
    dict(type='MAE', mask_key='mask', scaling=100), # The name of metrics to evaluate
    dict(type='PSNR'), # The name of metrics to evaluate
    dict(type='SSIM'), # The name of metrics to evaluate
]
test_evaluator = val_evaluator

input_shape = (256, 256) # The shape of input image

train_pipeline = [
    dict(type='LoadImageFromFile', key='gt'), # The config of loading image
    dict(
        type='LoadMask', # The type of loading mask pipeline
        mask_mode='bbox', # The type of mask
        mask_config=dict(
            max_bbox_shape=(128, 128), # The shape of bbox
            max_bbox_delta=40, # The changing delta of bbox height and width
            min_margin=20,  # The minimum margin from bbox to the image border
            img_shape=input_shape)),  # The input image shape
    dict(
        type='Crop', # The type of crop pipeline
        keys=['gt'],  # The keys of images to be cropped
        crop_size=(384, 384),  # The size of cropped patch
        random_crop=True,  # Whether to use random crop
    ),
    dict(
        type='Resize',  # The type of resizing pipeline
        keys=['gt'],  # They keys of images to be resized
        scale=input_shape,  # The scale of resizing function
        keep_ratio=False,  # Whether to keep ratio during resizing
    ),
    dict(
        type='Normalize',  # The type of normalizing pipeline
        keys=['gt_img'],  # The keys of images to be normed
        mean=[127.5] * 3,  # Mean value used in normalization
        std=[127.5] * 3,  # Std value used in normalization
        to_rgb=False),  # Whether to transfer image channels to rgb
    dict(type='GetMaskedImage'), # The config of getting masked image pipeline
    dict(type='PackEditInputs'), # The config of collecting data from the current␣
↪pipeline
]

test_pipeline = train_pipeline  # Constructing testing/validation pipeline
```

(continues on next page)

(continued from previous page)

```
dataset_type = 'BasicImageDataset' # The type of dataset
data_root = 'data/places'  # Root path of data

train_dataloader = dict(
    batch_size=12, # Batch size of a single GPU
    num_workers=4, # The number of workers to pre-fetch data for each single GPU
    persistent_workers=False, # Whether maintain the workers Dataset instances alive
    sampler=dict(type='InfiniteSampler', shuffle=False), # The type of data sampler
    dataset=dict(  # Train dataset config
        type=dataset_type, # Type of dataset
        data_root=data_root, # Root path of data
        data_prefix=dict(gt='data_large'), # Prefix of image path
        ann_file='meta/places365_train_challenge.txt', # Path of annotation file
        test_mode=False,
        pipeline=train_pipeline,
    ))

val_dataloader = dict(
    batch_size=1, # Batch size of a single GPU
    num_workers=4, # The number of workers to pre-fetch data for each single GPU
    persistent_workers=False, # Whether maintain the workers Dataset instances alive
    drop_last=False, # Whether drop the last incomplete batch
    sampler=dict(type='DefaultSampler', shuffle=False), # The type of data sampler
    dataset=dict( # Validation dataset config
        type=dataset_type, # Type of dataset
        data_root=data_root, # Root path of data
        data_prefix=dict(gt='val_large'), # Prefix of image path
        ann_file='meta/places365_val.txt', # Path of annotation file
        test_mode=True,
        pipeline=test_pipeline,
    ))

test_dataloader = val_dataloader

model_wrapper_cfg = dict(type='MMSeparateDistributedDataParallel') # The name of model␣
↪wrapper

optim_wrapper = dict( # Config used to build optimizer, support all the optimizers in␣
↪PyTorch whose arguments are also the same as those in PyTorch
    constructor='MultiOptimWrapperConstructor',
    generator=dict(
        type='OptimWrapper', optimizer=dict(type='Adam', lr=0.0004)),
    disc=dict(type='OptimWrapper', optimizer=dict(type='Adam', lr=0.0004)))

default_scope = 'mmedit' # Used to set registries location
save_dir = './work_dirs' # Directory to save the model checkpoints and logs for the␣
↪current experiments
exp_name = 'gl_places'  # The experiment name

default_hooks = dict( # Used to build default hooks
    timer=dict(type='IterTimerHook'),
    logger=dict(type='LoggerHook', interval=100), # Config to register logger hook
```

(continues on next page)

(continued from previous page)

```
    param_scheduler=dict(type='ParamSchedulerHook'),
    checkpoint=dict( # Config to set the checkpoint hook
        type='CheckpointHook',
        interval=50000,
        by_epoch=False,
        out_dir=save_dir),
    sampler_seed=dict(type='DistSamplerSeedHook'),
)

env_cfg = dict( # Parameters to setup distributed training, the port can also be set
    cudnn_benchmark=False,
    mp_cfg=dict(mp_start_method='fork', opencv_num_threads=0),
    dist_cfg=dict(backend='nccl'),
)

vis_backends = [dict(type='LocalVisBackend')] # The name of visualization backend
visualizer = dict( # Config used to build visualizer
    type='ConcatImageVisualizer',
    vis_backends=vis_backends,
    fn_key='gt_path',
    img_keys=['gt_img', 'input', 'pred_img'],
    bgr2rgb=True)
custom_hooks = [dict(type='BasicVisualizationHook', interval=1)] # Used to build custom␣
↪hooks

log_level = 'INFO' # The level of logging
log_processor = dict(type='LogProcessor', by_epoch=False) # Used to build log processor

load_from = None # load models as a pre-trained model from a given path. This will not␣
↪resume training.
resume = False # Resume checkpoints from a given path, the training will be resumed from␣
↪the epoch when the checkpoint's is saved.

find_unused_parameters = False  # Whether to set find unused parameters in ddp
```

An example of config system for matting

To help the users have a basic idea of a complete config, we make a brief comments on the config of the original DIM model we implemented as the following. For more detailed usage and the corresponding alternative for each modules, please refer to the API documentation.

```
# model settings
model = dict(
    type='DIM',  # The name of model (we call mattor).
    data_preprocessor=dict(  # The Config to build data preprocessor
        type='MattorPreprocessor',
        mean=[123.675, 116.28, 103.53],
        std=[58.395, 57.12, 57.375],
        bgr_to_rgb=True,
        proc_inputs='normalize',
        proc_trimap='rescale_to_zero_one',
```

(continues on next page)

(continued from previous page)

```
        proc_gt='rescale_to_zero_one',
    ),
    backbone=dict(  # The config of the backbone.
        type='SimpleEncoderDecoder',  # The type of the backbone.
        encoder=dict(  # The config of the encoder.
            type='VGG16'),  # The type of the encoder.
        decoder=dict(  # The config of the decoder.
            type='PlainDecoder')),  # The type of the decoder.
    pretrained='./weights/vgg_state_dict.pth',  # The pretrained weight of the encoder␣
↪to be loaded.
    loss_alpha=dict(  # The config of the alpha loss.
        type='CharbonnierLoss',  # The type of the loss for predicted alpha matte.
        loss_weight=0.5),  # The weight of the alpha loss.
    loss_comp=dict(  # The config of the composition loss.
        type='CharbonnierCompLoss',  # The type of the composition loss.
        loss_weight=0.5), # The weight of the composition loss.
    train_cfg=dict(  # Config of training DIM model.
        train_backbone=True,  # In DIM stage1, backbone is trained.
        train_refiner=False),  # In DIM stage1, refiner is not trained.
    test_cfg=dict(  # Config of testing DIM model.
        refine=False,  # Whether use refiner output as output, in stage1, we don't use it.
        resize_method='pad',
        resize_mode='reflect',
        size_divisor=32,
    ),
)

# data settings
dataset_type = 'AdobeComp1kDataset'  # Dataset type, this will be used to define the␣
↪dataset.
data_root = 'data/adobe_composition-1k'  # Root path of data.

train_pipeline = [  # Training data processing pipeline.
    dict(
        type='LoadImageFromFile',  # Load alpha matte from file.
        key='alpha',  # Key of alpha matte in annotation file. The pipeline will read␣
↪alpha matte from path `alpha_path`.
        color_type='grayscale'),  # Load as grayscale image which has shape (height,␣
↪width).
    dict(
        type='LoadImageFromFile',  # Load image from file.
        key='fg'),  # Key of image to load. The pipeline will read fg from path `fg_path`.
    dict(
        type='LoadImageFromFile',  # Load image from file.
        key='bg'),  # Key of image to load. The pipeline will read bg from path `bg_path`.
    dict(
        type='LoadImageFromFile',  # Load image from file.
        key='merged'),  # Key of image to load. The pipeline will read merged from path␣
↪`merged_path`.
    dict(
        type='CropAroundUnknown',  # Crop images around unknown area (semi-transparent␣
↪area).
```

(continues on next page)

(continued from previous page)

```
        keys=['alpha', 'merged', 'fg', 'bg'],  # Images to crop.
        crop_sizes=[320, 480, 640]),  # Candidate crop size.
    dict(
        type='Flip',  # Augmentation pipeline that flips the images.
        keys=['alpha', 'merged', 'fg', 'bg']),  # Images to be flipped.
    dict(
        type='Resize',  # Augmentation pipeline that resizes the images.
        keys=['alpha', 'merged', 'fg', 'bg'],  # Images to be resized.
        scale=(320, 320),  # Target size.
        keep_ratio=False),  # Whether to keep the ratio between height and width.
    dict(
        type='GenerateTrimap',  # Generate trimap from alpha matte.
        kernel_size=(1, 30)),  # Kernel size range of the erode/dilate kernel.
    dict(type='PackEditInputs'),  # The config of collecting data from the current␣
↪pipeline
]
test_pipeline = [
    dict(
        type='LoadImageFromFile',  # Load alpha matte.
        key='alpha',  # Key of alpha matte in annotation file. The pipeline will read␣
↪alpha matte from path `alpha_path`.
        color_type='grayscale',
        save_original_img=True),
    dict(
        type='LoadImageFromFile',  # Load image from file
        key='trimap',  # Key of image to load. The pipeline will read trimap from path␣
↪`trimap_path`.
        color_type='grayscale',  # Load as grayscale image which has shape (height,␣
↪width).
        save_original_img=True),  # Save a copy of trimap for calculating metrics. It␣
↪will be saved with key `ori_trimap`
    dict(
        type='LoadImageFromFile',  # Load image from file
        key='merged'),  # Key of image to load. The pipeline will read merged from path␣
↪`merged_path`.
    dict(type='PackEditInputs'),  # The config of collecting data from the current␣
↪pipeline
]

train_dataloader = dict(
    batch_size=1,  # Batch size of a single GPU
    num_workers=4,  # The number of workers to pre-fetch data for each single GPU
    persistent_workers=False,  # Whether maintain the workers Dataset instances alive
    sampler=dict(type='InfiniteSampler', shuffle=True),  # The type of data sampler
    dataset=dict(  # Train dataset config
        type=dataset_type,  # Type of dataset
        data_root=data_root,  # Root path of data
        ann_file='training_list.json',  # Path of annotation file
        test_mode=False,
        pipeline=train_pipeline,
    ))
```

(continues on next page)

(continued from previous page)

```
val_dataloader = dict(
    batch_size=1,  # Batch size of a single GPU
    num_workers=4,  # The number of workers to pre-fetch data for each single GPU
    persistent_workers=False,  # Whether maintain the workers Dataset instances alive
    drop_last=False,  # Whether drop the last incomplete batch
    sampler=dict(type='DefaultSampler', shuffle=False),  # The type of data sampler
    dataset=dict(  # Validation dataset config
        type=dataset_type,  # Type of dataset
        data_root=data_root,  # Root path of data
        ann_file='test_list.json',  # Path of annotation file
        test_mode=True,
        pipeline=test_pipeline,
    ))

test_dataloader = val_dataloader

val_evaluator = [
    dict(type='SAD'),  # The name of metrics to evaluate
    dict(type='MattingMSE'),  # The name of metrics to evaluate
    dict(type='GradientError'),  # The name of metrics to evaluate
    dict(type='ConnectivityError'),  # The name of metrics to evaluate
]
test_evaluator = val_evaluator

train_cfg = dict(
    type='IterBasedTrainLoop',  # The name of train loop type
    max_iters=1_000_000,  # The number of total iterations
    val_interval=40000,  # The number of validation interval iterations
)
val_cfg = dict(type='ValLoop')  # The name of validation loop type
test_cfg = dict(type='TestLoop')  # The name of test loop type

# optimizer
optim_wrapper = dict(
    dict(
        type='OptimWrapper',
        optimizer=dict(type='Adam', lr=0.00001),
    )
)  # Config used to build optimizer, support all the optimizers in PyTorch whose␣
↪arguments are also the same as those in PyTorch.

default_scope = 'mmedit'  # Used to set registries location
save_dir = './work_dirs'  # Directory to save the model checkpoints and logs for the␣
↪current experiments.

default_hooks = dict(  # Used to build default hooks
    timer=dict(type='IterTimerHook'),
    logger=dict(type='LoggerHook', interval=100),  # Config to register logger hook
    param_scheduler=dict(type='ParamSchedulerHook'),
    checkpoint=dict(  # Config to set the checkpoint hook
        type='CheckpointHook',
        interval=40000,  # The save interval is 40000 iterations.
```

(continues on next page)

(continued from previous page)

```
        by_epoch=False,  # Count by iterations.
        out_dir=save_dir),
    sampler_seed=dict(type='DistSamplerSeedHook'),
)

env_cfg = dict(  # Parameters to setup distributed training, the port can also be set
    cudnn_benchmark=False,
    mp_cfg=dict(mp_start_method='fork', opencv_num_threads=4),
    dist_cfg=dict(backend='nccl'),
)

log_level = 'INFO'  # The level of logging
log_processor = dict(type='LogProcessor', by_epoch=False)  # Used to build log processor

load_from = None  # load models as a pre-trained model from a given path. This will not␣
↪resume training.
resume = False  # Resume checkpoints from a given path, the training will be resumed␣
↪from the epoch when the checkpoint's is saved.
```

An example of config system for restoration

To help the users have a basic idea of a complete config, we make a brief comments on the config of the EDSR model we implemented as the following. For more detailed usage and the corresponding alternative for each modules, please refer to the API documentation.

```
exp_name = 'edsr_x2c64b16_1x16_300k_div2k'  # The experiment name
work_dir = f'./work_dirs/{experiment_name}'
save_dir = './work_dirs/'

load_from = None  # based on pre-trained x2 model

scale = 2  # Scale factor for upsampling
# model settings
model = dict(
    type='BaseEditModel',  # Name of the model
    generator=dict(  # Config of the generator
        type='EDSRNet',  # Type of the generator
        in_channels=3,  # Channel number of inputs
        out_channels=3,  # Channel number of outputs
        mid_channels=64,  # Channel number of intermediate features
        num_blocks=16,  # Block number in the trunk network
        upscale_factor=scale, # Upsampling factor
        res_scale=1,  # Used to scale the residual in residual block
        rgb_mean=(0.4488, 0.4371, 0.4040),  # Image mean in RGB orders
        rgb_std=(1.0, 1.0, 1.0)),  # Image std in RGB orders
    pixel_loss=dict(type='L1Loss', loss_weight=1.0, reduction='mean')  # Config for␣
↪pixel loss
    train_cfg=dict(),  # Config of training model.
    test_cfg=dict(),  # Config of testing model.
    data_preprocessor=dict(  # The Config to build data preprocessor
        type='EditDataPreprocessor', mean=[0., 0., 0.], std=[255., 255.,
```

(continues on next page)

(continued from previous page)

```
                                                    255.]))

train_pipeline = [  # Training data processing pipeline
    dict(type='LoadImageFromFile',  # Load images from files
        key='img',  # Keys in results to find corresponding path
        color_type='color',  # Color type of image
        channel_order='rgb',  # Channel order of image
        imdecode_backend='cv2'),  # decode backend
    dict(type='LoadImageFromFile',  # Load images from files
        key='gt',  # Keys in results to find corresponding path
        color_type='color',  # Color type of image
        channel_order='rgb',  # Channel order of image
        imdecode_backend='cv2'),  # decode backend
    dict(type='SetValues', dictionary=dict(scale=scale)),  # Set value to destination␣
↪keys
    dict(type='PairedRandomCrop', gt_patch_size=96),  # Paired random crop
    dict(type='Flip',  # Flip images
        keys=['lq', 'gt'],  # Images to be flipped
        flip_ratio=0.5,  # Flip ratio
        direction='horizontal'),  # Flip direction
    dict(type='Flip',  # Flip images
        keys=['lq', 'gt'],  # Images to be flipped
        flip_ratio=0.5,  # Flip ratio
        direction='vertical'),  # Flip direction
    dict(type='RandomTransposeHW',  # Random transpose h and w for images
        keys=['lq', 'gt'],  # Images to be transposed
        transpose_ratio=0.5  # Transpose ratio
        ),
    dict(type='ToTensor', keys=['img', 'gt']),  # Convert images to tensor
    dict(type='PackEditInputs')  # The config of collecting data from the current␣
↪pipeline
]
test_pipeline = [  # Test pipeline
    dict(type='LoadImageFromFile',  # Load images from files
        key='img',  # Keys in results to find corresponding path
        color_type='color',  # Color type of image
        channel_order='rgb',  # Channel order of image
        imdecode_backend='cv2'),  # decode backend
    dict(type='LoadImageFromFile',  # Load images from files
        key='gt',  # Keys in results to find corresponding path
        color_type='color',  # Color type of image
        channel_order='rgb',  # Channel order of image
        imdecode_backend='cv2'),  # decode backend
    dict(type='ToTensor', keys=['img', 'gt']),  # Convert images to tensor
    dict(type='PackEditInputs')  # The config of collecting data from the current␣
↪pipeline
]

# dataset settings
dataset_type = 'BasicImageDataset'  # The type of dataset
data_root = 'data'  # Root path of data
```

(continues on next page)

(continued from previous page)

```
train_dataloader = dict(
    num_workers=4,  # The number of workers to pre-fetch data for each single GPU
    persistent_workers=False,  # Whether maintain the workers Dataset instances alive
    sampler=dict(type='InfiniteSampler', shuffle=True),  # The type of data sampler
    dataset=dict(  # Train dataset config
        type=dataset_type,  # Type of dataset
        ann_file='meta_info_DIV2K800sub_GT.txt',  # Path of annotation file
        metainfo=dict(dataset_type='div2k', task_name='sisr'),
        data_root=data_root + '/DIV2K',  # Root path of data
        data_prefix=dict(  # Prefix of image path
            img='DIV2K_train_LR_bicubic/X2_sub', gt='DIV2K_train_HR_sub'),
        filename_tmpl=dict(img='{}', gt='{}'),  # Filename template
        pipeline=train_pipeline))
val_dataloader = dict(
    num_workers=4,  # The number of workers to pre-fetch data for each single GPU
    persistent_workers=False,  # Whether maintain the workers Dataset instances alive
    drop_last=False,  # Whether drop the last incomplete batch
    sampler=dict(type='DefaultSampler', shuffle=False),  # The type of data sampler
    dataset=dict(  # Validation dataset config
        type=dataset_type,  # Type of dataset
        metainfo=dict(dataset_type='set5', task_name='sisr'),
        data_root=data_root + '/Set5',  # Root path of data
        data_prefix=dict(img='LRbicx2', gt='GTmod12'),  # Prefix of image path
        pipeline=test_pipeline))
test_dataloader = val_dataloader

val_evaluator = [
    dict(type='MAE'),  # The name of metrics to evaluate
    dict(type='PSNR', crop_border=scale),  # The name of metrics to evaluate
    dict(type='SSIM', crop_border=scale),  # The name of metrics to evaluate
]
test_evaluator = val_evaluator

train_cfg = dict(
    type='IterBasedTrainLoop', max_iters=300000, val_interval=5000)  # Config of train␣
↪loop type
val_cfg = dict(type='ValLoop')  # The name of validation loop type
test_cfg = dict(type='TestLoop')  # The name of test loop type

# optimizer
optim_wrapper = dict(
    dict(
        type='OptimWrapper',
        optimizer=dict(type='Adam', lr=0.00001),
    )
)  # Config used to build optimizer, support all the optimizers in PyTorch whose␣
↪arguments are also the same as those in PyTorch.

param_scheduler = dict(  # Config of learning policy
    type='MultiStepLR', by_epoch=False, milestones=[200000], gamma=0.5)

default_hooks = dict(  # Used to build default hooks
```

(continues on next page)

(continued from previous page)

```
    checkpoint=dict(  # Config to set the checkpoint hook
        type='CheckpointHook',
        interval=5000,  # The save interval is 5000 iterations
        save_optimizer=True,
        by_epoch=False,  # Count by iterations
        out_dir=save_dir,
    ),
    timer=dict(type='IterTimerHook'),
    logger=dict(type='LoggerHook', interval=100),  # Config to register logger hook
    param_scheduler=dict(type='ParamSchedulerHook'),
    sampler_seed=dict(type='DistSamplerSeedHook'),
)

default_scope = 'mmedit'  # Used to set registries location
save_dir = './work_dirs'  # Directory to save the model checkpoints and logs for the␣
↪current experiments.

env_cfg = dict(  # Parameters to setup distributed training, the port can also be set
    cudnn_benchmark=False,
    mp_cfg=dict(mp_start_method='fork', opencv_num_threads=4),
    dist_cfg=dict(backend='nccl'),
)

log_level = 'INFO'  # The level of logging
log_processor = dict(type='LogProcessor', window_size=100, by_epoch=False)  # Used to␣
↪build log processor

load_from = None  # load models as a pre-trained model from a given path. This will not␣
↪resume training.
resume = False  # Resume checkpoints from a given path, the training will be resumed␣
↪from the epoch when the checkpoint's is saved.
```

1.4 Tutorial 2: Prepare Datasets

In this section, we will detail how to prepare data and adopt the proper dataset in our repo for different methods.

We support multiple datasets of different tasks. There are two ways to use datasets for training and testing models in MMEditing:

1. Using downloaded datasets directly
2. Preprocessing downloaded datasets before using them.

The structure of this guide is as follows:

1.4.1 Download datasets

You are supposed to download datasets from their homepage first. Most datasets are available after downloaded, so you only need to make sure the folder structure is correct and further preparation is not necessary. For example, you can simply prepare Vimeo90K-triplet datasets by downloading datasets from homepage.

1.4.2 Prepare datasets

Some datasets need to be preprocessed before training or testing. We support many scripts to prepare datasets in tools/dataset_converters. And you can follow the tutorials of every dataset to run scripts. For example, we recommend cropping the DIV2K images to sub-images. We provide a script to prepare cropped DIV2K dataset. You can run the following command:

```
python tools/dataset_converters/super-resolution/div2k/preprocess_div2k_dataset.py --
↪data-root ./data/DIV2K
```

1.4.3 The overview of the datasets in MMEditing

We support detailed tutorials and split them according to different tasks.

Please check our *dataset zoo* for data preparation of different tasks.

If you're interested in more details of datasets in MMEditing, please check the *advanced guides*.

1.5 Tutorial 3: Inference with Pre-trained Models

MMEditing provides APIs for you to easily play with state-of-the-art models on your own images or videos. Specifically, MMEditing supports various fundamental generative models, including: unconditional Generative Adversarial Networks (GANs), conditional GANs, internal learning, diffusion models, etc. MMEditing also supports various applications, including: image super-resolution, video super-resolution, video frame interpolation, image inpainting, image matting, image-to-image translation, etc.

In this section, we will specify how to play with our pre-trained models.

1.5.1 Sample images with unconditional GANs

MMEditing provides high-level APIs for sampling images with unconditional GANs. Here is an example of building StyleGAN2-256 and obtaining the synthesized images.

```
from mmedit.apis import init_model, sample_unconditional_model

# Specify the path to model config and checkpoint file
config_file = 'configs/styleganv2/stylegan2_c2_8xb4_ffhq-1024x1024.py'
# you can download this checkpoint in advance and use a local file path.
checkpoint_file = 'https://download.openmmlab.com/mmgen/stylegan2/stylegan2_c2_ffhq_1024_
↪b4x8_20210407_150045-618c9024.pth'

device = 'cuda:0'
# init a generative model
model = init_model(config_file, checkpoint_file, device=device)
# sample images
fake_imgs = sample_unconditional_model(model, 4)
```

Indeed, we have already provided a more friendly demo script to users. You can use demo/unconditional_demo.py with the following commands:

```
python demo/unconditional_demo.py \
    ${CONFIG_FILE} \
    ${CHECKPOINT} \
    [--save-path ${SAVE_PATH}] \
    [--device ${GPU_ID}]
```

Note that more arguments are also offered to customize your sampling procedure. Please use `python demo/unconditional_demo.py --help` to check more details.

1.5.2 Sample images with conditional GANs

MMEditing provides high-level APIs for sampling images with conditional GANs. Here is an example for building SAGAN-128 and obtaining the synthesized images.

```
from mmedit.apis import init_model, sample_conditional_model

# Specify the path to model config and checkpoint file
config_file = 'configs/sagan/sagan_woReLUinplace-Glr1e-4_Dlr4e-4_noaug-ndisc1-8xb32-
↪bigGAN-sch_imagenet1k-128x128.py'
# you can download this checkpoint in advance and use a local file path.
checkpoint_file = 'https://download.openmmlab.com/mmgen/sagan/sagan_128_woReLUinplace_
↪noaug_bigGAN_imagenet1k_b32x8_Glr1e-4_Dlr-4e-4_ndisc1_20210818_210232-3f5686af.pth'

device = 'cuda:0'
# init a generative model
model = init_model(config_file, checkpoint_file, device=device)
# sample images with random label
fake_imgs = sample_conditional_model(model, 4)

# sample images with the same label
fake_imgs = sample_conditional_model(model, 4, label=0)
```

(continues on next page)

(continued from previous page)

```
# sample images with specific labels
fake_imgs = sample_conditional_model(model, 4, label=[0, 1, 2, 3])
```

Indeed, we have already provided a more friendly demo script to users. You can use demo/conditional_demo.py with the following commands:

```
python demo/conditional_demo.py \
    ${CONFIG_FILE} \
    ${CHECKPOINT} \
    [--label] ${LABEL} \
    [--samples-per-classes] ${SAMPLES_PER_CLASSES} \
    [--sample-all-classes] \
    [--save-path ${SAVE_PATH}] \
    [--device ${GPU_ID}]
```

If `--label` is not passed, images with random labels would be generated. If `--label` is passed, we would generate `${SAMPLES_PER_CLASSES}` images for each input label. If `sample_all_classes` is set true in command line, `--label` would be ignored and the generator will output images for all categories.

Note that more arguments are also offered to customizing your sampling procedure. Please use `python demo/conditional_demo.py --help` to check more details.

1.5.3 Sample images with diffusion models

MMEditing provides high-level APIs for sampling images with diffusion models. Here is an example for building I-DDPM and obtaining the synthesized images.

```
from mmedit.apis import init_model, sample_ddpm_model

# Specify the path to model config and checkpoint file
config_file = 'configs/improved_ddpm/ddpm_cosine-hybird-timestep-4k_16xb8-1500kiters_
↪imagenet1k-64x64.py'
# you can download this checkpoint in advance and use a local file path.
checkpoint_file = 'https://download.openmmlab.com/mmgen/improved_ddpm/ddpm_cosine_hybird_
↪timestep-4k_imagenet1k_64x64_b8x16_1500k_20220103_223919-b8f1a310.pth'
device = 'cuda:0'
# init a generative model
model = init_model(config_file, checkpoint_file, device=device)
# sample images
fake_imgs = sample_ddpm_model(model, 4)
```

Indeed, we have already provided a more friendly demo script to users. You can use demo/ddpm_demo.py with the following commands:

```
python demo/ddpm_demo.py \
    ${CONFIG_FILE} \
    ${CHECKPOINT} \
    [--save-path ${SAVE_PATH}] \
    [--device ${GPU_ID}]
```

Note that more arguments are also offered to customizing your sampling procedure. Please use `python demo/ddpm_demo.py --help` to check more details.

1.5.4 Run a demo of image inpainting

You can use the following commands to test images for inpainting.

```
python demo/inpainting_demo.py \
    ${CONFIG_FILE} \
    ${CHECKPOINT_FILE} \
    ${MASKED_IMAGE_FILE} \
    ${MASK_FILE} \
    ${SAVE_FILE} \
    [--imshow] \
    [--device ${GPU_ID}]
```

If `--imshow` is specified, the demo will also show image with opencv. Examples:

```
python demo/inpainting_demo.py \
    configs/global_local/gl_256x256_8x12_celeba.py \
    https://download.openmmlab.com/mmediting/inpainting/global_local/gl_256x256_8x12_
↪celeba_20200619-5af0493f.pth \
    tests/data/image/celeba_test.png \
    tests/data/image/bbox_mask.png \
    tests/data/pred/inpainting_celeba.png
```

The predicted inpainting result will be saved in `tests/data/pred/inpainting_celeba.png`.

1.5.5 Run a demo of image matting

You can use the following commands to test a pair of images and trimap.

```
python demo/matting_demo.py \
    ${CONFIG_FILE} \
    ${CHECKPOINT_FILE} \
    ${IMAGE_FILE} \
    ${TRIMAP_FILE} \
    ${SAVE_FILE} \
    [--imshow] \
    [--device ${GPU_ID}]
```

If `--imshow` is specified, the demo will also show image with opencv. Examples:

```
python demo/matting_demo.py \
    configs/dim/dim_stage3_v16_pln_1x1_1000k_comp1k.py \
    https://download.openmmlab.com/mmediting/mattors/dim/dim_stage3_v16_pln_1x1_1000k_
↪comp1k_SAD-50.6_20200609_111851-647f24b6.pth \
    tests/data/matting_dataset/merged/GT05.jpg \
    tests/data/matting_dataset/trimap/GT05.png \
    tests/data/pred/GT05.png
```

The predicted alpha matte will be saved in `tests/data/pred/GT05.png`.

1.5.6 Run a demo of image super-resolution

You can use the following commands to test an image for restoration.

```
python demo/restoration_demo.py \
    ${CONFIG_FILE} \
    ${CHECKPOINT_FILE} \
    ${IMAGE_FILE} \
    ${SAVE_FILE} \
    [--imshow] \
    [--device ${GPU_ID}] \
    [--ref-path ${REF_PATH}]
```

If `--imshow` is specified, the demo will also show image with opencv. Examples:

```
python demo/restoration_demo.py \
    configs/esrgan/esrgan_x4c64b23g32_g1_400k_div2k.py \
    https://download.openmmlab.com/mmediting/restorers/esrgan/esrgan_x4c64b23g32_1x16_
↪400k_div2k_20200508-f8ccaf3b.pth \
    tests/data/image/lq/baboon_x4.png \
    demo/demo_out_baboon.png
```

You can test Ref-SR by providing `--ref-path`. Examples:

```
python demo/restoration_demo.py \
    configs/ttsr/ttsr-gan_x4_c64b16_g1_500k_CUFED.py \
    https://download.openmmlab.com/mmediting/restorers/ttsr/ttsr-gan_x4_c64b16_g1_500k_
↪CUFED_20210626-2ab28ca0.pth \
    tests/data/frames/sequence/gt/sequence_1/00000000.png \
    demo/demo_out.png \
    --ref-path tests/data/frames/sequence/gt/sequence_1/00000001.png
```

1.5.7 Run a demo of facial restoration

You can use the following commands to test a face image for restoration.

```
python demo/restoration_face_demo.py \
    ${CONFIG_FILE} \
    ${CHECKPOINT_FILE} \
    ${IMAGE_FILE} \
    ${SAVE_FILE} \
    [--upscale-factor] \
    [--face-size] \
    [--imshow] \
    [--device ${GPU_ID}]
```

If `--imshow` is specified, the demo will also show image with opencv. Examples:

```
python demo/restoration_face_demo.py \
    configs/glean/glean_in128out1024_4xb2-300k_ffhq-celeba-hq.py \
    https://download.openmmlab.com/mmediting/restorers/glean/glean_in128out1024_4x2_300k_
↪ffhq_celebahq_20210812-acbcb04f.pth \
    tests/data/image/face/000001.png \
```

(continues on next page)

(continued from previous page)

```
    tests/data/pred/000001.png \
    --upscale-factor 4
```

1.5.8 Run a demo of video super-resolution

You can use the following commands to test a video for restoration.

```
python demo/restoration_video_demo.py \
    ${CONFIG_FILE} \
    ${CHECKPOINT_FILE} \
    ${INPUT_DIR} \
    ${OUTPUT_DIR} \
    [--window-size=${WINDOW_SIZE}] \
    [--device ${GPU_ID}]
```

It supports both the sliding-window framework and the recurrent framework. Examples:

EDVR:

```
python demo/restoration_video_demo.py \
    configs/edvr/edvrm_wotsa_x4_g8_600k_reds.py \
    https://download.openmmlab.com/mmediting/restorers/edvr/edvrm_wotsa_x4_8x4_600k_reds_
↪20200522-0570e567.pth \
    data/Vid4/BIx4/calendar/ \
    demo/output \
    --window-size=5
```

BasicVSR:

```
python demo/restoration_video_demo.py \
    configs/basicvsr/basicvsr_reds4.py \
    https://download.openmmlab.com/mmediting/restorers/basicvsr/basicvsr_reds4_20120409-
↪0e599677.pth \
    data/Vid4/BIx4/calendar/ \
    demo/output
```

The restored video will be saved in `output/`.

1.5.9 Run a demo of video frame interpolation

You can use the following commands to test a video for frame interpolation.

```
python demo/video_interpolation_demo.py \
    ${CONFIG_FILE} \
    ${CHECKPOINT_FILE} \
    ${INPUT_DIR} \
    ${OUTPUT_DIR} \
    [--fps-multiplier ${FPS_MULTIPLIER}] \
    [--fps ${FPS}]
```

`${INPUT_DIR}` / `${OUTPUT_DIR}` can be a path of video file or the folder of a sequence of ordered images. If `${OUTPUT_DIR}` is a path of video file, its frame rate can be determined by the frame rate of input video and `fps_multiplier`, or be determined by `fps` directly (the former has higher priority). Examples:

The frame rate of output video is determined by the frame rate of input video and `fps_multiplier`

```
python demo/video_interpolation_demo.py \
    configs/cain/cain_b5_g1b32_vimeo90k_triplet.py \
    https://download.openmmlab.com/mmediting/video_interpolators/cain/cain_b5_320k_vimeo-
↪triple_20220117-647f3de2.pth \
    tests/data/test_inference.mp4 \
    tests/data/test_inference_vfi_out.mp4 \
    --fps-multiplier 2.0
```

The frame rate of output video is determined by `fps`:

```
python demo/video_interpolation_demo.py \
    configs/cain/cain_b5_g1b32_vimeo90k_triplet.py \
    https://download.openmmlab.com/mmediting/video_interpolators/cain/cain_b5_320k_vimeo-
↪triple_20220117-647f3de2.pth \
    tests/data/test_inference.mp4 \
    tests/data/test_inference_vfi_out.mp4 \
    --fps 60.0
```

1.5.10 Run a demo of image translation models

MMEditing provides high-level APIs for translating images by using image translation models. Here is an example of building Pix2Pix and obtaining the translated images.

```
from mmedit.apis import init_model, sample_img2img_model

# Specify the path to model config and checkpoint file
config_file = 'configs/pix2pix/pix2pix_vanilla-unet-bn_wo-jitter-flip-4xb1-190kiters_
↪edges2shoes.py'
# you can download this checkpoint in advance and use a local file path.
checkpoint_file = 'https://download.openmmlab.com/mmgen/pix2pix/refactor/pix2pix_vanilla_
↪unet_bn_wo_jitter_flip_1x4_186840_edges2shoes_convert-bgr_20210902_170902-0c828552.pth'
# Specify the path to image you want to translate
image_path = 'tests/data/paired/test/33_AB.jpg'
device = 'cuda:0'
# init a generative model
model = init_model(config_file, checkpoint_file, device=device)
# translate a single image
translated_image = sample_img2img_model(model, image_path, target_domain='photo')
```

Indeed, we have already provided a more friendly demo script to users. You can use demo/translation_demo.py with the following commands:

```
python demo/translation_demo.py \
    ${CONFIG_FILE} \
    ${CHECKPOINT} \
    ${IMAGE_PATH}
    [--save-path ${SAVE_PATH}] \
    [--device ${GPU_ID}]
```

Note that more customized arguments are also offered to customize your sampling procedure. Please use `python demo/translation_demo.py --help` to check more details.

1.6 Tutorial 4: Train and Test in MMEditing

In this section, we introduce how to test and train models in MMEditing.

In this section, we provide the following guides:

1.6.1 Prerequisite

Users need to *prepare dataset* first to enable training and testing models in MMEditing.

1.6.2 Test a model in MMEditing

Test with a single GPUs

You can use the following commands to test a pre-trained model with single GPUs.

```
python tools/test.py ${CONFIG_FILE} ${CHECKPOINT_FILE}
```

For example,

```
python tools/test.py configs/example_config.py work_dirs/example_exp/example_model_
↪20200202.pth
```

Test with multiple GPUs

MMEditing supports testing with multiple GPUs, which can largely save your time in testing models. You can use the following commands to test a pre-trained model with multiple GPUs.

```
./tools/dist_test.sh ${CONFIG_FILE} ${CHECKPOINT_FILE} ${GPU_NUM}
```

For example,

```
./tools/dist_test.sh configs/example_config.py work_dirs/example_exp/example_model_
↪20200202.pth
```

Test with Slurm

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

```
[GPUS=${GPUS}] ./tools/slurm_test.sh ${PARTITION} ${JOB_NAME} ${CONFIG_FILE} $
↪{CHECKPOINT_FILE}
```

Here is an example of using 8 GPUs to test an example model on the 'dev' partition with the job name 'test'.

```
GPUS=8 ./tools/slurm_test.sh dev test configs/example_config.py work_dirs/example_exp/
↪example_model_20200202.pth
```

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

Test with specific metrics

MMEditing provides various **evaluation metrics**, i.e., MS-SSIM, SWD, IS, FID, Precision&Recall, PPL, Equivarience, TransFID, TransIS, etc. We have provided unified evaluation scripts in tools/test.py for all models. If users want to evaluate their models with some metrics, you can add the `metrics` into your config file like this:

```
# at the end of the configs/styleganv2/stylegan2_c2_ffhq_256_b4x8_800k.py
metrics = [
    dict(
        type='FrechetInceptionDistance',
        prefix='FID-Full-50k',
        fake_nums=50000,
        inception_style='StyleGAN',
        sample_model='ema'),
    dict(type='PrecisionAndRecall', fake_nums=50000, prefix='PR-50K'),
    dict(type='PerceptualPathLength', fake_nums=50000, prefix='ppl-w')
]
```

As above, `metrics` consist of multiple metric dictionaries. Each metric will contain `type` to indicate the category of the metric. `fake_nums` denotes the number of images generated by the model. Some metrics will output a dictionary of results, you can also set `prefix` to specify the prefix of the results. If you set the prefix of FID as `FID-Full-50k`, then an example of output may be

```
FID-Full-50k/fid: 3.6561  FID-Full-50k/mean: 0.4263  FID-Full-50k/cov: 3.2298
```

Then users can test models with the command below:

```
bash tools/dist_test.sh ${CONFIG_FILE} ${CKPT_FILE}
```

If you are in slurm environment, please switch to the tools/slurm_test.sh by using the following commands:

```
sh slurm_test.sh ${PLATFORM} ${JOBNAME} ${CONFIG_FILE} ${CKPT_FILE}
```

1.6.3 Train a model in MMEditing

MMEditing supports multiple ways of training:

1. *Train with a single GPU*
2. *Train with multiple GPUs*
3. *Train with multiple nodes*
4. *Train with Slurm*

Specifically, all outputs (log files and checkpoints) will be saved to the working directory, which is specified by `work_dir` in the config file.

Train with a single GPU

```
CUDA_VISIBLE=0 python tools/train.py configs/example_config.py --work-dir work_dirs/
↪example
```

Train with multiple nodes

To launch distributed training on multiple machines, which can be accessed via IPs, run the following commands:

On the first machine:

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

On the second machine:

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

To speed up network communication, high speed network hardware, such as Infiniband, is recommended. Please refer to PyTorch docs for more information.

Train with multiple GPUs

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

Train with Slurm

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

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

Here is an example of using 8 GPUs to train an inpainting model on the dev partition.

```
GPUS=8 ./tools/slurm_train.sh dev configs/inpainting/gl_places.py /nfs/xxxx/gl_places_256
```

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

Optional arguments

- `--amp`: This argument is used for fixed-precision training.
- `--resume`: This argument is used for auto resume if the training is aborted.

1.6.4 Train with specific evaluation metrics

Benefit from the `mmengine`'s `Runner`. We can evaluate model during training in a simple way as below.

```
# define metrics
metrics = [
    dict(
        type='FrechetInceptionDistance',
        prefix='FID-Full-50k',
        fake_nums=50000,
        inception_style='StyleGAN')
]

# define dataloader
val_dataloader = dict(
    batch_size=128,
    num_workers=8,
    dataset=dict(
        type='BasicImageDataset',
        data_root='data/celeba-cropped/',
        pipeline=[
            dict(type='LoadImageFromFile', key='img'),
            dict(type='Resize', scale=(64, 64)),
            dict(type='PackEditInputs')
        ]),
    sampler=dict(type='DefaultSampler', shuffle=False),
    persistent_workers=True)

# define val interval
train_cfg = dict(by_epoch=False, val_begin=1, val_interval=10000)

# define val loop and evaluator
val_cfg = dict(type='GenValLoop')
val_evaluator = dict(type='GenEvaluator', metrics=metrics)
```

You can set `val_begin` and `val_interval` to adjust when to begin validation and interval of validation.

For details of metrics, refer to *metrics' guide*.

1.7 Tutorial 5: Visualization

The visualization of images is an important way to measure the quality of image processing, editing and synthesis. Using `visualizer` in config file can save visual results when training or testing. You can follow MMEngine Documents to learn the usage of visualization. MMEditing provides a rich set of visualization functions. In this tutorial, we introduce the usage of the visualization functions provided by MMEditing.

- *Overview*
- *Visualization hook*
- *Visualizer*
- *VisBackend*

1.7.1 Overview

In MMEditing, the visualization of the training or testing process requires the configuration of three components: VisualizationHook, Visualizer, and VisBackend.

VisualizationHook fetches the visualization results of the model output in fixed intervals during training and passes them to Visualizer. **Visualizer** is responsible for converting the original visualization results into the desired type (png, gif, etc.) and then transferring them to **VisBackend** for storage or display.

Visualization configuration of GANs

For GAN models, such as StyleGAN and SAGAN, a usual configuration is shown below:

```
# VisualizationHook
custom_hooks = [
    dict(
        type='GenVisualizationHook',
        interval=5000,  # visualization interval
        fixed_input=True,  # whether use fixed noise input to generate images
        vis_kwargs_list=dict(type='GAN', name='fake_img')  # pre-defined visualization␣
↪arguments for GAN models
    )
]
# VisBackend
vis_backends = [
    dict(type='GenVisBackend'),  # vis_backend for saving images to file system
    dict(type='WandbGenVisBackend',  # vis_backend for uploading images to Wandb
        init_kwargs=dict(
            project='MMEditing',   # project name for Wandb
            name='GAN-Visualization-Demo'  # name of the experiment for Wandb
        ))
]
# Visualizer
visualizer = dict(type='GenVisualizer', vis_backends=vis_backends)
```

If you apply Exponential Moving Average (EMA) to a generator and want to visualize the EMA model, you can modify config of `VisualizationHook` as below:

```
custom_hooks = [
    dict(
        type='GenVisualizationHook',
        interval=5000,
        fixed_input=True,
        # vis ema and orig in `fake_img` at the same time
        vis_kwargs_list=dict(
            type='Noise',
            name='fake_img',  # save images with prefix `fake_img`
            sample_model='ema/orig',  # specified kwargs for `NoiseSampler`
            target_keys=['ema.fake_img', 'orig.fake_img']  # specific key to␣
↪visualization
        ))
]
```

Visualization configuration of image translation models

For Translation models, such as CycleGAN and Pix2Pix, visualization configs can be formed as below:

```
# VisualizationHook
custom_hooks = [
    dict(
        type='GenVisualizationHook',
        interval=5000,
        fixed_input=True,
        vis_kwargs_list=[
            dict(
                type='Translation',  # Visualize results on the training set
                name='trans'),  #  save images with prefix `trans`
            dict(
                type='Translationval',  # Visualize results on the validation set
                name='trans_val'),  #  save images with prefix `trans_val`
        ])
]
# VisBackend
vis_backends = [
    dict(type='GenVisBackend'),  # vis_backend for saving images to file system
    dict(type='WandbGenVisBackend',  # vis_backend for uploading images to Wandb
        init_kwargs=dict(
            project='MMEditing',   # project name for Wandb
            name='Translation-Visualization-Demo'  # name of the experiment for Wandb
        ))
]
# Visualizer
visualizer = dict(type='GenVisualizer', vis_backends=vis_backends)
```

Visualization configuration of diffusion models

For Diffusion models, such as Improved-DDPM, we can use the following configuration to visualize the denoising process through a gif:

```
# VisualizationHook
custom_hooks = [
    dict(
        type='GenVisualizationHook',
        interval=5000,
        fixed_input=True,
        vis_kwargs_list=dict(type='DDPMDenoising'))  # pre-defined visualization␣
↪argument for DDPM models
]
# VisBackend
vis_backends = [
    dict(type='GenVisBackend'),  # vis_backend for saving images to file system
    dict(type='WandbGenVisBackend',  # vis_backend for uploading images to Wandb
        init_kwargs=dict(
            project='MMEditing',   # project name for Wandb
            name='Diffusion-Visualization-Demo'  # name of the experiment for Wandb
        ))
]
# Visualizer
visualizer = dict(type='GenVisualizer', vis_backends=vis_backends)
```

Visualization configuration of inpainting models

For inpainting models, such as AOT-GAN and Global&Local, a usual configuration is shown below:

```
# VisBackend
vis_backends = [dict(type='LocalVisBackend')]
# Visualizer
visualizer = dict(
    type='ConcatImageVisualizer',
    vis_backends=vis_backends,
    fn_key='gt_path',
    img_keys=['gt_img', 'input', 'pred_img'],
    bgr2rgb=True)
# VisualizationHook
custom_hooks = [dict(type='BasicVisualizationHook', interval=1)]
```

Visualization configuration of matting models

For matting models, such as DIM and GCA, a usual configuration is shown below:

```
# VisBackend
vis_backends = [dict(type='LocalVisBackend')]
# Visualizer
visualizer = dict(
    type='ConcatImageVisualizer',
```

(continues on next page)

(continued from previous page)

```
    vis_backends=vis_backends,
    fn_key='trimap_path',
    img_keys=['pred_alpha', 'trimap', 'gt_merged', 'gt_alpha'],
    bgr2rgb=True)
# VisualizationHook
custom_hooks = [dict(type='BasicVisualizationHook', interval=1)]
```

Visualization configuration of SISR/VSR/VFI models

For SISR/VSR/VFI models, such as EDSR, EDVR and CAIN, a usual configuration is shown below:

```
# VisBackend
vis_backends = [dict(type='LocalVisBackend')]
# Visualizer
visualizer = dict(
    type='ConcatImageVisualizer',
    vis_backends=vis_backends,
    fn_key='gt_path',
    img_keys=['gt_img', 'input', 'pred_img'],
    bgr2rgb=False)
# VisualizationHook
custom_hooks = [dict(type='BasicVisualizationHook', interval=1)]
```

The specific configuration of the `VisualizationHook`, `Visualizer` and `VisBackend` components are described below

1.7.2 Visualization Hook

In MMEditing, we use `BasicVisualizationHook` and `GenVisualizationHook` as `VisualizationHook`. `GenVisualizationHook` supports three following cases.

(1) Modify `vis_kwargs_list` to visualize the output of the model under specific inputs , which is suitable for visualization of the generated results of GAN and translation results of Image-to-Image-Translation models under specific data input, etc. Below are two typical examples:

```
# input as dict
vis_kwargs_list = dict(
    type='Noise',  # use 'Noise' sampler to generate model input
    name='fake_img',  # define prefix of saved images
)

# input as list of dict
vis_kwargs_list = [
    dict(type='Arguments',  # use `Arguments` sampler to generate model input
         name='arg_output',  # define prefix of saved images
         vis_mode='gif',  # specific visualization mode as GIF
         forward_kwargs=dict(forward_mode='sampling', sample_kwargs=dict(show_
↪pbar=True))  # specific kwargs for `Arguments` sampler
    ),
    dict(type='Data',  # use `Data` sampler to feed data in dataloader to model as input
         n_samples=36,  # specific how many samples want to generate
```

(continues on next page)

(continued from previous page)

```
        fixed_input=False,  # specific do not use fixed input for each visualization␣
↪process
    )
]
```

`vis_kwargs_list` takes dict or list of dict as input. Each of dict must contain a `type` field indicating the **type of sampler** used to generate the model input, and each of the dict must also contain the keyword fields necessary for the sampler (e.g. `ArgumentSampler` requires that the argument dictionary contain `forward_kwargs`).

To be noted that, this content is checked by the corresponding sampler and is not restricted by `GenVisHook`.

In addition, the other fields are generic fields (e.g. `n_samples`, `n_row`, `name`, `fixed_input`, etc.). If not passed in, the default values from the GenVisHook initialization will be used.

For the convenience of users, MMEditing has pre-defined visualization parameters for **GAN**, **Translation models**, **SinGAN** and **Diffusion models**, and users can directly use the predefined visualization methods by using the following configuration:

```
vis_kwargs_list = dict(type='GAN')
vis_kwargs_list = dict(type='SinGAN')
vis_kwargs_list = dict(type='Translation')
vis_kwargs_list = dict(type='TranslationVal')
vis_kwargs_list = dict(type='TranslationTest')
vis_kwargs_list = dict(type='DDPMDenoising')
```

1.7.3 Visualizer

In MMEditing, we implement `ConcatImageVisualizer` and `GenVisualizer`, which inherit from `mmengine.Visualizer`. The base class of `Visualizer` is `ManagerMixin` and this makes `Visualizer` a globally unique object. After being instantiated, `Visualizer` can be called at anywhere of the code by `Visualizer.get_current_instance()`, as shown below:

```
# configs
vis_backends = [dict(type='GenVisBackend')]
visualizer = dict(
    type='GenVisualizer', vis_backends=vis_backends, name='visualizer')
```

```
# `get_instance()` is called for globally unique instantiation
VISUALIZERS.build(cfg.visualizer)

# Once instantiated by the above code, you can call the `get_current_instance` method at␣
↪any location to get the visualizer
visualizer = Visualizer.get_current_instance()
```

The core interface of `Visualizer` is `add_datasample`. Through this interface, This interface will call the corresponding drawing function according to the corresponding `vis_mode` to obtain the visualization result in `np.ndarray` type. Then `show` or `add_image` will be called to directly show the results or pass the visualization result to the predefined vis_backend.

1.7.4 VisBackend

In general, users do not need to manipulate `VisBackend` objects, only when the current visualization storage can not meet the needs, users will want to manipulate the storage backend directly. MMEditing supports a variety of different visualization backends, including:

- Basic VisBackend of MMEngine: including LocalVisBackend, TensorboardVisBackend and WandbVisBackend. You can follow MMEngine Documents to learn more about them
- GenVisBackend: Backend for **File System**. Save the visualization results to the corresponding position.
- TensorboardGenVisBackend: Backend for **Tensorboard**. Send the visualization results to Tensorboard.
- PaviGenVisBackend: Backend for **Pavi**. Send the visualization results to Tensorboard.
- WandbGenVisBackend: Backend for **Wandb**. Send the visualization results to Tensorboard.

One `Visualizer` object can have access to any number of VisBackends and users can access to the backend by their class name in their code.

```
# configs
vis_backends = [dict(type='GenVisualizer'), dict(type='WandbVisBackend')]
visualizer = dict(
    type='GenVisualizer', vis_backends=vis_backends, name='visualizer')
```

```
# code
VISUALIZERS.build(cfg.visualizer)
visualizer = Visualizer.get_current_instance()

# access to the backend by class name
gen_vis_backend = visualizer.get_backend('GenVisBackend')
gen_wandb_vis_backend = visualizer.get_backend('GenWandbVisBackend')
```

When there are multiply VisBackend with the same class name, user must specific name for each VisBackend.

```
# configs
vis_backends = [
    dict(type='GenVisBackend', name='gen_vis_backend_1'),
    dict(type='GenVisBackend', name='gen_vis_backend_2')
]
visualizer = dict(
    type='GenVisualizer', vis_backends=vis_backends, name='visualizer')
```

```
# code
VISUALIZERS.build(cfg.visualizer)
visualizer = Visualizer.get_current_instance()

local_vis_backend_1 = visualizer.get_backend('gen_vis_backend_1')
local_vis_backend_2 = visualizer.get_backend('gen_vis_backend_2')
```

1.8 Tutorial 6: Useful Tools

We provide lots of useful tools under `tools/` directory.

The structure of this guide is as follows:

1.8.1 Get the FLOPs and params

We provide a script adapted from flops-counter.pytorch to compute the FLOPs and params of a given model.

```
python tools/analysis_tools/get_flops.py ${CONFIG_FILE} [--shape ${INPUT_SHAPE}]
```

For example,

```
python tools/analysis_tools/get_flops.py configs/resotorer/srresnet.py --shape 40 40
```

You will get the result like this.

```
==============================
Input shape: (3, 40, 40)
Flops: 4.07 GMac
Params: 1.52 M
==============================
```

Note: This tool is still experimental and we do not guarantee that the number is correct. You may well use the result for simple comparisons, but double check it before you adopt it in technical reports or papers.

(1) FLOPs are related to the input shape while parameters are not. The default input shape is (1, 3, 250, 250). (2) Some operators are not counted in FLOPs like GN and custom operators. You can add support for new operators by modifying `mmcv/cnn/utils/flops_counter.py`.

1.8.2 Publish a model

Before you upload a model to AWS, you may want to

1. convert model weights to CPU tensors
2. delete the optimizer states and
3. compute the hash of the checkpoint file and append time and the hash id to the filename.

```
python tools/model_converters/publish_model.py ${INPUT_FILENAME} ${OUTPUT_FILENAME}
```

E.g.,

```
python tools/model_converters/publish_model.py work_dirs/stylegan2/latest.pth stylegan2_
↪c2_8xb4_ffhq-1024x1024.pth
```

The final output filename will be `stylegan2_c2_8xb4_ffhq-1024x1024_{time}-{hash id}.pth`.

1.8.3 Print full config

MMGeneration incorporates config mechanism to set parameters used for training and testing models. With our *config* mechanism, users can easily conduct extensive experiments without hard coding. If you wish to inspect the config file, you may run `python tools/misc/print_config.py /PATH/TO/CONFIG` to see the complete config.

An Example:

```
python tools/misc/print_config.py configs/styleganv2/stylegan2_c2-PL_8xb4-fp16-partial-
↪GD-no-scaler-800kiters_ffhq-256x256.py
```

1.9 Tutorial 7: Deploy Models in MMEditing

MMDeploy is an open-source deep learning model deployment toolset. MMDeploy supports deploying models in MMEditing. Please refer to MMDeploy for more information.

1.10 Tutorial 8: Using Metrics in MMEditing

MMEditing supports **17 metrics** to assess the quality of models.

Please refer to *Train and Test in MMEditing* for usages.

Here, we will specify the details of different metrics one by one.

The structure of this guide are as follows:

1. *MAE*
2. *MSE*
3. *PSNR*
4. *SNR*
5. *SSIM*
6. *NIQE*
7. *SAD*
8. *MattingMSE*
9. *GradientError*
10. *ConnectivityError*
11. *FID and TransFID*
12. *IS and TransIS*
13. *Precision and Recall*
14. *PPL*
15. *SWD*
16. *MS-SSIM*
17. *Equivarience*

1.10.1 MAE

MAE is Mean Absolute Error metric for image. To evaluate with MAE, please add the following configuration in the config file:

```
val_evaluator = [
    dict(type='MAE'),
]
```

1.10.2 MSE

MSE is Mean Squared Error metric for image. To evaluate with MSE, please add the following configuration in the config file:

```
val_evaluator = [
    dict(type='MSE'),
]
```

1.10.3 PSNR

PSNR is Peak Signal-to-Noise Ratio. Our implement refers to https://en.wikipedia.org/wiki/Peak_signal-to-noise_ratio. To evaluate with PSNR, please add the following configuration in the config file:

```
val_evaluator = [
    dict(type='PSNR'),
]
```

1.10.4 SNR

SNR is Signal-to-Noise Ratio. Our implementation refers to https://en.wikipedia.org/wiki/Signal-to-noise_ratio. To evaluate with SNR, please add the following configuration in the config file:

```
val_evaluator = [
    dict(type='SNR'),
]
```

1.10.5 SSIM

SSIM is structural similarity for image, proposed in Image quality assessment: from error visibility to structural similarity. The results of our implementation are the same as that of the official released MATLAB code in https://ece.uwaterloo.ca/~z70wang/research/ssim/. To evaluate with SSIM, please add the following configuration in the config file:

```
val_evaluator = [
    dict(type='SSIM'),
]
```

1.10.6 NIQE

NIQE is Natural Image Quality Evaluator metric, proposed in Making a "Completely Blind" Image Quality Analyzer. Our implementation could produce almost the same results as the official MATLAB codes: http://live.ece.utexas.edu/research/quality/niqe_release.zip.

To evaluate with NIQE, please add the following configuration in the config file:

```
val_evaluator = [
    dict(type='NIQE'),
]
```

1.10.7 SAD

SAD is Sum of Absolute Differences metric for image matting. This metric compute per-pixel absolute difference and sum across all pixels. To evaluate with SAD, please add the following configuration in the config file:

```
val_evaluator = [
    dict(type='SAD'),
]
```

1.10.8 MattingMSE

MattingMSE is Mean Squared Error metric for image matting. To evaluate with MattingMSE, please add the following configuration in the config file:

```
val_evaluator = [
    dict(type='MattingMSE'),
]
```

1.10.9 GradientError

GradientError is Gradient error for evaluating alpha matte prediction. To evaluate with GradientError, please add the following configuration in the config file:

```
val_evaluator = [
    dict(type='GradientError'),
]
```

1.10.10 ConnectivityError

ConnectivityError is Connectivity error for evaluating alpha matte prediction. To evaluate with ConnectivityError, please add the following configuration in the config file:

```
val_evaluator = [
    dict(type='ConnectivityError'),
]
```

1.10.11 FID and TransFID

Fréchet Inception Distance is a measure of similarity between two datasets of images. It was shown to correlate well with the human judgment of visual quality and is most often used to evaluate the quality of samples of Generative Adversarial Networks. FID is calculated by computing the Fréchet distance between two Gaussians fitted to feature representations of the Inception network.

In `MMEditing`, we provide two versions for FID calculation. One is the commonly used PyTorch version and the other one is used in StyleGAN paper. Meanwhile, we have compared the difference between these two implementations in the StyleGAN2-FFHQ1024 model (the details can be found here). Fortunately, there is a marginal difference in the final results. Thus, we recommend users adopt the more convenient PyTorch version.

About PyTorch version and Tero's version: The commonly used PyTorch version adopts the modified InceptionV3 network to extract features for real and fake images. However, Tero's FID requires a script module for Tensorflow InceptionV3. Note that applying this script module needs `PyTorch >= 1.6.0`.

About extracting real inception data: For the users' convenience, the real features will be automatically extracted at test time and saved locally, and the stored features will be automatically read at the next test. Specifically, we will calculate a hash value based on the parameters used to calculate the real features, and use the hash value to mark the feature file, and when testing, if the `inception_pkl` is not set, we will look for the feature in `MMEDIT_CACHE_DIR` (~/.cache/openmmlab/mmedit/). If cached inception pkl is not found, then extracting will be performed.

To use the FID metric, you should add the metric in a config file like this:

```
metrics = [
    dict(
        type='FrechetInceptionDistance',
        prefix='FID-Full-50k',
        fake_nums=50000,
        inception_style='StyleGAN',
        sample_model='ema')
]
```

If you work on an new machine, then you can copy the `pkl` files in `MMEDIT_CACHE_DIR` and copy them to new machine and set `inception_pkl` field.

```
metrics = [
    dict(
        type='FrechetInceptionDistance',
        prefix='FID-Full-50k',
        fake_nums=50000,
        inception_style='StyleGAN',
        inception_pkl=
        'work_dirs/inception_pkl/inception_state-capture_mean_cov-full-
↪33ad4546f8c9152e4b3bdb1b0c08dbaf.pkl',  # copied from old machine
        sample_model='ema')
]
```

`TransFID` has same usage as `FID`, but it's designed for translation models like `Pix2Pix` and `CycleGAN`, which is adapted for our evaluator. You can refer to *evaluation* for details.

1.10.12 IS and TransIS

Inception score is an objective metric for evaluating the quality of generated images, proposed in Improved Techniques for Training GANs. It uses an InceptionV3 model to predict the class of the generated images, and suppose that 1) If an image is of high quality, it will be categorized into a specific class. 2) If images are of high diversity, the range of images' classes will be wide. So the KL-divergence of the conditional probability and marginal probability can indicate the quality and diversity of generated images. You can see the complete implementation in `metrics.py`, which refers to https://github.com/sbarratt/inception-score-pytorch/blob/master/inception_score.py. If you want to evaluate models with `IS` metrics, you can add the `metrics` into your config file like this:

```
# at the end of the configs/biggan/biggan_2xb25-500kiters_cifar10-32x32.py
metrics = [
    xxx,
    dict(
        type='IS',
        prefix='IS-50k',
        fake_nums=50000,
        inception_style='StyleGAN',
        sample_model='ema')
]
```

To be noted that, the selection of Inception V3 and image resize method can significantly influence the final IS score. Therefore, we strongly recommend users may download the Tero's script model of Inception V3 (load this script model need torch >= 1.6) and use `Bicubic` interpolation with `Pillow` backend.

Corresponding to config, you can set `resize_method` and `use_pillow_resize` for image resizing. You can also set `inception_style` as `StyleGAN` for recommended tero's inception model, or `PyTorch` for torchvision's implementation. For environment without internet, you can download the inception's weights, and set `inception_path` to your inception model.

We also perform a survey on the influence of data loading pipeline and the version of pretrained Inception V3 on the IS result. All IS are evaluated on the same group of images which are randomly selected from the ImageNet dataset.

`TransIS` has same usage as `IS`, but it's designed for translation models like `Pix2Pix` and `CycleGAN`, which is adapted for our evaluator. You can refer to *evaluation* for details.

1.10.13 Precision and Recall

Our `Precision and Recall` implementation follows the version used in StyleGAN2. In this metric, a VGG network will be adopted to extract the features for images. Unfortunately, we have not found a PyTorch VGG implementation leading to similar results with Tero's version used in StyleGAN2. (About the differences, please see this file.) Thus, in our implementation, we adopt Teor's VGG network by default. Importantly, applying this script module needs `PyTorch >= 1.6.0`. If with a lower PyTorch version, we will use the PyTorch official VGG network for feature extraction.

To evaluate with `P&R`, please add the following configuration in the config file:

```
metrics = [
    dict(type='PrecisionAndRecall', fake_nums=50000, prefix='PR-50K')
]
```

1.10.14 PPL

Perceptual path length measures the difference between consecutive images (their VGG16 embeddings) when interpolating between two random inputs. Drastic changes mean that multiple features have changed together and that they might be entangled. Thus, a smaller PPL score appears to indicate higher overall image quality by experiments.
As a basis for our metric, we use a perceptually-based pairwise image distance that is calculated as a weighted difference between two VGG16 embeddings, where the weights are fit so that the metric agrees with human perceptual similarity judgments. If we subdivide a latent space interpolation path into linear segments, we can define the total perceptual length of this segmented path as the sum of perceptual differences over each segment, and a natural definition for the perceptual path length would be the limit of this sum under infinitely fine subdivision, but in practice we approximate it using a small subdivision $`\epsilon=10^{-4}`$. The average perceptual path length in latent space Z, over all possible endpoints, is therefore

$$`L_Z = E[\frac{1}{\epsilon^2}d(G(slerp(z_1,z_2;t))), G(slerp(z_1,z_2;t+\epsilon)))]`$$

Computing the average perceptual path length in latent space W is carried out in a similar fashion:

$$`L_Z = E[\frac{1}{\epsilon^2}d(G(slerp(z_1,z_2;t))), G(slerp(z_1,z_2;t+\epsilon)))]`$$

Where $`z_1, z_2 \sim P(z)`$, and $` t \sim U(0,1)`$ if we set sampling to full, $` t \in \{0,1\}`$ if we set sampling to end. $` G`$ is the generator(i.e. $` g \circ f`$ for style-based networks), and $` d(., .)`$ evaluates the perceptual distance between the resulting images.We compute the expectation by taking 100,000 samples (set num_images to 50,000 in our code).

You can find the complete implementation in metrics.py, which refers to https://github.com/rosinality/stylegan2-pytorch/blob/master/ppl.py. If you want to evaluate models with PPL metrics, you can add the metrics into your config file like this:

```
# at the end of the configs/styleganv2/stylegan2_c2_ffhq_1024_b4x8.py
metrics = [
    xxx,
    dict(type='PerceptualPathLength', fake_nums=50000, prefix='ppl-w')
]
```

1.10.15 SWD

Sliced Wasserstein distance is a discrepancy measure for probability distributions, and smaller distance indicates generated images look like the real ones. We obtain the Laplacian pyramids of every image and extract patches from the Laplacian pyramids as descriptors, then SWD can be calculated by taking the sliced Wasserstein distance of the real and fake descriptors. You can see the complete implementation in metrics.py, which refers to https://github.com/tkarras/progressive_growing_of_gans/blob/master/metrics/sliced_wasserstein.py. If you want to evaluate models with SWD metrics, you can add the metrics into your config file like this:

```
# at the end of the configs/dcgan/dcgan_1xb128-5epoches_lsun-bedroom-64x64.py
metrics = [
    dict(
        type='SWD',
        prefix='swd',
        fake_nums=16384,
        sample_model='orig',
        image_shape=(3, 64, 64))
]
```

1.10.16 MS-SSIM

Multi-scale structural similarity is used to measure the similarity of two images. We use MS-SSIM here to measure the diversity of generated images, and a low MS-SSIM score indicates the high diversity of generated images. You can see the complete implementation in `metrics.py`, which refers to https://github.com/tkarras/progressive_growing_of_gans/blob/master/metrics/ms_ssim.py. If you want to evaluate models with `MS-SSIM` metrics, you can add the `metrics` into your config file like this:

```
# at the end of the configs/dcgan/dcgan_1xb128-5epoches_lsun-bedroom-64x64.py
metrics = [
    dict(
        type='MS_SSIM', prefix='ms-ssim', fake_nums=10000,
        sample_model='orig')
]
```

1.10.17 Equivarience

Equivarience of generative models refer to the exchangeability of model forward and geometric transformations. Currently this metric is only calculated for StyleGANv3, you can see the complete implementation in `metrics.py`, which refers to https://github.com/NVlabs/stylegan3/blob/main/metrics/equivariance.py. If you want to evaluate models with `Equivarience` metrics, you can add the `metrics` into your config file like this:

```
# at the end of the configs/styleganv3/stylegan3-t_gamma2.0_8xb4-fp16-noaug_ffhq-256x256.
↪py
metrics = [
    dict(
        type='Equivariance',
        fake_nums=50000,
        sample_mode='ema',
        prefix='EQ',
        eq_cfg=dict(
            compute_eqt_int=True, compute_eqt_frac=True, compute_eqr=True))
]
```

1.11 Design Your Own Models

MMEditing is built upon MMEngine and MMCV, which enables users to design new models quickly, train and evaluate them easily. In this section, you will learn how to design your own models.

The structure of this guide are as follows:

1.11.1 Overview of models in MMEditing

In MMEditing, one algorithm can be splited two compents: **Model** and **Module**.

- **Model** are topmost wrappers and always inherint from `BaseModel` provided in MMEngine. **Model** is responsible to network forward, loss calculation and backward, parameters updating, etc. In MMEditing, **Model** should be registered as `MODELS`.
- **Module** includes the neural network **architectures** to train or inference, pre-defined **loss classes**, and **data preprocessors** to preprocess the input data batch. **Module** always present as elements of **Model**. In MMEditing, **Module** should be registered as **MODULES**.

Take DCGAN model as an example, generator and discriminator are the **Module**, which generate images and discriminate real or fake images. `DCGAN` is the **Model**, which take data from dataloader and train generator and discriminator alternatively.

You can find the implementation of **Model** and **Module** by the following link.

- **Model**:
  - Editors
- **Module**:
  - Layers
  - Losses
  - Data Preprocessor

1.11.2 An example of SRCNN

Here, we take the implementation of the classical image super-resolution model, SRCNN [1], as an example.

Step 1: Define the network of SRCNN

SRCNN is the first deep learning method for single image super-resolution [1]. To implement the network architecture of SRCNN, we need to create a new file `mmedit/models/editors/srgan/sr_resnet.py` and implement `class MSRResNet`.

In this step, we implement `class MSRResNet` by inheriting from `mmengine.models.BaseModule` and define the network architecture in `__init__` function. In particular, we need to use `@MODELS.register_module()` to add the implementation of `class MSRResNet` into the registration of MMEditing.

```
import torch.nn as nn
from mmengine.model import BaseModule
from mmedit.registry import MODELS

from mmedit.models.utils import (PixelShufflePack, ResidualBlockNoBN,
                                 default_init_weights, make_layer)
```

(continues on next page)

(continued from previous page)

```python
@MODELS.register_module()
class MSRResNet(BaseModule):
    """Modified SRResNet.

    A compacted version modified from SRResNet in "Photo-Realistic Single
    Image Super-Resolution Using a Generative Adversarial Network".

    It uses residual blocks without BN, similar to EDSR.
    Currently, it supports x2, x3 and x4 upsampling scale factor.

    Args:
        in_channels (int): Channel number of inputs.
        out_channels (int): Channel number of outputs.
        mid_channels (int): Channel number of intermediate features.
            Default: 64.
        num_blocks (int): Block number in the trunk network. Default: 16.
        upscale_factor (int): Upsampling factor. Support x2, x3 and x4.
            Default: 4.
    """
    _supported_upscale_factors = [2, 3, 4]

    def __init__(self,
                 in_channels,
                 out_channels,
                 mid_channels=64,
                 num_blocks=16,
                 upscale_factor=4):

        super().__init__()
        self.in_channels = in_channels
        self.out_channels = out_channels
        self.mid_channels = mid_channels
        self.num_blocks = num_blocks
        self.upscale_factor = upscale_factor

        self.conv_first = nn.Conv2d(
            in_channels, mid_channels, 3, 1, 1, bias=True)
        self.trunk_net = make_layer(
            ResidualBlockNoBN, num_blocks, mid_channels=mid_channels)

        # upsampling
        if self.upscale_factor in [2, 3]:
            self.upsample1 = PixelShufflePack(
                mid_channels,
                mid_channels,
                self.upscale_factor,
                upsample_kernel=3)
        elif self.upscale_factor == 4:
            self.upsample1 = PixelShufflePack(
                mid_channels, mid_channels, 2, upsample_kernel=3)
            self.upsample2 = PixelShufflePack(
```

(continues on next page)

(continued from previous page)

```
                mid_channels, mid_channels, 2, upsample_kernel=3)
        else:
            raise ValueError(
                f'Unsupported scale factor {self.upscale_factor}. '
                f'Currently supported ones are '
                f'{self._supported_upscale_factors}.')

        self.conv_hr = nn.Conv2d(
            mid_channels, mid_channels, 3, 1, 1, bias=True)
        self.conv_last = nn.Conv2d(
            mid_channels, out_channels, 3, 1, 1, bias=True)

        self.img_upsampler = nn.Upsample(
            scale_factor=self.upscale_factor,
            mode='bilinear',
            align_corners=False)

        # activation function
        self.lrelu = nn.LeakyReLU(negative_slope=0.1, inplace=True)

        self.init_weights()

    def init_weights(self):
        """Init weights for models.

        Args:
            pretrained (str, optional): Path for pretrained weights. If given
                None, pretrained weights will not be loaded. Defaults to None.
            strict (boo, optional): Whether strictly load the pretrained model.
                Defaults to True.
        """

        for m in [self.conv_first, self.conv_hr, self.conv_last]:
            default_init_weights(m, 0.1)
```

Then, we implement the `forward` function of `class MSRResNet`, which takes as input tensor and then returns the results from `MSRResNet`.

```
    def forward(self, x):
        """Forward function.

        Args:
            x (Tensor): Input tensor with shape (n, c, h, w).

        Returns:
            Tensor: Forward results.
        """

        feat = self.lrelu(self.conv_first(x))
        out = self.trunk_net(feat)
```

(continues on next page)

(continued from previous page)

```
        if self.upscale_factor in [2, 3]:
            out = self.upsample1(out)
        elif self.upscale_factor == 4:
            out = self.upsample1(out)
            out = self.upsample2(out)

        out = self.conv_last(self.lrelu(self.conv_hr(out)))
        upsampled_img = self.img_upsampler(x)
        out += upsampled_img
        return out
```

After the implementation of `class MSRResNet`, we need to update the model list in `mmedit/models/editors/__init__.py`, so that we can import and use `class MSRResNet` by `mmedit.models.editors`.

```
from .srgan.sr_resnet import MSRResNet
```

Step 2: Define the model of SRCNN

After the implementation of the network architecture, we need to define our model `class BaseEditModel` and implement the forward loop of `class BaseEditModel`.

To implement `class BaseEditModel`, we create a new file `mmedit/models/base_models/base_edit_model.py`. Specifically, `class BaseEditModel` inherits from `mmengine.model.BaseModel`. In the `__init__` function, we define the loss functions, training and testing configurations, networks of `class BaseEditModel`.

```
from typing import List, Optional

import torch
from mmengine.model import BaseModel

from mmedit.registry import MODELS
from mmedit.structures import EditDataSample, PixelData


@MODELS.register_module()
class BaseEditModel(BaseModel):
    """Base model for image and video editing.

    It must contain a generator that takes frames as inputs and outputs an
    interpolated frame. It also has a pixel-wise loss for training.

    Args:
        generator (dict): Config for the generator structure.
        pixel_loss (dict): Config for pixel-wise loss.
        train_cfg (dict): Config for training. Default: None.
        test_cfg (dict): Config for testing. Default: None.
        init_cfg (dict, optional): The weight initialized config for
            :class:`BaseModule`.
        data_preprocessor (dict, optional): The pre-process config of
            :class:`BaseDataPreprocessor`.

    Attributes:
```

(continues on next page)

(continued from previous page)

```
        init_cfg (dict, optional): Initialization config dict.
        data_preprocessor (:obj:`BaseDataPreprocessor`): Used for
            pre-processing data sampled by dataloader to the format accepted by
            :meth:`forward`. Default: None.
    """

    def __init__(self,
                 generator,
                 pixel_loss,
                 train_cfg=None,
                 test_cfg=None,
                 init_cfg=None,
                 data_preprocessor=None):
        super().__init__(
            init_cfg=init_cfg, data_preprocessor=data_preprocessor)

        self.train_cfg = train_cfg
        self.test_cfg = test_cfg

        # generator
        self.generator = MODELS.build(generator)

        # loss
        self.pixel_loss = MODELS.build(pixel_loss)
```

Since `mmengine.model.BaseModel` provides the basic functions of the algorithmic model, such as weights initialize, batch inputs preprocess, parse losses, and update model parameters. Therefore, the subclasses inherit from BaseModel, i.e., `class BaseEditModel` in this example, only need to implement the forward method, which implements the logic to calculate loss and predictions.

Specifically, the implemented `forward` function of `class BaseEditModel` takes as input `batch_inputs` and `data_samples` and return results according to mode arguments.

```
    def forward(self,
                batch_inputs: torch.Tensor,
                data_samples: Optional[List[EditDataSample]] = None,
                mode: str = 'tensor',
                **kwargs):
        """Returns losses or predictions of training, validation, testing, and
        simple inference process.

        ``forward`` method of BaseModel is an abstract method, its subclasses
        must implement this method.

        Accepts ``batch_inputs`` and ``data_samples`` processed by
        :attr:`data_preprocessor`, and returns results according to mode
        arguments.

        During non-distributed training, validation, and testing process,
        ``forward`` will be called by ``BaseModel.train_step``,
        ``BaseModel.val_step`` and ``BaseModel.val_step`` directly.

        During distributed data parallel training process,
```

(continues on next page)

(continued from previous page)

```
        ``MMSeparateDistributedDataParallel.train_step`` will first call
        ``DistributedDataParallel.forward`` to enable automatic
        gradient synchronization, and then call ``forward`` to get training
        loss.

        Args:
            batch_inputs (torch.Tensor): batch input tensor collated by
                :attr:`data_preprocessor`.
            data_samples (List[BaseDataElement], optional):
                data samples collated by :attr:`data_preprocessor`.
            mode (str): mode should be one of ``loss``, ``predict`` and
                ``tensor``

                - ``loss``: Called by ``train_step`` and return loss ``dict``
                  used for logging
                - ``predict``: Called by ``val_step`` and ``test_step``
                  and return list of ``BaseDataElement`` results used for
                  computing metric.
                - ``tensor``: Called by custom use to get ``Tensor`` type
                  results.

        Returns:
            ForwardResults:

                - If ``mode == loss``, return a ``dict`` of loss tensor used
                  for backward and logging.
                - If ``mode == predict``, return a ``list`` of
                  :obj:`BaseDataElement` for computing metric
                  and getting inference result.
                - If ``mode == tensor``, return a tensor or ``tuple`` of tensor
                  or ``dict or tensor for custom use.
        """

        if mode == 'tensor':
            return self.forward_tensor(batch_inputs, data_samples, **kwargs)

        elif mode == 'predict':
            return self.forward_inference(batch_inputs, data_samples, **kwargs)

        elif mode == 'loss':
            return self.forward_train(batch_inputs, data_samples, **kwargs)
```

Specifically, in `forward_tensor`, `class BaseEditModel` returns the forward tensors of the network directly.

```
    def forward_tensor(self, batch_inputs, data_samples=None, **kwargs):
        """Forward tensor.
            Returns result of simple forward.

        Args:
            batch_inputs (torch.Tensor): batch input tensor collated by
                :attr:`data_preprocessor`.
            data_samples (List[BaseDataElement], optional):
```

(continues on next page)

(continued from previous page)

```
            data samples collated by :attr:`data_preprocessor`.

        Returns:
            Tensor: result of simple forward.
        """

        feats = self.generator(batch_inputs, **kwargs)

        return feats
```

In `forward_inference` function, `class BaseEditModel` first converts the forward tensors to images and then returns the images as output.

```
    def forward_inference(self, batch_inputs, data_samples=None, **kwargs):
        """Forward inference.
            Returns predictions of validation, testing, and simple inference.

        Args:
            batch_inputs (torch.Tensor): batch input tensor collated by
                :attr:`data_preprocessor`.
            data_samples (List[BaseDataElement], optional):
                data samples collated by :attr:`data_preprocessor`.

        Returns:
            List[EditDataSample]: predictions.
        """

        feats = self.forward_tensor(batch_inputs, data_samples, **kwargs)
        feats = self.data_preprocessor.destructor(feats)
        predictions = []
        for idx in range(feats.shape[0]):
            predictions.append(
                EditDataSample(
                    pred_img=PixelData(data=feats[idx].to('cpu')),
                    metainfo=data_samples[idx].metainfo))

        return predictions
```

In `forward_train`, `class BaseEditModel` calculate the loss function and returns a dictionary contains the losses as output.

```
    def forward_train(self, batch_inputs, data_samples=None, **kwargs):
        """Forward training.
            Returns dict of losses of training.

        Args:
            batch_inputs (torch.Tensor): batch input tensor collated by
                :attr:`data_preprocessor`.
            data_samples (List[BaseDataElement], optional):
                data samples collated by :attr:`data_preprocessor`.

        Returns:
```

(continues on next page)

(continued from previous page)

```
            dict: Dict of losses.
        """

        feats = self.forward_tensor(batch_inputs, data_samples, **kwargs)
        gt_imgs = [data_sample.gt_img.data for data_sample in data_samples]
        batch_gt_data = torch.stack(gt_imgs)

        loss = self.pixel_loss(feats, batch_gt_data)

        return dict(loss=loss)
```

After the implementation of `class BaseEditModel`, we need to update the model list in `mmedit/models/__init__.py`, so that we can import and use `class BaseEditModel` by `mmedit.models`.

```
from .base_models.base_edit_model import BaseEditModel
```

Step 3: Start training SRCNN

After implementing the network architecture and the forward loop of SRCNN, now we can create a new file `configs/srcnn/srcnn_x4k915_g1_1000k_div2k.py` to set the configurations needed by training SRCNN.

In the configuration file, we need to specify the parameters of our model, `class BaseEditModel`, including the generator network architecture, loss function, additional training and testing configuration, and data preprocessor of input tensors. Please refer to the *Introduction to the loss in MMEditing* for more details of losses in MMEditing.

```
# model settings
model = dict(
    type='BaseEditModel',
    generator=dict(
        type='SRCNNNet',
        channels=(3, 64, 32, 3),
        kernel_sizes=(9, 1, 5),
        upscale_factor=scale),
    pixel_loss=dict(type='L1Loss', loss_weight=1.0, reduction='mean'),
    data_preprocessor=dict(
        type='EditDataPreprocessor',
        mean=[0., 0., 0.],
        std=[255., 255., 255.],
    ))
```

We also need to specify the training dataloader and testing dataloader according to *create your own dataloader*. Finally we can start training our own model by

```
python train.py configs/srcnn/srcnn_x4k915_g1_1000k_div2k.py
```

1.11.3 An example of DCGAN

Here, we take the implementation of the classical gan model, DCGAN [2], as an example.

Step 1: Define the network of DCGAN

DCGAN is a classical image generative adversarial network [2]. To implement the network architecture of DCGAN, we need to create tow new files `mmedit/models/editors/dcgan/dcgan_generator.py` and `mmedit/models/editors/dcgan/dcgan_discriminator.py`, and implement generator (`class DCGANGenerator`) and discriminator (`class DCGANDiscriminator`).

In this step, we implement `class DCGANGenerator`, `class DCGANDiscriminator` and define the network architecture in `__init__` function. In particular, we need to use `@MODULES.register_module()` to add the generator and discriminator into the registration of MMEditing.

Take the following code as example:

```
import torch.nn as nn
from mmcv.cnn import ConvModule
from mmcv.runner import load_checkpoint
from mmcv.utils.parrots_wrapper import _BatchNorm
from mmengine.logging import MMLogger
from mmengine.model.utils import normal_init

from mmedit.models.builder import MODULES
from ..common import get_module_device


@MODULES.register_module()
class DCGANGenerator(nn.Module):
    def __init__(self,
                 output_scale,
                 out_channels=3,
                 base_channels=1024,
                 input_scale=4,
                 noise_size=100,
                 default_norm_cfg=dict(type='BN'),
                 default_act_cfg=dict(type='ReLU'),
                 out_act_cfg=dict(type='Tanh'),
                 pretrained=None):
        super().__init__()
        self.output_scale = output_scale
        self.base_channels = base_channels
        self.input_scale = input_scale
        self.noise_size = noise_size

        # the number of times for upsampling
        self.num_upsamples = int(np.log2(output_scale // input_scale))

        # output 4x4 feature map
        self.noise2feat = ConvModule(
            noise_size,
            base_channels,
            kernel_size=4,
```

(continues on next page)

(continued from previous page)

```
            stride=1,
            padding=0,
            conv_cfg=dict(type='ConvTranspose2d'),
            norm_cfg=default_norm_cfg,
            act_cfg=default_act_cfg)

        # build up upsampling backbone (excluding the output layer)
        upsampling = []
        curr_channel = base_channels
        for _ in range(self.num_upsamples - 1):
            upsampling.append(
                ConvModule(
                    curr_channel,
                    curr_channel // 2,
                    kernel_size=4,
                    stride=2,
                    padding=1,
                    conv_cfg=dict(type='ConvTranspose2d'),
                    norm_cfg=default_norm_cfg,
                    act_cfg=default_act_cfg))

            curr_channel //= 2

        self.upsampling = nn.Sequential(*upsampling)

        # output layer
        self.output_layer = ConvModule(
            curr_channel,
            out_channels,
            kernel_size=4,
            stride=2,
            padding=1,
            conv_cfg=dict(type='ConvTranspose2d'),
            norm_cfg=None,
            act_cfg=out_act_cfg)
```

Then, we implement the `forward` function of `DCGANGenerator`, which takes as `noise` tensor or `num_batches` and then returns the results from `DCGANGenerator`.

```
    def forward(self, noise, num_batches=0, return_noise=False):
        noise_batch = noise_batch.to(get_module_device(self))
        x = self.noise2feat(noise_batch)
        x = self.upsampling(x)
        x = self.output_layer(x)
        return x
```

If you want to implement specific weights initialization method for you network, you need add `init_weights` function by yourself.

```
    def init_weights(self, pretrained=None):
        if isinstance(pretrained, str):
            logger = MMLogger.get_current_instance()
```

(continues on next page)

(continued from previous page)

```
            load_checkpoint(self, pretrained, strict=False, logger=logger)
        elif pretrained is None:
            for m in self.modules():
                if isinstance(m, (nn.Conv2d, nn.ConvTranspose2d)):
                    normal_init(m, 0, 0.02)
                elif isinstance(m, _BatchNorm):
                    nn.init.normal_(m.weight.data)
                    nn.init.constant_(m.bias.data, 0)
        else:
            raise TypeError('pretrained must be a str or None but'
                            f' got {type(pretrained)} instead.')
```

After the implementation of class `DCGANGenerator`, we need to update the model list in `mmedit/models/editors/__init__.py`, so that we can import and use class `DCGANGenerator` by `mmedit.models.editors`.

Implementation of Class `DCGANDiscriminator` follows the similar logic, and you can find the implementation here.

Step 2: Design the model of DCGAN

After the implementation of the network **Module**, we need to define our **Model** class `DCGAN`.

Your **Model** should inherit from `BaseModel` provided by MMEngine and implement three functions, `train_step`, `val_step` and `test_step`.

- `train_step`: This function is responsible to update the parameters of the network and called by MMEngine's Loop (`IterBasedTrainLoop` or `EpochBasedTrainLoop`). `train_step` take data batch and `OptimWrapper` as input and return a dict of log.
- `val_step`: This function is responsible for getting output for validation during the training process. and is called by `GenValLoop`.
- `test_step`: This function is responsible for getting output in test process and is called by `GenTestLoop`.

  Note that, in `train_step`, `val_step` and `test_step`, `DataPreprocessor` is called to preprocess the input data batch before feed them to the neural network. To know more about `DataPreprocessor` please refer to this file and this tutorial.

For simplify using, we provide `BaseGAN` class in MMEditing, which implements generic `train_step`, `val_step` and `test_step` function for GAN models. With `BaseGAN` as base class, each specific GAN algorithm only need to implement `train_generator` and `train_discriminator`.

In `train_step`, we support data preprocessing, gradient accumulation (realized by `OptimWrapper`) and expontial moving averate (EMA) realized by (`ExponentialMovingAverage`). With `BaseGAN.train_step`, each specific GAN algorithm only need to implement `train_generator` and `train_discriminator`.

```
    def train_step(self, data: dict,
                   optim_wrapper: OptimWrapperDict) -> Dict[str, Tensor]:
        message_hub = MessageHub.get_current_instance()
        curr_iter = message_hub.get_info('iter')
        data = self.data_preprocessor(data, True)
        disc_optimizer_wrapper: OptimWrapper = optim_wrapper['discriminator']
        disc_accu_iters = disc_optimizer_wrapper._accumulative_counts

        # train discriminator, use context manager provided by MMEngine
        with disc_optimizer_wrapper.optim_context(self.discriminator):
```

(continues on next page)

(continued from previous page)

```
            # train_discriminator should be implemented!
            log_vars = self.train_discriminator(
                **data, optimizer_wrapper=disc_optimizer_wrapper)

        # add 1 to `curr_iter` because iter is updated in train loop.
        # Whether to update the generator. We update generator with
        # discriminator is fully updated for `self.n_discriminator_steps`
        # iterations. And one full updating for discriminator contains
        # `disc_accu_counts` times of grad accumulations.
        if (curr_iter + 1) % (self.discriminator_steps * disc_accu_iters) == 0:
            set_requires_grad(self.discriminator, False)
            gen_optimizer_wrapper = optim_wrapper['generator']
            gen_accu_iters = gen_optimizer_wrapper._accumulative_counts

            log_vars_gen_list = []
            # init optimizer wrapper status for generator manually
            gen_optimizer_wrapper.initialize_count_status(
                self.generator, 0, self.generator_steps * gen_accu_iters)
            # update generator, use context manager provided by MMEngine
            for _ in range(self.generator_steps * gen_accu_iters):
                with gen_optimizer_wrapper.optim_context(self.generator):
                    # train_generator should be implemented!
                    log_vars_gen = self.train_generator(
                        **data, optimizer_wrapper=gen_optimizer_wrapper)

                log_vars_gen_list.append(log_vars_gen)
            log_vars_gen = gather_log_vars(log_vars_gen_list)
            log_vars_gen.pop('loss', None)  # remove 'loss' from gen logs

            set_requires_grad(self.discriminator, True)

            # only do ema after generator update
            if self.with_ema_gen and (curr_iter + 1) >= (
                    self.ema_start * self.discriminator_steps *
                    disc_accu_iters):
                self.generator_ema.update_parameters(
                    self.generator.module
                    if is_model_wrapper(self.generator) else self.generator)

            log_vars.update(log_vars_gen)

        # return the log dict
        return log_vars
```

In `val_step` and `test_step`, we call data preprocessing and `BaseGAN.forward` progressively.

```
    def val_step(self, data: dict) -> SampleList:
        data = self.data_preprocessor(data)
        # call `forward`
        outputs = self(**data)
        return outputs
```

(continues on next page)

(continued from previous page)

```
    def test_step(self, data: dict) -> SampleList:
        data = self.data_preprocessor(data)
        # call `orward`
        outputs = self(**data)
        return outputs
```

Then, we implement `train_generator` and `train_discriminator` in `DCGAN` class.

```
from typing import Dict, Tuple

import torch
import torch.nn.functional as F
from mmengine.optim import OptimWrapper
from torch import Tensor

from mmedit.registry import MODELS
from .base_gan import BaseGAN


@MODELS.register_module()
class DCGAN(BaseGAN):
    def disc_loss(self, disc_pred_fake: Tensor,
                  disc_pred_real: Tensor) -> Tuple:
        losses_dict = dict()
        losses_dict['loss_disc_fake'] = F.binary_cross_entropy_with_logits(
            disc_pred_fake, 0. * torch.ones_like(disc_pred_fake))
        losses_dict['loss_disc_real'] = F.binary_cross_entropy_with_logits(
            disc_pred_real, 1. * torch.ones_like(disc_pred_real))

        loss, log_var = self.parse_losses(losses_dict)
        return loss, log_var

    def gen_loss(self, disc_pred_fake: Tensor) -> Tuple:
        losses_dict = dict()
        losses_dict['loss_gen'] = F.binary_cross_entropy_with_logits(
            disc_pred_fake, 1. * torch.ones_like(disc_pred_fake))
        loss, log_var = self.parse_losses(losses_dict)
        return loss, log_var

    def train_discriminator(
            self, inputs, data_sample,
            optimizer_wrapper: OptimWrapper) -> Dict[str, Tensor]:
        real_imgs = inputs['img']

        num_batches = real_imgs.shape[0]

        noise_batch = self.noise_fn(num_batches=num_batches)
        with torch.no_grad():
            fake_imgs = self.generator(noise=noise_batch, return_noise=False)

        disc_pred_fake = self.discriminator(fake_imgs)
        disc_pred_real = self.discriminator(real_imgs)
```

(continues on next page)

(continued from previous page)

```
        parsed_losses, log_vars = self.disc_loss(disc_pred_fake,
                                                 disc_pred_real)
        optimizer_wrapper.update_params(parsed_losses)
        return log_vars

    def train_generator(self, inputs, data_sample,
                        optimizer_wrapper: OptimWrapper) -> Dict[str, Tensor]:
        num_batches = inputs['img'].shape[0]

        noise = self.noise_fn(num_batches=num_batches)
        fake_imgs = self.generator(noise=noise, return_noise=False)

        disc_pred_fake = self.discriminator(fake_imgs)
        parsed_loss, log_vars = self.gen_loss(disc_pred_fake)

        optimizer_wrapper.update_params(parsed_loss)
        return log_vars
```

After the implementation of `class DCGAN`, we need to update the model list in `mmedit/models/__init__.py`, so that we can import and use `class DCGAN` by `mmedit.models`.

Step 3: Start training DCGAN

After implementing the network **Module** and the **Model** of DCGAN, now we can create a new file `configs/dcgan/dcgan_1xb128-5epoches_lsun-bedroom-64x64.py` to set the configurations needed by training DCGAN.

In the configuration file, we need to specify the parameters of our model, `class DCGAN`, including the generator network architecture and data preprocessor of input tensors.

```
# model settings
model = dict(
    type='DCGAN',
    noise_size=100,
    data_preprocessor=dict(type='GANDataPreprocessor'),
    generator=dict(type='DCGANGenerator', output_scale=64, base_channels=1024),
    discriminator=dict(
        type='DCGANDiscriminator',
        input_scale=64,
        output_scale=4,
        out_channels=1))
```

We also need to specify the training dataloader and testing dataloader according to *create your own dataloader*. Finally we can start training our own model by

```
python train.py configs/dcgan/dcgan_1xb128-5epoches_lsun-bedroom-64x64.py
```

1.11.4 References

1. Dong, Chao and Loy, Chen Change and He, Kaiming and Tang, Xiaoou. Image Super-Resolution Using Deep Convolutional Networks[J]. IEEE transactions on pattern analysis and machine intelligence, 2015.
2. Radford, Alec, Luke Metz, and Soumith Chintala. "Unsupervised representation learning with deep convolutional generative adversarial networks." arXiv preprint arXiv:1511.06434 (2015).

1.12 Prepare Your Own Datasets

In this document, we will introduce the design of each datasets in MMEditing and how users can design their own dataset.

1.12.1 Supported Data Format

In 1.x version of MMEditing, all datasets are inherited from `BaseDataset`. Each dataset load the list of data info (e.g., data path) by `load_data_list`. In `__getitem__`, `prepare_data` is called to get the preprocessed data. In `prepare_data`, data loading pipeline consists of the following steps:

1. fetch the data info by passed index, implemented by `get_data_info`
2. apply data transforms to the data, implemented by `pipeline`

BasicImageDataset

BasicImageDataset `mmedit.datasets.BasicImageDataset` General image dataset designed for low-level vision tasks with image, such as image super-resolution, inpainting and unconditional image generation. The annotation file is optional.

If use annotation file, the annotation format can be shown as follows.

```
  Case 1 (CelebA-HQ):

      000001.png
      000002.png

  Case 2 (DIV2K):

      0001_s001.png (480,480,3)
      0001_s002.png (480,480,3)
      0001_s003.png (480,480,3)
      0002_s001.png (480,480,3)
      0002_s002.png (480,480,3)

  Case 3 (Vimeo90k):

      00001/0266 (256, 448, 3)
      00001/0268 (256, 448, 3)
```

Here we give several examples showing how to use `BasicImageDataset`. Assume the file structure as the following:

```
mmediting (root)
├── mmedit
├── tools
├── configs
├── data
    ├── DIV2K
        ├── DIV2K_train_HR
            ├── image.png
        ├── DIV2K_train_LR_bicubic
            ├── X2
            ├── X3
            ├── X4
                ├── image_x4.png
        ├── DIV2K_valid_HR
        ├── DIV2K_valid_LR_bicubic
            ├── X2
            ├── X3
            ├── X4
    ├── places
        ├── test_set
        ├── train_set
    |   ├── meta
    |   |   ├── Places365_train.txt
    |   |   ├── Places365_val.txt
    |   ├── celebahq
    |   |   ├── imgs_1024
```

Case 1: Loading DIV2K dataset for training a SISR model.

```
   dataset = BasicImageDataset(
       ann_file='',
       metainfo=dict(
           dataset_type='div2k',
           task_name='sisr'),
       data_root='data/DIV2K',
       data_prefix=dict(
           gt='DIV2K_train_HR', img='DIV2K_train_LR_bicubic/X4'),
       filename_tmpl=dict(img='{}_x4', gt='{}'),
       pipeline=[])
```

Case 2: Loading places dataset for training an inpainting model.

```
   dataset = BasicImageDataset(
       ann_file='meta/Places365_train.txt',
       metainfo=dict(
           dataset_type='places365',
           task_name='inpainting'),
       data_root='data/places',
       data_prefix=dict(gt='train_set'),
       pipeline=[])
```

Case 3: Loading CelebA-HQ dataset for training an PGGAN.

```
dataset = BasicImageDataset(
        pipeline=[],
        data_root='./data/celebahq/imgs_1024')
```

BasicFramesDataset

BasicFramesDataset `mmedit.datasets.BasicFramesDataset` General frames dataset designed for low-level vision tasks with frames, such as video super-resolution and video frame interpolation. The annotation file is optional.

If use annotation file, the annotation format can be shown as follows.

```
Case 1 (Vid4):

   calendar 41
   city 34
   foliage 49
   walk 47

Case 2 (REDS):

   000/00000000.png (720, 1280, 3)
   000/00000001.png (720, 1280, 3)

Case 3 (Vimeo90k):

   00001/0266 (256, 448, 3)
   00001/0268 (256, 448, 3)
```

Assume the file structure as the following:

```
mmediting (root)
├── mmedit
├── tools
├── configs
├── data
    ├── Vid4
        ├── BIx4
            ├── city
                ├── img1.png
        ├── GT
            ├── city
                ├── img1.png
        ├── meta_info_Vid4_GT.txt
    ├── places
        ├── sequences
            ├── 00001
                ├── 0389
                    ├── img1.png
                    ├── img2.png
                    ├── img3.png
        ├── tri_trainlist.txt
```

Case 1: Loading Vid4 dataset for training a VSR model.

```
dataset = BasicFramesDataset(
    ann_file='meta_info_Vid4_GT.txt',
    metainfo=dict(dataset_type='vid4', task_name='vsr'),
    data_root='data/Vid4',
    data_prefix=dict(img='BIx4', gt='GT'),
    pipeline=[],
    depth=2,
    num_input_frames=5)
```

Case 2: Loading Vimeo90k dataset for training a VFI model.

```
dataset = BasicFramesDataset(
    ann_file='tri_trainlist.txt',
    metainfo=dict(dataset_type='vimeo90k', task_name='vfi'),
    data_root='data/vimeo-triplet',
    data_prefix=dict(img='sequences', gt='sequences'),
    pipeline=[],
    depth=2,
    load_frames_list=dict(
        img=['img1.png', 'img3.png'], gt=['img2.png']))
```

AdobeComp1kDataset

AdobeComp1kDataset `mmedit.datasets.AdobeComp1kDataset` Adobe composition-1k dataset.

The dataset loads (alpha, fg, bg) data and apply specified transforms to the data. You could specify whether composite merged image online or load composited merged image in pipeline.

Example for online comp-1k dataset:

```
[
   {
       "alpha": 'alpha/000.png',
       "fg": 'fg/000.png',
       "bg": 'bg/000.png'
   },
   {
       "alpha": 'alpha/001.png',
       "fg": 'fg/001.png',
       "bg": 'bg/001.png'
   },
]
```

Example for offline comp-1k dataset:

```
[
  {
      "alpha": 'alpha/000.png',
      "merged": 'merged/000.png',
      "fg": 'fg/000.png',
      "bg": 'bg/000.png'
  },
  {
      "alpha": 'alpha/001.png',
      "merged": 'merged/001.png',
      "fg": 'fg/001.png',
      "bg": 'bg/001.png'
  },
]
```

GrowScaleImgDataset

`GrowScaleImgDataset` is designed for dynamic GAN models (e.g., PGGAN and StyleGANv1). In this dataset, we support switching the data root during training to load training images of different resolutions. This procedure is implemented by `GrowScaleImgDataset.update_annotations` and is called by `PGGANFetchDataHook.before_train_iter` in the training process.

```
def update_annotations(self, curr_scale):
    # determine if the data root needs to be updated
    if curr_scale == self._actual_curr_scale:
        return False

    # fetch new data root by resolution (scale)
    for scale in self._img_scales:
        if curr_scale <= scale:
```

(continues on next page)

(continued from previous page)

```
            self._curr_scale = scale
            break
        if scale == self._img_scales[-1]:
            assert RuntimeError(
                f'Cannot find a suitable scale for {curr_scale}')
    self._actual_curr_scale = curr_scale
    self.data_root = self.data_roots[str(self._curr_scale)]

    # reload the data list with new data root
    self.load_data_list()

    # print basic dataset information to check the validity
    print_log('Update Dataset: ' + repr(self), 'current')
    return True
```

SinGANDataset

`SinGANDataset` is designed for SinGAN's training. In SinGAN's training, we do not iterate the images in the dataset but return a consistent preprocessed image dict.

Therefore, we bypass the default data loading logic of `BaseDataset` because we do not need to load the corresponding image data based on the given index.

```
def load_data_list(self, min_size, max_size, scale_factor_init):
    # load single image
    real = mmcv.imread(self.data_root)
    self.reals, self.scale_factor, self.stop_scale = create_real_pyramid(
        real, min_size, max_size, scale_factor_init)

    self.data_dict = {}

    # generate multi scale image
    for i, real in enumerate(self.reals):
        self.data_dict[f'real_scale{i}'] = real

    self.data_dict['input_sample'] = np.zeros_like(
        self.data_dict['real_scale0']).astype(np.float32)

def __getitem__(self, index):
    # directly return the transformed data dict
    return self.pipeline(self.data_dict)
```

PairedImageDataset

PairedImageDataset is designed for translation models that needs paried training data (e.g., Pix2Pix). The directory structure is shown below. Each image files are the concatenation of the image pair.

```
./data/dataset_name/
├── test
│   └── XXX.jpg
└── train
    └── XXX.jpg
```

In PairedImageDataset, we scan the file list in load_data_list and save path in pair_path field to fit the LoadPairedImageFromFile transformation.

```
def load_data_list(self):
    data_infos = []
    pair_paths = sorted(self.scan_folder(self.data_root))
    for pair_path in pair_paths:
        # save path in the specific field
        data_infos.append(dict(pair_path=pair_path))

    return data_infos
```

UnpairedImageDataset

UnpairedImageDataset is designed for translation models that do not need paired data (e.g., CycleGAN). The directory structure is shown below.

```
./data/dataset_name/
├── testA
│   └── XXX.jpg
├── testB
│   └── XXX.jpg
├── trainA
│   └── XXX.jpg
└── trainB
    └── XXX.jpg
```

In this dataset, we overwrite __getitem__ function to load random image pair in the training process.

```
def __getitem__(self, idx):
    if not self.test_mode:
        return self.prepare_train_data(idx)

    return self.prepare_test_data(idx)

def prepare_train_data(self, idx):
    img_a_path = self.data_infos_a[idx % self.len_a]['path']
    idx_b = np.random.randint(0, self.len_b)
    img_b_path = self.data_infos_b[idx_b]['path']
    results = dict()
    results[f'img_{self.domain_a}_path'] = img_a_path
```

(continues on next page)

(continued from previous page)

```
    results[f'img_{self.domain_b}_path'] = img_b_path
    return self.pipeline(results)

def prepare_test_data(self, idx):
    img_a_path = self.data_infos_a[idx % self.len_a]['path']
    img_b_path = self.data_infos_b[idx % self.len_b]['path']
    results = dict()
    results[f'img_{self.domain_a}_path'] = img_a_path
    results[f'img_{self.domain_b}_path'] = img_b_path
    return self.pipeline(results)
```

1.12.2 Design a new dataset

If you want to create a dataset for a new low level CV task (e.g. denoise, derain, defog, and de-reflection) or existing dataset format doesn't meet your need, you can reorganize new data formats to existing format.

Or create a new dataset in `mmedit/datasets` to load the data.

Inheriting from the base class of datasets such as `BasicImageDataset` and `BasicFramesDataset` will make it easier to create a new dataset.

And you can create a new dataset inherited from BaseDataset which is the base class of datasets in MMEngine.

Here is an example of creating a dataset for video frame interpolation:

```
from .basic_frames_dataset import BasicFramesDataset
from mmedit.registry import DATASETS


@DATASETS.register_module()
class NewVFIDataset(BasicFramesDataset):
    """Introduce the dataset

    Examples of file structure.

    Args:
        pipeline (list[dict | callable]): A sequence of data transformations.
        folder (str | :obj:`Path`): Path to the folder.
        ann_file (str | :obj:`Path`): Path to the annotation file.
        test_mode (bool): Store `True` when building test dataset.
            Default: `False`.
    """

    def __init__(self, ann_file, metainfo, data_root, data_prefix,
                 pipeline, test_mode=False):
        super().__init__(ann_file, metainfo, data_root, data_prefix,
                         pipeline, test_mode)
        self.data_infos = self.load_annotations()

    def load_annotations(self):
        """Load annoations for the dataset.

        Returns:
```

(continues on next page)

(continued from previous page)

```
            list[dict]: A list of dicts for paired paths and other information.
        """
        data_infos = []
        ...
        return data_infos
```

Welcome to submit new dataset classes to MMEditing.

Repeat dataset

We use RepeatDataset as wrapper to repeat the dataset. For example, suppose the original dataset is Dataset_A, to repeat it, the config looks like the following

```
dataset_A_train = dict(
        type='RepeatDataset',
        times=N,
        dataset=dict(  # This is the original config of Dataset_A
            type='Dataset_A',
            ...
            pipeline=train_pipeline
        )
    )
```

You may refer to tutorial in MMEngine.

1.13 Design Your Own Data Pipelines

In this tutorial, we introduce the design of transforms pipeline in MMEditing.

The structure of this guide are as follows:

1.13.1 Data pipelines in MMEditing

Following typical conventions, we use `Dataset` and `DataLoader` for data loading with multiple workers. `Dataset` returns a dict of data items corresponding the arguments of models' forward method.

The data preparation pipeline and the dataset is decomposed. Usually a dataset defines how to process the annotations and a data pipeline defines all the steps to prepare a data dict.

A pipeline consists of a sequence of operations. Each operation takes a dict as input and also output a dict for the next transform.

The operations are categorized into data loading, pre-processing, and formatting

In 1.x version of MMEditing, all data transformations are inherited from `BaseTransform`. The input and output types of transformations are both dict.

A simple example of data transform

```
>>> from mmgen.transforms import LoadPairedImageFromFile
>>> transforms = LoadPairedImageFromFile(
>>>     key='pair',
>>>     domain_a='horse',
>>>     domain_b='zebra',
>>>     flag='color'),
>>> data_dict = {'pair_path': './data/pix2pix/facades/train/1.png'}
>>> data_dict = transforms(data_dict)
>>> print(data_dict.keys())
dict_keys(['pair_path', 'pair', 'pair_ori_shape', 'img_mask', 'img_photo', 'img_mask_path
↪', 'img_photo_path', 'img_mask_ori_shape', 'img_photo_ori_shape'])
```

Generally, the last step of the transforms pipeline must be `PackEditInputs`. `PackEditInputs` will pack the processed data into a dict containing two fields: `inputs` and `data_samples`. `inputs` is the variable you want to use as the model's input, which can be the type of `torch.Tensor`, dict of `torch.Tensor`, or any type you want. `data_samples` is a list of `EditDataSample`. Each `EditDataSample` contains groundtruth and necessary information for corresponding input.

An example of BasicVSR

Here is a pipeline example for BasicVSR.

```
train_pipeline = [
    dict(type='LoadImageFromFile', key='img', channel_order='rgb'),
    dict(type='LoadImageFromFile', key='gt', channel_order='rgb'),
    dict(type='SetValues', dictionary=dict(scale=scale)),
    dict(type='PairedRandomCrop', gt_patch_size=256),
    dict(
        type='Flip',
        keys=['img', 'gt'],
        flip_ratio=0.5,
        direction='horizontal'),
    dict(
        type='Flip', keys=['img', 'gt'], flip_ratio=0.5, direction='vertical'),
    dict(type='RandomTransposeHW', keys=['img', 'gt'], transpose_ratio=0.5),
    dict(type='MirrorSequence', keys=['img', 'gt']),
```

(continues on next page)

(continued from previous page)

```
    dict(type='PackEditInputs')
]

val_pipeline = [
    dict(type='GenerateSegmentIndices', interval_list=[1]),
    dict(type='LoadImageFromFile', key='img', channel_order='rgb'),
    dict(type='LoadImageFromFile', key='gt', channel_order='rgb'),
    dict(type='PackEditInputs')
]

test_pipeline = [
    dict(type='LoadImageFromFile', key='img', channel_order='rgb'),
    dict(type='LoadImageFromFile', key='gt', channel_order='rgb'),
    dict(type='MirrorSequence', keys=['img']),
    dict(type='PackEditInputs')
]
```

For each operation, we list the related dict fields that are added/updated/removed, the dict fields marked by '*' are optional.

An example of Pix2Pix

Here is a pipeline example for Pix2Pix training on aerial2maps dataset.

```
source_domain = 'aerial'
target_domain = 'map'

pipeline = [
    dict(
        type='LoadPairedImageFromFile',
        io_backend='disk',
        key='pair',
        domain_a=domain_a,
        domain_b=domain_b,
        flag='color'),
    dict(
        type='TransformBroadcaster',
        mapping={'img': [f'img_{domain_a}', f'img_{domain_b}']},
        auto_remap=True,
        share_random_params=True,
        transforms=[
            dict(
                type='mmgen.Resize', scale=(286, 286),
                interpolation='bicubic'),
            dict(type='mmgen.FixedCrop', crop_size=(256, 256))
        ]),
    dict(
        type='Flip',
        keys=[f'img_{domain_a}', f'img_{domain_b}'],
        direction='horizontal'),
    dict(
```

(continues on next page)

(continued from previous page)

```
        type='PackEditInputs',
        keys=[f'img_{domain_a}', f'img_{domain_b}', 'pair'])
```

1.13.2 Supported transforms in MMEditing

Data loading

Pre-processing

Formatting

1.13.3 Extend and use custom pipelines

A simple example of MyTransform

1. Write a new pipeline in a file, e.g., in `my_pipeline.py`. It takes a dict as input and returns a dict.

```
import random
from mmcv.transforms import BaseTransform
from mmedit.registry import TRANSFORMS


@TRANSFORMS.register_module()
class MyTransform(BaseTransform):
    """Add your transform

    Args:
        p (float): Probability of shifts. Default 0.5.
    """

    def __init__(self, p=0.5):
        self.p = p

    def transform(self, results):
        if random.random() > self.p:
            results['dummy'] = True
        return results

    def __repr__(self):

        repr_str = self.__class__.__name__
        repr_str += (f'(p={self.p})')

        return repr_str
```

2. Import and use the pipeline in your config file.

Make sure the import is relative to where your train script is located.

```
train_pipeline = [
    ...
```

(continues on next page)

(continued from previous page)

```
    dict(type='MyTransform', p=0.2),
    ...
]
```

An example of flipping

Here we use a simple flipping transformation as example:

```
import random
import mmcv
from mmcv.transforms import BaseTransform, TRANSFORMS

@TRANSFORMS.register_module()
class MyFlip(BaseTransform):
    def __init__(self, direction: str):
        super().__init__()
        self.direction = direction

    def transform(self, results: dict) -> dict:
        img = results['img']
        results['img'] = mmcv.imflip(img, direction=self.direction)
        return results
```

Thus, we can instantiate a `MyFlip` object and use it to process the data dict.

```
import numpy as np

transform = MyFlip(direction='horizontal')
data_dict = {'img': np.random.rand(224, 224, 3)}
data_dict = transform(data_dict)
processed_img = data_dict['img']
```

Or, we can use `MyFlip` transformation in data pipeline in our config file.

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

Note that if you want to use `MyFlip` in config, you must ensure the file containing `MyFlip` is imported during the program run.

1.14 Design Your Own Loss Functions

`losses` are registered as `LOSSES` in `MMEditing`. Customizing losses is similar to customizing any other model. This section is mainly for clarifying the design of loss modules in MMEditing. Importantly, when writing your own loss modules, you should follow the same design, so that the new loss module can be adopted in our framework without extra effort.

This guides includes:

1.14.1 Introduction to supported losses

For convenient usage, you can directly use default loss calculation process we set for concrete algorithms like lsgan, biggan, styleganv2 etc. Take `stylegan2` as an example, we use R1 gradient penalty and generator path length regularization as configurable losses, and users can adjust related arguments like `r1_loss_weight` and `g_reg_weight`.

```
# stylegan2_base.py
loss_config = dict(
    r1_loss_weight=10. / 2. * d_reg_interval,
    r1_interval=d_reg_interval,
    norm_mode='HWC',
    g_reg_interval=g_reg_interval,
    g_reg_weight=2. * g_reg_interval,
    pl_batch_shrink=2)

model = dict(
    type='StyleGAN2',
    xxx,
    loss_config=loss_config)
```

1.14.2 Design a new loss function

An example of MSELoss

In general, to implement a loss module, we will write a function implementation and then wrap it with a class implementation. Take the MSELoss as an example:

```
@masked_loss
def mse_loss(pred, target):
    return F.mse_loss(pred, target, reduction='none')

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

    def __init__(self, loss_weight=1.0, reduction='mean', sample_wise=False):
        # codes can be found in ``mmedit/models/losses/pixelwise_loss.py``

    def forward(self, pred, target, weight=None, **kwargs):
        # codes can be found in ``mmedit/models/losses/pixelwise_loss.py``
```

Given the definition of the loss, we can now use the loss by simply defining it in the configuration file:

```
pixel_loss=dict(type='MSELoss', loss_weight=1.0, reduction='mean')
```

Note that `pixel_loss` above must be defined in the model. Please refer to `customize_models` for more details. Similar to model customization, in order to use your customized loss, you need to import the loss in `mmedit/models/losses/__init__.py` after writing it.

An example of DiscShiftLoss

In general, to implement a loss module, we will write a function implementation and then wrap it with a class implementation. However, in `MMEditing`, we provide another unified interface `data_info` for users to define the mapping between the input argument and data items.

```
@weighted_loss
def disc_shift_loss(pred):
    return pred**2

@MODULES.register_module()
class DiscShiftLoss(nn.Module):

    def __init__(self, loss_weight=1.0, data_info=None):
        super(DiscShiftLoss, self).__init__()
        # codes can be found in ``mmgen/models/losses/disc_auxiliary_loss.py``

    def forward(self, *args, **kwargs):
        # codes can be found in ``mmgen/models/losses/disc_auxiliary_loss.py``
```

The goal of this design for loss modules is to allow for using it automatically in the generative models (`MODELS`), without other complex codes to define the mapping between data and keyword arguments. Thus, different from other frameworks in `OpenMMLab`, our loss modules contain a special keyword, `data_info`, which is a dictionary defining the mapping between the input arguments and data from the generative models. Taking the `DiscShiftLoss` as an example, when writing the config file, users may use this loss as follows:

```
dict(type='DiscShiftLoss',
    loss_weight=0.001 * 0.5,
    data_info=dict(pred='disc_pred_real')
```

The information in `data_info` tells the module to use the `disc_pred_real` data as the input tensor for `pred` arguments. Once the `data_info` is not `None`, our loss module will automatically build up the computational graph.

```
@MODULES.register_module()
class DiscShiftLoss(nn.Module):

    def __init__(self, loss_weight=1.0, data_info=None):
        super(DiscShiftLoss, self).__init__()
        self.loss_weight = loss_weight
        self.data_info = data_info

    def forward(self, *args, **kwargs):
        # use data_info to build computational path
        if self.data_info is not None:
            # parse the args and kwargs
            if len(args) == 1:
                assert isinstance(args[0], dict), (
                    'You should offer a dictionary containing network outputs '
                    'for building up computational graph of this loss module.')
                outputs_dict = args[0]
            elif 'outputs_dict' in kwargs:
                assert len(args) == 0, (
                    'If the outputs dict is given in keyworded arguments, no'
                    ' further non-keyworded arguments should be offered.')
                outputs_dict = kwargs.pop('outputs_dict')
            else:
                raise NotImplementedError(
                    'Cannot parsing your arguments passed to this loss module.'
                    ' Please check the usage of this module')
            # link the outputs with loss input args according to self.data_info
            loss_input_dict = {
                k: outputs_dict[v]
                for k, v in self.data_info.items()
            }
            kwargs.update(loss_input_dict)
            kwargs.update(dict(weight=self.loss_weight))
            return disc_shift_loss(**kwargs)
        else:
            # if you have not define how to build computational graph, this
            # module will just directly return the loss as usual.
            return disc_shift_loss(*args, weight=self.loss_weight, **kwargs)

    @staticmethod
    def loss_name():
        return 'loss_disc_shift'
```

As shown in this part of codes, once users set the `data_info`, the loss module will receive a dictionary containing all of the necessary data and modules, which is provided by the `MODELS` in the training procedure. If this dictionary is given as a non-keyword argument, it should be offered as the first argument. If you are using a keyword argument,

please name it as `outputs_dict`.

An example of GANWithCustomizedLoss

To build the computational graph, the generative models have to provide a dictionary containing all kinds of data. Having a close look at any generative model, you will find that we collect all kinds of features and modules into a dictionary. We provide a customized `GANWithCustomizedLoss` here to show the process.

```
class GANWithCustomizedLoss(BaseModel):

    def __init__(self, gan_loss, disc_auxiliary_loss, gen_auxiliary_loss,
                 *args, **kwargs):
        # ...
        if gan_loss is not None:
            self.gan_loss = MODULES.build(gan_loss)
        else:
            self.gan_loss = None

        if disc_auxiliary_loss:
            self.disc_auxiliary_losses = MODULES.build(disc_auxiliary_loss)
            if not isinstance(self.disc_auxiliary_losses, nn.ModuleList):
                self.disc_auxiliary_losses = nn.ModuleList(
                    [self.disc_auxiliary_losses])
        else:
            self.disc_auxiliary_loss = None

        if gen_auxiliary_loss:
            self.gen_auxiliary_losses = MODULES.build(gen_auxiliary_loss)
            if not isinstance(self.gen_auxiliary_losses, nn.ModuleList):
                self.gen_auxiliary_losses = nn.ModuleList(
                    [self.gen_auxiliary_losses])
        else:
            self.gen_auxiliary_losses = None

    def train_step(self, data: dict,
                   optim_wrapper: OptimWrapperDict) -> Dict[str, Tensor]:
        # ...

        # get data dict to compute losses for disc
        data_dict_ = dict(
            iteration=curr_iter,
            gen=self.generator,
            disc=self.discriminator,
            disc_pred_fake=disc_pred_fake,
            disc_pred_real=disc_pred_real,
            fake_imgs=fake_imgs,
            real_imgs=real_imgs)

        loss_disc, log_vars_disc = self._get_disc_loss(data_dict_)

        # ...

    def _get_disc_loss(self, outputs_dict):
```

(continues on next page)

(continued from previous page)

```
        # Construct losses dict. If you hope some items to be included in the
        # computational graph, you have to add 'loss' in its name. Otherwise,
        # items without 'loss' in their name will just be used to print
        # information.
        losses_dict = {}
        # gan loss
        losses_dict['loss_disc_fake'] = self.gan_loss(
            outputs_dict['disc_pred_fake'], target_is_real=False, is_disc=True)
        losses_dict['loss_disc_real'] = self.gan_loss(
            outputs_dict['disc_pred_real'], target_is_real=True, is_disc=True)

        # disc auxiliary loss
        if self.with_disc_auxiliary_loss:
            for loss_module in self.disc_auxiliary_losses:
                loss_ = loss_module(outputs_dict)
                if loss_ is None:
                    continue

                # the `loss_name()` function return name as 'loss_xxx'
                if loss_module.loss_name() in losses_dict:
                    losses_dict[loss_module.loss_name(
                    )] = losses_dict[loss_module.loss_name()] + loss_
                else:
                    losses_dict[loss_module.loss_name()] = loss_
        loss, log_var = self.parse_losses(losses_dict)

        return loss, log_var
```

Here, the `_get_disc_loss` will help to combine all kinds of losses automatically.

Therefore, as long as users design the loss module with the same rules, any kind of loss can be inserted in the training of generative models, without other modifications in the code of models. What you only need to do is just defining the `data_info` in the config files.

1.14.3 Available losses

We list available losses with examples in configs as follows.

regular losses

```
# dic gan
loss_gan=dict(
    type='GANLoss',
    gan_type='vanilla',
    loss_weight=0.001,
)
```

```
# deepfillv1
loss_gan=dict(
    type='GANLoss',
    gan_type='wgan',
    loss_weight=0.0001,
)
```

```
# deepfillv2
loss_gan=dict(
    type='GANLoss',
    gan_type='hinge',
    loss_weight=0.1,
)
```

```
# aot-gan
loss_gan=dict(
    type='GANLoss',
    gan_type='smgan',
    loss_weight=0.01,
)
```

```
# deepfillv1
loss_gp=dict(type='GradientPenaltyLoss', loss_weight=10.)
```

```
# deepfillv1
loss_disc_shift=dict(type='DiscShiftLoss', loss_weight=0.001)
```

```
# dim
loss_comp=dict(type='CharbonnierCompLoss', loss_weight=0.5)
```

```
# dic gan
feature_loss=dict(
    type='LightCNNFeatureLoss',
    pretrained=pretrained_light_cnn,
    loss_weight=0.1,
    criterion='l1')
```

```
# dic gan
pixel_loss=dict(type='L1Loss', loss_weight=1.0, reduction='mean')
```

```
# dic gan
align_loss=dict(type='MSELoss', loss_weight=0.1, reduction='mean')
```

```
# dim
loss_alpha=dict(type='CharbonnierLoss', loss_weight=0.5)
```

```
# partial conv
loss_tv=dict(
    type='MaskedTVLoss',
```

(continues on next page)

(continued from previous page)

```
    loss_weight=0.1
)
```

```
# real_basicvsr
perceptual_loss=dict(
    type='PerceptualLoss',
    layer_weights={
        '2': 0.1,
        '7': 0.1,
        '16': 1.0,
        '25': 1.0,
        '34': 1.0,
    },
    vgg_type='vgg19',
    perceptual_weight=1.0,
    style_weight=0,
    norm_img=False)
```

```
# ttsr
transferal_perceptual_loss=dict(
    type='TransferalPerceptualLoss',
    loss_weight=1e-2,
    use_attention=False,
    criterion='mse')
```

losses components

For `GANWithCustomizedLoss`, we provide several components to build customized loss.

1.15 Overview

- Number of checkpoints: 178
- Number of configs: 174
- Number of papers: 46
  - ALGORITHM: 47
- Tasks:
  - inpainting
  - image restoration
  - text2image
  - 3d-aware generation
  - colorization

  - image super-resolution
  - video interpolation
  - unconditional gans
  - video super-resolution
  - conditional gans
  - image generation
  - matting
  - image2image
  - image2image translation
  - internal learning

For supported datasets, see *datasets overview*.

1.15.1 AOT-GAN (TVCG'2021)

- Tasks: inpainting
- Number of checkpoints: 1
- Number of configs: 1
- Number of papers: 1
  - [ALGORITHM] Aggregated Contextual Transformations for High-Resolution Image Inpainting ()

1.15.2 BasicVSR (CVPR'2021)

- Tasks: video super-resolution
- Number of checkpoints: 3
- Number of configs: 3
- Number of papers: 1
  - [ALGORITHM] Basicvsr: The Search for Essential Components in Video Super-Resolution and Beyond ()

1.15.3 BasicVSR++ (CVPR'2022)

- Tasks: video super-resolution
- Number of checkpoints: 7
- Number of configs: 7
- Number of papers: 1
  - [ALGORITHM] Basicvsr++: Improving Video Super-Resolution With Enhanced Propagation and Alignment ()

1.15.4 BigGAN (ICLR'2019)

- Tasks: conditional gans
- Number of checkpoints: 7
- Number of configs: 6
- Number of papers: 1
  - [ALGORITHM] Large Scale {Gan ()

1.15.5 CAIN (AAAI'2020)

- Tasks: video interpolation
- Number of checkpoints: 1
- Number of configs: 1
- Number of papers: 1
  - [ALGORITHM] Channel Attention Is All You Need for Video Frame Interpolation ()

1.15.6 CycleGAN: Unpaired Image-to-Image Translation Using Cycle-Consistent Adversarial Networks (ICCV'2017)

- Tasks: image2image translation
- Number of checkpoints: 6
- Number of configs: 6
- Number of papers: 1
  - [ALGORITHM] Unpaired Image-to-Image Translation Using Cycle-Consistent Adversarial Networks ()

1.15.7 Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks (ICLR'2016)

- Tasks: unconditional gans
- Number of checkpoints: 3
- Number of configs: 3
- Number of papers: 1
  - [ALGORITHM] Unsupervised Representation Learning With Deep Convolutional Generative Adversarial Networks ()

1.15.8 DeepFillv1 (CVPR'2018)

- Tasks: inpainting
- Number of checkpoints: 2
- Number of configs: 2
- Number of papers: 1
  - [ALGORITHM] Generative Image Inpainting With Contextual Attention ()

1.15.9 DeepFillv2 (CVPR'2019)

- Tasks: inpainting
- Number of checkpoints: 2
- Number of configs: 2
- Number of papers: 1
  - [ALGORITHM] Free-Form Image Inpainting With Gated Convolution ()

1.15.10 DIC (CVPR'2020)

- Tasks: image super-resolution
- Number of checkpoints: 2
- Number of configs: 2
- Number of papers: 1
  - [ALGORITHM] Deep Face Super-Resolution With Iterative Collaboration Between Attentive Recovery and Landmark Estimation ()

1.15.11 DIM (CVPR'2017)

- Tasks: matting
- Number of checkpoints: 3
- Number of configs: 3
- Number of papers: 1
  - [ALGORITHM] Deep Image Matting ()

1.15.12 Disco Diffusion

- Tasks: image2image,text2image
- Number of checkpoints: 2
- Number of configs: 0
- Number of papers: 1
  - [ALGORITHM] Disco-Diffusion ()

1.15.13 EDSR (CVPR'2017)

- Tasks: image super-resolution
- Number of checkpoints: 3
- Number of configs: 3
- Number of papers: 1
  - [ALGORITHM] Enhanced Deep Residual Networks for Single Image Super-Resolution ()

1.15.14 EDVR (CVPRW'2019)

- Tasks: video super-resolution
- Number of checkpoints: 4
- Number of configs: 4
- Number of papers: 1
  - [ALGORITHM] Edvr: Video Restoration With Enhanced Deformable Convolutional Networks ()

1.15.15 EG3D (CVPR'2022)

- Tasks: 3d-aware generation
- Number of checkpoints: 3
- Number of configs: 3
- Number of papers: 1
  - [ALGORITHM] Efficient Geometry-Aware 3d Generative Adversarial Networks ()

1.15.16 ESRGAN (ECCVW'2018)

- Tasks: image super-resolution
- Number of checkpoints: 2
- Number of configs: 2
- Number of papers: 1
  - [ALGORITHM] Esrgan: Enhanced Super-Resolution Generative Adversarial Networks ()

1.15.17 FLAVR (arXiv'2020)

- Tasks: video interpolation
- Number of checkpoints: 1
- Number of configs: 1
- Number of papers: 1
  - [ALGORITHM] Flavr: Flow-Agnostic Video Representations for Fast Frame Interpolation ()

1.15.18 GCA (AAAI'2020)

- Tasks: matting
- Number of checkpoints: 4
- Number of configs: 4
- Number of papers: 1
  - [ALGORITHM] Natural Image Matting via Guided Contextual Attention ()

1.15.19 GGAN (ArXiv'2017)

- Tasks: unconditional gans
- Number of checkpoints: 3
- Number of configs: 3
- Number of papers: 1
  - [ALGORITHM] Geometric Gan ()

1.15.20 GLEAN (CVPR'2021)

- Tasks: image super-resolution
- Number of checkpoints: 4
- Number of configs: 7
- Number of papers: 1
  - [ALGORITHM] Glean: Generative Latent Bank for Large-Factor Image Super-Resolution ()

1.15.21 Global&Local (ToG'2017)

- Tasks: inpainting
- Number of checkpoints: 2
- Number of configs: 2
- Number of papers: 1
  - [ALGORITHM] Globally and Locally Consistent Image Completion ()

1.15.22 Guided Diffusion (NeurIPS'2021)

- Tasks: image generation
- Number of checkpoints: 2
- Number of configs: 0
- Number of papers: 1
  - [ALGORITHM] Diffusion Models Beat Gans on Image Synthesis ()

1.15.23 IconVSR (CVPR'2021)

- Tasks: video super-resolution
- Number of checkpoints: 3
- Number of configs: 3
- Number of papers: 1
  - [ALGORITHM] Basicvsr: The Search for Essential Components in Video Super-Resolution and Beyond ()

1.15.24 IndexNet (ICCV'2019)

- Tasks: matting
- Number of checkpoints: 2
- Number of configs: 2
- Number of papers: 1
  - [ALGORITHM] Indices Matter: Learning to Index for Deep Image Matting ()

1.15.25 Instance-aware Image Colorization (CVPR'2020)

- Tasks: colorization
- Number of checkpoints: 1
- Number of configs: 1
- Number of papers: 1
  - [ALGORITHM] Instance-Aware Image Colorization ()

1.15.26 LIIF (CVPR'2021)

- Tasks: image super-resolution
- Number of checkpoints: 2
- Number of configs: 2
- Number of papers: 1
  - [ALGORITHM] Learning Continuous Image Representation With Local Implicit Image Function ()

1.15.27 LSGAN (ICCV'2017)

- Tasks: unconditional gans
- Number of checkpoints: 4
- Number of configs: 4
- Number of papers: 1
  - [ALGORITHM] Least Squares Generative Adversarial Networks ()

1.15.28 NAFNet (ECCV'2022)

- Tasks: image restoration
- Number of checkpoints: 2
- Number of configs: 2
- Number of papers: 1
  - [ALGORITHM] Simple Baselines for Image Restoration ()

1.15.29 PConv (ECCV'2018)

- Tasks: inpainting
- Number of checkpoints: 2
- Number of configs: 4
- Number of papers: 1
  - [ALGORITHM] Image Inpainting for Irregular Holes Using Partial Convolutions ()

1.15.30 PGGAN (ICLR'2018)

- Tasks: unconditional gans
- Number of checkpoints: 3
- Number of configs: 3
- Number of papers: 1
  - [ALGORITHM] Progressive Growing of Gans for Improved Quality, Stability, and Variation ()

1.15.31 Pix2Pix (CVPR'2017)

- Tasks: image2image translation
- Number of checkpoints: 4
- Number of configs: 4
- Number of papers: 1
  - [ALGORITHM] Image-to-Image Translation With Conditional Adversarial Networks ()

1.15.32 Positional Encoding in GANs

- Tasks: unconditional gans
- Number of checkpoints: 21
- Number of configs: 21
- Number of papers: 1
  - [ALGORITHM] Positional Encoding as Spatial Inductive Bias in Gans ()

1.15.33 RDN (CVPR'2018)

- Tasks: image super-resolution
- Number of checkpoints: 3
- Number of configs: 3
- Number of papers: 1
  - [ALGORITHM] Residual Dense Network for Image Super-Resolution ()

1.15.34 RealBasicVSR (CVPR'2022)

- Tasks: video super-resolution
- Number of checkpoints: 2
- Number of configs: 2
- Number of papers: 1
  - [ALGORITHM] Realbasicvsr: Investigating Tradeoffs in Real-World Video Super-Resolution ()

1.15.35 Real-ESRGAN (ICCVW'2021)

- Tasks: image super-resolution
- Number of checkpoints: 2
- Number of configs: 2
- Number of papers: 1
  - [ALGORITHM] Real-Esrgan: Training Real-World Blind Super-Resolution With Pure Synthetic Data ()

1.15.36 SAGAN (ICML'2019)

- Tasks: conditional gans
- Number of checkpoints: 9
- Number of configs: 6
- Number of papers: 1
  - [ALGORITHM] Self-Attention Generative Adversarial Networks ()

1.15.37 SinGAN (ICCV'2019)

- Tasks: internal learning
- Number of checkpoints: 3
- Number of configs: 3
- Number of papers: 1
  - [ALGORITHM] Singan: Learning a Generative Model From a Single Natural Image ()

1.15.38 SNGAN (ICLR'2018)

- Tasks: conditional gans
- Number of checkpoints: 10
- Number of configs: 6
- Number of papers: 1
  - [ALGORITHM] Spectral Normalization for Generative Adversarial Networks ()

1.15.39 SRCNN (TPAMI'2015)

- Tasks: image super-resolution
- Number of checkpoints: 1
- Number of configs: 1
- Number of papers: 1
  - [ALGORITHM] Image Super-Resolution Using Deep Convolutional Networks ()

1.15.40 SRGAN (CVPR'2016)

- Tasks: image super-resolution
- Number of checkpoints: 2
- Number of configs: 2
- Number of papers: 1
  - [ALGORITHM] Photo-Realistic Single Image Super-Resolution Using a Generative Adversarial Network ()

1.15.41 StyleGANv1 (CVPR'2019)

- Tasks: unconditional gans
- Number of checkpoints: 2
- Number of configs: 2
- Number of papers: 1
  - [ALGORITHM] A Style-Based Generator Architecture for Generative Adversarial Networks ()

1.15.42 StyleGANv2 (CVPR'2020)

- Tasks: unconditional gans
- Number of checkpoints: 12
- Number of configs: 12
- Number of papers: 1
  - [ALGORITHM] Analyzing and Improving the Image Quality of Stylegan ()

1.15.43 StyleGANv3 (NeurIPS'2021)

- Tasks: unconditional gans
- Number of checkpoints: 9
- Number of configs: 10
- Number of papers: 1
  - [ALGORITHM] Alias-Free Generative Adversarial Networks ()

1.15.44 TDAN (CVPR'2020)

- Tasks: video super-resolution
- Number of checkpoints: 2
- Number of configs: 4
- Number of papers: 1
  - [ALGORITHM] Tdan: Temporally-Deformable Alignment Network for Video Super-Resolution ()

1.15.45 TOFlow (IJCV'2019)

- Tasks: video super-resolution,video interpolation
- Number of checkpoints: 6
- Number of configs: 6
- Number of papers: 1
  - [ALGORITHM] Video Enhancement With Task-Oriented Flow ()

1.15.46 TTSR (CVPR'2020)

- Tasks: image super-resolution
- Number of checkpoints: 2
- Number of configs: 2
- Number of papers: 1
  - [ALGORITHM] Learning Texture Transformer Network for Image Super-Resolution ()

1.15.47 WGAN-GP (NeurIPS'2017)

- Tasks: unconditional gans
- Number of checkpoints: 2
- Number of configs: 2
- Number of papers: 1
  - [ALGORITHM] Improved Training of Wasserstein Gans ()

1.16 Overview

- *Prepare Super-Resolution Datasets*
  - *DF2K_OST* [Homepage]
  - *DIV2K* [Homepage]
  - *REDS* [Homepage]
  - *Vid4* [Homepage]
  - *Vimeo90K* [Homepage]
- *Prepare Inpainting Datasets*
  - *CelebA-HQ* [Homepage]
  - *Paris Street View* [Homepage]
  - *Places365* [Homepage]
- *Prepare Matting Datasets*
  - *Composition-1k* [Homepage]
- *Prepare Video Frame Interpolation Datasets*
  - *Vimeo90K-triplet* [Homepage]
- *Prepare Unconditional GANs Datasets*
- *Prepare Image Translation Datasets*

1.17 Super-Resolution Datasets

It is recommended to symlink the dataset root to `$MMEDITING/data`. If your folder structure is different, you may need to change the corresponding paths in config files.

MMEditing supported super-resolution datasets:

- Image Super-Resolution
  - *DF2K_OST* [Homepage]
  - *DIV2K* [Homepage]
- Video Super-Resolution
  - *REDS* [Homepage]
  - *Vid4* [Homepage]
  - *Vimeo90K* [Homepage]

1.17.1 DF2K_OST Dataset

```
@inproceedings{wang2021real,
  title={Real-ESRGAN: Training Real-World Blind Super-Resolution with Pure Synthetic␣
↪Data},
  author={Wang, Xintao and Xie, Liangbin and Dong, Chao and Shan, Ying},
  booktitle={Proceedings of the IEEE/CVF International Conference on Computer Vision},
  pages={1905--1914},
  year={2021}
}
```

- The DIV2K dataset can be downloaded from here (We use the training set only).
- The Flickr2K dataset can be downloaded here (We use the training set only).
- The OST dataset can be downloaded here (We use the training set only).

Please first put all the images into the `GT` folder (naming does not need to be in order):

```
mmediting
├── mmedit
├── tools
├── configs
├── data
    ├── df2k_ost
        ├── GT
            ├── 0001.png
            ├── 0002.png
            ├── ...
...
```

Crop sub-images

For faster IO, we recommend to crop the images to sub-images. We provide such a script:

```
python tools/dataset_converters/super-resolution/df2k_ost/preprocess_df2k_ost_dataset.py␣
↪--data-root ./data/df2k_ost
```

The generated data is stored under `df2k_ost` and the data structure is as follows, where `_sub` indicates the sub-images.

```
mmediting
├── mmedit
├── tools
├── configs
├── data
    ├── df2k_ost
        ├── GT
        ├── GT_sub
...
```

Prepare LMDB dataset for DF2K_OST

If you want to use LMDB datasets for faster IO speed, you can make LMDB files by:

```
python tools/dataset_converters/super-resolution/df2k_ost/preprocess_df2k_ost_dataset.py␣
↪--data-root ./data/df2k_ost --make-lmdb
```

1.17.2 DIV2K Dataset

```
@InProceedings{Agustsson_2017_CVPR_Workshops,
    author = {Agustsson, Eirikur and Timofte, Radu},
    title = {NTIRE 2017 Challenge on Single Image Super-Resolution: Dataset and Study},
    booktitle = {The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)␣
↪Workshops},
    month = {July},
    year = {2017}
}
```

- Training dataset: DIV2K dataset.
- Validation dataset: Set5 and Set14.

```
mmediting
├── mmedit
├── tools
├── configs
├── data
    ├── DIV2K
        ├── DIV2K_train_HR
        ├── DIV2K_train_LR_bicubic
            ├── X2
            ├── X3
            ├── X4
```

(continues on next page)

(continued from previous page)

```
│   │   ├── DIV2K_valid_HR
│   │   ├── DIV2K_valid_LR_bicubic
│   │   │   ├── X2
│   │   │   ├── X3
│   │   │   └── X4
│   ├── Set5
│   │   ├── GTmod12
│   │   ├── LRbicx2
│   │   ├── LRbicx3
│   │   └── LRbicx4
│   ├── Set14
│   │   ├── GTmod12
│   │   ├── LRbicx2
│   │   ├── LRbicx3
│   │   └── LRbicx4
```

Crop sub-images

For faster IO, we recommend to crop the DIV2K images to sub-images. We provide such a script:

```
python tools/dataset_converters/super-resolution/div2k/preprocess_div2k_dataset.py --
↪data-root ./data/DIV2K
```

The generated data is stored under `DIV2K` and the data structure is as follows, where `_sub` indicates the sub-images.

```
mmediting
├── mmedit
├── tools
├── configs
├── data
│   ├── DIV2K
│   │   ├── DIV2K_train_HR
│   │   ├── DIV2K_train_HR_sub
│   │   ├── DIV2K_train_LR_bicubic
│   │   │   ├── X2
│   │   │   ├── X3
│   │   │   ├── X4
│   │   │   ├── X2_sub
│   │   │   ├── X3_sub
│   │   │   └── X4_sub
│   │   ├── DIV2K_valid_HR
│   │   ├── ...
...
```

Prepare annotation list

If you use the annotation mode for the dataset, you first need to prepare a specific `txt` file.

Each line in the annotation file contains the image names and image shape (usually for the ground-truth images), separated by a white space.

Example of an annotation file:

```
0001_s001.png (480,480,3)
0001_s002.png (480,480,3)
```

Prepare LMDB dataset for DIV2K

If you want to use LMDB datasets for faster IO speed, you can make LMDB files by:

```
python tools/dataset_converters/super-resolution/div2k/preprocess_div2k_dataset.py --
↪data-root ./data/DIV2K --make-lmdb
```

1.17.3 REDS Dataset

```
@InProceedings{Nah_2019_CVPR_Workshops_REDS,
  author = {Nah, Seungjun and Baik, Sungyong and Hong, Seokil and Moon, Gyeongsik and␣
↪Son, Sanghyun and Timofte, Radu and Lee, Kyoung Mu},
  title = {NTIRE 2019 Challenge on Video Deblurring and Super-Resolution: Dataset and␣
↪Study},
  booktitle = {The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)␣
↪Workshops},
  month = {June},
  year = {2019}
}
```

- Training dataset: REDS dataset.
- Validation dataset: REDS dataset and Vid4.

Note that we merge train and val datasets in REDS for easy switching between REDS4 partition (used in EDVR) and the official validation partition. The original val dataset (clip names from 000 to 029) are modified to avoid conflicts with training dataset (total 240 clips). Specifically, the clip names are changed to 240, 241, … 269.

You can prepare the REDS dataset by running:

```
python tools/dataset_converters/super-resolution/reds/preprocess_reds_dataset.py --root-
↪path ./data/REDS
```

```
mmediting
├── mmedit
├── tools
├── configs
├── data
│   ├── REDS
│   │   ├── train_sharp
│   │   │   ├── 000
```

(continues on next page)

(continued from previous page)

```
│   │   │   ├── 001
│   │   │   ├── ...
│   │   ├── train_sharp_bicubic
│   │   │   ├── 000
│   │   │   ├── 001
│   │   │   ├── ...
│   ├── REDS4
│   │   ├── GT
│   │   ├── sharp_bicubic
```

Prepare LMDB dataset for REDS

If you want to use LMDB datasets for faster IO speed, you can make LMDB files by:

```
python tools/dataset_converters/super-resolution/reds/preprocess_reds_dataset.py --root-
↪path ./data/REDS --make-lmdb
```

Crop to sub-images

MMEditing also support cropping REDS images to sub-images for faster IO. We provide such a script:

```
python tools/dataset_converters/super-resolution/reds/crop_sub_images.py --data-root ./
↪data/REDS  -scales 4
```

The generated data is stored under `REDS` and the data structure is as follows, where `_sub` indicates the sub-images.

```
mmediting
├── mmedit
├── tools
├── configs
├── data
│   ├── REDS
│   │   ├── train_sharp
│   │   │   ├── 000
│   │   │   ├── 001
│   │   │   ├── ...
│   │   ├── train_sharp_sub
│   │   │   ├── 000_s001
│   │   │   ├── 000_s002
│   │   │   ├── ...
│   │   │   ├── 001_s001
│   │   │   ├── ...
│   │   ├── train_sharp_bicubic
│   │   │   ├── X4
│   │   │   │   ├── 000
│   │   │   │   ├── 001
│   │   │   │   ├── ...
│   │   │   ├── X4_sub
│   │   │   ├── 000_s001
│   │   │   ├── 000_s002
```

(continues on next page)

(continued from previous page)

```
│   │   │   ├── ...
│   │   │   ├── 001_s001
│   │   │   ├── ...
```

Note that by default `preprocess_reds_dataset.py` does not make lmdb and annotation file for the cropped dataset. You may need to modify the scripts a little bit for such operations.

1.17.4 Vid4 Dataset

```
@article{xue2019video,
  title={On Bayesian adaptive video super resolution},
  author={Liu, Ce and Sun, Deqing},
  journal={IEEE Transactions on Pattern Analysis and Machine Intelligence},
  volume={36},
  number={2},
  pages={346--360},
  year={2013},
  publisher={IEEE}
}
```

The Vid4 dataset can be downloaded from here. There are two degradations in the dataset.

1. BIx4 contains images downsampled by bicubic interpolation
2. BDx4 contains images blurred by Gaussian kernel with =1.6, followed by a subsampling every four pixels.

1.17.5 Vimeo90K Dataset

```
@article{xue2019video,
  title={Video Enhancement with Task-Oriented Flow},
  author={Xue, Tianfan and Chen, Baian and Wu, Jiajun and Wei, Donglai and Freeman,␣
↪William T},
  journal={International Journal of Computer Vision (IJCV)},
  volume={127},
  number={8},
  pages={1106--1125},
  year={2019},
  publisher={Springer}
}
```

The training and test datasets can be download from here.

The Vimeo90K dataset has a `clip/sequence/img` folder structure:

```
mmediting
├── mmedit
├── tools
├── configs
├── data
│   ├── vimeo_triplet
│   │   ├── BDx4
```

(continues on next page)

(continued from previous page)

```
│   │   │   ├── 00001
│   │   │   │   ├── 0001
│   │   │   │   │   ├── im1.png
│   │   │   │   │   ├── im2.png
│   │   │   │   │   ├── ...
│   │   │   │   ├── 0002
│   │   │   │   ├── 0003
│   │   │   │   ├── ...
│   │   │   ├── 00002
│   │   │   ├── ...
│   │   ├── BIx4
│   │   ├── GT
│   │   ├── meta_info_Vimeo90K_test_GT.txt
│   │   ├── meta_info_Vimeo90K_train_GT.txt
```

Prepare the annotation files for Vimeo90K dataset

To prepare the annotation file for training, you need to download the official training list path for Vimeo90K from the official website, and run the following command:

```
python tools/dataset_converters/super-resolution/vimeo90k/preprocess_vimeo90k_dataset.py␣
↪./data/Vimeo90K/official_train_list.txt
```

The annotation file for test is generated similarly.

Prepare LMDB dataset for Vimeo90K

If you want to use LMDB datasets for faster IO speed, you can make LMDB files by:

```
python tools/dataset_converters/super-resolution/vimeo90k/preprocess_vimeo90k_dataset.py␣
↪./data/Vimeo90K/official_train_list.txt --gt-path ./data/Vimeo90K/GT --lq-path ./data/
↪Vimeo90K/LQ  --make-lmdb
```

1.18 Inpainting Datasets

It is recommended to symlink the dataset root to `$MMEDITING/data`. If your folder structure is different, you may need to change the corresponding paths in config files.

MMEditing supported inpainting datasets:

- *CelebA-HQ* [Homepage]
- *Paris Street View* [Homepage]
- *Places365* [Homepage]

As we only need images for inpainting task, further preparation is not necessary and the folder structure can be different from the example. You can utilize the information provided by the original dataset like `Place365` (e.g. `meta`). Also, you can easily scan the data set and list all of the images to a specific `txt` file. Here is an example for the `Places365_val.txt` from Places365 and we will only use the image name information in inpainting.

```
Places365_val_00000001.jpg 165
Places365_val_00000002.jpg 358
Places365_val_00000003.jpg 93
Places365_val_00000004.jpg 164
Places365_val_00000005.jpg 289
Places365_val_00000006.jpg 106
Places365_val_00000007.jpg 81
Places365_val_00000008.jpg 121
Places365_val_00000009.jpg 150
Places365_val_00000010.jpg 302
Places365_val_00000011.jpg 42
```

1.18.1 CelebA-HQ Dataset

```
@article{karras2017progressive,
  title={Progressive growing of gans for improved quality, stability, and variation},
  author={Karras, Tero and Aila, Timo and Laine, Samuli and Lehtinen, Jaakko},
  journal={arXiv preprint arXiv:1710.10196},
  year={2017}
}
```

Follow the instructions here to prepare the dataset.

```
mmediting
├── mmedit
├── tools
├── configs
├── data
|   ├── CelebA-HQ
|   |   ├── train_256
|   |   ├── test_256
|   |   ├── train_celeba_img_list.txt
|   |   ├── val_celeba_img_list.txt
```

1.18.2 Paris Street View Dataset

```
@inproceedings{pathak2016context,
  title={Context encoders: Feature learning by inpainting},
  author={Pathak, Deepak and Krahenbuhl, Philipp and Donahue, Jeff and Darrell, Trevor␣
↪and Efros, Alexei A},
  booktitle={Proceedings of the IEEE conference on computer vision and pattern␣
↪recognition},
  pages={2536--2544},
  year={2016}
}
```

Obtain the dataset here.

```
mmediting
├── mmedit
```

(continues on next page)

(continued from previous page)

```
├── tools
├── configs
├── data
    ├── paris_street_view
        ├── train
|   |   ├── val
```

1.18.3 Places365 Dataset

```
@article{zhou2017places,
  title={Places: A 10 million Image Database for Scene Recognition},
  author={Zhou, Bolei and Lapedriza, Agata and Khosla, Aditya and Oliva, Aude and␣
↪Torralba, Antonio},
  journal={IEEE Transactions on Pattern Analysis and Machine Intelligence},
  year={2017},
  publisher={IEEE}
}
```

Prepare the data from Places365.

```
mmediting
├── mmedit
├── tools
├── configs
├── data
    ├── Places
        ├── data_large
        ├── val_large
|   |   ├── meta
|   |   |   ├── places365_train_challenge.txt
|   |   |   ├── places365_val.txt
```

1.19 Matting Datasets

It is recommended to symlink the dataset root to `$MMEDITING/data`. If your folder structure is different, you may need to change the corresponding paths in config files.

MMEditing supported matting datasets:

- *Composition-1k* [Homepage]

1.19.1 Composition-1k Dataset

Introduction

```
@inproceedings{xu2017deep,
  title={Deep image matting},
  author={Xu, Ning and Price, Brian and Cohen, Scott and Huang, Thomas},
  booktitle={Proceedings of the IEEE conference on computer vision and pattern␣
↪recognition},
  pages={2970--2979},
  year={2017}
}
```

The Adobe Composition-1k dataset consists of foreground images and their corresponding alpha images. To get the full dataset, one need to composite the foregrounds with selected backgrounds from the COCO dataset and the Pascal VOC dataset.

Obtain and Extract

Please follow the instructions of paper authors to obtain the Composition-1k (comp1k) dataset.

Composite the full dataset

The Adobe composition-1k dataset contains only `alpha` and `fg` (and `trimap` in test set). It is needed to merge `fg` with COCO data (training) or VOC data (test) before training or evaluation. Use the following script to perform image composition and generate annotation files for training or testing:

```
## The script is run under the root folder of MMEditing
python tools/dataset_converters/matting/comp1k/preprocess_comp1k_dataset.py data/adobe_
↪composition-1k data/coco data/VOCdevkit --composite
```

The generated data is stored under `adobe_composition-1k/Training_set` and `adobe_composition-1k/Test_set` respectively. If you only want to composite test data (since compositing training data is time-consuming), you can skip compositing the training set by removing the `--composite` option:

```
## skip compositing training set
python tools/dataset_converters/matting/comp1k/preprocess_comp1k_dataset.py data/adobe_
↪composition-1k data/coco data/VOCdevkit
```

If you only want to preprocess test data, i.e. for FBA, you can skip the train set by adding the `--skip-train` option:

```
## skip preprocessing training set
python tools/data/matting/comp1k/preprocess_comp1k_dataset.py data/adobe_composition-1k␣
↪data/coco data/VOCdevkit --skip-train
```

> Currently, `GCA` and `FBA` support online composition of training data. But you can modify the data pipeline of other models to perform online composition instead of loading composited images (we called it `merged` in our data pipeline).

Check Directory Structure for DIM

The result folder structure should look like:

```
mmediting
├── mmedit
├── tools
├── configs
├── data
│   ├── adobe_composition-1k
│   │   ├── Test_set
│   │   │   ├── Adobe-licensed images
│   │   │   │   ├── alpha
│   │   │   │   ├── fg
│   │   │   │   ├── trimaps
│   │   │   ├── merged  (generated by tools/dataset_converters/matting/comp1k/preprocess_
↪comp1k_dataset.py)
│   │   │   ├── bg      (generated by tools/dataset_converters/matting/comp1k/preprocess_
↪comp1k_dataset.py)
│   │   ├── Training_set
│   │   │   ├── Adobe-licensed images
│   │   │   │   ├── alpha
│   │   │   │   ├── fg
│   │   │   ├── Other
│   │   │   │   ├── alpha
│   │   │   │   ├── fg
│   │   │   ├── merged  (generated by tools/dataset_converters/matting/comp1k/preprocess_
↪comp1k_dataset.py)
│   │   │   ├── bg      (generated by tools/dataset_converters/matting/comp1k/preprocess_
↪comp1k_dataset.py)
│   │   ├── test_list.json     (generated by tools/dataset_converters/matting/comp1k/
↪preprocess_comp1k_dataset.py)
│   │   ├── training_list.json (generated by tools/dataset_converters/matting/comp1k/
↪preprocess_comp1k_dataset.py)
│   ├── coco
│   │   ├── train2014   (or train2017)
│   ├── VOCdevkit
│   │   ├── VOC2012
```

Prepare the dataset for FBA

FBA adopts dynamic dataset augmentation proposed in Learning-base Sampling for Natural Image Matting. In addition, to reduce artifacts during augmentation, it uses the extended version of foreground as foreground. We provide scripts to estimate foregrounds.

Prepare the test set as follows:

```
## skip preprocessing training set, as it composites online during training
python tools/dataset_converters/matting/comp1k/preprocess_comp1k_dataset.py data/adobe_
↪composition-1k data/coco data/VOCdevkit --skip-train
```

Extend the foreground of training set as follows:

```
python tools/dataset_converters/matting/comp1k/extend_fg.py data/adobe_composition-1k
```

Check Directory Structure for DIM

The final folder structure should look like:

```
mmediting
├── mmedit
├── tools
├── configs
├── data
    ├── adobe_composition-1k
        ├── Test_set
            ├── Adobe-licensed images
                ├── alpha
                ├── fg
                ├── trimaps
            ├── merged  (generated by tools/data/matting/comp1k/preprocess_comp1k_
↪dataset.py)
            ├── bg      (generated by tools/data/matting/comp1k/preprocess_comp1k_
↪dataset.py)
        ├── Training_set
            ├── Adobe-licensed images
                ├── alpha
                ├── fg
                ├── fg_extended (generated by tools/data/matting/comp1k/extend_fg.py)
            ├── Other
                ├── alpha
                ├── fg
                ├── fg_extended (generated by tools/data/matting/comp1k/extend_fg.py)
        ├── test_list.json         (generated by tools/data/matting/comp1k/preprocess_
↪comp1k_dataset.py)
        ├── training_list_fba.json (generated by tools/data/matting/comp1k/extend_fg.py)
    ├── coco
        ├── train2014   (or train2017)
    ├── VOCdevkit
        ├── VOC2012
```

1.20 Video Frame Interpolation Datasets

It is recommended to symlink the dataset root to `$MMEDITING/data`. If your folder structure is different, you may need to change the corresponding paths in config files.

MMEditing supported video frame interpolation datasets:

- *Vimeo90K-triplet* [Homepage]

1.20.1 Vimeo90K-triplet Dataset

```
@article{xue2019video,
  title={Video Enhancement with Task-Oriented Flow},
  author={Xue, Tianfan and Chen, Baian and Wu, Jiajun and Wei, Donglai and Freeman,␣
↪William T},
  journal={International Journal of Computer Vision (IJCV)},
  volume={127},
  number={8},
  pages={1106--1125},
  year={2019},
  publisher={Springer}
}
```

The training and test datasets can be download from here.

The Vimeo90K-triplet dataset has a `clip/sequence/img` folder structure:

```
mmediting
├── mmedit
├── tools
├── configs
├── data
    ├── vimeo_triplet
        ├── tri_testlist.txt
        ├── tri_trainlist.txt
        ├── sequences
            ├── 00001
                ├── 0001
                    ├── im1.png
                    ├── im2.png
                    └── im3.png
                ├── 0002
                ├── 0003
                ├── ...
            ├── 00002
            ├── ...
```

1.21 Unconditional GANs Datasets

Data preparation for unconditional model is simple. What you need to do is downloading the images and put them into a directory. Next, you should set a symlink in the `data` directory. For standard unconditional gans with static architectures, like DCGAN and StyleGAN2, `UnconditionalImageDataset` is designed to train such unconditional models. Here is an example config for FFHQ dataset:

```
dataset_type = 'BasicImageDataset'

train_pipeline = [
    dict(type='LoadImageFromFile', key='img'),
    dict(type='Flip', keys=['img'], direction='horizontal'),
    dict(type='PackEditInputs', keys=['img'], meta_keys=['img_path'])
]
```

(continues on next page)

(continued from previous page)

```
# `batch_size` and `data_root` need to be set.
train_dataloader = dict(
    batch_size=4,
    num_workers=8,
    persistent_workers=True,
    sampler=dict(type='InfiniteSampler', shuffle=True),
    dataset=dict(
        type=dataset_type,
        data_root=None,  # set by user
        pipeline=train_pipeline))
```

Here, we adopt `InfinitySampler` to avoid frequent dataloader reloading, which will accelerate the training procedure. As shown in the example, `pipeline` provides important data pipeline to process images, including loading from file system, resizing, cropping, transferring to `torch.Tensor` and packing to `EditDataSample`. All of supported data pipelines can be found in `mmedit/datasets/transforms`.

For unconditional GANs with dynamic architectures like PGGAN and StyleGANv1, `GrowScaleImgDataset` is recommended to use for training. Since such dynamic architectures need real images in different scales, directly adopting `UnconditionalImageDataset` will bring heavy I/O cost for loading multiple high-resolution images. Here is an example we use for training PGGAN in CelebA-HQ dataset:

```
dataset_type = 'GrowScaleImgDataset'

pipeline = [
    dict(type='LoadImageFromFile', key='img'),
    dict(type='Flip', keys=['img'], direction='horizontal'),
    dict(type='PackEditInputs')
]

# `samples_per_gpu` and `imgs_root` need to be set.
train_dataloader = dict(
    num_workers=4,
    batch_size=64,
    dataset=dict(
        type='GrowScaleImgDataset',
        data_roots={
            '1024': './data/ffhq/images',
            '256': './data/ffhq/ffhq_imgs/ffhq_256',
            '64': './data/ffhq/ffhq_imgs/ffhq_64'
        },
        gpu_samples_base=4,
        # note that this should be changed with total gpu number
        gpu_samples_per_scale={
            '4': 64,
            '8': 32,
            '16': 16,
            '32': 8,
            '64': 4,
            '128': 4,
            '256': 4,
            '512': 4,
            '1024': 4
```

(continues on next page)

(continued from previous page)

```
        },
        len_per_stage=300000,
        pipeline=pipeline),
    sampler=dict(type='InfiniteSampler', shuffle=True))
```

In this dataset, you should provide a dictionary of image paths to the `data_roots`. Thus, you should resize the images in the dataset in advance. For the resizing methods in the data pre-processing, we adopt bilinear interpolation methods in all of the experiments studied in MMEditing.

Note that this dataset should be used with `PGGANFetchDataHook`. In this config file, this hook should be added in the customized hooks, as shown below.

```
custom_hooks = [
    dict(
        type='GenVisualizationHook',
        interval=5000,
        fixed_input=True,
        # vis ema and orig at the same time
        vis_kwargs_list=dict(
            type='Noise',
            name='fake_img',
            sample_model='ema/orig',
            target_keys=['ema', 'orig'])),
    dict(type='PGGANFetchDataHook')
]
```

This fetching data hook helps the dataloader update the status of dataset to change the data source and batch size during training.

Here, we provide several download links of datasets frequently used in unconditional models: LSUN, CelebA, CelebA-HQ, FFHQ.

1.22 Image Translation Datasets

Data preparation for translation model needs a little attention. You should organize the files in the way we told you in `quick_run.md`. Fortunately, for most official datasets like facades and summer2winter_yosemite, they already have the right format. Also, you should set a symlink in the `data` directory. For paired-data trained translation model like Pix2Pix , `PairedImageDataset` is designed to train such translation models. Here is an example config for facades dataset:

```
train_dataset_type = 'PairedImageDataset'
val_dataset_type = 'PairedImageDataset'
img_norm_cfg = dict(mean=[0.5, 0.5, 0.5], std=[0.5, 0.5, 0.5])
train_pipeline = [
    dict(
        type='LoadPairedImageFromFile',
        io_backend='disk',
        key='pair',
        domain_a=domain_a,
        domain_b=domain_b,
        flag='color'),
    dict(
```

(continues on next page)

(continued from previous page)

```
        type='Resize',
        keys=[f'img_{domain_a}', f'img_{domain_b}'],
        scale=(286, 286),
        interpolation='bicubic')
]
test_pipeline = [
    dict(
        type='LoadPairedImageFromFile',
        io_backend='disk',
        key='image',
        domain_a=domain_a,
        domain_b=domain_b,
        flag='color'),
    dict(
        type='Resize',
        keys=[f'img_{domain_a}', f'img_{domain_b}'],
        scale=(256, 256),
        interpolation='bicubic')
]
dataroot = 'data/paired/facades'
train_dataloader = dict(
    batch_size=1,
    num_workers=4,
    persistent_workers=True,
    sampler=dict(type='InfiniteSampler', shuffle=True),
    dataset=dict(
        type=dataset_type,
        data_root=dataroot,  # set by user
        pipeline=train_pipeline))

val_dataloader = dict(
    batch_size=1,
    num_workers=4,
    dataset=dict(
        type=dataset_type,
        data_root=dataroot,  # set by user
        pipeline=test_pipeline),
    sampler=dict(type='DefaultSampler', shuffle=False),
    persistent_workers=True)

test_dataloader = dict(
    batch_size=1,
    num_workers=4,
    dataset=dict(
        type=dataset_type,
        data_root=dataroot,  # set by user
        pipeline=test_pipeline),
    sampler=dict(type='DefaultSampler', shuffle=False),
    persistent_workers=True)
```

Here, we adopt `LoadPairedImageFromFile` to load a paired image as the common loader does and crops it into two images with the same shape in different domains. As shown in the example, `pipeline` provides important data pipeline to process images, including loading from file system, resizing, cropping, flipping, transferring to `torch.Tensor` and

packing to `EditDataSample`. All of supported data pipelines can be found in `mmedit/datasets/transforms`.

For unpaired-data trained translation model like CycleGAN , `UnpairedImageDataset` is designed to train such translation models. Here is an example config for horse2zebra dataset:

```
train_dataset_type = 'UnpairedImageDataset'
val_dataset_type = 'UnpairedImageDataset'
img_norm_cfg = dict(mean=[0.5, 0.5, 0.5], std=[0.5, 0.5, 0.5])
domain_a, domain_b = 'horse', 'zebra'
train_pipeline = [
    dict(
        type='LoadImageFromFile',
        io_backend='disk',
        key=f'img_{domain_a}',
        flag='color'),
    dict(
        type='LoadImageFromFile',
        io_backend='disk',
        key=f'img_{domain_b}',
        flag='color'),
    dict(
        type='TransformBroadcaster',
        mapping={'img': [f'img_{domain_a}', f'img_{domain_b}']},
        auto_remap=True,
        share_random_params=True,
        transforms=[
            dict(type='Resize', scale=(286, 286), interpolation='bicubic'),
            dict(type='Crop', crop_size=(256, 256), random_crop=True),
        ]),
    dict(type='Flip', keys=[f'img_{domain_a}'], direction='horizontal'),
    dict(type='Flip', keys=[f'img_{domain_b}'], direction='horizontal'),
    dict(
        type='PackEditInputs',
        keys=[f'img_{domain_a}', f'img_{domain_b}'])
]
test_pipeline = [
    dict(type='LoadImageFromFile', io_backend='disk', key='img', flag='color'),
    dict(type='Resize', scale=(256, 256), interpolation='bicubic'),
    dict(
        type='PackEditInputs',
        keys=[f'img_{domain_a}', f'img_{domain_b}'])
]
data_root = './data/horse2zebra/'
# `batch_size` and `data_root` need to be set.
train_dataloader = dict(
    batch_size=1,
    num_workers=4,
    persistent_workers=True,
    sampler=dict(type='InfiniteSampler', shuffle=True),
    dataset=dict(
        type=dataset_type,
        data_root=data_root,  # set by user
        pipeline=train_pipeline))
```

(continues on next page)

(continued from previous page)

```
val_dataloader = dict(
    batch_size=None,
    num_workers=4,
    dataset=dict(
        type=dataset_type,
        data_root=data_root,  # set by user
        pipeline=test_pipeline),
    sampler=dict(type='DefaultSampler', shuffle=False),
    persistent_workers=True)

test_dataloader = dict(
    batch_size=None,
    num_workers=4,
    dataset=dict(
        type=dataset_type,
        data_root=data_root,  # set by user
        pipeline=test_pipeline),
    sampler=dict(type='DefaultSampler', shuffle=False),
    persistent_workers=True)
```

`UnpairedImageDataset` will load both images (domain A and B) from different paths and transform them at the same time.

Here, we provide download links of datasets used in Pix2Pix and CycleGAN.

1.23 Overview

This section introduce the following contents in terms of migration from MMEditing 0.x

- *New dependencies*
- *Overall structures*

1.23.1 New dependencies

MMEdit 1.x depends on some new packages, you can prepare a new clean environment and install again according to the *install tutorial*. Or install the below packages manually.

1. MMEngine: MMEngine is the core the OpenMMLab 2.0 architecture, and we splited many compentents unrelated to computer vision from MMCV to MMEngine.
2. MMCV: The computer vision package of OpenMMLab. This is not a new dependency, but you need to upgrade it to above 2.0.0rc0 version.
3. rich: A terminal formatting package, and we use it to beautify some outputs in the terminal.

1.23.2 Overall structures

We refactor overall structures in MMEdit 1.x as following.

- The `core` in the old versions of MMEdit is split into `engine`, `evaluation`, `structures`, and `visualization`
- The `pipelines` of `datasets` in the old versions of MMEdit is refactored to `transforms`
- The `models` in MMedit 1.x is refactored to five parts: `base_models`, `data_preprocessors`, `editors`, `layers` and `losses`.

1.23.3 Other config settings

We rename config file to new template: `{model_settings}_{module_setting}_{training_setting}_{datasets_info}`.

More details of config are shown in *config guides*.

1.24 Migration of Runtime Settings

We update runtime settings in MMEdit 1.x. Important modifications are as following.

- The `checkpoint_config` is moved to `default_hooks.checkpoint` and the `log_config` is moved to `default_hooks.logger`. And we move many hooks settings from the script code to the `default_hooks` field in the runtime configuration.
- The `resume_from` is removed. And we use `resume` to replace it.
  - If resume=True and load_from is not None, resume training from the checkpoint in load_from.
  - If resume=True and load_from is None, try to resume from the latest checkpoint in the work directory.
  - If resume=False and load_from is not None, only load the checkpoint, not resume training.
  - If resume=False and load_from is None, do not load nor resume.
- The `dist_params` field is a sub field of `env_cfg` now. And there are some new configurations in the `env_cfg`.
- The `workflow` related functionalities are removed.
- New field `visualizer`: The visualizer is a new design. We use a visualizer instance in the runner to handle results & log visualization and save to different backends, like Local, TensorBoard and Wandb.
- New field `default_scope`: The start point to search module for all registries.

```
checkpoint_config = dict(  # Config to set the checkpoint hook, Refer to https://github.
↪com/open-mmlab/mmcv/blob/master/mmcv/runner/hooks/checkpoint.py for implementation.
    interval=5000,  # The save interval is 5000 iterations
    save_optimizer=True,  # Also save optimizers
    by_epoch=False)  # Count by iterations
log_config = dict(  # Config to register logger hook
    interval=100,  # Interval to print the log
    hooks=[
        dict(type='TextLoggerHook', by_epoch=False),  # The logger used to record the␣
↪training process
        dict(type='TensorboardLoggerHook'),  # The Tensorboard logger is also supported
    ])
visual_config = None  # Visual config, we do not use it.
# runtime settings
```

(continues on next page)

(continued from previous page)

```
dist_params = dict(backend='nccl')  # Parameters to setup distributed training, the port␣
↪can also be set
log_level = 'INFO'  # The level of logging
load_from = None # load models as a pre-trained model from a given path. This will not␣
↪resume training
resume_from = None # Resume checkpoints from a given path, the training will be resumed␣
↪from the iteration when the checkpoint's is saved
workflow = [('train', 1)]  # Workflow for runner. [('train', 1)] means there is only one␣
↪workflow and the workflow named 'train' is executed once. Keep this unchanged when␣
↪training current matting models
```

```
default_hooks = dict(  # Used to build default hooks
    checkpoint=dict(  # Config to set the checkpoint hook
        type='CheckpointHook',
        interval=5000,  # The save interval is 5000 iterations
        save_optimizer=True,
        by_epoch=False,  # Count by iterations
        out_dir=save_dir,
    ),
    timer=dict(type='IterTimerHook'),
    logger=dict(type='LoggerHook', interval=100),  # Config to register logger hook
    param_scheduler=dict(type='ParamSchedulerHook'),
    sampler_seed=dict(type='DistSamplerSeedHook'),
)
default_scope = 'mmedit' # Used to set registries location
env_cfg = dict(  # Parameters to setup distributed training, the port can also be set
    cudnn_benchmark=False,
    mp_cfg=dict(mp_start_method='fork', opencv_num_threads=4),
    dist_cfg=dict(backend='nccl'),
)
log_level = 'INFO'  # The level of logging
log_processor = dict(type='LogProcessor', window_size=100, by_epoch=False)  # Used to␣
↪build log processor
load_from = None  # load models as a pre-trained model from a given path. This will not␣
↪resume training.
resume = False  # Resume checkpoints from a given path, the training will be resumed␣
↪from the epoch when the checkpoint's is saved.
```

1.25 Migration of Model Settings

We update model settings in MMEdit 1.x. Important modifications are as following.

- Remove `pretrained` fields.
- Add `train_cfg` and `test_cfg` fields in model settings.
- Add `data_preprocessor` fields. Normalization and color space transforms operations are moved from datasets transforms pipelines to data_preprocessor. We will introduce data_preprocessor later.

```
model = dict(
    type='BasicRestorer',  # Name of the model
```

(continues on next page)

(continued from previous page)

```
    generator=dict(  # Config of the generator
        type='EDSR',  # Type of the generator
        in_channels=3,  # Channel number of inputs
        out_channels=3,  # Channel number of outputs
        mid_channels=64,  # Channel number of intermediate features
        num_blocks=16,  # Block number in the trunk network
        upscale_factor=scale, # Upsampling factor
        res_scale=1,  # Used to scale the residual in residual block
        rgb_mean=(0.4488, 0.4371, 0.4040),  # Image mean in RGB orders
        rgb_std=(1.0, 1.0, 1.0)),  # Image std in RGB orders
    pretrained=None,
    pixel_loss=dict(type='L1Loss', loss_weight=1.0, reduction='mean'))  # Config for␣
↪pixel loss model training and testing settings
```

```
model = dict(
    type='BaseEditModel',  # Name of the model
    generator=dict(  # Config of the generator
        type='EDSRNet',  # Type of the generator
        in_channels=3,  # Channel number of inputs
        out_channels=3,  # Channel number of outputs
        mid_channels=64,  # Channel number of intermediate features
        num_blocks=16,  # Block number in the trunk network
        upscale_factor=scale, # Upsampling factor
        res_scale=1,  # Used to scale the residual in residual block
        rgb_mean=(0.4488, 0.4371, 0.4040),  # Image mean in RGB orders
        rgb_std=(1.0, 1.0, 1.0)),  # Image std in RGB orders
    pixel_loss=dict(type='L1Loss', loss_weight=1.0, reduction='mean')  # Config for␣
↪pixel loss
    train_cfg=dict(),  # Config of training model.
    test_cfg=dict(),  # Config of testing model.
    data_preprocessor=dict(  # The Config to build data preprocessor
        type='EditDataPreprocessor', mean=[0., 0., 0.], std=[255., 255.,
                                                             255.]))
```

We refactor models in MMEdit 1.x. Important modifications are as following.

- The `models` in MMedit 1.x is refactored to five parts: `base_models`, `data_preprocessors`, `editors`, `layers` and `losses`.
- Add `data_preprocessor` module in `models`. Normalization and color space transforms operations are moved from datasets transforms pipelines to data_preprocessor. The data out from the data pipeline is transformed by this module and then fed into the model.

More details of models are shown in *model guides*.

1.26 Migration of Evaluation and Testing Settings

We update evaluation settings in MMEdit 1.x. Important modifications are as following.

- The evaluation field is split to `val_evaluator` and `test_evaluator`. The `interval` is moved to `train_cfg.val_interval`.
- The metrics to evaluation are moved from `test_cfg` to `val_evaluator` and `test_evaluator`.

```
train_cfg = None  # Training config
test_cfg = dict(  # Test config
    metrics=['PSNR'],  # Metrics used during testing
    crop_border=scale)  # Crop border during evaluation

evaluation = dict(  # The config to build the evaluation hook
    interval=5000,  # Evaluation interval
    save_image=True,  # Save images during evaluation
    gpu_collect=True)  # Use gpu collect
```

```
val_evaluator = [
    dict(type='PSNR', crop_border=scale),  # The name of metrics to evaluate
]
test_evaluator = val_evaluator

train_cfg = dict(
    type='IterBasedTrainLoop', max_iters=300000, val_interval=5000)  # Config of train␣
↪loop type
val_cfg = dict(type='ValLoop')  # The name of validation loop type
test_cfg = dict(type='TestLoop')  # The name of test loop type
```

We have merged MMGeneration 1.x into MMEditing. Here is migration of Evaluation and Testing Settings about MMGeneration.

The evaluation field is splited to `val_evaluator` and `test_evaluator`. And it won't support `interval` and `save_best` arguments. The `interval` is moved to `train_cfg.val_interval`, see *the schedule settings* and the `save_best` is moved to `default_hooks.checkpoint.save_best`.

```
evaluation = dict(
    type='GenerativeEvalHook',
    interval=10000,
    metrics=[
        dict(
            type='FID',
            num_images=50000,
            bgr2rgb=True,
            inception_args=dict(type='StyleGAN')),
        dict(type='IS', num_images=50000)
    ],
    best_metric=['fid', 'is'],
    sample_kwargs=dict(sample_model='ema'))
```

```
val_evaluator = dict(
    type='GenEvaluator',
    metrics=[
```

(continues on next page)

(continued from previous page)

```
        dict(
            type='FID',
            prefix='FID-Full-50k',
            fake_nums=50000,
            inception_style='StyleGAN',
            sample_model='orig')
        dict(
            type='IS',
            prefix='IS-50k',
            fake_nums=50000)])
# set best config
default_hooks = dict(
    checkpoint=dict(
        type='CheckpointHook',
        interval=10000,
        by_epoch=False,
        less_keys=['FID-Full-50k/fid'],
        greater_keys=['IS-50k/is'],
        save_optimizer=True,
        save_best=['FID-Full-50k/fid', 'IS-50k/is'],
        rule=['less', 'greater']))
test_evaluator = val_evaluator
```

To evaluate and test the model correctly, we need to set specific loop in `val_cfg` and `test_cfg`.

```
total_iters = 1000000

runner = dict(
    type='DynamicIterBasedRunner',
    is_dynamic_ddp=False,
    pass_training_status=True)
```

```
train_cfg = dict(
    by_epoch=False,  # use iteration based training
    max_iters=1000000,  # max training iteration
    val_begin=1,
    val_interval=10000)  # evaluation interval
val_cfg = dict(type='GenValLoop')  # specific loop in validation
test_cfg = dict(type='GenTestLoop')  # specific loop in testing
```

1.27 Migration of Schedule Settings

We update schedule settings in MMEdit 1.x. Important modifications are as following.

- Now we use `optim_wrapper` field to specify all configuration about the optimization process. And the `optimizer` is a sub field of `optim_wrapper` now.
- The `lr_config` field is removed and we use new `param_scheduler` to replace it.
- The `total_iters` field is moved to `train_cfg` as `max_iters`, `val_cfg` and `test_cfg`, which configure the loop in training, validation and test.

```
optimizers = dict(generator=dict(type='Adam', lr=1e-4, betas=(0.9, 0.999)))  # Config␣
↪used to build optimizer, support all the optimizers in PyTorch whose arguments are␣
↪also the same as those in PyTorch
total_iters = 300000 # Total training iters
lr_config = dict( # Learning rate scheduler config used to register LrUpdater hook
    policy='Step', by_epoch=False, step=[200000], gamma=0.5)  # The policy of scheduler
```

```
optim_wrapper = dict(
    dict(
        type='OptimWrapper',
        optimizer=dict(type='Adam', lr=1e-4),
    )
)  # Config used to build optimizer, support all the optimizers in PyTorch whose␣
↪arguments are also the same as those in PyTorch.
param_scheduler = dict(  # Config of learning policy
    type='MultiStepLR', by_epoch=False, milestones=[200000], gamma=0.5)  # The policy of␣
↪scheduler
train_cfg = dict(
    type='IterBasedTrainLoop', max_iters=300000, val_interval=5000)  # Config of train␣
↪loop type
val_cfg = dict(type='ValLoop')  # The name of validation loop type
test_cfg = dict(type='TestLoop')  # The name of test loop type
```

More details of schedule settings are shown in MMEngine Documents.

1.28 Migration of Data Settings

This section introduces the migration of data settings:

- *Migration of Data Settings*
  - *Data pipelines*
  - *Dataloader*

1.28.1 Data pipelines

We update data pipelines settings in MMEdit 1.x. Important modifications are as following.

- Remove normalization and color space transforms operations. They are moved from datasets transforms pipelines to data_preprocessor.
- The original formatting transforms pipelines `Collect` and `ToTensor` are combined as `PackEditInputs`. More details of data pipelines are shown in *transform guides*.

```
train_pipeline = [  # Training data processing pipeline
    dict(type='LoadImageFromFile',  # Load images from files
        io_backend='disk',  # io backend
        key='lq',  # Keys in results to find corresponding path
        flag='unchanged'),  # flag for reading images
    dict(type='LoadImageFromFile',  # Load images from files
        io_backend='disk',  # io backend
```

(continues on next page)

(continued from previous page)

```
        key='gt',  # Keys in results to find corresponding path
        flag='unchanged'),  # flag for reading images
    dict(type='RescaleToZeroOne', keys=['lq', 'gt']),  # Rescale images from [0, 255] to␣
↪[0, 1]
    dict(type='Normalize',  # Augmentation pipeline that normalize the input images
        keys=['lq', 'gt'],  # Images to be normalized
        mean=[0, 0, 0],  # Mean values
        std=[1, 1, 1],  # Standard variance
        to_rgb=True),  # Change to RGB channel
    dict(type='PairedRandomCrop', gt_patch_size=96),  # Paired random crop
    dict(type='Flip',  # Flip images
        keys=['lq', 'gt'],  # Images to be flipped
        flip_ratio=0.5,  # Flip ratio
        direction='horizontal'),  # Flip direction
    dict(type='Flip',  # Flip images
        keys=['lq', 'gt'],  # Images to be flipped
        flip_ratio=0.5,  # Flip ratio
        direction='vertical'),  # Flip direction
    dict(type='RandomTransposeHW',  # Random transpose h and w for images
        keys=['lq', 'gt'],  # Images to be transposed
        transpose_ratio=0.5  # Transpose ratio
        ),
    dict(type='Collect',  # Pipeline that decides which keys in the data should be␣
↪passed to the model
        keys=['lq', 'gt'],  # Keys to pass to the model
        meta_keys=['lq_path', 'gt_path']), # Meta information keys. In training, meta␣
↪information is not needed
    dict(type='ToTensor',  # Convert images to tensor
        keys=['lq', 'gt'])  # Images to be converted to Tensor
]
test_pipeline = [  # Test pipeline
    dict(
        type='LoadImageFromFile',  # Load images from files
        io_backend='disk',  # io backend
        key='lq',  # Keys in results to find corresponding path
        flag='unchanged'),  # flag for reading images
    dict(
        type='LoadImageFromFile',  # Load images from files
        io_backend='disk',  # io backend
        key='gt',  # Keys in results to find corresponding path
        flag='unchanged'),  # flag for reading images
    dict(type='RescaleToZeroOne', keys=['lq', 'gt']),  # Rescale images from [0, 255] to␣
↪[0, 1]
    dict(
        type='Normalize',  # Augmentation pipeline that normalize the input images
        keys=['lq', 'gt'],  # Images to be normalized
        mean=[0, 0, 0],  # Mean values
        std=[1, 1, 1],  # Standard variance
        to_rgb=True),  # Change to RGB channel
    dict(type='Collect',  # Pipeline that decides which keys in the data should be␣
↪passed to the model
        keys=['lq', 'gt'],  # Keys to pass to the model
```

(continues on next page)

(continued from previous page)

```
        meta_keys=['lq_path', 'gt_path']),  # Meta information keys
    dict(type='ToTensor',  # Convert images to tensor
        keys=['lq', 'gt'])  # Images to be converted to Tensor
]
```

```
train_pipeline = [  # Training data processing pipeline
    dict(type='LoadImageFromFile',  # Load images from files
        key='img',  # Keys in results to find corresponding path
        color_type='color',  # Color type of image
        channel_order='rgb',  # Channel order of image
        imdecode_backend='cv2'),  # decode backend
    dict(type='LoadImageFromFile',  # Load images from files
        key='gt',  # Keys in results to find corresponding path
        color_type='color',  # Color type of image
        channel_order='rgb',  # Channel order of image
        imdecode_backend='cv2'),  # decode backend
    dict(type='SetValues', dictionary=dict(scale=scale)),  # Set value to destination␣
↪keys
    dict(type='PairedRandomCrop', gt_patch_size=96),  # Paired random crop
    dict(type='Flip',  # Flip images
        keys=['lq', 'gt'],  # Images to be flipped
        flip_ratio=0.5,  # Flip ratio
        direction='horizontal'),  # Flip direction
    dict(type='Flip',  # Flip images
        keys=['lq', 'gt'],  # Images to be flipped
        flip_ratio=0.5,  # Flip ratio
        direction='vertical'),  # Flip direction
    dict(type='RandomTransposeHW',  # Random transpose h and w for images
        keys=['lq', 'gt'],  # Images to be transposed
        transpose_ratio=0.5  # Transpose ratio
        ),
    dict(type='PackEditInputs')  # The config of collecting data from current pipeline
]
test_pipeline = [  # Test pipeline
    dict(type='LoadImageFromFile',  # Load images from files
        key='img',  # Keys in results to find corresponding path
        color_type='color',  # Color type of image
        channel_order='rgb',  # Channel order of image
        imdecode_backend='cv2'),  # decode backend
    dict(type='LoadImageFromFile',  # Load images from files
        key='gt',  # Keys in results to find corresponding path
        color_type='color',  # Color type of image
        channel_order='rgb',  # Channel order of image
        imdecode_backend='cv2'),  # decode backend
    dict(type='PackEditInputs')  # The config of collecting data from current pipeline
]
```

1.28.2 Dataloader

We update dataloader settings in MMEdit 1.x. Important modifications are as following.

- The original `data` field is split to `train_dataloader`, `val_dataloader` and `test_dataloader`. This allows us to configure them in fine-grained. For example, you can specify different sampler and batch size during training and test.
- The `samples_per_gpu` is renamed to `batch_size`.
- The `workers_per_gpu` is renamed to `num_workers`.

```
data = dict(
    # train
    samples_per_gpu=16,  # Batch size of a single GPU
    workers_per_gpu=4,  # Worker to pre-fetch data for each single GPU
    drop_last=True,  # Use drop_last in data_loader
    train=dict(  # Train dataset config
        type='RepeatDataset',  # Repeated dataset for iter-based model
        times=1000,  # Repeated times for RepeatDataset
        dataset=dict(
            type=train_dataset_type,  # Type of dataset
            lq_folder='data/DIV2K/DIV2K_train_LR_bicubic/X2_sub',  # Path for lq folder
            gt_folder='data/DIV2K/DIV2K_train_HR_sub',  # Path for gt folder
            ann_file='data/DIV2K/meta_info_DIV2K800sub_GT.txt',  # Path for annotation␣
↪file
            pipeline=train_pipeline,  # See above for train_pipeline
            scale=scale)),  # Scale factor for upsampling
    # val
    val_samples_per_gpu=1,  # Batch size of a single GPU for validation
    val_workers_per_gpu=4,  # Worker to pre-fetch data for each single GPU for validation
    val=dict(
        type=val_dataset_type,  # Type of dataset
        lq_folder='data/val_set5/Set5_bicLRx2',  # Path for lq folder
        gt_folder='data/val_set5/Set5_mod12',  # Path for gt folder
        pipeline=test_pipeline,  # See above for test_pipeline
        scale=scale,  # Scale factor for upsampling
        filename_tmpl='{}'),  # filename template
    # test
    test=dict(
        type=val_dataset_type,  # Type of dataset
        lq_folder='data/val_set5/Set5_bicLRx2',  # Path for lq folder
        gt_folder='data/val_set5/Set5_mod12',  # Path for gt folder
        pipeline=test_pipeline,  # See above for test_pipeline
        scale=scale,  # Scale factor for upsampling
        filename_tmpl='{}'))  # filename template
```

```
dataset_type = 'BasicImageDataset'  # The type of dataset
data_root = 'data'  # Root path of data
train_dataloader = dict(
    batch_size=16,
    num_workers=4,  # The number of workers to pre-fetch data for each single GPU
    persistent_workers=False,  # Whether maintain the workers Dataset instances alive
    sampler=dict(type='InfiniteSampler', shuffle=True),  # The type of data sampler
    dataset=dict(  # Train dataset config
```

(continues on next page)

(continued from previous page)

```
        type=dataset_type,  # Type of dataset
        ann_file='meta_info_DIV2K800sub_GT.txt',  # Path of annotation file
        metainfo=dict(dataset_type='div2k', task_name='sisr'),
        data_root=data_root + '/DIV2K',  # Root path of data
        data_prefix=dict(  # Prefix of image path
            img='DIV2K_train_LR_bicubic/X2_sub', gt='DIV2K_train_HR_sub'),
        filename_tmpl=dict(img='{}', gt='{}'),  # Filename template
        pipeline=train_pipeline))
val_dataloader = dict(
    batch_size=1,
    num_workers=4,  # The number of workers to pre-fetch data for each single GPU
    persistent_workers=False,  # Whether maintain the workers Dataset instances alive
    drop_last=False,  # Whether drop the last incomplete batch
    sampler=dict(type='DefaultSampler', shuffle=False),  # The type of data sampler
    dataset=dict(  # Validation dataset config
        type=dataset_type,  # Type of dataset
        metainfo=dict(dataset_type='set5', task_name='sisr'),
        data_root=data_root + '/Set5',  # Root path of data
        data_prefix=dict(img='LRbicx2', gt='GTmod12'),  # Prefix of image path
        pipeline=test_pipeline))
test_dataloader = val_dataloader
```

1.29 Migration of Distributed Training Settings

We have merged MMGeneration 1.x into MMEditing. Here is migration of Distributed Training Settings about MMGeneration.

In 0.x version, MMGeneration uses `DDPWrapper` and `DynamicRunner` to train static and dynamic model (e.g., PGGAN and StyleGANv2) respectively. In 1.x version, we use `MMSeparateDistributedDataParallel` provided by MMEngine to implement distributed training.

The configuration differences are shown below:

```
# Use DDPWrapper
use_ddp_wrapper = True
find_unused_parameters = False

runner = dict(
    type='DynamicIterBasedRunner',
    is_dynamic_ddp=False)
```

```
model_wrapper_cfg = dict(
    type='MMSeparateDistributedDataParallel',
    broadcast_buffers=False,
    find_unused_parameters=False)
```

```
use_ddp_wrapper = False
find_unused_parameters = False

# Use DynamicRunner
```

(continues on next page)

(continued from previous page)

```
runner = dict(
    type='DynamicIterBasedRunner',
    is_dynamic_ddp=True)
```

```
model_wrapper_cfg = dict(
    type='MMSeparateDistributedDataParallel',
    broadcast_buffers=False,
    find_unused_parameters=True) # set `find_unused_parameters` for dynamic models
```

1.30 Migration of Optimizers

We have merged MMGeneration 1.x into MMEditing. Here is migration of Optimizers about MMGeneration.

In version 0.x, MMGeneration uses PyTorch's native Optimizer, which only provides general parameter optimization. In version 1.x, we use `OptimizerWrapper` provided by MMEngine.

Compared to PyTorch's `Optimizer`, `OptimizerWrapper` supports the following features:

- `OptimizerWrapper.update_params` implement `zero_grad`, `backward` and `step` in a single function.
- Support gradient accumulation automatically.
- Provide a context manager named `OptimizerWrapper.optim_context` to warp the forward process. `optim_context` can automatically call `torch.no_sync` according to current number of updating iteration. In AMP (auto mixed precision) training, `autocast` is called in `optim_context` as well.

For GAN models, generator and discriminator use different optimizer and training schedule. To ensure that the GAN model's function signature of `train_step` is consistent with other models, we use `OptimWrapperDict`, inherited from `OptimizerWrapper`, to wrap the optimizer of the generator and discriminator. To automate this process MMGeneration implement `GenOptimWrapperContructor`. And you should specify this constructor in your config is you want to train GAN model.

The config for the 0.x and 1.x versions are shown below:

```
optimizer = dict(
    generator=dict(type='Adam', lr=0.0001, betas=(0.0, 0.999), eps=1e-6),
    discriminator=dict(type='Adam', lr=0.0004, betas=(0.0, 0.999), eps=1e-6))
```

```
optim_wrapper = dict(
    # Use constructor implemented by MMGeneration
    constructor='GenOptimWrapperConstructor',
    generator=dict(optimizer=dict(type='Adam', lr=0.0002, betas=(0.0, 0.999), eps=1e-6)),
    discriminator=dict(
        optimizer=dict(type='Adam', lr=0.0004, betas=(0.0, 0.999), eps=1e-6)))
```

  Note that, in the 1.x, MMGeneration uses `OptimWrapper` to realize gradient accumulation. This make the config of `discriminator_steps` (training trick for updating the generator once after multiple updates of the discriminator) and gradient accumulation different between 0.x and 1.x version.

- In 0.x version, we use `disc_steps`, `gen_steps` and `batch_accumulation_steps` in configs. `disc_steps` and `batch_accumulation_steps` are counted by the number of calls of `train_step` (is also the number of data reads from the dataloader). Therefore the number of consecutive updates of the discriminator is `disc_steps // batch_accumulation_steps`. And for generators, `gen_steps` is the number of times the generator actually updates continuously.

- In 1.x version, we use `discriminator_steps`, `generator_steps` and `accumulative_counts` in configs. `discriminator_steps` and `generator_steps` are the number of consecutive updates to itself before updating other modules.

Take config of BigGAN-128 as example.

```
model = dict(
    type='BasiccGAN',
    generator=dict(
        type='BigGANGenerator',
        output_scale=128,
        noise_size=120,
        num_classes=1000,
        base_channels=96,
        shared_dim=128,
        with_shared_embedding=True,
        sn_eps=1e-6,
        init_type='ortho',
        act_cfg=dict(type='ReLU', inplace=True),
        split_noise=True,
        auto_sync_bn=False),
    discriminator=dict(
        type='BigGANDiscriminator',
        input_scale=128,
        num_classes=1000,
        base_channels=96,
        sn_eps=1e-6,
        init_type='ortho',
        act_cfg=dict(type='ReLU', inplace=True),
        with_spectral_norm=True),
    gan_loss=dict(type='GANLoss', gan_type='hinge'))

# continuous update discriminator for `disc_steps // batch_accumulation_steps = 8 // 8 =␣
↪1` times
# continuous update generator for `gen_steps = 1` times
# generators and discriminators perform `batch_accumulation_steps = 8` times gradient␣
↪accumulations before each update
train_cfg = dict(
    disc_steps=8, gen_steps=1, batch_accumulation_steps=8, use_ema=True)
```

```
model = dict(
    type='BigGAN',
    num_classes=1000,
    data_preprocessor=dict(type='GANDataPreprocessor'),
    generator=dict(
        type='BigGANGenerator',
        output_scale=128,
        noise_size=120,
        num_classes=1000,
        base_channels=96,
        shared_dim=128,
        with_shared_embedding=True,
        sn_eps=1e-6,
        init_type='ortho',
```

(continues on next page)

(continued from previous page)

```
        act_cfg=dict(type='ReLU', inplace=True),
        split_noise=True,
        auto_sync_bn=False),
    discriminator=dict(
        type='BigGANDiscriminator',
        input_scale=128,
        num_classes=1000,
        base_channels=96,
        sn_eps=1e-6,
        init_type='ortho',
        act_cfg=dict(type='ReLU', inplace=True),
        with_spectral_norm=True),
    # continuous update discriminator for `discriminator_steps = 1` times
    # continuous update generator for `generator_steps = 1` times
    generator_steps=1,
    discriminator_steps=1)

optim_wrapper = dict(
    constructor='GenOptimWrapperConstructor',
    generator=dict(
        # generator perform `accumulative_counts = 8` times gradient accumulations before␣
↪each update
        accumulative_counts=8,
        optimizer=dict(type='Adam', lr=0.0001, betas=(0.0, 0.999), eps=1e-6)),
    discriminator=dict(
        # discriminator perform `accumulative_counts = 8` times gradient accumulations␣
↪before each update
        accumulative_counts=8,
        optimizer=dict(type='Adam', lr=0.0004, betas=(0.0, 0.999), eps=1e-6)))
```

1.31 Migration of Visualization

In 0.x, MMEditing use `VisualizationHook` to visualize results in training process. In 1.x version, we unify the function of those hooks into `BasicVisualizationHook` / `GenVisualizationHook`. Additionally, follow the design of MMEngine, we implement `ConcatImageVisualizer` / `GenVisualizer` and a group of `VisBackend` to draw and save the visualization results.

```
visual_config = dict(
    type='VisualizationHook',
    output_dir='visual',
    interval=1000,
    res_name_list=['gt_img', 'masked_img', 'fake_res', 'fake_img'],
)
```

```
vis_backends = [dict(type='LocalVisBackend')]
visualizer = dict(
    type='ConcatImageVisualizer',
    vis_backends=vis_backends,
    fn_key='gt_path',
    img_keys=['gt_img', 'input', 'pred_img'],
```

(continues on next page)

(continued from previous page)

```
    bgr2rgb=True)
custom_hooks = [dict(type='BasicVisualizationHook', interval=1)]
```

To learn more about the visualization function, please refers to *this tutorial*.

1.32 Migration of AMP Training

In 0.x, MMEditing do not support AMP training for the entire forward process. Instead, users must use `auto_fp16` decorator to warp the specific submodule and convert the parameter of submodule to fp16. This allows for fine-grained control of the model parameters, but is more cumbersome to use. In addition, users need to handle operations such as scaling of the loss function during the training process by themselves.

In 1.x version, MMEditing use `AmpOptimWrapper` provided by MMEngine. In `AmpOptimWrapper.update_params`, gradient scaling and `GradScaler` updating is automatically performed. And in `optim_context` context manager, `auto_cast` is applied to the entire forward process.

Specifically, the difference between the 0.x and 1.x is as follows:

```
# config
runner = dict(fp16_loss_scaler=dict(init_scale=512))
```

```
# code
import torch.nn as nn
from mmedit.models.builder import build_model
from mmedit.core.runners.fp16_utils import auto_fp16


class DemoModule(nn.Module):
    def __init__(self, cfg):
        self.net = build_model(cfg)

    @auto_fp16
    def forward(self, x):
        return self.net(x)

class DemoModel(nn.Module):

    def __init__(self, cfg):
        super().__init__(self)
        self.demo_network = DemoModule(cfg)

    def train_step(self,
                   data_batch,
                   optimizer,
                   ddp_reducer=None,
                   loss_scaler=None,
                   use_apex_amp=False,
                   running_status=None):
        # get data from data_batch
        inputs = data_batch['img']
        output = self.demo_network(inputs)
```

(continues on next page)

(continued from previous page)

```
        optimizer.zero_grad()
        loss, log_vars = self.get_loss(data_dict_)

        if ddp_reducer is not None:
            ddp_reducer.prepare_for_backward(_find_tensors(loss_disc))

        if loss_scaler:
            # add support for fp16
            loss_scaler.scale(loss_disc).backward()
        elif use_apex_amp:
            from apex import amp
            with amp.scale_loss(loss_disc, optimizer,
                    loss_id=0) as scaled_loss_disc:
                scaled_loss_disc.backward()
        else:
            loss_disc.backward()

        if loss_scaler:
            loss_scaler.unscale_(optimizer)
            loss_scaler.step(optimizer)
        else:
            optimizer.step()
```

```
# config
optim_wrapper = dict(
    constructor='OptimWrapperConstructor',
    generator=dict(
        accumulative_counts=8,
        optimizer=dict(type='Adam', lr=0.0001, betas=(0.0, 0.999), eps=1e-06),
        type='AmpOptimWrapper',  # use amp wrapper
        loss_scale='dynamic'),
    discriminator=dict(
        accumulative_counts=8,
        optimizer=dict(type='Adam', lr=0.0004, betas=(0.0, 0.999), eps=1e-06),
        type='AmpOptimWrapper',  # use amp wrapper
        loss_scale='dynamic'))
```

```
# code
import torch.nn as nn
from mmedit.registry import MODULES
from mmengine.model import BaseModel


class DemoModule(nn.Module):
    def __init__(self, cfg):
        self.net = MODULES.build(cfg)

    def forward(self, x):
        return self.net(x)
```

(continues on next page)

(continued from previous page)

```
class DemoModel(BaseModel):
    def __init__(self, cfg):
        super().__init__(self)
        self.demo_network = DemoModule(cfg)

    def train_step(self, data, optim_wrapper):
        # get data from data_batch
        data = self.data_preprocessor(data, True)
        inputs = data['inputs']

        with optim_wrapper.optim_context(self.discriminator):
            output = self.demo_network(inputs)
        loss_dict = self.get_loss(output)
        # use parse_loss provide by `BaseModel`
        loss, log_vars = self.parse_loss(loss_dict)
        optimizer_wrapper.update_params(loss)

        return log_vars
```

To avoid user modifications to the configuration file, MMEditing provides the `--amp` option in `train.py`, which allows the user to start AMP training without modifying the configuration file. Users can start AMP training by following command:

```
bash tools/dist_train.sh CONFIG GPUS --amp

# for slurm users
bash tools/slurm_train.sh PARTITION JOB_NAME CONFIG WORK_DIR --amp
```

1.33 mmedit.apis

1.33.1 APIS

matting_inference	Inference image(s) with the model.
inpainting_inference	Inference image with the model.
restoration_inference	Inference image with the model.
restoration_video_inference	Inference image with the model.
restoration_face_inference	Inference image with the model.
video_interpolation_inference	Inference image with the model.
init_model	Initialize a model from config file.
delete_cfg	Delete key from config object.
set_random_seed	Set random seed.
sample_conditional_model	Sampling from conditional models.
sample_unconditional_model	Sampling from unconditional models.
sample_img2img_model	Sampling from translation models.

mmedit.apis.matting_inference

mmedit.apis.**matting_inference**(*model*, *img*, *trimap*)

Inference image(s) with the model.

Parameters

- **model** (*nn.Module*) – The loaded model.
- **img** (*str*) – Image file path.
- **trimap** (*str*) – Trimap file path.

Returns The predicted alpha matte.

Return type np.ndarray

mmedit.apis.inpainting_inference

mmedit.apis.**inpainting_inference**(*model*, *masked_img*, *mask*)

Inference image with the model.

Parameters

- **model** (*nn.Module*) – The loaded model.
- **masked_img** (*str*) – File path of image with mask.
- **mask** (*str*) – Mask file path.

Returns The predicted inpainting result.

Return type Tensor

mmedit.apis.restoration_inference

mmedit.apis.**restoration_inference**(*model*, *img*, *ref=None*)

Inference image with the model.

Parameters

- **model** (*nn.Module*) – The loaded model.
- **img** (*str*) – File path of input image.
- **ref** (*str | None*) – File path of reference image. Default: None.

Returns The predicted restoration result.

Return type Tensor

mmedit.apis.restoration_video_inference

mmedit.apis.**restoration_video_inference**(*model*, *img_dir*, *window_size*, *start_idx*, *filename_tmpl*, *max_seq_len=None*)

Inference image with the model.

Parameters

- **model** (*nn.Module*) – The loaded model.
- **img_dir** (*str*) – Directory of the input video.
- **window_size** (*int*) – The window size used in sliding-window framework. This value should be set according to the settings of the network. A value smaller than 0 means using recurrent framework.
- **start_idx** (*int*) – The index corresponds to the first frame in the sequence.
- **filename_tmpl** (*str*) – Template for file name.
- **max_seq_len** (*int | None*) – The maximum sequence length that the model processes. If the sequence length is larger than this number, the sequence is split into multiple segments. If it is None, the entire sequence is processed at once.

Returns The predicted restoration result.

Return type Tensor

mmedit.apis.restoration_face_inference

mmedit.apis.**restoration_face_inference**(*model*, *img*, *upscale_factor=1*, *face_size=1024*)

Inference image with the model.

Parameters

- **model** (*nn.Module*) – The loaded model.
- **img** (*str*) – File path of input image.
- **upscale_factor** (*int, optional*) – The number of times the input image is upsampled. Default: 1.
- **face_size** (*int, optional*) – The size of the cropped and aligned faces. Default: 1024.

Returns The predicted restoration result.

Return type Tensor

mmedit.apis.video_interpolation_inference

mmedit.apis.**video_interpolation_inference**(*model*, *input_dir*, *output_dir*, *start_idx=0*, *end_idx=None*, *batch_size=4*, *fps_multiplier=0*, *fps=0*, *filename_tmpl='{:08d}.png'*)

Inference image with the model.

Parameters

- **model** (*nn.Module*) – The loaded model.
- **input_dir** (*str*) – Directory of the input video.
- **output_dir** (*str*) – Directory of the output video.

- **start_idx** (*int*) – The index corresponding to the first frame in the sequence. Default: 0
- **end_idx** (*int | None*) – The index corresponding to the last interpolated frame in the sequence. If it is None, interpolate to the last frame of video or sequence. Default: None
- **batch_size** (*int*) – Batch size. Default: 4
- **fps_multiplier** (*float*) – multiply the fps based on the input video. Default: 0.
- **fps** (*float*) – frame rate of the output video. Default: 0.
- **filename_tmpl** (*str*) – template of the file names. Default: '{:08d}.png'

mmedit.apis.init_model

mmedit.apis.**init_model**(*config*, *checkpoint=None*, *device='cuda:0'*)

Initialize a model from config file.

Parameters

- **config** (str or mmengine.Config) – Config file path or the config object.
- **checkpoint** (*str, optional*) – Checkpoint path. If left as None, the model will not load any weights.
- **device** (*str*) – Which device the model will deploy. Default: 'cuda:0'.

Returns The constructed model.

Return type nn.Module

mmedit.apis.delete_cfg

mmedit.apis.**delete_cfg**(*cfg*, *key='init_cfg'*)

Delete key from config object.

Parameters

- **cfg** (str or mmengine.Config) – Config object.
- **key** (*str*) – Which key to delete.

mmedit.apis.set_random_seed

mmedit.apis.**set_random_seed**(*seed*, *deterministic=False*, *use_rank_shift=True*)

Set random seed.

In this function, we just modify the default behavior of the similar function defined in MMCV.

Parameters

- **seed** (*int*) – Seed to be used.
- **deterministic** (*bool*) – Whether to set the deterministic option for CUDNN backend, i.e., set *torch.backends.cudnn.deterministic* to True and *torch.backends.cudnn.benchmark* to False. Default: False.
- **rank_shift** (*bool*) – Whether to add rank number to the random seed to have different random seed in different threads. Default: True.

mmedit.apis.sample_conditional_model

mmedit.apis.**sample_conditional_model**(*model*, *num_samples=16*, *num_batches=4*, *sample_model='ema'*, *label=None*, ***kwargs*)

Sampling from conditional models.

Parameters

- **model** (*nn.Module*) – Conditional models in MMGeneration.
- **num_samples** (*int, optional*) – The total number of samples. Defaults to 16.
- **num_batches** (*int, optional*) – The number of batch size for inference. Defaults to 4.
- **sample_model** (*str, optional*) – Which model you want to use. [‘ema’, ‘orig’]. Defaults to ‘ema’.
- **label** (*int | torch.Tensor | list[int], optional*) – Labels used to generate images. Default to None.,

Returns Generated image tensor.

Return type Tensor

mmedit.apis.sample_unconditional_model

mmedit.apis.**sample_unconditional_model**(*model*, *num_samples=16*, *num_batches=4*, *sample_model='ema'*, ***kwargs*)

Sampling from unconditional models.

Parameters

- **model** (*nn.Module*) – Unconditional models in MMGeneration.
- **num_samples** (*int, optional*) – The total number of samples. Defaults to 16.
- **num_batches** (*int, optional*) – The number of batch size for inference. Defaults to 4.
- **sample_model** (*str, optional*) – Which model you want to use. [‘ema’, ‘orig’]. Defaults to ‘ema’.

Returns Generated image tensor.

Return type Tensor

mmedit.apis.sample_img2img_model

mmedit.apis.**sample_img2img_model**(*model*, *image_path*, *target_domain=None*, ***kwargs*)

Sampling from translation models.

Parameters

- **model** (*nn.Module*) – The loaded model.
- **image_path** (*str*) – File path of input image.
- **style** (*str*) – Target style of output image.

Returns Translated image tensor.

Return type Tensor

1.34 mmedit.datasets

AdobeComp1kDataset	Adobe composition-1k dataset.
BasicImageDataset	BasicImageDataset for open source projects in OpenMMLab/MMEditing.
BasicFramesDataset	BasicFramesDataset for open source projects in OpenMMLab/MMEditing.
BasicConditionalDataset	Custom dataset for conditional GAN.
UnpairedImageDataset	General unpaired image folder dataset for image generation.
PairedImageDataset	General paired image folder dataset for image generation.
ImageNet	ImageNet Dataset.
CIFAR10	CIFAR10 Dataset.
GrowScaleImgDataset	Grow Scale Unconditional Image Dataset.

1.34.1 AdobeComp1kDataset

class mmedit.datasets.**AdobeComp1kDataset**(*ann_file: str = ''*, *metainfo: Optional[dict] = None*, *data_root: str = ''*, *data_prefix: dict = {'img_path': ''}*, *filter_cfg: Optional[dict] = None*, *indices: Optional[Union[int, Sequence[int]]] = None*, *serialize_data: bool = True*, *pipeline: List[Union[dict, Callable]] = []*, *test_mode: bool = False*, *lazy_init: bool = False*, *max_refetch: int = 1000*)

Adobe composition-1k dataset.

The dataset loads (alpha, fg, bg) data and apply specified transforms to the data. You could specify whether composite merged image online or load composited merged image in pipeline.

Example for online comp-1k dataset:

```
[
    {
        "alpha": 'alpha/000.png',
        "fg": 'fg/000.png',
        "bg": 'bg/000.png'
    },
    {
        "alpha": 'alpha/001.png',
        "fg": 'fg/001.png',
        "bg": 'bg/001.png'
    },
]
```

Example for offline comp-1k dataset:

```
[
    {
        "alpha": 'alpha/000.png',
        "merged": 'merged/000.png',
        "fg": 'fg/000.png',
```

(continues on next page)

(continued from previous page)

```
        "bg": 'bg/000.png'
    },
    {
        "alpha": 'alpha/001.png',
        "merged": 'merged/001.png',
        "fg": 'fg/001.png',
        "bg": 'bg/001.png'
    },
]
```

Parameters

- **ann_file** (*str*) – Annotation file path. Defaults to ‘’.
- **data_root** (*str, optional*) – The root directory for `data_prefix` and `ann_file`. Defaults to None.
- **pipeline** (*list, optional*) – Processing pipeline. Defaults to [].
- **test_mode** (*bool, optional*) – `test_mode=True` means in test phase. Defaults to False.
- ****kwargs** – Other arguments passed to `mmengine.dataset.BaseDataset`.

Examples

See unit-tests TODO: Move some codes in unittest here

load_data_list() → List[dict]

Load annotations from an annotation file named as `self.ann_file`

In order to be compoatible to both new and old annotation format, we copy implementations from mmengine and do some modificatoins.

Returns A list of annotation.

Return type list[dict]

parse_data_info(*raw_data_info: dict*) → Union[dict, List[dict]]

Join data_root to each path in data_info.

1.34.2 BasicImageDataset

class mmedit.datasets.**BasicImageDataset**(*ann_file: str = ''*, *metainfo: Optional[dict] = None*, *data_root: Optional[str] = None*, *data_prefix: dict = {'img': ''}*, *pipeline: List[Union[dict, Callable]] = []*, *test_mode: bool = False*, *filename_tmpl: dict = {}*, *search_key: Optional[str] = None*, *file_client_args: Optional[dict] = None*, *img_suffix: Optional[Union[str, Tuple[str]]] = ('.jpg', '.JPG', '.jpeg', '.JPEG', '.png', '.PNG', '.ppm', '.PPM', '.bmp', '.BMP', '.tif', '.TIF', '.tiff', '.TIFF')*, *recursive: bool = False*, ***kwards*)

BasicImageDataset for open source projects in OpenMMLab/MMEditing.

This dataset is designed for low-level vision tasks with image, such as super-resolution and inpainting.

The annotation file is optional.

If use annotation file, the annotation format can be shown as follows.

```
Case 1 (CelebA-HQ):

    000001.png
    000002.png

Case 2 (DIV2K):

    0001_s001.png (480,480,3)
    0001_s002.png (480,480,3)
    0001_s003.png (480,480,3)
    0002_s001.png (480,480,3)
    0002_s002.png (480,480,3)

Case 3 (Vimeo90k):

    00001/0266 (256, 448, 3)
    00001/0268 (256, 448, 3)
```

Parameters

- **ann_file** (*str*) – Annotation file path. Defaults to ‘’.
- **metainfo** (*dict, optional*) – Meta information for dataset, such as class information. Defaults to None.
- **data_root** (*str, optional*) – The root directory for `data_prefix` and `ann_file`. Defaults to None.
- **data_prefix** (*dict, optional*) – Prefix for training data. Defaults to dict(img=None, ann=None).
- **pipeline** (*list, optional*) – Processing pipeline. Defaults to [].
- **test_mode** (*bool, optional*) – `test_mode=True` means in test phase. Defaults to False.
- **filename_tmpl** (*dict*) – Template for each filename. Note that the template excludes the file extension. Default: dict().
- **search_key** (*str*) – The key used for searching the folder to get data_list. Default: ‘gt’.
- **file_client_args** (*dict, optional*) – Arguments to instantiate a FileClient. See `mmengine.fileio.FileClient` for details. Default: None.
- **suffix** (*str or tuple[str], optional*) – File suffix that we are interested in. Default: None.
- **recursive** (*bool*) – If set to True, recursively scan the directory. Default: False.

Note: Assume the file structure as the following:

```
mmediting (root)
├── mmedit
├── tools
├── configs
├── data
```

(continues on next page)

(continued from previous page)

```
|   ├── DIV2K
|   |   ├── DIV2K_train_HR
|   |   |   ├── image.png
|   |   ├── DIV2K_train_LR_bicubic
|   |   |   ├── X2
|   |   |   ├── X3
|   |   |   ├── X4
|   |   |   |   ├── image_x4.png
|   |   ├── DIV2K_valid_HR
|   |   ├── DIV2K_valid_LR_bicubic
|   |   |   ├── X2
|   |   |   ├── X3
|   |   |   ├── X4
|   ├── places
|   |   ├── test_set
|   |   ├── train_set
|   |   ├── meta
|   |   |   ├── Places365_train.txt
|   |   |   ├── Places365_val.txt
```

Examples

Case 1: Loading DIV2K dataset for training a SISR model.

```
dataset = BasicImageDataset(
    ann_file='',
    metainfo=dict(
        dataset_type='div2k',
        task_name='sisr'),
    data_root='data/DIV2K',
    data_prefix=dict(
        gt='DIV2K_train_HR', img='DIV2K_train_LR_bicubic/X4'),
    filename_tmpl=dict(img='{}_x4', gt='{}'),
    pipeline=[])
```

Case 2: Loading places dataset for training an inpainting model.

```
dataset = BasicImageDataset(
    ann_file='meta/Places365_train.txt',
    metainfo=dict(
        dataset_type='places365',
        task_name='inpainting'),
    data_root='data/places',
    data_prefix=dict(gt='train_set'),
    pipeline=[])
```

load_data_list() → List[dict]

Load data list from folder or annotation file.

Returns A list of annotation.

Return type list[dict]

1.34.3 BasicFramesDataset

class mmedit.datasets.**BasicFramesDataset**(*ann_file: str = ''*, *metainfo: Optional[dict] = None*, *data_root: Optional[str] = None*, *data_prefix: dict = {'img': ''}*, *pipeline: List[Union[dict, Callable]] = []*, *test_mode: bool = False*, *filename_tmpl: dict = {}*, *search_key: Optional[str] = None*, *file_client_args: Optional[str] = None*, *depth: int = 1*, *num_input_frames: Optional[int] = None*, *num_output_frames: Optional[int] = None*, *fixed_seq_len: Optional[int] = None*, *load_frames_list: dict = {}*, ***kwargs*)

BasicFramesDataset for open source projects in OpenMMLab/MMEditing.

This dataset is designed for low-level vision tasks with frames, such as video super-resolution and video frame interpolation.

The annotation file is optional.

If use annotation file, the annotation format can be shown as follows.

```
Case 1 (Vid4):

    calendar 41
    city 34
    foliage 49
    walk 47

Case 2 (REDS):

    000/00000000.png (720, 1280, 3)
    000/00000001.png (720, 1280, 3)

Case 3 (Vimeo90k):

    00001/0266 (256, 448, 3)
    00001/0268 (256, 448, 3)
```

Parameters

- **ann_file** (*str*) – Annotation file path. Defaults to ‘’.
- **metainfo** (*dict, optional*) – Meta information for dataset, such as class information. Defaults to None.
- **data_root** (*str, optional*) – The root directory for `data_prefix` and `ann_file`. Defaults to None.
- **data_prefix** (*dict, optional*) – Prefix for training data. Defaults to dict(img=”, gt=”).
- **pipeline** (*list, optional*) – Processing pipeline. Defaults to [].
- **test_mode** (*bool, optional*) – `test_mode=True` means in test phase. Defaults to False.
- **filename_tmpl** (*str*) – Template for each filename. Note that the template excludes the file extension. Default: ‘{}’.
- **search_key** (*str*) – The key used for searching the folder to get data_list. Default: ‘gt’.
- **file_client_args** (*dict, optional*) – Arguments to instantiate a FileClient. See `mmengine.fileio.FileClient` for details. Default: None.

- **depth** (*int*) – The depth of path. Default: 1
- **num_input_frames** (*None | int*) – Number of input frames. Default: None.
- **num_output_frames** (*None | int*) – Number of output frames. Default: None.
- **fixed_seq_len** (*None | int*) – The fixed sequence length. If None, BasicFramesDataset will obtain the length of each sequence. Default: None.
- **load_frames_list** (*dict*) – Load frames list for each key. Default: dict().

Examples

Assume the file structure as the following:

mmediting (root) ├── mmedit ├── tools ├── configs ├── data │ ├── Vid4 │ │ ├── BIx4 │ │ │ ├── city │ │ │ │ ├── img1.png │ │ ├── GT │ │ │ ├── city │ │ │ │ ├── img1.png │ │ ├── meta_info_Vid4_GT.txt │ ├── places │ │ ├── sequences │ │ │ ├── 00001 │ │ │ │ ├── 0389 │ │ │ │ │ ├── img1.png │ │ │ │ │ ├── img2.png │ │ │ │ │ ├── img3.png │ │ ├── tri_trainlist.txt

Case 1: Loading Vid4 dataset for training a VSR model.

```
dataset = BasicFramesDataset(
    ann_file='meta_info_Vid4_GT.txt',
    metainfo=dict(dataset_type='vid4', task_name='vsr'),
    data_root='data/Vid4',
    data_prefix=dict(img='BIx4', gt='GT'),
    pipeline=[],
    depth=2,
    num_input_frames=5)
```

Case 2: Loading Vimeo90k dataset for training a VFI model.

```
dataset = BasicFramesDataset(
    ann_file='tri_trainlist.txt',
    metainfo=dict(dataset_type='vimeo90k', task_name='vfi'),
    data_root='data/vimeo-triplet',
    data_prefix=dict(img='sequences', gt='sequences'),
    pipeline=[],
    depth=2,
    load_frames_list=dict(
        img=['img1.png', 'img3.png'], gt=['img2.png']))
```

See more details in unittest

tests/test_datasets/test_base_frames_dataset.py TestFramesDatasets().test_version_1_method()

load_data_list() → List[dict]

Load data list from folder or annotation file.

Returns A list of annotation.

Return type list[dict]

1.34.4 BasicConditionalDataset

class mmedit.datasets.**BasicConditionalDataset**(*ann_file: str = ''*, *metainfo: Optional[dict] = None*, *data_root: str = ''*, *data_prefix: Union[str, dict] = ''*, *extensions: Sequence[str] = ('.jpg', '.jpeg', '.png', '.ppm', '.bmp', '.pgm', '.tif')*, *lazy_init: bool = False*, *classes: Optional[Union[str, Sequence[str]]] = None*, ***kwargs*)

Custom dataset for conditional GAN. This class is the combination of *BaseDataset* (https://github.com/open-mmlab/mmclassification/blob/1.x/mmcls/datasets/base_dataset.py) # noqa and *CustomDataset* (https://github.com/open- mmlab/mmclassification/blob/1.x/mmcls/datasets/custom.py). # noqa.

The dataset supports two kinds of annotation format.

1. An annotation file is provided, and each line indicates a sample:

   The sample files:

   ```
   data_prefix/
   ├── folder_1
   │   ├── xxx.png
   │   ├── xxy.png
   │   └── ...
   └── folder_2
       ├── 123.png
       ├── nsdf3.png
       └── ...
   ```

   The annotation file (the first column is the image path and the second column is the index of category):

   ```
   folder_1/xxx.png 0
   folder_1/xxy.png 1
   folder_2/123.png 5
   folder_2/nsdf3.png 3
   ...
   ```

   Please specify the name of categories by the argument `classes` or `metainfo`.

2. The samples are arranged in the specific way:

   ```
   data_prefix/
   ├── class_x
   │   ├── xxx.png
   │   ├── xxy.png
   │   └── ...
   │       └── xxz.png
   └── class_y
       ├── 123.png
       ├── nsdf3.png
       ├── ...
       └── asd932_.png
   ```

If the `ann_file` is specified, the dataset will be generated by the first way, otherwise, try the second way.

Parameters

- **ann_file** (*str*) – Annotation file path. Defaults to ‘’.

- **metainfo** (*dict, optional*) – Meta information for dataset, such as class information. Defaults to None.
- **data_root** (*str*) – The root directory for `data_prefix` and `ann_file`. Defaults to ''.
- **data_prefix** (*str | dict*) – Prefix for the data. Defaults to ''.
- **extensions** (*Sequence[str]*) – A sequence of allowed extensions. Defaults to ('.jpg', '.jpeg', '.png', '.ppm', '.bmp', '.pgm', '.tif').
- **lazy_init** (*bool*) – Whether to load annotation during instantiation. In some cases, such as visualization, only the meta information of the dataset is needed, which is not necessary to load annotation file. `Basedataset` can skip load annotations to save time by set `lazy_init=False`. Defaults to False.
- ****kwargs** – Other keyword arguments in `BaseDataset`.

property CLASSES

Return all categories names.

property class_to_idx

Map mapping class name to class index.

Returns mapping from class name to class index.

Return type dict

extra_repr() → List[str]

The extra repr information of the dataset.

full_init()

Load annotation file and set `BaseDataset._fully_initialized` to True.

get_cat_ids(*idx: int*) → List[int]

Get category id by index.

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

Returns Image category of specified index.

Return type cat_ids (List[int])

get_gt_labels()

Get all ground-truth labels (categories).

Returns categories for all images.

Return type np.ndarray

property img_prefix

The prefix of images.

is_valid_file(*filename: str*) → bool

Check if a file is a valid sample.

load_data_list()

Load image paths and gt_labels.

1.34.5 UnpairedImageDataset

class mmedit.datasets.**UnpairedImageDataset**(*data_root*, *pipeline*, *io_backend: Optional[str] = None*, *test_mode=False*, *domain_a='A'*, *domain_b='B'*)

General unpaired image folder dataset for image generation.

It assumes that the training directory of images from domain A is '/path/to/data/trainA', and that from domain B is '/path/to/data/trainB', respectively. '/path/to/data' can be initialized by args 'dataroot'. During test time, the directory is '/path/to/data/testA' and '/path/to/data/testB', respectively.

Parameters

- **dataroot** (str | Path) – Path to the folder root of unpaired images.
- **pipeline** (*List[dict | callable]*) – A sequence of data transformations.
- **io_backend** (*str, optional*) – The storage backend type. Options are "disk", "ceph", "memcached", "lmdb", "http" and "petrel". Default: None.
- **test_mode** (*bool*) – Store *True* when building test dataset. Default: *False*.
- **domain_a** (*str, optional*) – Domain of images in trainA / testA. Defaults to 'A'.
- **domain_b** (*str, optional*) – Domain of images in trainB / testB. Defaults to 'B'.

get_data_info(*idx*) → dict

Get annotation by index and automatically call `full_init` if the dataset has not been fully initialized.

Parameters **idx** (*int*) – The index of data.

Returns The idx-th annotation of the dataset.

Return type dict

load_data_list()

Load the data list.

Returns The data info list of source and target domain.

Return type list

scan_folder(*path*)

Obtain image path list (including sub-folders) from a given folder.

Parameters **path** (str | Path) – Folder path.

Returns Image list obtained from the given folder.

Return type list[str]

1.34.6 PairedImageDataset

class mmedit.datasets.**PairedImageDataset**(*data_root*, *pipeline*, *io_backend: Optional[str] = None*, *test_mode=False*, *test_dir='test'*)

General paired image folder dataset for image generation.

It assumes that the training directory is '/path/to/data/train'. During test time, the directory is '/path/to/data/test'. '/path/to/data' can be initialized by args 'dataroot'. Each sample contains a pair of images concatenated in the w dimension (A|B).

Parameters

- **dataroot** (str | Path) – Path to the folder root of paired images.
- **pipeline** (*List[dict | callable]*) – A sequence of data transformations.
- **test_mode** (*bool*) – Store *True* when building test dataset. Default: *False*.
- **test_dir** (*str*) – Subfolder of dataroot which contain test images. Default: 'test'.

load_data_list()

Load paired image paths.

Returns List that contains paired image paths.

Return type list[dict]

scan_folder(*path*)

Obtain image path list (including sub-folders) from a given folder.

Parameters **path** (str | Path) – Folder path.

Returns Image list obtained from the given folder.

Return type list[str]

1.34.7 ImageNet

class mmedit.datasets.**ImageNet**(*ann_file: str = ''*, *metainfo: Optional[dict] = None*, *data_root: str = ''*, *data_prefix: Union[str, dict] = ''*, ***kwargs*)

ImageNet Dataset.

The dataset supports two kinds of annotation format. More details can be found in CustomDataset.

Parameters

- **ann_file** (*str*) – Annotation file path. Defaults to ''.
- **metainfo** (*dict, optional*) – Meta information for dataset, such as class information. Defaults to None.
- **data_root** (*str*) – The root directory for data_prefix and ann_file. Defaults to ''.
- **data_prefix** (*str | dict*) – Prefix for training data. Defaults to ''.
- ****kwargs** – Other keyword arguments in CustomDataset and BaseDataset.

1.34.8 CIFAR10

class mmedit.datasets.**CIFAR10**(*data_prefix: str*, *test_mode: bool*, *metainfo: Optional[dict] = None*, *data_root: str = ''*, *download: bool = True*, ***kwargs*)

CIFAR10 Dataset.

This implementation is modified from https://github.com/pytorch/vision/blob/master/torchvision/datasets/cifar.py

Parameters

- **data_prefix** (*str*) – Prefix for data.
- **test_mode** (*bool*) – test_mode=True means in test phase. It determines to use the training set or test set.

- **metainfo** (*dict, optional*) – Meta information for dataset, such as categories information. Defaults to None.
- **data_root** (*str*) – The root directory for `data_prefix`. Defaults to ''.
- **download** (*bool*) – Whether to download the dataset if not exists. Defaults to True.
- ****kwargs** – Other keyword arguments in `BaseDataset`.

extra_repr() → List[str]

The extra repr information of the dataset.

load_data_list()

Load images and ground truth labels.

1.34.9 GrowScaleImgDataset

class mmedit.datasets.**GrowScaleImgDataset**(*data_roots: dict*, *pipeline*, *len_per_stage=1000000*, *gpu_samples_per_scale=None*, *gpu_samples_base=32*, *io_backend: Optional[str] = None*, *file_lists: Optional[Union[str, dict]] = None*, *test_mode=False*)

Grow Scale Unconditional Image Dataset.

This dataset is similar with `UnconditionalImageDataset`, but offer more dynamic functionalities for the supporting complex algorithms, like PGGAN.

Highlight functionalities:

1. Support growing scale dataset. The motivation is to decrease data pre-processing load in CPU. In this dataset, you can provide `imgs_roots` like:

```
{'64': 'path_to_64x64_imgs',
 '512': 'path_to_512x512_imgs'}
```

   Then, in training scales lower than 64x64, this dataset will set `self.imgs_root` as 'path_to_64x64_imgs';

2. Offer `samples_per_gpu` according to different scales. In this dataset, `self.samples_per_gpu` will help runner to know the updated batch size.

Basically, This dataset contains raw images for training unconditional GANs. Given a root dir, we will recursively find all images in this root. The transformation on data is defined by the pipeline.

Parameters

- **imgs_root** (*str*) – Root path for unconditional images.
- **pipeline** (*list[dict | callable]*) – A sequence of data transforms.
- **len_per_stage** (*int, optional*) – The length of dataset for each scale. This args change the length dataset by concatenating or extracting subset. If given a value less than 0., the original length will be kept. Defaults to 1e6.
- **gpu_samples_per_scale** (*dict | None, optional*) – Dict contains `samples_per_gpu` for each scale. For example, `{'32': 4}` will set the scale of 32 with `samples_per_gpu=4`, despite other scale with `samples_per_gpu=self.gpu_samples_base`.
- **gpu_samples_base** (*int, optional*) – Set default `samples_per_gpu` for each scale. Defaults to 32.

- **io_backend** (*str, optional*) – The storage backend type. Options are “disk”, “ceph”, “memcached”, “lmdb”, “http” and “petrel”. Default: None.
- **test_mode** (*bool, optional*) – If True, the dataset will work in test mode. Otherwise, in train mode. Default to False.

concat_imgs_list_to(*num*)

Concat image list to specified length.

Parameters **num** (*int*) – The length of the concatenated image list.

load_data_list()

Load annotations.

prepare_test_data(*idx*)

Prepare testing data.

Parameters **idx** (*int*) – Index of current batch.

Returns Prepared training data batch.

Return type dict

prepare_train_data(*idx*)

Prepare training data.

Parameters **idx** (*int*) – Index of current batch.

Returns Prepared training data batch.

Return type dict

update_annotations(*curr_scale*)

Update annotations.

Parameters **curr_scale** (*int*) – Current image scale.

Returns Whether to update.

Return type bool

1.35 mmedit.datasets.transforms

BinarizeImage	Binarize image.
Clip	Clip the pixels.
ColorJitter	An interface for torch color jitter so that it can be invoked in mmediting pipeline.
CopyValues	Copy the value of source keys to destination keys.
Crop	Crop data to specific size for training.
CropLike	Crop/pad the image in the target_key according to the size of image in the reference_key .
DegradationsWithShuffle	Apply random degradations to input, with degradations being shuffled.
LoadImageFromFile	Load a single image or image frames from corresponding paths.
LoadMask	Load Mask for multiple types.
Flip	Flip the input data with a probability.

continues on next page

Table 1 – continued from previous page

FixedCrop	Crop paired data (at a specific position) to specific size for training.
GenerateCoordinateAndCell	Generate coordinate and cell.
GenerateFacialHeatmap	Generate heatmap from keypoint.
GenerateFrameIndices	Generate frame index for REDS datasets.
GenerateFrameIndiceswithPadding	Generate frame index with padding for REDS dataset and Vid4 dataset during testing.
GenerateSegmentIndices	Generate frame indices for a segment.
GetMaskedImage	Get masked image.
GetSpatialDiscountMask	Get spatial discounting mask constant.
MATLABLikeResize	Resize the input image using MATLAB-like downsampling.
MirrorSequence	Extend short sequences (e.g.
ModCrop	Mod crop images, used during testing.
Normalize	Normalize images with the given mean and std value.
PackEditInputs	Pack the inputs data for SR, VFI, matting and inpainting.
PairedRandomCrop	Paried random crop.
RandomAffine	Apply random affine to input images.
RandomBlur	Apply random blur to the input.
RandomDownSampling	Generate LQ image from GT (and crop), which will randomly pick a scale.
RandomJPEGCompression	Apply random JPEG compression to the input.
RandomMaskDilation	Randomly dilate binary masks.
RandomNoise	Apply random noise to the input.
RandomResize	Randomly resize the input.
RandomResizedCrop	Crop data to random size and aspect ratio.
RandomRotation	Rotate the image by a randomly-chosen angle, measured in degree.
RandomTransposeHW	Randomly transpose images in H and W dimensions with a probability.
RandomVideoCompression	Apply random video compression to the input.
RescaleToZeroOne	Transform the images into a range between 0 and 1.
Resize	Resize data to a specific size for training or resize the images to fit the network input regulation for testing.
SetValues	Set value to destination keys.
TemporalReverse	Reverse frame lists for temporal augmentation.
ToTensor	Convert some values in results dict to *torch.Tensor* type in data loader pipeline.
UnsharpMasking	Apply unsharp masking to an image or a sequence of images.
CropAroundCenter	Randomly crop the images around unknown area in the center 1/4 images.
CropAroundFg	Crop around the whole foreground in the segmentation mask.
GenerateSeg	Generate segmentation mask from alpha matte.
CropAroundUnknown	Crop around unknown area with a randomly selected scale.
GenerateSoftSeg	Generate soft segmentation mask from input segmentation mask.
FormatTrimap	Convert trimap (tensor) to one-hot representation.
TransformTrimap	Transform trimap into two-channel and six-channel.

continues on next page

Table 1 – continued from previous page

GenerateTrimap	Using random erode/dilate to generate trimap from alpha matte.
GenerateTrimapWithDistTransform	Generate trimap with distance transform function.
CompositeFg	Composite foreground with a random foreground.
RandomLoadResizeBg	Randomly load a background image and resize it.
MergeFgAndBg	Composite foreground image and background image with alpha.
PerturbBg	Randomly add gaussian noise or gamma change to background image.
RandomJitter	Randomly jitter the foreground in hsv space.
LoadPairedImageFromFile	Load a pair of images from file.
CenterCropLongEdge	Center crop the given image by the long edge.
RandomCropLongEdge	Random crop the given image by the long edge.
NumpyPad	Numpy Padding.

1.35.1 BinarizeImage

class mmedit.datasets.transforms.**BinarizeImage**(*keys*, *binary_thr*, *a_min=0*, *a_max=1*, *dtype=<class 'numpy.uint8'>*)

Binarize image.

Parameters

- **keys** (*Sequence[str]*) – The images to be binarized.
- **binary_thr** (*float*) – Threshold for binarization.
- **amin** (*int*) – Lower limits of pixel value.
- **amx** (*int*) – Upper limits of pixel value.
- **dtype** (*np.dtype*) – Set the data type of the output. Default: np.uint8

transform(*results*)

The transform function of BinarizeImage.

Parameters **results** (*dict*) – A dict containing the necessary information and data for augmentation.

Returns A dict containing the processed data and information.

Return type dict

1.35.2 Clip

class mmedit.datasets.transforms.**Clip**(*keys*, *a_min=0*, *a_max=255*)

Clip the pixels.

Modified keys are the attributes specified in "keys".

Parameters

- **keys** (*list[str]*) – The keys whose values are clipped.
- **amin** (*int*) – Lower limits of pixel value.
- **amx** (*int*) – Upper limits of pixel value.

transform(*results*)

transform function.

Parameters **results** (*dict*) – A dict containing the necessary information and data for augmentation.

Returns

A dict with the values of the specified keys are rounded and clipped.

Return type dict

1.35.3 ColorJitter

class mmedit.datasets.transforms.**ColorJitter**(*keys*, *channel_order='rgb'*, ***kwargs*)

An interface for torch color jitter so that it can be invoked in mmediting pipeline.

Randomly change the brightness, contrast and saturation of an image. Modified keys are the attributes specified in "keys".

Required Keys:

- [KEYS]

Modified Keys:

- [KEYS]

Parameters

- **keys** (*list[str]*) – The images to be resized.
- **channel_order** (*str*) – Order of channel, candidates are 'bgr' and 'rgb'. Default: 'rgb'.

Notes

**kwards follows the args list of `torchvision.transforms.ColorJitter`.

brightness (float or tuple of float (min, max)): How much to jitter brightness. brightness_factor is chosen uniformly from [max(0, 1 - brightness), 1 + brightness] or the given [min, max]. Should be non negative numbers.

contrast (float or tuple of float (min, max)): How much to jitter contrast. contrast_factor is chosen uniformly from [max(0, 1 - contrast), 1 + contrast] or the given [min, max]. Should be non negative numbers.

saturation (float or tuple of float (min, max)): How much to jitter saturation. saturation_factor is chosen uniformly from [max(0, 1 - saturation), 1 + saturation] or the given [min, max]. Should be non negative numbers.

hue (float or tuple of float (min, max)): How much to jitter hue. hue_factor is chosen uniformly from [-hue, hue] or the given [min, max]. Should have 0<= hue <= 0.5 or -0.5 <= min <= max <= 0.5.

transform(*results: Dict*) → Dict

The transform function of ColorJitter.

Parameters **results** (*dict*) – The result dict.

Returns The result dict.

Return type dict

1.35.4 CopyValues

class mmedit.datasets.transforms.**CopyValues**(*src_keys*, *dst_keys*)

Copy the value of source keys to destination keys.

TODO Change to dict(dst=src)

It does the following: results[dst_key] = results[src_key] for (src_key, dst_key) in zip(src_keys, dst_keys).

Added keys are the keys in the attribute "dst_keys".

Required Keys:

- [SRC_KEYS]

Added Keys:

- [DST_KEYS]

Parameters

- **src_keys** (*list[str]*) – The source keys.
- **dst_keys** (*list[str]*) – The destination keys.

transform(*results*)

transform function.

Parameters **results** (*dict*) – A dict containing the necessary information and data for augmentation.

Returns A dict with a key added/modified.

Return type dict

1.35.5 Crop

class mmedit.datasets.transforms.**Crop**(*keys*, *crop_size*, *random_crop=True*, *is_pad_zeros=False*)

Crop data to specific size for training.

Parameters

- **keys** (*Sequence[str]*) – The images to be cropped.
- **crop_size** (*Tuple[int]*) – Target spatial size (h, w).
- **random_crop** (*bool*) – If set to True, it will random crop image. Otherwise, it will work as center crop. Default: True.
- **is_pad_zeros** (*bool, optional*) – Whether to pad the image with 0 if crop_size is greater than image size. Default: False.

transform(*results*)

Transform function.

Parameters **results** (*dict*) – A dict containing the necessary information and data for augmentation.

Returns A dict containing the processed data and information.

Return type dict

1.35.6 CropLike

class mmedit.datasets.transforms.**CropLike**(*target_key*, *reference_key=None*)

Crop/pad the image in the target_key according to the size of image in the reference_key .

Parameters

- **target_key** (*str*) – The key needs to be cropped.
- **reference_key** (*str | None*) – The reference key, need its size. Default: None.

transform(*results*)

Transform function.

Parameters **results** (*dict*) – A dict containing the necessary information and data for augmentation. Require self.target_key and self.reference_key.

Returns

A dict containing the processed data and information. Modify self.target_key.

Return type dict

1.35.7 DegradationsWithShuffle

class mmedit.datasets.transforms.**DegradationsWithShuffle**(*degradations*, *keys*, *shuffle_idx=None*)

Apply random degradations to input, with degradations being shuffled.

Degradation groups are supported. The order of degradations within the same group is preserved. For example, if we have degradations = [a, b, [c, d]] and shuffle_idx = None, then the possible orders are

```
[a, b, [c, d]]
[a, [c, d], b]
[b, a, [c, d]]
[b, [c, d], a]
[[c, d], a, b]
[[c, d], b, a]
```

Modified keys are the attributed specified in "keys".

Parameters

- **degradations** (*list[dict]*) – The list of degradations.
- **keys** (*list[str]*) – A list specifying the keys whose values are modified.
- **shuffle_idx** (*list | None, optional*) – The degradations corresponding to these indices are shuffled. If None, all degradations are shuffled. Default: None.

1.35.8 LoadImageFromFile

class mmedit.datasets.transforms.**LoadImageFromFile**(*key: str*, *color_type: str = 'color'*, *channel_order: str = 'bgr'*, *imdecode_backend: Optional[str] = None*, *use_cache: bool = False*, *to_float32: bool = False*, *to_y_channel: bool = False*, *save_original_img: bool = False*, *file_client_args: Optional[dict] = None*)

Load a single image or image frames from corresponding paths. Required Keys: - [Key]_path

New Keys: - [KEY] - ori_[KEY]_shape - ori_[KEY]

Parameters

- **key** (*str*) – Keys in results to find corresponding path.
- **color_type** (*str*) – The flag argument for :func:mmcv.imfrombytes. Defaults to 'color'.
- **channel_order** (*str*) – Order of channel, candidates are 'bgr' and 'rgb'. Default: 'bgr'.
- **imdecode_backend** (*str*) – The image decoding backend type. The backend argument for :func:mmcv.imfrombytes. See :func:mmcv.imfrombytes for details. candidates are 'cv2', 'turbojpeg', 'pillow', and 'tifffile'. Defaults to None.
- **use_cache** (*bool*) – If True, load all images at once. Default: False.
- **to_float32** (*bool*) – Whether to convert the loaded image to a float32 numpy array. If set to False, the loaded image is an uint8 array. Defaults to False.
- **to_y_channel** (*bool*) – Whether to convert the loaded image to y channel. Only support 'rgb2ycbcr' and 'rgb2ycbcr' Defaults to False.
- **file_client_args** (*dict*) – Arguments to instantiate a FileClient. If not specified, will infer from file uri. See mmengine.fileio.FileClient for details. Defaults to None.

transform(*results: dict*) → dict

Functions to load image or frames.

Parameters **results** (*dict*) – Result dict from :obj:mmcv.BaseDataset.

Returns The dict contains loaded image and meta information.

Return type dict

1.35.9 LoadMask

class mmedit.datasets.transforms.**LoadMask**(*mask_mode='bbox'*, *mask_config=None*)

Load Mask for multiple types.

For different types of mask, users need to provide the corresponding config dict.

Example config for bbox:

```
config = dict(img_shape=(256, 256), max_bbox_shape=128)
```

Example config for irregular:

```
config = dict(
    img_shape=(256, 256),
    num_vertices=(4, 12),
    max_angle=4.,
    length_range=(10, 100),
    brush_width=(10, 40),
    area_ratio_range=(0.15, 0.5))
```

Example config for ff:

```
config = dict(
    img_shape=(256, 256),
    num_vertices=(4, 12),
    mean_angle=1.2,
    angle_range=0.4,
    brush_width=(12, 40))
```

Example config for set:

```
config = dict(
    mask_list_file='xxx/xxx/ooxx.txt',
    prefix='/xxx/xxx/ooxx/',
    io_backend='disk',
    color_type='unchanged',
    file_client_kwargs=dict()
)

The mask_list_file contains the list of mask file name like this:
    test1.jpeg
    test2.jpeg
    ...
    ...

The prefix gives the data path.
```

Parameters

- **mask_mode** (*str*) – Mask mode in [‘bbox’, ‘irregular’, ‘ff’, ‘set’, ‘file’]. Default: ‘bbox’. * bbox: square bounding box masks. * irregular: irregular holes. * ff: free-form holes from DeepFillv2. * set: randomly get a mask from a mask set. * file: get mask from ‘mask_path’ in results.
- **mask_config** (*dict*) – Params for creating masks. Each type of mask needs different configs. Default: None.

transform(*results*)

transform function.

Parameters **results** (*dict*) – A dict containing the necessary information and data for augmentation.

Returns A dict containing the processed data and information.

Return type dict

1.35.10 Flip

class mmedit.datasets.transforms.**Flip**(*keys*, *flip_ratio=0.5*, *direction='horizontal'*)

Flip the input data with a probability.

Reverse the order of elements in the given data with a specific direction. The shape of the data is preserved, but the elements are reordered. Required keys are the keys in attributes "keys", added or modified keys are "flip", "flip_direction" and the keys in attributes "keys". It also supports flipping a list of images with the same flip.

Required Keys:

- [KEYS]

Modified Keys:

- [KEYS]

Parameters

- **keys** (*Union[str, List[str]]*) – The images to be flipped.
- **flip_ratio** (*float*) – The probability to flip the images. Default: 0.5.
- **direction** (*str*) – Flip images horizontally or vertically. Options are "horizontal" | "vertical". Default: "horizontal".

transform(*results*)

transform function.

Parameters **results** (*dict*) – A dict containing the necessary information and data for augmentation.

Returns A dict containing the processed data and information.

Return type dict

1.35.11 FixedCrop

class mmedit.datasets.transforms.**FixedCrop**(*keys*, *crop_size*, *crop_pos=None*)

Crop paired data (at a specific position) to specific size for training.

Parameters

- **keys** (*Sequence[str]*) – The images to be cropped.
- **crop_size** (*Tuple[int]*) – Target spatial size (h, w).
- **crop_pos** (*Tuple[int]*) – Specific position (x, y). If set to None, random initialize the position to crop paired data batch. Default: None.

transform(*results*)

Transform function.

Parameters **results** (*dict*) – A dict containing the necessary information and data for augmentation.

Returns A dict containing the processed data and information.

Return type dict

1.35.12 GenerateCoordinateAndCell

class mmedit.datasets.transforms.**GenerateCoordinateAndCell**(*sample_quantity=None*, *scale=None*, *target_size=None*, *reshape_gt=True*)

Generate coordinate and cell. Generate coordinate from the desired size of SR image.

Train or val:

1. Generate coordinate from GT.

#. Reshape GT image to (HgWg, 3) and transpose to (3, HgWg). where *Hg* and *Wg* represent the height and width of GT.

Test:

1. Generate coordinate from LQ and scale or target_size.
2. Then generate cell from coordinate.

Parameters

- **sample_quantity** (*int | None*) – The quantity of samples in coordinates. To ensure that the GT tensors in a batch have the same dimensions. Default: None.
- **scale** (*float*) – Scale of upsampling. Default: None.
- **target_size** (*tuple[int]*) – Size of target image. Default: None.
- **reshape_gt** (*bool*) – Whether reshape gt to (-1, 3). Default: True If sample_quantity is not None, reshape_gt = True.

The priority of getting 'size of target image' is:

1. results['gt'].shape[-2:]
2. results['lq'].shape[-2:] * scale
3. target_size

transform(*results*)

Call function.

Parameters

- **results** (*Require either in*) – A dict containing the necessary information
- **augmentation.** (*and data for*) –
- **results** –
- **'lq'** (*1.*) –
- **'gt'** (*2.*) –
- **None** (*3.*) –
- **and** (*the premise is self.target_size*) –
- **len** (*self.target_size*) –

Returns A dict containing the processed data and information. Reshape 'gt' to (-1, 3) and transpose to (3, -1) if 'gt' in results. Add 'coord' and 'cell'.

Return type dict

1.35.13 GenerateFacialHeatmap

class mmedit.datasets.transforms.**GenerateFacialHeatmap**(*image_key*, *ori_size*, *target_size*, *sigma=1.0*, *use_cache=True*)

Generate heatmap from keypoint.

Parameters

- **image_key** (*str*) – Key of facial image in dict.
- **ori_size** (*int | Tuple[int]*) – Original image size of keypoint.
- **target_size** (*int | Tuple[int]*) – Target size of heatmap.
- **sigma** (*float*) – Sigma parameter of heatmap. Default: 1.0
- **use_cache** (*bool*) – If True, load all heatmap at once. Default: True.

generate_heatmap_from_img(*image*)

Generate heatmap from img.

Parameters **image** (*np.ndarray*) – Face image.

results: heatmap (np.ndarray): Heatmap the face image.

transform(*results*)

transform function.

Parameters **results** (*dict*) – A dict containing the necessary information and data for augmentation. Require keypoint.

Returns

A dict containing the processed data and information. Add 'heatmap'.

Return type dict

1.35.14 GenerateFrameIndices

class mmedit.datasets.transforms.**GenerateFrameIndices**(*interval_list*, *frames_per_clip=99*)

Generate frame index for REDS datasets. It also performs temporal augmention with random interval.

Required Keys:

- img_path
- gt_path
- key
- num_input_frames

Modified Keys:

- img_path
- gt_path

Added Keys:

- interval
- reverse

Parameters

- **interval_list** (*list[int]*) – Interval list for temporal augmentation. It will randomly pick an interval from interval_list and sample frame index with the interval.
- **frames_per_clip** (*int*) – Number of frames per clips. Default: 99 for REDS dataset.

transform(*results*)

transform function.

Parameters **results** (*dict*) – A dict containing the necessary information and data for augmentation.

Returns A dict containing the processed data and information.

Return type dict

1.35.15 GenerateFrameIndiceswithPadding

class mmedit.datasets.transforms.**GenerateFrameIndiceswithPadding**(*padding*, *filename_tmpl='{:08d}'*)

Generate frame index with padding for REDS dataset and Vid4 dataset during testing.

Required Keys:

- img_path
- gt_path
- key
- num_input_frames
- sequence_length

Modified Keys:

- img_path
- gt_path

Parameters **padding** – padding mode, one of 'replicate' | 'reflection' | 'reflection_circle' | 'circle'.

Examples: current_idx = 0, num_input_frames = 5 The generated frame indices under different padding mode:

replicate: [0, 0, 0, 1, 2] reflection: [2, 1, 0, 1, 2] reflection_circle: [4, 3, 0, 1, 2] circle: [3, 4, 0, 1, 2]

transform(*results*)

transform function.

Parameters **results** (*dict*) – A dict containing the necessary information and data for augmentation.

Returns A dict containing the processed data and information.

Return type dict

1.35.16 GenerateSegmentIndices

class mmedit.datasets.transforms.**GenerateSegmentIndices**(*interval_list*, *start_idx=0*, *filename_tmpl='{:08d}.png'*)

Generate frame indices for a segment. It also performs temporal augmention with random interval.

Required Keys:

- img_path
- gt_path
- key
- num_input_frames
- sequence_length

Modified Keys:

- img_path
- gt_path

Added Keys:

- interval
- reverse

Parameters

- **interval_list** (*list[int]*) – Interval list for temporal augmentation. It will randomly pick an interval from interval_list and sample frame index with the interval.
- **start_idx** (*int*) – The index corresponds to the first frame in the sequence. Default: 0.
- **filename_tmpl** (*str*) – Template for file name. Default: '{:08d}.png'.

transform(*results*)

transform function.

Parameters **results** (*dict*) – A dict containing the necessary information and data for augmentation.

Returns A dict containing the processed data and information.

Return type dict

1.35.17 GetMaskedImage

class mmedit.datasets.transforms.**GetMaskedImage**(*img_key='gt'*, *mask_key='mask'*, *out_key='img'*, *zero_value=127.5*)

Get masked image.

Parameters

- **img_key** (*str*) – Key for clean image. Default: 'gt'.
- **mask_key** (*str*) – Key for mask image. The mask shape should be (h, w, 1) while '1' indicate holes and '0' indicate valid regions. Default: 'mask'.
- **img_key** – Key for output image. Default: 'img'.

- **zero_value** (*float*) – Pixel value of masked area.

transform(*results*)

transform function.

Parameters **results** (*dict*) – A dict containing the necessary information and data for augmentation.

Returns A dict containing the processed data and information.

Return type dict

1.35.18 GetSpatialDiscountMask

class mmedit.datasets.transforms.**GetSpatialDiscountMask**(*gamma=0.99*, *beta=1.5*)

Get spatial discounting mask constant.

Spatial discounting mask is first introduced in: Generative Image Inpainting with Contextual Attention.

Parameters

- **gamma** (*float, optional*) – Gamma for computing spatial discounting. Defaults to 0.99.
- **beta** (*float, optional*) – Beta for computing spatial discounting. Defaults to 1.5.

spatial_discount_mask(*mask_width*, *mask_height*)

Generate spatial discounting mask constant.

Parameters

- **mask_width** (*int*) – The width of bbox hole.
- **mask_height** (*int*) – The height of bbox height.

Returns Spatial discounting mask.

Return type np.ndarray

transform(*results*)

transform function.

Parameters **results** (*dict*) – A dict containing the necessary information and data for augmentation.

Returns A dict containing the processed data and information.

Return type dict

1.35.19 MATLABLikeResize

class mmedit.datasets.transforms.**MATLABLikeResize**(*keys*, *scale=None*, *output_shape=None*, *kernel='bicubic'*, *kernel_width=4.0*)

Resize the input image using MATLAB-like downsampling.

Currently support bicubic interpolation only. Note that the output of this function is slightly different from the official MATLAB function.

Required keys are the keys in attribute "keys". Added or modified keys are "scale" and "output_shape", and the keys in attribute "keys".

Parameters

- **keys** (*list[str]*) – A list of keys whose values are modified.
- **scale** (*float | None, optional*) – The scale factor of the resize operation. If None, it will be determined by output_shape. Default: None.
- **output_shape** (*tuple(int) | None, optional*) – The size of the output image. If None, it will be determined by scale. Note that if scale is provided, output_shape will not be used. Default: None.
- **kernel** (*str, optional*) – The kernel for the resize operation. Currently support 'bicubic' only. Default: 'bicubic'.
- **kernel_width** (*float*) – The kernel width. Currently support 4.0 only. Default: 4.0.

transform(*results*)

transform function.

Parameters **results** (*dict*) – A dict containing the necessary information and data for augmentation.

Returns A dict containing the processed data and information.

Return type dict

1.35.20 MirrorSequence

class mmedit.datasets.transforms.**MirrorSequence**(*keys*)

Extend short sequences (e.g. Vimeo-90K) by mirroring the sequences.

Given a sequence with N frames (x1, …, xN), extend the sequence to (x1, …, xN, xN, …, x1).

Required Keys:

- [KEYS]

Modified Keys:

- [KEYS]

Parameters **keys** (*list[str]*) – The frame lists to be extended.

transform(*results*)

transform function.

Parameters **results** (*dict*) – A dict containing the necessary information and data for augmentation.

Returns A dict containing the processed data and information.

Return type dict

1.35.21 ModCrop

class mmedit.datasets.transforms.**ModCrop**(*key='gt'*)

Mod crop images, used during testing.

Required keys are "scale" and "KEY", added or modified keys are "KEY".

Parameters **key** (*str*) – The key of image. Default: 'gt'

transform(*results*)

Transform function.

Parameters **results** (*dict*) – A dict containing the necessary information and data for augmentation.

Returns A dict containing the processed data and information.

Return type dict

1.35.22 Normalize

class mmedit.datasets.transforms.**Normalize**(*keys*, *mean*, *std*, *to_rgb=False*, *save_original=False*)

Normalize images with the given mean and std value.

Required keys are the keys in attribute "keys", added or modified keys are the keys in attribute "keys" and these keys with postfix '_norm_cfg'. It also supports normalizing a list of images.

Parameters

- **keys** (*Sequence[str]*) – The images to be normalized.
- **mean** (*np.ndarray*) – Mean values of different channels.
- **std** (*np.ndarray*) – Std values of different channels.
- **to_rgb** (*bool*) – Whether to convert channels from BGR to RGB. Default: False.
- **save_original** (*bool*) – Whether to save original images. Default: False.

transform(*results*)

transform function.

Parameters **results** (*dict*) – A dict containing the necessary information and data for augmentation.

Returns A dict containing the processed data and information.

Return type dict

1.35.23 PackEditInputs

class mmedit.datasets.transforms.**PackEditInputs**(*keys: Optional[Tuple[List[str], str, None]] = None*, *pack_all: bool = False*)

Pack the inputs data for SR, VFI, matting and inpainting.

Keys for images include img, gt, ref, mask, gt_heatmap, trimap, gt_alpha, gt_fg, gt_bg. All of them will be packed into data field of EditDataSample.

pack_all (bool): Whether pack all variables in *results* to *inputs* dict. This is useful when keys of the input dict is not fixed. Please be careful when using this function, because we do not Defaults to False.

Others will be packed into metainfo field of EditDataSample.

transform(*results: dict*) → dict

Method to pack the input data.

Parameters **results** (*dict*) – Result dict from the data pipeline.

Returns

- 'inputs' (obj:*torch.Tensor*): The forward data of models.
- **'data_samples' (obj:*EditDataSample*): The annotation info of the** sample.

Return type dict

1.35.24 PairedRandomCrop

class mmedit.datasets.transforms.**PairedRandomCrop**(*gt_patch_size*, *lq_key='img'*, *gt_key='gt'*)

Paried random crop.

It crops a pair of img and gt images with corresponding locations. It also supports accepting img list and gt list. Required keys are "scale", "lq_key", and "gt_key", added or modified keys are "lq_key" and "gt_key".

Parameters

- **gt_patch_size** (*int*) – cropped gt patch size.
- **lq_key** (*str*) – Key of LQ img. Default: 'img'.
- **gt_key** (*str*) – Key of GT img. Default: 'gt'.

transform(*results*)

Transform function.

Parameters **results** (*dict*) – A dict containing the necessary information and data for augmentation.

Returns A dict containing the processed data and information.

Return type dict

1.35.25 RandomAffine

class mmedit.datasets.transforms.**RandomAffine**(*keys*, *degrees*, *translate=None*, *scale=None*, *shear=None*, *flip_ratio=None*)

Apply random affine to input images.

This class is adopted from https://github.com/pytorch/vision/blob/v0.5.0/torchvision/transforms/ transforms.py#L1015 It should be noted that in https://github.com/Yaoyi-Li/GCA-Matting/blob/master/dataloader/ data_generator.py#L70 random flip is added. See explanation of *flip_ratio* below. Required keys are the keys in attribute "keys", modified keys are keys in attribute "keys".

Parameters

- **keys** (*Sequence[str]*) – The images to be affined.
- **degrees** (*float | tuple[float]*) – Range of degrees to select from. If it is a float instead of a tuple like (min, max), the range of degrees will be (-degrees, +degrees). Set to 0 to deactivate rotations.

- **translate** (*tuple, optional*) – Tuple of maximum absolute fraction for horizontal and vertical translations. For example translate=(a, b), then horizontal shift is randomly sampled in the range -img_width * a < dx < img_width * a and vertical shift is randomly sampled in the range -img_height * b < dy < img_height * b. Default: None.
- **scale** (*tuple, optional*) – Scaling factor interval, e.g (a, b), then scale is randomly sampled from the range a <= scale <= b. Default: None.
- **shear** (*float | tuple[float], optional*) – Range of shear degrees to select from. If shear is a float, a shear parallel to the x axis and a shear parallel to the y axis in the range (-shear, +shear) will be applied. Else if shear is a tuple of 2 values, a x-axis shear and a y-axis shear in (shear[0], shear[1]) will be applied. Default: None.
- **flip_ratio** (*float, optional*) – Probability of the image being flipped. The flips in horizontal direction and vertical direction are independent. The image may be flipped in both directions. Default: None.

transform(*results*)

transform function.

Parameters **results** (*dict*) – A dict containing the necessary information and data for augmentation.

Returns A dict containing the processed data and information.

Return type dict

1.35.26 RandomBlur

class mmedit.datasets.transforms.**RandomBlur**(*params*, *keys*)

Apply random blur to the input.

Modified keys are the attributed specified in "keys".

Parameters

- **params** (*dict*) – A dictionary specifying the degradation settings.
- **keys** (*list[str]*) – A list specifying the keys whose values are modified.

get_kernel(*num_kernels: int*)

This is the function to create kernel.

Parameters **num_kernels** (*int*) – the number of kernels

Returns _description_

Return type _type_

1.35.27 RandomDownSampling

class mmedit.datasets.transforms.**RandomDownSampling**(*scale_min=1.0*, *scale_max=4.0*, *patch_size=None*, *interpolation='bicubic'*, *backend='pillow'*)

Generate LQ image from GT (and crop), which will randomly pick a scale.

Parameters

- **scale_min** (*float*) – The minimum of upsampling scale, inclusive. Default: 1.0.

- **scale_max** (*float*) – The maximum of upsampling scale, exclusive. Default: 4.0.
- **patch_size** (*int*) – The cropped lr patch size. Default: None, means no crop.
- **interpolation** (*str*) – Interpolation method, accepted values are "nearest", "bilinear", "bicubic", "area", "lanczos" for 'cv2' backend, "nearest", "bilinear", "bicubic", "box", "lanczos", "hamming" for 'pillow' backend. Default: "bicubic".
- **backend** (*str | None*) – The image resize backend type. Options are *cv2*, *pillow*, *None*. If backend is None, the global imread_backend specified by `mmcv.use_backend()` will be used. Default: "pillow".
- **[scale_min** (*Scale will be picked in the range of*) –
- **scale_max).** –

transform(*results*)

transform function.

Parameters **results** (*dict*) – A dict containing the necessary information and data for augmentation. 'gt' is required.

Returns

A dict containing the processed data and information. modified 'gt', supplement 'lq' and 'scale' to keys.

Return type dict

1.35.28 RandomJPEGCompression

class mmedit.datasets.transforms.**RandomJPEGCompression**(*params*, *keys*)

Apply random JPEG compression to the input.

Modified keys are the attributed specified in "keys".

Parameters

- **params** (*dict*) – A dictionary specifying the degradation settings.
- **keys** (*list[str]*) – A list specifying the keys whose values are modified.

1.35.29 RandomMaskDilation

class mmedit.datasets.transforms.**RandomMaskDilation**(*keys*, *binary_thr=0.0*, *kernel_min=9*, *kernel_max=49*)

Randomly dilate binary masks.

Parameters

- **keys** (*Sequence[str]*) – The images to be resized.
- **binary_thr** (*float*) – Threshold for obtaining binary mask. Default: 0.
- **kernel_min** (*int*) – Min size of dilation kernel. Default: 9.
- **kernel_max** (*int*) – Max size of dilation kernel. Default: 49.

transform(*results*)

transform function.

Parameters **results** (*dict*) – A dict containing the necessary information and data for augmentation.

Returns A dict containing the processed data and information.

Return type dict

1.35.30 RandomNoise

class mmedit.datasets.transforms.**RandomNoise**(*params*, *keys*)

Apply random noise to the input.

Currently support Gaussian noise and Poisson noise.

Modified keys are the attributed specified in "keys".

Parameters

- **params** (*dict*) – A dictionary specifying the degradation settings.
- **keys** (*list[str]*) – A list specifying the keys whose values are modified.

1.35.31 RandomResize

class mmedit.datasets.transforms.**RandomResize**(*params*, *keys*)

Randomly resize the input.

Modified keys are the attributed specified in "keys".

Parameters

- **params** (*dict*) – A dictionary specifying the degradation settings.
- **keys** (*list[str]*) – A list specifying the keys whose values are modified.

1.35.32 RandomResizedCrop

class mmedit.datasets.transforms.**RandomResizedCrop**(*keys*, *crop_size*, *scale=(0.08, 1.0)*, *ratio=(0.75, 1.3333333333333333)*, *interpolation='bilinear'*)

Crop data to random size and aspect ratio.

A crop of a random proportion of the original image and a random aspect ratio of the original aspect ratio is made. The cropped image is finally resized to a given size specified by 'crop_size'. Modified keys are the attributes specified in "keys".

This code is partially adopted from torchvision.transforms.RandomResizedCrop: [https://pytorch.org/vision/stable/_modules/torchvision/transforms/ transforms.html#RandomResizedCrop].

Parameters

- **keys** (*list[str]*) – The images to be resized and random-cropped.
- **crop_size** (*int | tuple[int]*) – Target spatial size (h, w).
- **scale** (*tuple[float], optional*) – Range of the proportion of the original image to be cropped. Default: (0.08, 1.0).

- **ratio** (*tuple[float], optional*) – Range of aspect ratio of the crop. Default: (3. / 4., 4. / 3.).
- **interpolation** (*str, optional*) – Algorithm used for interpolation. It can be only either one of the following: "nearest" | "bilinear" | "bicubic" | "area" | "lanczos". Default: "bilinear".

get_params(*data*)

Get parameters for a random sized crop.

Parameters **data** (*np.ndarray*) – Image of type numpy array to be cropped.

Returns A tuple containing the coordinates of the top left corner and the chosen crop size.

transform(*results*)

Transform function.

Parameters **results** (*dict*) – A dict containing the necessary information and data for augmentation.

Returns A dict containing the processed data and information.

Return type dict

1.35.33 RandomRotation

class mmedit.datasets.transforms.**RandomRotation**(*keys*, *degrees*)

Rotate the image by a randomly-chosen angle, measured in degree.

Parameters

- **keys** (*list[str]*) – The images to be rotated.
- **degrees** (*tuple[float] | tuple[int] | float | int*) – If it is a tuple, it represents a range (min, max). If it is a float or int, the range is constructed as (-degrees, degrees).

transform(*results*)

transform function.

Parameters **results** (*dict*) – A dict containing the necessary information and data for augmentation.

Returns A dict containing the processed data and information.

Return type dict

1.35.34 RandomTransposeHW

class mmedit.datasets.transforms.**RandomTransposeHW**(*keys*, *transpose_ratio=0.5*)

Randomly transpose images in H and W dimensions with a probability.

(TransposeHW = horizontal flip + anti-clockwise rotation by 90 degrees) When used with horizontal/vertical flips, it serves as a way of rotation augmentation. It also supports randomly transposing a list of images.

Required keys are the keys in attributes "keys", added or modified keys are "transpose" and the keys in attributes "keys".

Parameters

- **keys** (*list[str]*) – The images to be transposed.

- **transpose_ratio** (*float*) – The probability to transpose the images. Default: 0.5.

transform(*results*)

transform function.

Parameters **results** (*dict*) – A dict containing the necessary information and data for augmentation.

Returns A dict containing the processed data and information.

Return type dict

1.35.35 RandomVideoCompression

class mmedit.datasets.transforms.**RandomVideoCompression**(*params*, *keys*)

Apply random video compression to the input.

Modified keys are the attributed specified in "keys".

Parameters

- **params** (*dict*) – A dictionary specifying the degradation settings.
- **keys** (*list[str]*) – A list specifying the keys whose values are modified.

1.35.36 RescaleToZeroOne

class mmedit.datasets.transforms.**RescaleToZeroOne**(*keys*)

Transform the images into a range between 0 and 1.

Required keys are the keys in attribute "keys", added or modified keys are the keys in attribute "keys". It also supports rescaling a list of images.

Parameters **keys** (*Sequence[str]*) – The images to be transformed.

transform(*results*)

transform function.

Parameters **results** (*dict*) – A dict containing the necessary information and data for augmentation.

Returns A dict containing the processed data and information.

Return type dict

1.35.37 Resize

class mmedit.datasets.transforms.**Resize**(*keys: Union[str, List[str]] = 'img'*, *scale=None*, *keep_ratio=False*, *size_factor=None*, *max_size=None*, *interpolation='bilinear'*, *backend=None*, *output_keys=None*)

Resize data to a specific size for training or resize the images to fit the network input regulation for testing.

When used for resizing images to fit network input regulation, the case is that a network may have several downsample and then upsample operation, then the input height and width should be divisible by the downsample factor of the network. For example, the network would downsample the input for 5 times with stride 2, then the downsample factor is 2^5 = 32 and the height and width should be divisible by 32.

Required keys are the keys in attribute "keys", added or modified keys are "keep_ratio", "scale_factor", "interpolation" and the keys in attribute "keys".

Required Keys:

- Required keys are the keys in attribute "keys"

Modified Keys:

- Modified the keys in attribute "keys" or save as new key ([OUT_KEY])

Added Keys:

- [OUT_KEY]_shape
- keep_ratio
- scale_factor
- interpolation

All keys in "keys" should have the same shape. "test_trans" is used to record the test transformation to align the input's shape.

Parameters

- **keys** (*str | list[str]*) – The image(s) to be resized.
- **scale** (*float | tuple[int]*) – If scale is tuple[int], target spatial size (h, w). Otherwise, target spatial size is scaled by input size. Note that when it is used, *size_factor* and *max_size* are useless. Default: None
- **keep_ratio** (*bool*) – If set to True, images will be resized without changing the aspect ratio. Otherwise, it will resize images to a given size. Default: False. Note that it is used together with *scale*.
- **size_factor** (*int*) – Let the output shape be a multiple of size_factor. Default:None. Note that when it is used, *scale* should be set to None and *keep_ratio* should be set to False.
- **max_size** (*int*) – The maximum size of the longest side of the output. Default:None. Note that it is used together with *size_factor*.
- **interpolation** (*str*) – Algorithm used for interpolation: "nearest" | "bilinear" | "bicubic" | "area" | "lanczos". Default: "bilinear".
- **backend** (*str | None*) – The image resize backend type. Options are *cv2*, *pillow*, *None*. If backend is None, the global imread_backend specified by `mmcv.use_backend()` will be used. Default: None.
- **output_keys** (*list[str] | None*) – The resized images. Default: None Note that if it is not *None*, its length should be equal to keys.

transform(*results: Dict*) → Dict

Transform function to resize images.

Parameters **results** (*dict*) – A dict containing the necessary information and data for augmentation.

Returns A dict containing the processed data and information.

Return type dict

1.35.38 SetValues

class mmedit.datasets.transforms.**SetValues**(*dictionary*)

Set value to destination keys.

It does the following: results[key] = value

Added keys are the keys in the dictionary.

Required Keys:

- None

Added or Modifyed Keys:

- keys in the dictionary

Parameters **dictionary** (*dict*) – The dictionary to update.

transform(*results: Dict*)

transform function.

Parameters **results** (*dict*) – A dict containing the necessary information and data for augmentation.

Returns A dict with a key added/modified.

Return type dict

1.35.39 TemporalReverse

class mmedit.datasets.transforms.**TemporalReverse**(*keys*, *reverse_ratio=0.5*)

Reverse frame lists for temporal augmentation.

Required keys are the keys in attributes “lq” and “gt”, added or modified keys are “lq”, “gt” and “reverse”.

Parameters

- **keys** (*list[str]*) – The frame lists to be reversed.
- **reverse_ratio** (*float*) – The probability to reverse the frame lists. Default: 0.5.

transform(*results*)

transform function.

Parameters **results** (*dict*) – A dict containing the necessary information and data for augmentation.

Returns A dict containing the processed data and information.

Return type dict

1.35.40 ToTensor

class mmedit.datasets.transforms.**ToTensor**(*keys*, *to_float32=True*)

Convert some values in results dict to *torch.Tensor* type in data loader pipeline.

Parameters

- **keys** (*Sequence[str]*) – Required keys to be converted.
- **to_float32** (*bool*) – Whether convert tensors of images to float32. Default: True.

transform(*results*)

transform function.

Parameters **results** (*dict*) – A dict containing the necessary information and data for augmentation.

Returns A dict containing the processed data and information.

Return type dict

1.35.41 UnsharpMasking

class mmedit.datasets.transforms.**UnsharpMasking**(*kernel_size*, *sigma*, *weight*, *threshold*, *keys*)

Apply unsharp masking to an image or a sequence of images.

Parameters

- **kernel_size** (*int*) – The kernel_size of the Gaussian kernel.
- **sigma** (*float*) – The standard deviation of the Gaussian.
- **weight** (*float*) – The weight of the "details" in the final output.
- **threshold** (*float*) – Pixel differences larger than this value are regarded as "details".
- **keys** (*list[str]*) – The keys whose values are processed.

Added keys are "xxx_unsharp", where "xxx" are the attributes specified in "keys".

transform(*results*)

transform function.

Parameters **results** (*dict*) – A dict containing the necessary information and data for augmentation.

Returns A dict containing the processed data and information.

Return type dict

1.35.42 CropAroundCenter

class mmedit.datasets.transforms.**CropAroundCenter**(*crop_size*)

Randomly crop the images around unknown area in the center 1/4 images.

This cropping strategy is adopted in GCA matting. The *unknown area* is the same as *semi-transparent area*. https://arxiv.org/pdf/2001.04069.pdf

It retains the center 1/4 images and resizes the images to 'crop_size'. Required keys are "fg", "bg", "trimap" and "alpha", added or modified keys are "crop_bbox", "fg", "bg", "trimap" and "alpha".

Parameters **crop_size** (*int | tuple*) – Desired output size. If int, square crop is applied.

transform(*results*)

Transform function.

Parameters **results** (*dict*) – A dict containing the necessary information and data for augmentation.

Returns A dict containing the processed data and information.

Return type dict

1.35.43 CropAroundFg

class mmedit.datasets.transforms.**CropAroundFg**(*keys*, *bd_ratio_range=(0.1, 0.4)*, *test_mode=False*)

Crop around the whole foreground in the segmentation mask.

Required keys are "seg" and the keys in argument *keys*. Meanwhile, "seg" must be in argument *keys*. Added or modified keys are "crop_bbox" and the keys in argument *keys*.

Parameters

- **keys** (*Sequence[str]*) – The images to be cropped. It must contain 'seg'.
- **bd_ratio_range** (*tuple, optional*) – The range of the boundary (bd) ratio to select from. The boundary ratio is the ratio of the boundary to the minimal bbox that contains the whole foreground given by segmentation. Default to (0.1, 0.4).
- **test_mode** (*bool*) – Whether use test mode. In test mode, the tight crop area of foreground will be extended to the a square. Default to False.

transform(*results*)

Transform function.

Parameters **results** (*dict*) – A dict containing the necessary information and data for augmentation.

Returns A dict containing the processed data and information.

Return type dict

1.35.44 GenerateSeg

class mmedit.datasets.transforms.**GenerateSeg**(*kernel_size=5*, *erode_iter_range=(10, 20)*, *dilate_iter_range=(15, 30)*, *num_holes_range=(0, 3)*, *hole_sizes=[(15, 15), (25, 25), (35, 35), (45, 45)]*, *blur_ksizes=[(21, 21), (31, 31), (41, 41)]*)

Generate segmentation mask from alpha matte.

Parameters

- **kernel_size** (*int, optional*) – Kernel size for both erosion and dilation. The kernel will have the same height and width. Defaults to 5.
- **erode_iter_range** (*tuple, optional*) – Iteration of erosion. Defaults to (10, 20).
- **dilate_iter_range** (*tuple, optional*) – Iteration of dilation. Defaults to (15, 30).
- **num_holes_range** (*tuple, optional*) – Range of number of holes to randomly select from. Defaults to (0, 3).

- **hole_sizes** (*list, optional*) – List of (h, w) to be selected as the size of the rectangle hole. Defaults to [(15, 15), (25, 25), (35, 35), (45, 45)].
- **blur_ksizes** (*list, optional*) – List of (h, w) to be selected as the kernel_size of the gaussian blur. Defaults to [(21, 21), (31, 31), (41, 41)].

transform(*results: dict*) → dict

Transform function.

Parameters **results** (*dict*) – A dict containing the necessary information and data for augmentation.

Returns A dict containing the processed data and information.

Return type dict

1.35.45 CropAroundUnknown

class mmedit.datasets.transforms.**CropAroundUnknown**(*keys*, *crop_sizes*, *unknown_source='alpha'*, *interpolations='bilinear'*)

Crop around unknown area with a randomly selected scale.

Randomly select the w and h from a list of (w, h). Required keys are the keys in argument *keys*, added or modified keys are "crop_bbox" and the keys in argument *keys*. This class assumes value of "alpha" ranges from 0 to 255.

Parameters

- **keys** (*Sequence[str]*) – The images to be cropped. It must contain 'alpha'. If unknown_source is set to 'trimap', then it must also contain 'trimap'.
- **crop_sizes** (*list[int | tuple[int]]*) – List of (w, h) to be selected.
- **unknown_source** (*str, optional*) – Unknown area to select from. It must be 'alpha' or 'trimap'. Default to 'alpha'.
- **interpolations** (*str | list[str], optional*) – Interpolation method of mmcv.imresize. The interpolation operation will be applied when image size is smaller than the crop_size. If given as a list of str, it should have the same length as *keys*. Or if given as a str all the keys will be resized with the same method. Default to 'bilinear'.

transform(*results*)

Transform function.

Parameters **results** (*dict*) – A dict containing the necessary information and data for augmentation.

Returns A dict containing the processed data and information.

Return type dict

1.35.46 GenerateSoftSeg

class mmedit.datasets.transforms.**GenerateSoftSeg**(*fg_thr=0.2*, *border_width=25*, *erode_ksize=3*, *dilate_ksize=5*, *erode_iter_range=(10, 20)*, *dilate_iter_range=(3, 7)*, *blur_ksizes=[(21, 21), (31, 31), (41, 41)]*)

Generate soft segmentation mask from input segmentation mask.

Required key is "seg", added key is "soft_seg".

Parameters

- **fg_thr** (*float, optional*) – Threshold of the foreground in the normalized input segmentation mask. Defaults to 0.2.
- **border_width** (*int, optional*) – Width of border to be padded to the bottom of the mask. Defaults to 25.
- **erode_ksize** (*int, optional*) – Fixed kernel size of the erosion. Defaults to 5.
- **dilate_ksize** (*int, optional*) – Fixed kernel size of the dilation. Defaults to 5.
- **erode_iter_range** (*tuple, optional*) – Iteration of erosion. Defaults to (10, 20).
- **dilate_iter_range** (*tuple, optional*) – Iteration of dilation. Defaults to (3, 7).
- **blur_ksizes** (*list, optional*) – List of (h, w) to be selected as the kernel_size of the gaussian blur. Defaults to [(21, 21), (31, 31), (41, 41)].

transform(*results: dict*) → dict

Transform function.

Parameters **results** (*dict*) – A dict containing the necessary information and data for augmentation.

Returns A dict containing the processed data and information.

Return type dict

1.35.47 FormatTrimap

class mmedit.datasets.transforms.**FormatTrimap**(*to_onehot=False*)

Convert trimap (tensor) to one-hot representation.

It transforms the trimap label from (0, 128, 255) to (0, 1, 2). If `to_onehot` is set to True, the trimap will convert to one-hot tensor of shape (3, H, W). Required key is "trimap", added or modified key are "trimap" and "format_trimap_to_onehot".

Parameters **to_onehot** (*bool*) – whether convert trimap to one-hot tensor. Default: `False`.

transform(*results*)

Transform function.

Parameters **results** (*dict*) – A dict containing the necessary information and data for augmentation.

Returns A dict containing the processed data and information.

Return type dict

1.35.48 TransformTrimap

class mmedit.datasets.transforms.**TransformTrimap**

Transform trimap into two-channel and six-channel.

This class will generate a two-channel trimap composed of definite foreground and background masks and encode it into a six-channel trimap using Gaussian blurs of the generated two-channel trimap at three different scales. The transformed trimap has 6 channels.

Required key is "trimap", added key is "transformed_trimap" and "two_channel_trimap".

Adopted from the following repository: https://github.com/MarcoForte/FBA_Matting/blob/master/networks/transforms.py.

transform(*results: dict*) → dict

Transform function.

Parameters **results** (*dict*) – A dict containing the necessary information and data for augmentation.

Returns A dict containing the processed data and information.

Return type dict

1.35.49 GenerateTrimap

class mmedit.datasets.transforms.**GenerateTrimap**(*kernel_size*, *iterations=1*, *random=True*)

Using random erode/dilate to generate trimap from alpha matte.

Required key is "alpha", added key is "trimap".

Parameters

- **kernel_size** (*int | tuple[int]*) – The range of random kernel_size of erode/dilate; int indicates a fixed kernel_size. If *random* is set to False and kernel_size is a tuple of length 2, then it will be interpreted as (erode kernel_size, dilate kernel_size). It should be noted that the kernel of the erosion and dilation has the same height and width.
- **iterations** (*int | tuple[int], optional*) – The range of random iterations of erode/dilate; int indicates a fixed iterations. If *random* is set to False and iterations is a tuple of length 2, then it will be interpreted as (erode iterations, dilate iterations). Default to 1.
- **random** (*bool, optional*) – Whether use random kernel_size and iterations when generating trimap. See *kernel_size* and *iterations* for more information. Default to True.

transform(*results: dict*) → dict

Transform function.

Parameters **results** (*dict*) – A dict containing the necessary information and data for augmentation.

Returns A dict containing the processed data and information.

Return type dict

1.35.50 GenerateTrimapWithDistTransform

class mmedit.datasets.transforms.**GenerateTrimapWithDistTransform**(*dist_thr=20*, *random=True*)

Generate trimap with distance transform function.

Parameters

- **dist_thr** (*int, optional*) – Distance threshold. Area with alpha value between (0, 255) will be considered as initial unknown area. Then area with distance to unknown area smaller than the distance threshold will also be consider as unknown area. Defaults to 20.
- **random** (*bool, optional*) – If True, use random distance threshold from [1, dist_thr). If False, use *dist_thr* as the distance threshold directly. Defaults to True.

transform(*results: dict*) → dict

Transform function.

Parameters **results** (*dict*) – A dict containing the necessary information and data for augmentation.

Returns A dict containing the processed data and information.

Return type dict

1.35.51 CompositeFg

class mmedit.datasets.transforms.**CompositeFg**(*fg_dirs*, *alpha_dirs*, *interpolation='nearest'*)

Composite foreground with a random foreground.

This class composites the current training sample with additional data randomly (could be from the same dataset). With probability 0.5, the sample will be composited with a random sample from the specified directory. The composition is performed as:

$$fg_{new} = \alpha_1 * fg_1 + (1 - \alpha_1) * fg_2$$
$$\alpha_{new} = 1 - (1 - \alpha_1) * (1 - \alpha_2)$$

where (fg_1, α_1) is from the current sample and (fg_2, α_2) is the randomly loaded sample. With the above composition, α_{new} is still in *[0, 1]*.

Required keys are "alpha" and "fg". Modified keys are "alpha" and "fg".

Parameters

- **fg_dirs** (*str | list[str]*) – Path of directories to load foreground images from.
- **alpha_dirs** (*str | list[str]*) – Path of directories to load alpha mattes from.
- **interpolation** (*str*) – Interpolation method of *mmcv.imresize* to resize the randomly loaded images. Default: 'nearest'.

transform(*results: dict*) → dict

Transform function.

Parameters **results** (*dict*) – A dict containing the necessary information and data for augmentation.

Returns A dict containing the processed data and information.

Return type dict

1.35.52 RandomLoadResizeBg

class mmedit.datasets.transforms.**RandomLoadResizeBg**(*bg_dir*, *flag='color'*, *channel_order='bgr'*)

Randomly load a background image and resize it.

Required key is "fg", added key is "bg".

Parameters

- **bg_dir** (*str*) – Path of directory to load background images from.
- **flag** (*str*) – Loading flag for images. Default: 'color'.
- **channel_order** (*str*) – Order of channel, candidates are 'bgr' and 'rgb'. Default: 'bgr'.
- **kwargs** (*dict*) – Args for file client.

transform(*results: dict*) → dict

Transform function.

Parameters **results** (*dict*) – A dict containing the necessary information and data for augmentation.

Returns A dict containing the processed data and information.

Return type dict

1.35.53 MergeFgAndBg

class mmedit.datasets.transforms.**MergeFgAndBg**

Composite foreground image and background image with alpha.

Required keys are "alpha", "fg" and "bg", added key is "merged".

transform(*results: dict*) → dict

Transform function.

Parameters **results** (*dict*) – A dict containing the necessary information and data for augmentation.

Returns A dict containing the processed data and information.

Return type dict

1.35.54 PerturbBg

class mmedit.datasets.transforms.**PerturbBg**(*gamma_ratio=0.6*)

Randomly add gaussian noise or gamma change to background image.

Required key is "bg", added key is "noisy_bg".

Parameters **gamma_ratio** (*float, optional*) – The probability to use gamma correction instead of gaussian noise. Defaults to 0.6.

transform(*results: dict*) → dict

Transform function.

Parameters **results** (*dict*) – A dict containing the necessary information and data for augmentation.

Returns A dict containing the processed data and information.

Return type dict

1.35.55 RandomJitter

class mmedit.datasets.transforms.**RandomJitter**(*hue_range=40*)

Randomly jitter the foreground in hsv space.

The jitter range of hue is adjustable while the jitter ranges of saturation and value are adaptive to the images. Side effect: the "fg" image will be converted to *np.float32*. Required keys are "fg" and "alpha", modified key is "fg".

Parameters **hue_range** (*float | tuple[float]*) – Range of hue jittering. If it is a float instead of a tuple like (min, max), the range of hue jittering will be (-hue_range, +hue_range). Default: 40.

transform(*results*)

transform function.

Parameters **results** (*dict*) – A dict containing the necessary information and data for augmentation.

Returns A dict containing the processed data and information.

Return type dict

1.35.56 LoadPairedImageFromFile

class mmedit.datasets.transforms.**LoadPairedImageFromFile**(*key: str*, *domain_a: str = 'A'*, *domain_b: str = 'B'*, *color_type: str = 'color'*, *channel_order: str = 'bgr'*, *imdecode_backend: Optional[str] = None*, *use_cache: bool = False*, *to_float32: bool = False*, *to_y_channel: bool = False*, *save_original_img: bool = False*, *file_client_args: Optional[dict] = None*)

Load a pair of images from file.

Each sample contains a pair of images, which are concatenated in the w dimension (a|b). This is a special loading class for generation paired dataset. It loads a pair of images as the common loader does and crops it into two images with the same shape in different domains.

Required key is "pair_path". Added or modified keys are "pair", "pair_ori_shape", "ori_pair", "img_{domain_a}", "img_{domain_b}", "img_{domain_a}_path", "img_{domain_b}_path", "img_{domain_a}_ori_shape", "img_{domain_b}_ori_shape", "ori_img_{domain_a}" and "ori_img_{domain_b}".

Parameters

- **key** (*str*) – Keys in results to find corresponding path.
- **domain_a** (*str, Optional*) – One of the paired image domain. Defaults to 'A'.
- **domain_b** (*str, Optional*) – The other of the paired image domain. Defaults to 'B'.
- **color_type** (*str*) – The flag argument for :func:mmcv.imfrombytes. Defaults to 'color'.

- **channel_order** (*str*) – Order of channel, candidates are 'bgr' and 'rgb'. Default: 'bgr'.
- **imdecode_backend** (*str*) – The image decoding backend type. The backend argument for :func:mmcv.imfrombytes. See :func:mmcv.imfrombytes for details. candidates are 'cv2', 'turbojpeg', 'pillow', and 'tifffile'. Defaults to None.
- **use_cache** (*bool*) – If True, load all images at once. Default: False.
- **to_float32** (*bool*) – Whether to convert the loaded image to a float32 numpy array. If set to False, the loaded image is an uint8 array. Defaults to False.
- **to_y_channel** (*bool*) – Whether to convert the loaded image to y channel. Only support 'rgb2ycbcr' and 'rgb2ycbcr' Defaults to False.
- **file_client_args** (*dict*) – Arguments to instantiate a FileClient. If not specified, will infer from file uri. See mmengine.fileio.FileClient for details. Defaults to None.
- **io_backend** (*str, optional*) – io backend where images are store. Defaults to None.

transform(*results: dict*) → dict

Functions to load paired images.

Parameters **results** (*dict*) – A dict containing the necessary information and data for augmentation.

Returns A dict containing the processed data and information.

Return type dict

1.35.57 CenterCropLongEdge

class mmedit.datasets.transforms.**CenterCropLongEdge**(*keys='img'*)

Center crop the given image by the long edge.

Parameters **keys** (*list[str]*) – The images to be cropped.

transform(*results*)

Call function.

Parameters **results** (*dict*) – A dict containing the necessary information and data for augmentation.

Returns A dict containing the processed data and information.

Return type dict

1.35.58 RandomCropLongEdge

class mmedit.datasets.transforms.**RandomCropLongEdge**(*keys='img'*)

Random crop the given image by the long edge.

Parameters **keys** (*list[str]*) – The images to be cropped.

transform(*results*)

Call function.

Parameters **results** (*dict*) – A dict containing the necessary information and data for augmentation.

Returns A dict containing the processed data and information.

Return type dict

1.35.59 NumpyPad

class mmedit.datasets.transforms.**NumpyPad**(*keys*, *padding*, ***kwargs*)

Numpy Padding.

In this augmentation, numpy padding is adopted to customize padding augmentation. Please carefully read the numpy manual in: https://numpy.org/doc/stable/reference/generated/numpy.pad.html

If you just hope a single dimension to be padded, you must set `padding` like this:

```
padding = ((2, 2), (0, 0), (0, 0))
```

In this case, if you adopt an input with three dimension, only the first dimension will be padded.

Parameters

- **keys** (*Union[str, List[str]]*) – The images to be padded.
- **padding** (*int | tuple(int)*) – Please refer to the args `pad_width` in `numpy.pad`.

transform(*results*)

Call function.

Parameters **results** (*dict*) – A dict containing the necessary information and data for augmentation.

Returns A dict containing the processed data and information.

Return type dict

1.36 mmedit.engine.hooks

ReduceLRSchedulerHook	A hook to update learning rate.
BasicVisualizationHook	Basic hook that invoke visualizers during validation and test.
GenVisualizationHook	Generation Visualization Hook.
ExponentialMovingAverageHook	Exponential Moving Average Hook.
GenIterTimerHook	GenIterTimerHooks inherits from *:class:~mmengine.hooks.IterTimerHook* and overwrites *:meth:self._after_iter*.
PGGANFetchDataHook	PGGAN Fetch Data Hook.
PickleDataHook	Pickle Useful Data Hook.

1.36.1 ReduceLRSchedulerHook

class mmedit.engine.hooks.**ReduceLRSchedulerHook**(*val_metric: Optional[str] = None*, *by_epoch=True*, *interval=1*)

A hook to update learning rate.

Parameters

- **val_metric** (*str*) – The metric of validation. If val_metric is not None, we check val_metric to reduce learning. Default: None.
- **by_epoch** (*bool*) – Whether to update by epoch. Default: True.
- **interval** (*int*) – The interval of iterations to update. Default: 1.

after_train_epoch(*runner: mmengine.runner.runner.Runner*)

Call step function for each scheduler after each train epoch.

Parameters **runner** (*Runner*) – The runner of the training process.

after_train_iter(*runner: mmengine.runner.runner.Runner*, *batch_idx: int*, *data_batch: Optional[Sequence[dict]] = None*, *outputs: Optional[dict] = None*) → None

Call step function for each scheduler after each iteration.

Parameters

- **runner** (*Runner*) – The runner of the training process.
- **batch_idx** (*int*) – The index of the current batch in the train loop.
- **data_batch** (*Sequence[dict], optional*) – Data from dataloader. In order to keep this interface consistent with other hooks, we keep `data_batch` here. Defaults to None.
- **outputs** (*dict, optional*) – Outputs from model. In order to keep this interface consistent with other hooks, we keep `data_batch` here. Defaults to None.

after_val_epoch(*runner*, *metrics: Optional[Dict[str, float]] = None*)

Call step function for each scheduler after each validation epoch.

Parameters

- **runner** (*Runner*) – The runner of the training process.
- **metrics** (*dict, optional*) – The metrics of validation. Default: None.

1.36.2 BasicVisualizationHook

class mmedit.engine.hooks.**BasicVisualizationHook**(*interval: dict = {}*, *on_train=False*, *on_val=True*, *on_test=True*)

Basic hook that invoke visualizers during validation and test.

Parameters

- **interval** (*int | dict*) – Visualization interval. Default: {}.
- **on_train** (*bool*) – Whether to call hook during train. Default to False.
- **on_val** (*bool*) – Whether to call hook during validation. Default to True.
- **on_test** (*bool*) – Whether to call hook during test. Default to True.

1.36.3 GenVisualizationHook

class mmedit.engine.hooks.**GenVisualizationHook**(*interval: int = 1000*, *vis_kwargs_list: Optional[Tuple[List[dict], dict]] = None*, *fixed_input: bool = True*, *n_samples: Optional[int] = 64*, *n_row: Optional[int] = 8*, *message_hub_vis_kwargs: Optional[Tuple[str, dict, List[str], List[Dict]]] = None*, *save_at_test: bool = True*, *max_save_at_test: int = 100*, *test_vis_keys: Optional[Union[str, List[str]]] = None*, *show: bool = False*, *wait_time: float = 0*)

Generation Visualization Hook. Used to visual output samples in training, validation and testing. In this hook, we use a list called *sample_kwargs_list* to control how to generate samples and how to visualize them. Each element in *sample_kwargs_list*, called *sample_kwargs*, may contains the following keywords:

- **Required key words:**
  - **'type': Value must be string. Denotes what kind of sampler is used to** generate image. Refers to *:meth:~mmgen.core.sampler.get_sampler*.
- **Optional key words (If not passed, will use the default value):**
  - 'n_rows': Value must be int. The number of images in one row.
  - 'num_samples': Value must be int. The number of samples to visualize.
  - **'vis_mode': Value must be string. How to visualize the generated** samples (e.g. image, gif).
  - **'fixed_input': Value must be bool. Whether use the fixed input** during the loop.
  - 'draw_gt': Value must be bool. Whether save the real images.
  - **'target_keys': Value must be string or list of string. The keys of** the target image to visualize.
  - **'name': Value must be string. If not passed, will use** *sample_kwargs['type']* as default.

For convenience, we also define a group of alias of samplers' type for models supported in MMEditing. Refers to *:attr:self.SAMPLER_TYPE_MAPPING*.

Example

```
>>> # for GAN models
>>> custom_hooks = [
>>>     dict(
>>>         type='GenVisualizationHook',
>>>         interval=1000,
>>>         fixed_input=True,
>>>         vis_kwargs_list=dict(type='GAN', name='fake_img'))]
>>> # for Translation models
>>> custom_hooks = [
>>>     dict(
>>>         type='GenVisualizationHook',
>>>         interval=10,
>>>         fixed_input=False,
>>>         vis_kwargs_list=[dict(type='Translation',
>>>                               name='translation_train',
>>>                               n_samples=6, draw_gt=True,
>>>                               n_rows=3),
```

(continues on next page)

(continued from previous page)

```
>>>                                 dict(type='TranslationVal',
>>>                                      name='translation_val',
>>>                                      n_samples=16, draw_gt=True,
>>>                                      n_rows=4)])]
```

NOTE: user-defined vis_kwargs > vis_kwargs_mapping > hook init args

Parameters

- **interval** (*int*) – Visualization interval. Default: 1000.
- **sampler_kwargs_list** (*Tuple[List[dict], dict]*) – The list of sampling behavior to generate images.
- **fixed_input** (*bool*) – The default action of whether use fixed input to generate samples during the loop. Defaults to True.
- **n_samples** (*Optional[int]*) – The default value of number of samples to visualize. Defaults to 64.
- **n_row** (*Optional[int]*) – The default value of number of images in each row in the visualization results. Defaults to 8.
- **(Optional[Tuple[str** (*message_hub_vis_kwargs*) – List[Dict]]]): Key arguments visualize images in message hub. Defaults to None.
- **dict** – List[Dict]]]): Key arguments visualize images in message hub. Defaults to None.
- **List[str]** – List[Dict]]]): Key arguments visualize images in message hub. Defaults to None.

:param [List[Dict]]]): Key arguments visualize images in message hub.] Defaults to None.

Parameters

- **save_at_test** (*bool*) – Whether save images during test. Defaults to True.
- **max_save_at_test** (*int*) – Maximum number of samples saved at test time. If None is passed, all samples will be saved. Defaults to 100.
- **show** (*bool*) – Whether to display the drawn image. Default to False.
- **wait_time** (*float*) – The interval of show (s). Defaults to 0.

after_test_iter(*runner: mmengine.runner.runner.Runner*, *batch_idx: int*, *data_batch: dict*, *outputs*)

Visualize samples after test iteraiton.

Parameters

- **runner** (*Runner*) – The runner of the training process.
- **batch_idx** (*int*) – The index of the current batch in the test loop.
- **data_batch** (*dict, optional*) – Data from dataloader. Defaults to None.
- **outputs** – outputs of the generation model Defaults to None.

after_train_iter(*runner: mmengine.runner.runner.Runner*, *batch_idx: int*, *data_batch: dict = None*, *outputs: Optional[dict] = None*) → None

Visualize samples after train iteration.

Parameters

- **runner** (*Runner*) – The runner of the training process.
- **batch_idx** (*int*) – The index of the current batch in the train loop.
- **data_batch** (*dict*) – Data from dataloader. Defaults to None.
- **outputs** (*dict, optional*) – Outputs from model. Defaults to None.

after_val_iter(*runner: mmengine.runner.runner.Runner*, *batch_idx: int*, *data_batch: dict*, *outputs*) → None

GenVisualizationHook do not support visualize during validation.

Parameters

- **runner** (*Runner*) – The runner of the training process.
- **batch_idx** (*int*) – The index of the current batch in the test loop.
- **data_batch** (*Sequence[dict], optional*) – Data from dataloader. Defaults to None.
- **outputs** – outputs of the generation model

vis_from_message_hub(*batch_idx: int*, *color_order: str*, *target_mean: Sequence[Union[float, int]]*, *target_std: Sequence[Union[float, int]]*)

Visualize samples from message hub.

Parameters

- **batch_idx** (*int*) – The index of the current batch in the test loop.
- **color_order** (*str*) – The color order of generated images.
- **target_mean** (*Sequence[Union[float, int]]*) – The original mean of the image tensor before preprocessing. Image will be re-shifted to `target_mean` before visualizing.
- **target_std** (*Sequence[Union[float, int]]*) – The original std of the image tensor before preprocessing. Image will be re-scaled to `target_std` before visualizing.

vis_sample(*runner: mmengine.runner.runner.Runner*, *batch_idx: int*, *data_batch: dict*, *outputs: Optional[dict] = None*) → None

Visualize samples.

Parameters

- **runner** (*Runner*) – The runner conatians model to visualize.
- **batch_idx** (*int*) – The index of the current batch in loop.
- **data_batch** (*dict*) – Data from dataloader. Defaults to None.
- **outputs** (*dict, optional*) – Outputs from model. Defaults to None.

1.36.4 ExponentialMovingAverageHook

class mmedit.engine.hooks.**ExponentialMovingAverageHook**(*module_keys*, *interp_mode='lerp'*, *interp_cfg=None*, *interval=- 1*, *start_iter=0*)

Exponential Moving Average Hook.

Exponential moving average is a trick that widely used in current GAN literature, e.g., PGGAN, StyleGAN, and BigGAN. This general idea of it is maintaining a model with the same architecture, but its parameters are updated as a moving average of the trained weights in the original model. In general, the model with moving averaged weights achieves better performance.

Parameters

- **module_keys** (*str | tuple[str]*) – The name of the ema model. Note that we require these keys are followed by '_ema' so that we can easily find the original model by discarding the last four characters.
- **interp_mode** (*str, optional*) – Mode of the interpolation method. Defaults to 'lerp'.
- **interp_cfg** (*dict | None, optional*) – Set arguments of the interpolation function. Defaults to None.
- **interval** (*int, optional*) – Evaluation interval (by iterations). Default: -1.
- **start_iter** (*int, optional*) – Start iteration for ema. If the start iteration is not reached, the weights of ema model will maintain the same as the original one. Otherwise, its parameters are updated as a moving average of the trained weights in the original model. Default: 0.

after_train_iter(*runner: mmengine.runner.runner.Runner*, *batch_idx: int*, *data_batch: Optional[Sequence[dict]] = None*, *outputs: Optional[dict] = None*) → None

This is the function to perform after each training iteration.

Parameters

- **runner** (*Runner*) – runner to drive the pipeline
- **batch_idx** (*int*) – the id of batch
- **data_batch** (*DATA_BATCH, optional*) – data batch. Defaults to None.
- **outputs** (*Optional[dict], optional*) – output. Defaults to None.

before_run(*runner: mmengine.runner.runner.Runner*)

This is the function perform before each run.

Parameters **runner** (*Runner*) – runner used to drive the whole pipeline

Raises **RuntimeError** – error message

every_n_iters(*runner: mmengine.runner.runner.Runner*, *n: int*)

This is the function to perform every n iterations.

Parameters

- **runner** (*Runner*) – runner used to drive the whole pipeline
- **n** (*int*) – the number of iterations

Returns the latest iterations

Return type int

static lerp(*a*, *b*, *momentum=0.999*, *momentum_nontrainable=0.0*, *trainable=True*)

This is the function to perform linear interpolation between a and b.

Parameters

- **a** (*float*) – number a
- **b** (*float*) – bumber b
- **momentum** (*float, optional*) – momentum. Defaults to 0.999.
- **momentum_nontrainable** (*float, optional*) – Defaults to 0.
- **trainable** (*bool, optional*) – trainable flag. Defaults to True.

Returns _description_

Return type _type_

1.36.5 GenIterTimerHook

class mmedit.engine.hooks.**GenIterTimerHook**

GenIterTimerHooks inherits from *:class:~mmengine.hooks.IterTimerHook* and overwrites *:meth:self._after_iter*.

This hooks should be used along with *:class:mmedit.engine.runners.loops.GenValLoop* and *:class:mmedit.engine.runners.loops.GenTestLoop*.

1.36.6 PGGANFetchDataHook

class mmedit.engine.hooks.**PGGANFetchDataHook**

PGGAN Fetch Data Hook.

Parameters **interval** (*int, optional*) – The interval of calling this hook. If set to -1, the visualization hook will not be called. Defaults to 1.

before_train_iter(*runner*, *batch_idx: int*, *data_batch: Optional[Sequence[dict]] = None*) → None

All subclasses should override this method, if they need any operations before each training iteration.

Parameters

- **runner** (*Runner*) – The runner of the training process.
- **batch_idx** (*int*) – The index of the current batch in the train loop.
- **data_batch** (*dict or tuple or list, optional*) – Data from dataloader.

update_dataloader(*dataloader: torch.utils.data.dataloader.DataLoader*, *curr_scale: int*) → Optional[torch.utils.data.dataloader.DataLoader]

Updata the data loader.

Parameters

- **dataloader** (*DataLoader*) – The dataloader to be updated.
- **curr_scale** (*int*) – The current scale of the generated image.

Returns

The updated dataloader. If the dataloader do not need to update, return None.

Return type Optional[DataLoader]

1.36.7 PickleDataHook

class mmedit.engine.hooks.**PickleDataHook**(*output_dir*, *data_name_list*, *interval=- 1*, *before_run=False*, *after_run=False*, *filename_tmpl='iter_{}.pkl'*)

Pickle Useful Data Hook.

This hook will be used in SinGAN training for saving some important data that will be used in testing or inference.

Parameters

- **output_dir** (*str*) – The output path for saving pickled data.

- **data_name_list** (*list[str]*) – The list contains the name of results in outputs dict.
- **interval** (*int*) – The interval of calling this hook. If set to -1, the PickleDataHook will not be called during training. Default: -1.
- **before_run** (*bool, optional*) – Whether to save before running. Defaults to False.
- **after_run** (*bool, optional*) – Whether to save after running. Defaults to False.
- **filename_tmpl** (*str, optional*) – Format string used to save images. The output file name will be formatted as this args. Defaults to 'iter_{}.pkl'.

after_run(*runner*)

The behavior after each train iteration.

Parameters **runner** (*object*) – The runner.

after_train_iter(*runner*, *batch_idx: int*, *data_batch: Optional[Sequence[dict]] = None*, *outputs: Optional[dict] = None*)

The behavior after each train iteration.

Parameters

- **runner** (*Runner*) – The runner of the training process.
- **batch_idx** (*int*) – The index of the current batch in the train loop.
- **data_batch** (*Sequence[dict], optional*) – Data from dataloader. Defaults to None.
- **outputs** (*dict, optional*) – Outputs from model. Defaults to None.

before_run(*runner*)

The behavior after each train iteration.

Parameters **runner** (*object*) – The runner.

1.37 mmedit.engine.optimizers

MultiOptimWrapperConstructor	OptimizerConstructor for GAN models.
PGGANOptimWrapperConstructor	OptimizerConstructor for PGGAN models.
SinGANOptimWrapperConstructor	OptimizerConstructor for SinGAN models.

1.37.1 MultiOptimWrapperConstructor

class mmedit.engine.optimizers.**MultiOptimWrapperConstructor**(*optim_wrapper_cfg: dict*, *paramwise_cfg=None*)

OptimizerConstructor for GAN models. This class construct optimizer for the submodules of the model separately, and return a :class:~ *mmengine.optim.OptimWrapperDict*.

Example

```
>>> # build GAN model
>>> model = dict(
>>>     type='GANModel',
>>>     num_classes=10,
>>>     generator=dict(type='Generator'),
>>>     discriminator=dict(type='Discriminator'))
>>> gan_model = MODELS.build(model)
>>> # build constructor
>>> optim_wrapper = dict(
>>>     constructor='MultiOptimWrapperConstructor',
>>>     generator=dict(
>>>         type='OptimWrapper',
>>>         accumulative_counts=1,
>>>         optimizer=dict(type='Adam', lr=0.0002,
>>>                        betas=(0.5, 0.999))),
>>>     discriminator=dict(
>>>         type='OptimWrapper',
>>>         accumulative_counts=1,
>>>         optimizer=dict(type='Adam', lr=0.0002,
>>>                            betas=(0.5, 0.999))))
>>> optim_dict_builder = MultiOptimWrapperConstructor(optim_wrapper)
>>> # build optim wrapper dict
>>> optim_wrapper_dict = optim_dict_builder(gan_model)
```

Parameters

- **optim_wrapper_cfg_dict** (*dict*) – Config of the optimizer wrapper.
- **paramwise_cfg** (*dict*) – Config of parameter-wise settings. Default: None.

1.37.2 PGGANOptimWrapperConstructor

class mmedit.engine.optimizers.**PGGANOptimWrapperConstructor**(*optim_wrapper_cfg: dict*, *paramwise_cfg: Optional[dict] = None*)

OptimizerConstructor for PGGAN models. Set optimizers for each stage of PGGAN. All submodule must be contained in a :class:~`torch.nn.ModuleList` named 'blocks'. And we access each submodule by *MODEL.blocks[SCALE]*, where *MODLE* is generator or discriminator, and the scale is the index of the resolution scale.

More detail about the resolution scale and naming rule please refers to :class:~`mmgen.models.PGGANGenerator` and :class:~`mmgen.models.PGGANDiscriminator`.

Example

```
>>> # build PGGAN model
>>> model = dict(
>>>     type='ProgressiveGrowingGAN',
>>>     data_preprocessor=dict(type='GANDataPreprocessor'),
>>>     noise_size=512,
>>>     generator=dict(type='PGGANGenerator', out_scale=1024,
>>>                    noise_size=512),
>>>     discriminator=dict(type='PGGANDiscriminator', in_scale=1024),
>>>     nkimgs_per_scale={
>>>         '4': 600,
>>>         '8': 1200,
>>>         '16': 1200,
>>>         '32': 1200,
>>>         '64': 1200,
>>>         '128': 1200,
>>>         '256': 1200,
>>>         '512': 1200,
>>>         '1024': 12000,
>>>     },
>>>     transition_kimgs=600,
>>>     ema_config=dict(interval=1))
>>> pggan = MODELS.build(model)
>>> # build constructor
>>> optim_wrapper = dict(
>>>     generator=dict(optimizer=dict(type='Adam', lr=0.001,
>>>                                   betas=(0., 0.99))),
>>>     discriminator=dict(
>>>         optimizer=dict(type='Adam', lr=0.001, betas=(0., 0.99))),
>>>     lr_schedule=dict(
>>>         generator={
>>>             '128': 0.0015,
>>>             '256': 0.002,
>>>             '512': 0.003,
>>>             '1024': 0.003
>>>         },
>>>         discriminator={
>>>             '128': 0.0015,
>>>             '256': 0.002,
>>>             '512': 0.003,
>>>             '1024': 0.003
>>>         }))
>>> optim_wrapper_dict_builder = PGGANOptimWrapperConstructor(
>>>     optim_wrapper)
>>> # build optim wrapper dict
>>> optim_wrapper_dict = optim_wrapper_dict_builder(pggan)
```

Parameters

- **optim_wrapper_cfg** (*dict*) – Config of the optimizer wrapper.
- **paramwise_cfg** (*Optional[dict]*) – Parameter-wise options.

1.37.3 SinGANOptimWrapperConstructor

class mmedit.engine.optimizers.**SinGANOptimWrapperConstructor**(*optim_wrapper_cfg: dict*, *paramwise_cfg: Optional[dict] = None*)

OptimizerConstructor for SinGAN models. Set optimizers for each submodule of SinGAN. All submodule must be contained in a :class:~`torch.nn.ModuleList` named 'blocks'. And we access each submodule by *MODEL.blocks[SCALE]*, where *MODLE* is generator or discriminator, and the scale is the index of the resolution scale.

More detail about the resolution scale and naming rule please refers to :class:~`mmgen.models.SinGANMultiScaleGenerator` and :class:~`mmgen.models.SinGANMultiScaleDiscriminator`.

Example

```
>>> # build SinGAN model
>>> model = dict(
>>>     type='SinGAN',
>>>     data_preprocessor=dict(
>>>         type='GANDataPreprocessor',
>>>         non_image_keys=['input_sample']),
>>>     generator=dict(
>>>         type='SinGANMultiScaleGenerator',
>>>         in_channels=3,
>>>         out_channels=3,
>>>         num_scales=2),
>>>     discriminator=dict(
>>>         type='SinGANMultiScaleDiscriminator',
>>>         in_channels=3,
>>>         num_scales=3))
>>> singan = MODELS.build(model)
>>> # build constructor
>>> optim_wrapper = dict(
>>>     generator=dict(optimizer=dict(type='Adam', lr=0.0005,
>>>                                   betas=(0.5, 0.999))),
>>>     discriminator=dict(
>>>         optimizer=dict(type='Adam', lr=0.0005,
>>>                        betas=(0.5, 0.999))))
>>> optim_wrapper_dict_builder = SinGANOptimWrapperConstructor(
>>>     optim_wrapper)
>>> # build optim wrapper dict
>>> optim_wrapper_dict = optim_wrapper_dict_builder(singan)
```

Parameters

- **optim_wrapper_cfg** (*dict*) – Config of the optimizer wrapper.
- **paramwise_cfg** (*Optional[dict]*) – Parameter-wise options.

1.38 mmedit.engine.runner

MultiValLoop	Loop for validation multi-datasets.
MultiTestLoop	Loop for validation multi-datasets.
GenTestLoop	Validation loop for generative models.
GenValLoop	Validation loop for generative models.
GenLogProcessor	GenLogProcessor inherits from *:class:~mmengine.logging.LogProcessor* and overwrites *:meth:self.get_log_after_iter*.

1.38.1 MultiValLoop

class mmedit.engine.runner.**MultiValLoop**(*runner*, *dataloader: Union[torch.utils.data.dataloader.DataLoader, Dict]*, *evaluator: Union[mmengine.evaluator.evaluator.Evaluator, Dict, List]*, *fp16: bool = False*)

Loop for validation multi-datasets.

Parameters

- **runner** (*Runner*) – A reference of runner.
- **dataloader** (*list[Dataloader or dic]*) – A dataloader object or a dict to build a dataloader.
- **evaluator** (*list[]*) – Used for computing metrics.
- **fp16** (*bool*) – Whether to enable fp16 validation. Defaults to False.

run()

Launch validation.

run_iter(*idx: int*, *data_batch: Sequence[dict]*)

Iterate one mini-batch.

Parameters

- **idx** (*int*) – The index of the current batch in the loop.
- **data_batch** (*Sequence[dict]*) – Batch of data from dataloader.

1.38.2 MultiTestLoop

class mmedit.engine.runner.**MultiTestLoop**(*runner*, *dataloader: Union[torch.utils.data.dataloader.DataLoader, Dict]*, *evaluator: Union[mmengine.evaluator.evaluator.Evaluator, Dict, List]*, *fp16: bool = False*)

Loop for validation multi-datasets.

Parameters

- **runner** (*Runner*) – A reference of runner.
- **dataloader** (*Dataloader or dict*) – A dataloader object or a dict to build a dataloader.
- **evaluator** (*Evaluator or dict or list*) – Used for computing metrics.

- **fp16** (*bool*) – Whether to enable fp16 validation. Defaults to False.

run()

Launch test.

run_iter(*idx: int*, *data_batch: Sequence[dict]*)

Iterate one mini-batch.

Parameters

- **idx** (*int*) – The index of the current batch in the loop.
- **data_batch** (*Sequence[dict]*) – Batch of data from dataloader.

1.38.3 GenTestLoop

class mmedit.engine.runner.**GenTestLoop**(*runner: mmengine.runner.runner.Runner*, *dataloader: Union[torch.utils.data.dataloader.DataLoader, Dict]*, *evaluator: Union[mmengine.evaluator.evaluator.Evaluator, Dict, List]*)

Validation loop for generative models. This class support evaluate metrics with different sample mode.

Parameters

- **runner** (*Runner*) – A reference of runner.
- **dataloader** (*Dataloader or dict*) – A dataloader object or a dict to build a dataloader.
- **evaluator** (*Evaluator or dict or list*) – Used for computing metrics.

run()

Launch validation. The evaluation process consists of four steps.

1. Prepare pre-calculated items for all metrics by calling `self.evaluator.prepare_metrics()`.
2. Get a list of metrics-sampler pair. Each pair contains a list of metrics with the same sampler mode and a shared sampler.
3. Generate images for the each metrics group. Loop for elements in each sampler and feed to the model as input by calling `self.run_iter()`.
4. Evaluate all metrics by calling `self.evaluator.evaluate()`.

run_iter(*idx*, *data_batch: dict*, *metrics: Sequence[mmengine.evaluator.metric.BaseMetric]*)

Iterate one mini-batch and feed the output to corresponding *metrics*.

Parameters

- **idx** (*int*) – Current idx for the input data.
- **data_batch** (*dict*) – Batch of data from dataloader.
- **metrics** (*Sequence[BaseMetric]*) – Specific metrics to evaluate.

1.38.4 GenValLoop

class mmedit.engine.runner.**GenValLoop**(*runner: mmengine.runner.runner.Runner*, *dataloader: Union[torch.utils.data.dataloader.DataLoader, Dict]*, *evaluator: Union[mmengine.evaluator.evaluator.Evaluator, Dict, List]*)

Validation loop for generative models. This class support evaluate metrics with different sample mode.

Parameters

- **runner** (*Runner*) – A reference of runner.
- **dataloader** (*Dataloader or dict*) – A dataloader object or a dict to build a dataloader.
- **evaluator** (*Evaluator or dict or list*) – Used for computing metrics.

run()

Launch validation. The evaluation process consists of four steps.

1. Prepare pre-calculated items for all metrics by calling `self.evaluator.prepare_metrics()`.
2. Get a list of metrics-sampler pair. Each pair contains a list of metrics with the same sampler mode and a shared sampler.
3. Generate images for the each metrics group. Loop for elements in each sampler and feed to the model as input by calling `self.run_iter()`.
4. Evaluate all metrics by calling `self.evaluator.evaluate()`.

run_iter(*idx*, *data_batch: dict*, *metrics: Sequence[mmengine.evaluator.metric.BaseMetric]*)

Iterate one mini-batch and feed the output to corresponding *metrics*.

Parameters

- **idx** (*int*) – Current idx for the input data.
- **data_batch** (*dict*) – Batch of data from dataloader.
- **metrics** (*Sequence[BaseMetric]*) – Specific metrics to evaluate.

1.38.5 GenLogProcessor

class mmedit.engine.runner.**GenLogProcessor**(*window_size=10*, *by_epoch=True*, *custom_cfg: Optional[List[dict]] = None*, *num_digits: int = 4*)

GenLogProcessor inherits from *:class:~mmengine.logging.LogProcessor* and overwrites *:meth:self.get_log_after_iter*.

This log processor should be used along with *:class:mmedit.engine.runners.loops.GenValLoop* and *:class:mmedit.engine.runners.loops.GenTestLoop*.

get_log_after_epoch(*runner*, *batch_idx: int*, *mode: str*) → Tuple[dict, str]

Format log string after validation or testing epoch.

We use *runner.val_loop.total_length* and *runner.test_loop.total_length* as the total number of iterations shown in log. If you want to know how *total_length* is calculated, please refers to *:meth:mmgen.core.runners.loops.GenValLoop.run* and *:meth:mmgen.core.runners.loops.GenTestLoop.run*.

Parameters

- **runner** (*Runner*) – The runner of validation/testing phase.

- **batch_idx** (*int*) – The index of the current batch in the current loop.
- **mode** (*str*) – Current mode of runner.

Returns Formatted log dict/string which will be recorded by `runner.message_hub` and `runner.visualizer`.

Return type Tuple(dict, str)

get_log_after_iter(*runner*, *batch_idx: int*, *mode: str*) → Tuple[dict, str]

Format log string after training, validation or testing epoch.

If *mode* is in 'val' or 'test', we use *runner.val_loop.total_length* and *runner.test_loop.total_length* as the total number of iterations shown in log. If you want to know how *total_length* is calculated, please refers to *:meth:mmedit.engine.runners.loops.GenValLoop.run* and *:meth:mmedit.engien.runners.loops.GenTestLoop.run*.

Parameters

- **runner** (*Runner*) – The runner of training phase.
- **batch_idx** (*int*) – The index of the current batch in the current loop.
- **mode** (*str*) – Current mode of runner, train, test or val.

Returns

Formatted log dict/string which will be recorded by `runner.message_hub` and `runner.visualizer`.

Return type Tuple(dict, str)

1.39 mmedit.engine.schedulers

LinearLrInterval	Linear learning rate scheduler for image generation.
ReduceLR	Decays the learning rate of each parameter group by linearly changing small multiplicative factor until the number of epoch reaches a pre-defined milestone: end.

1.39.1 LinearLrInterval

class mmedit.engine.schedulers.**LinearLrInterval**(**args*, *interval=1*, ***kwargs*)

Linear learning rate scheduler for image generation.

In the beginning, the learning rate is 'start_factor' defined in mmengine. We give a target learning rate 'end_factor' and a start point 'begin'. If :attr:self.by_epoch is True, 'begin' is calculated by epoch, otherwise, calculated by iteration." Before 'begin', we fix learning rate as 'start_factor'; After 'begin', we linearly update learning rate to 'end_factor'.

Parameters **interval** (*int*) – The interval to update the learning rate. Default: 1.

1.39.2 ReduceLR

class mmedit.engine.schedulers.**ReduceLR**(*optimizer*, *mode: str = 'min'*, *factor: float = 0.1*, *patience: int = 10*, *threshold: float = 0.0001*, *threshold_mode: str = 'rel'*, *cooldown: int = 0*, *min_lr: float = 0.0*, *eps: float = 1e-08*, ***kwargs*)

Decays the learning rate of each parameter group by linearly changing small multiplicative factor until the number of epoch reaches a pre-defined milestone: end.

Notice that such decay can happen simultaneously with other changes to the learning rate from outside this scheduler.

Note:

The learning rate of each parameter group will be update at regular intervals.

Parameters

- **optimizer** (*Optimizer or OptimWrapper*) – Wrapped optimizer.
- **mode** (*str, optional*) – One of *min*, *max*. In *min* mode, lr will be reduced when the quantity monitored has stopped decreasing; in *max* mode it will be reduced when the quantity monitored has stopped increasing. Default: 'min'.
- **factor** (*float, optional*) – Factor by which the learning rate will be reduced. new_lr = lr * factor. Default: 0.1.
- **patience** (*int, optional*) – Number of epochs with no improvement after which learning rate will be reduced. For example, if *patience = 2*, then we will ignore the first 2 epochs with no improvement, and will only decrease the LR after the 3rd epoch if the loss still hasn't improved then. Default: 10.
- **threshold** (*float, optional*) – Threshold for measuring the new optimum, to only focus on significant changes. Default: 1e-4.
- **threshold_mode** (*str, optional*) – One of *rel*, *abs*. In *rel* mode, dynamic_threshold = best * (1 + threshold) in 'max' mode or best * (1 - threshold) in *min* mode. In *abs* mode, dynamic_threshold = best + threshold in *max* mode or best - threshold in *min* mode. Default: 'rel'.
- **cooldown** (*int, optional*) – Number of epochs to wait before resuming normal operation after lr has been reduced. Default: 0.
- **min_lr** (*float, optional*) – Minimum LR value to keep. If LR after decay is lower than *min_lr*, it will be clipped to this value. Default: 0.
- **eps** (*float, optional*) – Minimal decay applied to lr. If the difference between new and old lr is smaller than eps, the update is ignored. Default: 1e-8.
- **begin** (*int*) – Step at which to start updating the learning rate. Defaults to 0.
- **end** (*int*) – Step at which to stop updating the learning rate.
- **last_step** (*int*) – The index of last step. Used for resume without state dict. Defaults to -1.
- **by_epoch** (*bool*) – Whether the scheduled learning rate is updated by epochs. Defaults to True.

1.40 mmedit.evaluation.metrics

niqe	

1.40.1 niqe

class mmedit.evaluation.metrics.niqe.**niqe**(*img*, *crop_border*, *input_order='HWC'*, *convert_to='y'*)

Calculate NIQE (Natural Image Quality Evaluator) metric.

Ref: Making a "Completely Blind" Image Quality Analyzer. This implementation could produce almost the same results as the official MATLAB codes: http://live.ece.utexas.edu/research/quality/niqe_release.zip

We use the official params estimated from the pristine dataset. We use the recommended block size (96, 96) without overlaps.

Parameters

- **img** (*np.ndarray*) – Input image whose quality needs to be computed. The input image must be in range [0, 255] with float/int type. The input_order of image can be 'HW' or 'HWC' or 'CHW'. (BGR order) If the input order is 'HWC' or 'CHW', it will be converted to gray or Y (of YCbCr) image according to the `convert_to` argument.
- **crop_border** (*int*) – Cropped pixels in each edge of an image. These pixels are not involved in the metric calculation.
- **input_order** (*str*) – Whether the input order is 'HW', 'HWC' or 'CHW'. Default: 'HWC'.
- **convert_to** (*str*) – Whether converted to 'y' (of MATLAB YCbCr) or 'gray'. Default: 'y'.

Returns NIQE result.

Return type niqe_result (float)

1.41 mmedit.evaluation.functional

InceptionV3	Pretrained InceptionV3 network returning feature maps.

1.41.1 InceptionV3

class mmedit.evaluation.functional.**InceptionV3**(*output_blocks=[3]*, *resize_input=True*, *normalize_input=True*, *requires_grad=False*, *use_fid_inception=True*, *load_fid_inception=True*)

Pretrained InceptionV3 network returning feature maps.

forward(*inp*)

Get Inception feature maps.

Parameters **inp** (*torch.Tensor*) – Input tensor of shape Bx3xHxW. Values are expected to be in range (0, 1)

Returns Corresponding to the selected output block, sorted ascending by index.

Return type list(torch.Tensor)

gauss_gradient	Gaussian gradient.
disable_gpu_fuser_on_pt19	On PyTorch 1.9 a CUDA fuser bug prevents the Inception JIT model to run.
load_inception	Load Inception Model from given `inception_args` and `metric`.
prepare_vgg_feat	Prepare vgg feature for the input metric.
prepare_inception_feat	Prepare inception feature for the input metric.

1.41.2 mmedit.evaluation.functional.gauss_gradient

`mmedit.evaluation.functional.`**`gauss_gradient`**(*img*, *sigma*)

Gaussian gradient.

From https://www.mathworks.com/matlabcentral/mlc-downloads/downloads/ submissions/8060/versions/2/previews/gaussgradient/gaussgradient.m/ index.html

Parameters

- **`img`** (*`np.ndarray`*) – Input image.
- **`sigma`** (*`float`*) – Standard deviation of the gaussian kernel.

Returns Gaussian gradient of input *img*.

Return type np.ndarray

1.41.3 mmedit.evaluation.functional.disable_gpu_fuser_on_pt19

`mmedit.evaluation.functional.`**`disable_gpu_fuser_on_pt19`**()

On PyTorch 1.9 a CUDA fuser bug prevents the Inception JIT model to run.

Refers to: https://github.com/GaParmar/clean-fid/blob/5e1e84cdea9654b9ac7189306dfa4057ea2213d8/cleanfid/inception_torchscript.py#L9 # noqa https://github.com/GaParmar/clean-fid/issues/5 https://github.com/pytorch/pytorch/issues/64062

1.41.4 mmedit.evaluation.functional.load_inception

`mmedit.evaluation.functional.`**`load_inception`**(*inception_args*, *metric*)

Load Inception Model from given `inception_args` and `metric`.

This function would try to load Inception under the guidance of 'type' given in *inception_args*, if not given, we would try best to load Tero's ones. In detailly, we would first try to load the model from disk with the given 'inception_path', and then try to download the checkpoint from 'inception_url'. If both method are failed, pytorch version of Inception would be loaded.

Parameters

- **`inception_args`** (*`dict`*) – Keyword args for inception net.
- **`metric`** (*`string`*) – Metric to use the Inception. This argument would influence the pytorch's Inception loading.

Returns Loaded Inception model. style (string): The version of the loaded Inception.

Return type model (torch.nn.Module)

1.41.5 mmedit.evaluation.functional.prepare_vgg_feat

mmedit.evaluation.functional.**prepare_vgg_feat**(*dataloader: torch.utils.data.dataloader.DataLoader*, *metric: mmengine.evaluator.metric.BaseMetric*, *data_preprocessor: Optional[torch.nn.modules.module.Module] = None*, *auto_save=True*) → numpy.ndarray

Prepare vgg feature for the input metric.

- If *metric.vgg_pkl* is an online path, try to download and load it. If cannot download or load, corresponding error will be raised.
- If *metric.vgg_pkl* is local path and file exists, try to load the file. If cannot load, corresponding error will be raised.
- If *metric.vgg_pkl* is local path and file not exists, we will extract the vgg feature manually and save to 'vgg_pkl'.
- If *metric.vgg_pkl* is not defined, we will extrace the vgg feature and save it to default cache dir with default name.

Parameters

- **datalaoder** (*Dataloader*) – The dataloader of real images.
- **metric** (*BaseMetric*) – The metric which needs vgg features.
- **data_preprocessor** (*Optional[nn.Module]*) – Data preprocessor of the module. Used to preprocess the real images. If not passed, real images will automatically normalized to [-1, 1]. Defaults to None.
- **Returns** – np.ndarray: Loaded vgg feature.

1.41.6 mmedit.evaluation.functional.prepare_inception_feat

mmedit.evaluation.functional.**prepare_inception_feat**(*dataloader: torch.utils.data.dataloader.DataLoader*, *metric: mmengine.evaluator.metric.BaseMetric*, *data_preprocessor: Optional[torch.nn.modules.module.Module] = None*, *capture_mean_cov: bool = False*, *capture_all: bool = False*) → dict

Prepare inception feature for the input metric.

- If *metric.inception_pkl* is an online path, try to download and load it. If cannot download or load, corresponding error will be raised.
- If *metric.inception_pkl* is local path and file exists, try to load the file. If cannot load, corresponding error will be raised.
- If *metric.inception_pkl* is local path and file not exists, we will extract the inception feature manually and save to 'inception_pkl'.
- If *metric.inception_pkl* is not defined, we will extrace the inception feature and save it to default cache dir with default name.

Parameters

- **datalaoder** (*Dataloader*) – The dataloader of real images.
- **metric** (*BaseMetric*) – The metric which needs inception features.
- **data_preprocessor** (*Optional[nn.Module]*) – Data preprocessor of the module. Used to preprocess the real images. If not passed, real images will automatically normalized to [-1, 1]. Defaults to None.
- **capture_mean_cov** (*bool*) – Whether save the mean and covariance of inception feature. Defaults to False.
- **capture_all** (*bool*) – Whether save the raw inception feature. If true, it will take a lot of time to save the inception feature. Defaults to False.

Returns Dict contains inception feature.

Return type dict

1.42 mmedit.models.base_models

BaseEditModel	Base model for image and video editing.
BaseGAN	Base class for GAN models.
BaseConditionalGAN	Base class for Conditional GAM models.
BaseMattor	Base class for trimap-based matting models.
BasicInterpolator	Basic model for video interpolation.
BaseTranslationModel	Base Translation Model.
OneStageInpaintor	Standard one-stage inpaintor with commonly used losses.
TwoStageInpaintor	Standard two-stage inpaintor with commonly used losses.
ExponentialMovingAverage	Implements the exponential moving average (EMA) of the model.
RampUpEMA	Implements the exponential moving average with ramping up momentum.

1.42.1 BaseEditModel

class mmedit.models.base_models.**BaseEditModel**(*generator*, *pixel_loss*, *train_cfg=None*, *test_cfg=None*, *init_cfg=None*, *data_preprocessor=None*)

Base model for image and video editing.

It must contain a generator that takes frames as inputs and outputs an interpolated frame. It also has a pixel-wise loss for training.

Parameters

- **generator** (*dict*) – Config for the generator structure.
- **pixel_loss** (*dict*) – Config for pixel-wise loss.
- **train_cfg** (*dict*) – Config for training. Default: None.
- **test_cfg** (*dict*) – Config for testing. Default: None.
- **init_cfg** (*dict, optional*) – The weight initialized config for BaseModule.

- **data_preprocessor** (*dict, optional*) – The pre-process config of BaseDataPreprocessor.

init_cfg

Initialization config dict.

Type dict, optional

data_preprocessor

Used for pre-processing data sampled by dataloader to the format accepted by *forward()*. Default: None.

Type BaseDataPreprocessor

forward(*inputs: torch.Tensor*, *data_samples: Optional[List[*mmedit.structures.edit_data_sample.EditDataSample*]] = None*, *mode: str = 'tensor'*, ***kwargs*)

Returns losses or predictions of training, validation, testing, and simple inference process.

forward method of BaseModel is an abstract method, its subclasses must implement this method.

Accepts inputs and data_samples processed by *data_preprocessor*, and returns results according to mode arguments.

During non-distributed training, validation, and testing process, forward will be called by BaseModel.train_step, BaseModel.val_step and BaseModel.val_step directly.

During distributed data parallel training process, MMSeparateDistributedDataParallel.train_step will first call DistributedDataParallel.forward to enable automatic gradient synchronization, and then call forward to get training loss.

Parameters

- **inputs** (*torch.Tensor*) – batch input tensor collated by *data_preprocessor*.
- **data_samples** (*List[BaseDataElement], optional*) – data samples collated by *data_preprocessor*.
- **mode** (*str*) – mode should be one of loss, predict and tensor. Default: 'tensor'.
  - loss: Called by train_step and return loss dict used for logging
  - predict: Called by val_step and test_step and return list of BaseDataElement results used for computing metric.
  - tensor: Called by custom use to get Tensor type results.

Returns

- If mode == loss, return a dict of loss tensor used for backward and logging.
- If mode == predict, return a list of BaseDataElement for computing metric and getting inference result.
- If mode == tensor, return a tensor or tuple of tensor or dict or tensor for custom use.

Return type ForwardResults

forward_inference(*inputs*, *data_samples=None*, ***kwargs*)

Forward inference. Returns predictions of validation, testing, and simple inference.

Parameters

- **inputs** (*torch.Tensor*) – batch input tensor collated by *data_preprocessor*.

- **data_samples** (*List[BaseDataElement], optional*) – data samples collated by *data_preprocessor*.

Returns predictions.

Return type List[*EditDataSample*]

forward_tensor(*inputs*, *data_samples=None*, ***kwargs*)

Forward tensor. Returns result of simple forward.

Parameters

- **inputs** (*torch.Tensor*) – batch input tensor collated by *data_preprocessor*.
- **data_samples** (*List[BaseDataElement], optional*) – data samples collated by *data_preprocessor*.

Returns result of simple forward.

Return type Tensor

forward_train(*inputs*, *data_samples=None*, ***kwargs*)

Forward training. Returns dict of losses of training.

Parameters

- **inputs** (*torch.Tensor*) – batch input tensor collated by *data_preprocessor*.
- **data_samples** (*List[BaseDataElement], optional*) – data samples collated by *data_preprocessor*.

Returns Dict of losses.

Return type dict

1.42.2 BaseGAN

class mmedit.models.base_models.**BaseGAN**(*generator: Union[Dict, torch.nn.modules.module.Module]*, *discriminator: Optional[Union[Dict, torch.nn.modules.module.Module]] = None*, *data_preprocessor: Optional[Union[dict, mmengine.config.config.Config]] = None*, *generator_steps: int = 1*, *discriminator_steps: int = 1*, *noise_size: Optional[int] = None*, *ema_config: Optional[Dict] = None*, *loss_config: Optional[Dict] = None*)

Base class for GAN models.

Parameters

- **generator** (*ModelType*) – The config or model of the generator.
- **discriminator** (*Optional[ModelType]*) – The config or model of the discriminator. Defaults to None.
- **data_preprocessor** (*Optional[Union[dict, Config]]*) – The pre-process config or GenDataPreprocessor.
- **generator_steps** (*int*) – The number of times the generator is completely updated before the discriminator is updated. Defaults to 1.
- **discriminator_steps** (*int*) – The number of times the discriminator is completely updated before the generator is updated. Defaults to 1.

- **ema_config** (*Optional[Dict]*) – The config for generator's exponential moving average setting. Defaults to None.

property device: torch.device

Get current device of the model.

Returns The current device of the model.

Return type torch.device

property discriminator_steps: int

The number of times the discriminator is completely updated before the generator is updated.

Type int

forward(*inputs: Tuple[Dict[str, Union[torch.Tensor, str, int]], torch.Tensor]*, *data_samples: Optional[list] = None*, *mode: Optional[str] = None*) → Sequence[mmengine.structures.base_data_element.BaseDataElement]

Sample images with the given inputs. If forward mode is 'ema' or 'orig', the image generated by corresponding generator will be returned. If forward mode is 'ema/orig', images generated by original generator and EMA generator will both be returned in a dict.

Parameters

- **batch_inputs** (*ForwardInputs*) – Dict containing the necessary information (e.g. noise, num_batches, mode) to generate image.
- **data_samples** (*Optional[list]*) – Data samples collated by `data_preprocessor`. Defaults to None.
- **mode** (*Optional[str]*) – *mode* is not used in *BaseGAN*. Defaults to None.

Returns A list of `EditDataSample` contain generated results.

Return type SampleList

static gather_log_vars(*log_vars_list: List[Dict[str, torch.Tensor]]*) → Dict[str, torch.Tensor]

Gather a list of log_vars. :param log_vars_list: List[Dict[str, Tensor]]

Returns Dict[str, Tensor]

property generator_steps: int

The number of times the generator is completely updated before the discriminator is updated.

Type int

noise_fn(*noise: Optional[Union[torch.Tensor, Callable]] = None*, *num_batches: int = 1*)

Sampling function for noise. There are three scenarios in this function:

- If *noise* is a callable function, sample *num_batches* of noise with passed *noise*.
- If *noise* is *None*, sample *num_batches* of noise from gaussian distribution.
- If *noise* is a *torch.Tensor*, directly return *noise*.

Parameters

- **noise** (*Union[Tensor, Callable, List[int], None]*) – You can directly give a batch of label through a `torch.Tensor` or offer a callable function to sample a batch of label data. Otherwise, the `None` indicates to use the default noise sampler. Defaults to *None*.
- **num_batches** (*int, optional*) – The number of batches label want to sample. If *label* is a Tensor, this will be ignored. Defaults to 1.

Returns Sampled noise tensor.

Return type Tensor

test_step(*data: dict*) → Sequence[mmengine.structures.base_data_element.BaseDataElement]

Gets the generated image of given data. Same as *val_step()*.

Parameters **data** (*dict*) – Data sampled from metric specific sampler. More detials in *Metrics* and *Evaluator*.

Returns Generated image or image dict.

Return type List[*EditDataSample*]

train_discriminator(*inputs: dict*, *data_samples: List[*mmedit.structures.edit_data_sample.EditDataSample*]*, *optimizer_wrapper: mmengine.optim.optimizer.optimizer_wrapper.OptimWrapper*) → Dict[str, torch.Tensor]

Training function for discriminator. All GANs should implement this function by themselves.

Parameters

- **inputs** (*dict*) – Inputs from dataloader.
- **data_samples** (*List[*EditDataSample*]*) – Data samples from dataloader.
- **optim_wrapper** (*OptimWrapper*) – OptimWrapper instance used to update model parameters.

Returns A `dict` of tensor for logging.

Return type Dict[str, Tensor]

train_generator(*inputs: dict*, *data_samples: List[*mmedit.structures.edit_data_sample.EditDataSample*]*, *optimizer_wrapper: mmengine.optim.optimizer.optimizer_wrapper.OptimWrapper*) → Dict[str, torch.Tensor]

Training function for discriminator. All GANs should implement this function by themselves.

Parameters

- **inputs** (*dict*) – Inputs from dataloader.
- **data_samples** (*List[*EditDataSample*]*) – Data samples from dataloader.
- **optim_wrapper** (*OptimWrapper*) – OptimWrapper instance used to update model parameters.

Returns A `dict` of tensor for logging.

Return type Dict[str, Tensor]

train_step(*data: dict*, *optim_wrapper: mmengine.optim.optimizer.optimizer_wrapper_dict.OptimWrapperDict*) → Dict[str, torch.Tensor]

Train GAN model. In the training of GAN models, generator and discriminator are updated alternatively. In MMEditing's design, *self.train_step* is called with data input. Therefore we always update discriminator, whose updating is relay on real data, and then determine if the generator needs to be updated based on the current number of iterations. More details about whether to update generator can be found in `should_gen_update()`.

Parameters

- **data** (*dict*) – Data sampled from dataloader.

- **optim_wrapper** (*OptimWrapperDict*) – OptimWrapperDict instance contains OptimWrapper of generator and discriminator.

Returns A `dict` of tensor for logging.

Return type Dict[str, torch.Tensor]

val_step(*data: dict*) → Sequence[mmengine.structures.base_data_element.BaseDataElement]

Gets the generated image of given data.

Calls `self.data_preprocessor(data)` and `self(inputs, data_sample, mode=None)` in order. Return the generated results which will be passed to evaluator.

Parameters **data** (*dict*) – Data sampled from metric specific sampler. More detials in *Metrics* and *Evaluator*.

Returns Generated image or image dict.

Return type SampleList

property with_ema_gen: bool

Whether the GAN adopts exponential moving average.

Returns

If *True*, means this GAN model is adopted to exponential moving average and vice versa.

Return type bool

1.42.3 BaseConditionalGAN

class mmedit.models.base_models.**BaseConditionalGAN**(*generator: Union[Dict, torch.nn.modules.module.Module]*, *discriminator: Optional[Union[Dict, torch.nn.modules.module.Module]] = None*, *data_preprocessor: Optional[Union[dict, mmengine.config.config.Config]] = None*, *generator_steps: int = 1*, *discriminator_steps: int = 1*, *noise_size: Optional[int] = None*, *num_classes: Optional[int] = None*, *ema_config: Optional[Dict] = None*, *loss_config: Optional[Dict] = None*)

Base class for Conditional GAM models.

Parameters

- **generator** (*ModelType*) – The config or model of the generator.
- **discriminator** (*Optional[ModelType]*) – The config or model of the discriminator. Defaults to None.
- **data_preprocessor** (*Optional[Union[dict, Config]]*) – The pre-process config or `GenDataPreprocessor`.
- **generator_steps** (*int*) – The number of times the generator is completely updated before the discriminator is updated. Defaults to 1.
- **discriminator_steps** (*int*) – The number of times the discriminator is completely updated before the generator is updated. Defaults to 1.
- **noise_size** (*Optional[int]*) – Size of the input noise vector. Default to None.

- **num_classes** (*Optional[int]*) – The number classes you would like to generate. Defaults to None.
- **ema_config** (*Optional[Dict]*) – The config for generator's exponential moving average setting. Defaults to None.

data_sample_to_label(*data_sample: List[*mmedit.structures.edit_data_sample.EditDataSample*]*) → Optional[torch.Tensor]

Get labels from input *data_sample* and pack to *torch.Tensor*. If no label is found in the passed *data_sample*, *None* would be returned.

Parameters **data_sample** (*List[EditDataSample]*) – Input data samples.

Returns Packed label tensor.

Return type Optional[torch.Tensor]

forward(*inputs: Tuple[Dict[str, Union[torch.Tensor, str, int]], torch.Tensor]*, *data_samples: Optional[list] = None*, *mode: Optional[str] = None*) → List[*mmedit.structures.edit_data_sample.EditDataSample*]

Sample images with the given inputs. If forward mode is 'ema' or 'orig', the image generated by corresponding generator will be returned. If forward mode is 'ema/orig', images generated by original generator and EMA generator will both be returned in a dict.

Parameters

- **inputs** (*ForwardInputs*) – Dict containing the necessary information (e.g. noise, num_batches, mode) to generate image.
- **data_samples** (*Optional[list]*) – Data samples collated by `data_preprocessor`. Defaults to None.
- **mode** (*Optional[str]*) – *mode* is not used in *BaseConditionalGAN*. Defaults to None.

Returns Generated images or image dict.

Return type List[*EditDataSample*]

label_fn(*label: Optional[Union[torch.Tensor, Callable, List[int]]] = None*, *num_batches: int = 1*) → torch.Tensor

Sampling function for label. There are three scenarios in this function:

- If *label* is a callable function, sample *num_batches* of labels with passed *label*.
- If *label* is *None*, sample *num_batches* of labels in range of *[0, self.num_classes-1]* uniformly.
- If *label* is a *torch.Tensor*, check the range of the tensor is in *[0, self.num_classes-1]*. If all values are in valid range, directly return *label*.

Parameters

- **label** (*Union[Tensor, Callable, List[int], None]*) – You can directly give a batch of label through a `torch.Tensor` or offer a callable function to sample a batch of label data. Otherwise, the `None` indicates to use the default label sampler. Defaults to *None*.
- **num_batches** (*int, optional*) – The number of batches label want to sample. If *label* is a Tensor, this will be ignored. Defaults to 1.

Returns Sampled label tensor.

Return type Tensor

train_discriminator(*inputs: dict*, *data_samples: List[mmedit.structures.edit_data_sample.EditDataSample]*, *optimizer_wrapper: mmengine.optim.optimizer.optimizer_wrapper.OptimWrapper*) → Dict[str, torch.Tensor]

Training function for discriminator. All GANs should implement this function by themselves.

Parameters

- **inputs** (*dict*) – Inputs from dataloader.
- **data_samples** (*List[EditDataSample]*) – Data samples from dataloader.
- **optim_wrapper** (*OptimWrapper*) – OptimWrapper instance used to update model parameters.

Returns A `dict` of tensor for logging.

Return type Dict[str, Tensor]

train_generator(*inputs: dict*, *data_samples: List[mmedit.structures.edit_data_sample.EditDataSample]*, *optimizer_wrapper: mmengine.optim.optimizer.optimizer_wrapper.OptimWrapper*) → Dict[str, torch.Tensor]

Training function for discriminator. All GANs should implement this function by themselves.

Parameters

- **inputs** (*dict*) – Inputs from dataloader.
- **data_samples** (*List[EditDataSample]*) – Data samples from dataloader.
- **optim_wrapper** (*OptimWrapper*) – OptimWrapper instance used to update model parameters.

Returns A `dict` of tensor for logging.

Return type Dict[str, Tensor]

1.42.4 BaseMattor

class mmedit.models.base_models.**BaseMattor**(*data_preprocessor: Union[dict, mmengine.config.config.Config]*, *backbone: dict*, *init_cfg: Optional[dict] = None*, *train_cfg: Optional[dict] = None*, *test_cfg: Optional[dict] = None*)

Base class for trimap-based matting models.

A matting model must contain a backbone which produces *pred_alpha*, a dense prediction with the same height and width of input image. In some cases (such as DIM), the model has a refiner which refines the prediction of the backbone.

Subclasses should overwrite the following functions:

- `_forward_train()`, to return a loss
- `_forward_test()`, to return a prediction
- `_forward()`, to return raw tensors

For test, this base class provides functions to resize inputs and post-process pred_alphas to get predictions

Parameters

- **backbone** (*dict*) – Config of backbone.

- **data_preprocessor** (*dict*) – Config of data_preprocessor. See MattorPreprocessor for details.
- **init_cfg** (*dict, optional*) – Initialization config dict.
- **train_cfg** (*dict*) – Config of training. Customized by subclassesCustomized bu In train_cfg, train_backbone should be specified. If the model has a refiner, train_refiner should be specified.
- **test_cfg** (*dict*) – Config of testing. In test_cfg, If the model has a refiner, train_refiner should be specified.

forward(*inputs: torch.Tensor*, *data_samples: Optional[Union[list, torch.Tensor]] = None*, *mode: str = 'tensor'*) → List[*mmedit.structures.edit_data_sample.EditDataSample*]

General forward function.

Parameters

- **inputs** (*torch.Tensor*) – A batch of inputs. with image and trimap concatenated alone channel dimension.
- **data_samples** (*List[EditDataSample], optional*) – A list of data samples, containing: - Ground-truth alpha / foreground / background to compute loss - other meta information
- **mode** (*str*) – mode should be one of loss, predict and tensor. Default: 'tensor'.
  - loss: Called by train_step and return loss dict used for logging
  - predict: Called by val_step and test_step and return list of BaseDataElement results used for computing metric.
  - tensor: Called by custom use to get Tensor type results.

Returns Sequence of predictions packed into EditDataElement

Return type List[EditDataElement]

postprocess(*batch_pred_alpha: torch.Tensor*, *data_samples: List[*mmedit.structures.edit_data_sample.EditDataSample*]*) → List[*mmedit.structures.edit_data_sample.EditDataSample*]

Post-process alpha predictions.

This function contains the following steps:

1. Restore padding or interpolation
2. Mask alpha prediction with trimap
3. Clamp alpha prediction to 0-1
4. Convert alpha prediction to uint8
5. Pack alpha prediction into EditDataSample

Currently only batch_size 1 is actually supported.

Parameters

- **batch_pred_alpha** (*torch.Tensor*) – A batch of predicted alpha of shape (N, 1, H, W).
- **data_samples** (*List[EditDataSample]*) – List of data samples.

Returns

A list of predictions. Each data sample contains a pred_alpha, which is a torch.Tensor with dtype=uint8, device=cuda:0

Return type List[*EditDataSample*]

resize_inputs(*batch_inputs*)

Pad or interpolate images and trimaps to multiple of given factor.

restore_size(*pred_alpha*, *data_sample*)

Restore the predicted alpha to the original shape.

The shape of the predicted alpha may not be the same as the shape of original input image. This function restores the shape of the predicted alpha.

Parameters

- **pred_alpha** (*torch.Tensor*) – A single predicted alpha of shape (1, H, W).
- **data_sample** (EditDataSample) – Data sample containing original shape as meta data.

Returns The reshaped predicted alpha.

Return type torch.Tensor

1.42.5 BasicInterpolator

class mmedit.models.base_models.**BasicInterpolator**(*generator*, *pixel_loss*, *train_cfg=None*, *test_cfg=None*, *required_frames=2*, *step_frames=1*, *init_cfg=None*, *data_preprocessor=None*)

Basic model for video interpolation.

It must contain a generator that takes frames as inputs and outputs an interpolated frame. It also has a pixel-wise loss for training.

Parameters

- **generator** (*dict*) – Config for the generator structure.
- **pixel_loss** (*dict*) – Config for pixel-wise loss.
- **train_cfg** (*dict*) – Config for training. Default: None.
- **test_cfg** (*dict*) – Config for testing. Default: None.
- **required_frames** (*int*) – Required frames in each process. Default: 2
- **step_frames** (*int*) – Step size of video frame interpolation. Default: 1
- **init_cfg** (*dict, optional*) – The weight initialized config for BaseModule.
- **data_preprocessor** (*dict, optional*) – The pre-process config of BaseDataPreprocessor.

init_cfg

Initialization config dict.

Type dict, optional

data_preprocessor

Used for pre-processing data sampled by dataloader to the format accepted by forward().

Type BaseDataPreprocessor

static merge_frames(*input_tensors*, *output_tensors*)

merge input frames and output frames.

Interpolate a frame between the given two frames.

Merged from [[in1, in2], [in2, in3], [in3, in4], ...] [[out1], [out2], [out3], ...]

to [in1, out1, in2, out2, in3, out3, in4, ...]

Parameters

- **input_tensors** (*Tensor*) – The input frames with shape [n, 2, c, h, w]
- **output_tensors** (*Tensor*) – The output frames with shape [n, 1, c, h, w].

Returns The final frames.

Return type list[np.array]

split_frames(*input_tensors*)

split input tensors for inference.

Parameters **input_tensors** (*Tensor*) – Tensor of input frames with shape [1, t, c, h, w]

Returns Split tensor with shape [t-1, 2, c, h, w]

Return type Tensor

1.42.6 BaseTranslationModel

class mmedit.models.base_models.**BaseTranslationModel**(*generator*, *discriminator*, *default_domain: str*, *reachable_domains: List[str]*, *related_domains: List[str]*, *data_preprocessor*, *discriminator_steps: int = 1*, *disc_init_steps: int = 0*, *real_img_key: str = 'real_img'*, *loss_config: Optional[dict] = None*)

Base Translation Model.

Translation models can transfer images from one domain to another. Domain information like *default_domain*, *reachable_domains* are needed to initialize the class. And we also provide query functions like *is_domain_reachable*, *get_other_domains*.

You can get a specific generator based on the domain, and by specifying *target_domain* in the forward function, you can decide the domain of generated images. Considering the difference among different image translation models, we only provide the external interfaces mentioned above. When you implement image translation with a specific method, you can inherit both *BaseTranslationModel* and the method (e.g BaseGAN) and implement abstract methods.

Parameters

- **default_domain** (*str*) – Default output domain.
- **reachable_domains** (*list[str]*) – Domains that can be generated by the model.
- **related_domains** (*list[str]*) – Domains involved in training and testing. *reachable_domains* must be contained in *related_domains*. However, related_domains may contain source domains that are used to retrieve source images from data_batch but not in reachable_domains.
- **discriminator_steps** (*int*) – The number of times the discriminator is completely updated before the generator is updated. Defaults to 1.

- **disc_init_steps** (*int*) – The number of initial steps used only to train discriminators.

forward(*img*, *test_mode=False*, ***kwargs*)

Forward function.

Parameters

- **img** (*tensor*) – Input image tensor.
- **test_mode** (*bool*) – Whether in test mode or not. Default: False.
- **kwargs** (*dict*) – Other arguments.

forward_test(*img*, *target_domain*, ***kwargs*)

Forward function for testing.

Parameters

- **img** (*tensor*) – Input image tensor.
- **target_domain** (*str*) – Target domain of output image.
- **kwargs** (*dict*) – Other arguments.

Returns Forward results.

Return type dict

forward_train(*img*, *target_domain*, ***kwargs*)

Forward function for training.

Parameters

- **img** (*tensor*) – Input image tensor.
- **target_domain** (*str*) – Target domain of output image.
- **kwargs** (*dict*) – Other arguments.

Returns Forward results.

Return type dict

get_module(*module*)

Get *nn.ModuleDict* to fit the *MMDistributedDataParallel* interface.

Parameters **module** (*MMDistributedDataParallel | nn.ModuleDict*) – The input module that needs processing.

Returns The ModuleDict of multiple networks.

Return type nn.ModuleDict

get_other_domains(*domain*)

get other domains.

init_weights(*pretrained=None*)

Initialize weights for the model.

Parameters **pretrained** (*str, optional*) – Path for pretrained weights. If given None, pretrained weights will not be loaded. Default: None.

is_domain_reachable(*domain*)

Whether image of this domain can be generated.

translation(*image*, *target_domain=None*, ***kwargs*)

Translation Image to target style.

Parameters

- **image** (*tensor*) – Image tensor with a shape of (N, C, H, W).
- **target_domain** (*str, optional*) – Target domain of output image. Default to None.

Returns Image tensor of target style.

Return type dict

1.42.7 OneStageInpaintor

class mmedit.models.base_models.**OneStageInpaintor**(*data_preprocessor: Union[dict, mmengine.config.config.Config]*, *encdec*, *disc=None*, *loss_gan=None*, *loss_gp=None*, *loss_disc_shift=None*, *loss_composed_percep=None*, *loss_out_percep=False*, *loss_l1_hole=None*, *loss_l1_valid=None*, *loss_tv=None*, *train_cfg=None*, *test_cfg=None*, *init_cfg: Optional[dict] = None*)

Standard one-stage inpaintor with commonly used losses.

An inpaintor must contain an encoder-decoder style generator to inpaint masked regions. A discriminator will be adopted when adversarial training is needed.

In this class, we provide a common interface for inpaintors. For other inpaintors, only some funcs may be modified to fit the input style or training schedule.

Parameters

- **data_preprocessor** (*dict*) – Config of data_preprocessor.
- **encdec** (*dict*) – Config for encoder-decoder style generator.
- **disc** (*dict*) – Config for discriminator.
- **loss_gan** (*dict*) – Config for adversarial loss.
- **loss_gp** (*dict*) – Config for gradient penalty loss.
- **loss_disc_shift** (*dict*) – Config for discriminator shift loss.
- **loss_composed_percep** (*dict*) – Config for perceptual and style loss with composed image as input.
- **loss_out_percep** (*dict*) – Config for perceptual and style loss with direct output as input.
- **loss_l1_hole** (*dict*) – Config for l1 loss in the hole.
- **loss_l1_valid** (*dict*) – Config for l1 loss in the valid region.
- **loss_tv** (*dict*) – Config for total variation loss.
- **train_cfg** (*dict*) – Configs for training scheduler. *disc_step* must be contained for indicates the discriminator updating steps in each training step.
- **test_cfg** (*dict*) – Configs for testing scheduler.
- **init_cfg** (*dict, optional*) – Initialization config dict.

forward(*inputs*, *data_samples*, *mode='tensor'*)

Forward function.

Parameters

- **inputs** (*torch.Tensor*) – batch input tensor collated by `data_preprocessor`.
- **data_samples** (*List[BaseDataElement], optional*) – data samples collated by `data_preprocessor`.
- **mode** (*str*) – mode should be one of `loss`, `predict` and `tensor`. Default: 'tensor'.
  - `loss`: Called by `train_step` and return loss `dict` used for logging
  - `predict`: Called by `val_step` and `test_step` and return list of `BaseDataElement` results used for computing metric.
  - `tensor`: Called by custom use to get `Tensor` type results.

Returns

- If `mode == loss`, return a `dict` of loss tensor used for backward and logging.
- If `mode == predict`, return a `list` of `BaseDataElement` for computing metric and getting inference result.
- If `mode == tensor`, return a tensor or `tuple` of tensor or `dict` or tensor for custom use.

Return type ForwardResults

forward_dummy(*x*)

Forward dummy function for getting flops.

Parameters **x** (*torch.Tensor*) – Input tensor with shape of (n, c, h, w).

Returns Results tensor with shape of (n, 3, h, w).

Return type torch.Tensor

forward_tensor(*inputs*, *data_samples*)

Forward function in tensor mode.

Parameters

- **inputs** (*torch.Tensor*) – Input tensor.
- **data_samples** (*List[dict]*) – List of data sample dict.

Returns

Direct output of the generator and composition of *fake_res* and ground-truth image.

Return type tuple

forward_test(*inputs*, *data_samples*)

Forward function for testing.

Parameters

- **inputs** (*torch.Tensor*) – Input tensor.
- **data_samples** (*List[dict]*) – List of data sample dict.

Returns

List of prediction saved in DataSample.

Return type predictions (List[DataSample])

forward_train(**args*, ***kwargs*)

Forward function for training.

In this version, we do not use this interface.

forward_train_d(*data_batch*, *is_real*, *is_disc*)

Forward function in discriminator training step.

In this function, we compute the prediction for each data batch (real or fake). Meanwhile, the standard gan loss will be computed with several proposed losses for stable training.

Parameters

- **data_batch** (*torch.Tensor*) – Batch of real data or fake data.
- **is_real** (*bool*) – If True, the gan loss will regard this batch as real data. Otherwise, the gan loss will regard this batch as fake data.
- **is_disc** (*bool*) – If True, this function is called in discriminator training step. Otherwise, this function is called in generator training step. This will help us to compute different types of adversarial loss, like LSGAN.

Returns Contains the loss items computed in this function.

Return type dict

generator_loss(*fake_res*, *fake_img*, *gt*, *mask*, *masked_img*)

Forward function in generator training step.

In this function, we mainly compute the loss items for generator with the given (fake_res, fake_img). In general, the *fake_res* is the direct output of the generator and the *fake_img* is the composition of direct output and ground-truth image.

Parameters

- **fake_res** (*torch.Tensor*) – Direct output of the generator.
- **fake_img** (*torch.Tensor*) – Composition of *fake_res* and ground-truth image.
- **gt** (*torch.Tensor*) – Ground-truth image.
- **mask** (*torch.Tensor*) – Mask image.
- **masked_img** (*torch.Tensor*) – Composition of mask image and ground-truth image.

Returns Dict contains the results computed within this function for visualization and dict contains the loss items computed in this function.

Return type tuple(dict)

train_step(*data: List[dict]*, *optim_wrapper*)

Train step function.

In this function, the inpaintor will finish the train step following the pipeline:

1. get fake res/image
2. optimize discriminator (if have)
3. optimize generator

If *self.train_cfg.disc_step > 1*, the train step will contain multiple iterations for optimizing discriminator with different input data and only one iteration for optimizing gerator after *disc_step* iterations for discriminator.

Parameters

- **data** (*List[dict]*) – Batch of data as input.
- **optim_wrapper** (*dict[torch.optim.Optimizer]*) – Dict with optimizers for generator and discriminator (if have).

Returns

Dict with loss, information for logger, the number of samples and results for visualization.

Return type dict

1.42.8 TwoStageInpaintor

class mmedit.models.base_models.**TwoStageInpaintor**(*data_preprocessor: Union[dict, mmengine.config.config.Config]*, *encdec: dict*, *disc=None*, *loss_gan=None*, *loss_gp=None*, *loss_disc_shift=None*, *loss_composed_percep=None*, *loss_out_percep=False*, *loss_l1_hole=None*, *loss_l1_valid=None*, *loss_tv=None*, *train_cfg=None*, *test_cfg=None*, *init_cfg: Optional[dict] = None*, *stage1_loss_type=('loss_l1_hole',)*, *stage2_loss_type=('loss_l1_hole', 'loss_gan')*, *input_with_ones=True*, *disc_input_with_mask=False*)

Standard two-stage inpaintor with commonly used losses. A two-stage inpaintor contains two encoder-decoder style generators to inpaint masked regions. Currently, we support these loss types in each of two stage inpaintors:

['loss_gan', 'loss_l1_hole', 'loss_l1_valid', 'loss_composed_percep', 'loss_out_percep', 'loss_tv'] The *stage1_loss_type* and *stage2_loss_type* should be chosen from these loss types.

Parameters

- **data_preprocessor** (*dict*) – Config of data_preprocessor.
- **encdec** (*dict*) – Config for encoder-decoder style generator.
- **disc** (*dict*) – Config for discriminator.
- **loss_gan** (*dict*) – Config for adversarial loss.
- **loss_gp** (*dict*) – Config for gradient penalty loss.
- **loss_disc_shift** (*dict*) – Config for discriminator shift loss.
- **loss_composed_percep** (*dict*) – Config for perceptual and style loss with composed image as input.
- **loss_out_percep** (*dict*) – Config for perceptual and style loss with direct output as input.
- **loss_l1_hole** (*dict*) – Config for l1 loss in the hole.
- **loss_l1_valid** (*dict*) – Config for l1 loss in the valid region.
- **loss_tv** (*dict*) – Config for total variation loss.
- **train_cfg** (*dict*) – Configs for training scheduler. *disc_step* must be contained for indicates the discriminator updating steps in each training step.
- **test_cfg** (*dict*) – Configs for testing scheduler.

- **init_cfg** (*dict, optional*) – Initialization config dict.
- **stage1_loss_type** (*tuple[str]*) – Contains the loss names used in the first stage model. Default: ('loss_l1_hole').
- **stage2_loss_type** (*tuple[str]*) – Contains the loss names used in the second stage model. Default: ('loss_l1_hole', 'loss_gan').
- **input_with_ones** (*bool*) – Whether to concatenate an extra ones tensor in input. Default: True.
- **disc_input_with_mask** (*bool*) – Whether to add mask as input in discriminator. Default: False.

calculate_loss_with_type(*loss_type*, *fake_res*, *fake_img*, *gt*, *mask*, *prefix='stage1_'*)

Calculate multiple types of losses.

Parameters

- **loss_type** (*str*) – Type of the loss.
- **fake_res** (*torch.Tensor*) – Direct results from model.
- **fake_img** (*torch.Tensor*) – Composited results from model.
- **gt** (*torch.Tensor*) – Ground-truth tensor.
- **mask** (*torch.Tensor*) – Mask tensor.
- **prefix** (*str, optional*) – Prefix for loss name. Defaults to 'stage1_'. # noqa

Returns Contain loss value with its name.

Return type dict

forward_tensor(*inputs*, *data_samples*)

Forward function in tensor mode.

Parameters

- **inputs** (*torch.Tensor*) – Input tensor.
- **data_samples** (*List[dict]*) – List of data sample dict.

Returns Dict contains output results.

Return type dict

train_step(*data: List[dict]*, *optim_wrapper*)

Train step function.

In this function, the inpaintor will finish the train step following the pipeline: 1. get fake res/image 2. optimize discriminator (if have) 3. optimize generator

If *self.train_cfg.disc_step > 1*, the train step will contain multiple iterations for optimizing discriminator with different input data and only one iteration for optimizing gerator after *disc_step* iterations for discriminator.

Parameters

- **data** (*List[dict]*) – Batch of data as input.
- **optim_wrapper** (*dict[torch.optim.Optimizer]*) – Dict with optimizers for generator and discriminator (if have).

Returns Dict with loss, information for logger, the number of samples and results for visualization.

Return type dict

two_stage_loss(*stage1_data*, *stage2_data*, *gt*, *mask*, *masked_img*)

Calculate two-stage loss.

Parameters

- **stage1_data** (*dict*) – Contain stage1 results.
- **stage2_data** (*dict*) – Contain stage2 results..
- **gt** (*torch.Tensor*) – Ground-truth image.
- **mask** (*torch.Tensor*) – Mask image.
- **masked_img** (*torch.Tensor*) – Composition of mask image and ground-truth image.

Returns Dict contains the results computed within this function for visualization and dict contains the loss items computed in this function.

Return type tuple(dict)

1.42.9 ExponentialMovingAverage

class mmedit.models.base_models.**ExponentialMovingAverage**(*model: torch.nn.modules.module.Module*, *momentum: float = 0.0002*, *interval: int = 1*, *device: Optional[torch.device] = None*, *update_buffers: bool = False*)

Implements the exponential moving average (EMA) of the model.

All parameters are updated by the formula as below:

$$Xema_{t+1} = (1 - momentum) * Xema_t + momentum * X_t$$

Parameters

- **model** (*nn.Module*) – The model to be averaged.
- **momentum** (*float*) – The momentum used for updating ema parameter. Defaults to 0.0002. Ema's parameter are updated with the formula $averaged\_param = (1 - momentum) * averaged\_param + momentum * source\_param$.
- **interval** (*int*) – Interval between two updates. Defaults to 1.
- **device** (*torch.device, optional*) – If provided, the averaged model will be stored on the `device`. Defaults to None.
- **update_buffers** (*bool*) – if True, it will compute running averages for both the parameters and the buffers of the model. Defaults to False.

avg_func(*averaged_param: torch.Tensor*, *source_param: torch.Tensor*, *steps: int*) → None

Compute the moving average of the parameters using exponential moving average.

Parameters

- **averaged_param** (*Tensor*) – The averaged parameters.
- **source_param** (*Tensor*) – The source parameters.
- **steps** (*int*) – The number of times the parameters have been updated.

sync_buffers(*model: torch.nn.modules.module.Module*) → None

Copy buffer from model to averaged model.

Parameters **model** (*nn.Module*) – The model whose parameters will be averaged.

sync_parameters(*model: torch.nn.modules.module.Module*) → None

Copy buffer and parameters from model to averaged model.

Parameters **model** (*nn.Module*) – The model whose parameters will be averaged.

1.42.10 RampUpEMA

class mmedit.models.base_models.**RampUpEMA**(*model: torch.nn.modules.module.Module*, *interval: int = 1*, *ema_kimg: int = 10*, *ema_rampup: float = 0.05*, *batch_size: int = 32*, *eps: float = 1e-08*, *start_iter: int = 0*, *device: Optional[torch.device] = None*, *update_buffers: bool = False*)

Implements the exponential moving average with ramping up momentum.

Ref: https://github.com/NVlabs/stylegan3/blob/master/training/training_loop.py # noqa

Parameters

- **model** (*nn.Module*) – The model to be averaged.
- **interval** (*int*) – Interval between two updates. Defaults to 1.
- **ema_kimg** (*int, optional*) – EMA kimgs. Defaults to 10.
- **ema_rampup** (*float, optional*) – Ramp up rate. Defaults to 0.05.
- **batch_size** (*int, optional*) – Global batch size. Defaults to 32.
- **eps** (*float, optional*) – Ramp up epsilon. Defaults to 1e-8.
- **start_iter** (*int, optional*) – EMA start iter. Defaults to 0.
- **device** (*torch.device, optional*) – If provided, the averaged model will be stored on the `device`. Defaults to None.
- **update_buffers** (*bool*) – if True, it will compute running averages for both the parameters and the buffers of the model. Defaults to False.

avg_func(*averaged_param: torch.Tensor*, *source_param: torch.Tensor*, *steps: int*) → None

Compute the moving average of the parameters using exponential moving average.

Parameters

- **averaged_param** (*Tensor*) – The averaged parameters.
- **source_param** (*Tensor*) – The source parameters.
- **steps** (*int*) – The number of times the parameters have been updated.

static rampup(*steps*, *ema_kimg=10*, *ema_rampup=0.05*, *batch_size=4*, *eps=1e-08*)

Ramp up ema momentum.

Ref: https://github.com/NVlabs/stylegan3/blob/a5a69f58294509598714d1e88c9646c3d7c6ec94/training/training_loop.py#L300-L308 # noqa

Parameters

- **steps** –

- **ema_kimg** (*int, optional*) – Half-life of the exponential moving average of generator weights. Defaults to 10.
- **ema_rampup** (*float, optional*) – EMA ramp-up coefficient.If set to None, then rampup will be disabled. Defaults to 0.05.
- **batch_size** (*int, optional*) – Total batch size for one training iteration. Defaults to 4.
- **eps** (*float, optional*) – Epsiolon to avoid `batch_size` divided by zero. Defaults to 1e-8.

Returns Updated momentum.

Return type dict

sync_buffers(*model: torch.nn.modules.module.Module*) → None

Copy buffer from model to averaged model.

Parameters **model** (*nn.Module*) – The model whose parameters will be averaged.

sync_parameters(*model: torch.nn.modules.module.Module*) → None

Copy buffer and parameters from model to averaged model.

Parameters **model** (*nn.Module*) – The model whose parameters will be averaged.

1.43 mmedit.models.data_preprocessors

EditDataPreprocessor	Basic data pre-processor used for collating and copying data to the target device in mmediting.
MattorPreprocessor	DataPreprocessor for matting models.
split_batch	reverse operation of `stack_batch`.
stack_batch	Stack multiple tensors to form a batch and pad the images to the max shape use the right bottom padding mode in these images.
GenDataPreprocessor	Image pre-processor for generative models.

1.43.1 EditDataPreprocessor

class mmedit.models.data_preprocessors.**EditDataPreprocessor**(*mean: Sequence[Union[float, int]] = (0, 0, 0)*, *std: Sequence[Union[float, int]] = (255, 255, 255)*, *pad_size_divisor: int = 1*, *input_view=(- 1, 1, 1)*, *output_view=None*, *pad_args: dict = {}*)

Basic data pre-processor used for collating and copying data to the target device in mmediting.

`EditDataPreprocessor` performs data pre-processing according to the following steps:

- Collates the data sampled from dataloader.
- Copies data to the target device.
- Stacks the input tensor at the first dimension.

and post-processing of the output tensor of model.

TODO: Most editing methods have crop inputs to a same size, batched padding will be faster.

Parameters

- **mean** (*Sequence[float or int]*) – The pixel mean of R, G, B channels. Defaults to (0, 0, 0). If `mean` and `std` are not specified, ImgDataPreprocessor will normalize images to [0, 1].
- **std** (*Sequence[float or int]*) – The pixel standard deviation of R, G, B channels. (255, 255, 255). If `mean` and `std` are not specified, ImgDataPreprocessor will normalize images to [0, 1].
- **pad_size_divisor** (*int*) – The size of padded image should be divisible by `pad_size_divisor`. Defaults to 1.
- **input_view** (*Tuple | List*) – Tensor view of mean and std for input (without batch). Defaults to (-1, 1, 1) for (C, H, W).
- **output_view** (*Tuple | List | None*) – Tensor view of mean and std for output (without batch). If None, output_view=input_view. Defaults: None.
- **pad_args** (*dict*) – Args of F.pad. Default: dict().

destructor(*batch_tensor: torch.Tensor*)

Destructor of data processor. Destruct padding, normalization and dissolve batch.

Parameters **batch_tensor** (*Tensor*) – Batched output.

Returns Destructed output.

Return type Tensor

forward(*data: Sequence[dict]*, *training: bool = False*) → Tuple[torch.Tensor, Optional[list]]

Pre-process the data into the model input format.

After the data pre-processing of `collate_data()`, `forward` will stack the input tensor list to a batch tensor at the first dimension.

Parameters

- **data** (*Sequence[dict]*) – data sampled from dataloader.
- **training** (*bool*) – Whether to enable training time augmentation. Default: False.

Returns Data in the same format as the model input.

Return type Tuple[torch.Tensor, Optional[list]]

1.43.2 MattorPreprocessor

class mmedit.models.data_preprocessors.**MattorPreprocessor**(*mean: float = [123.675, 116.28, 103.53]*, *std: float = [58.395, 57.12, 57.375]*, *bgr_to_rgb: bool = True*, *proc_inputs: str = 'normalize'*, *proc_trimap: str = 'rescale_to_zero_one'*, *proc_gt: str = 'rescale_to_zero_one'*)

DataPreprocessor for matting models.

See base class `BaseDataPreprocessor` for detailed information.

Workflow as follow :

- Collate and move data to the target device.
- Convert inputs from bgr to rgb if the shape of input is (3, H, W).
- Normalize image with defined std and mean.
- Stack inputs to batch_inputs.

Parameters

- **mean** (*Sequence[float or int]*) – The pixel mean of R, G, B channels. Defaults to [123.675, 116.28, 103.53].
- **std** (*Sequence[float or int]*) – The pixel standard deviation of R, G, B channels. [58.395, 57.12, 57.375].
- **bgr_to_rgb** (*bool*) – whether to convert image from BGR to RGB. Defaults to True.
- **proc_inputs** (*str*) – Methods to process inputs. Default: 'normalize'. Available options are `normalize`.
- **proc_trimap** (*str*) – Methods to process gt tensors. Default: 'rescale_to_zero_one'. Available options are `rescale_to_zero_one` and `as-is`.
- **proc_gt** (*str*) – Methods to process gt tensors. Default: 'rescale_to_zero_one'. Available options are `rescale_to_zero_one` and `ignore`.

collate_data(*data: Sequence[dict]*) → Tuple[list, list, list]

Collating and moving data to the target device.

See base class `BaseDataPreprocessor` for detailed information.

forward(*data: Sequence[dict]*, *training: bool = False*) → Tuple[torch.Tensor, list]

Pre-process input images, trimaps, ground-truth as configured.

Parameters

- **data** (*Sequence[dict]*) – data sampled from dataloader.
- **training** (*bool*) – Whether to enable training time augmentation. Default: False.

Returns Batched inputs and list of data samples.

Return type Tuple[torch.Tensor, list]

1.43.3 split_batch

class mmedit.models.data_preprocessors.**split_batch**(*batch_tensor: torch.Tensor*, *padded_sizes: torch.Tensor*)

reverse operation of `stack_batch`.

Parameters

- **batch_tensor** (*Tensor*) – The 4D-tensor or 5D-tensor. Tensor.dim == tensor_list[0].dim + 1
- **padded_sizes** (*Tensor*) – The padded sizes of each tensor.

Returns A list of tensors with the same dim.

Return type tensor_list (List[Tensor])

1.43.4 stack_batch

class mmedit.models.data_preprocessors.**stack_batch**(*tensor_list: List[torch.Tensor]*, *pad_size_divisor: int = 1*, *pad_args: dict = {}*)

Stack multiple tensors to form a batch and pad the images to the max shape use the right bottom padding mode in these images.

If `pad_size_divisor > 0`, add padding to ensure the shape of each dim is divisible by `pad_size_divisor`.

Parameters

- **tensor_list** (*List[Tensor]*) – A list of tensors with the same dim.
- **pad_size_divisor** (*int*) – If `pad_size_divisor > 0`, add padding to ensure the shape of each dim is divisible by `pad_size_divisor`. This depends on the model, and many models need to be divisible by 32. Defaults to 1
- **pad_args** (*dict*) – The padding args.

Returns The 4D-tensor or 5D-tensor. Tensor.dim == tensor_list[0].dim + 1 padded_sizes (Tensor): The padded sizes of each tensor.

Return type batch_tensor (Tensor)

1.43.5 GenDataPreprocessor

class mmedit.models.data_preprocessors.**GenDataPreprocessor**(*mean: Sequence[Union[float, int]] = (127.5, 127.5, 127.5)*, *std: Sequence[Union[float, int]] = (127.5, 127.5, 127.5)*, *pad_size_divisor: int = 1*, *pad_value: Union[float, int] = 0*, *bgr_to_rgb: bool = False*, *rgb_to_bgr: bool = False*, *non_image_keys: Optional[Tuple[str, List[str]]] = None*, *non_concentate_keys: Optional[Tuple[str, List[str]]] = None*)

Image pre-processor for generative models. This class provide normalization and bgr to rgb conversion for image tensor inputs. The input of this classes should be dict which keys are *inputs* and *data_samples*.

Besides to process tensor *inputs*, this class support dict as *inputs*. - If the value is *Tensor* and the corresponding key is not contained in `_NON_IMAGE_KEYS`, it will be processed as image tensor. - If the value is *Tensor* and the corresponding key belongs to `_NON_IMAGE_KEYS`, it will not remains unchanged. - If value is string or integer, it will not remains unchanged.

Parameters

- **mean** (*Sequence[float or int], optional*) – The pixel mean of image channels. If `bgr_to_rgb=True` it means the mean value of R, G, B channels. If it is not specified, images will not be normalized. Defaults None.
- **std** (*Sequence[float or int], optional*) – The pixel standard deviation of image channels. If `bgr_to_rgb=True` it means the standard deviation of R, G, B channels. If it is not specified, images will not be normalized. Defaults None.
- **pad_size_divisor** (*int*) – The size of padded image should be divisible by `pad_size_divisor`. Defaults to 1.
- **pad_value** (*float or int*) – The padded pixel value. Defaults to 0.

- **bgr_to_rgb** (*bool*) – whether to convert image from BGR to RGB. Defaults to False.
- **rgb_to_bgr** (*bool*) – whether to convert image from RGB to RGB. Defaults to False.

cast_data(*data: Union[tuple, dict, mmengine.structures.base_data_element.BaseDataElement, torch.Tensor, list]*) → Union[tuple, dict, mmengine.structures.base_data_element.BaseDataElement, torch.Tensor, list]

Copying data to the target device.

Parameters **data** (*dict*) – Data returned by `DataLoader`.

Returns Inputs and data sample at target device.

Return type CollatedResult

forward(*data: dict*, *training: bool = False*) → dict

Performs normalizationpadding and bgr2rgb conversion based on `BaseDataPreprocessor`.

Parameters

- **data** (*dict*) – Input data to process.
- **training** (*bool*) – Whether to enable training time augmentation. This is ignored for *GenDataPreprocessor*. Defaults to False.

Returns Data in the same format as the model input.

Return type dict

process_dict_inputs(*batch_inputs: dict*) → dict

Preprocess dict type inputs.

Parameters **batch_inputs** (*dict*) – Input dict.

Returns Preprocessed dict.

Return type dict

1.44 mmedit.models.losses

1.45 mmedit.models.utils

default_init_weights	Initialize network weights.
make_layer	Make layers by stacking the same blocks.
flow_warp	Warp an image or a feature map with optical flow.
generation_init_weights	Default initialization of network weights for image generation.
set_requires_grad	Set requires_grad for all the networks.
extract_bbox_patch	Extract patch from a given bbox.
extract_around_bbox	Extract patches around the given bbox.
get_unknown_tensor	Get 1-channel unknown area tensor from the 3 or 1-channel trimap tensor.
noise_sample_fn	Sample noise with respect to the given *num_batches*, *noise_size* and *device*.
label_sample_fn	Sample random label with respect to *num_batches*, *num_classes* and *device*.
get_valid_num_batches	Try get the valid batch size from inputs.
get_valid_noise_size	Get the value of *noise_size* from input, *generator* and check the consistency of these values.
get_module_device	Get the device of a module.

1.45.1 mmedit.models.utils.default_init_weights

mmedit.models.utils.**default_init_weights**(*module*, *scale=1*)

Initialize network weights.

Parameters

- **modules** (*nn.Module*) – Modules to be initialized.
- **scale** (*float*) – Scale initialized weights, especially for residual blocks. Default: 1.

1.45.2 mmedit.models.utils.make_layer

mmedit.models.utils.**make_layer**(*block*, *num_blocks*, ***kwarg*)

Make layers by stacking the same blocks.

Parameters

- **block** (*nn.module*) – nn.module class for basic block.
- **num_blocks** (*int*) – number of blocks.

Returns Stacked blocks in nn.Sequential.

Return type nn.Sequential

1.45.3 mmedit.models.utils.flow_warp

mmedit.models.utils.**flow_warp**(*x*, *flow*, *interpolation='bilinear'*, *padding_mode='zeros'*, *align_corners=True*)

Warp an image or a feature map with optical flow.

Parameters

- **x** (*Tensor*) – Tensor with size (n, c, h, w).
- **flow** (*Tensor*) – Tensor with size (n, h, w, 2). The last dimension is a two-channel, denoting the width and height relative offsets. Note that the values are not normalized to [-1, 1].
- **interpolation** (*str*) – Interpolation mode: 'nearest' or 'bilinear'. Default: 'bilinear'.
- **padding_mode** (*str*) – Padding mode: 'zeros' or 'border' or 'reflection'. Default: 'zeros'.
- **align_corners** (*bool*) – Whether align corners. Default: True.

Returns Warped image or feature map.

Return type Tensor

1.45.4 mmedit.models.utils.generation_init_weights

mmedit.models.utils.**generation_init_weights**(*module*, *init_type='normal'*, *init_gain=0.02*)

Default initialization of network weights for image generation.

By default, we use normal init, but xavier and kaiming might work better for some applications.

Parameters

- **module** (*nn.Module*) – Module to be initialized.
- **init_type** (*str*) – The name of an initialization method: normal | xavier | kaiming | orthogonal. Default: 'normal'.
- **init_gain** (*float*) – Scaling factor for normal, xavier and orthogonal. Default: 0.02.

1.45.5 mmedit.models.utils.set_requires_grad

mmedit.models.utils.**set_requires_grad**(*nets*, *requires_grad=False*)

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

1.45.6 mmedit.models.utils.extract_bbox_patch

mmedit.models.utils.**extract_bbox_patch**(*bbox*, *img*, *channel_first=True*)

Extract patch from a given bbox.

Parameters

- **bbox** (*torch.Tensor | numpy.array*) – Bbox with (top, left, h, w). If *img* has batch dimension, the *bbox* must be stacked at first dimension. The shape should be (4,) or (n, 4).
- **img** (*torch.Tensor | numpy.array*) – Image data to be extracted. If organized in batch dimension, the batch dimension must be the first order like (n, h, w, c) or (n, c, h, w).
- **channel_first** (*bool*) – If True, the channel dimension of img is before height and width, e.g. (c, h, w). Otherwise, the img shape (samples in the batch) is like (h, w, c). Default: True.

Returns Extracted patches. The dimension of the output should be the same as *img*.

Return type (torch.Tensor | numpy.array)

1.45.7 mmedit.models.utils.extract_around_bbox

mmedit.models.utils.**extract_around_bbox**(*img*, *bbox*, *target_size*, *channel_first=True*)

Extract patches around the given bbox.

Parameters

- **img** (*torch.Tensor | numpy.array*) – Image data to be extracted. If organized in batch dimension, the batch dimension must be the first order like (n, h, w, c) or (n, c, h, w).
- **bbox** (*np.ndarray | torch.Tensor*) – Bboxes to be modified. Bbox can be in batch or not.
- **target_size** (*List(int)*) – Target size of final bbox.
- **channel_first** (*bool*) – If True, the channel dimension of img is before height and width, e.g. (c, h, w). Otherwise, the img shape (samples in the batch) is like (h, w, c). Default: True.

Returns Extracted patches. The dimension of the output should be the same as *img*.

Return type (torch.Tensor | np.ndarray)

1.45.8 mmedit.models.utils.get_unknown_tensor

mmedit.models.utils.**get_unknown_tensor**(*trimap*, *unknown_value=0.5019607843137255*)

Get 1-channel unknown area tensor from the 3 or 1-channel trimap tensor.

Parameters

- **trimap** (*Tensor*) – Tensor with shape (N, 3, H, W) or (N, 1, H, W).
- **unknown_value** (*float*) – Scalar value indicating unknown region in trimap. If trimap is pre-processed using *'rescale_to_zero_one'*, then 0 for bg, 128/255 for unknown, 1 for fg, and unknown_value should set to 128 / 255. If trimap is pre-processed by `FormatTrimap(to_onehot=False)()`, then 0 for bg, 1 for unknown, 2 for fg and unknown_value should set to 1. If trimap is pre-processed by

`FormatTrimap(to_onehot=True)()`, then trimap is 3-channeled, and this value is not used.

Returns Unknown area mask of shape (N, 1, H, W).

Return type Tensor

1.45.9 mmedit.models.utils.noise_sample_fn

mmedit.models.utils.**noise_sample_fn**(*noise: Optional[Union[torch.Tensor, Callable]] = None*, ***, *num_batches: int = 1*, *noise_size: Optional[Union[int, Sequence[int]]] = None*, *device: Optional[str] = None*) → torch.Tensor

Sample noise with respect to the given *num_batches*, *noise_size* and *device*.

Parameters

- **noise** (*torch.Tensor | callable | None*) – You can directly give a batch of noise through a `torch.Tensor` or offer a callable function to sample a batch of noise data. Otherwise, the `None` indicates to use the default noise sampler. Defaults to None.
- **num_batches** (*int, optional*) – The number of batch size. Defaults to 1.
- **noise_size** (*Union[int, Sequence[int], None], optional*) – The size of random noise. Defaults to None.
- **device** (*Optional[str], optional*) – The target device of the random noise. Defaults to None.

Returns Sampled random noise.

Return type Tensor

1.45.10 mmedit.models.utils.label_sample_fn

mmedit.models.utils.**label_sample_fn**(*label: Optional[Union[torch.Tensor, Callable, List[int]]] = None*, ***, *num_batches: int = 1*, *num_classes: Optional[int] = None*, *device: Optional[str] = None*) → Optional[torch.Tensor]

Sample random label with respect to *num_batches*, *num_classes* and *device*.

Parameters

- **label** (*Union[Tensor, Callable, List[int], None], optional*) – You can directly give a batch of label through a `torch.Tensor` or offer a callable function to sample a batch of label data. Otherwise, the `None` indicates to use the default label sampler. Defaults to None.
- **num_batches** (*int, optional*) – The number of batch size. Defaults to 1.
- **num_classes** (*Optional[int], optional*) – The number of classes. Defaults to None.
- **device** (*Optional[str], optional*) – The target device of the label. Defaults to None.

Returns Sampled random label.

Return type Union[Tensor, None]

1.45.11 mmedit.models.utils.get_valid_num_batches

mmedit.models.utils.**get_valid_num_batches**(*batch_inputs: Tuple[Dict[str, Union[torch.Tensor, str, int]], torch.Tensor]*) → int

Try get the valid batch size from inputs.

- If some values in *batch_inputs* are *Tensor* and 'num_batches' is in *batch_inputs*, we check whether the value of 'num_batches' and the the length of first dimension of all tensors are same. If the values are not same, *AssertionError* will be raised. If all values are the same, return the value.
- If no values in *batch_inputs* is *Tensor*, 'num_batches' must be contained in *batch_inputs*. And this value will be returned.
- If some values are *Tensor* and 'num_batches' is not contained in *batch_inputs*, we check whether all tensor have the same length on the first dimension. If the length are not same, *AssertionError* will be raised. If all length are the same, return the length as batch size.
- If batch_inputs is a *Tensor*, directly return the length of the first dimension as batch size.

Parameters **batch_inputs** (*ForwardInputs*) – Inputs passed to `forward()`.

Returns The batch size of samples to generate.

Return type int

1.45.12 mmedit.models.utils.get_valid_noise_size

mmedit.models.utils.**get_valid_noise_size**(*noise_size: Optional[int]*, *generator: Union[Dict, torch.nn.modules.module.Module]*) → Optional[int]

Get the value of *noise_size* from input, *generator* and check the consistency of these values. If no conflict is found, return that value.

Parameters

- **noise_size** (*Optional[int]*) – *noise_size* passed to *BaseGAN_refactor*'s initialize function.
- **generator** (*ModelType*) – The config or the model of generator.

Returns The noise size feed to generator.

Return type int | None

1.45.13 mmedit.models.utils.get_module_device

mmedit.models.utils.**get_module_device**(*module*)

Get the device of a module.

Parameters **module** (*nn.Module*) – A module contains the parameters.

Returns The device of the module.

Return type torch.device

1.46 mmedit.models.editors

1.47 mmedit.structures

EditDataSample	A data structure interface of MMEditing.
PixelData	Data structure for pixel-level annnotations or predictions.

1.47.1 EditDataSample

class mmedit.structures.**EditDataSample**(*, *metainfo: Optional[dict] = None*, ***kwargs*)

A data structure interface of MMEditing. They are used as interfaces between different components.

The attributes in `EditDataSample` are divided into several parts:

- `gt_img`: Ground truth image(s).
- `pred_img`: Image(s) of model predictions.
- `ref_img`: Reference image(s).
- `mask`: Mask in Inpainting.
- `trimap`: Trimap in Matting.
- `gt_alpha`: Ground truth alpha image in Matting.
- `pred_alpha`: Predicted alpha image in Matting.
- `gt_fg`: Ground truth foreground image in Matting.
- `pred_fg`: Predicted foreground image in Matting.
- `gt_bg`: Ground truth background image in Matting.
- `pred_bg`: Predicted background image in Matting.
- `gt_merged`: Ground truth merged image in Matting.

Examples:

```
>>> import torch
>>> import numpy as np
>>> from mmedit.structures import EditDataSample, PixelData
>>> data_sample = EditDataSample()
>>> img_meta = dict(img_shape=(800, 1196, 3))
>>> img = torch.rand((3, 800, 1196))
>>> gt_img = PixelData(data=img, metainfo=img_meta)
>>> data_sample.gt_img = gt_img
>>> assert 'img_shape' in data_sample.gt_img.metainfo_keys()
<EditDataSample(

    META INFORMATION
```

(continues on next page)

(continued from previous page)

```
    DATA FIELDS
    _gt_img: <PixelData(

            META INFORMATION
            img_shape: (800, 1196, 3)

            DATA FIELDS
            data: tensor([[[0.8069, 0.4279,  ..., 0.6603, 0.0292],

                        ...,

                        [0.8139, 0.0908,  ..., 0.4964, 0.9672]]])
        ) at 0x1f6ae000af0>
    gt_img: <PixelData(

            META INFORMATION
            img_shape: (800, 1196, 3)

            DATA FIELDS
            data: tensor([[[0.8069, 0.4279,  ..., 0.6603, 0.0292],

                        ...,

                        [0.8139, 0.0908,  ..., 0.4964, 0.9672]]])
        ) at 0x1f6ae000af0>
) at 0x1f6a5a99a00>
```

property ema: *mmedit.structures.edit_data_sample.EditDataSample*

This is the function to fetch ema results.

Returns Results of the ema model.

Return type *EditDataSample*

property fake_img: **Union[***mmedit.structures.pixel_data.PixelData***, torch.Tensor]**

This is the function to fetch fake_img.

Returns The fake img.

Return type Union[*PixelData*, Tensor]

property gt_alpha: *mmedit.structures.pixel_data.PixelData*

This is the function to fetch gt_alpha.

Returns data element

Return type *PixelData*

property gt_bg: *mmedit.structures.pixel_data.PixelData*

This is the function to fetch gt_bg.

Returns data element

Return type *PixelData*

property gt_fg: *mmedit.structures.pixel_data.PixelData*

This is the function to fetch gt_fg.

Returns data element

Return type *PixelData*

property gt_heatmap: *mmedit.structures.pixel_data.PixelData*

This is the function to fetch gt_heatmap.

Returns data element

Return type *PixelData*

property gt_img: *mmedit.structures.pixel_data.PixelData*

This is the function to fetch gt_img in PixelData.

Returns data element

Return type *PixelData*

property gt_label

This the function to fetch gt label.

Returns gt label.

Return type LabelData

property gt_merged: *mmedit.structures.pixel_data.PixelData*

This is the function to fetch gt_merged in PixelData.

Returns _description_

Return type *PixelData*

property gt_samples: *mmedit.structures.edit_data_sample.EditDataSample*

This is the function to fetch gt_samples.

Returns gt samples.

Return type *EditDataSample*

property gt_unsharp: *mmedit.structures.pixel_data.PixelData*

This is the function to fetch gt_unsharp in PixelData.

Returns data element

Return type *PixelData*

property img_lq: *mmedit.structures.pixel_data.PixelData*

This is the function to fetch img_lq in PixelData.

Returns data element

Return type *PixelData*

property mask: *mmedit.structures.pixel_data.PixelData*

This is the function to fetch mask.

Returns data element

Return type *PixelData*

property noise: torch.Tensor

This is the function to fetch noise.

Returns noise.

Return type torch.Tensor

property orig: *mmedit.structures.edit_data_sample.EditDataSample*

This is the function to fetch original results.

Returns Results of the ema model.

Return type *EditDataSample*

property pred_alpha: *mmedit.structures.pixel_data.PixelData*

This is the function to fetch pred_alpha.

Returns data element

Return type *PixelData*

property pred_bg: *mmedit.structures.pixel_data.PixelData*

This is the function to fetch pred_bg in PixelData.

Returns data element

Return type *PixelData*

property pred_fg: *mmedit.structures.pixel_data.PixelData*

This is the function to fetch pred_fg.

Returns _description_

Return type *PixelData*

property pred_heatmap: *mmedit.structures.pixel_data.PixelData*

This is the function to fetch pred_heatmap.

Returns data element

Return type *PixelData*

property pred_img: *mmedit.structures.pixel_data.PixelData*

This is the function to fetch pred_img in PixelData.

Returns data element

Return type *PixelData*

property ref_img: *mmedit.structures.pixel_data.PixelData*

This is the function to fetch ref_img.

Returns data element

Return type *PixelData*

property ref_lq: *mmedit.structures.pixel_data.PixelData*

This is the function to fetch ref_lq.

Returns data element

Return type *PixelData*

property sample_model: **str**

This is the function to fetch sample model.

Returns Mode of Sample model.

Return type str

set_gt_label(*value: Union[numpy.ndarray, torch.Tensor, Sequence[numbers.Number], numbers.Number]*) → *mmedit.structures.edit_data_sample.EditDataSample*

Set label of `gt_label`.

property trimap: *mmedit.structures.pixel_data.PixelData*

This is the function to fetch trimap.

Returns data element

Return type *PixelData*

1.47.2 PixelData

class mmedit.structures.**PixelData**(**, metainfo: Optional[dict] = None, **kwargs*)

Data structure for pixel-level annnotations or predictions.

Different from parent class: Support value.ndim == 4 for frames.

All data items in `data_fields` of `PixelData` meet the following requirements:

- They all have 3 dimensions in orders of channel, height, and width.
- They should have the same height and width.

Examples

```
>>> metainfo = dict(
...     img_id=random.randint(0, 100),
...     img_shape=(random.randint(400, 600), random.randint(400, 600)))
>>> image = np.random.randint(0, 255, (4, 20, 40))
>>> featmap = torch.randint(0, 255, (10, 20, 40))
>>> pixel_data = PixelData(metainfo=metainfo,
...                        image=image,
...                        featmap=featmap)
>>> print(pixel_data)
>>> (20, 40)
```

```
>>> # slice
>>> slice_data = pixel_data[10:20, 20:40]
>>> assert slice_data.shape == (10, 10)
>>> slice_data = pixel_data[10, 20]
>>> assert slice_data.shape == (1, 1)
```

1.48 mmedit.visualization

ConcatImageVisualizer	Visualize multiple images by concatenation.
GenVisualizer	MMEditing Visualizer.
GenVisBackend	Generation visualization backend class.
PaviGenVisBackend	Visualization backend for Pavi.
TensorboardGenVisBackend	
WandbGenVisBackend	Wandb visualization backend for MMEditing.

1.48.1 ConcatImageVisualizer

class mmedit.visualization.**ConcatImageVisualizer**(*fn_key: str*, *img_keys: Sequence[str]*, *pixel_range={}*, *bgr2rgb=False*, *name: str = 'visualizer'*, **args*, ***kwargs*)

Visualize multiple images by concatenation.

This visualizer will horizontally concatenate images belongs to different keys and vertically concatenate images belongs to different frames to visualize.

Image to be visualized can be:

- torch.Tensor or np.array
- Image sequences of shape (T, C, H, W)
- Multi-channel image of shape (1/3, H, W)
- Single-channel image of shape (C, H, W)

Parameters

- **fn_key** (*str*) – key used to determine file name for saving image. Usually it is the path of some input image. If the value is *dir/basename.ext*, the name used for saving will be basename.
- **img_keys** (*str*) – keys, values of which are images to visualize.
- **pixel_range** (*dict*) – min and max pixel value used to denormalize images, note that only float array or tensor will be denormalized, uint8 arrays are assumed to be unnormalized.
- **bgr2rgb** (*bool*) – whether to convert the image from BGR to RGB.
- **name** (*str*) – name of visualizer. Default: 'visualizer'.
- ****kwargs** (**args and*) – Other arguments are passed to *Visualizer*. # noqa

add_datasample(*data_sample:* mmedit.structures.edit_data_sample.EditDataSample, *step=0*) → None

Concatenate image and draw.

Parameters

- **input** (*torch.Tensor*) – Single input tensor from data_batch.
- **data_sample** (EditDataSample) – Single data_sample from data_batch.
- **output** (EditDataSample) – Single prediction output by model.
- **step** (*int*) – Global step value to record. Default: 0.

1.48.2 GenVisualizer

class mmedit.visualization.**GenVisualizer**(*name='visualizer'*, *vis_backends: Optional[List[Dict]] = None*, *save_dir: Optional[str] = None*)

MMEditing Visualizer.

Parameters

- **name** (*str*) – Name of the instance. Defaults to 'visualizer'.
- **vis_backends** (*list, optional*) – Visual backend config list. Defaults to None.

- **save_dir** (*str, optional*) – Save file dir for all storage backends. If it is None, the backend storage will not save any data.

Examples:

```
>>> # Draw image
>>> vis = GenVisualizer()
>>> vis.add_datasample(
>>>     'random_noise',
>>>     gen_samples=torch.rand(2, 3, 10, 10),
>>>     gt_samples=dict(imgs=torch.randn(2, 3, 10, 10)),
>>>     gt_keys='imgs',
>>>     vis_mode='image',
>>>     n_rows=2,
>>>     step=10)
```

add_datasample(*name: str*, *, *gen_samples: Sequence[mmedit.structures.edit_data_sample.EditDataSample]*, *target_keys: Optional[Tuple[str, List[str]]] = None*, *vis_mode: Optional[str] = None*, *n_row: Optional[int] = 1*, *color_order: str = 'bgr'*, *target_mean: Sequence[Union[float, int]] = 127.5*, *target_std: Sequence[Union[float, int]] = 127.5*, *show: bool = False*, *wait_time: int = 0*, *step: int = 0*, ***kwargs*) → None

Draw datasample and save to all backends.

If GT and prediction are plotted at the same time, they are displayed in a stitched image where the left image is the ground truth and the right image is the prediction.

If `show` is True, all storage backends are ignored, and the images will be displayed in a local window.

Parameters

- **name** (*str*) – The image identifier.
- **gen_samples** (*List[EditDataSample]*) – Data samples to visualize.
- **vis_mode** (*str, optional*) – Visualization mode. If not passed, will visualize results as image. Defaults to None.
- **n_rows** (*int, optional*) – Number of images in one row. Defaults to 1.
- **color_order** (*str*) – The color order of the passed images. Defaults to 'bgr'.
- **target_mean** (*Sequence[Union[float, int]]*) – The target mean of the visualization results. Defaults to 127.5.
- **target_std** (*Sequence[Union[float, int]]*) – The target std of the visualization resutts. Defaults to 127.5.
- **show** (*bool*) – Whether to display the drawn image. Default to False.
- **wait_time** (*float*) – The interval of show (s). Defaults to 0.
- **step** (*int*) – Global step value to record. Defaults to 0.

add_image(*name: str*, *image: numpy.ndarray*, *step: int = 0*, ***kwargs*) → None

Record the image. Support input kwargs.

Parameters

- **name** (*str*) – The image identifier.
- **image** (*np.ndarray, optional*) – The image to be saved. The format should be RGB. Default to None.

- **step** (*int*) – Global step value to record. Default to 0.

1.48.3 GenVisBackend

class mmedit.visualization.**GenVisBackend**(*save_dir: str*, *img_save_dir: str = 'vis_image'*, *config_save_file: str = 'config.py'*, *scalar_save_file: str = 'scalars.json'*, *ceph_path: Optional[str] = None*, *delete_local_image: bool = True*)

Generation visualization backend class. It can write image, config, scalars, etc. to the local hard disk and ceph path. You can get the drawing backend through the experiment property for custom drawing.

Examples

```
>>> from mmgen.visualization import GenVisBackend
>>> import numpy as np
>>> vis_backend = GenVisBackend(save_dir='temp_dir',
>>>                             ceph_path='s3://temp-bucket')
>>> img = np.random.randint(0, 256, size=(10, 10, 3))
>>> vis_backend.add_image('img', img)
>>> vis_backend.add_scalar('mAP', 0.6)
>>> vis_backend.add_scalars({'loss': [1, 2, 3], 'acc': 0.8})
>>> cfg = Config(dict(a=1, b=dict(b1=[0, 1])))
>>> vis_backend.add_config(cfg)
```

Parameters

- **save_dir** (*str*) – The root directory to save the files produced by the visualizer.
- **img_save_dir** (*str*) – The directory to save images. Default to 'vis_image'.
- **config_save_file** (*str*) – The file name to save config. Default to 'config.py'.
- **scalar_save_file** (*str*) – The file name to save scalar values. Default to 'scalars.json'.
- **ceph_path** (*Optional[str]*) – The remote path of Ceph cloud storage. Defaults to None.
- **delete_local** (*bool*) – Whether eelete local after uploading to ceph or not. If `ceph_path` is None, this will be ignored. Defaults to True.

add_config(*config: mmengine.config.config.Config*, ***kwargs*) → None

Record the config to disk.

Parameters **config** (*Config*) – The Config object

add_image(*name: str*, *image: numpy.array*, *step: int = 0*, ***kwargs*) → None

Record the image to disk.

Parameters

- **name** (*str*) – The image identifier.
- **image** (*np.ndarray*) – The image to be saved. The format should be RGB. Default to None.
- **step** (*int*) – Global step value to record. Default to 0.

add_scalar(*name: str*, *value: Union[int, float, torch.Tensor, numpy.ndarray]*, *step: int = 0*, ***kwargs*) → None

Record the scalar data to disk.

Parameters

- **name** (*str*) – The scalar identifier.
- **value** (*int, float, torch.Tensor, np.ndarray*) – Value to save.
- **step** (*int*) – Global step value to record. Default to 0.

add_scalars(*scalar_dict: dict*, *step: int = 0*, *file_path: Optional[str] = None*, ***kwargs*) → None

Record the scalars to disk.

The scalar dict will be written to the default and specified files if `file_path` is specified.

Parameters

- **scalar_dict** (*dict*) – Key-value pair storing the tag and corresponding values. The value must be dumped into json format.
- **step** (*int*) – Global step value to record. Default to 0.
- **file_path** (*str, optional*) – The scalar's data will be saved to the `file_path` file at the same time if the `file_path` parameter is specified. Default to None.

property experiment: *mmedit.visualization.vis_backend.GenVisBackend*

Return the experiment object associated with this visualization backend.

1.48.4 PaviGenVisBackend

class mmedit.visualization.**PaviGenVisBackend**(*save_dir: str*, *exp_name: Optional[str] = None*, *labels: Optional[str] = None*, *project: Optional[str] = None*, *model: Optional[str] = None*, *description: Optional[str] = None*)

Visualization backend for Pavi.

add_image(*name: str*, *image: numpy.array*, *step: int = 0*, ***kwargs*) → None

Record the image to Pavi.

Parameters

- **name** (*str*) – The image identifier.
- **image** (*np.ndarray*) – The image to be saved. The format should be RGB. Default to None.
- **step** (*int*) – Global step value to record. Default to 0.

add_scalar(*name: str*, *value: Union[int, float, torch.Tensor, numpy.ndarray]*, *step: int = 0*, ***kwargs*) → None

Record the scalar data to Pavi.

Parameters

- **name** (*str*) – The scalar identifier.
- **value** (*int, float, torch.Tensor, np.ndarray*) – Value to save.
- **step** (*int*) – Global step value to record. Default to 0.

add_scalars(*scalar_dict: dict*, *step: int = 0*, *file_path: Optional[str] = None*, ***kwargs*) → None

Record the scalars to Pavi.

The scalar dict will be written to the default and specified files if `file_path` is specified.

Parameters

- **scalar_dict** (*dict*) – Key-value pair storing the tag and corresponding values. The value must be dumped into json format.
- **step** (*int*) – Global step value to record. Default to 0.
- **file_path** (*str, optional*) – The scalar's data will be saved to the `file_path` file at the same time if the `file_path` parameter is specified. Default to None.

property experiment: *mmedit.visualization.vis_backend.GenVisBackend*

Return the experiment object associated with this visualization backend.

1.48.5 TensorboardGenVisBackend

class mmedit.visualization.**TensorboardGenVisBackend**(*save_dir: str*)

add_image(*name: str*, *image: numpy.array*, *step: int = 0*, ***kwargs*)

Record the image to Tensorboard. Additional support upload gif files.

Parameters

- **name** (*str*) – The image identifier.
- **image** (*np.ndarray*) – The image to be saved. The format should be RGB.
- **step** (*int*) – Useless parameter. Wandb does not need this parameter. Default to 0.

1.48.6 WandbGenVisBackend

class mmedit.visualization.**WandbGenVisBackend**(*save_dir: str*, *init_kwargs: Optional[dict] = None*, *define_metric_cfg: Optional[dict] = None*, *commit: Optional[bool] = True*, *log_code_name: Optional[str] = None*, *watch_kwargs: Optional[dict] = None*)

Wandb visualization backend for MMEditing.

add_image(*name: str*, *image: numpy.array*, *step: int = 0*, ***kwargs*)

Record the image to wandb. Additional support upload gif files.

Parameters

- **name** (*str*) – The image identifier.
- **image** (*np.ndarray*) – The image to be saved. The format should be RGB.
- **step** (*int*) – Useless parameter. Wandb does not need this parameter. Default to 0.

1.49 mmedit.utils

modify_args	Modify args of argparse.ArgumentParser.
print_colored_log	Print colored log with default logger.
register_all_modules	Register all modules in mmedit into the registries.
download_from_url	Download object at the given URL to a local path.
get_sampler	Get a sampler to loop input data.
tensor2img	Convert torch Tensors into image numpy arrays.
random_choose_unknown	Randomly choose an unknown start (top-left) point for a given crop_size.
add_gaussian_noise	Add Gaussian Noise on the input image.
adjust_gamma	Performs Gamma Correction on the input image.
make_coord	Make coordinates at grid centers.
bbox2mask	Generate mask in np.ndarray from bbox.
brush_stroke_mask	Generate free-form mask.
get_irregular_mask	Get irregular mask with the constraints in mask ratio.
random_bbox	Generate a random bbox for the mask on a given image.
reorder_image	Reorder images to 'HWC' order.
to_numpy	Convert data into numpy arrays of dtype.

1.49.1 mmedit.utils.modify_args

mmedit.utils.**modify_args**()

Modify args of argparse.ArgumentParser.

1.49.2 mmedit.utils.print_colored_log

mmedit.utils.**print_colored_log**(*msg*, *level=20*, *color='magenta'*)

Print colored log with default logger.

Parameters

- **msg** (*str*) – Message to log.
- **level** (*int*) – The root logger level. Note that only the process of rank 0 is affected, while other processes will set the level to "Error" and be silent most of the time.Log level, default to 'info'.
- **color** (*str, optional*) – Color 'magenta'.

1.49.3 mmedit.utils.register_all_modules

mmedit.utils.**register_all_modules**(*init_default_scope: bool = True*) → None

Register all modules in mmedit into the registries.

Parameters **init_default_scope** (*bool*) – Whether initialize the mmedit default scope. When *init_default_scope=True*, the global default scope will be set to *mmedit*, and all registries will build modules from mmedit'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.

1.49.4 mmedit.utils.download_from_url

mmedit.utils.**download_from_url**(*url*, *dest_path=None*, *dest_dir='/home/docs/.cache/openmmlab/mmedit/'*, *hash_prefix=None*)

Download object at the given URL to a local path.

Parameters

- **url** (*str*) – URL of the object to download.
- **dest_path** (*str*) – Path where object will be saved.
- **dest_dir** (*str*) – The directory of the destination. Defaults to `'~/.cache/openmmlab/mmgen/'`.
- **hash_prefix** (*string, optional*) – If not None, the SHA256 downloaded file should start with *hash_prefix*. Default: None.

Returns path for the downloaded file.

Return type str

1.49.5 mmedit.utils.get_sampler

mmedit.utils.**get_sampler**(*sample_kwargs: dict*, *runner: Optional[mmengine.runner.runner.Runner]*)

Get a sampler to loop input data.

Parameters

- **sample_kwargs** (*dict*) – _description_
- **runner** (*Optional[Runner]*) – _description_

Returns _description_

Return type _type_

1.49.6 mmedit.utils.tensor2img

mmedit.utils.**tensor2img**(*tensor*, *out_type=<class 'numpy.uint8'>*, *min_max=(0, 1)*)

Convert torch Tensors into image numpy arrays.

After clamping to (min, max), image values will be normalized to [0, 1].

For different tensor shapes, this function will have different behaviors:

1. **4D mini-batch Tensor of shape (N x 3/1 x H x W):** Use *make_grid* to stitch images in the batch dimension, and then convert it to numpy array.
2. **3D Tensor of shape (3/1 x H x W) and 2D Tensor of shape (H x W):** Directly change to numpy array.

Note that the image channel in input tensors should be RGB order. This function will convert it to cv2 convention, i.e., (H x W x C) with BGR order.

Parameters

- **tensor** (*Tensor | list[Tensor]*) – Input tensors.
- **out_type** (*numpy type*) – Output types. If `np.uint8`, transform outputs to uint8 type with range [0, 255]; otherwise, float type with range [0, 1]. Default: `np.uint8`.

- **min_max** (*tuple*) – min and max values for clamp.

Returns 3D ndarray of shape (H x W x C) or 2D ndarray of shape (H x W).

Return type (Tensor | list[Tensor])

1.49.7 mmedit.utils.random_choose_unknown

mmedit.utils.**random_choose_unknown**(*unknown*, *crop_size*)

Randomly choose an unknown start (top-left) point for a given crop_size.

Parameters

- **unknown** (*np.ndarray*) – The binary unknown mask.
- **crop_size** (*tuple[int]*) – The given crop size.

Returns The top-left point of the chosen bbox.

Return type tuple[int]

1.49.8 mmedit.utils.add_gaussian_noise

mmedit.utils.**add_gaussian_noise**(*img: numpy.ndarray*, *mu*, *sigma*)

Add Gaussian Noise on the input image.

Parameters

- **img** (*np.ndarray*) – Input image.
- **mu** (*float*) – The mu value of the Gaussian function.
- **sigma** (*float*) – The sigma value of the Gaussian function.

Returns Gaussian noisy output image.

Return type noisy_img (np.ndarray)

1.49.9 mmedit.utils.adjust_gamma

mmedit.utils.**adjust_gamma**(*image*, *gamma=1*, *gain=1*)

Performs Gamma Correction on the input image.

This function is adopted from skimage: https://github.com/scikit-image/scikit-image/blob/7e4840bd9439d1dfb6beaf549998452c99f97fdd/skimage/exposure/ exposure.py#L439-L494

Also known as Power Law Transform. This function transforms the input image pixelwise according to the equation `O = I**gamma` after scaling each pixel to the range 0 to 1.

Parameters

- **image** (*np.ndarray*) – Input image.
- **gamma** (*float, optional*) – Non negative real number. Defaults to 1.
- **gain** (*float, optional*) – The constant multiplier. Defaults to 1.

Returns Gamma corrected output image.

Return type np.ndarray

1.49.10 mmedit.utils.make_coord

mmedit.utils.**make_coord**(*shape*, *ranges=None*, *flatten=True*)

Make coordinates at grid centers.

Parameters

- **shape** (*tuple*) – shape of image.
- **ranges** (*tuple*) – range of coordinate value. Default: None.
- **flatten** (*bool*) – flatten to (n, 2) or Not. Default: True.

Returns coordinates.

Return type coord (Tensor)

1.49.11 mmedit.utils.bbox2mask

mmedit.utils.**bbox2mask**(*img_shape*, *bbox*, *dtype='uint8'*)

Generate mask in np.ndarray from bbox.

The returned mask has the shape of (h, w, 1). '1' indicates the hole and '0' indicates the valid regions.

We prefer to use *uint8* as the data type of masks, which may be different from other codes in the community.

Parameters

- **img_shape** (*tuple[int]*) – The size of the image.
- **bbox** (*tuple[int]*) – Configuration tuple, (top, left, height, width)
- **np.dtype** (*str*) – Indicate the data type of returned masks. Default: 'uint8'

Returns Mask in the shape of (h, w, 1).

Return type mask (np.ndarray)

1.49.12 mmedit.utils.brush_stroke_mask

mmedit.utils.**brush_stroke_mask**(*img_shape*, *num_vertices=(4, 12)*, *mean_angle=1.2566370614359172*, *angle_range=0.41887902047863906*, *brush_width=(12, 40)*, *max_loops=4*, *dtype='uint8'*)

Generate free-form mask.

The method of generating free-form mask is in the following paper: Free-Form Image Inpainting with Gated Convolution.

When you set the config of this type of mask. You may note the usage of *np.random.randint* and the range of *np.random.randint* is [left, right).

We prefer to use *uint8* as the data type of masks, which may be different from other codes in the community.

TODO: Rewrite the implementation of this function.

Parameters

- **img_shape** (*tuple[int]*) – Size of the image.
- **num_vertices** (*int | tuple[int]*) – Min and max number of vertices. If only give an integer, we will fix the number of vertices. Default: (4, 12).

- **mean_angle** (*float*) – Mean value of the angle in each vertex. The angle is measured in radians. Default: 2 * math.pi / 5.
- **angle_range** (*float*) – Range of the random angle. Default: 2 * math.pi / 15.
- **brush_width** (*int | tuple[int]*) – (min_width, max_width). If only give an integer, we will fix the width of brush. Default: (12, 40).
- **max_loops** (*int*) – The max number of for loops of drawing strokes. Default: 4.
- **np.dtype** (*str*) – Indicate the data type of returned masks. Default: 'uint8'.

Returns Mask in the shape of (h, w, 1).

Return type mask (np.ndarray)

1.49.13 mmedit.utils.get_irregular_mask

mmedit.utils.**get_irregular_mask**(*img_shape*, *area_ratio_range=(0.15, 0.5)*, ***kwargs*)

Get irregular mask with the constraints in mask ratio.

Parameters

- **img_shape** (*tuple[int]*) – Size of the image.
- **area_ratio_range** (*tuple(float)*) – Contain the minimum and maximum area
- **Default** (*ratio.*) – (0.15, 0.5).

Returns Mask in the shape of (h, w, 1).

Return type mask (np.ndarray)

1.49.14 mmedit.utils.random_bbox

mmedit.utils.**random_bbox**(*img_shape*, *max_bbox_shape*, *max_bbox_delta=40*, *min_margin=20*)

Generate a random bbox for the mask on a given image.

In our implementation, the max value cannot be obtained since we use *np.random.randint*. And this may be different with other standard scripts in the community.

Parameters

- **img_shape** (*tuple[int]*) – The size of a image, in the form of (h, w).
- **max_bbox_shape** (*int | tuple[int]*) – Maximum shape of the mask box, in the form of (h, w). If it is an integer, the mask box will be square.
- **max_bbox_delta** (*int | tuple[int]*) – Maximum delta of the mask box, in the form of (delta_h, delta_w). If it is an integer, delta_h and delta_w will be the same. Mask shape will be randomly sampled from the range of *max_bbox_shape - max_bbox_delta* and *max_bbox_shape*. Default: (40, 40).
- **min_margin** (*int | tuple[int]*) – The minimum margin size from the edges of mask box to the image boarder, in the form of (margin_h, margin_w). If it is an integer, margin_h and margin_w will be the same. Default: (20, 20).

Returns The generated box, (top, left, h, w).

Return type tuple[int]

1.49.15 mmedit.utils.reorder_image

mmedit.utils.**reorder_image**(*img*, *input_order='HWC'*)

Reorder images to 'HWC' order.

If the input_order is (h, w), return (h, w, 1); If the input_order is (c, h, w), return (h, w, c); If the input_order is (h, w, c), return as it is.

Parameters

- **img** (*np.ndarray*) – Input image.
- **input_order** (*str*) – Whether the input order is 'HWC' or 'CHW'. If the input image shape is (h, w), input_order will not have effects. Default: 'HWC'.

Returns Reordered image.

Return type np.ndarray

1.49.16 mmedit.utils.to_numpy

mmedit.utils.**to_numpy**(*img*, *dtype=<class 'numpy.float64'>*)

Convert data into numpy arrays of dtype.

Parameters

- **img** (*Tensor | np.ndarray*) – Input data.
- **dtype** (*np.dtype*) – Set the data type of the output. Default: np.float64

Returns Converted numpy arrays data.

Return type img (np.ndarray)

1.50 Contributing to MMEditing

This section introduces following contents:

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

1.50.1 Workflow

1. fork and pull the latest MMEditing repository (MMEditing)
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.

1.50.2 Code style

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 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.

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

C++ and CUDA

We follow the Google C++ Style Guide.

1.51 Projects based on MMEditing

There are many projects built upon MMEditing. We list some of them as examples of how to extend MMEditing for your own projects. As the page might not be completed, please feel free to create a PR to update this page.

1.51.1 Research papers

There are also projects released with papers. Some of the papers are published in top-tier conferences (CVPR, ECCV, and NeurIPS). Methods already supported and maintained by MMEditing are not listed.

- Towards Interpretable Video Super-Resolution via Alternating Optimization, ECCV 2022 [paper][github]
- SepLUT:Separable Image-adaptive Lookup Tables for Real-time Image Enhancement, ECCV 2022 [paper][github]
- Investigating Tradeoffs in Real-World Video Super-Resolution(RealBasicVSR), CVPR 2022 [paper][github]
- BasicVSR++: Improving Video Super-Resolution with Enhanced Propagation and Alignment, CVPR 2022 [paper][github]
- Multi-Scale Memory-Based Video Deblurring, CVPR 2022 [paper][github]
- AdaInt:Learning Adaptive Intervals for 3D Lookup Tables on Real-time Image Enhancement, CVPR 2022 [paper][github]
- A New Dataset and Transformer for Stereoscopic Video Super-Resolution, CVPRW 2022 [paper][github]
- BasicVSR:The Search for Essential Components in Video Super-Resolution and Beyond, CVPR 2021 [paper][github]
- GLEAN:Generative Latent Bank for Large-Factor Image Super-Resolution, CVPR 2021 [paper][github]
- DAN:Unfolding the Alternating Optimization for Blind Super Resolution, NeurIPS 2020 [paper][github]
- Positional Encoding as Spatial Inductive Bias in GANs, CVPR 2021[paper][github]
- A Multi-Modality Ovarian Tumor Ultrasound Image Dataset for Unsupervised Cross-Domain Semantic Segmentation, arXiv 2022 [paper][github]
- Arbitrary-Scale Image Synthesis, CVPR 2022 [paper][github]

1.51.2 Open-source projects

Some open-source projects extend MMEditing for more functions and fields. They reveal the potential of what MMEditing can do. We list several of them as below.

- PowerVQE: Open framework for quality enhancement of compressed videos based on PyTorch and MMEditing.
- VR-Baseline: Video Restoration Toolbox.
- Manga-Colorization-with-CycleGAN: Colorizing Black&White Japanese Manga using Generative Adversarial Network.

1.52 Changelog

1.52.1 v1.0.0rc4 (05/12/2022)

Highlights

We are excited to announce the release of MMEditing 1.0.0rc4. This release supports 45+ models, 176+ configs and 175+ checkpoints in MMGeneration and MMEditing. We highlight the following new features

- Support High-level APIs.
- Support diffusion models.
- Support Text2Image Task.
- Support 3D-Aware Generation.

New Features & Improvements

- Refactor high-level APIs. (#1410)
- Support disco-diffusion text-2-image. (#1234, #1504)
- Support EG3D. (#1482, #1493, #1494, #1499)
- Support NAFNet model. (#1369)

Bug Fixes

- fix srgan train config. (#1441)
- fix cain config. (#1404)
- fix rdn and srcnn train configs. (#1392)
- Revise config and pretrain model loading in esrgan. (#1407)

Contributors A total of 14 developers contributed to this release. Thanks @plyfager, @LeoXing1996, @Z-Fran, @zengyh1900, @VongolaWu, @gaoyang07, @ChangjianZhao, @zxczrx123, @jackghosts, @liuwenran, @CCODING04, @RoseZhao929, @shaocongliu, @liangzelong.

New Contributors

- @gaoyang07 made their first contribution in https://github.com/open-mmlab/mmediting/pull/1372
- @ChangjianZhao made their first contribution in https://github.com/open-mmlab/mmediting/pull/1461
- @zxczrx123 made their first contribution in https://github.com/open-mmlab/mmediting/pull/1462
- @jackghosts made their first contribution in https://github.com/open-mmlab/mmediting/pull/1463
- @liuwenran made their first contribution in https://github.com/open-mmlab/mmediting/pull/1410
- @CCODING04 made their first contribution in https://github.com/open-mmlab/mmediting/pull/783
- @RoseZhao929 made their first contribution in https://github.com/open-mmlab/mmediting/pull/1474
- @shaocongliu made their first contribution in https://github.com/open-mmlab/mmediting/pull/1470
- @liangzelong made their first contribution in https://github.com/open-mmlab/mmediting/pull/1488

1.52.2 v1.0.0rc3 (03/11/2022)

Highlights

We are excited to announce the release of MMEditing 1.0.0rc3. This release supports 43+ models, 170+ configs and 169+ checkpoints in MMGeneration and MMEditing. We highlight the following new features

- convert `mmdet` and `clip` to optional requirements.

New Features & Improvements

- Support `try_import` for `mmdet`. (#1408)
- Support `try_import` for `flip`. (#1420)
- Complete requirements (#1419)
- Update `.gitignore`. ($1416)
- Set `real_feat` to cpu in `inception_utils`. (#1415)
- Modify README and configs of StyleGAN2 and PEGAN (#1418)
- Improve the rendering of Docs-API (#1373)

Bug Fixes

- Revise config and pretrain model loading in ESRGAN (#1407)
- Revise config of LSGAN (#1409)
- Revise config of CAIN (#1404)

Contributors

A total of 5 developers contributed to this release. @Z-Fran, @zengyh1900, @plyfager, @LeoXing1996, @ruoningYu.

1.52.3 v1.0.0rc2 (02/11/2022)

Highlights

We are excited to announce the release of MMEditing 1.0.0rc2. This release supports 43+ models, 170+ configs and 169+ checkpoints in MMGeneration and MMEditing. We highlight the following new features

- patch-based and slider-based image and video comparison viewer.
- image colorization.

We want to sincerely thank our community for continuously improving MMEditing.

New Features & Improvements

- Support qualitative comparison tools. (#1303)
- Support instance aware colorization. (#1370)
- Support multi-metrics with different sample-model. (#1171)
- Improve the implementation
  - refactoring evaluation metrics. (#1164)
  - Save gt images in PGGAN's `forward`. (#1332)
  - Improve type and change default number of `preprocess_div2k_dataset.py`. (#1380)
  - Support pixel value clip in visualizer. (#1365)

  - Support SinGAN Dataset and SinGAN demo. (#1363)
  - Avoid cast int and float in GenDataPreprocessor. (#1385)
- Improve the documentation
  - Update a menu switcher. (#1162)
  - Fix TTSR's README. (#1325)
  - Revise docs (change `PackGenInputs` and `GenDataSample`). (#1382)

Bug Fixes

- Fix PPL bug. (#1172)
- Fix RDN number of channels. (#1328)
- Fix types of exceptions in demos. (#1372)
- Fix realesrgan ema. (#1341)
- Improve the assertion to ensuer `GenerateFacialHeatmap` as `np.float32`. (#1310)
- Fix sampling behavior of `unpaired_dataset.py` and urls in cyclegan's README. (#1308)
- Fix vsr models in pytorch2onnx. (#1300)
- Fix incorrect settings in configs. (#1167,#1200,#1236,#1293,#1302,#1304,#1319,#1331,#1336,#1349,#1352,#1353,#1358,#1364,

New Contributors

- @gaoyang07 made their first contribution in https://github.com/open-mmlab/mmediting/pull/1372

Contributors

A total of 7 developers contributed to this release. Thanks @LeoXing1996, @Z-Fran, @zengyh1900, @plyfager, @ryanxingql, @ruoningYu, @gaoyang07.

1.52.4 v1.0.0rc1(23/9/2022)

MMEditing 1.0.0rc1 has merged MMGeneration 1.x.

- Support 42+ algorithms, 169+ configs and 168+ checkpoints.
- Support 26+ loss functions, 20+ metrics.
- Support tensorboard, wandb.
- Support unconditional GANs, conditional GANs, image2image translation and internal learning.

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

MMEditing 1.0.0rc0 is the first version of MMEditing 1.x, a part of the OpenMMLab 2.0 projects.

Built upon the new training engine, MMEditing 1.x unifies the interfaces of dataset, models, evaluation, and visualization.

And there are some BC-breaking changes. Please check the migration tutorial for more details.

1.52.6 v0.15.0 (01/06/2022)

Highlights

1. Support FLAVR
2. Support AOT-GAN
3. Support CAIN with ReduceLROnPlateau Scheduler

New Features

- Add configs for AOT-GAN (#681)
- Support Vimeo90k-triplet dataset (#810)
- Add default config for mm-assistant (#827)
- Support CPU demo (#848)
- Support `use_cache` and `backend` in LoadImageFromFileList (#857)
- Support VFIVimeo90K7FramesDataset (#858)
- Support ColorJitter for VFI (#859)
- Support ReduceLrUpdaterHook (#860)
- Support `after_val_epoch` in IterBaseRunner (#861)
- Support FLAVR Net (#866, #867, #897)
- Support MAE metric (#871)
- Use mdformat (#888)
- Support CAIN with ReduceLROnPlateau Scheduler (#906)

Bug Fixes

- Change - to _ for restoration_demo.py (#834)
- Remove recommonmark in requirements/docs.txt (#844)
- Move EDVR to VSR category in README.md (#849)
- Remove , in multi-line F-string in crop.py (#855)
- Modify double `lq_path` to `gt_path` in test_pipeline (#862)
- Fix unittest of TOF-VFI (#873)
- Fix wrong frames in VFI demo (#891)
- Fix logo & contrib guideline on README (#898)
- Normalizing trimap in indexnet_dimaug_mobv2_1x16_78k_comp1k.py (#901)

Improvements

- Add `--cfg-options` in train/test scripts (#826)
- Update MMCV_MAX to 1.6 (#829)
- Update TOFlow in README (#835)
- Recover beirf installation steps & merge optional requirements (#836)
- Use {MMEditing Contributors} in citation (#838)
- Add tutorial for customizing losses (#839)

- Add installation guide (wiki ver) in README (#845)
- Add a 'need help to traslate' note on Chinese documentation (#850)
- Add wechat QR code in README_zh-CN.md (#851)
- Support non-zero frame index for SRFolderVideoDataset & Fix Typos (#853)
- Create README.md for docker (#856)
- Optimize IO for flow_warp (#881)
- Move wiki/installation to docs (#883)
- Add `myst_heading_anchors` (#887)
- Use checkpoint link in inpainting demo (#892)

Contributors

@wangruohui @quincylin1 @nijkah @jayagami @ckkelvinchan @ryanxingql @NK-CS-ZZL @Yshuo-Li

1.52.7 v0.14.0 (01/04/2022)

Highlights

1. Support TOFlow in video frame interpolation

New Features

- Support AOT-GAN (#677)
- Use `--diff-seed` to set different torch seed on different rank (#781)
- Support streaming reading of frames in video interpolation demo (#790)
- Support dist_train without slurm (#791)
- Put LQ into CPU for restoration_video_demo (#792)
- Support gray normalization constant in EDSR (#793)
- Support TOFlow in video frame interpolation (#806, #811)
- Support seed in DistributedSampler and sync seed across ranks (#815)

Bug Fixes

- Update link in README files (#782, #786, #819, #820)
- Fix matting tutorial, and fix links to colab (#795)
- Invert `flip_ratio` in `RandomAffine` pipeline (#799)
- Update preprocess_div2k_dataset.py (#801)
- Update SR Colab Demo Installation Method and Set5 link (#807)
- Fix Y/GRB mistake in EDSR README (#812)
- Replace pytorch install command to conda in README(_zh-CN).md (#816)

Improvements

- Update CI (#650)
- Update requirements.txt (#725, #817)
- Add Tutorial of dataset (#758), pipeline (#779), model (#766)

- Update index and TOC tree (#767)
- Make update_model_index.py compatible on windows (#768)
- Update doc build system (#769)
- Update keyword and classifier for setuptools (#773)
- Renovate installation (#776, #800)
- Update BasicVSR++ and RealBasicVSR docs (#778)
- Update citation (#785, #787)
- Regroup docs (#788)
- Use full name of config as 'Name' in metafile (#798)
- Update figure and video demo in README (#802)
- Add `clamp(0, 1)` in test of video frame interpolation (#805)
- Use hyphen for command line args in demo & tools (#808), and keep underline for required arguments in python files (#822)
- Make dataset.pipeline a dedicated section in doc (#813)
- Update mmcv-full>=1.3.13 to support DCN on CPU (#823)

Contributors

@wangruohui @ckkelvinchan @Yshuo-Li @nijkah @wdmwhh @freepoet @quincylin1

1.52.8 v0.13.0 (01/03/2022)

Highlights

1. Support CAIN
2. Support EDVR-L
3. Support running in Windows

New Features

- Add test-time ensemble for images and videos and support ensemble in BasicVSR series (#585)
- Support AOT-GAN (work in progress) (#674, #675, #676)
- Support CAIN (#683, #691, #709, #713)
- Add basic interpolater (#687)
- Add BaseVFIDataset and VFIVimeo90KDataset (#695, #697)
- Add video interpolation demo (#688, #717)
- Support various scales in RRDBNet (#699)
- Support Ref-SR inference (#716)
- Support EDVR-L on REDS (#719)
- Support CPU training (#720)
- Support running in Windows (#732, #738)
- Support DCN on CPU (#735)

Bug Fixes

- Fix link address in docs (#703, #704)
- Fix ARG `MMCV` in Dockerfile (#708)
- Fix file permission of non-executable files (#718)
- Fix some deprecation warning related to numpy (#728)
- Delete `__init__` in `TestVFIDataset` (#731)
- Fix data type in docstring of several Datasets (#739)
- Fix math notation in docstring (#741)
- Fix missing folders in copyright commit hook (#754)
- Delete duplicate test in loading (#756)

Improvements

- Update Pillow from 6.2.2 to 8.4 in CI (#693)
- Add argument 'repeat' to SRREDSMultipleGTDataset (#672)
- Deprecate the support for "python setup.py test" (#701)
- Add setup multi-processing both in train and test (#707)
- Add OpenMMLab website and platform links (#710)
- Refact README files of all methods (#712)
- Replace string version comparison with `package.version.parse` (#723)
- Add docs of Ref-SR demo and video frame interpolation demo (#724)
- Add interpolation and refact README.md (#726)
- Update isort version in pre-commit hook (#727)
- Redesign CI for Linux (#734)
- Update install.md (#763)
- Reorganizing OpenMMLab projects in readme (#764)
- Add deprecation message for deploy tools (#765)

Contributors

@wangruohui @ckkelvinchan @Yshuo-Li @quincylin1 @Juggernaut93 @anse3832 @nijkah

1.52.9 v0.12.0 (31/12/2021)

Highlights

1. Support RealBasicVSR
2. Support Real-ESRGAN checkpoint

New Features

- Support video input and output in restoration demo (#622)
- Support RealBasicVSR (#632, #633, #647, #680)
- Support Real-ESRGAN checkpoint (#635)

- Support conversion to y-channel when loading images (643)
- Support random video compression during training (#646)
- Support crop sequence (#648)
- Support pixel_unshuffle (#684)

Bug Fixes

- Change 'target_size' for RandomResize from list to tuple (#617)
- Fix folder creation in preprocess_df2k_ost_dataset.py (#623)
- Change TDAN config path in README (#625)
- Change 'radius' to 'kernel_size' for UnsharpMasking in Real-ESRNet config (#626)
- Fix bug in MATLABLikeResize (#630)
- Fix 'flow_warp' comment (#655)
- Fix the error of Model Zoo and Datasets in docs (#664)
- Fix bug in 'random_degradations' (#673)
- Limit opencv-python version (#689)

Improvements

- Translate docs to Chinese (#576, #577, #578, #579, #581, #582, #584, #587, #588, #589, #590, #591, #592, #593, #594, #595, #596, #641, #647, #656, #665, #666)
- Add UNetDiscriminatorWithSpectralNorm (#605)
- Use PyTorch sphinx theme (#607, #608)
- Update mmcv (#609), mmflow (#621), mmfewshot (#634) and mmhuman3d (#649) in docs
- Convert minimum GCC version to 5.4 (#612)
- Add tiff in SRDataset IMG_EXTENSIONS (#614)
- Update metafile and update_model_index.py (#615)
- Update preprocess_df2k_ost_dataset.py (#624)
- Add Abstract to README (#628, #636)
- Align NIQE to MATLAB results (#631)
- Add official markdown lint hook (#639)
- Skip CI when some specific files were changed (#640)
- Update docs/conf.py (#644, #651)
- Try to create a symbolic link on windows (#645)
- Cancel previous runs that are not completed (#650)
- Update path of configs in demo.md and getting_started.md (#658, #659)
- Use mmcv root model registry (#660)
- Update README.md (#654, #663)
- Refactor the structure of documentation (#668)
- Add script to crop REDS images into sub-images for faster IO (#669)

- Capitalize the first letter of the task name in the metafile (#678)
- Update FixedCrop for cropping image sequence (#682)

1.52.10 v0.11.0 (03/11/2021)

Highlights

- GLEAN for blind face image restoration #530
- Real-ESRGAN model #546

New Features

- Exponential Moving Average Hook #542
- Support DF2K_OST dataset #566

Improvements

- Add MATLAB-like bicubic interpolation #507
- Support random degradations during training #504
- Support torchserve #568

1.52.11 v0.10.0 (12/08/2021).

Highlights

1. Support LIIF-RDN (CVPR'2021)
2. Support BasicVSR++ (NTIRE'2021)

New Features

- Support loading annotation from file for video SR datasets (#423)
- Support persistent worker (#426)
- Support LIIF-RDN (#428, #440)
- Support BasicVSR++ (#451, #467)
- Support mim (#455)

Bug Fixes

- Fix bug in stat.py (#420)
- Fix astype error in function tensor2img (#429)
- Fix device error caused by torch.new_tensor when pytorch >= 1.7 (#465)
- Fix _non_dist_train in .mmedit/apis/train.py (#473)
- Fix multi-node distributed test (#478)

Breaking Changes

- Refactor LIIF for pytorch2onnx (#425)

Improvements

- Update Chinese docs (#415, #416, #418, #421, #424, #431, #442)
- Add CI of pytorch 1.9.0 (#444)

- Refactor README.md of configs (#452)
- Avoid loading pretrained VGG in unittest (#466)
- Support specifying scales in preprocessing div2k dataset (#472)
- Support all formats in readthedocs (#479)
- Use version_info of mmcv (#480)
- Remove unnecessary codes in restoration_video_demo.py (#484)
- Change priority of DistEvalIterHook to 'LOW' (#489)
- Reset resource limit (#491)
- Update QQ QR code in README_CN.md (#494)
- Add `myst_parser` (#495)
- Add license header (#496)
- Fix typo of StyleGAN modules (#427)
- Fix typo in docs/demo.md (#453, #454)
- Fix typo in tools/data/super-resolution/reds/README.md (#469)

1.52.12 v0.9.0 (30/06/2021).

Highlights

1. Support DIC and DIC-GAN (CVPR'2020)
2. Support GLEAN Cat 8x (CVPR'2021)
3. Support TTSR-GAN (CVPR'2020)
4. Add colab tutorial for super-resolution

New Features

- Add DIC (#342, #345, #348, #350, #351, #357, #363, #365, #366)
- Add SRFolderMultipleGTDataset (#355)
- Add GLEAN Cat 8x (#367)
- Add SRFolderVideoDataset (#370)
- Add colab tutorial for super-resolution (#380)
- Add TTSR-GAN (#372, #381, #383, #398)
- Add DIC-GAN (#392, #393, #394)

Bug Fixes

- Fix bug in restoration_video_inference.py (#379)
- Fix Config of LIIF (#368)
- Change the path to pre-trained EDVR-M (#396)
- Fix normalization in restoration_video_inference (#406)
- Fix [brush_stroke_mask] error in unittest (#409)

Breaking Changes

- Change mmcv minimum version to v1.3 (#378)

Improvements

- Correct Typos in code (#371)
- Add Custom_hooks (#362)
- Refactor unittest folder structure (#386)
- Add documents and download link for Vid4 (#399)
- Update model zoo for documents (#400)
- Update metafile (407)

1.52.13 v0.8.0 (31/05/2021).

Highlights

1. Support GLEAN (CVPR'2021)
2. Support TTSR (CVPR'2020)
3. Support TDAN (CVPR'2020)

New Features

- Add GLEAN (#296, #332)
- Support PWD metafile (#298)
- Support CropLike in pipeline (#299)
- Add TTSR (#301, #304, #307, #311, #311, #312, #313, #314, #321, #326, #327)
- Add TDAN (#316, #334, #347)
- Add onnx2tensorrt (#317)
- Add tensorrt evaluation (#328)
- Add SRFacicalLandmarkDataset (#329)
- Add key point auxiliary model for DIC (#336, #341)
- Add demo for video super-resolution methods (#275)
- Add SR Folder Ref Dataset (#292)
- Support FLOPs calculation of video SR models (#309)

Bug Fixes

- Fix find_unused_parameters in PyTorch 1.8 for BasicVSR (#290)
- Fix error in publish_model.py for pt>=1.6 (#291)
- Fix PSNR when input is uint8 (#294)

Improvements

- Support backend in LoadImageFromFile (#293, #303)
- Update `metric_average_mode` of video SR dataset (#319)
- Add error message in restoration_demo.py (324)
- Minor correction in getting_started.md (#339)

- Update description for Vimeo90K (#349)
- Support start_index in GenerateSegmentIndices (#338)
- Support different filename templates in GenerateSegmentIndices (#325)
- Support resize by scale-factor (#295, #310)

1.52.14 v0.7.0 (30/04/2021).

Highlights

1. Support BasicVSR (CVPR'2021)
2. Support IconVSR (CVPR'2021)
3. Support RDN (CVPR'2018)
4. Add onnx evaluation tool

New Features

- Add RDN (#233, #260, #280)
- Add MultipleGT datasets (#238)
- Add BasicVSR and IconVSR (#245, #252, #253, #254, #264, #274, #258, #252, #264)
- Add onnx evaluation tool (#279)

Bug Fixes

- Fix onnx conversion of maxunpool2d (#243)
- Fix inpainting in `demo.md` (#248)
- Tiny fix of config file of EDSR (#251)
- Fix link in README (#256)
- Fix restoration_inference key missing bug (#270)
- Fix the usage of channel_order in `loading.py` (#271)
- Fix the command of inpainting (#278)
- Fix `preprocess_vimeo90k_dataset.py` args name (#281)

Improvements

- Support `empty_cache` option in `test.py` (#261)
- Update projects in README (#249, #276)
- Support Y-channel PSNR and SSIM (#250)
- Add zh-CN README (#262)
- Update pytorch2onnx doc (#265)
- Remove extra quotation in English readme (#268)
- Change tags to comment (#269)
- List `model zoo` in README (#284, #285, #286)

1.52.15 v0.6.0 (08/04/2021).

Highlights

1. Support Local Implicit Image Function (LIIF)
2. Support exporting DIM and GCA from Pytorch to ONNX

New Features

- Add readthedocs config files and fix docstring (#92)
- Add github action file (#94)
- Support exporting DIM and GCA from Pytorch to ONNX (#105)
- Support concatenating datasets (#106)
- Support `non_dist_train` validation (#110)
- Add matting colab tutorial (#111)
- Support niqe metric (#114)
- Support PoolDataLoader for parrots (#134)
- Support collect-env (#137, #143)
- Support pt1.6 cpu/gpu in CI (#138)
- Support fp16 (139, #144)
- Support publishing to pypi (#149)
- Add modelzoo statistics (#171, #182, #186)
- Add doc of datasets (194)
- Support extended foreground option. (#195, #199, #200, #210)
- Support nn.MaxUnpool2d (#196)
- Add some FBA components (#203, #209, #215, #220)
- Support random down sampling in pipeline (#222)
- Support SR folder GT Dataset (#223)
- Support Local Implicit Image Function (LIIF) (#224, #226, #227, #234, #239)

Bug Fixes

- Fix `_non_dist_train` in train api (#104)
- Fix setup and CI (#109)
- Fix redundant loop bug in Normalize (#121)
- Fix `get_hash` in `setup.py` (#124)
- Fix `tool/preprocess_reds_dataset.py` (#148)
- Fix slurm train tutorial in `getting_started.md` (#162)
- Fix pip install bug (#173)
- Fix bug in config file (#185)
- Fix broken links of datasets (#236)
- Fix broken links of model zoo (#242)

Breaking Changes

- Refactor data loader configs (#201)

Improvements

- Updata requirements.txt (#95, #100)
- Update teaser (#96)
- Updata README (#93, #97, #98, #152)
- Updata model_zoo (#101)
- Fix typos (#102, #188, #191, #197, #208)
- Adopt adjust_gamma from skimage and reduce dependencies (#112)
- remove `.gitlab-ci.yml` (#113)
- Update import of first party (#115)
- Remove citation and contact (#122)
- Update version file (#136)
- Update download url (#141)
- Update `setup.py` (#150)
- Update the highest version of supported mmcv (#153, #154)
- modify `Crop` to handle a sequence of video frames (#164)
- Add links to other mm projects (#179, #180)
- Add config type (#181)
- Refactor docs (#184)
- Add config link (#187)
- Update file structure (#192)
- Update config doc (#202)
- Update `slurm_train.md` script (#204)
- Improve code style (#206, #207)
- Use `file_client` in CompositeFg (#212)
- Replace `random` with `numpy.random` (#213)
- Refactor `loader_cfg` (#214)

1.52.16 v0.5.0 (09/07/2020).

Note that **MMSR** has been merged into this repo, as a part of MMEditing. With elaborate designs of the new framework and careful implementations, hope MMEditing could provide better experience.

1.53 Frequently Asked Questions

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

1.53.1 FAQ

Q1: "xxx: 'yyy is not in the zzz registry'".

A1: The registry mechanism will be triggered only when the file of the module is imported. So you need to import that file somewhere.

Q2: What's the folder structure of xxx dataset?

A2: You can make sure the folder structure is correct following tutorials of *dataset preparation*.

Q3: How to use LMDB data to train the model?

A3: You can use scripts in `tools/data` to make LMDB files. More details are shown in tutorials of *dataset preparation*.

Q4: Why `MMCV==xxx is used but incompatible` is raised when import I try to import `mmgen`?

A4: This is because the version of MMCV and MMGeneration are incompatible. Compatible MMGeneration and MMCV versions are shown as below. Please choose the correct version of MMCV to avoid installation issues.

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

Q5: How can I ignore some fields in the base configs?

A5: Sometimes, you may set `_delete_=True` to ignore some of fields in base configs. You may refer to MMEngine for simple illustration.

You may have a careful look at this tutorial for better understanding of this feature.

Q6:: How can I use intermediate variables in configs?

A6: Some intermediate variables are used in the config files, like `train_pipeline/test_pipeline` in datasets. It's worth noting that when modifying intermediate variables in the children configs, users need to pass the intermediate variables into corresponding fields again.

1.54 English

1.55

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

D

E

F

G

I

L

M

N

O

P

R

S

T

U

V

W